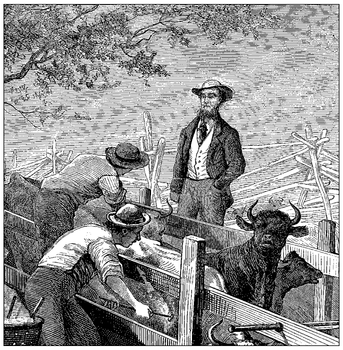

CHAPTER THREE

CHAR DRIVERS

The goal of this chapter is to write a complete char device driver. We'll develop a character driver because this class is suitable for most simple hardware devices. Char drivers are also easier to understand than, for example, block drivers or network drivers. Our ultimate aim is to write a *modularized* char driver, but we won't talk about modularization issues in this chapter.

Throughout the chapter, we'll present code fragments extracted from a real device driver: *scull*, short for Simple Character Utility for Loading Localities. *scull* is a char driver that acts on a memory area as though it were a device. A side effect of this behavior is that, as far as *scull* is concerned, the word *device* can be used interchangeably with "the memory area used by *scull*."

The advantage of *scull* is that it isn't hardware dependent, since every computer has memory. *scull* just acts on some memory, allocated using *kmalloc*. Anyone can compile and run *scull*, and *scull* is portable across the computer architectures on which Linux runs. On the other hand, the device doesn't do anything "useful" other than demonstrating the interface between the kernel and char drivers and allowing the user to run some tests.

The Design of scull

The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs. Since our "device" is part of the computer's memory, we're free to do what we want with it. It can be a sequential or random-access device, one device or many, and so on.

To make *scull* be useful as a template for writing real drivers for real devices, we'll show you how to implement several device abstractions on top of the computer memory, each with a different personality.

The *scull* source implements the following devices. Each kind of device implemented by the module is referred to as a *type*:

scull0 to *scull3*

Four devices each consisting of a memory area that is both global and persistent. Global means that if the device is opened multiple times, the data contained within the device is shared by all the file descriptors that opened it. Persistent means that if the device is closed and reopened, data isn't lost. This device can be fun to work with, because it can be accessed and tested using conventional commands such as *cp*, *cat*, and shell I/O redirection; we'll examine its internals in this chapter.

scullpipe0 to *scullpipe3*

Four FIFO (first-in-first-out) devices, which act like pipes. One process reads what another process writes. If multiple processes read the same device, they contend for data. The internals of *scullpipe* will show how blocking and nonblocking *read* and *write* can be implemented without having to resort to interrupts. Although real drivers synchronize with their devices using hardware interrupts, the topic of blocking and nonblocking operations is an important one and is separate from interrupt handling (covered in Chapter 9).

scullsingle
scullpriv
sculluid
scullwuid

These devices are similar to *scull0*, but with some limitations on when an *open* is permitted. The first (*scullsingle*) allows only one process at a time to use the driver, whereas *scullpriv* is private to each virtual console (or X terminal session) because processes on each console/terminal will get a different memory area from processes on other consoles. *sculluid* and *scullwuid* can be opened multiple times, but only by one user at a time; the former returns an error of "Device Busy" if another user is locking the device, whereas the latter implements blocking *open*. These variations of *scull* add more "policy" than "mechanism;" this kind of behavior is interesting to look at anyway, because some devices require types of management like the ones shown in these *scull* variations as part of their mechanism.

Each of the *scull* devices demonstrates different features of a driver and presents different difficulties. This chapter covers the internals of *scull0* to *skull3*; the more advanced devices are covered in Chapter 5: *scullpipe* is described in "A Sample Implementation: scullpipe" and the others in "Access Control on a Device File."

Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory. Special files for char drivers are identified by a "c" in the first column of the output of *ls –l*. Block devices appear in */dev* as well, but they are identified by a "b." The focus of this chapter is on char devices, but much of the following information applies to block devices as well.

If you issue the *ls –l* command, you'll see two numbers (separated by a comma) in the device file entries before the date of last modification, where the file length normally appears. These numbers are the major device number and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
crw-rw-rw- 1 root   root    1, 3   Feb 23 1999  null
crw------- 1 root   root   10, 1   Feb 23 1999  psaux
crw------- 1 rubini tty     4, 1   Aug 16 22:22 tty1
crw-rw-rw- 1 root   dialout 4, 64  Jun 30 11:19 ttyS0
crw-rw-rw- 1 root   dialout 4, 65  Aug 16 00:00 ttyS1
crw------- 1 root   sys     7, 1   Feb 23 1999  vcs1
crw------- 1 root   sys     7, 129 Feb 23 1999  vcsa1
crw-rw-rw- 1 root   root    1, 5   Feb 23 1999  zero
```

The major number identifies the driver associated with the device. For example, */dev/null* and */dev/zero* are both managed by driver 1, whereas virtual consoles and serial terminals are managed by driver 4; similarly, both *vcs1* and *vcsa1* devices are managed by driver 7. The kernel uses the major number at *open* time to dispatch execution to the appropriate driver.

The minor number is used only by the driver specified by the major number; other parts of the kernel don't use it, and merely pass it along to the driver. It is common for a driver to control several devices (as shown in the listing); the minor number provides a way for the driver to differentiate among them.

Version 2.4 of the kernel, though, introduced a new (optional) feature, the device file system or *devfs*. If this file system is used, management of device files is simplified and quite different; on the other hand, the new filesystem brings several user-visible incompatibilities, and as we are writing it has not yet been chosen as a default feature by system distributors. The previous description and the following instructions about adding a new driver and special file assume that *devfs* is not present. The gap is filled later in this chapter, in "The Device Filesystem."

When *devfs* is not being used, adding a new driver to the system means assigning a major number to it. The assignment should be made at driver (module) initialization by calling the following function, defined in `<linux/fs.h>`:

```
int register_chrdev(unsigned int major, const char *name,
     struct file_operations *fops);
```

The return value indicates success or failure of the operation. A negative return code signals an error; a 0 or positive return code reports successful completion. The `major` argument is the major number being requested, `name` is the name of your device, which will appear in */proc/devices*, and `fops` is the pointer to an array of function pointers, used to invoke your driver's entry points, as explained in "File Operations," later in this chapter.

The major number is a small integer that serves as the index into a static array of char drivers; "Dynamic Allocation of Major Numbers" later in this chapter explains how to select a major number. The 2.0 kernel supported 128 devices; 2.2 and 2.4 increased that number to 256 (while reserving the values 0 and 255 for future uses). Minor numbers, too, are eight-bit quantities; they aren't passed to *register_chrdev* because, as stated, they are only used by the driver itself. There is tremendous pressure from the developer community to increase the number of possible devices supported by the kernel; increasing device numbers to at least 16 bits is a stated goal for the 2.5 development series.

Once the driver has been registered in the kernel table, its operations are associated with the given major number. Whenever an operation is performed on a character device file associated with that major number, the kernel finds and invokes the proper function from the `file_operations` structure. For this reason, the pointer passed to *register_chrdev* should point to a global structure within the driver, not to one local to the module's initialization function.

The next question is how to give programs a name by which they can request your driver. A name must be inserted into the */dev* directory and associated with your driver's major and minor numbers.

The command to create a device node on a filesystem is *mknod*; superuser privileges are required for this operation. The command takes three arguments in addition to the name of the file being created. For example, the command

```
mknod /dev/scull0 c 254 0
```

creates a char device (`c`) whose major number is 254 and whose minor number is 0. Minor numbers should be in the range 0 to 255 because, for historical reasons, they are sometimes stored in a single byte. There are sound reasons to extend the range of available minor numbers, but for the time being, the eight-bit limit is still in force.

Please note that once created by *mknod*, the special device file remains unless it is explicitly deleted, like any information stored on disk. You may want to remove the device created in this example by issuing *rm /dev/scull0*.

Dynamic Allocation of Major Numbers

Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel source tree. Because many numbers are already assigned, choosing a unique number for a new driver can be difficult—there are far more custom drivers than available major numbers. You could use one of the major numbers reserved for "experimental or local use,"* but if you experiment with several "local" drivers or you publish your driver for third parties to use, you'll again experience the problem of choosing a suitable number.

Fortunately (or rather, thanks to someone's ingenuity), you can request dynamic assignment of a major number. If the argument `major` is set to 0 when you call *register_chrdev*, the function selects a free number and returns it. The major number returned is always positive, while negative return values are error codes. Please note the behavior is slightly different in the two cases: the function returns the allocated major number if the caller requests a dynamic number, but returns 0 (not the major number) when successfully registering a predefined major number.

For private drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. If, on the other hand, your driver is meant to be useful to the community at large and be included into the official kernel tree, you'll need to apply to be assigned a major number for exclusive use.

The disadvantage of dynamic assignment is that you can't create the device nodes in advance because the major number assigned to your module can't be guaranteed to always be the same. This means that you won't be able to use loading-on-demand of your driver, an advanced feature introduced in Chapter 11. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from */proc/devices*.

To load a driver using a dynamic major number, therefore, the invocation of *insmod* can be replaced by a simple script that after calling *insmod* reads */proc/devices* in order to create the special file(s).

A typical */proc/devices* file looks like the following:

```
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 ttyS
 6 lp
 7 vcs
 10 misc
 13 input
 14 sound
 21 sg
180 usb
```

---

[*] Major numbers in the ranges 60 to 63, 120 to 127, and 240 to 254 are reserved for local and experimental use: no real device will be assigned such major numbers.

```
Block devices:
 2 fd
 8 sd
 11 sr
 65 sd
 66 sd
```

The script to load a module that has been assigned a dynamic number can thus be written using a tool such as *awk* to retrieve information from */proc/devices* in order to create the files in */dev*.

The following script, *scull_load*, is part of the *scull* distribution. The user of a driver that is distributed in the form of a module can invoke such a script from the system's *rc.local* file or call it manually whenever the module is needed.

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# invoke insmod with all arguments we were passed
# and use a pathname, as newer modutils don't look in . by default
/sbin/insmod -f ./$module.o $* || exit 1

# remove stale nodes
rm -f /dev/${device}[0-3]

major=`awk "\\$2==\"$module\" {print \\$1}" /proc/devices`

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# give appropriate group/permissions, and change the group.
# Not all distributions have staff; some have "wheel" instead.
group="staff"
grep '^staff:' /etc/group > /dev/null || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

The script can be adapted for another driver by redefining the variables and adjusting the *mknod* lines. The script just shown creates four devices because four is the default in the *scull* sources.

The last few lines of the script may seem obscure: why change the group and mode of a device? The reason is that the script must be run by the superuser, so newly created special files are owned by root. The permission bits default so that only root has write access, while anyone can get read access. Normally, a device node requires a different access policy, so in some way or another access rights must be changed. The default in our script is to give access to a group of users, but your needs may vary. Later, in the section "Access Control on a Device File" in Chapter 5, the code for *sculluid* will demonstrate how the driver can enforce its own kind of authorization for device access. A *scull_unload* script is then available to clean up the */dev* directory and remove the module.

As an alternative to using a pair of scripts for loading and unloading, you could write an init script, ready to be placed in the directory your distribution uses for these scripts.* As part of the *scull* source, we offer a fairly complete and configurable example of an init script, called *scull.init*; it accepts the conventional arguments—either "start" or "stop" or "restart"—and performs the role of both *scull_load* and *scull_unload*.

If repeatedly creating and destroying */dev* nodes sounds like overkill, there is a useful workaround. If you are only loading and unloading a single driver, you can just use *rmmod* and *insmod* after the first time you create the special files with your script: dynamic numbers are not randomized, and you can count on the same number to be chosen if you don't mess with other (dynamic) modules. Avoiding lengthy scripts is useful during development. But this trick, clearly, doesn't scale to more than one driver at a time.

The best way to assign major numbers, in our opinion, is by defaulting to dynamic allocation while leaving yourself the option of specifying the major number at load time, or even at compile time. The code we suggest using is similar to the code introduced for autodetection of port numbers. The *scull* implementation uses a global variable, `scull_major`, to hold the chosen number. The variable is initialized to `SCULL_MAJOR`, defined in *scull.h*. The default value of `SCULL_MAJOR` in the distributed source is 0, which means "use dynamic assignment." The user can accept the default or choose a particular major number, either by modifying the macro before compiling or by specifying a value for `scull_major` on the *insmod* command line. Finally, by using the *scull_load* script, the user can pass arguments to *insmod* on *scull_load*'s command line.†

Here's the code we use in *scull*'s source to get a major number:

```
result = register_chrdev(scull_major, "scull", &scull_fops);
if (result < 0) {
 printk(KERN_WARNING "scull: can’t get major %d\n",scull_major);
 return result;
}
if (scull_major == 0) scull_major = result; /* dynamic */
```

---

* Distributions vary widely on the location of init scripts; the most common directories used are */etc/init.d*, */etc/rc.d/init.d*, and */sbin/init.d*. In addition, if your script is to be run at boot time, you will need to make a link to it from the appropriate run-level directory (i.e., *. . . /rc3.d*).

† The init script *scull.init* doesn't accept driver options on the command line, but it supports a configuration file because it's designed for automatic use at boot and shutdown time.

Removing a Driver from the System

When a module is unloaded from the system, the major number must be released. This is accomplished with the following function, which you call from the module's cleanup function:

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

The arguments are the major number being released and the name of the associated device. The kernel compares the name to the registered name for that number, if any: if they differ, `-EINVAL` is returned. The kernel also returns `-EINVAL` if the major number is out of the allowed range.

Failing to unregister the resource in the cleanup function has unpleasant effects. */proc/devices* will generate a fault the next time you try to read it, because one of the `name` strings still points to the module's memory, which is no longer mapped. This kind of fault is called an *oops* because that's the message the kernel prints when it tries to access invalid addresses.*

When you unload the driver without unregistering the major number, recovery will be difficult because the *strcmp* function in *unregister_chrdev* must dereference a pointer (`name`) to the original module. If you ever fail to unregister a major number, you must reload both the same module and another one built on purpose to unregister the major. The faulty module will, with luck, get the same address, and the `name` string will be in the same place, if you didn't change the code. The safer alternative, of course, is to reboot the system.

In addition to unloading the module, you'll often need to remove the device files for the removed driver. The task can be accomplished by a script that pairs to the one used at load time. The script *scull_unload* does the job for our sample device; as an alternative, you can invoke *scull.init stop*.

If dynamic device files are not removed from */dev*, there's a possibility of unexpected errors: a spare */dev/framegrabber* on a developer's computer might refer to a fire-alarm device one month later if both drivers used a dynamic major number. "No such file or directory" is a friendlier response to opening */dev/framegrabber* than the new driver would produce.

dev_t and kdev_t

So far we've talked about the major number. Now it's time to discuss the minor number and how the driver uses it to differentiate among devices.

Every time the kernel calls a device driver, it tells the driver which device is being acted upon. The major and minor numbers are paired in a single data type that the driver uses to identify a particular device. The combined device number (the major

---

* The word *oops* is used as both a noun and a verb by Linux enthusiasts.

and minor numbers concatenated together) resides in the field `i_rdev` of the `inode` structure, which we introduce later. Some driver functions receive a pointer to `struct inode` as the first argument. So if you call the pointer `inode` (as most driver writers do), the function can extract the device number by looking at `inode->i_rdev`.

Historically, Unix declared `dev_t` (device type) to hold the device numbers. It used to be a 16-bit integer value defined in `<sys/types.h>`. Nowadays, more than 256 minor numbers are needed at times, but changing `dev_t` is difficult because there are applications that "know" the internals of `dev_t` and would break if the structure were to change. Thus, while much of the groundwork has been laid for larger device numbers, they are still treated as 16-bit integers for now.

Within the Linux kernel, however, a different type, `kdev_t`, is used. This data type is designed to be a black box for every kernel function. User programs do not know about `kdev_t` at all, and kernel functions are unaware of what is inside a `kdev_t`. If `kdev_t` remains hidden, it can change from one kernel version to the next as needed, without requiring changes to everyone's device drivers.

The information about `kdev_t` is confined in `<linux/kdev_t.h>`, which is mostly comments. The header makes instructive reading if you're interested in the reasoning behind the code. There's no need to include the header explicitly in the drivers, however, because `<linux/fs.h>` does it for you.

The following macros and functions are the operations you can perform on `kdev_t`:

`MAJOR(kdev_t dev);`
  Extract the major number from a `kdev_t` structure.

`MINOR(kdev_t dev);`
  Extract the minor number.

`MKDEV(int ma, int mi);`
  Create a `kdev_t` built from major and minor numbers.

`kdev_t_to_nr(kdev_t dev);`
  Convert a `kdev_t` type to a number (a `dev_t`).

`to_kdev_t(int dev);`
  Convert a number to `kdev_t`. Note that `dev_t` is not defined in kernel mode, and therefore `int` is used.

As long as your code uses these operations to manipulate device numbers, it should continue to work even as the internal data structures change.

File Operations

In the next few sections, we'll look at the various operations a driver can perform on the devices it manages. An open device is identified internally by a `file` structure, and the kernel uses the `file_operations` structure to access the driver's functions. The structure, defined in `<linux/fs.h>`, is an array of function pointers. Each file is associated with its own set of functions (by including a field called `f_op` that points to a `file_operations` structure). The operations are mostly in charge of implementing the system calls and are thus named *open*, *read*, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on itself. This is the first sign of object-oriented programming we see in the Linux kernel, and we'll see more in later chapters.

Conventionally, a `file_operations` structure or a pointer to one is called `fops` (or some variation thereof); we've already seen one such pointer as an argument to the *register_chrdev* call. Each field in the structure must point to the function in the driver that implements a specific operation, or be left `NULL` for unsupported operations. The exact behavior of the kernel when a `NULL` pointer is specified is different for each function, as the list later in this section shows.

The `file_operations` structure has been slowly getting bigger as new functionality is added to the kernel. The addition of new operations can, of course, create portability problems for device drivers. Instantiations of the structure in each driver used to be declared using standard C syntax, and new operations were normally added to the end of the structure; a simple recompilation of the drivers would place a `NULL` value for that operation, thus selecting the default behavior, usually what you wanted.

Since then, kernel developers have switched to a "tagged" initialization format that allows initialization of structure fields by name, thus circumventing most problems with changed data structures. The tagged initialization, however, is not standard C but a (useful) extension specific to the GNU compiler. We will look at an example of tagged structure initialization shortly.

The following list introduces all the operations that an application can invoke on a device. We've tried to keep the list brief so it can be used as a reference, merely summarizing each operation and the default kernel behavior when a `NULL` pointer is used. You can skip over this list on your first reading and return to it later.

The rest of the chapter, after describing another important data structure (the `file`, which actually includes a pointer to its own `file_operations`), explains the role of the most important operations and offers hints, caveats, and real code examples. We defer discussion of the more complex operations to later chapters because we aren't ready to dig into topics like memory management, blocking operations, and asynchronous notification quite yet.

The following list shows what operations appear in `struct file_operations` for the 2.4 series of kernels, in the order in which they appear. Although there are minor differences between 2.4 and earlier kernels, they will be dealt with later in this chapter, so we are just sticking to 2.4 for a while. The return value of each operation is 0 for success or a negative error code to signal an error, unless otherwise noted.

`loff_t (*llseek) (struct file *, loff_t, int);`

The *llseek* method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The `loff_t` is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If the function is not specified for the driver, a seek relative to end-of-file fails, while other seeks succeed by modifying the position counter in the `file` structure (described in "The file Structure" later in this chapter).

`ssize_t (*read) (struct file *, char *, size_t, loff_t *);`

Used to retrieve data from the device. A null pointer in this position causes the *read* system call to fail with `-EINVAL` ("Invalid argument"). A non-negative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

`ssize_t (*write) (struct file *, const char *, size_t, loff_t *);`

Sends data to the device. If missing, `-EINVAL` is returned to the program calling the *write* system call. The return value, if non-negative, represents the number of bytes successfully written.

`int (*readdir) (struct file *, void *, filldir_t);`

This field should be `NULL` for device files; it is used for reading directories, and is only useful to filesystems.

`unsigned int (*poll) (struct file *, struct poll_table_struct *);`

The *poll* method is the back end of two system calls, *poll* and *select*, both used to inquire if a device is readable or writable or in some special state. Either system call can block until a device becomes readable or writable. If a driver doesn't define its *poll* method, the device is assumed to be both readable and writable, and in no special state. The return value is a bit mask describing the status of the device.

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

The *ioctl* system call offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few *ioctl* commands are recognized by the kernel without referring to the `fops` table. If the device doesn't offer an *ioctl* entry point, the system call returns an error for any request that isn't predefined (`-ENOTTY`, "No such ioctl for device"). If the device method returns a non-negative value, the same value is passed back to the calling program to indicate successful completion.

`int (*mmap) (struct file *, struct vm_area_struct *);`

mmap is used to request a mapping of device memory to a process's address space. If the device doesn't implement this method, the *mmap* system call returns `-ENODEV`.

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

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is `NULL`, opening the device always succeeds, but your driver isn't notified.

`int (*flush) (struct file *);`

The *flush* operation is invoked when a process closes its copy of a file descriptor for a device; it should execute (and wait for) any outstanding operations on the device. This must not be confused with the *fsync* operation requested by user programs. Currently, *flush* is used only in the network file system (NFS) code. If *flush* is `NULL`, it is simply not invoked.

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

This operation is invoked when the `file` structure is being released. Like *open*, *release* can be missing.[*]

`int (*fsync) (struct inode *, struct dentry *, int);`

This method is the back end of the *fsync* system call, which a user calls to flush any pending data. If not implemented in the driver, the system call returns `-EINVAL`.

`int (*fasync) (int, struct file *, int);`

This operation is used to notify the device of a change in its `FASYNC` flag. Asynchronous notification is an advanced topic and is described in Chapter 5. The field can be `NULL` if the driver doesn't support asynchronous notification.

`int (*lock) (struct file *, int, struct file_lock *);`

The *lock* method is used to implement file locking; locking is an indispensable feature for regular files, but is almost never implemented by device drivers.

```
ssize_t (*readv) (struct file *, const struct iovec *,
      unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *,
      unsigned long, loff_t *);
```

---

[*] Note that *release* isn't invoked every time a process calls *close*. Whenever a `file` structure is shared (for example, after a *fork* or a *dup*), *release* won't be invoked until all copies are closed. If you need to flush pending data when any copy is closed, you should implement the *flush* method.

These methods, added late in the 2.3 development cycle, implement scatter/gather read and write operations. Applications occasionally need to do a single read or write operation involving multiple memory areas; these system calls allow them to do so without forcing extra copy operations on the data.

`struct module *owner;`

This field isn't a method like everything else in the `file_operations` structure. Instead, it is a pointer to the module that "owns" this structure; it is used by the kernel to maintain the module's usage count.

The *scull* device driver implements only the most important device methods, and uses the tagged format to declare its `file_operations` structure:

```
struct file_operations scull_fops = {
 llseek:  scull_llseek,
 read:  scull_read,
 write:  scull_write,
 ioctl:  scull_ioctl,
 open:  scull_open,
 release: scull_release,
};
```

This declaration uses the tagged structure initialization syntax, as we described earlier. This syntax is preferred because it makes drivers more portable across changes in the definitions of the structures, and arguably makes the code more compact and readable. Tagged initialization allows the reordering of structure members; in some cases, substantial performance improvements have been realized by placing frequently accessed members in the same hardware cache line.

It is also necessary to set the `owner` field of the `file_operations` structure. In some kernel code, you will often see `owner` initialized with the rest of the structure, using the tagged syntax as follows:

```
owner: THIS_MODULE,
```

That approach works, but only on 2.4 kernels. A more portable approach is to use the `SET_MODULE_OWNER` macro, which is defined in `<linux/module.h>`. *scull* performs this initialization as follows:

```
SET_MODULE_OWNER(&scull_fops);
```

This macro works on any structure that has an `owner` field; we will encounter this field again in other contexts later in the book.

The file Structure

`struct file`, defined in `<linux/fs.h>`, is the second most important data structure used in device drivers. Note that a `file` has nothing to do with the

`FILE`s of user-space programs. A `FILE` is defined in the C library and never appears in kernel code. A `struct file`, on the other hand, is a kernel structure that never appears in user programs.

The `file` structure represents an *open file*. (It is not specific to device drivers; every open file in the system has an associated `struct file` in kernel space.) It is created by the kernel on *open* and is passed to any function that operates on the file, until the last *close*. After all instances of the file are closed, the kernel releases the data structure. An open file is different from a disk file, represented by `struct inode`.

In the kernel sources, a pointer to `struct file` is usually called either `file` or `filp` ("file pointer"). We'll consistently call the pointer `filp` to prevent ambiguities with the structure itself. Thus, `file` refers to the structure and `filp` to a pointer to the structure.

The most important fields of `struct file` are shown here. As in the previous section, the list can be skipped on a first reading. In the next section though, when we face some real C code, we'll discuss some of the fields, so they are here for you to refer to.

`mode_t f_mode;`

The file mode identifies the file as either readable or writable (or both), by means of the bits `FMODE_READ` and `FMODE_WRITE`. You might want to check this field for read/write permission in your *ioctl* function, but you don't need to check permissions for *read* and *write* because the kernel checks before invoking your method. An attempt to write without permission, for example, is rejected without the driver even knowing about it.

`loff_t f_pos;`

The current reading or writing position. `loff_t` is a 64-bit value (`long long` in *gcc* terminology). The driver can read this value if it needs to know the current position in the file, but should never change it (*read* and *write* should update a position using the pointer they receive as the last argument instead of acting on `filp->f_pos` directly).

`unsigned int f_flags;`

These are the file flags, such as `O_RDONLY`, `O_NONBLOCK`, and `O_SYNC`. A driver needs to check the flag for nonblocking operation, while the other flags are seldom used. In particular, read/write permission should be checked using `f_mode` instead of `f_flags`. All the flags are defined in the header `<linux/fcntl.h>`.

`struct file_operations *f_op;`

The operations associated with the file. The kernel assigns the pointer as part of its implementation of *open*, and then reads it when it needs to dispatch any operations. The value in `filp->f_op` is never saved for later reference; this means that you can change the file operations associated with your file whenever you want, and the new methods will be effective immediately after you return to the caller. For example, the code for *open* associated with major number 1 (*/dev/null*, */dev/zero*, and so on) substitutes the operations in `filp->f_op` depending on the minor number being opened. This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of “method overriding” in object-oriented programming.

`void *private_data;`

The *open* system call sets this pointer to `NULL` before calling the *open* method for the driver. The driver is free to make its own use of the field or to ignore it. The driver can use the field to point to allocated data, but then must free memory in the *release* method before the `file` structure is destroyed by the kernel. `private_data` is a useful resource for preserving state information across system calls and is used by most of our sample modules.

`struct dentry *f_dentry;`

The directory entry (*dentry*) structure associated with the file. Dentries are an optimization introduced in the 2.1 development series. Device driver writers normally need not concern themselves with dentry structures, other than to access the `inode` structure as `filp->f_dentry->d_inode`.

The real structure has a few more fields, but they aren’t useful to device drivers. We can safely ignore those fields because drivers never fill `file` structures; they only access structures created elsewhere.

open and release

Now that we’ve taken a quick look at the fields, we’ll start using them in real *scull* functions.

The open Method

The *open* method is provided for a driver to do any initialization in preparation for later operations. In addition, *open* usually increments the usage count for the device so that the module won’t be unloaded before the file is closed. The count, described in “The Usage Count” in Chapter 2, is then decremented by the *release* method.

In most drivers, *open* should perform the following tasks:

- Increment the usage count
- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device, if it is being opened for the first time
- Identify the minor number and update the `f_op` pointer, if necessary
- Allocate and fill any data structure to be put in `filp->private_data`

In *scull*, most of the preceding tasks depend on the minor number of the device being opened. Therefore, the first thing to do is identify which device is involved. We can do that by looking at `inode->i_rdev`.

We've already talked about how the kernel doesn't use the minor number of the device, so the driver is free to use it at will. In practice, different minor numbers are used to access different devices or to open the same device in a different way. For example, */dev/st0* (minor number 0) and */dev/st1* (minor 1) refer to different SCSI tape drives, whereas */dev/nst0* (minor 128) is the same physical device as */dev/st0*, but it acts differently (it doesn't rewind the tape when it is closed). All of the tape device files have different minor numbers, so that the driver can tell them apart.

A driver never actually knows the name of the device being opened, just the device number—and users can play on this indifference to names by aliasing new names to a single device for their own convenience. If you create two special files with the same major/minor pair, the devices are one and the same, and there is no way to differentiate between them. The same effect can be obtained using a symbolic or hard link, and the preferred way to implement aliasing is creating a symbolic link.

The *scull* driver uses the minor number like this: the most significant nibble (upper four bits) identifies the type (personality) of the device, and the least significant nibble (lower four bits) lets you distinguish between individual devices if the type supports more than one device instance. Thus, *scull0* is different from *scullpipe0* in the top nibble, while *scull0* and *scull1* differ in the bottom nibble.* Two macros (`TYPE` and `NUM`) are defined in the source to extract the bits from a device number, as shown here:

```
#define TYPE(dev) (MINOR(dev) >> 4) /* high nibble */
#define NUM(dev) (MINOR(dev) & 0xf) /* low nibble */
```

---

* Bit splitting is a typical way to use minor numbers. The IDE driver, for example, uses the top two bits for the disk number, and the bottom six bits for the partition number.

For each device type, *scull* defines a specific `file_operations` structure, which is placed in `filp->f_op` at open time. The following code shows how multiple `fops` are implemented:

```
struct file_operations *scull_fop_array[]={
 &scull_fops,  /* type 0 */
 &scull_priv_fops, /* type 1 */
 &scull_pipe_fops, /* type 2 */
 &scull_sngl_fops, /* type 3 */
 &scull_user_fops, /* type 4 */
 &scull_wusr_fops /* type 5 */
};
#define SCULL_MAX_TYPE 5

/* In scull_open, the fop_array is used according to TYPE(dev) */
 int type = TYPE(inode->i_rdev);

  if (type > SCULL_MAX_TYPE) return -ENODEV;
  filp->f_op = scull_fop_array[type];
```

The kernel invokes *open* according to the major number; *scull* uses the minor number in the macros just shown. `TYPE` is used to index into `scull_fop_array` in order to extract the right set of methods for the device type being opened.

In *scull*, `filp->f_op` is assigned to the correct `file_operations` structure as determined by the device type, found in the minor number. The *open* method declared in the new `fops` is then invoked. Usually, a driver doesn't invoke its own `fops`, because they are used by the kernel to dispatch the right driver method. But when your *open* method has to deal with different device types, you might want to call `fops->open` after modifying the `fops` pointer according to the minor number being opened.

The actual code for *scull_open* follows. It uses the `TYPE` and `NUM` macros defined in the previous code snapshot to split the minor number:

```
int scull_open(struct inode *inode, struct file *filp)
{
 Scull_Dev *dev; /* device information */
 int num = NUM(inode->i_rdev);
 int type = TYPE(inode->i_rdev);

 /*
  * If private data is not valid, we are not using devfs
  * so use the type (from minor nr.) to select a new f_op
  */
 if (!filp->private_data && type) {
  if (type > SCULL_MAX_TYPE) return -ENODEV;
  filp->f_op = scull_fop_array[type];
  return filp->f_op->open(inode, filp); /* dispatch to specific open */
 }
```

```
    /* type 0, check the device number (unless private_data valid) */
    dev = (Scull_Dev *)filp->private_data;
    if (!dev) {
     if (num >= scull_nr_devs) return -ENODEV;
     dev = &scull_devices[num];
     filp->private_data = dev; /* for other methods */
    }

    MOD_INC_USE_COUNT; /* Before we maybe sleep */
    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
     if (down_interruptible(&dev->sem)) {
      MOD_DEC_USE_COUNT;
      return -ERESTARTSYS;
     }
     scull_trim(dev); /* ignore errors */
     up(&dev->sem);
    }

    return 0;   /* success */
   }
```

A few explanations are due here. The data structure used to hold the region of memory is `Scull_Dev`, which will be introduced shortly. The global variables `scull_nr_devs` and `scull_devices[]` (all lowercase) are the number of available devices and the actual array of pointers to `Scull_Dev`.

The calls to *down_interruptible* and *up* can be ignored for now; we will get to them shortly.

The code looks pretty sparse because it doesn't do any particular device handling when *open* is called. It doesn't need to, because the *scull0-3* device is global and persistent by design. Specifically, there's no action like "initializing the device on first open" because we don't keep an open count for *scull*s, just the module usage count.

Given that the kernel can maintain the usage count of the module via the `owner` field in the `file_operations` structure, you may be wondering why we increment that count manually here. The answer is that older kernels required modules to do all of the work of maintaining their usage count—the `owner` mechanism did not exist. To be portable to older kernels, *scull* increments its own usage count. This behavior will cause the usage count to be too high on 2.4 systems, but that is not a problem because it will still drop to zero when the module is not being used.

The only real operation performed on the device is truncating it to a length of zero when the device is opened for writing. This is performed because, by design, overwriting a p*scull* device with a shorter file results in a shorter device data area. This is similar to the way opening a regular file for writing truncates it to zero length. The operation does nothing if the device is opened for reading.

We'll see later how a real initialization works when we look at the code for the other *scull* personalities.

The release Method

The role of the *release* method is the reverse of *open*. Sometimes you'll find that the method implementation is called `device_close` instead of `device_release`. Either way, the device method should perform the following tasks:

- Deallocate anything that *open* allocated in `filp->private_data`
- Shut down the device on last close
- Decrement the usage count

The basic form of *scull* has no hardware to shut down, so the code required is minimal:[*]

```
int scull_release(struct inode *inode, struct file *filp)
{
 MOD_DEC_USE_COUNT;
 return 0;
}
```

It is important to decrement the usage count if you incremented it at *open* time, because the kernel will never be able to unload the module if the counter doesn't drop to zero.

How can the counter remain consistent if sometimes a file is closed without having been opened? After all, the *dup* and *fork* system calls will create copies of open files without calling *open*; each of those copies is then closed at program termination. For example, most programs don't open their *stdin* file (or device), but all of them end up closing it.

The answer is simple: not every *close* system call causes the *release* method to be invoked. Only the ones that actually release the device data structure invoke the method—hence its name. The kernel keeps a counter of how many times a `file` structure is being used. Neither *fork* nor *dup* creates a new `file` structure (only *open* does that); they just increment the counter in the existing structure.

The *close* system call executes the *release* method only when the counter for the `file` structure drops to zero, which happens when the structure is destroyed. This relationship between the *release* method and the *close* system call guarantees that the usage count for modules is always consistent.

---

[*] The other flavors of the device are closed by different functions, because *scull_open* substituted a different `filp->f_op` for each device. We'll see those later.

Note that the *flush* method *is* called every time an application calls *close*. However, very few drivers implement *flush*, because usually there's nothing to perform at close time unless *release* is involved.

As you may imagine, the previous discussion applies even when the application terminates without explicitly closing its open files: the kernel automatically closes any file at process exit time by internally using the *close* system call.

scull's Memory Usage

Before introducing the *read* and *write* operations, we'd better look at how and why *scull* performs memory allocation. "How" is needed to thoroughly understand the code, and "why" demonstrates the kind of choices a driver writer needs to make, although *scull* is definitely not typical as a device.

This section deals only with the memory allocation policy in *scull* and doesn't show the hardware management skills you'll need to write real drivers. Those skills are introduced in Chapter 8, and in Chapter 9. Therefore, you can skip this section if you're not interested in understanding the inner workings of the memory-oriented *scull* driver.

The region of memory used by *scull*, also called a *device* here, is variable in length. The more you write, the more it grows; trimming is performed by overwriting the device with a shorter file.

The implementation chosen for *scull* is not a smart one. The source code for a smart implementation would be more difficult to read, and the aim of this section is to show *read* and *write*, not memory management. That's why the code just uses *kmalloc* and *kfree* without resorting to allocation of whole pages, although that would be more efficient.

On the flip side, we didn't want to limit the size of the "device" area, for both a philosophical reason and a practical one. Philosophically, it's always a bad idea to put arbitrary limits on data items being managed. Practically, *scull* can be used to temporarily eat up your system's memory in order to run tests under low-memory conditions. Running such tests might help you understand the system's internals. You can use the command *cp /dev/zero /dev/scull0* to eat all the real RAM with *scull*, and you can use the *dd* utility to choose how much data is copied to the *scull* device.

In *scull*, each device is a linked list of pointers, each of which points to a `Scull_Dev` structure. Each such structure can refer, by default, to at most four million bytes, through an array of intermediate pointers. The released source uses an array of 1000 pointers to areas of 4000 bytes. We call each memory area a *quantum* and the array (or its length) a *quantum set*. A *scull* device and its memory areas are shown in Figure 3-1.

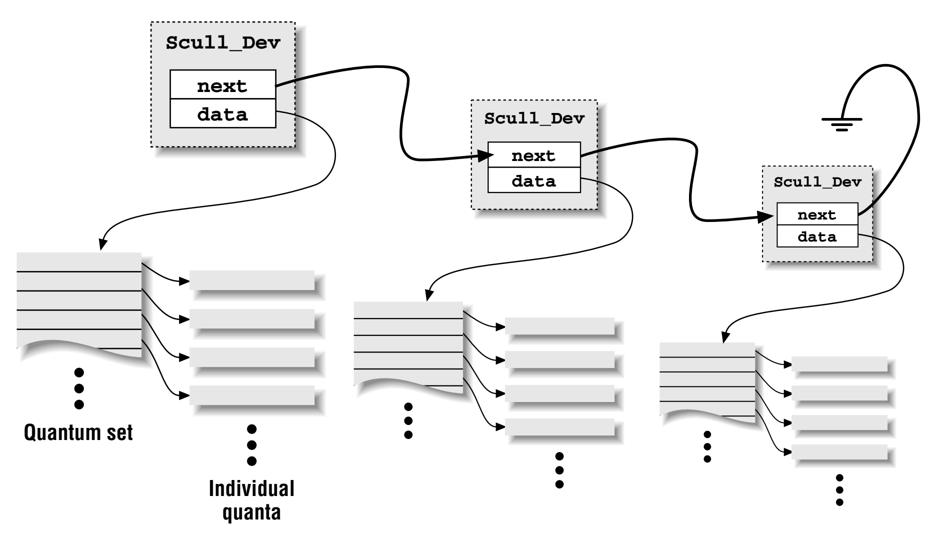

Figure 3-1. The layout of a scull device

The chosen numbers are such that writing a single byte in *scull* consumes eight or twelve thousand bytes of memory: four thousand for the quantum and four or eight thousand for the quantum set (according to whether a pointer is represented in 32 bits or 64 bits on the target platform). If, instead, you write a huge amount of data, the overhead of the linked list is not too bad. There is only one list element for every four megabytes of data, and the maximum size of the device is limited by the computer's memory size.

Choosing the appropriate values for the quantum and the quantum set is a question of policy, rather than mechanism, and the optimal sizes depend on how the device is used. Thus, the *scull* driver should not force the use of any particular values for the quantum and quantum set sizes. In *scull*, the user can change the values in charge in several ways: by changing the macros `SCULL_QUANTUM` and `SCULL_QSET` in *scull.h* at compile time, by setting the integer values `scull_quantum` and `scull_qset` at module load time, or by changing both the current and default values using *ioctl* at runtime.

Using a macro and an integer value to allow both compile-time and load-time configuration is reminiscent of how the major number is selected. We use this technique for whatever value in the driver is arbitrary, or related to policy.

The only question left is how the default numbers have been chosen. In this particular case, the problem is finding the best balance between the waste of memory resulting from half-filled quanta and quantum sets and the overhead of allocation, deallocation, and pointer chaining that occurs if quanta and sets are small.

Additionally, the internal design of *kmalloc* should be taken into account. We won't touch the point now, though; the innards of *kmalloc* are explored in "The Real Story of kmalloc" in Chapter 7.

The choice of default numbers comes from the assumption that massive amounts of data are likely to be written to *scull* while testing it, although normal use of the device will most likely transfer just a few kilobytes of data.

The data structure used to hold device information is as follows:

```
typedef struct Scull_Dev {
 void **data;
 struct Scull_Dev *next; /* next list item */
 int quantum;    /* the current quantum size */
 int qset;     /* the current array size */
 unsigned long size;
 devfs_handle_t handle; /* only used if devfs is there */
 unsigned int access_key; /* used by sculluid and scullpriv */
 struct semaphore sem;  /* mutual exclusion semaphore  */
} Scull_Dev;
```

The next code fragment shows in practice how `Scull_Dev` is used to hold data. The function *scull_trim* is in charge of freeing the whole data area and is invoked by *scull_open* when the file is opened for writing. It simply walks through the list and frees any quantum and quantum set it finds.

```
int scull_trim(Scull_Dev *dev)
{
 Scull_Dev *next, *dptr;
 int qset = dev->qset; /* "dev" is not null */
 int i;

 for (dptr = dev; dptr; dptr = next) { /* all the list items */
  if (dptr->data) {
   for (i = 0; i < qset; i++)
    if (dptr->data[i])
     kfree(dptr->data[i]);
   kfree(dptr->data);
   dptr->data=NULL;
  }
  next=dptr->next;
  if (dptr != dev) kfree(dptr); /* all of them but the first */
 }
 dev->size = 0;
 dev->quantum = scull_quantum;
 dev->qset = scull_qset;
 dev->next = NULL;
 return 0;
}
```

A Brief Introduction to Race Conditions

Now that you understand how *scull*'s memory management works, here is a scenario to consider. Two processes, A and B, both have the same *scull* device open for writing. Both attempt simultaneously to append data to the device. A new quantum is required for this operation to succeed, so each process allocates the required memory and stores a pointer to it in the quantum set.

The result is trouble. Because both processes see the same *scull* device, each will store its new memory in the same place in the quantum set. If A stores its pointer first, B will overwrite that pointer when it does its store. Thus the memory allocated by A, and the data written therein, will be lost.

This situation is a classic *race condition*; the results vary depending on who gets there first, and usually something undesirable happens in any case. On uniprocessor Linux systems, the *scull* code would not have this sort of problem, because processes running kernel code are not preempted. On SMP systems, however, life is more complicated. Processes A and B could easily be running on different processors and could interfere with each other in this manner.

The Linux kernel provides several mechanisms for avoiding and managing race conditions. A full description of these mechanisms will have to wait until Chapter 9, but a beginning discussion is appropriate here.

A *semaphore* is a general mechanism for controlling access to resources. In its simplest form, a semaphore may be used for *mutual exclusion*; processes using semaphores in the mutual exclusion mode are prevented from simultaneously running the same code or accessing the same data. This sort of semaphore is often called a *mutex*, from "mutual exclusion."

Semaphores in Linux are defined in `<asm/semaphore.h>`. They have a type of `struct semaphore`, and a driver should only act on them using the provided interface. In *scull*, one semaphore is allocated for each device, in the `Scull_Dev` structure. Since the devices are entirely independent of each other, there is no need to enforce mutual exclusion across multiple devices.

Semaphores must be initialized prior to use by passing a numeric argument to *sema_init*. For mutual exclusion applications (i.e., keeping multiple threads from accessing the same data simultaneously), the semaphore should be initialized to a value of 1, which means that the semaphore is available. The following code in *scull*'s module initialization function (*scull_init*) shows how the semaphores are initialized as part of setting up the devices.

```
for (i=0; i < scull_nr_devs; i++) {
 scull_devices[i].quantum = scull_quantum;
 scull_devices[i].qset = scull_qset;
 sema_init(&scull_devices[i].sem, 1);
}
```

A process wishing to enter a section of code protected by a semaphore must first ensure that no other process is already there. Whereas in classical computer science the function to obtain a semaphore is often called *P*, in Linux you'll need to call *down* or *down_interruptible*. These functions test the value of the semaphore to see if it is greater than 0; if so, they decrement the semaphore and return. If the semaphore is 0, the functions will sleep and try again after some other process, which has presumably freed the semaphore, wakes them up.

The *down_interruptible* function can be interrupted by a signal, whereas *down* will not allow signals to be delivered to the process. You almost always want to allow signals; otherwise, you risk creating unkillable processes and other undesirable behavior. A complication of allowing signals, however, is that you always have to check if the function (here *down_interruptible*) was interrupted. As usual, the function returns 0 for success and nonzero in case of failure. If the process is interrupted, it will not have acquired the semaphores; thus, you won't need to call *up*. A typical call to invoke a semaphore therefore normally looks something like this:

```
if (down_interruptible (&sem))
      return -ERESTARTSYS;
```

The `-ERESTARTSYS` return value tells the system that the operation was interrupted by a signal. The kernel function that called the device method will either retry it or return `-EINTR` to the application, according to how signal handling has been configured by the application. Of course, your code may have to perform cleanup work before returning if interrupted in this mode.

A process that obtains a semaphore must always release it afterward. Whereas computer science calls the release function *V*, Linux uses *up* instead. A simple call like

```
up (&sem);
```

will increment the value of the semaphore and wake up any processes that are waiting for the semaphore to become available.

Care must be taken with semaphores. The data protected by the semaphore must be clearly defined, and *all* code that accesses that data must obtain the semaphore first. Code that uses *down_interruptible* to obtain a semaphore must not call another function that also attempts to obtain that semaphore, or deadlock will result. If a routine in your driver fails to release a semaphore it holds (perhaps as a result of an error return), any further attempts to obtain that semaphore will stall. Mutual exclusion in general can be tricky, and benefits from a well-defined and methodical approach.

In *scull*, the per-device semaphore is used to protect access to the stored data. Any code that accesses the `data` field of the `Scull_Dev` structure must first have obtained the semaphore. To avoid deadlocks, only functions that implement device methods will try to obtain the semaphore. Internal routines, such as *scull_trim* shown earlier, assume that the semaphore has already been obtained. As long as these invariants hold, access to the `Scull_Dev` data structure is safe from race conditions.

read and write

The *read* and *write* methods perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar and it's worth introducing them at the same time:

```
ssize_t read(struct file *filp, char *buff,
    size_t count, loff_t *offp);
ssize_t write(struct file *filp, const char *buff,
    size_t count, loff_t *offp);
```

For both methods, `filp` is the file pointer and `count` is the size of the requested data transfer. The `buff` argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, `offp` is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type;" its use is discussed later.

As far as data transfer is concerned, the main issue associated with the two device methods is the need to transfer data between the kernel address space and the user address space. The operation cannot be carried out through pointers in the usual way, or through *memcpy*. User-space addresses cannot be used directly in kernel space, for a number of reasons.

One big difference between kernel-space addresses and user-space addresses is that memory in user-space can be swapped out. When the kernel accesses a user-space pointer, the associated page may not be present in memory, and a page fault is generated. The functions we introduce in this section and in "Using the ioctl Argument" in Chapter 5 use some hidden magic to deal with page faults in the proper way even when the CPU is executing in kernel space.

Also, it's interesting to note that the x86 port of Linux 2.0 used a completely different memory map for user space and kernel space. Thus, user-space pointers couldn't be dereferenced at all from kernel space.

If the target device is an expansion board instead of RAM, the same problem arises, because the driver must nonetheless copy data between user buffers and kernel space (and possibly between kernel space and I/O memory).

Cross-space copies are performed in Linux by special functions, defined in `<asm/uaccess.h>`. Such a copy is either performed by a generic (*memcpy*-like) function or by functions optimized for a specific data size (`char`, `short`, `int`, `long`); most of them are introduced in "Using the ioctl Argument" in Chapter 5.

The code for *read* and *write* in *scull* needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of every *read* and *write* implementation:

```
unsigned long copy_to_user(void *to, const void *from,
       unsigned long count);
unsigned long copy_from_user(void *to, const void *from,
       unsigned long count);
```

Although these functions behave like normal *memcpy* functions, a little extra care must be used when accessing user space from kernel code. The user pages being addressed might not be currently present in memory, and the page-fault handler can put the process to sleep while the page is being transferred into place. This happens, for example, when the page must be retrieved from swap space. The net result for the driver writer is that any function that accesses user space must be reentrant and must be able to execute concurrently with other driver functions (see also “Writing Reentrant Code” in Chapter 5). That’s why we use semaphores to control concurrent access.

The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied. In both cases, the return value is the amount of memory still to be copied. The *scull* code looks for this error return, and returns `-EFAULT` to the user if it’s not 0.

The topic of user-space access and invalid user space pointers is somewhat advanced, and is discussed in “Using the ioctl Argument” in Chapter 5. However, it’s worth suggesting that if you don’t need to check the user-space pointer you can invoke *__copy_to_user* and *__copy_from_user* instead. This is useful, for example, if you know you already checked the argument.

As far as the actual device methods are concerned, the task of the *read* method is to copy data from the device to user space (using *copy_to_user*), while the *write* method must copy data from user space to the device (using *copy_from_user*). Each *read* or *write* system call requests transfer of a specific number of bytes, but the driver is free to transfer less data—the exact rules are slightly different for reading and writing and are described later in this chapter.

Whatever the amount of data the methods transfer, they should in general update the file position at `*offp` to represent the current file position after successful completion of the system call. Most of the time the `offp` argument is just a pointer to `filp->f_pos`, but a different pointer is used in order to support the *pread* and *pwrite* system calls, which perform the equivalent of *lseek* and *read* or *write* in a single, atomic operation.

Figure 3-2 represents how a typical *read* implementation uses its arguments.

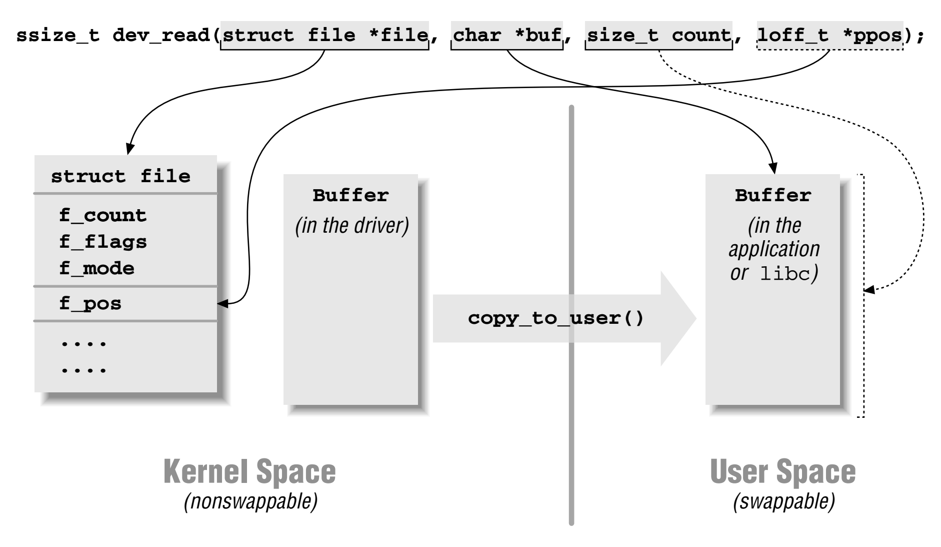

Figure 3-2. The arguments to read

Both the *read* and *write* methods return a negative value if an error occurs. A return value greater than or equal to 0 tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called.

Although kernel functions return a negative number to signal an error, and the value of the number indicates the kind of error that occurred (as introduced in Chapter 2 in "Error Handling in init_module"), programs that run in user space always see –1 as the error return value. They need to access the `errno` variable to find out what happened. The difference in behavior is dictated by the POSIX calling standard for system calls and the advantage of not dealing with `errno` in the kernel.

The read Method

The return value for *read* is interpreted by the calling application program as follows:

- If the value equals the `count` argument passed to the *read* system call, the requested number of bytes has been transferred. This is the optimal case.

- If the value is positive, but smaller than `count`, only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program will retry the read. For instance, if you read using the *fread* function, the library function reissues the system call till completion of the requested data transfer.
- If the value is 0, end-of-file was reached.
- A negative value means there was an error. The value specifies what the error was, according to `<linux/errno.h>`. These errors look like `-EINTR` (interrupted system call) or `-EFAULT` (bad address).

What is missing from the preceding list is the case of "there is no data, but it may arrive later." In this case, the *read* system call should block. We won't deal with blocking input until "Blocking I/O" in Chapter 5.

The *scull* code takes advantage of these rules. In particular, it takes advantage of the partial-read rule. Each invocation of *scull_read* deals only with a single data quantum, without implementing a loop to gather all the data; this makes the code shorter and easier to read. If the reading program really wants more data, it reiterates the call. If the standard I/O library (i.e., *fread* and friends) is used to read the device, the application won't even notice the quantization of the data transfer.

If the current read position is greater than the device size, the *read* method of *scull* returns 0 to signal that there's no data available (in other words, we're at end-of-file). This situation can happen if process A is reading the device while process B opens it for writing, thus truncating the device to a length of zero. Process A suddenly finds itself past end-of-file, and the next *read* call returns 0.

Here is the code for *read*:

```
ssize_t scull_read(struct file *filp, char *buf, size_t count,
    loff_t *f_pos)
{
 Scull_Dev *dev = filp->private_data; /* the first list item */
 Scull_Dev *dptr;
 int quantum = dev->quantum;
 int qset = dev->qset;
 int itemsize = quantum * qset; /* how many bytes in the list item */
 int item, s_pos, q_pos, rest;
 ssize_t ret = 0;

 if (down_interruptible(&dev->sem))
   return -ERESTARTSYS;
 if (*f_pos >= dev->size)
  goto out;
 if (*f_pos + count > dev->size)
  count = dev->size - *f_pos;
 /* find list item, qset index, and offset in the quantum */
 item = (long)*f_pos / itemsize;
 rest = (long)*f_pos % itemsize;
```

```
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);

    if (!dptr->data)
     goto out; /* don't fill holes */
    if (!dptr->data[s_pos])
     goto out;
    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
     count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos]+q_pos, count)) {
     ret = -EFAULT;
          goto out;
    }
    *f_pos += count;
    ret = count;

    out:
    up(&dev->sem);
    return ret;
   }
```

The write Method

write, like *read*, can transfer less data than was requested, according to the following rules for the return value:

- If the value equals `count`, the requested number of bytes has been transferred.
- If the value is positive, but smaller than `count`, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
- If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to *write*. We'll examine the exact meaning of this case in "Blocking I/O" in Chapter 5, where blocking *write* is introduced.
- A negative value means an error occurred; like for *read*, valid error values are those defined in `<linux/errno.h>`.

Unfortunately, there may be misbehaving programs that issue an error message and abort when a partial transfer is performed. This happens because some programmers are accustomed to seeing *write* calls that either fail or succeed completely, which is actually what happens most of the time and should be supported by devices as well. This limitation in the *scull* implementation could be fixed, but we didn't want to complicate the code more than necessary.

The *scull* code for *write* deals with a single quantum at a time, like the *read* method does:

```
ssize_t scull_write(struct file *filp, const char *buf, size_t count,
    loff_t *f_pos)
{
 Scull_Dev *dev = filp->private_data;
 Scull_Dev *dptr;
 int quantum = dev->quantum;
 int qset = dev->qset;
 int itemsize = quantum * qset;
 int item, s_pos, q_pos, rest;
 ssize_t ret = -ENOMEM; /* value used in "goto out" statements */

 if (down_interruptible(&dev->sem))
   return -ERESTARTSYS;

 /* find list item, qset index and offset in the quantum */
 item = (long)*f_pos / itemsize;
 rest = (long)*f_pos % itemsize;
 s_pos = rest / quantum; q_pos = rest % quantum;

 /* follow the list up to the right position */
 dptr = scull_follow(dev, item);
 if (!dptr->data) {
  dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
  if (!dptr->data)
   goto out;
  memset(dptr->data, 0, qset * sizeof(char *));
 }
 if (!dptr->data[s_pos]) {
  dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
  if (!dptr->data[s_pos])
   goto out;
 }
 /* write only up to the end of this quantum */
 if (count > quantum - q_pos)
  count = quantum - q_pos;

 if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
  ret = -EFAULT;
       goto out;
 }
 *f_pos += count;
 ret = count;

 /* update the size */
 if (dev->size < *f_pos)
  dev-> size = *f_pos;
 out:
 up(&dev->sem);
 return ret;
}
```

readv and writev

Unix systems have long supported two alternative system calls named *readv* and *writev*. These "vector" versions take an array of structures, each of which contains a pointer to a buffer and a length value. A *readv* call would then be expected to read the indicated amount into each buffer in turn. *writev*, instead, would gather together the contents of each buffer and put them out as a single write operation.

Until version 2.3.44 of the kernel, however, Linux always emulated *readv* and *writev* with multiple calls to *read* and *write*. If your driver does not supply methods to handle the vector operations, they will still be implemented that way. In many situations, however, greater efficiency is achieved by implementing *readv* and *writev* directly in the driver.

The prototypes for the vector operations are as follows:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov,
     unsigned long count, loff_t *ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov,
     unsigned long count, loff_t *ppos);
```

Here, the `filp` and `ppos` arguments are the same as for *read* and *write*. The `iovec` structure, defined in `<linux/uio.h>`, looks like this:

```
struct iovec
{
 void *iov_base;
 _&thinsp;_kernel_size_t iov_len;
};
```

Each `iovec` describes one chunk of data to be transferred; it starts at `iov_base` (in user space) and is `iov_len` bytes long. The `count` parameter to the method tells how many `iovec` structures there are. These structures are created by the application, but the kernel copies them into kernel space before calling the driver.

The simplest implementation of the vectored operations would be a simple loop that just passes the address and length out of each `iovec` to the driver's *read* or *write* function. Often, however, efficient and correct behavior requires that the driver do something smarter. For example, a *writev* on a tape drive should write the contents of all the `iovec` structures as a single record on the tape.

Many drivers, though, will gain no benefit from implementing these methods themselves. Thus, *scull* omits them. The kernel will emulate them with *read* and *write*, and the end result is the same.

Playing with the New Devices

Once you are equipped with the four methods just described, the driver can be compiled and tested; it retains any data you write to it until you overwrite it with new data. The device acts like a data buffer whose length is limited only by the amount of real RAM available. You can try using *cp*, *dd*, and input/output redirection to test the driver.

The *free* command can be used to see how the amount of free memory shrinks and expands according to how much data is written into *scull*.

To get more confident with reading and writing one quantum at a time, you can add a *printk* at an appropriate point in the driver and watch what happens while an application reads or writes large chunks of data. Alternatively, use the *strace* utility to monitor the system calls issued by a program, together with their return values. Tracing a *cp* or an *ls -l > /dev/scull0* will show quantized reads and writes. Monitoring (and debugging) techniques are presented in detail in the next chapter.

The Device Filesystem

As suggested at the beginning of the chapter, recent versions of the Linux kernel offer a special filesystem for device entry points. The filesystem has been available for a while as an unofficial patch; it was made part of the official source tree in 2.3.46. A backport to 2.2 is available as well, although not included in the official 2.2 kernels.

Although use of the special filesystem is not widespread as we write this, the new features offer a few advantages to the device driver writer. Therefore, our version of *scull* exploits *devfs* if it is being used in the target system. The module uses kernel configuration information at compile time to know whether particular features have been enabled, and in this case we depend on `CONFIG_DEVFS_FS` being defined or not.

The main advantages of *devfs* are as follows:

- Device entry points in */dev* are created at device initialization and removed at device removal.
- The device driver can specify device names, ownership, and permission bits, but user-space programs can still change ownership and permission (but not the filename).
- There is no need to allocate a major number for the device driver and deal with minor numbers.

As a result, there is no need to run a script to create device special files when a module is loaded or unloaded, because the driver is autonomous in managing its own special files.

To handle device creation and removal, the driver should call the following functions:

```
#include <linux/devfs_fs_kernel.h>

devfs_handle_t devfs_mk_dir (devfs_handle_t dir,
  const char *name, void *info);

devfs_handle_t devfs_register (devfs_handle_t dir,
  const char *name, unsigned int flags,
  unsigned int major, unsigned int minor,
  umode_t mode, void *ops, void *info);

  void devfs_unregister (devfs_handle_t de);
```

The *devfs* implementation offers several other functions for kernel code to use. They allow creation of symbolic links, access to the internal data structures to retrieve `devfs_handle_t` items from inodes, and other tasks. Those other functions are not covered here because they are not very important or easily understood. The curious reader could look at the header file for further information.

The various arguments to the register/unregister functions are as follows:

`dir`
: The parent directory where the new special file should be created. Most drivers will use `NULL` to create special files in */dev* directly. To create an owned directory, a driver should call *devfs_mk_dir*.

`name`
: The name of the device, without the leading `/dev/`. The name can include slashes if you want the device to be in a subdirectory; the subdirectory is created during the registration process. Alternatively, you can specify a valid `dir` pointer to the hosting subdirectory.

`flags`
: A bit mask of *devfs* flags. `DEVFS_FL_DEFAULT` can be a good choice, and `DEVFS_FL_AUTO_DEVNUM` is the flag you need for automatic assignment of major and minor numbers. The actual flags are described later.

`major`
`minor`
: The major and minor numbers for the device. Unused if `DEVFS_FL_AUTO_DEVNUM` is specified in the flags.

`mode`
: Access mode of the new device.

`ops`
: A pointer to the file operation structure for the device.

`info`
A default value for `filp->private_data`. The filesystem will initialize the pointer to this value when the device is opened. The `info` pointer passed to *devfs_mk_dir* is not used by *devfs* and acts as a "client data" pointer.

`de` A "*devfs* entry" obtained by a previous call to *devfs_register*.

The flags are used to select specific features to be enabled for the special file being created. Although the flags are briefly and clearly documented in `<linux/devfs_fs_kernel.h>`, it's worth introducing some of them.

`DEVFS_FL_NONE`
`DEVFS_FL_DEFAULT`
The former symbol is simply `0`, and is suggested for code readability. The latter macro is currently defined to `DEVFS_FL_NONE`, but is a good choice to be forward compatible with future implementations of the filesystem.

`DEVFS_FL_AUTO_OWNER`
The flag makes the device appear to be owned by the last uid/gid that opened it, and read/write for anybody when no process has it opened. The feature is useful for tty device files but is also interesting for device drivers to prevent concurrent access to a nonshareable device. We'll see access policy issues in Chapter 5.

`DEVFS_FL_SHOW_UNREG`
`DEVFS_FL_HIDE`
The former flag requests not to remove the device file from */dev* when it is unregistered. The latter requests never to show it in */dev*. The flags are not usually needed for normal devices.

`DEVFS_FL_AUTO_DEVNUM`
Automatically allocate a device number for this device. The number will remain associated with the device name even after the *devfs* entry is unregistered, so if the driver is reloaded before the system is shut down, it will receive the same major/minor pair.

`DEVFS_FL_NO_PERSISTENCE`
Don't keep track of this entry after it is removed. This flags saves some system memory after module removal, at the cost of losing persistence of device features across module unload/reload. Persistent features are access mode, file ownership, and major/minor numbers.

It is possible to query the flags associated with a device or to change them at runtime. The following two functions perform the tasks:

```
int devfs_get_flags (devfs_handle_t de, unsigned int *flags);
int devfs_set_flags (devfs_handle_t de, unsigned int flags);
```

Using devfs in Practice

Because *devfs* leads to serious user-space incompatibilities as far as device names are concerned, not all installed systems use it. Independently of how the new feature will be accepted by Linux users, it's unlikely you'll write *devfs*-only drivers anytime soon; thus, you'll need to add support for the "older" way of dealing with file creation and permission from user space and using major/minor numbers in kernel space.

The code needed to implement a device driver that only runs with *devfs* installed is a subset of the code you need to support both environments, so we only show the dual-mode initialization. Instead of writing a specific sample driver to try out *devfs*, we added *devfs* support to the *scull* driver. If you load *scull* to a kernel that uses *devfs*, you'll need to directly invoke *insmod* instead of running the *scull_load* script.

We chose to create a directory to host all *scull* special files because the structure of *devfs* is highly hierarchical and there's no reason not to adhere to this convention. Moreover, we can thus show how a directory is created and removed.

Within *scull_init*, the following code deals with device creation, using a field within the device structure (called `handle`) to keep track of what devices have been registered:

```
/* If we have devfs, create /dev/scull to put files in there */
scull_devfs_dir = devfs_mk_dir(NULL, "scull", NULL);
if (!scull_devfs_dir) return -EBUSY; /* problem */

for (i=0; i < scull_nr_devs; i++) {
 sprintf(devname, "%i", i);
 devfs_register(scull_devfs_dir, devname,
     DEVFS_FL_AUTO_DEVNUM,
     0, 0, S_IFCHR | S_IRUGO | S_IWUGO,
     &scull_fops,
     scull_devices+i);
}
```

The previous code is paired by the two lines that are part of the following excerpt from *scull_cleanup*:

```
if (scull_devices) {
 for (i=0; i<scull_nr_devs; i++) {
  scull_trim(scull_devices+i);
  /* the following line is only used for devfs */
  devfs_unregister(scull_devices[i].handle);
 }
 kfree(scull_devices);
}

/* once again, only for devfs */
devfs_unregister(scull_devfs_dir);
```

Part of the previous code fragments is protected by `#ifdef CONFIG_DEVFS_FS`. If the feature is not enabled in the current kernel, *scull* will revert to *register_chrdev*.

The only extra task that needs to be performed in order to support both environments is dealing with initialization of `filp->f_ops` and `filp->private_data` in the *open* device method. The former pointer is simply not modified, since the right file operations have been specified in *devfs_register*. The latter will only need to be initialized by the *open* method if it is `NULL`, since it will only be `NULL` if *devfs* is not being used.

```
/*
 * If private data is not valid, we are not using devfs
 * so use the type (from minor nr.) to select a new f_op
 */
if (!filp->private_data && type) {
 if (type > SCULL_MAX_TYPE) return -ENODEV;
 filp->f_op = scull_fop_array[type];
 return filp->f_op->open(inode, filp); /* dispatch to specific open */
}

/* type 0, check the device number (unless private_data valid) */
dev = (Scull_Dev *)filp->private_data;
if (!dev) {
 if (num >= scull_nr_devs) return -ENODEV;
 dev = &scull_devices[num];
 filp->private_data = dev; /* for other methods */
}
```

Once equipped with the code shown, the *scull* module can be loaded to a system running *devfs*. It will show the following lines as output of *ls -l /dev/scull*:

```
crw-rw-rw- 1 root  root  144,  1 Jan 1 1970 0
crw-rw-rw- 1 root  root  144,  2 Jan 1 1970 1
crw-rw-rw- 1 root  root  144,  3 Jan 1 1970 2
crw-rw-rw- 1 root  root  144,  4 Jan 1 1970 3
crw-rw-rw- 1 root  root  144,  5 Jan 1 1970 pipe0
crw-rw-rw- 1 root  root  144,  6 Jan 1 1970 pipe1
crw-rw-rw- 1 root  root  144,  7 Jan 1 1970 pipe2
crw-rw-rw- 1 root  root  144,  8 Jan 1 1970 pipe3
crw-rw-rw- 1 root  root  144, 12 Jan 1 1970 priv
crw-rw-rw- 1 root  root  144,  9 Jan 1 1970 single
crw-rw-rw- 1 root  root  144, 10 Jan 1 1970 user
crw-rw-rw- 1 root  root  144, 11 Jan 1 1970 wuser
```

The functionality of the various files is the same as that of the "normal" *scull* module, the only difference being in device pathnames: what used to be */dev/scull0* is now */dev/scull/0*.

Portability Issues and devfs

The source files of *scull* are somewhat complicated by the need to be able to compile and run well with Linux versions 2.0, 2.2, and 2.4. This portability requirement brings in several instances of conditional compilation based on `CONFIG_DEVFS_FS`.

Fortunately, most developers agree that `#ifdef` constructs are basically bad when they appear in the body of function definitions (as opposed to being used in header files). Therefore, the addition of *devfs* brings in the needed machinery to completely avoid `#ifdef` in your code. We still have conditional compilation in *scull* because older versions of the kernel headers can't offer support for that.

If your code is meant to only be used with version 2.4 of the kernel, you can avoid conditional compilation by calling kernel functions to initialize the driver in both ways; things are arranged so that one of the initializations will do nothing at all, while returning success. The following is an example of what initialization might look like:

```
#include <devfs_fs_kernel.h>

int init_module()
{
 /* request a major: does nothing if devfs is used */
 result = devfs_register_chrdev(major, "name", &fops);
 if (result < 0) return result;

 /* register using devfs: does nothing if not in use */
 devfs_register(NULL, "name", /* .... */ );
 return 0;
}
```

You can resort to similar tricks in your own header files, as long as you are careful not to redefine functions that are already defined by kernel headers. Removing conditional compilation is a good thing because it improves readability of the code and reduces the amount of possible bugs by letting the compiler parse the whole input file. Whenever conditional compilation is used, there is the risk of introducing typos or other errors that can slip through unnoticed if they happen in a place that is discarded by the C preprocessor because of `#ifdef`.

This is, for example, how *scull.h* avoids conditional compilation in the cleanup part of the program. This code is portable to all kernel versions because it doesn't depend on *devfs* being known to the header files:

```
#ifdef CONFIG_DEVFS_FS /* only if enabled, to avoid errors in 2.0 */
#include <linux/devfs_fs_kernel.h>
#else
 typedef void * devfs_handle_t; /* avoid #ifdef inside the structure */
#endif
```

Nothing is defined in *sysdep.h* because it is very hard to implement this kind of hack generically enough to be of general use. Each driver should arrange for its own needs to avoid excessive `#ifdef` statements in function code. Also, we chose not to support *devfs* in the sample code for this book, with the exception of *scull*. We hope this discussion is enough to help readers exploit *devfs* if they want to; *devfs* support has been omitted from the rest of the sample files in order to keep the code simple.

Backward Compatibility

This chapter, so far, has described the kernel programming interface for version 2.4 of the Linux kernel. Unfortunately, this interface has changed significantly over the course of kernel development. These changes represent improvements in how things are done, but, once again, they also pose a challenge for those who wish to write drivers that are compatible across multiple versions of the kernel.

Insofar as this chapter is concerned, there are few noticeable differences between versions 2.4 and 2.2. Version 2.2, however, changed many of the prototypes of the `file_operations` methods from what 2.0 had; access to user space was greatly modified (and simplified) as well. The semaphore mechanism was not as well developed in Linux 2.0. And, finally, the 2.1 development series introduced the directory entry (dentry) cache.

Changes in the File Operations Structure

A number of factors drove the changes in the `file_operations` methods. The longstanding 2 GB file-size limit caused problems even in the Linux 2.0 days. As a result, the 2.1 development series started using the `loff_t` type, a 64-bit value, to represent file positions and lengths. Large file support was not completely integrated until version 2.4 of the kernel, but much of the groundwork was done earlier and had to be accommodated by driver writers.

Another change introduced during 2.1 development was the addition of the `f_pos` pointer argument to the *read* and *write* methods. This change was made to support the POSIX *pread* and *pwrite* system calls, which explicitly set the file offset where data is to be read or written. Without these system calls, threaded programs can run into race conditions when moving around in files.

Almost all methods in Linux 2.0 received an explicit `inode` pointer argument. The 2.1 development series removed this parameter from several of the methods, since it was rarely needed. If you need the `inode` pointer, you can still retrieve it from the `filp` argument.

The end result is that the prototypes of the commonly used `file_operations` methods looked like this in 2.0:

`int (*lseek) (struct inode *, struct file *, off_t, int);`

Note that this method is called *lseek* in Linux 2.0, instead of *llseek*. The name change was made to recognize that seeks could now happen with 64-bit offset values.

`int (*read) (struct inode *, struct file *, char *, int);`
`int (*write) (struct inode *, struct file *, const char *, int);`

As mentioned, these functions in Linux 2.0 had the `inode` pointer as an argument, and lacked the position argument.

`void (*release) (struct inode *, struct file *);`

In the 2.0 kernel, the *release* method could not fail, and thus returned `void`.

There have been many other changes to the `file_operations` structure; we will cover them in the following chapters as we get to them. Meanwhile, it is worth a moment to look at how portable code can be written that accounts for the changes we have seen so far. The changes in these methods are large, and there is no simple, elegant way to cover them over.

The way the sample code handles these changes is to define a set of small wrapper functions that "translate" from the old API to the new. These wrappers are only used when compiling under 2.0 headers, and must be substituted for the "real" device methods within the `file_operations` structure. This is the code implementing the wrappers for the *scull* driver:

```
/*
 * The following wrappers are meant to make things work with 2.0 kernels
 */
#ifdef LINUX_20
int scull_lseek_20(struct inode *ino, struct file *f,
    off_t offset, int whence)
{
 return (int)scull_llseek(f, offset, whence);
}

int scull_read_20(struct inode *ino, struct file *f, char *buf,
   int count)
{
 return (int)scull_read(f, buf, count, &f->f_pos);
}

int scull_write_20(struct inode *ino, struct file *f, const char *b,
   int c)
{
 return (int)scull_write(f, b, c, &f->f_pos);
}

void scull_release_20(struct inode *ino, struct file *f)
{
 scull_release(ino, f);
}

/* Redefine "real" names to the 2.0 ones */
#define scull_llseek scull_lseek_20
#define scull_read scull_read_20
#define scull_write scull_write_20
#define scull_release scull_release_20
#define llseek lseek
#endif /* LINUX_20 */
```

Redefining names in this manner can also account for structure members whose names have changed over time (such as the change from *lseek* to *llseek*).

Needless to say, this sort of redefinition of the names should be done with care; these lines should appear before the definition of the `file_operations` structure, but after any other use of those names.

Two other incompatibilities are related to the `file_operations` structure. One is that the *flush* method was added during the 2.1 development cycle. Driver writers almost never need to worry about this method, but its presence in the middle of the structure can still create problems. The best way to avoid dealing with the *flush* method is to use the tagged initialization syntax, as we did in all the sample source files.

The other difference is in the way an `inode` pointer is retrieved from a `filp` pointer. Whereas modern kernels use a `dentry` (directory entry) data structure, version 2.0 had no such structure. Therefore, *sysdep.h* defines a macro that should be used to portably access an `inode` from a `filp`:

```
#ifdef LINUX_20
# define INODE_FROM_F(filp) ((filp)->f_inode)
#else
# define INODE_FROM_F(filp) ((filp)->f_dentry->d_inode)
#endif
```

The Module Usage Count

In 2.2 and earlier kernels, the Linux kernel did not offer any assistance to modules in maintaining the usage count. Modules had to do that work themselves. This approach was error prone and required the duplication of a lot of work. It also encouraged race conditions. The new method is thus a definite improvement.

Code that is written to be portable, however, must be prepared to deal with the older way of doing things. That means that the usage count must still be incremented when a new reference is made to the module, and decremented when that reference goes away. Portable code must also work around the fact that the `owner` field did not exist in the `file_operations` structure in earlier kernels.

The easiest way to handle that is to use `SET_MODULE_OWNER`, rather than working with the `owner` field directly. In *sysdep.h*, we provide a null `SET_FILE_OWNER` for kernels that do not have this facility.

Changes in Semaphore Support

Semaphore support was less developed in the 2.0 kernel; support for SMP systems in general was primitive at that time. Drivers written for only that kernel version may not need to use semaphores at all, since only one CPU was allowed to be running kernel code at that time. Nonetheless, there may still be a need for semaphores, and it does not hurt to have the full protection needed by later kernel versions.

Most of the semaphore functions covered in this chapter existed in the 2.0 kernel. The one exception is *sema_init*; in version 2.0, programmers had to initialize semaphores manually. The *sysdep.h* header file handles this problem by defining a version of *sema_init* when compiled under the 2.0 kernel:

```
#ifdef LINUX_20
# ifdef MUTEX_LOCKED /* Only if semaphore.h included */
  extern inline void sema_init (struct semaphore *sem, int val)
  {
   sem->count = val;
   sem->waking = sem->lock = 0;
   sem->wait = NULL;
  }
# endif
#endif /* LINUX_20 */
```

Changes in Access to User Space

Finally, access to user space changed completely at the beginning of the 2.1 development series. The new interface has a better design and makes much better use of the hardware in ensuring safe access to user-space memory. But, of course, the interface is different. The 2.0 memory-access functions were as follows:

```
void memcpy_fromfs(void *to, const void *from, unsigned long count);
void memcpy_tofs(void *to, const void *from, unsigned long count);
```

The names of these functions come from the historical use of the `FS` segment register on the i386. Note that there is no return value from these functions; if the user supplies an invalid address, the data copy will silently fail. *sysdep.h* hides the renaming and allows you to portably call *copy_to_user* and *copy_from_user*.

Quick Reference

This chapter introduced the following symbols and header files. The list of the fields in `struct file_operations` and `struct file` is not repeated here.

`#include <linux/fs.h>`
: The "file system" header is the header required for writing device drivers. All the important functions are declared in here.

```
int register_chrdev(unsigned int major, const char
      *name, struct file_operations *fops);
```

Registers a character device driver. If the major number is not 0, it is used unchanged; if the number is 0, then a dynamic number is assigned for this device.

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

Unregisters the driver at unload time. Both `major` and the `name` string must contain the same values that were used to register the driver.

`kdev_t inode->i_rdev;`

The device "number" for the current device is accessible from the `inode` structure.

`int MAJOR(kdev_t dev);`
`int MINOR(kdev_t dev);`

These macros extract the major and minor numbers from a device item.

`kdev_t MKDEV(int major, int minor);`

This macro builds a `kdev_t` data item from the major and minor numbers.

`SET_MODULE_OWNER(struct file_operations *fops)`

This macro sets the `owner` field in the given `file_operations` structure.

`#include <asm/semaphore.h>`

Defines functions and types for the use of semaphores.

`void sema_init (struct semaphore *sem, int val);`

Initializes a semaphore to a known value. Mutual exclusion semaphores are usually initialized to a value of 1.

`int down_interruptible (struct semaphore *sem);`
`void up (struct semaphore *sem);`

Obtains a semaphore (sleeping, if necessary) and releases it, respectively.

`#include <asm/segment.h>`
`#include <asm/uaccess.h>`

segment.h defines functions related to cross-space copying in all kernels up to and including 2.0. The name was changed to *uaccess.h* in the 2.1 development series.

```
unsigned long __copy_from_user (void *to, const void *from,
      unsigned long count);
unsigned long __copy_to_user (void *to, const void *from,
      unsigned long count);
```

Copy data between user space and kernel space.

```
void memcpy_fromfs(void *to, const void *from, unsigned long
      count);
void memcpy_tofs(void *to, const void *from, unsigned long
      count);
```

These functions were used to copy an array of bytes from user space to kernel space and vice versa in version 2.0 of the kernel.

```
#include <linux/devfs_fs_kernel.h>
devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char
      *name, void *info);
devfs_handle_t devfs_register (devfs_handle_t dir, const
      char *name, unsigned int flags,
 unsigned int major, unsigned int minor, umode_t mode, void
      *ops, void *info);
void devfs_unregister (devfs_handle_t de);
```

These are the basic functions for registering devices with the device filesystem (*devfs*).