CHAPTER FIVE

ENHANCED CHAR DRIVER OPERATIONS

In Chapter 3, we built a complete device driver that the user can write to and read from. But a real device usually offers more functionality than synchronous *read* and *write*. Now that we're equipped with debugging tools should something go awry, we can safely go ahead and implement new operations.

What is normally needed, in addition to reading and writing the device, is the ability to perform various types of hardware control via the device driver. Control operations are usually supported via the *ioctl* method. The alternative is to look at the data flow being written to the device and use special sequences as control commands. This latter technique should be avoided because it requires reserving some characters for controlling purposes; thus, the data flow can't contain those characters. Moreover, this technique turns out to be more complex to handle than *ioctl*. Nonetheless, sometimes it's a useful approach to device control and is used by tty's and other devices. We'll describe it later in this chapter in "Device Control Without ioctl."

As we suggested in the previous chapter, the *ioctl* system call offers a device specific entry point for the driver to handle "commands." *ioctl* is device specific in that, unlike *read* and other methods, it allows applications to access features unique to the hardware being driven, such as configuring the device and entering or exiting operating modes. These control operations are usually not available through the read/write file abstraction. For example, everything you write to a serial port is used as communication data, and you cannot change the baud rate by writing to the device. That is what *ioctl* is for: controlling the I/O channel.

Another important feature of real devices (unlike *scull*) is that data being read or written is exchanged with other hardware, and some synchronization is needed. The concepts of blocking I/O and asynchronous notification fill the gap and are introduced in this chapter by means of a modified *scull* device. The driver uses interaction between different processes to create asynchronous events. As with the original *scull*, you don't need special hardware to test the driver's workings. We *will* definitely deal with real hardware, but not until Chapter 8.

ioctl

The *ioctl* function call in user space corresponds to the following prototype:

```
int ioctl(int fd, int cmd, ...);
```

The prototype stands out in the list of Unix system calls because of the dots, which usually represent not a variable number of arguments. In a real system, however, a system call can't actually have a variable number of arguments. System calls must have a well-defined number of arguments because user programs can access them only through hardware "gates," as outlined in "User Space and Kernel Space" in Chapter 2. Therefore, the dots in the prototype represent not a variable number of arguments but a single optional argument, traditionally identified as `char *argp`. The dots are simply there to prevent type checking during compilation. The actual nature of the third argument depends on the specific control command being issued (the second argument). Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the *ioctl* call; the device will then be able to exchange any amount of data with user space.

The *ioctl* driver method, on the other hand, receives its arguments according to this declaration:

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

The `inode` and `filp` pointers are the values corresponding to the file descriptor `fd` passed on by the application and are the same parameters passed to the *open* method. The `cmd` argument is passed from the user unchanged, and the optional `arg` argument is passed in the form of an `unsigned long`, regardless of whether it was given by the user as an integer or a pointer. If the invoking program doesn't pass a third argument, the `arg` value received by the driver operation has no meaningful value.

Because type checking is disabled on the extra argument, the compiler can't warn you if an invalid argument is passed to *ioctl*, and the programmer won't notice the error until runtime. This lack of checking can be seen as a minor problem with the *ioctl* definition, but it is a necessary price for the general functionality that *ioctl* provides.

As you might imagine, most *ioctl* implementations consist of a `switch` statement that selects the correct behavior according to the `cmd` argument. Different commands have different numeric values, which are usually given symbolic names to simplify coding. The symbolic name is assigned by a preprocessor definition. Custom drivers usually declare such symbols in their header files; *scull.h* declares them for *scull*. User programs must, of course, include that header file as well to have access to those symbols.

Choosing the ioctl Commands

Before writing the code for *ioctl*, you need to choose the numbers that correspond to commands. Unfortunately, the simple choice of using small numbers starting from 1 and going up doesn't work well.

The command numbers should be unique across the system in order to prevent errors caused by issuing the right command to the wrong device. Such a mismatch is not unlikely to happen, and a program might find itself trying to change the baud rate of a non-serial-port input stream, such as a FIFO or an audio device. If each *ioctl* number is unique, then the application will get an `EINVAL` error rather than succeeding in doing something unintended.

To help programmers create unique *ioctl* command codes, these codes have been split up into several bitfields. The first versions of Linux used 16-bit numbers: the top eight were the "magic" number associated with the device, and the bottom eight were a sequential number, unique within the device. This happened because Linus was "clueless" (his own word); a better division of bitfields was conceived only later. Unfortunately, quite a few drivers still use the old convention. They have to: changing the command codes would break no end of binary programs. In our sources, however, we will use the new command code convention exclusively.

To choose *ioctl* numbers for your driver according to the new convention, you should first check *include/asm/ioctl.h* and *Documentation/ioctl-number.txt*. The header defines the bitfields you will be using: type (magic number), ordinal number, direction of transfer, and size of argument. The *ioctl-number.txt* file lists the magic numbers used throughout the kernel, so you'll be able to choose your own magic number and avoid overlaps. The text file also lists the reasons why the convention should be used.

The old, and now deprecated, way of choosing an *ioctl* number was easy: authors chose a magic eight-bit number, such as "k" (hex `0x6b`), and added an ordinal number, like this:

```
#define SCULL_IOCTL1 0x6b01
#define SCULL_IOCTL2 0x6b02
/* .... */
```

If both the application and the driver agreed on the numbers, you only needed to implement the `switch` statement in your driver. However, this way of defining *ioctl* numbers, which had its foundations in Unix tradition, shouldn't be used any more. We've only shown the old way to give you a taste of what *ioctl* numbers look like.

The new way to define numbers uses four bitfields, which have the following meanings. Any new symbols we introduce in the following list are defined in `<linux/ioctl.h>`.

`type`
The magic number. Just choose one number (after consulting *ioctl-number.txt*) and use it throughout the driver. This field is eight bits wide (`_IOC_TYPEBITS`).

`number`
The ordinal (sequential) number. It's eight bits (`_IOC_NRBITS`) wide.

`direction`
The direction of data transfer, if the particular command involves a data transfer. The possible values are `_IOC_NONE` (no data transfer), `_IOC_READ`, `_IOC_WRITE`, and `_IOC_READ | _IOC_WRITE` (data is transferred both ways). Data transfer is seen from the application's point of view; `_IOC_READ` means reading *from* the device, so the driver must write to user space. Note that the field is a bit mask, so `_IOC_READ` and `_IOC_WRITE` can be extracted using a logical AND operation.

`size`
The size of user data involved. The width of this field is architecture dependent and currently ranges from 8 to 14 bits. You can find its value for your specific architecture in the macro `_IOC_SIZEBITS`. If you intend your driver to be portable, however, you can only count on a size up to 255. It's not mandatory that you use the `size` field. If you need larger data structures, you can just ignore it. We'll see soon how this field is used.

The header file `<asm/ioctl.h>`, which is included by `<linux/ioctl.h>`, defines macros that help set up the command numbers as follows: `_IO(type,nr)`, `_IOR(type,nr,dataitem)`, `_IOW(type,nr,dataitem)`, and `_IOWR(type,nr,dataitem)`. Each macro corresponds to one of the possible values for the direction of the transfer. The `type` and `number` fields are passed as arguments, and the `size` field is derived by applying *sizeof* to the `dataitem` argument. The header also defines macros to decode the numbers: `_IOC_DIR(nr)`, `_IOC_TYPE(nr)`, `_IOC_NR(nr)`, and `_IOC_SIZE(nr)`. We won't go into any more detail about these macros because the header file is clear, and sample code is shown later in this section.

Here is how some *ioctl* commands are defined in *scull*. In particular, these commands set and get the driver's configurable parameters.

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC 'k'

#define SCULL_IOCRESET _IO(SCULL_IOC_MAGIC, 0)

/*
 * S means "Set" through a ptr
 * T means "Tell" directly with the argument value
 * G means "Get": reply by setting through a pointer
 * Q means "Query": response is on the return value
 * X means "eXchange": G and S atomically
 * H means "sHift": T and Q atomically
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, scull_quantum)
#define SCULL_IOCSQSET  _IOW(SCULL_IOC_MAGIC, 2, scull_qset)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,  3)
#define SCULL_IOCTQSET  _IO(SCULL_IOC_MAGIC,  4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, scull_quantum)
#define SCULL_IOCGQSET  _IOR(SCULL_IOC_MAGIC, 6, scull_qset)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,  7)
#define SCULL_IOCQQSET  _IO(SCULL_IOC_MAGIC,  8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, scull_quantum)
#define SCULL_IOCXQSET  _IOWR(SCULL_IOC_MAGIC,10, scull_qset)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQSET  _IO(SCULL_IOC_MAGIC, 12)
#define SCULL_IOCHARDRESET _IO(SCULL_IOC_MAGIC, 15) /* debugging tool */

#define SCULL_IOC_MAXNR 15
```

The last command, `HARDRESET`, is used to reset the module's usage count to 0 so that the module can be unloaded should something go wrong with the counter. The actual source file also defines all the commands between `IOCHQSET` and `HARDRESET`, although they're not shown here.

We chose to implement both ways of passing integer arguments—by pointer and by explicit value, although by an established convention *ioctl* should exchange values by pointer. Similarly, both ways are used to return an integer number: by pointer or by setting the return value. This works as long as the return value is a positive integer; on return from any system call, a positive value is preserved (as we saw for *read* and *write*), while a negative value is considered an error and is used to set `errno` in user space.

The "exchange" and "shift" operations are not particularly useful for *scull*. We implemented "exchange" to show how the driver can combine separate operations into a single *atomic* one, and "shift" to pair "tell" and "query." There are times when atomic* test-and-set operations like these are needed, in particular, when applications need to set or release locks.

The explicit ordinal number of the command has no specific meaning. It is used only to tell the commands apart. Actually, you could even use the same ordinal number for a read command and a write command, since the actual *ioctl* number is different in the "direction" bits, but there is no reason why you would want to do so. We chose not to use the ordinal number of the command anywhere but in the declaration, so we didn't assign a symbolic value to it. That's why explicit

---

* A fragment of program code is said to be atomic when it will always be executed as though it were a single instruction, without the possibility of the processor being interrupted and something happening in between (such as somebody else's code running).

numbers appear in the definition given previously. The example shows one way to use the command numbers, but you are free to do it differently.

The value of the *ioctl* `cmd` argument is not currently used by the kernel, and it's quite unlikely it will be in the future. Therefore, you could, if you were feeling lazy, avoid the complex declarations shown earlier and explicitly declare a set of scalar numbers. On the other hand, if you did, you wouldn't benefit from using the bitfields. The header `<linux/kd.h>` is an example of this old-fashioned approach, using 16-bit scalar values to define the *ioctl* commands. That source file relied on scalar numbers because it used the technology then available, not out of laziness. Changing it now would be a gratuitous incompatibility.

The Return Value

The implementation of *ioctl* is usually a `switch` statement based on the command number. But what should the `default` selection be when the command number doesn't match a valid operation? The question is controversial. Several kernel functions return `-EINVAL` ("Invalid argument"), which makes sense because the command argument is indeed not a valid one. The POSIX standard, however, states that if an inappropriate *ioctl* command has been issued, then `-ENOTTY` should be returned. The string associated with that value used to be "Not a typewriter" under all libraries up to and including *libc5*. Only *libc6* changed the message to "Inappropriate ioctl for device," which looks more to the point. Because most recent Linux system are *libc6* based, we'll stick to the standard and return `-ENOTTY`. It's still pretty common, though, to return `-EINVAL` in response to an invalid *ioctl* command.

The Predefined Commands

Though the *ioctl* system call is most often used to act on devices, a few commands are recognized by the kernel. Note that these commands, when applied to your device, are decoded *before* your own file operations are called. Thus, if you choose the same number for one of your *ioctl* commands, you won't ever see any request for that command, and the application will get something unexpected because of the conflict between the *ioctl* numbers.

The predefined commands are divided into three groups:

- Those that can be issued on any file (regular, device, *FIFO*, or socket)
- Those that are issued only on regular files
- Those specific to the filesystem type

Commands in the last group are executed by the implementation of the hosting filesystem (see the *chattr* command). Device driver writers are interested only in the first group of commands, whose magic number is "T." Looking at the workings of the other groups is left to the reader as an exercise; *ext2_ioctl* is a most interesting function (though easier than you may expect), because it implements the append-only flag and the immutable flag.

The following *ioctl* commands are predefined for any file:

`FIOCLEX`
: Set the close-on-exec flag (File IOctl CLose on EXec). Setting this flag will cause the file descriptor to be closed when the calling process executes a new program.

`FIONCLEX`
: Clear the close-on-exec flag.

`FIOASYNC`
: Set or reset asynchronous notification for the file (as discussed in "Asynchronous Notification" later in this chapter). Note that kernel versions up to Linux 2.2.4 incorrectly used this command to modify the `O_SYNC` flag. Since both actions can be accomplished in other ways, nobody actually uses the `FIOASYNC` command, which is reported here only for completeness.

`FIONBIO`
: "File IOctl Non-Blocking I/O" (described later in this chapter in "Blocking and Nonblocking Operations"). This call modifies the `O_NONBLOCK` flag in `filp->f_flags`. The third argument to the system call is used to indicate whether the flag is to be set or cleared. We'll look at the role of the flag later in this chapter. Note that the flag can also be changed by the *fcntl* system call, using the *F_SETFL* command.

The last item in the list introduced a new system call, *fcntl*, which looks like *ioctl*. In fact, the *fcntl* call is very similar to *ioctl* in that it gets a command argument and an extra (optional) argument. It is kept separate from *ioctl* mainly for historical reasons: when Unix developers faced the problem of controlling I/O operations, they decided that files and devices were different. At the time, the only devices with *ioctl* implementations were ttys, which explains why `-ENOTTY` is the standard reply for an incorrect *ioctl* command. Things have changed, but *fcntl* remains in the name of backward compatibility.

Using the ioctl Argument

Another point we need to cover before looking at the *ioctl* code for the *scull* driver is how to use the extra argument. If it is an integer, it's easy: it can be used directly. If it is a pointer, however, some care must be taken.

When a pointer is used to refer to user space, we must ensure that the user address is valid and that the corresponding page is currently mapped. If kernel code tries to access an out-of-range address, the processor issues an exception.

Exceptions in kernel code are turned to oops messages by every Linux kernel up through 2.0.*x*; version 2.1 and later handle the problem more gracefully. In any case, it's the driver's responsibility to make proper checks on every user-space address it uses and to return an error if it is invalid.

Address verification for kernels 2.2.*x* and beyond is implemented by the function *access_ok*, which is declared in `<asm/uaccess.h>`:

```
int access_ok(int type, const void *addr, unsigned long size);
```

The first argument should be either `VERIFY_READ` or `VERIFY_WRITE`, depending on whether the action to be performed is reading the user-space memory area or writing it. The `addr` argument holds a user-space address, and `size` is a byte count. If *ioctl*, for instance, needs to read an integer value from user space, `size` is `sizeof(int)`. If you need to both read and write at the given address, use `VERIFY_WRITE`, since it is a superset of `VERIFY_READ`.

Unlike most functions, *access_ok* returns a boolean value: 1 for success (access is OK) and 0 for failure (access is not OK). If it returns false, the driver will usually return `-EFAULT` to the caller.

There are a couple of interesting things to note about *access_ok*. First is that it does not do the complete job of verifying memory access; it only checks to see that the memory reference is in a region of memory that the process might reasonably have access to. In particular, *access_ok* ensures that the address does not point to kernel-space memory. Second, most driver code need not actually call *access_ok*. The memory-access routines described later take care of that for you. We will nonetheless demonstrate its use so that you can see how it is done, and for backward compatibility reasons that we will get into toward the end of the chapter.

The *scull* source exploits the bitfields in the *ioctl* number to check the arguments before the `switch`:

```
int err = 0, tmp;
int ret = 0;

/*
 * extract the type and number bitfields, and don't decode
 * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bitmask, and VERIFY_WRITE catches R/W
 * transfers. `Type' is user oriented, while
 * access_ok is kernel oriented, so the concept of "read" and
 * "write" is reversed
 */
```

```
    if (_IOC_DIR(cmd) & _IOC_READ)
      err = !access_ok(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
      err = !access_ok(VERIFY_READ, (void *)arg, _IOC_SIZE(cmd));
    if (err) return -EFAULT;
```

After calling *access_ok*, the driver can safely perform the actual transfer. In addition to the *copy_from_user* and *copy_to_user* functions, the programmer can exploit a set of functions that are optimized for the most-used data sizes (one, two, and four bytes, as well as eight bytes on 64-bit platforms). These functions are described in the following list and are defined in `<asm/uaccess.h>`.

`put_user(datum, ptr)`
`__put_user(datum, ptr)`

These macros write the datum to user space; they are relatively fast, and should be called instead of *copy_to_user* whenever single values are being transferred. Since type checking is not performed on macro expansion, you can pass any type of pointer to *put_user*, as long as it is a user-space address. The size of the data transfer depends on the type of the `ptr` argument and is determined at compile time using a special *gcc* pseudo-function that isn't worth showing here. As a result, if `ptr` is a char pointer, one byte is transferred, and so on for two, four, and possibly eight bytes.

put_user checks to ensure that the process is able to write to the given memory address. It returns 0 on success, and `-EFAULT` on error. *__put_user* performs less checking (it does not call *access_ok*), but can still fail on some kinds of bad addresses. Thus, *__put_user* should only be used if the memory region has already been verified with *access_ok*.

As a general rule, you'll call *__put_user* to save a few cycles when you are implementing a *read* method, or when you copy several items and thus call *access_ok* just once before the first data transfer.

`get_user(local, ptr)`
`__get_user(local, ptr)`

These macros are used to retrieve a single datum from user space. They behave like *put_user* and *__put_user*, but transfer data in the opposite direction. The value retrieved is stored in the local variable `local`; the return value indicates whether the operation succeeded or not. Again, *__get_user* should only be used if the address has already been verified with *access_ok*.

If an attempt is made to use one of the listed functions to transfer a value that does not fit one of the specific sizes, the result is usually a strange message from the compiler, such as "conversion to non-scalar type requested." In such cases, *copy_to_user* or *copy_from_user* must be used.

Capabilities and Restricted Operations

Access to a device is controlled by the permissions on the device file(s), and the driver is not normally involved in permissions checking. There are situations, however, where any user is granted read/write permission on the device, but some other operations should be denied. For example, not all users of a tape drive should be able to set its default block size, and the ability to work with a disk device does not mean that the user can reformat the drive. In cases like these, the driver must perform additional checks to be sure that the user is capable of performing the requested operation.

Unix systems have traditionally restricted privileged operations to the superuser account. Privilege is an all-or-nothing thing—the superuser can do absolutely anything, but all other users are highly restricted. The Linux kernel as of version 2.2 provides a more flexible system called *capabilities*. A capability-based system leaves the all-or-nothing mode behind and breaks down privileged operations into separate subgroups. In this way, a particular user (or program) can be empowered to perform a specific privileged operation without giving away the ability to perform other, unrelated operations. Capabilities are still little used in user space, but kernel code uses them almost exclusively.

The full set of capabilities can be found in `<linux/capability.h>`. A subset of those capabilities that might be of interest to device driver writers includes the following:

`CAP_DAC_OVERRIDE`
The ability to override access restrictions on files and directories.

`CAP_NET_ADMIN`
The ability to perform network administration tasks, including those which affect network interfaces.

`CAP_SYS_MODULE`
The ability to load or remove kernel modules.

`CAP_SYS_RAWIO`
The ability to perform "raw" I/O operations. Examples include accessing device ports or communicating directly with USB devices.

`CAP_SYS_ADMIN`
A catch-all capability that provides access to many system administration operations.

`CAP_SYS_TTY_CONFIG`
The ability to perform tty configuration tasks.

Before performing a privileged operation, a device driver should check that the calling process has the appropriate capability with the *capable* function (defined in `<sys/sched.h>`):

```
int capable(int capability);
```

In the *scull* sample driver, any user is allowed to query the quantum and quantum set sizes. Only privileged users, however, may change those values, since inappropriate values could badly affect system performance. When needed, the *scull* implementation of *ioctl* checks a user's privilege level as follows:

```
if (! capable (CAP_SYS_ADMIN))
  return -EPERM;
```

In the absence of a more specific capability for this task, `CAP_SYS_ADMIN` was chosen for this test.

The Implementation of the ioctl Commands

The *scull* implementation of *ioctl* only transfers the configurable parameters of the device and turns out to be as easy as the following:

```
    switch(cmd) {

  #ifdef SCULL_DEBUG
      case SCULL_IOCHARDRESET:
        /*
         * reset the counter to 1, to allow unloading in case
         * of problems. Use 1, not 0, because the invoking
         * process has the device open.
         */
        while (MOD_IN_USE)
          MOD_DEC_USE_COUNT;
        MOD_INC_USE_COUNT;
        /* don't break: fall through and reset things */
  #endif /* SCULL_DEBUG */

      case SCULL_IOCRESET:
       scull_quantum = SCULL_QUANTUM;
       scull_qset = SCULL_QSET;
       break;

      case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
       if (! capable (CAP_SYS_ADMIN))
         return -EPERM;
       ret = __get_user(scull_quantum, (int *)arg);
       break;

      case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
       if (! capable (CAP_SYS_ADMIN))
         return -EPERM;
       scull_quantum = arg;
       break;
      case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
       ret = __put_user(scull_quantum, (int *)arg);
       break;

      case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
       return scull_quantum;

      case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
       if (! capable (CAP_SYS_ADMIN))
         return -EPERM;
       tmp = scull_quantum;
       ret = __get_user(scull_quantum, (int *)arg);
       if (ret == 0)
         ret = __put_user(tmp, (int *)arg);
       break;

      case SCULL_IOCHQUANTUM: /* sHift: like Tell + Query */
       if (! capable (CAP_SYS_ADMIN))
         return -EPERM;
       tmp = scull_quantum;
       scull_quantum = arg;
       return tmp;

      default: /* redundant, as cmd was checked against MAXNR */
       return -ENOTTY;
    }
    return ret;
```

scull also includes six entries that act on `scull_qset`. These entries are identical to the ones for `scull_quantum` and are not worth showing in print.

The six ways to pass and receive arguments look like the following from the caller's point of view (i.e., from user space):

```
int quantum;

ioctl(fd,SCULL_IOCSQUANTUM, &quantum);
ioctl(fd,SCULL_IOCTQUANTUM, quantum);

ioctl(fd,SCULL_IOCGQUANTUM, &quantum);
quantum = ioctl(fd,SCULL_IOCQQUANTUM);

ioctl(fd,SCULL_IOCXQUANTUM, &quantum);
quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum);
```

Of course, a normal driver would not implement such a mix of calling modes in one place. We have done so here only to demonstrate the different ways in which things could be done. Normally, however, data exchanges would be consistently performed, either through pointers (more common) or by value (less common), and mixing of the two techniques would be avoided.

Device Control Without ioctl

Sometimes controlling the device is better accomplished by writing control sequences to the device itself. This technique is used, for example, in the console driver, where so-called escape sequences are used to move the cursor, change the default color, or perform other configuration tasks. The benefit of implementing device control this way is that the user can control the device just by writing data, without needing to use (or sometimes write) programs built just for configuring the device.

For example, the *setterm* program acts on the console (or another terminal) configuration by printing escape sequences. This behavior has the advantage of permitting the remote control of devices. The controlling program can live on a different computer than the controlled device, because a simple redirection of the data stream does the configuration job. You're already used to this with ttys, but the technique is more general.

The drawback of controlling by printing is that it adds policy constraints to the device; for example, it is viable only if you are sure that the control sequence can't appear in the data being written to the device during normal operation. This is only partly true for ttys. Although a text display is meant to display only ASCII characters, sometimes control characters can slip through in the data being written and can thus affect the console setup. This can happen, for example, when you issue *grep* on a binary file; the extracted lines can contain anything, and you often end up with the wrong font on your console.*

Controlling by write *is* definitely the way to go for those devices that don't transfer data but just respond to commands, such as robotic devices.

For instance, a driver written for fun by one of your authors moves a camera on two axes. In this driver, the "device" is simply a pair of old stepper motors, which can't really be read from or written to. The concept of "sending a data stream" to a stepper motor makes little or no sense. In this case, the driver interprets what is being written as ASCII commands and converts the requests to sequences of impulses that manipulate the stepper motors. The idea is similar, somewhat, to the AT commands you send to the modem in order to set up communication, the main difference being that the serial port used to communicate with the modem must transfer real data as well. The advantage of direct device control is that you can use *cat* to move the camera without writing and compiling special code to issue the *ioctl* calls.

---

* CTRL-N sets the alternate font, which is made up of graphic symbols and thus isn't a friendly font for typing input to your shell; if you encounter this problem, echo a CTRL-O character to restore the primary font.

When writing command-oriented drivers, there's no reason to implement the *ioctl* method. An additional command in the interpreter is easier to implement and use.

Sometimes, though, you might choose to act the other way around: instead of making *write* into an interpreter and avoiding *ioctl*, you might choose to avoid *write* altogether and use *ioctl* commands exclusively, while accompanying the driver with a specific command-line tool to send those commands to the driver. This approach moves the complexity from kernel space to user space, where it may be easier to deal with, and helps keep the driver small while denying use of simple *cat* or *echo* commands.

Blocking I/O

One problem that might arise with *read* is what to do when there's no data *yet*, but we're not at end-of-file.

The default answer is "go to sleep waiting for data." This section shows how a process is put to sleep, how it is awakened, and how an application can ask if there is data without just blindly issuing a *read* call and blocking. We then apply the same concepts to *write*.

As usual, before we show actual code, we'll explain a few concepts.

Going to Sleep and Awakening

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. At some future time, when the event being waited for occurs, the process will be woken up and will continue with its job. This section discusses the 2.4 machinery for putting a process to sleep and waking it up. Earlier versions are discussed in "Backward Compatibility" later in this chapter.

There are several ways of handling sleeping and waking up in Linux, each suited to different needs. All, however, work with the same basic data type, a wait queue (`wait_queue_head_t`). A *wait queue* is exactly that—a queue of processes that are waiting for an event. Wait queues are declared and initialized as follows:

```
wait_queue_head_t my_queue;
init_waitqueue_head (&my_queue);
```

When a wait queue is declared statically (i.e., not as an automatic variable of a procedure or as part of a dynamically-allocated data structure), it is also possible to initialize the queue at compile time:

```
DECLARE_WAIT_QUEUE_HEAD (my_queue);
```

It is a common mistake to neglect to initialize a wait queue (especially since earlier versions of the kernel did not require this initialization); if you forget, the results will usually not be what you intended.

Once the wait queue is declared and initialized, a process may use it to go to sleep. Sleeping is accomplished by calling one of the variants of *sleep_on*, depending on how deep a sleep is called for.

`sleep_on(wait_queue_head_t *queue);`
: Puts the process to sleep on this queue. *sleep_on* has the disadvantage of not being interruptible; as a result, the process can end up being stuck (and unkillable) if the event it's waiting for never happens.

`interruptible_sleep_on(wait_queue_head_t *queue);`
: The interruptible variant works just like *sleep_on*, except that the sleep can be interrupted by a signal. This is the form that device driver writers have been using for a long time, before *wait_event_interruptible* (described later) appeared.

`sleep_on_timeout(wait_queue_head_t *queue, long timeout);`
`interruptible_sleep_on_timeout(wait_queue_head_t *queue, long timeout);`
: These two functions behave like the previous two, with the exception that the sleep will last no longer than the given timeout period. The timeout is specified in "jiffies," which are covered in Chapter 6.

`void wait_event(wait_queue_head_t queue, int condition);`
`int wait_event_interruptible(wait_queue_head_t queue, int condition);`
: These macros are the preferred way to sleep on an event. They combine waiting for an event and testing for its arrival in a way that avoids race conditions. They will sleep until the condition, which may be any boolean C expression, evaluates true. The macros expand to a *while* loop, and the condition is reevaluated over time—the behavior is different from that of a function call or a simple macro, where the arguments are evaluated only at call time. The latter macro is implemented as an expression that evaluates to 0 in case of success and `-ERESTARTSYS` if the loop is interrupted by a signal.

It is worth repeating that driver writers should almost always use the *interruptible* instances of these functions/macros. The noninterruptible version exists for the small number of situations in which signals cannot be dealt with, for example, when waiting for a data page to be retrieved from swap space. Most drivers do not present such special situations.

Of course, sleeping is only half of the problem; something, somewhere will have to wake the process up again. When a device driver sleeps directly, there is usually code in another part of the driver that performs the wakeup, once it knows that the event has occurred. Typically a driver will wake up sleepers in its interrupt handler once new data has arrived. Other scenarios are possible, however.

Just as there is more than one way to sleep, so there is also more than one way to wake up. The high-level functions provided by the kernel to wake up processes are as follows:

`wake_up(wait_queue_head_t *queue);`
: This function will wake up all processes that are waiting on this event queue.

`wake_up_interruptible(wait_queue_head_t *queue);`
: *wake_up_interruptible* wakes up only the processes that are in interruptible sleeps. Any process that sleeps on the wait queue using a noninterruptible function or macro will continue to sleep.

`wake_up_sync(wait_queue_head_t *queue);`
`wake_up_interruptible_sync(wait_queue_head_t *queue);`
: Normally, a *wake_up* call can cause an immediate reschedule to happen, meaning that other processes might run before *wake_up* returns. The "synchronous" variants instead make any awakened processes runnable, but do not reschedule the CPU. This is used to avoid rescheduling when the current process is known to be going to sleep, thus forcing a reschedule anyway. Note that awakened processes could run immediately on a different processor, so these functions should not be expected to provide mutual exclusion.

If your driver is using *interruptible_sleep_on*, there is little difference between *wake_up* and *wake_up_interruptible*. Calling the latter is a common convention, however, to preserve consistency between the two calls.

As an example of wait queue usage, imagine you want to put a process to sleep when it reads your device and awaken it when someone else writes to the device. The following code does just that:

```
DECLARE_WAIT_QUEUE_HEAD(wq);

ssize_t sleepy_read (struct file *filp, char *buf, size_t count,
   loff_t *pos)
{
  printk(KERN_DEBUG "process %i (%s) going to sleep\n",
      current->pid, current->comm);
  interruptible_sleep_on(&wq);
  printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
  return 0; /* EOF */
}
```

```
ssize_t sleepy_write (struct file *filp, const char *buf, size_t count,
              loff_t *pos)
{
  printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
     current->pid, current->comm);
  wake_up_interruptible(&wq);
  return count; /* succeed, to avoid retrial */
}
```

The code for this device is available as sleepy in the example programs and can be tested using *cat* and input/output redirection, as usual.

An important thing to remember with wait queues is that being woken up does not guarantee that the event you were waiting for has occurred; a process can be woken for other reasons, mainly because it received a signal. Any code that sleeps should do so in a loop that tests the condition after returning from the sleep, as discussed in “A Sample Implementation: scullpipe” later in this chapter.

A Deeper Look at Wait Queues

The previous discussion is all that most driver writers will need to know to get their job done. Some, however, will want to dig deeper. This section attempts to get the curious started; everybody else can skip to the next section without missing much that is important.

The `wait_queue_head_t` type is a fairly simple structure, defined in `<linux/wait.h>`. It contains only a lock variable and a linked list of sleeping processes. The individual data items in the list are of type `wait_queue_t`, and the list is the generic list defined in `<linux/list.h>` and described in “Linked Lists” in Chapter 10. Normally the `wait_queue_t` structures are allocated on the stack by functions like *interruptible_sleep_on*; the structures end up in the stack because they are simply declared as automatic variables in the relevant functions. In general, the programmer need not deal with them.

Some advanced applications, however, can require dealing with `wait_queue_t` variables directly. For these, it’s worth a quick look at what actually goes on inside a function like *interruptible_sleep_on*. The following is a *simplified* version of the implementation of *interruptible_sleep_on* to put a process to sleep:

```
void simplified_sleep_on(wait_queue_head_t *queue)
{
  wait_queue_t wait;

  init_waitqueue_entry(&wait, current);
  current->state = TASK_INTERRUPTIBLE;

  add_wait_queue(queue, &wait);
  schedule();
  remove_wait_queue (queue, &wait);
 }
```

The code here creates a new `wait_queue_t` variable (`wait`, which gets allocated on the stack) and initializes it. The state of the task is set to `TASK_INTERRUPTIBLE`, meaning that it is in an interruptible sleep. The wait queue entry is then added to the queue (the `wait_queue_head_t *` argument). Then *schedule* is called, which relinquishes the processor to somebody else. *schedule* returns only when somebody else has woken up the process and set its state to `TASK_RUNNING`. At that point, the wait queue entry is removed from the queue, and the sleep is done.

Figure 5-1 shows the internals of the data structures involved in wait queues and how they are used by processes.

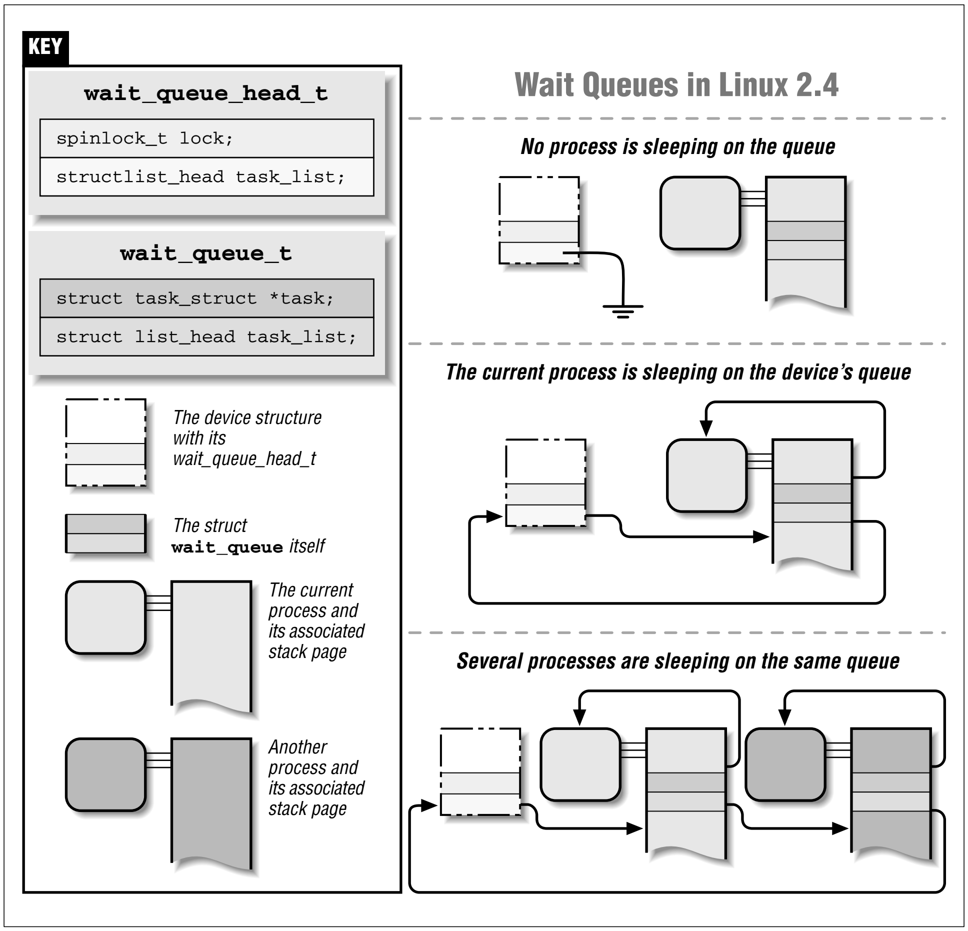
KEY
wait_queue_head_t
spinlock_t lock;
structlist_head task_list;
wait_queue_t
struct task_struct *task;
struct list_head task_list;
The device structure with its wait_queue_head_t
The struct **wait_queue** itself
The current process and its associated stack page
Another process and its associated stack page
Wait Queues in Linux 2.4
No process is sleeping on the queue
The current process is sleeping on the device's queue
Several processes are sleeping on the same queue

Figure 5-1. Wait queues in Linux 2.4

A quick look through the kernel shows that a great many procedures do their sleeping "manually" with code that looks like the previous example. Most of those implementations date back to kernels prior to 2.2.3, before *wait_event* was introduced. As suggested, *wait_event* is now the preferred way to sleep on an event, because *interruptible_sleep_on* is subject to unpleasant race conditions. A full description of how that can happen will have to wait until "Going to Sleep Without Races" in Chapter 9; the short version, simply, is that things can change in the time between when your driver decides to sleep and when it actually gets around to calling *interruptible_sleep_on*.

One other reason for calling the scheduler explicitly, however, is to do *exclusive* waits. There can be situations in which several processes are waiting on an event; when *wake_up* is called, all of those processes will try to execute. Suppose that the event signifies the arrival of an atomic piece of data. Only one process will be able to read that data; all the rest will simply wake up, see that no data is available, and go back to sleep.

This situation is sometimes referred to as the "thundering herd problem." In high-performance situations, thundering herds can waste resources in a big way. The creation of a large number of runnable processes that can do no useful work generates a large number of context switches and processor overhead, all for nothing. Things would work better if those processes simply remained asleep.

For this reason, the 2.3 development series added the concept of an *exclusive sleep*. If processes sleep in an exclusive mode, they are telling the kernel to wake only one of them. The result is improved performance in some situations.

The code to perform an exclusive sleep looks very similar to that for a regular sleep:

```
void simplified_sleep_exclusive(wait_queue_head_t *queue)
{
  wait_queue_t wait;

  init_waitqueue_entry(&wait, current);
  current->state = TASK_INTERRUPTIBLE | TASK_EXCLUSIVE;

  add_wait_queue_exclusive(queue, &wait);
  schedule();
  remove_wait_queue (queue, &wait);
 }
```

Adding the `TASK_EXCLUSIVE` flag to the task state indicates that the process is in an exclusive wait. The call to *add_wait_queue_exclusive* is also necessary, however. That function adds the process to the *end* of the wait queue, behind all others. The purpose is to leave any processes in nonexclusive sleeps at the beginning, where they will always be awakened. As soon as *wake_up* hits the first exclusive sleeper, it knows it can stop.

The attentive reader may have noticed another reason to manipulate wait queues and the scheduler explicitly. Whereas functions like *sleep_on* will block a process on exactly one wait queue, working with the queues directly allows sleeping on multiple queues simultaneously. Most drivers need not sleep on more than one queue; if yours is the exception, you will need to use code like what we've shown.

Those wanting to dig even deeper into the wait queue code can look at `<linux/sched.h>` and `kernel/sched.c`.

Writing Reentrant Code

When a process is put to sleep, the driver is still alive and can be called by another process. Let's consider the console driver as an example. While an application is waiting for keyboard input on `tty1`, the user switches to `tty2` and spawns a new shell. Now both shells are waiting for keyboard input within the console driver, although they sleep on different wait queues: one on the queue associated with `tty1` and the other on the queue associated with `tty2`. Each process is blocked within the *interruptible_sleep_on* function, but the driver can still receive and answer requests from other ttys.

Of course, on SMP systems, multiple simultaneous calls to your driver can happen even when you do not sleep.

Such situations can be handled painlessly by writing *reentrant code*. Reentrant code is code that doesn't keep status information in global variables and thus is able to manage interwoven invocations without mixing anything up. If all the status information is process specific, no interference will ever happen.

If status information is needed, it can either be kept in local variables within the driver function (each process has a different stack page in kernel space where local variables are stored), or it can reside in `private_data` within the `filp` accessing the file. Using local variables is preferred because sometimes the same `filp` can be shared between two processes (usually parent and child).

If you need to save large amounts of status data, you can keep the pointer in a local variable and use *kmalloc* to retrieve the actual storage space. In this case you must remember to *kfree* the data, because there's no equivalent to "everything is released at process termination" when you're working in kernel space. Using local variables for large items is not good practice, because the data may not fit the single page of memory allocated for stack space.

You need to make reentrant any function that matches either of two conditions. First, if it calls *schedule*, possibly by calling *sleep_on* or *wake_up*. Second, if it copies data to or from user space, because access to user space might page-fault, and the process will be put to sleep while the kernel deals with the missing page.

Every function that calls any such functions must be reentrant as well. For example, if *sample_read* calls *sample_getdata*, which in turn can block, then *sample_read* must be reentrant as well as *sample_getdata*, because nothing prevents another process from calling it while it is already executing on behalf of a process that went to sleep.

Finally, of course, code that sleeps should always keep in mind that the state of the system can change in almost any way while a process is sleeping. The driver should be careful to check any aspect of its environment that might have changed while it wasn't paying attention.

Blocking and Nonblocking Operations

Another point we need to touch on before we look at the implementation of full-featured *read* and *write* methods is the role of the `O_NONBLOCK` flag in `filp->f_flags`. The flag is defined in `<linux/fcntl.h>`, which is automatically included by `<linux/fs.h>`.

The flag gets its name from "open-nonblock," because it can be specified at open time (and originally could only be specified there). If you browse the source code, you'll find some references to an `O_NDELAY` flag; this is an alternate name for `O_NONBLOCK`, accepted for compatibility with System V code. The flag is cleared by default, because the normal behavior of a process waiting for data is just to sleep. In the case of a blocking operation, which is the default, the following behavior should be implemented in order to adhere to the standard semantics:

- If a process calls *read* but no data is (yet) available, the process must block. The process is awakened as soon as some data arrives, and that data is returned to the caller, even if there is less than the amount requested in the `count` argument to the method.
- If a process calls *write* and there is no space in the buffer, the process must block, and it must be on a different wait queue from the one used for reading. When some data has been written to the hardware device, and space becomes free in the output buffer, the process is awakened and the *write* call succeeds, although the data may be only partially written if there isn't room in the buffer for the `count` bytes that were requested.

Both these statements assume that there are both input and output buffers; in practice, almost every device driver has them. The input buffer is required to avoid losing data that arrives when nobody is reading. In contrast, data can't be lost on *write*, because if the system call doesn't accept data bytes, they remain in the user-space buffer. Even so, the output buffer is almost always useful for squeezing more performance out of the hardware.

The performance gain of implementing an output buffer in the driver results from the reduced number of context switches and user-level/kernel-level transitions. Without an output buffer (assuming a slow device), only one or a few characters are accepted by each system call, and while one process sleeps in *write*, another process runs (that's one context switch). When the first process is awakened, it resumes (another context switch), *write* returns (kernel/user transition), and the process reiterates the system call to write more data (user/kernel transition); the call blocks, and the loop continues. If the output buffer is big enough, the *write* call succeeds on the first attempt—the buffered data will be pushed out to the device later, at interrupt time—without control needing to go back to user space for a second or third *write* call. The choice of a suitable size for the output buffer is clearly device specific.

We didn't use an input buffer in *scull*, because data is already available when *read* is issued. Similarly, no output buffer was used, because data is simply copied to the memory area associated with the device. Essentially, the device *is* a buffer, so the implementation of additional buffers would be superfluous. We'll see the use of buffers in Chapter 9, in the section titled "Interrupt-Driven I/O."

The behavior of *read* and *write* is different if `O_NONBLOCK` is specified. In this case, the calls simply return `-EAGAIN` if a process calls *read* when no data is available or if it calls *write* when there's no space in the buffer.

As you might expect, nonblocking operations return immediately, allowing the application to poll for data. Applications must be careful when using the *stdio* functions while dealing with nonblocking files, because they can easily mistake a nonblocking return for `EOF`. They always have to check `errno`.

Naturally, `O_NONBLOCK` is meaningful in the *open* method also. This happens when the call can actually block for a long time; for example, when opening a FIFO that has no writers (yet), or accessing a disk file with a pending lock. Usually, opening a device either succeeds or fails, without the need to wait for external events. Sometimes, however, opening the device requires a long initialization, and you may choose to support `O_NONBLOCK` in your *open* method by returning immediately with `-EAGAIN` ("try it again") if the flag is set, after initiating device initialization. The driver may also implement a blocking *open* to support access policies in a way similar to file locks. We'll see one such implementation in the section "Blocking open as an Alternative to EBUSY" later in this chapter.

Some drivers may also implement special semantics for `O_NONBLOCK`; for example, an open of a tape device usually blocks until a tape has been inserted. If the tape drive is opened with `O_NONBLOCK`, the open succeeds immediately regardless of whether the media is present or not.

Only the *read*, *write*, and *open* file operations are affected by the nonblocking flag.

A Sample Implementation: scullpipe

The */dev/scullpipe* devices (there are four of them by default) are part of the *scull* module and are used to show how blocking I/O is implemented.

Within a driver, a process blocked in a *read* call is awakened when data arrives; usually the hardware issues an interrupt to signal such an event, and the driver awakens waiting processes as part of handling the interrupt. The *scull* driver works differently, so that it can be run without requiring any particular hardware or an interrupt handler. We chose to use another process to generate the data and wake the reading process; similarly, reading processes are used to wake sleeping writer processes. The resulting implementation is similar to that of a FIFO (or named pipe) filesystem node, whence the name.

The device driver uses a device structure that embeds two wait queues and a buffer. The size of the buffer is configurable in the usual ways (at compile time, load time, or runtime).

```
typedef struct Scull_Pipe {
  wait_queue_head_t inq, outq;  /* read and write queues */
  char *buffer, *end;       /* begin of buf, end of buf */
  int buffersize;         /* used in pointer arithmetic */
  char *rp, *wp;         /* where to read, where to write */
  int nreaders, nwriters;     /* number of openings for r/w */
  struct fasync_struct *async_queue; /* asynchronous readers */
  struct semaphore sem;      /* mutual exclusion semaphore */
  devfs_handle_t handle;     /* only used if devfs is there */
} Scull_Pipe;
```

The *read* implementation manages both blocking and nonblocking input and looks like this (the puzzling first line of the function is explained later, in “Seeking a Device”):

```
ssize_t scull_p_read (struct file *filp, char *buf, size_t count,
        loff_t *f_pos)
{
  Scull_Pipe *dev = filp->private_data;

  if (f_pos != &filp->f_pos) return -ESPIPE;

  if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
  while (dev->rp == dev->wp) { /* nothing to read */
    up(&dev->sem); /* release the lock */
    if (filp->f_flags & O_NONBLOCK)
      return -EAGAIN;
    PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
    if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
      return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
    /* otherwise loop, but first reacquire the lock */
    if (down_interruptible(&dev->sem))
```

```
        return -ERESTARTSYS;
    }
    /* ok, data is there, return something */
    if (dev->wp > dev->rp)
      count = min(count, dev->wp - dev->rp);
    else /* the write pointer has wrapped, return data up to dev->end */
      count = min(count, dev->end - dev->rp);
    if (copy_to_user(buf, dev->rp, count)) {
      up (&dev->sem);
      return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp == dev->end)
      dev->rp = dev->buffer; /* wrapped */
    up (&dev->sem);

    /* finally, awaken any writers and return */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
    return count;
  }
```

As you can see, we left some `PDEBUG` statements in the code. When you compile the driver, you can enable messaging to make it easier to follow the interaction of different processes.

Note also, once again, the use of semaphores to protect critical regions of the code. The *scull* code has to be careful to avoid going to sleep when it holds a semaphore—otherwise, writers would never be able to add data, and the whole thing would deadlock. This code uses *wait_event_interruptible* to wait for data if need be; it has to check for available data again after the wait, though. Somebody else could grab the data between when we wake up and when we get the semaphore back.

It's worth repeating that a process can go to sleep both when it calls *schedule*, either directly or indirectly, and when it copies data to or from user space. In the latter case the process may sleep if the user array is not currently present in main memory. If *scull* sleeps while copying data between kernel and user space, it will sleep with the device semaphore held. Holding the semaphore in this case is justified since it will not deadlock the system, and since it is important that the device memory array not change while the driver sleeps.

The `if` statement that follows *interruptible_sleep_on* takes care of signal handling. This statement ensures the proper and expected reaction to signals, which could have been responsible for waking up the process (since we were in an interruptible sleep). If a signal has arrived and it has not been blocked by the process, the proper behavior is to let upper layers of the kernel handle the event. To this aim, the driver returns `-ERESTARTSYS` to the caller; this value is used internally by the virtual filesystem (VFS) layer, which either restarts the system call or returns `-EINTR` to user space. We'll use the same statement to deal with signal handling for every *read* and *write* implementation. Because *signal_pending* was introduced only in version 2.1.57 of the kernel, `sysdep.h` defines it for earlier kernels to preserve portability of source code.

The implementation for *write* is quite similar to that for *read* (and, again, its first line will be explained later). Its only "peculiar" feature is that it never completely fills the buffer, always leaving a hole of at least one byte. Thus, when the buffer is empty, `wp` and `rp` are equal; when there is data there, they are always different.

```
static inline int spacefree(Scull_Pipe *dev)
{
  if (dev->rp == dev->wp)
    return dev->buffersize - 1;
  return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}

ssize_t scull_p_write(struct file *filp, const char *buf, size_t count,
        loff_t *f_pos)
{
  Scull_Pipe *dev = filp->private_data;

  if (f_pos != &filp->f_pos) return -ESPIPE;

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

  /* Make sure there's space to write */
  while (spacefree(dev) == 0) { /* full */
    up(&dev->sem);
    if (filp->f_flags & O_NONBLOCK)
      return -EAGAIN;
    PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
    if (wait_event_interruptible(dev->outq, spacefree(dev) > 0))
      return -ERESTARTSYS; /* signal: tell the fs layer to handle it */
    if (down_interruptible(&dev->sem))
      return -ERESTARTSYS;
  }
  /* ok, space is there, accept something */
  count = min(count, spacefree(dev));
  if (dev->wp >= dev->rp)
    count = min(count, dev->end - dev->wp); /* up to end-of-buffer */
  else /* the write pointer has wrapped, fill up to rp-1 */
    count = min(count, dev->rp - dev->wp - 1);
  PDEBUG("Going to accept %li bytes to %p from %p\n",
      (long)count, dev->wp, buf);
  if (copy_from_user(dev->wp, buf, count)) {
    up (&dev->sem);
    return -EFAULT;
  }
  dev->wp += count;
  if (dev->wp == dev->end)
    dev->wp = dev->buffer; /* wrapped */
  up(&dev->sem);

  /* finally, awaken any reader */
  wake_up_interruptible(&dev->inq); /* blocked in read() and select() */

  /* and signal asynchronous readers, explained later in Chapter 5 */
  if (dev->async_queue)
    kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
  PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
  return count;
}
```

The device, as we conceived it, doesn't implement blocking *open* and is simpler than a real FIFO. If you want to look at the real thing, you can find it in *fs/pipe.c*, in the kernel sources.

To test the blocking operation of the *scullpipe* device, you can run some programs on it, using input/output redirection as usual. Testing nonblocking activity is trickier, because the conventional programs don't perform nonblocking operations. The *misc-progs* source directory contains the following simple program, called *nbtest*, for testing nonblocking operations. All it does is copy its input to its output, using nonblocking I/O and delaying between retrials. The delay time is passed on the command line and is one second by default.

```
int main(int argc, char **argv)
{
  int delay=1, n, m=0;

  if (argc>1) delay=atoi(argv[1]);
  fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
  fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */

  while (1) {
    n=read(0, buffer, 4096);
    if (n>=0)
      m=write(1, buffer, n);
    if ((n<0 || m<0) && (errno != EAGAIN))
      break;
    sleep(delay);
  }
  perror( n<0 ? "stdin" : "stdout");
  exit(1);
}
```

poll and select

Applications that use nonblocking I/O often use the *poll* and *select* system calls as well. *poll* and *select* have essentially the same functionality: both allow a process to determine whether it can read from or write to one or more open files without blocking. They are thus often used in applications that must use multiple input or output streams without blocking on any one of them. The same functionality is offered by two separate functions because they were implemented in Unix almost at the same time by two different groups: *select* was introduced in BSD Unix, whereas *poll* was the System V solution.

Support for either system call requires support from the device driver to function. In version 2.0 of the kernel the device method was modeled on *select* (and no *poll* was available to user programs); from version 2.1.23 onward both were offered, and the device method was based on the newly introduced *poll* system call because *poll* offered more detailed control than *select*.

Implementations of the *poll* method, implementing both the *poll* and *select* system calls, have the following prototype:

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

The driver's method will be called whenever the user-space program performs a *poll* or *select* system call involving a file descriptor associated with the driver. The device method is in charge of these two steps:

1. Call *poll_wait* on one or more wait queues that could indicate a change in the poll status.
2. Return a bit mask describing operations that could be immediately performed without blocking.

Both of these operations are usually straightforward, and tend to look very similar from one driver to the next. They rely, however, on information that only the driver can provide, and thus must be implemented individually by each driver.

The `poll_table` structure, the second argument to the *poll* method, is used within the kernel to implement the *poll* and *select* calls; it is declared in `<linux/poll.h>`, which must be included by the driver source. Driver writers need know nothing about its internals and must use it as an opaque object; it is passed to the driver method so that every event queue that could wake up the process and change the status of the *poll* operation can be added to the `poll_table` structure by calling the function *poll_wait*:

```
void poll_wait (struct file *, wait_queue_head_t *, poll_table *);
```

The second task performed by the poll method is returning the bit mask describing which operations could be completed immediately; this is also straightforward. For example, if the device has data available, a *read* would complete without sleeping; the *poll* method should indicate this state of affairs. Several flags (defined in `<linux/poll.h>`) are used to indicate the possible operations:

`POLLIN`
This bit must be set if the device can be read without blocking.

`POLLRDNORM`
This bit must be set if "normal" data is available for reading. A readable device returns `(POLLIN | POLLRDNORM)`.

`POLLRDBAND`
This bit indicates that out-of-band data is available for reading from the device. It is currently used only in one place in the Linux kernel (the DECnet code) and is not generally applicable to device drivers.

`POLLPRI`
High-priority data (out-of-band) can be read without blocking. This bit causes *select* to report that an exception condition occurred on the file, because *select* reports out-of-band data as an exception condition.

`POLLHUP`
When a process reading this device sees end-of-file, the driver must set `POLLHUP` (hang-up). A process calling *select* will be told that the device is readable, as dictated by the *select* functionality.

`POLLERR`
An error condition has occurred on the device. When *poll* is invoked, the device is reported as both readable and writable, since both *read* and *write* will return an error code without blocking.

`POLLOUT`
This bit is set in the return value if the device can be written to without blocking.

`POLLWRNORM`
This bit has the same meaning as `POLLOUT`, and sometimes it actually is the same number. A writable device returns `(POLLOUT | POLLWRNORM)`.

`POLLWRBAND`
Like `POLLRDBAND`, this bit means that data with nonzero priority can be written to the device. Only the datagram implementation of *poll* uses this bit, since a datagram can transmit out of band data.

It's worth noting that `POLLRDBAND` and `POLLWRBAND` are meaningful only with file descriptors associated with sockets: device drivers won't normally use these flags.

The description of *poll* takes up a lot of space for something that is relatively simple to use in practice. Consider the *scullpipe* implementation of the *poll* method:

```
unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
  Scull_Pipe *dev = filp->private_data;
  unsigned int mask = 0;

  /*
   * The buffer is circular; it is considered full
   * if "wp" is right behind "rp". "left" is 0 if the
   * buffer is empty, and it is "1" if it is completely full.
   */
  int left = (dev->rp + dev->buffersize - dev->wp) % dev->buffersize;

  poll_wait(filp, &dev->inq, wait);
  poll_wait(filp, &dev->outq, wait);
  if (dev->rp != dev->wp) mask |= POLLIN | POLLRDNORM; /* readable */
  if (left != 1)     mask |= POLLOUT | POLLWRNORM; /* writable */

  return mask;
}
```

This code simply adds the two *scullpipe* wait queues to the `poll_table`, then sets the appropriate mask bits depending on whether data can be read or written.

The *poll* code as shown is missing end-of-file support. The *poll* method should return `POLLHUP` when the device is at the end of the file. If the caller used the *select* system call, the file will be reported as readable; in both cases the application will know that it can actually issue the *read* without waiting forever, and the *read* method will return 0 to signal end-of-file.

With real FIFOs, for example, the reader sees an end-of-file when all the writers close the file, whereas in *scullpipe* the reader never sees end-of-file. The behavior is different because a FIFO is intended to be a communication channel between two processes, while *scullpipe* is a trashcan where everyone can put data as long as there's at least one reader. Moreover, it makes no sense to reimplement what is already available in the kernel.

Implementing end-of-file in the same way as FIFOs do would mean checking `dev->nwriters`, both in *read* and in *poll*, and reporting end-of-file (as just described) if no process has the device opened for writing. Unfortunately, though, if a reader opened the *scullpipe* device before the writer, it would see end-of-file without having a chance to wait for data. The best way to fix this problem would be to implement blocking within *open*; this task is left as an exercise for the reader.

Interaction with read and write

The purpose of the *poll* and *select* calls is to determine in advance if an I/O operation will block. In that respect, they complement *read* and *write*. More important, *poll* and *select* are useful because they let the application wait simultaneously for several data streams, although we are not exploiting this feature in the *scull* examples.

A correct implementation of the three calls is essential to make applications work correctly. Though the following rules have more or less already been stated, we'll summarize them here.

Reading data from the device

- If there is data in the input buffer, the *read* call should return immediately, with no noticeable delay, even if less data is available than the application requested and the driver is sure the remaining data will arrive soon. You can always return less data than you're asked for if this is convenient for any reason (we did it in *scull*), provided you return at least one byte.
- If there is no data in the input buffer, by default *read* must block until at least one byte is there. If `O_NONBLOCK` is set, on the other hand, *read* returns immediately with a return value of `-EAGAIN` (although some old versions of System V return 0 in this case). In these cases *poll* must report that the device is unreadable until at least one byte arrives. As soon as there is some data in the buffer, we fall back to the previous case.
- If we are at end-of-file, *read* should return immediately with a return value of 0, independent of `O_NONBLOCK`. *poll* should report `POLLHUP` in this case.

Writing to the device

- If there is space in the output buffer, *write* should return without delay. It can accept less data than the call requested, but it must accept at least one byte. In this case, *poll* reports that the device is writable.
- If the output buffer is full, by default *write* blocks until some space is freed. If `O_NONBLOCK` is set, *write* returns immediately with a return value of `-EAGAIN` (older System V Unices returned 0). In these cases *poll* should report that the file is not writable. If, on the other hand, the device is not able to accept any more data, *write* returns `-ENOSPC` ("No space left on device"), independently of the setting of `O_NONBLOCK`.
- Never make a *write* call wait for data transmission before returning, even if `O_NONBLOCK` is clear. This is because many applications use *select* to find out whether a *write* will block. If the device is reported as writable, the call must

consistently not block. If the program using the device wants to ensure that the data it enqueues in the output buffer is actually transmitted, the driver must provide an *fsync* method. For instance, a removable device should have an *fsync* entry point.

Although these are a good set of general rules, one should also recognize that each device is unique and that sometimes the rules must be bent slightly. For example, record-oriented devices (such as tape drives) cannot execute partial writes.

Flushing pending output

We've seen how the *write* method by itself doesn't account for all data output needs. The *fsync* function, invoked by the system call of the same name, fills the gap. This method's prototype is

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

If some application will ever need to be assured that data has been sent to the device, the *fsync* method must be implemented. A call to *fsync* should return only when the device has been completely flushed (i.e., the output buffer is empty), even if that takes some time, regardless of whether `O_NONBLOCK` is set. The `datasync` argument, present only in the 2.4 kernel, is used to distinguish between the *fsync* and *fdatasync* system calls; as such, it is only of interest to filesystem code and can be ignored by drivers.

The *fsync* method has no unusual features. The call isn't time critical, so every device driver can implement it to the author's taste. Most of the time, char drivers just have a `NULL` pointer in their `fops`. Block devices, on the other hand, always implement the method with the general-purpose *block_fsync*, which in turn flushes all the blocks of the device, waiting for I/O to complete.

The Underlying Data Structure

The actual implementation of the *poll* and *select* system calls is reasonably simple, for those who are interested in how it works. Whenever a user application calls either function, the kernel invokes the *poll* method of all files referenced by the system call, passing the same `poll_table` to each of them. The structure is, for all practical purposes, an array of `poll_table_entry` structures allocated for a specific *poll* or *select* call. Each `poll_table_entry` contains the `struct file` pointer for the open device, a `wait_queue_head_t` pointer, and a `wait_queue_t` entry. When a driver calls *poll_wait*, one of these entries gets filled in with the information provided by the driver, and the wait queue entry gets put onto the driver's queue. The pointer to `wait_queue_head_t` is used to track the wait queue where the current poll table entry is registered, in order for *free_wait* to be able to dequeue the entry before the wait queue is awakened.

If none of the drivers being polled indicates that I/O can occur without blocking, the *poll* call simply sleeps until one of the (perhaps many) wait queues it is on wakes it up.

What's interesting in the implementation of *poll* is that the file operation may be called with a `NULL` pointer as `poll_table` argument. This situation can come about for a couple of reasons. If the application calling *poll* has provided a timeout value of 0 (indicating that no wait should be done), there is no reason to accumulate wait queues, and the system simply does not do it. The `poll_table` pointer is also set to `NULL` immediately after any driver being *poll*ed indicates that I/O is possible. Since the kernel knows at that point that no wait will occur, it does not build up a list of wait queues.

When the *poll* call completes, the `poll_table` structure is deallocated, and all wait queue entries previously added to the poll table (if any) are removed from the table and their wait queues.

Actually, things are somewhat more complex than depicted here, because the poll table is not a simple array but rather a set of one or more pages, each hosting an array. This complication is meant to avoid putting too low a limit (dictated by the page size) on the maximum number of file descriptors involved in a *poll* or *select* system call.

We tried to show the data structures involved in polling in Figure 5-2; the figure is a simplified representation of the real data structures because it ignores the multipage nature of a poll table and disregards the file pointer that is part of each `poll_table_entry`. The reader interested in the actual implementation is urged to look in `<linux/poll.h>` and *fs/select.c*.

Asynchronous Notification

Though the combination of blocking and nonblocking operations and the *select* method are sufficient for querying the device most of the time, some situations aren't efficiently managed by the techniques we've seen so far.

Let's imagine, for example, a process that executes a long computational loop at low priority, but needs to process incoming data as soon as possible. If the input channel is the keyboard, you are allowed to send a signal to the application (using the 'INTR' character, usually CTRL-C), but this signaling ability is part of the tty abstraction, a software layer that isn't used for general char devices. What we need for asynchronous notification is something different. Furthermore, *any* input data should generate an interrupt, not just CTRL-C.

User programs have to execute two steps to enable asynchronous notification from an input file. First, they specify a process as the "owner" of the file. When a process invokes the `F_SETOWN` command using the *fcntl* system call, the process ID of the owner process is saved in `filp->f_owner` for later use. This step is necessary for the kernel to know just who to notify. In order to actually enable

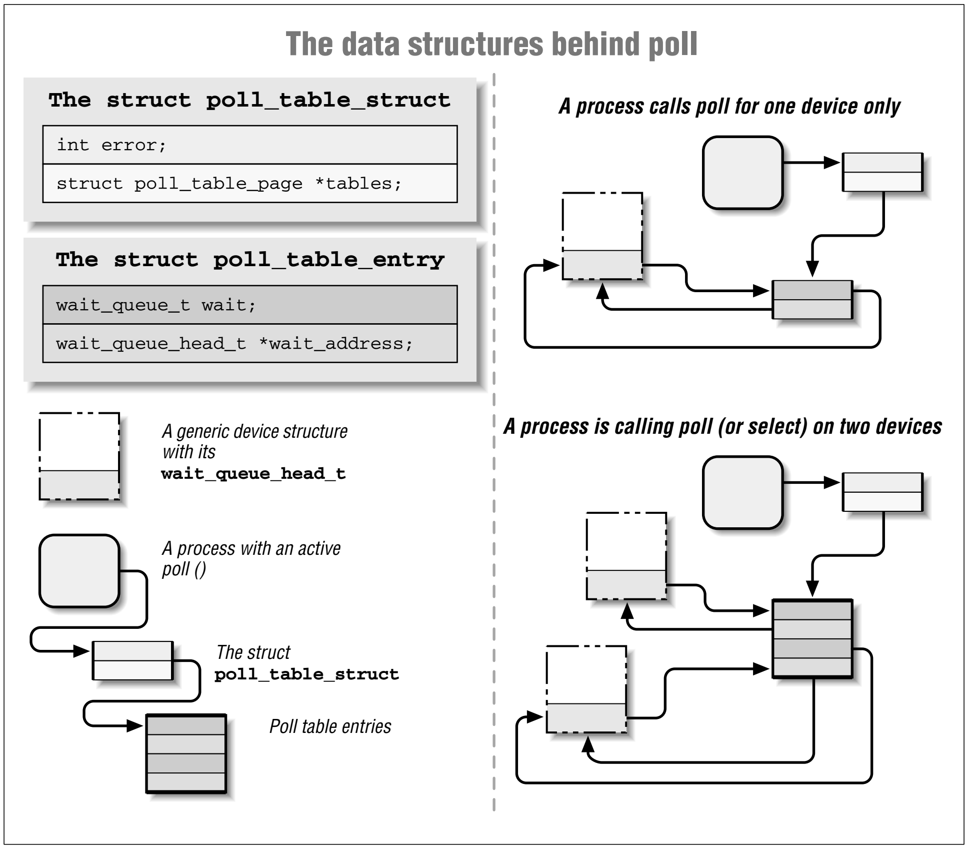
The data structures behind poll
The struct poll_table_struct
int error;
struct poll_table_page *tables;
The struct poll_table_entry
wait_queue_t wait;
wait_queue_head_t *wait_address;
A generic device structure with its wait_queue_head_t
A process with an active poll ()
The struct poll_table_struct
Poll table entries
A process calls poll for one device only
A process is calling poll (or select) on two devices

Figure 5-2. The data structures of poll

asynchronous notification, the user programs must set the `FASYNC` flag in the device by means of the `F_SETFL` *fcntl* command.

After these two calls have been executed, the input file can request delivery of a `SIGIO` signal whenever new data arrives. The signal is sent to the process (or process group, if the value is negative) stored in `filp->f_owner`.

For example, the following lines of code in a user program enable asynchronous notification to the current process for the `stdin` input file:

```
signal(SIGIO, &input_handler); /* dummy sample; sigaction() is better */
fcntl(STDIN_FILENO, F_SETOWN, getpid());
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

The program named *asynctest* in the sources is a simple program that reads `stdin` as shown. It can be used to test the asynchronous capabilities of *scullpipe*. The program is similar to *cat*, but doesn't terminate on end-of-file; it responds only to input, not to the absence of input.

Note, however, that not all the devices support asynchronous notification, and you can choose not to offer it. Applications usually assume that the asynchronous capability is available only for sockets and ttys. For example, pipes and FIFOs don't support it, at least in the current kernels. Mice offer asynchronous notification because some programs expect a mouse to be able to send `SIGIO` like a tty does.

There is one remaining problem with input notification. When a process receives a `SIGIO`, it doesn't know which input file has new input to offer. If more than one file is enabled to asynchronously notify the process of pending input, the application must still resort to *poll* or *select* to find out what happened.

The Driver's Point of View

A more relevant topic for us is how the device driver can implement asynchronous signaling. The following list details the sequence of operations from the kernel's point of view:

1. When `F_SETOWN` is invoked, nothing happens, except that a value is assigned to `filp->f_owner`.
2. When `F_SETFL` is executed to turn on `FASYNC`, the driver's *fasync* method is called. This method is called whenever the value of `FASYNC` is changed in `filp->f_flags`, to notify the driver of the change so it can respond properly. The flag is cleared by default when the file is opened. We'll look at the standard implementation of the driver method soon.
3. When data arrives, all the processes registered for asynchronous notification must be sent a `SIGIO` signal.

While implementing the first step is trivial—there's nothing to do on the driver's part—the other steps involve maintaining a dynamic data structure to keep track of the different asynchronous readers; there might be several of these readers. This dynamic data structure, however, doesn't depend on the particular device involved, and the kernel offers a suitable general-purpose implementation so that you don't have to rewrite the same code in every driver.

The general implementation offered by Linux is based on one data structure and two functions (which are called in the second and third steps described earlier). The header that declares related material is `<linux/fs.h>`—nothing new here—and the data structure is called `struct fasync_struct`. As we did with wait queues, we need to insert a pointer to the structure in the device-specific data structure. Actually, we've already seen such a field in the section "A Sample Implementation: scullpipe."

The two functions that the driver calls correspond to the following prototypes:

```
int fasync_helper(int fd, struct file *filp,
       int mode, struct fasync_struct **fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

`fasync_helper` is invoked to add files to or remove files from the list of interested processes when the `FASYNC` flag changes for an open file. All of its arguments except the last are provided to the *fasync* method and can be passed through directly. `kill_fasync` is used to signal the interested processes when data arrives. Its arguments are the signal to send (usually `SIGIO`) and the band, which is almost always `POLL_IN` (but which may be used to send "urgent" or out-of-band data in the networking code).

Here's how *scullpipe* implements the *fasync* method:

```
int scull_p_fasync(fasync_file fd, struct file *filp, int mode)
{
  Scull_Pipe *dev = filp->private_data;

  return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

It's clear that all the work is performed by *fasync_helper*. It wouldn't be possible, however, to implement the functionality without a method in the driver, because the helper function needs to access the correct pointer to `struct fasync_struct *` (here `&dev->async_queue`), and only the driver can provide the information.

When data arrives, then, the following statement must be executed to signal asynchronous readers. Since new data for the *scullpipe* reader is generated by a process issuing a *write*, the statement appears in the *write* method of *scullpipe*.

```
if (dev->async_queue)
  kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

It might appear that we're done, but there's still one thing missing. We must invoke our *fasync* method when the file is closed to remove the file from the list of active asynchronous readers. Although this call is required only if `filp->f_flags` has `FASYNC` set, calling the function anyway doesn't hurt and is the usual implementation. The following lines, for example, are part of the *close* method for *scullpipe*:

```
/* remove this filp from the asynchronously notified filp's */
scull_p_fasync(-1, filp, 0);
```

The data structure underlying asynchronous notification is almost identical to the structure `struct wait_queue`, because both situations involve waiting on an event. The difference is that `struct file` is used in place of `struct task_struct`. The `struct file` in the queue is then used to retrieve `f_owner`, in order to signal the process.

Seeking a Device

The difficult part of the chapter is over; now we'll quickly detail the *llseek* method, which is useful and easy to implement.

The llseek Implementation

The *llseek* method implements the *lseek* and *llseek* system calls. We have already stated that if the *llseek* method is missing from the device's operations, the default implementation in the kernel performs seeks from the beginning of the file and from the current position by modifying `filp->f_pos`, the current reading/writing position within the file. Please note that for the *lseek* system call to work correctly, the *read* and *write* methods must cooperate by updating the offset item they receive as argument (the argument is usually a pointer to `filp->f_pos`).

You may need to provide your own *llseek* method if the seek operation corresponds to a physical operation on the device or if seeking from end-of-file, which is not implemented by the default method, makes sense. A simple example can be seen in the *scull* driver:

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
  Scull_Dev *dev = filp->private_data;
  loff_t newpos;

  switch(whence) {
   case 0: /* SEEK_SET */
    newpos = off;
    break;

   case 1: /* SEEK_CUR */
    newpos = filp->f_pos + off;
    break;

   case 2: /* SEEK_END */
    newpos = dev->size + off;
    break;

   default: /* can't happen */
    return -EINVAL;
  }
  if (newpos<0) return -EINVAL;
  filp->f_pos = newpos;
  return newpos;
}
```

The only device-specific operation here is retrieving the file length from the device. In *scull* the *read* and *write* methods cooperate as needed, as shown in "read and write" in Chapter 3.

Although the implementation just shown makes sense for *scull*, which handles a well-defined data area, most devices offer a data flow rather than a data area (just think about the serial ports or the keyboard), and seeking those devices does not make sense. If this is the case, you can't just refrain from declaring the *llseek* operation, because the default method allows seeking. Instead, you should use the following code:

```
loff_t scull_p_llseek(struct file *filp, loff_t off, int whence)
{
  return -ESPIPE; /* unseekable */
}
```

This function comes from the *scullpipe* device, which isn't seekable; the error code is translated to "Illegal seek," though the symbolic name means "is a pipe." Because the position indicator is meaningless for nonseekable devices, neither *read* nor *write* needs to update it during data transfer.

It's interesting to note that since *pread* and *pwrite* have been added to the set of supported system calls, the *lseek* device method is not the only way a user-space program can seek a file. A proper implementation of unseekable devices should allow normal *read* and *write* calls while preventing *pread* and *pwrite*. This is accomplished by the following line—the first in both the *read* and *write* methods of *scullpipe*—we didn't explain when introducing those methods:

```
 if (f_pos != &filp->f_pos) return -ESPIPE;
```

Access Control on a Device File

Offering access control is sometimes vital for the reliability of a device node. Not only should unauthorized users not be permitted to use the device (a restriction is enforced by the filesystem permission bits), but sometimes only one authorized user should be allowed to open the device at a time.

The problem is similar to that of using ttys. In that case, the *login* process changes the ownership of the device node whenever a user logs into the system, in order to prevent other users from interfering with or sniffing the tty data flow. However, it's impractical to use a privileged program to change the ownership of a device every time it is opened, just to grant unique access to it.

None of the code shown up to now implements any access control beyond the filesystem permission bits. If the *open* system call forwards the request to the driver, *open* will succeed. We now introduce a few techniques for implementing some additional checks.

Every device shown in this section has the same behavior as the bare *scull* device (that is, it implements a persistent memory area) but differs from *scull* in access control, which is implemented in the *open* and *close* operations.

Single-Open Devices

The brute-force way to provide access control is to permit a device to be opened by only one process at a time (single openness). This technique is best avoided because it inhibits user ingenuity. A user might well want to run different processes on the same device, one reading status information while the other is writing data. In some cases, users can get a lot done by running a few simple programs through a shell script, as long as they can access the device concurrently. In other words, implementing a single-open behavior amounts to creating policy, which may get in the way of what your users want to do.

Allowing only a single process to open a device has undesirable properties, but it is also the easiest access control to implement for a device driver, so it's shown here. The source code is extracted from a device called *scullsingle*.

The *open* call refuses access based on a global integer flag:

```
int scull_s_open(struct inode *inode, struct file *filp)
{
  Scull_Dev *dev = &scull_s_device; /* device information */
  int num = NUM(inode->i_rdev);

  if (!filp->private_data && num > 0)
    return -ENODEV; /* not devfs: allow 1 device only */
  spin_lock(&scull_s_lock);
  if (scull_s_count) {
    spin_unlock(&scull_s_lock);
    return -EBUSY; /* already open */
  }
  scull_s_count++;
  spin_unlock(&scull_s_lock);
  /* then, everything else is copied from the bare scull device */

  if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
    scull_trim(dev);
  if (!filp->private_data)
    filp->private_data = dev;
  MOD_INC_USE_COUNT;
  return 0;     /* success */
}
```

The *close* call, on the other hand, marks the device as no longer busy.

```
int scull_s_release(struct inode *inode, struct file *filp)
{
  scull_s_count--; /* release the device */
  MOD_DEC_USE_COUNT;
  return 0;
}
```

Normally, we recommend that you put the open flag `scull_s_count` (with the accompanying spinlock, `scull_s_lock`, whose role is explained in the next subsection) within the device structure (`Scull_Dev` here) because, conceptually, it belongs to the device. The *scull* driver, however, uses standalone variables to hold the flag and the lock in order to use the same device structure and methods as the bare *scull* device and minimize code duplication.

Another Digression into Race Conditions

Consider once again the test on the variable `scull_s_count` just shown. Two separate actions are taken there: (1) the value of the variable is tested, and the open is refused if it is not 0, and (2) the variable is incremented to mark the device as taken. On a single-processor system, these tests are safe because no other process will be able to run between the two actions.

As soon as you get into the SMP world, however, a problem arises. If two processes on two processors attempt to open the device simultaneously, it is possible that they could both test the value of `scull_s_count` before either modifies it. In this scenario you'll find that, at best, the single-open semantics of the device is not enforced. In the worst case, unexpected concurrent access could create data structure corruption and system crashes.

In other words, we have another race condition here. This one could be solved in much the same way as the races we already saw in Chapter 3. Those race conditions were triggered by access to a status variable of a potentially shared data structure and were solved using semaphores. In general, however, semaphores can be expensive to use, because they can put the calling process to sleep. They are a heavyweight solution for the problem of protecting a quick check on a status variable.

Instead, *scullsingle* uses a different locking mechanism called a *spinlock*. Spinlocks will never put a process to sleep. Instead, if a lock is not available, the spinlock primitives will simply retry, over and over (i.e., "spin"), until the lock is freed. Spinlocks thus have very little locking overhead, but they also have the potential to cause a processor to spin for a long time if somebody hogs the lock. Another advantage of spinlocks over semaphores is that their implementation is empty when compiling code for a uniprocessor system (where these SMP-specific races can't happen). Semaphores are a more general resource that make sense on uniprocessor computers as well as SMP, so they don't get optimized away in the uniprocessor case.

Spinlocks can be the ideal mechanism for small critical sections. Processes should hold spinlocks for the minimum time possible, and must never sleep while holding a lock. Thus, the main *scull* driver, which exchanges data with user space and can therefore sleep, is not suitable for a spinlock solution. But spinlocks work nicely for controlling access to `scull_s_single` (even if they still are not the optimal solution, which we will see in Chapter 9).

Spinlocks are declared with a type of `spinlock_t`, which is defined in `<linux/spinlock.h>`. Prior to use, they must be initialized:

```
spin_lock_init(spinlock_t *lock);
```

A process entering a critical section will obtain the lock with `spin_lock`:

```
spin_lock(spinlock_t *lock);
```

The lock is released at the end with `spin_unlock`:

```
spin_unlock(spinlock_t *lock);
```

Spinlocks can be more complicated than this, and we'll get into the details in Chapter 9. But the simple case as shown here suits our needs for now, and all of the access-control variants of *scull* will use simple spinlocks in this manner.

The astute reader may have noticed that whereas *scull_s_open* acquires the `scull_s_lock` lock prior to incrementing the `scull_s_count` flag, *scull_s_close* takes no such precautions. This code is safe because no other code will change the value of `scull_s_count` if it is nonzero, so there will be no conflict with this particular assignment.

Restricting Access to a Single User at a Time

The next step beyond a single system-wide lock is to let a single user open a device in multiple processes but allow only one user to have the device open at a time. This solution makes it easy to test the device, since the user can read and write from several processes at once, but assumes that the user takes some responsibility for maintaining the integrity of the data during multiple accesses. This is accomplished by adding checks in the *open* method; such checks are performed *after* the normal permission checking and can only make access more restrictive than that specified by the owner and group permission bits. This is the same access policy as that used for ttys, but it doesn't resort to an external privileged program.

Those access policies are a little trickier to implement than single-open policies. In this case, two items are needed: an open count and the uid of the "owner" of the device. Once again, the best place for such items is within the device structure; our example uses global variables instead, for the reason explained earlier for *scullsingle*. The name of the device is *sculluid*.

The *open* call grants access on first open, but remembers the owner of the device. This means that a user can open the device multiple times, thus allowing cooperating processes to work concurrently on the device. At the same time, no other user can open it, thus avoiding external interference. Since this version of the function is almost identical to the preceding one, only the relevant part is reproduced here:

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
  (scull_u_owner != current->uid) && /* allow user */
  (scull_u_owner != current->euid) && /* allow whoever did su */
```

```
        !capable(CAP_DAC_OVERRIDE)) { /* still allow root */
    spin_unlock(&scull_u_lock);
    return -EBUSY;  /* -EPERM would confuse the user */
}

if (scull_u_count == 0)
  scull_u_owner = current->uid; /* grab it */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

We chose to return `-EBUSY` and not `-EPERM`, even though the code is performing a permission check, in order to point a user who is denied access in the right direction. The reaction to "Permission denied" is usually to check the mode and owner of the */dev* file, while "Device busy" correctly suggests that the user should look for a process already using the device.

This code also checks to see if the process attempting the open has the ability to override file access permissions; if so, the open will be allowed even if the opening process is not the owner of the device. The `CAP_DAC_OVERRIDE` capability fits the task well in this case.

The code for *close* is not shown, since all it does is decrement the usage count.

Blocking open as an Alternative to EBUSY

When the device isn't accessible, returning an error is usually the most sensible approach, but there are situations in which you'd prefer to wait for the device.

For example, if a data communication channel is used both to transmit reports on a timely basis (using *crontab*) and for casual usage according to people's needs, it's much better for the timely report to be slightly delayed rather than fail just because the channel is currently busy.

This is one of the choices that the programmer must make when designing a device driver, and the right answer depends on the particular problem being solved.

The alternative to `EBUSY`, as you may have guessed, is to implement blocking *open*.

The *scullwuid* device is a version of *sculluid* that waits for the device on *open* instead of returning `-EBUSY`. It differs from *sculluid* only in the following part of the *open* operation:

```
spin_lock(&scull_w_lock);
while (scull_w_count &&
 (scull_w_owner != current->uid) && /* allow user */
 (scull_w_owner != current->euid) && /* allow whoever did su */
 !capable(CAP_DAC_OVERRIDE)) {
  spin_unlock(&scull_w_lock);
```

```
      if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
      interruptible_sleep_on(&scull_w_wait);
      if (signal_pending(current)) /* a signal arrived */
       return -ERESTARTSYS; /* tell the fs layer to handle it */
      /* else, loop */
      spin_lock(&scull_w_lock);
    }
    if (scull_w_count == 0)
      scull_w_owner = current->uid; /* grab it */
    scull_w_count++;
    spin_unlock(&scull_w_lock);
```

The implementation is based once again on a wait queue. Wait queues were created to maintain a list of processes that sleep while waiting for an event, so they fit perfectly here.

The *release* method, then, is in charge of awakening any pending process:

```
int scull_w_release(struct inode *inode, struct file *filp)
{
  scull_w_count--;
  if (scull_w_count == 0)
    wake_up_interruptible(&scull_w_wait); /* awaken other uid's */
  MOD_DEC_USE_COUNT;
  return 0;
}
```

The problem with a blocking-open implementation is that it is really unpleasant for the interactive user, who has to keep guessing what is going wrong. The interactive user usually invokes precompiled commands such as *cp* and *tar* and can't just add `O_NONBLOCK` to the *open* call. Someone who's making a backup using the tape drive in the next room would prefer to get a plain "device or resource busy" message instead of being left to guess why the hard drive is so silent today while *tar* is scanning it.

This kind of problem (different, incompatible policies for the same device) is best solved by implementing one device node for each access policy. An example of this practice can be found in the Linux tape driver, which provides multiple device files for the same device. Different device files will, for example, cause the drive to record with or without compression, or to automatically rewind the tape when the device is closed.

Cloning the Device on Open

Another technique to manage access control is creating different private copies of the device depending on the process opening it.

Clearly this is possible only if the device is not bound to a hardware object; *scull* is an example of such a "software" device. The internals of */dev/tty* use a similar technique in order to give its process a different "view" of what the */dev* entry point represents. When copies of the device are created by the software driver, we call them *virtual devices*—just as virtual consoles use a single physical tty device.

Although this kind of access control is rarely needed, the implementation can be enlightening in showing how easily kernel code can change the application's perspective of the surrounding world (i.e., the computer). The topic is quite exotic, actually, so if you aren't interested, you can jump directly to the next section.

The */dev/scullpriv* device node implements virtual devices within the *scull* package. The *scullpriv* implementation uses the minor number of the process's controlling tty as a key to access the virtual device. You can nonetheless easily modify the sources to use any integer value for the key; each choice leads to a different policy. For example, using the `uid` leads to a different virtual device for each user, while using a `pid` key creates a new device for each process accessing it.

The decision to use the controlling terminal is meant to enable easy testing of the device using input/output redirection: the device is shared by all commands run on the same virtual terminal and is kept separate from the one seen by commands run on another terminal.

The *open* method looks like the following code. It must look for the right virtual device and possibly create one. The final part of the function is not shown because it is copied from the bare *scull*, which we've already seen.

```
/* The clone-specific data structure includes a key field */
struct scull_listitem {
  Scull_Dev device;
  int key;
  struct scull_listitem *next;

};

/* The list of devices, and a lock to protect it */
struct scull_listitem *scull_c_head;
spinlock_t scull_c_lock;

/* Look for a device or create one if missing */
static Scull_Dev *scull_c_lookfor_device(int key)
{
  struct scull_listitem *lptr, *prev = NULL;

  for (lptr = scull_c_head; lptr && (lptr->key != key); lptr = lptr->next)
    prev=lptr;
  if (lptr) return &(lptr->device);

  /* not found */
  lptr = kmalloc(sizeof(struct scull_listitem), GFP_ATOMIC);
  if (!lptr) return NULL;
  /* initialize the device */
  memset(lptr, 0, sizeof(struct scull_listitem));
  lptr->key = key;
  scull_trim(&(lptr->device)); /* initialize it */
  sema_init(&(lptr->device.sem), 1);

  /* place it in the list */
  if (prev) prev->next = lptr;
  else    scull_c_head = lptr;

  return &(lptr->device);
}

int scull_c_open(struct inode *inode, struct file *filp)
{
  Scull_Dev *dev;
  int key, num = NUM(inode->i_rdev);

  if (!filp->private_data && num > 0)
    return -ENODEV; /* not devfs: allow 1 device only */

  if (!current->tty) {
    PDEBUG("Process \"%s\" has no ctl tty\n",current->comm);
    return -EINVAL;
  }
  key = MINOR(current->tty->device);

  /* look for a scullc device in the list */
  spin_lock(&scull_c_lock);
  dev = scull_c_lookfor_device(key);
  spin_unlock(&scull_c_lock);

  if (!dev) return -ENOMEM;

  /* then, everything else is copied from the bare scull device */
```

The *release* method does nothing special. It would normally release the device on last close, but we chose not to maintain an open count in order to simplify the testing of the driver. If the device were released on last close, you wouldn't be able to read the same data after writing to the device unless a background process were to keep it open. The sample driver takes the easier approach of keeping the data, so that at the next *open*, you'll find it there. The devices are released when *scull_cleanup* is called.

Here's the *release* implementation for */dev/scullpriv*, which closes the discussion of device methods.

```
int scull_c_release(struct inode *inode, struct file *filp)
{
  /*
   * Nothing to do, because the device is persistent.
   * A 'real' cloned device should be freed on last close
```

```
     */
    MOD_DEC_USE_COUNT;
    return 0;
}
```

Backward Compatibility

Many parts of the device driver API covered in this chapter have changed between the major kernel releases. For those of you needing to make your driver work with Linux 2.0 or 2.2, here is a quick rundown of the differences you will encounter.

Wait Queues in Linux 2.2 and 2.0

A relatively small amount of the material in this chapter changed in the 2.3 development cycle. The one significant change is in the area of wait queues. The 2.2 kernel had a different and simpler implementation of wait queues, but it lacked some important features, such as exclusive sleeps. The new implementation of wait queues was introduced in kernel version 2.3.1.

The 2.2 wait queue implementation used variables of the type `struct wait_queue *` instead of `wait_queue_head_t`. This pointer had to be initialized to `NULL` prior to its first use. A typical declaration and initialization of a wait queue looked like this:

```
 struct wait_queue *my_queue = NULL;
```

The various functions for sleeping and waking up looked the same, with the exception of the variable type for the queue itself. As a result, writing code that works for all 2.*x* kernels is easily done with a bit of code like the following, which is part of the `sysdep.h` header we use to compile our sample code.

```
# define DECLARE_WAIT_QUEUE_HEAD(head) struct wait_queue *head = NULL
  typedef struct wait_queue *wait_queue_head_t;
# define init_waitqueue_head(head) (*(head)) = NULL
```

The synchronous versions of *wake_up* were added in 2.3.29, and `sysdep.h` provides macros with the same names so that you can use the feature in your code while maintaining portability. The replacement macros expand to normal *wake_up*, since the underlying mechanisms were missing from earlier kernels. The timeout versions of *sleep_on* were added in kernel 2.1.127. The rest of the wait queue interface has remained relatively unchanged. The *sysdep.h* header defines the needed macros in order to compile and run your modules with Linux 2.2 and Linux 2.0 without cluttering the code with lots of `#ifdef`s.

The *wait_event* macro did not exist in the 2.0 kernel. For those who need it, we have provided an implementation in *sysdep.h*

Asynchronous Notification

Some small changes have been made in how asynchronous notification works for both the 2.2 and 2.4 releases.

In Linux 2.3.21, *kill_fasync* got its third argument. Prior to this release, *kill_fasync* was called as

```
kill_fasync(struct fasync_struct *queue, int signal);
```

Fortunately, *sysdep.h* takes care of the issue.

In the 2.2 release, the type of the first argument to the *fasync* method changed. In the 2.0 kernel, a pointer to the `inode` structure for the device was passed, instead of the integer file descriptor:

```
int (*fasync) (struct inode *inode, struct file *filp, int on);
```

To solve this incompatibility, we use the same approach taken for *read* and *write*: use of a wrapper function when the module is compiled under 2.0 headers.

The `inode` argument to the *fasync* method was also passed in when called from the *release* method, rather than the `-1` value used with later kernels.

The fsync Method

The third argument to the *fsync* `file_operations` method (the integer `datasync` value) was added in the 2.3 development series, meaning that portable code will generally need to include a wrapper function for older kernels. There is a trap, however, for people trying to write portable *fsync* methods: at least one distributor, which will remain nameless, patched the 2.4 *fsync* API into its 2.2 kernel. The kernel developers usually (*usually . . .*) try to avoid making API changes within a stable series, but they have little control over what the distributors do.

Access to User Space in Linux 2.0

Memory access was handled differently in the 2.0 kernels. The Linux virtual memory system was less well developed at that time, and memory access was handled a little differently. The new system was the key change that opened 2.1 development, and it brought significant improvements in performance; unfortunately, it was accompanied by yet another set of compatibility headaches for driver writers.

The functions used to access memory under Linux 2.0 were as follows:

`verify_area(int mode, const void *ptr, unsigned long size);`
: This function worked similarly to *access_ok*, but performed more extensive checking and was slower. The function returned 0 in case of success and

`-EFAULT` in case of errors. Recent kernel headers still define the function, but it's now just a wrapper around *access_ok*. When using version 2.0 of the kernel, calling *verify_area* is never optional; no access to user space can safely be performed without a prior, explicit verification.

`put_user(datum, ptr)`
: The *put_user* macro looks much like its modern-day equivalent. It differed, however, in that no verification was done, and there was no return value.

`get_user(ptr)`
: This macro fetched the value at the given address, and returned it as its return value. Once again, no verification was done by the execution of the macro.

verify_area had to be called explicitly because no user-area copy function performed the check. The great news introduced by Linux 2.1, which forced the incompatible change in the *get_user* and *put_user* functions, was that the task of verifying user addresses was left to the hardware, because the kernel was now able to trap and handle processor exceptions generated during data copies to user space.

As an example of how the older calls are used, consider *scull* one more time. A version of *scull* using the 2.0 API would call *verify_area* in this way:

```
int err = 0, tmp;

/*
 * extract the type and number bitfields, and don't decode
 * wrong cmds: return ENOTTY before verify_area()
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * the direction is a bit mask, and VERIFY_WRITE catches R/W
 * transfers. 'Type' is user oriented, while
 * verify_area is kernel oriented, so the concept of "read" and
 * "write" is reversed
 */
if (_IOC_DIR(cmd) & _IOC_READ)
  err = verify_area(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
  err = verify_area(VERIFY_READ, (void *)arg, _IOC_SIZE(cmd));
if (err) return err;
```

Then *get_user* and *put_user* can be used as follows:

```
case SCULL_IOCXQUANTUM: /* eXchange: use arg as pointer */
 tmp = scull_quantum;
 scull_quantum = get_user((int *)arg);
 put_user(tmp, (int *)arg);
 break;
```

```
default: /* redundant, as cmd was checked against MAXNR */
 return -ENOTTY;
}
 return 0;
```

Only a small portion of the *ioctl* switch code has been shown, since it is little different from the version for 2.2 and beyond.

Life would be relatively easy for the compatibility-conscious driver writer if it weren't for the fact that *put_user* and *get_user* are implemented as macros in all Linux versions, and their interfaces changed. As a result, a straightforward fix using macros cannot be done.

One possible solution is to define a new set of version-independent macros. The path taken by *sysdep.h* consists in defining upper-case macros: `GET_USER`, `__GET_USER`, and so on. The arguments are the same as with the kernel macros of Linux 2.4, but the caller must be sure that *verify_area* has been called first (because that call is needed when compiling for 2.0).

Capabilities in 2.0

The 2.0 kernel did not support the capabilities abstraction at all. All permissions checks simply looked to see if the calling process was running as the superuser; if so, the operation would be allowed. The function *suser* was used for this purpose; it takes no arguments and returns a nonzero value if the process has superuser privileges.

suser still exists in later kernels, but its use is strongly discouraged. It is better to define a version of *capable* for 2.0, as is done in *sysdep.h*:

```
# define capable(anything) suser()
```

In this way, code can be written that is portable but which works with modern, capability-oriented systems.

The Linux 2.0 select Method

The 2.0 kernel did not support the *poll* system call; only the BSD-style *select* call was available. The corresponding device driver method was thus called *select*, and operated in a slightly different way, though the actions to be performed are almost identical.

The *select* method is passed a pointer to a `select_table`, and must pass that pointer to *select_wait* only if the calling process should wait for the requested condition (one of `SEL_IN`, `SEL_OUT`, or `SEL_EX`).

The *scull* driver deals with the incompatibility by declaring a specific *select* method to be used when it is compiled for version 2.0 of the kernel:

```
#ifdef __USE_OLD_SELECT__
int scull_p_poll(struct inode *inode, struct file *filp,
         int mode, select_table *table)
{
  Scull_Pipe *dev = filp->private_data;

  if (mode == SEL_IN) {
    if (dev->rp != dev->wp) return 1; /* readable */
    PDEBUG("Waiting to read\n");
    select_wait(&dev->inq, table); /* wait for data */
    return 0;
  }
  if (mode == SEL_OUT) {
    /*
     * The buffer is circular; it is considered full
     * if "wp" is right behind "rp". "left" is 0 if the
     * buffer is empty, and it is "1" if it is completely full.
     */
    int left = (dev->rp + dev->buffersize - dev->wp) % dev->buffersize;
    if (left != 1) return 1; /* writable */
    PDEBUG("Waiting to write\n");
    select_wait(&dev->outq, table); /* wait for free space */
    return 0;
  }
  return 0; /* never exception-able */
}
#else /* Use poll instead, already shown */
```

The `__USE_OLD_SELECT__` preprocessor symbol used here is set by the `sysdep.h` include file according to kernel version.

Seeking in Linux 2.0

Prior to Linux 2.1, the *llseek* device method was called *lseek* instead, and it received different parameters from the current implementation. For that reason, under Linux 2.0 you were not allowed to seek a file, or a device, past the 2 GB limit, even though the *llseek* system call was already supported.

The prototype of the file operation in the 2.0 kernel was the following:

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

Those working to write drivers compatible with 2.0 and 2.2 usually end up defining separate implementations of the seek method for the two interfaces.

2.0 and SMP

Because Linux 2.0 only minimally supported SMP systems, race conditions of the type mentioned in this chapter did not normally come about. The 2.0 kernel *did* have a spinlock implementation, but, since only one processor could be running kernel code at a time, there was less need for locking.

Quick Reference

This chapter introduced the following symbols and header files.

`#include <linux/ioctl.h>`
: This header declares all the macros used to define *ioctl* commands. It is currently included by `<linux/fs.h>`.

`_IOC_NRBITS`
`_IOC_TYPEBITS`
`_IOC_SIZEBITS`
`_IOC_DIRBITS`
: The number of bits available for the different bitfields of *ioctl* commands. There are also four macros that specify the `MASK`s and four that specify the `SHIFT`s, but they're mainly for internal use. `_IOC_SIZEBITS` is an important value to check, because it changes across architectures.

`_IOC_NONE`
`_IOC_READ`
`_IOC_WRITE`
: The possible values for the "direction" bitfield. "Read" and "write" are different bits and can be OR'd to specify read/write. The values are 0 based.

`_IOC(dir,type,nr,size)`
`_IO(type,nr)`
`_IOR(type,nr,size)`
`_IOW(type,nr,size)`
`_IOWR(type,nr,size)`
: Macros used to create an *ioctl* command.

`_IOC_DIR(nr)`
`_IOC_TYPE(nr)`
`_IOC_NR(nr)`
`_IOC_SIZE(nr)`
: Macros used to decode a command. In particular, `_IOC_TYPE(nr)` is an OR combination of `_IOC_READ` and `_IOC_WRITE`.

`#include <asm/uaccess.h>`
`int access_ok(int type, const void *addr, unsigned long size);`
: This function checks that a pointer to user space is actually usable. *access_ok* returns a nonzero value if the access should be allowed.

`VERIFY_READ`
`VERIFY_WRITE`
The possible values for the `type` argument in *access_ok*. `VERIFY_WRITE` is a superset of `VERIFY_READ`.

`#include <asm/uaccess.h>`
`int put_user(datum,ptr);`
`int get_user(local,ptr);`
`int __put_user(datum,ptr);`
`int __get_user(local,ptr);`
Macros used to store or retrieve a datum to or from user space. The number of bytes being transferred depends on `sizeof(*ptr)`. The regular versions call *access_ok* first, while the qualified versions (*__put_user* and *__get_user*) assume that *access_ok* has already been called.

`#include <linux/capability.h>`
Defines the various `CAP_` symbols for capabilities under Linux 2.2 and later.

`int capable(int capability);`
Returns nonzero if the process has the given capability.

`#include <linux/wait.h>`
`typedef struct { /* ... */ } wait_queue_head_t;`
`void init_waitqueue_head(wait_queue_head_t *queue);`
`DECLARE_WAIT_QUEUE_HEAD(queue);`
The defined type for Linux wait queues. A `wait_queue_head_t` must be explicitly initialized with either *init_waitqueue_head* at runtime or *declare_wait_queue_head* at compile time.

```
#include <linux/sched.h>
void interruptible_sleep_on(wait_queue_head_t *q);
void sleep_on(wait_queue_head_t *q);
void interruptible_sleep_on_timeout(wait_queue_head_t *q,
      long timeout);
void sleep_on_timeout(wait_queue_head_t *q, long timeout);
```

Calling any of these functions puts the current process to sleep on a queue. Usually, you'll choose the *interruptible* form to implement blocking *read* and *write*.

```
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
void wake_up_sync(struct wait_queue **q);
void wake_up_interruptible_sync(struct wait_queue **q);
```

These functions wake processes that are sleeping on the queue `q`. The *_interruptible* form wakes only interruptible processes. The *_sync* versions will not reschedule the CPU before returning.

`typedef struct { /* ... */ } wait_queue_t;`

`init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);`

The `wait_queue_t` type is used when sleeping without calling *sleep_on*. Wait queue entries must be initialized prior to use; the `task` argument used is almost always `current`.

`void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);`

`void add_wait_queue_exclusive(wait_queue_head_t *q, wait_queue_t *wait);`

`void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);`

These functions add an entry to a wait queue; *add_wait_queue_exclusive* adds the entry to the end of the queue for exclusive waits. Entries should be removed from the queue after sleeping with *remove_wait_queue*.

`void wait_event(wait_queue_head_t q, int condition);`

`int wait_event_interruptible(wait_queue_head_t q, int condition);`

These two macros will cause the process to sleep on the given queue until the given `condition` evaluates to a true value.

`void schedule(void);`

This function selects a runnable process from the run queue. The chosen process can be `current` or a different one. You won't usually call *schedule* directly, because the *sleep_on* functions do it internally.

`#include <linux/poll.h>`

`void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p)`

This function puts the current process into a wait queue without scheduling immediately. It is designed to be used by the *poll* method of device drivers.

`int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);`

This function is a "helper" for implementing the *fasync* device method. The `mode` argument is the same value that is passed to the method, while `fa` points to a device-specific `fasync_struct *`.

`void kill_fasync(struct fasync_struct *fa, int sig, int band);`

If the driver supports asynchronous notification, this function can be used to send a signal to processes registered in `fa`.

`#include <linux/spinlock.h>`
`typedef struct { /* ... */ } spinlock_t;`
`void spin_lock_init(spinlock_t *lock);`
The `spinlock_t` type defines a spinlock, which must be initialized (with *spin_lock_init*) prior to use.

`spin_lock(spinlock_t *lock);`
`spin_unlock(spinlock_t *lock);`
spin_lock locks the given lock, perhaps waiting until it becomes available. The lock can then be released with *spin_unlock*.