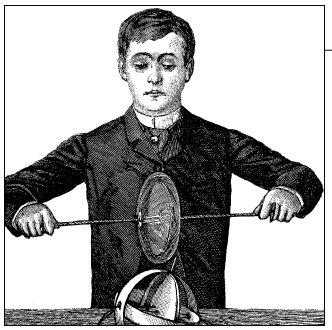

CHAPTER 4

Debugging Techniques

Kernel programming brings its own, unique debugging challenges. Kernel code cannot be easily executed under a debugger, nor can it be easily traced, because it is a set of functionalities not related to a specific process. Kernel code errors can also be exceedingly hard to reproduce and can bring down the entire system with them, thus destroying much of the evidence that could be used to track them down.

This chapter introduces techniques you can use to monitor kernel code and trace errors under such trying circumstances.

Debugging Support in the Kernel

In Chapter 2, we recommended that you build and install your own kernel, rather than running the stock kernel that comes with your distribution. One of the strongest reasons for running your own kernel is that the kernel developers have built several debugging features into the kernel itself. These features can create extra output and slow performance, so they tend not to be enabled in production kernels from distributors. As a kernel developer, however, you have different priorities and will gladly accept the (minimal) overhead of the extra kernel debugging support.

Here, we list the configuration options that should be enabled for kernels used for development. Except where specified otherwise, all of these options are found under the "kernel hacking" menu in whatever kernel configuration tool you prefer. Note that some of these options are not supported by all architectures.

`CONFIG_DEBUG_KERNEL`

This option just makes other debugging options available; it should be turned on but does not, by itself, enable any features.

`CONFIG_DEBUG_SLAB`

This crucial option turns on several types of checks in the kernel memory allocation functions; with these checks enabled, it is possible to detect a number of memory overrun and missing initialization errors. Each byte of allocated memory

is set to 0xa5 before being handed to the caller and then set to 0x6b when it is freed. If you ever see either of those "poison" patterns repeating in output from your driver (or often in an oops listing), you'll know exactly what sort of error to look for. When debugging is enabled, the kernel also places special guard values before and after every allocated memory object; if those values ever get changed, the kernel knows that somebody has overrun a memory allocation, and it complains loudly. Various checks for more obscure errors are enabled as well.

CONFIG_DEBUG_PAGEALLOC
: Full pages are removed from the kernel address space when freed. This option can slow things down significantly, but it can also quickly point out certain kinds of memory corruption errors.

CONFIG_DEBUG_SPINLOCK
: With this option enabled, the kernel catches operations on uninitialized spinlocks and various other errors (such as unlocking a lock twice).

CONFIG_DEBUG_SPINLOCK_SLEEP
: This option enables a check for attempts to sleep while holding a spinlock. In fact, it complains if you call a function that could potentially sleep, even if the call in question would not sleep.

CONFIG_INIT_DEBUG
: Items marked with `__init` (or `__initdata`) are discarded after system initialization or module load time. This option enables checks for code that attempts to access initialization-time memory after initialization is complete.

CONFIG_DEBUG_INFO
: This option causes the kernel to be built with full debugging information included. You'll need that information if you want to debug the kernel with *gdb*. You may also want to enable CONFIG_FRAME_POINTER if you plan to use *gdb*.

CONFIG_MAGIC_SYSRQ
: Enables the "magic SysRq" key. We look at this key in the section "System Hangs," later in this chapter.

CONFIG_DEBUG_STACKOVERFLOW
CONFIG_DEBUG_STACK_USAGE
: These options can help track down kernel stack overflows. A sure sign of a stack overflow is an oops listing without any sort of reasonable back trace. The first option adds explicit overflow checks to the kernel; the second causes the kernel to monitor stack usage and make some statistics available via the magic SysRq key.

CONFIG_KALLSYMS
: This option (under "General setup/Standard features") causes kernel symbol information to be built into the kernel; it is enabled by default. The symbol information is used in debugging contexts; without it, an oops listing can give you a kernel traceback only in hexadecimal, which is not very useful.

`CONFIG_IKCONFIG`
`CONFIG_IKCONFIG_PROC`
: These options (found in the "General setup" menu) cause the full kernel configuration state to be built into the kernel and to be made available via */proc*. Most kernel developers know which configuration they used and do not need these options (which make the kernel bigger). They can be useful, though, if you are trying to debug a problem in a kernel built by somebody else.

`CONFIG_ACPI_DEBUG`
: Under "Power management/ACPI." This option turns on verbose ACPI (Advanced Configuration and Power Interface) debugging information, which can be useful if you suspect a problem related to ACPI.

`CONFIG_DEBUG_DRIVER`
: Under "Device drivers." Turns on debugging information in the driver core, which can be useful for tracking down problems in the low-level support code. We'll look at the driver core in Chapter 14.

`CONFIG_SCSI_CONSTANTS`
: This option, found under "Device drivers/SCSI device support," builds in information for verbose SCSI error messages. If you are working on a SCSI driver, you probably want this option.

`CONFIG_INPUT_EVBUG`
: This option (under "Device drivers/Input device support") turns on verbose logging of input events. If you are working on a driver for an input device, this option may be helpful. Be aware of the security implications of this option, however: it logs everything you type, including your passwords.

`CONFIG_PROFILING`
: This option is found under "Profiling support." Profiling is normally used for system performance tuning, but it can also be useful for tracking down some kernel hangs and related problems.

We will revisit some of the above options as we look at various ways of tracking down kernel problems. But first, we will look at the classic debugging technique: print statements.

Debugging by Printing

The most common debugging technique is monitoring, which in applications programming is done by calling *printf* at suitable points. When you are debugging kernel code, you can accomplish the same goal with *printk*.

printk

We used the *printk* function in earlier chapters with the simplifying assumption that it works like *printf*. Now it's time to introduce some of the differences.

One of the differences is that *printk* lets you classify messages according to their severity by associating different *loglevels*, or priorities, with the messages. You usually indicate the loglevel with a macro. For example, `KERN_INFO`, which we saw prepended to some of the earlier print statements, is one of the possible loglevels of the message. The loglevel macro expands to a string, which is concatenated to the message text at compile time; that's why there is no comma between the priority and the format string in the following examples. Here are two examples of *printk* commands, a debug message and a critical message:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

There are eight possible loglevel strings, defined in the header *<linux/kernel.h>*; we list them in order of decreasing severity:

`KERN_EMERG`
: Used for emergency messages, usually those that precede a crash.

`KERN_ALERT`
: A situation requiring immediate action.

`KERN_CRIT`
: Critical conditions, often related to serious hardware or software failures.

`KERN_ERR`
: Used to report error conditions; device drivers often use `KERN_ERR` to report hardware difficulties.

`KERN_WARNING`
: Warnings about problematic situations that do not, in themselves, create serious problems with the system.

`KERN_NOTICE`
: Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

`KERN_INFO`
: Informational messages. Many drivers print information about the hardware they find at startup time at this level.

`KERN_DEBUG`
: Used for debugging messages.

Each string (in the macro expansion) represents an integer in angle brackets. Integers range from 0 to 7, with smaller values representing higher priorities.

A *printk* statement with no specified priority defaults to DEFAULT_MESSAGE_LOGLEVEL, specified in *kernel/printk.c* as an integer. In the 2.6.10 kernel, DEFAULT_MESSAGE_LOGLEVEL is KERN_WARNING, but that has been known to change in the past.

Based on the loglevel, the kernel may print the message to the current console, be it a text-mode terminal, a serial port, or a parallel printer. If the priority is less than the integer variable console_loglevel, the message is delivered to the console one line at a time (nothing is sent unless a trailing newline is provided). If both *klogd* and *syslogd* are running on the system, kernel messages are appended to */var/log/messages* (or otherwise treated depending on your *syslogd* configuration), independent of console_loglevel. If *klogd* is not running, the message won't reach user space unless you read */proc/kmsg* (which is often most easily done with the *dmesg* command). When using *klogd*, you should remember that it doesn't save consecutive identical lines; it only saves the first such line and, at a later time, the number of repetitions it received.

The variable console_loglevel is initialized to DEFAULT_CONSOLE_LOGLEVEL and can be modified through the *sys_syslog* system call. One way to change it is by specifying the *–c* switch when invoking *klogd*, as specified in the *klogd* manpage. Note that to change the current value, you must first kill *klogd* and then restart it with the *–c* option. Alternatively, you can write a program to change the console loglevel. You'll find a version of such a program in *misc-progs/setlevel.c* in the source files provided on O'Reilly's FTP site. The new level is specified as an integer value between 1 and 8, inclusive. If it is set to 1, only messages of level 0 (KERN_EMERG) reach the console; if it is set to 8, all messages, including debugging ones, are displayed.

It is also possible to read and modify the console loglevel using the text file */proc/sys/kernel/printk*. The file hosts four integer values: the current loglevel, the default level for messages that lack an explicit loglevel, the minimum allowed loglevel, and the boot-time default loglevel. Writing a single value to this file changes the current loglevel to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

```
# echo 8 > /proc/sys/kernel/printk
```

It should now be apparent why the *hello.c* sample had the KERN_ALERT; markers; they are there to make sure that the messages appear on the console.

Redirecting Console Messages

Linux allows for some flexibility in console logging policies by letting you send messages to a specific virtual console (if your console lives on the text screen). By default, the "console" is the current virtual terminal. To select a different virtual terminal to receive messages, you can issue ioctl(TIOCLINUX) on any console device. The following program, *setconsole*, can be used to choose which console receives kernel messages; it must be run by the superuser and is available in the *misc-progs* directory.

The following is the program in its entirety. You should invoke it with a single argument specifying the number of the console that is to receive messages.

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11 is the TIOCLINUX cmd number */

    if (argc==2) bytes[1] = atoi(argv[1]); /* the chosen console */
    else {
        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1);
    }
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) {    /* use stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
                argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

setconsole uses the special *ioctl* command `TIOCLINUX`, which implements Linux-specific functions. To use `TIOCLINUX`, you pass it an argument that is a pointer to a byte array. The first byte of the array is a number that specifies the requested subcommand, and the following bytes are subcommand specific. In *setconsole*, subcommand 11 is used, and the next byte (stored in `bytes[1]`) identifies the virtual console. The complete description of `TIOCLINUX` can be found in *drivers/char/tty_io.c*, in the kernel sources.

How Messages Get Logged

The *printk* function writes messages into a circular buffer that is `__LOG_BUF_LEN` bytes long: a value from 4 KB to 1 MB chosen while configuring the kernel. The function then wakes any process that is waiting for messages, that is, any process that is sleeping in the *syslog* system call or that is reading */proc/kmsg*. These two interfaces to the logging engine are almost equivalent, but note that reading from */proc/kmsg* consumes the data from the log buffer, whereas the *syslog* system call can optionally return log data while leaving it for other processes as well. In general, reading the */proc* file is easier and is the default behavior for *klogd*. The *dmesg* command can be used to look at the content of the buffer without flushing it; actually, the command returns to *stdout* the whole content of the buffer, whether or not it has already been read.

If you happen to read the kernel messages by hand, after stopping *klogd*, you'll find that the */proc* file looks like a FIFO, in that the reader blocks, waiting for more data. Obviously, you can't read messages this way if *klogd* or another process is already reading the same data, because you'll contend for it.

If the circular buffer fills up, *printk* wraps around and starts adding new data to the beginning of the buffer, overwriting the oldest data. Therefore, the logging process

loses the oldest data. This problem is negligible compared with the advantages of using such a circular buffer. For example, a circular buffer allows the system to run even without a logging process, while minimizing memory waste by overwriting old data should nobody read it. Another feature of the Linux approach to messaging is that *printk* can be invoked from anywhere, even from an interrupt handler, with no limit on how much data can be printed. The only disadvantage is the possibility of losing some data.

If the *klogd* process is running, it retrieves kernel messages and dispatches them to *syslogd*, which in turn checks */etc/syslog.conf* to find out how to deal with them. *syslogd* differentiates between messages according to a facility and a priority; allowable values for both the facility and the priority are defined in *<sys/syslog.h>*. Kernel messages are logged by the `LOG_KERN` facility at a priority corresponding to the one used in *printk* (for example, `LOG_ERR` is used for `KERN_ERR` messages). If *klogd* isn't running, data remains in the circular buffer until someone reads it or the buffer overflows.

If you want to avoid clobbering your system log with the monitoring messages from your driver, you can either specify the *–f* (file) option to *klogd* to instruct it to save messages to a specific file, or customize */etc/syslog.conf* to suit your needs. Yet another possibility is to take the brute-force approach: kill *klogd* and verbosely print messages on an unused virtual terminal,* or issue the command *cat /proc/kmsg* from an unused *xterm*.

Turning the Messages On and Off

During the early stages of driver development, *printk* can help considerably in debugging and testing new code. When you officially release the driver, on the other hand, you should remove, or at least disable, such print statements. Unfortunately, you're likely to find that as soon as you think you no longer need the messages and remove them, you implement a new feature in the driver (or somebody finds a bug), and you want to turn at least one of the messages back on. There are several ways to solve both issues, to globally enable or disable your debug messages and to turn individual messages on or off.

Here we show one way to code *printk* calls so you can turn them on and off individually or globally; the technique depends on defining a macro that resolves to a *printk* (or *printf*) call when you want it to:

- Each print statement can be enabled or disabled by removing or adding a single letter to the macro's name.
- All the messages can be disabled at once by changing the value of the `CFLAGS` variable before compiling.

* For example, use *setlevel 8; setconsole 10* to set up terminal 10 to display messages.

- The same print statement can be used in kernel code and user-level code, so that the driver and test programs can be managed in the same way with regard to extra messages.

The following code fragment implements these features and comes directly from the header *scull.h*:

```
#undef PDEBUG             /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
     /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

The symbol `PDEBUG` is defined or undefined, depending on whether `SCULL_DEBUG` is defined, and displays information in whatever manner is appropriate to the environment where the code is running: it uses the kernel call *printk* when it's in the kernel and the *libc* call *fprintf* to the standard error when run in user space. The `PDEBUGG` symbol, on the other hand, does nothing; it can be used to easily "comment" print statements without removing them entirely.

To simplify the process further, add the following lines to your makefile:

```
# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
  DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
  DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

The macros shown in this section depend on a *gcc* extension to the ANSI C preprocessor that supports macros with a variable number of arguments. This *gcc* dependency shouldn't be a problem, because the kernel proper depends heavily on *gcc* features anyway. In addition, the makefile depends on GNU's version of *make*; once again, the kernel already depends on GNU *make*, so this dependency is not a problem.

If you're familiar with the C preprocessor, you can expand on the given definitions to implement the concept of a "debug level," defining different levels and assigning an integer (or bit mask) value to each level to determine how verbose it should be.

But every driver has its own features and monitoring needs. The art of good programming is in choosing the best trade-off between flexibility and efficiency, and we can't tell what is the best for you. Remember that preprocessor conditionals (as well as constant expressions in the code) are executed at compile time, so you must recompile to turn messages on or off. A possible alternative is to use C conditionals, which are executed at runtime and, therefore, permit you to turn messaging on and off during program execution. This is a nice feature, but it requires additional processing every time the code is executed, which can affect performance even when the messages are disabled. Sometimes this performance hit is unacceptable.

The macros shown in this section have proven themselves useful in a number of situations, with the only disadvantage being the requirement to recompile a module after any changes to its messages.

Rate Limiting

If you are not careful, you can find yourself generating thousands of messages with *printk*, overwhelming the console and, possibly, overflowing the system log file. When using a slow console device (e.g., a serial port), an excessive message rate can also slow down the system or just make it unresponsive. It can be very hard to get a handle on what is wrong with a system when the console is spewing out data nonstop. Therefore, you should be very careful about what you print, especially in production versions of drivers and especially once initialization is complete. In general, production code should never print anything during normal operation; printed output should be an indication of an exceptional situation requiring attention.

On the other hand, you may want to emit a log message if a device you are driving stops working. But you should be careful not to overdo things. An unintelligent process that continues forever in the face of failures can generate thousands of retries per second; if your driver prints a "my device is broken" message every time, it could create vast amounts of output and possibly hog the CPU if the console device is slow—no interrupts can be used to driver the console, even if it is a serial port or a line printer.

In many cases, the best behavior is to set a flag saying, "I have already complained about this," and not print any further messages once the flag gets set. In others, though, there are reasons to emit an occasional "the device is still broken" notice. The kernel has provided a function that can be helpful in such cases:

```
int printk ratelimit(void);
```

This function should be called before you consider printing a message that could be repeated often. If the function returns a nonzero value, go ahead and print your message, otherwise skip it. Thus, typical calls look like this:

```
if (printk_ratelimit( ))
    printk(KERN_NOTICE "The printer is still on fire\n");
```

printk_ratelimit works by tracking how many messages are sent to the console. When the level of output exceeds a threshold, *printk_ratelimit* starts returning `0` and causing messages to be dropped.

The behavior of *printk_ratelimit* can be customized by modifying */proc/sys/kernel/printk_ratelimit* (the number of seconds to wait before re-enabling messages) and are */proc/sys/kernel/printk_ratelimit_burst* (the number of messages accepted before rate-limiting).

Printing Device Numbers

Occasionally, when printing a message from a driver, you will want to print the device number associated withp the hardware of interest. It is not particularly hard to print the major and minor numbers, but, in the interest of consistency, the kernel provides a couple of utility macros (defined in *<linux/kdev_t.h>*) for this purpose:

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

Both macros encode the device number into the given `buffer`; the only difference is that *print_dev_t* returns the number of characters printed, while *format_dev_t* returns `buffer`; therefore, it can be used as a parameter to a *printk* call directly, although one must remember that *printk* doesn't flush until a trailing newline is provided. The buffer should be large enough to hold a device number; given that 64-bit device numbers are a distinct possibility in future kernel releases, the buffer should probably be at least 20 bytes long.

Debugging by Querying

The previous section described how *printk* works and how it can be used. What it didn't talk about are its disadvantages.

A massive use of *printk* can slow down the system noticeably, even if you lower *console_loglevel* to avoid loading the console device, because *syslogd* keeps syncing its output files; thus, every line that is printed causes a disk operation. This is the right implementation from *syslogd*'s perspective. It tries to write everything to disk in case the system crashes right after printing the message; however, you don't want to slow down your system just for the sake of debugging messages. This problem can be solved by prefixing the name of your log file as it appears in */etc/syslogd.conf* with a

hyphen.* The problem with changing the configuration file is that the modification will likely remain there after you are done debugging, even though during normal system operation you do want messages to be flushed to disk as soon as possible. An alternative to such a permanent change is running a program other than *klogd* (such as *cat /proc/kmsg*, as suggested earlier), but this may not provide a suitable environment for normal system operation.

More often than not, the best way to get relevant information is to query the system when you need the information, instead of continually producing data. In fact, every Unix system provides many tools for obtaining system information: *ps*, *netstat*, *vmstat*, and so on.

A few techniques are available to driver developers for querying the system: creating a file in the */proc* filesystem, using the *ioctl* driver method, and exporting attributes via *sysfs*. The use of *sysfs* requires quite some background on the driver model. It is discussed in Chapter 14.

Using the /proc Filesystem

The */proc* filesystem is a special, software-created filesystem that is used by the kernel to export information to the world. Each file under */proc* is tied to a kernel function that generates the file's "contents" on the fly when the file is read. We have already seen some of these files in action; */proc/modules*, for example, always returns a list of the currently loaded modules.

/proc is heavily used in the Linux system. Many utilities on a modern Linux distribution, such as *ps*, *top*, and *uptime*, get their information from */proc*. Some device drivers also export information via */proc*, and yours can do so as well. The */proc* filesystem is dynamic, so your module can add or remove entries at any time.

Fully featured */proc* entries can be complicated beasts; among other things, they can be written to as well as read from. Most of the time, however, */proc* entries are read-only files. This section concerns itself with the simple read-only case. Those who are interested in implementing something more complicated can look here for the basics; the kernel source may then be consulted for the full picture.

Before we continue, however, we should mention that adding files under */proc* is discouraged. The */proc* filesystem is seen by the kernel developers as a bit of an uncontrolled mess that has gone far beyond its original purpose (which was to provide information about the processes running in the system). The recommended way of making information available in new code is via sysfs. As suggested, working with sysfs requires an understanding of the Linux device model, however, and we do not

* The hyphen, or minus sign, is a "magic" marker to prevent *syslogd* from flushing the file to disk at every new message, documented in *syslog.conf(5)*, a manpage worth reading.

get to that until Chapter 14. Meanwhile, files under */proc* are slightly easier to create, and they are entirely suitable for debugging purposes, so we cover them here.

Implementing files in /proc

All modules that work with */proc* should include *<linux/proc_fs.h>* to define the proper functions.

To create a read-only */proc* file, your driver must implement a function to produce the data when the file is read. When some process reads the file (using the *read* system call), the request reaches your module by means of this function. We'll look at this function first and get to the registration interface later in this section.

When a process reads from your */proc* file, the kernel allocates a page of memory (i.e., `PAGE_SIZE` bytes) where the driver can write data to be returned to user space. That buffer is passed to your function, which is a method called *read_proc*:

```
int (*read_proc)(char *page, char **start, off_t offset, int count,
                 int *eof, void *data);
```

The `page` pointer is the buffer where you'll write your data; `start` is used by the function to say where the interesting data has been written in `page` (more on this later); `offset` and `count` have the same meaning as for the *read* method. The `eof` argument points to an integer that must be set by the driver to signal that it has no more data to return, while `data` is a driver-specific data pointer you can use for internal bookkeeping.

This function should return the number of bytes of data actually placed in the `page` buffer, just like the *read* method does for other files. Other output values are `*eof` and `*start`. `eof` is a simple flag, but the use of the `start` value is somewhat more complicated; its purpose is to help with the implementation of large (greater than one page) */proc* files.

The `start` parameter has a somewhat unconventional use. Its purpose is to indicate where (within `page`) the data to be returned to the user is found. When your *proc_read* method is called, `*start` will be `NULL`. If you leave it `NULL`, the kernel assumes that the data has been put into `page` as if `offset` were `0`; in other words, it assumes a simple-minded version of *proc_read*, which places the entire contents of the virtual file in `page` without paying attention to the `offset` parameter. If, instead, you set `*start` to a non-`NULL` value, the kernel assumes that the data pointed to by `*start` takes `offset` into account and is ready to be returned directly to the user. In general, simple *proc_read* methods that return tiny amounts of data just ignore `start`. More complex methods set `*start` to `page` and only place data beginning at the requested offset there.

There has long been another major issue with */proc* files, which `start` is meant to solve as well. Sometimes the ASCII representation of kernel data structures changes between successive calls to *read*, so the reader process could find inconsistent data from one call to the next. If `*start` is set to a small integer value, the caller uses it to

increment `filp->f_pos` independently of the amount of data you return, thus making `f_pos` an internal record number of your *read_proc* procedure. If, for example, your *read_proc* function is returning information from a big array of structures, and five of those structures were returned in the first call, `*start` could be set to `5`. The next call provides that same value as the offset; the driver then knows to start returning data from the sixth structure in the array. This is acknowledged as a "hack" by its authors and can be seen in *fs/proc/generic.c*.

Note that there is a better way to implement large */proc* files; it's called `seq_file`, and we'll discuss it shortly. First, though, it is time for an example. Here is a simple (if somewhat ugly) *read_proc* implementation for the *scull* device:

```
int scull_read_procmem(char *buf, char **start, off_t offset,
                   int count, int *eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* Don't print more than this */

    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
                i, d->qset, d->quantum, d->size);
        for (; qs && len <= limit; qs = qs->next) { /* scan the list */
            len += sprintf(buf + len, "  item at %p, qset at %p\n",
                     qs, qs->data);
            if (qs->data && !qs->next) /* dump only the last item */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        len += sprintf(buf + len,
                                "    % 4i: %8p\n",
                                j, qs->data[j]);
                }
        }
        up(&scull_devices[i].sem);
    }
    *eof = 1;
    return len;
}
```

This is a fairly typical *read_proc* implementation. It assumes that there will never be a need to generate more than one page of data and so ignores the `start` and `offset` values. It is, however, careful not to overrun its buffer, just in case.

An older interface

If you read through the kernel source, you may encounter code implementing */proc* files with an older interface:

```
int (*get_info)(char *page, char **start, off_t offset, int count);
```

All of the arguments have the same meaning as they do for *read_proc*, but the `eof` and `data` arguments are missing. This interface is still supported, but it could go away in the future; new code should use the *read_proc* interface instead.

Creating your /proc file

Once you have a *read_proc* function defined, you need to connect it to an entry in the */proc* hierarchy. This is done with a call to *create_proc_read_entry*:

```
struct proc_dir_entry *create_proc_read_entry(const char *name,
                              mode_t mode, struct proc_dir_entry *base,
                              read_proc_t *read_proc, void *data);
```

Here, `name` is the name of the file to create, `mode` is the protection mask for the file (it can be passed as 0 for a system-wide default), `base` indicates the directory in which the file should be created (if `base` is `NULL`, the file is created in the */proc* root), `read_proc` is the *read_proc* function that implements the file, and `data` is ignored by the kernel (but passed to *read_proc*). Here is the call used by *scull* to make its */proc* function available as */proc/scullmem*:

```
create_proc_read_entry("scullmem", 0 /* default mode */,
        NULL /* parent dir */, scull_read_procmem,
        NULL /* client data */);
```

Here, we create a file called *scullmem* directly under */proc*, with the default, world-readable protections.

The directory entry pointer can be used to create entire directory hierarchies under */proc*. Note, however, that an entry may be more easily placed in a subdirectory of */proc* simply by giving the directory name as part of the name of the entry—as long as the directory itself already exists. For example, an (often ignored) convention says that */proc* entries associated with device drivers should go in the subdirectory *driver/*; *scull* could place its entry there simply by giving its name as *driver/scullmem*.

Entries in */proc*, of course, should be removed when the module is unloaded. *remove_proc_entry* is the function that undoes what *create_proc_read_entry* already did:

```
remove_proc_entry("scullmem", NULL /* parent dir */);
```

Failure to remove entries can result in calls at unwanted times, or, if your module has been unloaded, kernel crashes.

When using */proc* files as shown, you must remember a few nuisances of the implementation—no surprise its use is discouraged nowadays.

The most important problem is with removal of */proc* entries. Such removal may well happen while the file is in use, as there is no owner associated to */proc* entries, so using them doesn't act on the module's reference count. This problem is simply triggered by running *sleep 100 < /proc/myfile* just before removing the module, for example.

Another issue is about registering two entries with the same name. The kernel trusts the driver and doesn't check if the name is already registered, so if you are not careful you might end up with two or more entries with the same name. This is a problem known to happen in classrooms, and such entries are indistinguishable, both when you access them and when you call *remove_proc_entry*.

The seq_file interface

As we noted above, the implementation of large files under */proc* is a little awkward. Over time, */proc* methods have become notorious for buggy implementations when the amount of output grows large. As a way of cleaning up the */proc* code and making life easier for kernel programmers, the `seq_file` interface was added. This interface provides a simple set of functions for the implementation of large kernel virtual files.

The `seq_file` interface assumes that you are creating a virtual file that steps through a sequence of items that must be returned to user space. To use `seq_file`, you must create a simple "iterator" object that can establish a position within the sequence, step forward, and output one item in the sequence. It may sound complicated, but, in fact, the process is quite simple. We'll step through the creation of a */proc* file in the *scull* driver to show how it is done.

The first step, inevitably, is the inclusion of *<linux/seq_file.h>*. Then you must create four iterator methods, called *start*, *next*, *stop*, and *show*.

The *start* method is always called first. The prototype for this function is:

```
void *start(struct seq_file *sfile, loff_t *pos);
```

The `sfile` argument can almost always be ignored. `pos` is an integer position indicating where the reading should start. The interpretation of the position is entirely up to the implementation; it need not be a byte position in the resulting file. Since `seq_file` implementations typically step through a sequence of interesting items, the position is often interpreted as a cursor pointing to the next item in the sequence. The *scull* driver interprets each device as one item in the sequence, so the incoming `pos` is simply an index into the `scull_devices` array. Thus, the *start* method used in *scull* is:

```
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
    if (*pos >= scull_nr_devs)
        return NULL;   /* No more to read */
    return scull_devices + *pos;
}
```

The return value, if non-`NULL`, is a private value that can be used by the iterator implementation.

The *next* function should move the iterator to the next position, returning `NULL` if there is nothing left in the sequence. This method's prototype is:

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

Here, v is the iterator as returned from the previous call to *start* or *next*, and pos is the current position in the file. *next* should increment the value pointed to by pos; depending on how your iterator works, you might (though probably won't) want to increment pos by more than one. Here's what *scull* does:

```
static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    (*pos)++;
    if (*pos >= scull_nr_devs)
        return NULL;
    return scull_devices + *pos;
}
```

When the kernel is done with the iterator, it calls *stop* to clean up:

```
void stop(struct seq_file *sfile, void *v);
```

The *scull* implementation has no cleanup work to do, so its *stop* method is empty.

It is worth noting that the seq_file code, by design, does not sleep or perform other nonatomic tasks between the calls to *start* and *stop*. You are also guaranteed to see one *stop* call sometime shortly after a call to *start*. Therefore, it is safe for your *start* method to acquire semaphores or spinlocks. As long as your other seq_file methods are atomic, the whole sequence of calls is atomic. (If this paragraph does not make sense to you, come back to it after you've read the next chapter.)

In between these calls, the kernel calls the *show* method to actually output something interesting to the user space. This method's prototype is:

```
int show(struct seq_file *sfile, void *v);
```

This method should create output for the item in the sequence indicated by the iterator v. It should not use *printk*, however; instead, there is a special set of functions for seq_file output:

`int seq_printf(struct seq_file *sfile, const char *fmt, ...);`
: This is the *printf* equivalent for seq_file implementations; it takes the usual format string and additional value arguments. You must also pass it the seq_file structure given to the *show* function, however. If *seq_printf* returns a nonzero value, it means that the buffer has filled, and output is being discarded. Most implementations ignore the return value, however.

`int seq_putc(struct seq_file *sfile, char c);`
`int seq_puts(struct seq_file *sfile, const char *s);`
: These are the equivalents of the user-space *putc* and *puts* functions.

`int seq_escape(struct seq_file *m, const char *s, const char *esc);`
: This function is equivalent to *seq_puts* with the exception that any character in s that is also found in esc is printed in octal format. A common value for esc is `" \t\n\\"`, which keeps embedded white space from messing up the output and possibly confusing shell scripts.

```
int seq_path(struct seq_file *sfile, struct vfsmount *m, struct dentry
  *dentry, char *esc);
```

This function can be used for outputting the file name associated with a given directory entry. It is unlikely to be useful in device drivers; we have included it here for completeness.

Getting back to our example; the *show* method used in *scull* is:

```
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
            (int) (dev - scull_devices), dev->qset,
            dev->quantum, dev->size);
    for (d = dev->data; d; d = d->next) { /* scan the list */
        seq_printf(s, "  item at %p, qset at %p\n", d, d->data);
        if (d->data && !d->next) /* dump only the last item */
            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, "    % 4i: %8p\n",
                            i, d->data[i]);
            }
    }
    up(&dev->sem);
    return 0;
}
```

Here, we finally interpret our "iterator" value, which is simply a pointer to a `scull_dev` structure.

Now that it has a full set of iterator operations, *scull* must package them up and connect them to a file in */proc*. The first step is done by filling in a `seq_operations` structure:

```
static struct seq_operations scull_seq_ops = {
    .start = scull_seq_start,
    .next  = scull_seq_next,
    .stop  = scull_seq_stop,
    .show  = scull_seq_show
};
```

With that structure in place, we must create a file implementation that the kernel understands. We do not use the *read_proc* method described previously; when using `seq_file`, it is best to connect in to */proc* at a slightly lower level. That means creating a `file_operations` structure (yes, the same structure used for char drivers) implementing all of the operations needed by the kernel to handle reads and seeks on the

file. Fortunately, this task is straightforward. The first step is to create an *open* method that connects the file to the `seq_file` operations:

```
static int scull_proc_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &scull_seq_ops);
}
```

The call to *seq_open* connects the `file` structure with our sequence operations defined above. As it turns out, *open* is the only file operation we must implement ourselves, so we can now set up our `file_operations` structure:

```
static struct file_operations scull_proc_ops = {
    .owner   = THIS_MODULE,
    .open    = scull_proc_open,
    .read    = seq_read,
    .llseek  = seq_lseek,
    .release = seq_release
};
```

Here we specify our own *open* method, but use the canned methods *seq_read*, *seq_lseek*, and *seq_release* for everything else.

The final step is to create the actual file in */proc*:

```
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

Rather than using *create_proc_read_entry*, we call the lower-level *create_proc_entry*, which has this prototype:

```
struct proc_dir_entry *create_proc_entry(const char *name,
                                         mode_t mode,
                                         struct proc_dir_entry *parent);
```

The arguments are the same as their equivalents in *create_proc_read_entry*: the name of the file, its protections, and the parent directory.

With the above code, *scull* has a new */proc* entry that looks much like the previous one. It is superior, however, because it works regardless of how large its output becomes, it handles seeks properly, and it is generally easier to read and maintain. We recommend the use of `seq_file` for the implementation of files that contain more than a very small number of lines of output.

The ioctl Method

ioctl, which we show you how to use in Chapter 1, is a system call that acts on a file descriptor; it receives a number that identifies a command to be performed and (optionally) another argument, usually a pointer. As an alternative to using the */proc* filesystem, you can implement a few *ioctl* commands tailored for debugging. These

commands can copy relevant data structures from the driver to user space where you can examine them.

Using *ioctl* this way to get information is somewhat more difficult than using */proc*, because you need another program to issue the *ioctl* and display the results. This program must be written, compiled, and kept in sync with the module you're testing. On the other hand, the driver-side code can be easier than what is needed to implement a */proc* file.

There are times when *ioctl* is the best way to get information, because it runs faster than reading */proc*. If some work must be performed on the data before it's written to the screen, retrieving the data in binary form is more efficient than reading a text file. In addition, *ioctl* doesn't require splitting data into fragments smaller than a page.

Another interesting advantage of the *ioctl* approach is that information-retrieval commands can be left in the driver even when debugging would otherwise be disabled. Unlike a */proc* file, which is visible to anyone who looks in the directory (and too many people are likely to wonder "what that strange file is"), undocumented *ioctl* commands are likely to remain unnoticed. In addition, they will still be there should something weird happen to the driver. The only drawback is that the module will be slightly bigger.

Debugging by Watching

Sometimes minor problems can be tracked down by watching the behavior of an application in user space. Watching programs can also help in building confidence that a driver is working correctly. For example, we were able to feel confident about *scull* after looking at how its *read* implementation reacted to read requests for different amounts of data.

There are various ways to watch a user-space program working. You can run a debugger on it to step through its functions, add print statements, or run the program under *strace*. Here we'll discuss just the last technique, which is most interesting when the real goal is examining kernel code.

The *strace* command is a powerful tool that shows all the system calls issued by a user-space program. Not only does it show the calls, but it can also show the arguments to the calls and their return values in symbolic form. When a system call fails, both the symbolic value of the error (e.g., `ENOMEM`) and the corresponding string (`Out of memory`) are displayed. *strace* has many command-line options; the most useful of which are *-t* to display the time *when* each call is executed, *-T* to display the time spent in the call, *-e* to limit the types of calls traced, and *-o* to redirect the output to a file. By default, *strace* prints tracing information on `stderr`.

strace receives information from the kernel itself. This means that a program can be traced regardless of whether or not it was compiled with debugging support (the *-g*

option to *gcc*) and whether or not it is stripped. You can also attach tracing to a running process, similar to the way a debugger can connect to a running process and control it.

The trace information is often used to support bug reports sent to application developers, but it's also invaluable to kernel programmers. We've seen how driver code executes by reacting to system calls; *strace* allows us to check the consistency of input and output data of each call.

For example, the following screen dump shows (most of) the last lines of running the command *strace ls /dev > /dev/scull0*:

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC)         = 0
getdents64(3, /* 141 entries */, 4096)  = 4088
[...]
getdents64(3, /* 0 entries */, 4096)    = 0
close(3)                                = 0
[...]
fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1)                                = 0
exit_group(0)                           = ?
```

It's apparent from the first *write* call that after *ls* finished looking in the target directory, it tried to write 4 KB. Strangely (for *ls*), only 4000 bytes were written, and the operation was retried. However, we know that the *write* implementation in *scull* writes a single quantum at a time, so we could have expected the partial write. After a few steps, everything sweeps through, and the program exits successfully.

As another example, let's *read* the *scull* device (using the *wc* command):

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865
read(3, "", 16384)                      = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17)      = 17
close(3)                                = 0
exit_group(0)                           = ?
```

As expected, *read* is able to retrieve only 4000 bytes at a time, but the total amount of data is the same that was written in the previous example. It's interesting to note how retries are organized in this example, as opposed to the previous trace. *wc* is

optimized for fast reading and, therefore, bypasses the standard library, trying to read more data with a single system call. You can see from the `read` lines in the trace how *wc* tried to read 16 KB at a time.

Linux experts can find much useful information in the output of *strace*. If you're put off by all the symbols, you can limit yourself to watching how the file methods (*open*, *read*, and so on) work with the `efile` flag.

Personally, we find *strace* most useful for pinpointing runtime errors from system calls. Often the *perror* call in the application or demo program isn't verbose enough to be useful for debugging, and being able to tell exactly which arguments to which system call triggered the error can be a great help.

Debugging System Faults

Even if you've used all the monitoring and debugging techniques, sometimes bugs remain in the driver, and the system faults when the driver is executed. When this happens, it's important to be able to collect as much information as possible to solve the problem.

Note that "fault" doesn't mean "panic." The Linux code is robust enough to respond gracefully to most errors: a fault usually results in the destruction of the current process while the system goes on working. The system *can* panic, and it may if a fault happens outside of a process's context or if some vital part of the system is compromised. But when the problem is due to a driver error, it usually results only in the sudden death of the process unlucky enough to be using the driver. The only unrecoverable damage when a process is destroyed is that some memory allocated to the process's context is lost; for instance, dynamic lists allocated by the driver through *kmalloc* might be lost. However, since the kernel calls the *close* operation for any open device when a process dies, your driver can release what was allocated by the *open* method.

Even though an oops usually does not bring down the entire system, you may well find yourself needing to reboot after one happens. A buggy driver can leave hardware in an unusable state, leave kernel resources in an inconsistent state, or, in the worst case, corrupt kernel memory in random places. Often you can simply unload your buggy driver and try again after an oops. If, however, you see anything that suggests that the system as a whole is not well, your best bet is usually to reboot immediately.

We've already said that when kernel code misbehaves, an informative message is printed on the console. The next section explains how to decode and use such messages. Even though they appear rather obscure to the novice, processor dumps are full of interesting information, often sufficient to pinpoint a program bug without the need for additional testing.

Oops Messages

Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values. The usual outcome of such bugs is an oops message.

Almost any address used by the processor is a virtual address and is mapped to physical addresses through a complex structure of page tables (the exceptions are physical addresses used with the memory management subsystem itself). When an invalid pointer is dereferenced, the paging mechanism fails to map the pointer to a physical address, and the processor signals a *page fault* to the operating system. If the address is not valid, the kernel is not able to "page in" the missing address; it (usually) generates an oops if this happens while the processor is in supervisor mode.

An oops displays the processor status at the time of the fault, including the contents of the CPU registers and other seemingly incomprehensible information. The message is generated by *printk* statements in the fault handler (*arch/*/kernel/traps.c*) and is dispatched as described earlier in the section "printk."

Let's look at one such message. Here's what results from dereferencing a NULL pointer on a PC running Version 2.6 of the kernel. The most relevant information here is the instruction pointer (EIP), the address of the faulty instruction.

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246   (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000   edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005   esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       ffffffff 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

This message was generated by writing to a device owned by the *faulty* module, a module built deliberately to demonstrate failures. The implementation of the *write* method of *faulty.c* is trivial:

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t count,
        loff_t *pos)
{
```

```
    /* make a simple fault by dereferencing a NULL pointer */
    *(int *)0 = 0;
    return 0;
}
```

As you can see, what we do here is dereference a `NULL` pointer. Since `0` is never a valid pointer value, a fault occurs, which the kernel turns into the oops message shown earlier. The calling process is then killed.

The *faulty* module has a different fault condition in its *read* implementation:

```
ssize_t faulty_read(struct file *filp, char __user *buf,
            size_t count, loff_t *pos)
{
    int ret;
    char stack_buf[4];

    /* Let's try a buffer overflow  */
    memset(stack_buf, 0xff, 20);
    if (count > 4)
        count = 4; /* copy 4 bytes to the user */
    ret = copy_to_user(buf, stack_buf, count);
    if (!ret)
        return count;
    return ret;
}
```

This method copies a string into a local variable; unfortunately, the string is longer than the destination array. The resulting buffer overflow causes an oops when the function returns. Since the `return` instruction brings the instruction pointer to nowhere land, this kind of fault is much harder to trace, and you can get something such as the following:

```
EIP:    0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff
 printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU:    0
EIP:    0060:[<ffffffff>]    Not tainted
EFLAGS: 00010296   (2.6.6)
EIP is at 0xffffffff
eax: 0000000c   ebx: ffffffff   ecx: 00000000   edx: bfffda7c
esi: cf434f00   edi: ffffffff   ebp: 00002000   esp: c27fff78
ds: 007b   es: 007b   ss: 0068
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace:
 [<c0150612>] sys_read+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code:  Bad EIP value.
```

In this case, we see only part of the call stack (*vfs_read* and *faulty_read* are missing), and the kernel complains about a "bad EIP value." That complaint, and the offending address (`ffffffff`) listed at the beginning are both hints that the kernel stack has been corrupted.

In general, when you are confronted with an oops, the first thing to do is to look at the location where the problem happened, which is usually listed separately from the call stack. In the first oops shown above, the relevant line is:

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

Here we see that we were in the function *faulty_write*, which is located in the *faulty* module (which is listed in square brackets). The hex numbers indicate that the instruction pointer was 4 bytes into the function, which appears to be 10 (hex) bytes long. Often that is enough to figure out what the problem is.

If you need more information, the call stack shows you how you got to where things fell apart. The stack itself is printed in hex form; with a bit of work, you can often determine the values of local variables and function parameters from the stack listing. Experienced kernel developers can benefit from a certain amount of pattern recognition here; for example, if we look at the stack listing from the *faulty_read* oops:

```
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
```

The `ffffffff` at the top of the stack is part of our string that broke things. On the x86 architecture, by default, the user-space stack starts just below `0xc0000000`; thus, the recurring value `0xbfffda70` is probably a user-space stack address; it is, in fact, the address of the buffer passed to the *read* system call, replicated each time it is passed down the kernel call chain. On the x86 (again, by default), kernel space starts at `0xc0000000`, so values above that are almost certainly kernel-space addresses, and so on.

Finally, when looking at oops listings, always be on the lookout for the "slab poisoning" values discussed at the beginning of this chapter. Thus, for example, if you get a kernel oops where the offending address is `0xa5a5a5a5`, you are almost certainly forgetting to initialize dynamic memory somewhere.

Please note that you see a symbolic call stack (as shown above) only if your kernel is built with the `CONFIG_KALLSYMS` option turned on. Otherwise, you see a bare, hexadecimal listing, which is far less useful until you have decoded it in other ways.

System Hangs

Although most bugs in kernel code end up as oops messages, sometimes they can completely hang the system. If the system hangs, no message is printed. For example,

if the code enters an endless loop, the kernel stops scheduling,* and the system doesn't respond to any action, including the magic Ctrl-Alt-Del combination. You have two choices for dealing with system hangs—either prevent them beforehand or be able to debug them after the fact.

You can prevent an endless loop by inserting *schedule* invocations at strategic points. The *schedule* call (as you might guess) invokes the scheduler and, therefore, allows other processes to steal CPU time from the current process. If a process is looping in kernel space due to a bug in your driver, the *schedule* calls enable you to kill the process after tracing what is happening.

You should be aware, of course, that any call to *schedule* may create an additional source of reentrant calls to your driver, since it allows other processes to run. This reentrancy should not normally be a problem, assuming that you have used suitable locking in your driver. Be sure, however, not to call *schedule* any time that your driver is holding a spinlock.

If your driver really hangs the system, and you don't know where to insert *schedule* calls, the best way to go may be to add some print messages and write them to the console (by changing the `console_loglevel` value if need be).

Sometimes the system may appear to be hung, but it isn't. This can happen, for example, if the keyboard remains locked in some strange way. These false hangs can be detected by looking at the output of a program you keep running for just this purpose. A clock or system load meter on your display is a good status monitor; as long as it continues to update, the scheduler is working.

An indispensable tool for many lockups is the "magic SysRq key," which is available on most architectures. Magic SysRq is invoked with the combination of the Alt and SysRq keys on the PC keyboard, or with other special keys on other platforms (see *Documentation/sysrq.txt* for details), and is available on the serial console as well. A third key, pressed along with these two, performs one of a number of useful actions:

`r` Turns off keyboard raw mode; useful in situations where a crashed application (such as the X server) may have left your keyboard in a strange state.

`k` Invokes the "secure attention key" (SAK) function. SAK kills all processes running on the current console, leaving you with a clean terminal.

`s` Performs an emergency synchronization of all disks.

`u` Umount. Attempts to remount all disks in a read-only mode. This operation, usually invoked immediately after *s*, can save a lot of filesystem checking time in cases where the system is in serious trouble.

* Actually, multiprocessor systems still schedule on the other processors, and even a uniprocessor machine might reschedule if kernel preemption is enabled. For the most common case (uniprocessor with preemption disabled), however, the system stops scheduling altogether.

`b` Boot. Immediately reboots the system. Be sure to synchronize and remount the disks first.

`p` Prints processor registers information.

`t` Prints the current task list.

`m` Prints memory information.

Other magic SysRq functions exist; see *sysrq.txt* in the *Documentation* directory of the kernel source for the full list. Note that magic SysRq must be explicitly enabled in the kernel configuration and that most distributions do not enable it, for obvious security reasons. For a system used to develop drivers, however, enabling magic SysRq is worth the trouble of building a new kernel in itself. Magic SysRq may be disabled at runtime with a command such as the following:

```
echo 0 > /proc/sys/kernel/sysrq
```

You should consider disabling it if unprivileged users can reach your system keyboard, to prevent accidental or willing damages. Some previous kernel versions had *sysrq* disabled by default, so you needed to enable it at runtime by writing 1 to that same */proc/sys* file.

The *sysrq* operations are exceedingly useful, so they have been made available to system administrators who can't reach the console. The file */proc/sysrq-trigger* is a write-only entry point, where you can trigger a specific *sysrq* action by writing the associated command character; you can then collect any output data from the kernel logs. This entry point to *sysrq* is always working, even if *sysrq* is disabled on the console.

If you are experiencing a "live hang," in which your driver is stuck in a loop but the system as a whole is still functioning, there are a couple of techniques worth knowing. Often, the SysRq *p* function points the finger directly at the guilty routine. Failing that, you can also use the kernel profiling function. Build a kernel with profiling enabled, and boot it with `profile=2` on the command line. Reset the profile counters with the *readprofile* utility, then send your driver into its loop. After a little while, use *readprofile* again to see where the kernel is spending its time. Another more advanced alternative is *oprofile*, that you may consider as well. The file *Documentation/basic_profiling.txt* tells you everything you need to know to get started with the profilers.

One precaution worth using when chasing system hangs is to mount all your disks as read-only (or unmount them). If the disks are read-only or unmounted, there's no risk of damaging the filesystem or leaving it in an inconsistent state. Another possibility is using a computer that mounts all of its filesystems via NFS, the network file system. The "NFS-Root" capability must be enabled in the kernel, and special parameters must be passed at boot time. In this case, you'll avoid filesystem corruption without even resorting to SysRq, because filesystem coherence is managed by the NFS server, which is not brought down by your device driver.

Debuggers and Related Tools

The last resort in debugging modules is using a debugger to step through the code, watching the value of variables and machine registers. This approach is time-consuming and should be avoided whenever possible. Nonetheless, the fine-grained perspective on the code that is achieved through a debugger is sometimes invaluable.

Using an interactive debugger on the kernel is a challenge. The kernel runs in its own address space on behalf of all the processes on the system. As a result, a number of common capabilities provided by user-space debuggers, such as breakpoints and single-stepping, are harder to come by in the kernel. In this section we look at several ways of debugging the kernel; each of them has advantages and disadvantages.

Using gdb

gdb can be quite useful for looking at the system internals. Proficient use of the debugger at this level requires some confidence with *gdb* commands, some understanding of assembly code for the target platform, and the ability to match source code and optimized assembly.

The debugger must be invoked as though the kernel were an application. In addition to specifying the filename for the ELF kernel image, you need to provide the name of a core file on the command line. For a running kernel, that core file is the kernel core image, */proc/kcore*. A typical invocation of *gdb* looks like the following:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

The first argument is the name of the uncompressed ELF kernel executable, not the *zImage* or *bzImage* or anything built specifically for the boot environment.

The second argument on the *gdb* command line is the name of the core file. Like any file in */proc*, */proc/kcore* is generated when it is read. When the *read* system call executes in the */proc* filesystem, it maps to a data-generation function rather than a data-retrieval one; we've already exploited this feature in the section "Using the /proc Filesystem" earlier in this chapter. *kcore* is used to represent the kernel "executable" in the format of a core file; it is a huge file, because it represents the whole kernel address space, which corresponds to all physical memory. From within *gdb*, you can look at kernel variables by issuing the standard *gdb* commands. For example, `p jiffies` prints the number of clock ticks from system boot to the current time.

When you print data from *gdb*, the kernel is still running, and the various data items have different values at different times; *gdb*, however, optimizes access to the core file by caching data that has already been read. If you try to look at the `jiffies` variable once again, you'll get the same answer as before. Caching values to avoid extra disk access is a correct behavior for conventional core files but is inconvenient when a "dynamic" core image is used. The solution is to issue the command *core-file /proc/kcore* whenever you want to flush the *gdb* cache; the debugger gets ready to use a

new core file and discards any old information. You won't, however, always need to issue *core-file* when reading a new datum; *gdb* reads the core in chunks of a few kilobytes and caches only chunks it has already referenced.

Numerous capabilities normally provided by *gdb* are not available when you are working with the kernel. For example, *gdb* is not able to modify kernel data; it expects to be running a program to be debugged under its own control before playing with its memory image. It is also not possible to set breakpoints or watchpoints, or to single-step through kernel functions.

Note that, in order to have symbol information available for *gdb*, you must compile your kernel with the `CONFIG_DEBUG_INFO` option set. The result is a far larger kernel image on disk, but, without that information, digging through kernel variables is almost impossible.

With the debugging information available, you can learn a lot about what is going on inside the kernel. *gdb* happily prints out structures, follows pointers, etc. One thing that is harder, however, is examining modules. Since modules are not part of the *vmlinux* image passed to *gdb*, the debugger knows nothing about them. Fortunately, as of kernel 2.6.7, it is possible to teach *gdb* what it needs to know to examine loadable modules.

Linux loadable modules are ELF-format executable images; as such, they have been divided up into numerous sections. A typical module can contain a dozen or more sections, but there are typically three that are relevant in a debugging session:

`.text`
: This section contains the executable code for the module. The debugger must know where this section is to be able to give tracebacks or set breakpoints. (Neither of these operations is relevant when running the debugger on */proc/kcore*, but they can useful when working with *kgdb*, described below).

`.bss`
`.data`
: These two sections hold the module's variables. Any variable that is not initialized at compile time ends up in `.bss`, while those that are initialized go into `.data`.

Making *gdb* work with loadable modules requires informing the debugger about where a given module's sections have been loaded. That information is available in sysfs, under */sys/module*. For example, after loading the *scull* module, the directory */sys/module/scull/sections* contains files with names such as *.text*; the content of each file is the base address for that section.

We are now in a position to issue a *gdb* command telling it about our module. The command we need is `add-symbol-file`; this command takes as parameters the name of the module object file, the *.text* base address, and a series of optional parameters

describing where any other sections of interest have been put. After digging through the module section data in sysfs, we can construct a command such as:

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
        -s .bss 0xd0837100 \
        -s .data 0xd0836be0
```

We have included a small script in the sample source (*gdbline*) that can create this command for a given module.

We can now use *gdb* to examine variables in our loadable module. Here is a quick example taken from a *scull* debugging session:

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
      -s .bss 0xd0837100 \
      -s .data 0xd0836be0
add symbol table from file "scull.ko" at
        .text_addr = 0xd0832000
        .bss_addr = 0xd0837100
        .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
      quantum = 4000,
      qset = 1000,
      size = 20881,
      access_key = 0,
      ...}
```

Here we see that the first *scull* device currently holds 20,881 bytes. If we wanted, we could follow the `data` chain, or look at anything else of interest in the module.

One other useful trick worth knowing about is this:

```
(gdb) print *(address)
```

Here, fill in a hex address for `address`; the output is a file and line number for the code corresponding to that address. This technique may be useful, for example, to find out where a function pointer really points.

We still cannot perform typical debugging tasks like setting breakpoints or modifying data; to perform those operations, we need to use a tool like *kdb* (described next) or *kgdb* (which we get to shortly).

The kdb Kernel Debugger

Many readers may be wondering why the kernel does not have any more advanced debugging features built into it. The answer, quite simply, is that Linus does not believe in interactive debuggers. He fears that they lead to poor fixes, those which patch up symptoms rather than addressing the real cause of problems. Thus, no built-in debuggers.

Other kernel developers, however, see an occasional use for interactive debugging tools. One such tool is the *kdb* built-in kernel debugger, available as a nonofficial patch from *oss.sgi.com*. To use *kdb*, you must obtain the patch (be sure to get a version that matches your kernel version), apply it, and rebuild and reinstall the kernel. Note that, as of this writing, *kdb* works only on IA-32 (x86) systems (though a version for the IA-64 existed for a while in the mainline kernel source before being removed).

Once you are running a *kdb*-enabled kernel, there are a couple of ways to enter the debugger. Pressing the Pause (or Break) key on the console starts up the debugger. *kdb* also starts up when a kernel oops happens or when a breakpoint is hit. In any case, you see a message that looks something like this:

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

Note that just about everything the kernel does stops when *kdb* is running. Nothing else should be running on a system where you invoke *kdb*; in particular, you should not have networking turned on—unless, of course, you are debugging a network driver. It is generally a good idea to boot the system in single-user mode if you will be using *kdb*.

As an example, consider a quick *scull* debugging session. Assuming that the driver is already loaded, we can tell *kdb* to set a breakpoint in *scull_read* as follows:

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
    is enabled globally adjust 1
[0]kdb> go
```

The *bp* command tells *kdb* to stop the next time the kernel enters *scull_read*. You then type **go** to continue execution. After putting something into one of the *scull* devices, we can attempt to read it by running *cat* under a shell on another terminal, yielding the following:

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read:          int3

Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
Breakpoint @ 0xd087c5dc
[0]kdb>
```

We are now positioned at the beginning of *scull_read*. To see how we got there, we can get a stack trace:

```
[0]kdb> bt
    ESP    EIP        Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb attempts to print out the arguments to every function in the call trace. It gets confused, however, by optimization tricks used by the compiler. Therefore, it fails to print the arguments to *scull_read*.

Time to look at some data. The *mds* command manipulates data; we can query the value of the `scull_devices` pointer with a command such as:

```
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00    ....
```

Here we asked for one (4-byte) word of data starting at the location of `scull_devices`; the answer tells us that our device array is at the address `0xd0880de8`; the first device structure itself is at `0xcf36ac00`. To look at that device structure, we need to use that address:

```
[0]kdb> mds cf36ac00
0xcf36ac00 ce137dbc ....
0xcf36ac04 00000fa0 ....
0xcf36ac08 000003e8 ....
0xcf36ac0c 0000009b ....
0xcf36ac10 00000000 ....
0xcf36ac14 00000001 ....
0xcf36ac18 00000000 ....
0xcf36ac1c 00000001 ....
```

The eight lines here correspond to the beginning part of the `scull_dev` structure. Therefore, we see that the memory for the first device is allocated at `0xce137dbc`, the quantum is 4000 (hex `fa0`), the quantum set size is 1000 (hex `3e8`), and there are currently 155 (hex `9b`) bytes stored in the device.

kdb can change data as well. Suppose we wanted to trim some of the data from the device:

```
[0]kdb> mm cf26ac0c 0x50
0xcf26ac0c = 0x50
```

A subsequent *cat* on the device will now return less data than before.

kdb has a number of other capabilities, including single-stepping (by instructions, not lines of C source code), setting breakpoints on data access, disassembling code, stepping through linked lists, accessing register data, and more. After you have applied the *kdb* patch, a full set of manual pages can be found in the *Documentation/kdb* directory in your kernel source tree.

The kgdb Patches

The two interactive debugging approaches we have looked at so far (using *gdb* on */proc/kcore* and *kdb*) both fall short of the sort of environment that user-space application developers have become used to. Wouldn't it be nice if there were a true debugger for the kernel that supported features like changing variables, breakpoints, etc.?

As it turns out, such a solution does exist. There are, as of this writing, two separate patches in circulation that allow *gdb*, with full capabilities, to be run against the kernel. Confusingly, both of these patches are called *kgdb*. They work by separating the system running the test kernel from the system running the debugger; the two are typically connected via a serial cable. Therefore, the developer can run *gdb* on his or her stable desktop system, while operating on a kernel running on a sacrificial test box. Setting up *gdb* in this mode takes a little time at the outset, but that investment can pay off quickly when a difficult bug shows up.

These patches are in a strong state of flux, and may even be merged at some point, so we avoid saying much about them beyond where they are and their basic features. Interested readers are encouraged to look and see the current state of affairs.

The first *kgdb* patch is currently found in the `-mm` kernel tree—the staging area for patches on their way into the 2.6 mainline. This version of the patch supports the x86, SuperH, ia64, x86_64, SPARC, and 32-bit PPC architectures. In addition to the usual mode of operation over a serial port, this version of *kgdb* can also communicate over a local-area network. It is simply a matter of enabling the Ethernet mode and booting with the `kgdboe` parameter set to indicate the IP address from which debugging commands can originate. The documentation under *Documentation/i386/kgdb* describes how to set things up.*

As an alternative, you can use the *kgdb* patch found on *http://kgdb.sf.net/*. This version of the debugger does not support the network communication mode (though that is said to be under development), but it does have some built-in support for working with loadable modules. It supports the x86, x86_64, PowerPC, and S/390 architectures.

The User-Mode Linux Port

User-Mode Linux (UML) is an interesting concept. It is structured as a separate port of the Linux kernel with its own *arch/um* subdirectory. It does not run on a new type of hardware, however; instead, it runs on a virtual machine implemented on the Linux system call interface. Thus, UML allows the Linux kernel to run as a separate, user-mode process on a Linux system.

Having a copy of the kernel running as a user-mode process brings a number of advantages. Because it is running on a constrained, virtual processor, a buggy kernel cannot damage the "real" system. Different hardware and software configurations can be tried easily on the same box. And, perhaps most significantly for kernel developers, the user-mode kernel can be easily manipulated with *gdb* or another debugger.

* It does neglect to point out that you should have your network adapter driver built into the kernel, however, or the debugger fails to find it at boot time and will shut itself down.

After all, it is just another process. UML clearly has the potential to accelerate kernel development.

However, UML has a big shortcoming from the point of view of driver writers: the user-mode kernel has no access to the host system's hardware. Thus, while it can be useful for debugging most of the sample drivers in this book, UML is not yet useful for debugging drivers that have to deal with real hardware.

See *http://user-mode-linux.sf.net/* for more information on UML.

The Linux Trace Toolkit

The Linux Trace Toolkit (LTT) is a kernel patch and a set of related utilities that allow the tracing of events in the kernel. The trace includes timing information and can create a reasonably complete picture of what happened over a given period of time. Thus, it can be used not only for debugging but also for tracking down performance problems.

LTT, along with extensive documentation, can be found at *http://www.opersys.com/LTT*.

Dynamic Probes

Dynamic Probes (or DProbes) is a debugging tool released (under the GPL) by IBM for Linux on the IA-32 architecture. It allows the placement of a "probe" at almost any place in the system, in both user and kernel space. The probe consists of some code (written in a specialized, stack-oriented language) that is executed when control hits the given point. This code can report information back to user space, change registers, or do a number of other things. The useful feature of DProbes is that once the capability has been built into the kernel, probes can be inserted anywhere within a running system without kernel builds or reboots. DProbes can also work with the LTT to insert new tracing events at arbitrary locations.

The DProbes tool can be downloaded from IBM's open source site: *http://oss.software.ibm.com*.