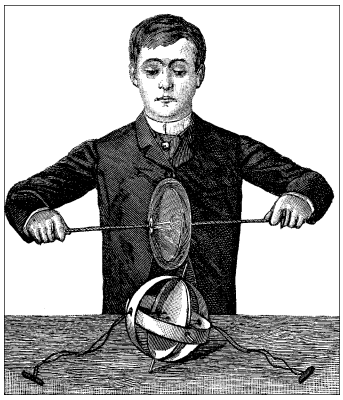
CHAPTER FOUR

DEBUGGING TECHNIQUES



One of the most compelling problems for anyone writing kernel code is how to approach debugging. 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 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>`:

`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 will 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. The default loglevel value has changed several times during Linux development, so we suggest that you always specify an explicit loglevel.

Based on the loglevel, the kernel may print the message to the current console, be it a text-mode terminal, a serial line printer, or a parallel printer. If the priority is less than the integer variable `console_loglevel`, the message is displayed. 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*.

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 the O'Reilly 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`) will reach the console; if it is set to 8, all messages, including debugging ones, will be displayed.

You'll probably want to lower the loglevel if you work on the console and you experience a kernel fault (see "Debugging System Faults" later in this chapter), because the fault-handling code raises the `console_loglevel` to its maximum value, causing every subsequent message to appear on the console. You'll want to raise the loglevel if you need to see your debugging messages; this is useful if you are developing kernel code remotely and the text console is not being used for an interactive session.

From version 2.1.31 on it is possible to read and modify the console loglevel using the text file */proc/sys/kernel/printk*. The file hosts four integer values. You may be interested in the first two: the current console loglevel and the default level for messages. With recent kernels, for instance, you can cause all kernel messages to appear at the console by simply entering

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

If you run 2.0, however, you still need the *setlevel* tool.

It should now be apparent why the *hello.c* sample had the `<1>` markers; they are there to make sure that the messages appear on the console.

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.

This is how the program works:

```
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` (defined in *kernel/printk.c*) bytes long. It 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, which is why it is the default behavior for *klogd*.

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. The logging process thus 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 modify */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'll implement a new feature in the driver (or somebody will find a bug) and you'll 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.
- 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)
```

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

```
#  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` depends on whether or not `SCULL_DEBUG` is defined, and it 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.

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, 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 minus.* 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.

Two main techniques are available to driver developers for querying the system: creating a file in the */proc* filesystem and using the *ioctl* driver method. You may use *devfs* as an alternative to */proc*, but */proc* is an easier tool to use for information retrieval.

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

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

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 will concern 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.

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 will reach your module by means of one of two different interfaces, according to what you registered. We'll leave registration for later in this section and jump directly to the description of the reading interfaces.

In both cases the kernel allocates a page of memory (i.e., `PAGE_SIZE` bytes) where the driver can write data to be returned to user space.

The recommended interface is *read_proc*, but an older interface named *get_info* also exists.

```
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 in the *read* implementation. 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.* The function is available in version 2.4 of the kernel, and 2.2 as well if you use our *sysdep.h* header.

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

get_info is an older interface used to read from a */proc* file. The arguments all have the same meaning as for *read_proc*. What it lacks is the pointer to report end-of-file and the object-oriented flavor brought in by the `data` pointer. The

* We'll find several of these pointers throughout the book; they represent the "object" involved in this action and correspond somewhat to `this` in C++.

function is available in all the kernel versions we are interested in (although it had an extra unused argument in its 2.0 implementation).

Both functions should return the number of bytes of data actually placed in the `page` buffer, just like the *read* implementation 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.

The main problem with the original implementation of user extensions to the */proc* filesystem was use of a single memory page for data transfer. This limited the total size of a user file to 4 KB (or whatever was appropriate for the host platform). The `start` argument is there to implement large data files, but it can be ignored.

If your *proc_read* function does not set the `*start` pointer (it starts out `NULL`), the kernel assumes that the `offset` parameter has been ignored and that the data page contains the whole file you want to return to user space. If, on the other hand, you need to build a bigger file from pieces, you can set `*start` to be equal to `page` so that the caller knows your new data is placed at the beginning of the buffer. You should then, of course, skip the first `offset` bytes of data, which will have already been returned in a previous call.

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 will use 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* or *get_info* 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 will provide that same value as the offset; the driver then knows to start returning data from the sixth structure in the array. This is defined as a "hack" by its authors and can be seen in *fs/proc/generic.c*.

Time for an example. Here is a simple *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++) {
        Scull_Dev *d = &scull_devices[i];
        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 (; d && len <= limit; d = d->next) { /* scan the list */
```

```
                len += sprintf(buf+len, "  item at %p, qset at %p\n", d,
                                        d->data);
                if (d->data && !d->next) /* dump only the last item
                                                    - save space */
                    for (j = 0; j < d->qset; j++) {
                        if (d->data[j])
                            len += sprintf(buf+len,"    % 4i: %8p\n",
                                                    j,d->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.

A */proc* function using the *get_info* interface would look very similar to the one just shown, with the exception that the last two arguments would be missing. The end-of-file condition, in this case, is signaled by returning less data than the caller expects (i.e., less than `count`).

Once you have a *read_proc* function defined, you need to connect it to an entry in the */proc* hierarchy. There are two ways of setting up this connection, depending on what versions of the kernel you wish to support. The easiest method, only available in the 2.4 kernel (and 2.2 too if you use our *sysdep.h* header), is to simply call *create_proc_read_entry*. 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 */);
```

The arguments to this function are, as shown, the name of the */proc* entry, the file permissions to apply to the entry (the value 0 is treated as a special case and is turned to a default, world-readable mask), the `proc_dir_entry` pointer to the parent directory for this file (we use `NULL` to make the driver appear directly under */proc*), the pointer to the *read_proc* function, and the data pointer that will be passed back to the *read_proc* function.

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 emerging 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* did:

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

The alternative method for creating a */proc* entry is to create and initialize a `proc_dir_entry` structure and pass it to *proc_register_dynamic* (version 2.0) or *proc_register* (version 2.2, which assumes a dynamic file if the inode number in the structure is 0). As an example, consider the following code that *scull* uses when compiled against 2.0 headers:

```
static int scull_get_info(char *buf, char **start, off_t offset,
                int len, int unused)
{
    int eof = 0;
    return scull_read_procmem (buf, start, offset, len, &eof, NULL);
}

struct proc_dir_entry scull_proc_entry = {
        namelen:    8,
        name:       "scullmem",
        mode:       S_IFREG | S_IRUGO,
        nlink:      1,
        get_info:   scull_get_info,
};

static void scull_create_proc()
{
    proc_register_dynamic(&proc_root, &scull_proc_entry);
}

static void scull_remove_proc()
{
    proc_unregister(&proc_root, scull_proc_entry.low_ino);
}
```

The code declares a function using the *get_info* interface and fills in a `proc_dir_entry` structure that is registered with the filesystem.

This code provides compatibility across the 2.0 and 2.2 kernels, with a little support from macro definitions in *sysdep.h*. It uses the *get_info* interface because the 2.0 kernel did not support *read_proc*. Some more work with `#ifdef` could have made it use *read_proc* with Linux 2.2, but the benefits would be minor.

The ioctl Method

ioctl, which we show you how to use in the next chapter, 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's code is 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, as well as 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 the last lines of running the command *strace ls /dev > /dev/scull0*:

```
[...]
open("/dev", O_RDONLY|O_NONBLOCK)     = 4
fcntl(4, F_SETFD, FD_CLOEXEC)         = 0
brk(0x8055000)                        = 0x8055000
lseek(4, 0, SEEK_CUR)                 = 0
getdents(4, /* 70 entries */, 3933)   = 1260
[...]
getdents(4, /* 0 entries */, 3933)    = 0
close(4)                              = 0
fstat(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(253, 0), ...}) = 0
ioctl(1, TCGETS, 0xbffffa5c)          = -1 ENOTTY (Inappropriate ioctl
                                                   for device)
write(1, "MAKEDEV\natibm\naudio\naudio1\na"..., 4096) = 4000
write(1, "d2\nsdd3\nsdd4\nsdd5\nsdd6\nsdd7"..., 96) = 96
write(1, "4\nsde5\nsde6\nsde7\nsde8\nsde9\n"..., 3325) = 3325
close(1)                              = 0
_exit(0)                              = ?
```

It's apparent in the first *write* call that after *ls* finished looking in the target directory, it tried to write 4 KB. Strangely (for *ls*), only four thousand 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)           = 4
fstat(4, {st_mode=S_IFCHR|0664, st_rdev=makedev(253, 0), ...}) = 0
```

```
read(4, "MAKEDEV\natibm\naudio\naudio1\na"..., 16384) = 4000
read(4, "d2\nsdd3\nsdd4\nsdd5\nsdd6\nsdd7"..., 16384) = 3421
read(4, "", 16384)                      = 0
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(3, 7), ...}) = 0
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
write(1, "   7421 /dev/scull0\n", 20)   = 20
close(4)                                = 0
_exit(0)                                = ?
```

As expected, *read* is able to retrieve only four thousand 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 thus 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.

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.

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.

Any address used by the processor is a virtual address and is mapped to physical addresses through a complex structure of so-called page tables (see "Page Tables" in Chapter 13). 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 generates an oops if this happens while the processor is in supervisor mode.

It's worth noting that the first enhancement introduced after version 2.0 was automatic handling of invalid address faults when moving data to and from user space. Linus chose to let the hardware catch erroneous memory references, so that the normal case (where the addresses are correct) is handled more efficiently.

An oops displays the processor status at the time of the fault, including the contents of the CPU registers, the location of page descriptor tables, 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 2.4 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:
c48370c3
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[<c48370c3>]
EFLAGS: 00010286
eax: fffffffea   ebx: c2281a20   ecx: c48370c0   edx: c2281a40
esi: 4000c000   edi: 4000c000   ebp: c38adf8c   esp: c38adf8c
ds: 0018   es: 0018   ss: 0018
Process ls (pid: 23171, stackpage=c38ad000)
Stack: 0000010e c01356e6 c2281a20 4000c000 0000010e c2281a40 c38ac000 \
          0000010e
       4000c000 bffffc1c 00000000 00000000 c38adfc4 c010b860 00000001 \
          4000c000
       0000010e 0000010e 4000c000 bffffc1c 00000004 0000002b 0000002b \
```

```
            00000004
Call Trace: [<c01356e6>] [<c010b860>]
Code: c7 05 00 00 00 00 00 00 00 00 31 c0 89 ec 5d c3 8d b6 00 00
```

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 *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 more interesting fault conditions in its *read* implementation:

```
char faulty_buf[1024];

ssize_t faulty_read (struct file *filp, char *buf, size_t count,
                     loff_t *pos)
{
    int ret, ret2;
    char stack_buf[4];

    printk(KERN_DEBUG "read: buf %p, count %li\n", buf, (long)count);
    /* the next line oopses with 2.0, but not with 2.2 and later */
    ret = copy_to_user(buf, faulty_buf, count);
    if (!ret) return count; /* we survived */

    printk(KERN_DEBUG "didn't fail: retry\n");
    /* For 2.2 and 2.4, let's try a buffer overflow  */
    sprintf(stack_buf, "1234567\n");
    if (count > 8) count = 8; /* copy 8 bytes to the user */
    ret2 = copy_to_user(buf, stack_buf, count);
    if (!ret2) return count;
    return ret2;
}
```

It first reads from a global buffer without checking the size of the data, and then performs a buffer overrun by writing to a local buffer. The first situation results in an oops only in version 2.0 of the kernel, because later versions automatically deal with user copy functions. The buffer overflow results in an oops with all kernel versions; however, 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 like the following:

```
EIP:    0010:[<00000000>]
[...]
Call Trace: [<c010b860>]
Code:  Bad EIP value.
```

The main problem with users dealing with oops messages is in the little intrinsic meaning carried by hexadecimal values; to be meaningful to the programmer they need to be resolved to symbols. A couple of utilities are available to perform this resolution for developers: *klogd* and *ksymoops*. The former tool performs symbol decoding by itself whenever it is running; the latter needs to be purposely invoked by the user. In the following discussion we use the data generated in our first oops example by dereferencing a `NULL` pointer.

Using klogd

The *klogd* daemon can decode oops messages before they reach the log files. In many situations, *klogd* can provide all the information a developer needs to track down a problem, though sometimes the developer must give it a little help.

A dump of the oops for *faulty*, as it reaches the system log, looks like this (note the decoded symbols on the `EIP` line and in the stack trace):

```
Unable to handle kernel NULL pointer dereference at virtual address \
     00000000
 printing eip:
c48370c3
*pde = 00000000
Oops: 0002
CPU:    0
EIP:    0010:[faulty:faulty_write+3/576]
EFLAGS: 00010286
eax: fffffffea   ebx: c2c55ae0   ecx: c48370c0   edx: c2c55b00
esi: 0804d038   edi: 0804d038   ebp: c2337f8c   esp: c2337f8c
ds: 0018   es: 0018   ss: 0018
Process cat (pid: 23413, stackpage=c2337000)
Stack: 00000001 c01356e6 c2c55ae0 0804d038 00000001 c2c55b00 c2336000 \
            00000001
       0804d038 bffffbd4 00000000 00000000 bffffbd4 c010b860 00000001 \
            0804d038
       00000001 00000001 0804d038 bffffbd4 00000004 0000002b 0000002b \
            00000004
Call Trace: [sys_write+214/256] [system_call+52/56]
Code: c7 05 00 00 00 00 00 00 00 00 31 c0 89 ec 5d c3 8d b6 00 00
```

klogd provides most of the necessary information to track down the problem. In this case we see that the instruction pointer (`EIP`) was executing in the function *faulty_write*, so we know where to start looking. The `3/576` string tells us that the processor was at byte 3 of a function that appears to be 576 bytes long. Note that the values are decimal, not hex.

The developer must exercise some care, however, to get useful information for errors that occur within loadable modules. *klogd* loads all of the available symbol information when it starts, and uses those symbols thereafter. If you load a module after *klogd* has initialized itself (usually at system boot), *klogd* will not have your module's symbol information. To force *klogd* to go out and get that information, send the *klogd* process a `SIGUSR1` signal after your module has been loaded (or reloaded), and before you do anything that could cause it to oops.

It is also possible to run *klogd* with the *–p* ("paranoid") option, which will cause it to reread symbol information anytime it sees an oops message. The *klogd* manpage recommends against this mode of operation, however, since it makes *klogd* query the kernel for information after the problem has occurred. Information obtained after an error could be plain wrong.

For *klogd* to work properly, it must have a current copy of the *System.map* symbol table file. Normally this file is found in */boot*; if you have built and installed a kernel from a nonstandard location you may have to copy *System.map* into */boot*, or tell *klogd* to look elsewhere. *klogd* refuses to decode symbols if the symbol table doesn't match the current kernel. If a symbol is decoded on the system log, you can be reasonably sure it is decoded correctly.

Using ksymoops

At times *klogd* may not be enough for your tracing purposes. Usually, you need to get both the hexadecimal address and the associated symbol, and you often need offsets printed as hex numbers. You may need more information than address decoding. Also, it is common for *klogd* to get killed during the fault. In such situations, a stronger oops analyzer may be called for; *ksymoops* is such a tool.

Prior to the 2.3 development series, *ksymoops* was distributed with the kernel source, in the *scripts* directory. It now lives on its own FTP site and is maintained independently of the kernel. Even if you are working with an older kernel, you probably should go to *ftp://ftp.ocs.com.au/pub/ksymoops* and get an updated version of the tool.

To operate at its best, *ksymoops* needs a lot of information in addition to the error message; you can use command-line options to tell it where to find the various items. The program needs the following items:

A System.map file
: This map must correspond to the kernel that was running at the time the oops occurred. The default is */usr/src/linux/System.map*.

A list of modules
: *ksymoops* needs to know what modules were loaded when the oops occurred, in order to extract symbolic information from them. If you do not supply this list, *ksymoops* will look at */proc/modules*.

A list of kernel symbols defined when the oops occurred
The default is to get this list from */proc/ksyms*.

A copy of the kernel image that was running
Note that *ksymoops* needs a straight kernel image, not the compressed version (*vmlinuz*, *zImage*, or *bzImage*) that most systems boot. The default is to use no kernel image because most people don't keep it. If you have the exact image handy, you should tell the program where it is by using the *-v* option.

The locations of the object files for any kernel modules that were loaded
ksymoops will look in the standard directories for modules, but during development you will almost certainly have to tell it where your module lives using the *-o* option

Although *ksymoops* will go to files in */proc* for some of its needed information, the results can be unreliable. The system, of course, will almost certainly have been rebooted between the time the oops occurs and when *ksymoops* is run, and the information from */proc* may not match the state of affairs when the failure occurred. When possible, it is better to save copies of */proc/modules* and */proc/ksyms* prior to causing the oops to happen.

We urge driver developers to read the manual page for *ksymoops* because it is a very informative document.

The last argument on the tool's command line is the location of the oops message; if it is missing, the tool will read `stdin` in the best Unix tradition. The message can be recovered from the system logs with luck; in the case of a very bad crash you may end up writing it down off the screen and typing it back in (unless you were using a serial console, a nice tool for kernel developers).

Note that *ksymoops* will be confused by an oops message that has already been processed by *klogd*. If you are running *klogd*, and your system is still running after an oops occurs, a clean oops message can often be obtained by invoking the *dmesg* command.

If you do not provide all of the listed information explicitly, *ksymoops* will issue warnings. It will also issue warnings about things like loaded modules that define no symbols. A warning-free run of *ksymoops* is rare.

Output from *ksymoops* tends to look like the following:

```
>>EIP; c48370c3 <[faulty]faulty_write+3/20>   <=====
Trace; c01356e6 <sys_write+d6/100>
Trace; c010b860 <system_call+34/38>
Code;  c48370c3 <[faulty]faulty_write+3/20>
00000000 <_EIP>:
Code;  c48370c3 <[faulty]faulty_write+3/20>   <=====
   0:   c7 05 00 00 00    movl   $0x0,0x0   <=====
Code;  c48370c8 <[faulty]faulty_write+8/20>
   5:   00 00 00 00 00
```

```
Code;  c48370cd <[faulty]faulty_write+d/20>
   a:   31 c0             xorl   %eax,%eax
Code;  c48370cf <[faulty]faulty_write+f/20>
   c:   89 ec             movl   %ebp,%esp
Code;  c48370d1 <[faulty]faulty_write+11/20>
   e:   5d                popl   %ebp
Code;  c48370d2 <[faulty]faulty_write+12/20>
   f:   c3                ret
Code;  c48370d3 <[faulty]faulty_write+13/20>
  10:   8d b6 00 00 00    leal   0x0(%esi),%esi
Code;  c48370d8 <[faulty]faulty_write+18/20>
  15:   00
```

As you can see, *ksymoops* provides `EIP` and kernel stack information much like *klogd* does, but more precisely and in hexadecimal. You'll note that the *faulty_write* function is correctly reported to be 0x20 bytes long. This is because *ksymoops* reads the object file of your module and extracts all available information.

In this case, moreover, you also get an assembly language dump of the code where the fault occurred. This information can often be used to figure out exactly what was happening; here it's clearly an instruction that writes a 0 to address 0.

One interesting feature of *ksymoops* is that it is ported to nearly all the platforms where Linux runs and exploits the *bfd* (binary format description) library in order to support several computer architectures at the same time. To step outside of the PC world, let's see how the same oops message appears on the *SPARC64* platform (several lines have been broken for typographical needs):

```
Unable to handle kernel NULL pointer dereference
tsk->mm->context = 0000000000000734
tsk->mm->pgd = fffff80003499000
              \/ ____
              "@'/ .. \`@"
              /_| \__/ |_\
                 \__U_/
ls(16740): Oops
TSTATE: 0000004400009601 TPC: 0000000001000128 TNPC: 0000000000457fbc \
Y: 00800000
g0: 000000007002ea88 g1: 0000000000000004 g2: 0000000070029fb0 \
g3: 0000000000000018
g4: fffff80000000000 g5: 0000000000000001 g6: fffff8000119c000 \
g7: 0000000000000001
o0: 0000000000000000 o1: 000000007001a000 o2: 0000000000000178 \
o3: fffff8001224f168
o4: 0000000001000120 o5: 0000000000000000 sp: fffff8000119f621 \
ret_pc: 0000000000457fb4
l0: fffff8001223376c0 l1: ffffffffffffffea l2: 000000000002c400 \
l3: 000000000002c400
l4: 0000000000000000 l5: 0000000000000000 l6: 0000000000019c00 \
l7: 0000000070028cbc
i0: fffff8001224f140 i1: 000000007001a000 i2: 0000000000000178 \
```

```
i3: 000000000002c400
i4: 000000000002c400 i5: 000000000002c000 i6: ffffff8000119f6e1 \
i7: 0000000000410114
Caller[0000000000410114]
Caller[000000007007cba4]
Instruction DUMP: 01000000 90102000 81c3e008 <c0202000> \
30680005 01000000 01000000 01000000 01000000
```

Note how the instruction dump doesn't start from the instruction that caused the fault but three instructions earlier: that's because the RISC platforms execute several instructions in parallel and may generate deferred exceptions, so one must be able to look back at the last few instructions.

This is what *ksymoops* prints when fed with input data starting at the `TSTATE` line:

```
>>TPC; 0000000001000128 <[faulty].text.start+88/a0>   <=====
>>O7;  0000000000457fb4 <sys_write+114/160>
>>I7;  0000000000410114 <linux_sparc_syscall+34/40>
Trace; 0000000000410114 <linux_sparc_syscall+34/40>
Trace; 000000007007cba4 <END_OF_CODE+6f07c40d/????>
Code;  000000000100011c <[faulty].text.start+7c/a0>
0000000000000000 <_TPC>:
Code;  000000000100011c <[faulty].text.start+7c/a0>
   0:   01 00 00 00       nop
Code;  0000000001000120 <[faulty].text.start+80/a0>
   4:   90 10 20 00       clr  %o0     ! 0 <_TPC>
Code;  0000000001000124 <[faulty].text.start+84/a0>
   8:   81 c3 e0 08       retl
Code;  0000000001000128 <[faulty].text.start+88/a0>   <=====
   c:   c0 20 20 00       clr  [ %g0 ]   <=====
Code;  000000000100012c <[faulty].text.start+8c/a0>
  10:   30 68 00 05       b,a   %xcc, 24 <_TPC+0x24> \
                         0000000001000140 <[faulty]faulty_write+0/20>
Code;  0000000001000130 <[faulty].text.start+90/a0>
  14:   01 00 00 00       nop
Code;  0000000001000134 <[faulty].text.start+94/a0>
  18:   01 00 00 00       nop
Code;  0000000001000138 <[faulty].text.start+98/a0>
  1c:   01 00 00 00       nop
Code;  000000000100013c <[faulty].text.start+9c/a0>
  20:   01 00 00 00       nop
```

To print the disassembled code shown we had to tell *ksymoops* the target file format and architecture (this is needed because the native architecture for *SPARC64* user space is 32 bit). In this case, the options *-t elf64-sparc -a sparc:v9* did the job.

You may complain that this call trace doesn't carry any interesting information; however, the *SPARC* processors don't save all the call trace on the stack: the `O7` and `I7` registers hold the instruction pointers of the last two calling functions, which is why they are shown near the call trace. In this case, the faulty instruction was in a function invoked by *sys_write*.

Note that, whatever the platform/architecture pair, the format used to show disassembled code is the same as that used by the *objdump* program. *objdump* is a powerful utility; if you want to look at the whole function that failed, you can invoke the command *objdump –d faulty.o* (once again, on *SPARC64*, you need special options: *—target elf64-sparc—architecture sparc:v9*). For more information on *objdump* and its command-line options, see the manpage for the command.

Learning to decode an oops message requires some practice and an understanding of the target processor you are using, as well as of the conventions used to represent assembly language, but it's worth doing. The time spent learning will be quickly repaid. Even if you have previous expertise with the PC assembly language under non-Unix operating systems, you may need to devote some time to learning, because the Unix syntax is different from Intel syntax. (A good description of the differences is in the Info documentation file for *as*, in the chapter called "i386-specific.")

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 thus 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 is to add some print messages and write them to the console (by changing the `console_loglevel` value).

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. If you are not using a graphic display, you can check the scheduler by running a program that

flashes the keyboard LEDs, turns on the floppy motor every now and then, or ticks the speaker—conventional beeps are quite annoying and should be avoided; look for the `KDMKTONE` *ioctl* command instead. A sample program (*misc-progs/heartbeat.c*) that flashes a keyboard LED in a heartbeat fashion is available in the sources on the O'Reilly FTP site.

If the keyboard isn't accepting input, the best thing to do is log into the system through your network and kill any offending processes, or reset the keyboard (with *kbd_mode –a*). However, discovering that the hang is only a keyboard lockup is of little use if you don't have a network available to help you recover. If this is the case, you could set up alternative input devices to be able at least to reboot the system cleanly. A shutdown and reboot cycle is easier on your computer than hitting the so-called big red button, and it saves you from the lengthy *fsck* scanning of your disks.

Such an alternative input device can be, for example, the mouse. Version 1.10 or newer of the *gpm* mouse server features a command-line option to enable a similar capability, but it works only in text mode. If you don't have a network connection and run in graphics mode, we suggest running some custom solution, like a switch connected to the DCD pin of the serial line and a script that polls for status change.

An indispensable tool for these situations is the "magic SysRq key," which is available on more architectures in 2.2 and later kernels. Magic SysRq is invoked with the combination of the ALT and SysRq keys on the PC keyboard, or with the ALT and Stop keys on SPARC keyboards. A third key, pressed along with these two, performs one of a number of useful actions, as follows:

`r` Turns off keyboard raw mode in situations where you cannot run *kbd_mode*.

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

`s` Performs an emergency synchronization of all disks.

`u` 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.

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

`p` Prints the current register 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 must be enabled at runtime with a command like the following:

```
echo 1 > /proc/sys/kernel/sysrq
```

Another precaution to use when reproducing 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 any 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 the 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 uncompressed 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 kernel executable, not the *zImage* or *bzImage* or anything compressed.

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 "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 prepares 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.

If you compile the kernel with debugging support (*–g*), the resulting *vmlinux* file turns out to work better with *gdb* than the same file compiled without *–g*. Note, however, that a large amount of disk space is needed to compile the kernel with the *–g* option (each object file and the kernel itself are three or more times bigger than usual).

On non-PC computers, the game is different. On the Alpha, *make boot* strips the kernel before creating the bootable image, so you end up with both the *vmlinux* and the *vmlinux.gz* files. The former is usable by *gdb*, and you can boot from the latter. On the SPARC, the kernel (at least the 2.0 kernel) is not stripped by default.

When you compile the kernel with *–g* and run the debugger using *vmlinux* together with */proc/kcore*, *gdb* can return a lot of information about the kernel internals. You can, for example, use commands such as *p *module_list*, *p *module_list->next*, and *p *chrdevs[4]->fops* to dump structures. To get the best out of *p*, you'll need to keep a kernel map and the source code handy.

Another useful task that *gdb* performs on the running kernel is disassembling functions, via the *disassemble* command (which can be abbreviated to *disass*) or the "examine instructions" (*x/i*) command. The *disassemble* command can take as its argument either a function name or a memory range, whereas *x/i* takes a single memory address, also in the form of a symbol name. You can invoke, for example, *x/20i* to disassemble 20 instructions. Note that you can't disassemble a module function, because the debugger is acting on *vmlinux*, which doesn't know about your module. If you try to disassemble a module by address, *gdb* is most likely to reply "Cannot access memory at xxxx." For the same reason, you can't look at data items belonging to a module. They can be read from */dev/mem* if you know the address of your variables, but it's hard to make sense out of raw data extracted from system RAM.

If you want to disassemble a module function, you're better off running the *objdump* utility on the module object file. Unfortunately, the tool runs on the disk copy of the file, not the running one; therefore, the addresses as shown by *objdump* will be the addresses before relocation, unrelated to the module's execution environment. Another disadvantage of disassembling an unlinked object file is that function calls are still unresolved, so you can't easily tell a call to *printk* from a call to *kmalloc*.

As you see, *gdb* is a useful tool when your aim is to peek into the running kernel, but it lacks some features that are vital to debugging device drivers.

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. Hitting the Pause (or Break) key on the console will start up the debugger. *kdb* also starts up when a kernel oops happens, or when a breakpoint is hit. In any case, you will see a message that looks something like this:

```
Entering kdb (0xc1278000) on processor 1 due to Keyboard Entry
[1]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:

```
[1]kdb> bp scull_read
Instruction(i) BP #0 at 0xc8833514 (scull_read)
    is enabled on cpu 1
[1]kdb> go
```

The *bp* command tells *kdb* to stop the next time the kernel enters *scull_read*. We 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:

```
Entering kdb (0xc3108000) on processor 0 due to Breakpoint @ 0xc8833515
Instruction(i) breakpoint #0 at 0xc8833514
scull_read+0x1:   movl   %esp,%ebp
[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
    EBP       EIP         Function(args)
0xc3109c5c 0xc8833515  scull_read+0x1
0xc3109fbc 0xfc458b10  scull_read+0x33c255fc( 0x3, 0x803ad78, 0x1000,
0x1000, 0x804ad78)
0xbffffc88 0xc010bec0  system_call
[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. Thus it prints five arguments for *scull_read*, which only has four.

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

```
[0]kdb> mds scull_devices 1
c8836104: c4c125c0 ....
```

Here we asked for one (four-byte) word of data starting at the location of `scull_devices`; the answer tells us that our device array was allocated starting at the address `c4c125c0`. To look at a device structure itself we need to use that address:

```
[0]kdb> mds c4c125c0
c4c125c0: c3785000  ....
c4c125c4: 00000000  ....
c4c125c8: 00000fa0  ....
c4c125cc: 000003e8  ....
c4c125d0: 0000009a  ....
c4c125d4: 00000000  ....
c4c125d8: 00000000  ....
c4c125dc: 00000001  ....
```

The eight lines here correspond to the eight fields in the `Scull_Dev` structure. Thus we see that the memory for the first device is allocated at `0xc3785000`, that there is no next item in the list, that the quantum is 4000 (hex fa0) and the array size is 1000 (hex 3e8), that there are 154 bytes of data in the device (hex 9a), and so on.

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

```
[0]kdb> mm c4c125d0 0x50
0xc4c125d0 = 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 Integrated Kernel Debugger Patch

A number of kernel developers have contributed to an unofficial patch called the *integrated kernel debugger*, or IKD. IKD provides a number of interesting kernel debugging facilities. The x86 is the primary platform for this patch, but much of it works on other architectures as well. As of this writing, the IKD patch can be found at *ftp://ftp.kernel.org/pub/linux/kernel/people/andrea/ikd*. It is a patch that must be applied to the source for your kernel; the patch is version specific, so be sure to download the one that matches the kernel you are working with.

One of the features of the IKD patch is a kernel stack debugger. If you turn this feature on, the kernel will check the amount of free space on the kernel stack at every function call, and force an oops if it gets too small. If something in your kernel is causing stack corruption, this tool may help you to find it. There is also a "stack meter" feature that you can use to see how close to filling up the stack you get at any particular time.

The IKD patch also includes some tools for finding kernel lockups. A "soft lockup" detector forces an oops if a kernel procedure goes for too long without scheduling. It is implemented by simply counting the number of function calls that are made and shutting things down if that number exceeds a preconfigured threshold. Another feature can continuously print the program counter on a virtual console for truly last-resort lockup tracking. The semaphore deadlock detector forces an oops if a process spends too long waiting on a *down* call.

Other debugging capabilities in IKD include the kernel trace capability, which can record the paths taken through the kernel code. There are some memory debugging tools, including a leak detector and a couple of "poisoners," that can be useful in tracking down memory corruption problems.

Finally, IKD also includes a version of the *kdb* debugger discussed in the previous section. As of this writing, however, the version of *kdb* included in the IKD patch is somewhat old. If you need *kdb*, we recommend that you go directly to the source at *oss.sgi.com* for the current version.

The kgdb Patch

kgdb is a patch that allows the full use of the *gdb* debugger on the Linux kernel, but only on x86 systems. It works by hooking into the system to be debugged via a serial line, with *gdb* running on the far end. You thus need two systems to use *kgdb*—one to run the debugger and one to run the kernel of interest. Like *kdb*, *kgdb* is currently available from *oss.sgi.com*.

Setting up *kgdb* involves installing a kernel patch and booting the modified kernel. You need to connect the two systems with a serial cable (of the null modem variety) and to install some support files on the *gdb* side of the connection. The patch places detailed instructions in the file *Documentation/i386/gdb-serial.txt*; we won't reproduce them here. Be sure to read the instructions on debugging modules: toward the end there are some nice *gdb* macros that have been written for this purpose.

Kernel Crash Dump Analyzers

Crash dump analyzers enable the system to record its state when an oops occurs, so that it may be examined at leisure afterward. They can be especially useful if you are supporting a driver for a user at a different site. Users can be somewhat reluctant to copy down oops messages for you so installing a crash dump system can let you get the information you need to track down a user's problem without requiring work from him. It is thus not surprising that the available crash dump analyzers have been written by companies in the business of supporting systems for users.

There are currently two crash dump analyzer patches available for Linux. Both were relatively new when this section was written, and both were in a state of flux. Rather than provide detailed information that is likely to go out of date, we'll restrict ourselves to providing an overview and pointers to where more information can be found.

The first analyzer is LKCD (Linux Kernel Crash Dumps). It's available, once again, from *oss.sgi.com*. When a kernel oops occurs, LKCD will write a copy of the current system state (memory, primarily) into the dump device you specified in advance. The dump device must be a system swap area. A utility called *LCRASH* is run on the next reboot (before swapping is enabled) to generate a summary of the crash, and optionally to save a copy of the dump in a conventional file. *LCRASH* can be run interactively and provides a number of debugger-like commands for querying the state of the system.

LKCD is currently supported for the Intel 32-bit architecture only, and only works with swap partitions on SCSI disks.

Another crash dump facility is available from *www.missioncriticallinux.com*. This crash dump subsystem creates crash dump files directly in */var/dumps* and does not use the swap area. That makes certain things easier, but it also means that the system will be modifying the file system while in a state where things are known to have gone wrong. The crash dumps generated are in a standard core file format, so tools like *gdb* can be used for post-mortem analysis. This package also provides a separate analyzer that is able to extract more information than *gdb* from the crash dump files.

The User-Mode Linux Port

User-Mode Linux 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, User-Mode Linux 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. After all, it is just another process. User-Mode Linux clearly has the potential to accelerate kernel development.

As of this writing, User-Mode Linux is not distributed with the mainline kernel; it must be downloaded from its web site (*http://user-mode-linux.sourceforge.net*). The word is that it will be integrated into an early 2.4 release after 2.4.0; it may well be there by the time this book is published.

User-Mode Linux also has some significant limitations as of this writing, most of which will likely be addressed soon. The virtual processor currently works in a uniprocessor mode only; the port runs on SMP systems without a problem, but it can only emulate a uniprocessor host. The biggest problem for driver writers, though, is that 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, User-Mode Linux is not yet useful for debugging drivers that have to deal with real hardware. Finally, User-Mode Linux only runs on the IA-32 architecture.

Because work is under way to fix all of these problems, User-Mode Linux will likely be an indispensable tool for Linux device driver programmers in the very near future.

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 on the Web at *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 Linux Trace Toolkit to insert new tracing events at arbitrary locations.

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