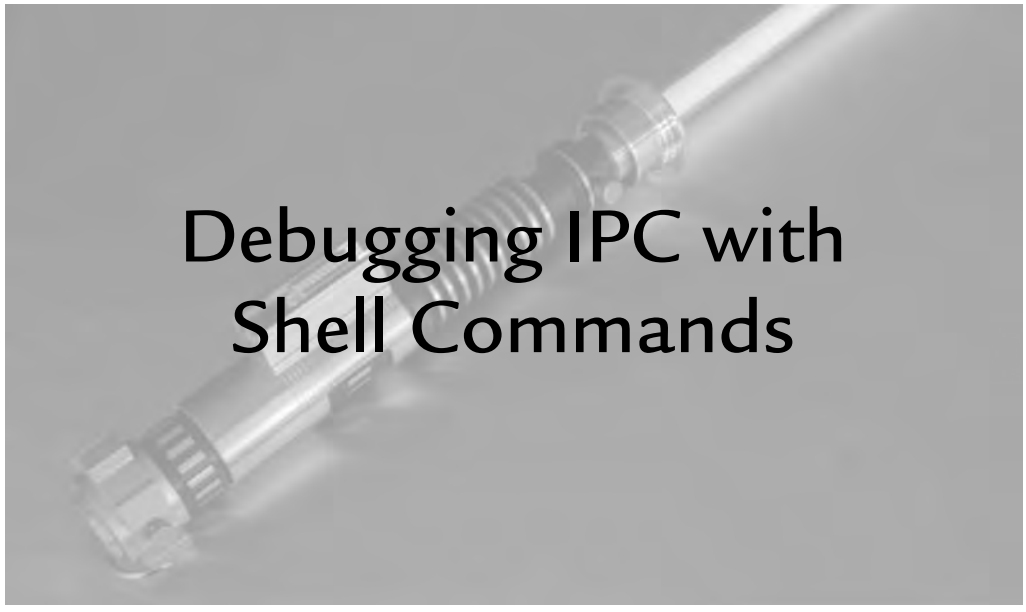
Chapter 8



Debugging IPC with Shell Commands

8.1 Introduction

In this chapter, I look at techniques and commands you can use from the shell for debugging interprocess communication (IPC). When you are debugging communication between processes, it's always nice to have a neutral third party to intervene when things go wrong.

8.2 Tools for Working with Open Files

Processes that leave files open can cause problems. File descriptors can be "leaked" like memory, for example, consuming resources unnecessarily. Each process has a finite number of file descriptors it may keep open, so if some broken code continues to open file descriptors without closing them, eventually it will fail with an `errno` value of `EMFILE`. If you have some thoughtful error handling in your code, it will be obvious what has happened. But then what?

The `procfs` file system is very useful for debugging such problems. You can see all the open files of a particular process in the directory `/proc/PID/fd`. Each open file here shows up as a symbolic link. The name of the link is the file descriptor number, and the link points to the open file. Following is an example:

```
$ stty tostop                                   Force background task to stop on output.
$ echo hello | cat ~/.bashrc 2>/dev/null &      Run cat in the background.
[1] 16894                                       It's stopped.
$ ls -l /proc/16894/fd                          Let's see what files it has open.
total 4
lr-x------  1 john john 64 Apr  9 12:15 0 -> pipe:[176626]
lrwx------  1 john john 64 Apr  9 12:15 1 -> /dev/pts/2
l-wx------  1 john john 64 Apr  9 12:15 2 -> /dev/null
lr-x------  1 john john 64 Apr  9 12:15 3 -> /home/john/.bashrc
```

Here, I piped the output of `echo` to the `cat` command, which shows up as a pipe for file descriptor zero (standard input). The standard output points to the current terminal, and I redirected the standard error (file descriptor 2) to `/dev/null`. Finally, the file I am trying to print shows up in file descriptor 3. All this shows fairly clearly in the output.

8.2.1 lsof

You can see a more comprehensive listing by using the `lsof` command. With no arguments, `lsof` will show all open files in the system, which can be overwhelming. Even then, it will show you only what you have permission to see. You can restrict output to a single process with the `-p` option, as follows:

```
$ lsof -p 16894
COMMAND   PID USER   FD   TYPE DEVICE     SIZE   NODE NAME
cat     16894 john  cwd    DIR  253,0     4096 575355 /home/john
cat     16894 john  rtd    DIR  253,0     4096      2 /
cat     16894 john  txt    REG  253,0    21104 159711 /bin/cat
cat     16894 john  mem    REG  253,0   126648 608855 /lib/ld-2.3.5.so
cat     16894 john  mem    REG  253,0  1489572 608856 /lib/libc-2.3.5.so
cat     16894 john  mem    REG    0,0           0 [heap]
cat     16894 john  mem    REG  253,0 48501472 801788 .../locale-archive
cat     16894 john    0r  FIFO    0,5          176626 pipe
cat     16894 john    1u   CHR  136,2             4 /dev/pts/2
cat     16894 john    2w   CHR    1,3          1510 /dev/null
cat     16894 john    3r   REG  253,0      167 575649 /home/john/.bashrc
```

This output shows not only file descriptors, but memory-mapped files as well. The `FD` heading tells you whether the output is a file descriptor or a mapping. A mapping does not require a file descriptor after `mmap` has been called, so the `FD`

column includes some text for each mapping to indicate the type of mapping. File descriptors are shown by number as well as the type of access, as summarized in Table 8-1.

You also can use `lsof` to discover which process has a particular file open by providing the filename as an argument. There are many more options to the `lsof` command; see `lsof(8)` for details.

8.2.2 fuser

Another utility for tracking down open files is the `fuser` command. Suppose that you need to track down a process that is writing a huge file that is filling up your file system. You could use `fuser` as follows:

```
$ fuser some-huge-file.txt              What process has this file open?
some-huge-file.txt:  17005
```

If that's all you care about, you could go ahead and kill the process. `fuser` allows you to do this with the `-k` option as follows:

```
]$ fuser -k -KILL some-huge-file.txt
some-huge-file.txt:  17005
[1]+  Killed                  cat some-huge-file.txt
```

TABLE 8-1 Text Used in the FD Column of lsof Output

Identifier	Meaning
`cwd`	Current working directory
`ltx`	Shared library text (code and data)
`mem`	Memory-mapped file
`mmap`	Memory-mapped device
`pd`	Parent directory
`rtd`	Root directory
`txt`	Program text (code and data)
`{digit}r`	File descriptor opened read-only
`{digit}w`	File descriptor opened write-only
`{digit}u`	File descriptor opened read/write.

This sends the `SIGKILL` signal to any and all processes that have this file open. Another time `fuser` comes in handy is when you are trying to unmount a file system but can't because a process has a file open. In this case, the `-m` option is very helpful:

```
$ fuser -m /mnt/flash                    What process has files open on this file system?
/mnt/flash:          17118
```

Now you can decide whether you want to kill the process or let it finish what it needs to do. `fuser` has more options that are documented in the `fuser(1)` `man` page.

8.2.3 ls

You will be interested in the *long* listing available with the `-l` option. No doubt you are aware that this gives you the filename, permissions, and size of the file. The output also tells you what kind of file you are looking at. For example:

```
$ ls -l /dev/log /dev/initctl /dev/sda /dev/zero
prw-------  1 root root    0 Oct  8 09:13 /dev/initctl      A pipe (p)
srw-rw-rw-  1 root root    0 Oct  8 09:10 /dev/log          A socket (s)
brw-r-----  1 root disk 8, 0 Oct  8 04:09 /dev/sda          A block device (b)
crw-rw-rw-  1 root root 1, 5 Oct  8 04:09 /dev/zero         A char device (c)
```

For files other than plain files, the first column indicates the type of file you are looking at. You can also use the `-F` option for a more concise listing that uses unique suffixes for special files:

```
$ ls -F /dev/log /dev/initctl /dev/zero /dev/sda
/dev/initctl|  /dev/log=  /dev/sda  /dev/zero
```

A pipe is indicated by adding a `|` to the filename, and a socket is indicated by adding a `=` to the filename. The `-F` option does not use any unique character to identify block or character devices, however.

8.2.4 file

This simple utility can tell you in a very user-friendly way the type of file you are looking at. For example:

```
file /dev/log /dev/initctl /dev/sda /dev/zero
/dev/log:     socket
/dev/initctl: fifo (named pipe)
/dev/sda:     block special (8/0)          Includes major/minor numbers
/dev/zero:    character special (1/5)      Includes major/minor numbers
```

Each file is listed with a simple, human-readable description of its type. The `file` command can also recognize many plain file types, such as ELF files and image files. It maintains an extensive database of magic numbers to recognize file types. This database can be extended by the user as well. See `file(1)` for more information.

8.2.5 stat

The `stat` command is a wrapper for the `stat` system that can be used from the shell. The output consists of all the data you would get from the `stat` system call in human-readable format. For example:

```
stat /dev/sda
  File: `/dev/sda'
  Size: 0               Blocks: 0          IO Block: 4096   block special file
Device: eh/14d  Inode: 1137        Links: 1     Device type: 8,0
Access: (0640/brw-r-----)  Uid: (    0/    root)   Gid: (    6/    disk)
Access: 2006-10-08 04:09:34.750000000 -0500
Modify: 2006-10-08 04:09:34.750000000 -0500
Change: 2006-10-08 04:09:50.000000000 -0500
```

`stat` also allows formatting like the `printf` function, using specially defined format characters defined in the `stat(1) man` page. To see only the name of each file followed by its access rights in human-readable form and octal, you could use the following command:

```
stat --format="%-15n %A,%a" /dev/log /dev/initctl /dev/sda /dev/zero
/dev/log        srw-rw-rw-,666
/dev/initctl    prw-------,600
/dev/sda        brw-r-----,640
/dev/zero       crw-rw-rw-,666
```

`stat` can be very useful in scripts to monitor particular files on disk. During debugging, such scripts can act like watchdogs. You can watch a UNIX socket to look for periods of inactivity as follows:

```
while [ true ]; do
        ta=$(stat -c %X $filename)    # Time of most recent activity
        tnow=$(date +%s)              # Current time

        if [ $(($tnow - $ta)) -gt 5 ]; then
                echo No activity on $filename in the last 5 seconds.
        fi
        sleep 1
done
```

In this example, the script checks a file every second for the most recent access to the file, which is given with the `%X` format option to `stat`. Whenever a process writes to the socket, the time is updated, so the difference between the current time and the time from the `stat` command is the amount of elapsed time (in seconds) since the last write or read from the socket.

8.3 Dumping Data from a File

You probably are familiar with a few tools for this purpose, including your favorite text editor for looking at text files. All the regular text processing tools are at your disposal for working with ASCII text files. Some of these tools have the ability to work with additional encodings—if not through a command-line option, maybe via the locale setting. For example:

```
$ wc -w konnichiwa.txt                          Contains the Japanese phrase "konnichiwa" (one word).
0 konnichiwa.txt                                wc reports 0 words based on current locale.
$ LANG=ja_JP.UTF-8 wc -w konnichiwa.txt
1 konnichiwa.txt                                Forced Japanese locale gives us the correct answer.
```

Several tools can help with looking at binary data, but not all of them help interpret the data. To appreciate the differences among tools, you'll need an example (Listing 8-1).

LISTING 8-1 filedat.c: A Program That Creates a Data File with Mixed Formats

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 const char message[] = { // UTF-8 message
 5     0xbf, 0xe3, 0x81, 0x93, 0xe3, 0x82, 0x93, 0xe3, 0x81, 0xab,
 6     0xe3, 0x81, 0xa1, 0xe3, 0x81, 0xaf, '\r', 0x20, 0x20, 0x20,
 7     0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x0a, 0x0a, 0
 8 };
 9
10 int main(int argc, char *argv[])
11 {
12     const char *filename = "floats-ints.dat";
13     FILE *fp = fopen(filename, "wb");
14
15     /* error checking omitted. */
16
17     fprintf(fp, "Hello World\r%12s\n", "");
18     fwrite(message, sizeof(message), 1, fp);
19
20     /* write 250 zeros to the file. */
```

```
21      char *zeros = calloc(250, 1);
22      fwrite(zeros, 250, 1, fp);
23
24      int i;
25
26      /* Write four ints to the file 90000, 90001, ... */
27      for (i = 0; i < 4; i++) {
28          int idatum = i + 90000;
29          fwrite((char *) &idatum, sizeof(idatum), 1, fp);
30      }
31
32      /* Write four floats to the file 90000, 90001, ... */
33      for (i = 0; i < 4; i++) {
34          float fdatum = (float) i + 90000.0;
35          fwrite((char *) &fdatum, sizeof(fdatum), 1, fp);
36      }
37      printf("wrote %s\n", filename);
38      fclose(fp);
39  }
```

Listing 8-1 creates a file that contains a mix of ASCII, UTF-8, and binary data. The binary data is in native integer format (32 bits on my machine) and IEEE float (also 32 bits). A simple `cat` command produces nothing but garbage:

```
$ ./filedat
wrote floats-ints.dat
$ cat floats-ints.dat
```

```
floats-ints.dat____È⁻GÈ⁻GÉ⁻GÉ⁻G$
```

Not even "Hello World" is printed!

The problem, of course, is that `cat` just streams bytes out to the terminal, which then interprets those bytes as whatever encoding the locale is using. In the "`Hello World`" string on line 17 of Listing 8-1, I included a carriage return followed by 12 spaces. This has the effect of writing "`Hello World`" but then overwriting it with 12 spaces, which effectively makes the string invisible on the terminal.

You could use a text editor on this file, but the results may vary based on your text editor. Earlier, I looked at the `bvi` editor, which is a Vi clone for files with binary data.

Figure 8-1 shows that `bvi` does a good job of representing raw bytes and ASCII strings, and even lets you modify the data, but it is not able to represent data encoded in UTF-8, IEEE floats, or native integers. For that, you'll need other tools.

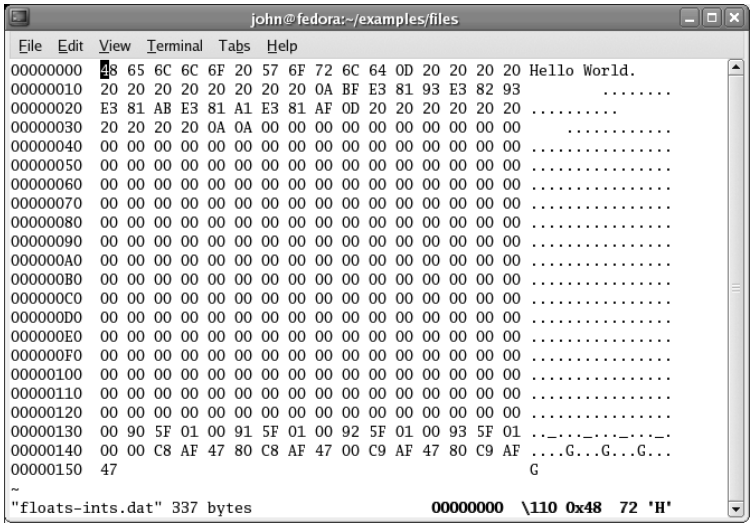


FIGURE 8-1 The Output from Listing 8-1 As Seen in `bvi`

8.3.1 The strings Command

Often, the text strings in a data file can give you a clue as to its contents. Sometimes, the text can tell you all you need to know. When the text is embedded in a bunch of binary data, however, you need something better than a simple `cat` command.

Looking back at the output of Listing 8-1, you can use the `strings` command to look at the text strings in this data:

```
$ strings floats-ints.dat
Hello World
                                        Invisible characters? Newlines? Who knows?
$
```

Now you can see `Hello World` and the spaces, but something is still missing. Remember that `message` array on line 18? It's actually UTF-8 text I encoded in binary. `strings` can look for 8-bit encodings (that is, non-ASCII) when you use the `-e` option as follows:

```
$ strings -eS floats-ints.dat          Tell strings to look for 8-bit encodings (-eS)

Hello World

こんにちは                              Japanese "konnichiwa," "good day" in UTF-8

ôG ôG                                  Our floats and ints produce this gobbledygook.
wG wG
```

The example above shows that the UTF-8 output is in Japanese, but I glossed over one detail: To show this on your screen, your terminal must support UTF-8 characters. Technically, you also need the correct font to go with it, but it seems that most UTF-8 font sets have the Hiragana[1] characters required for the message above. With `gnome-terminal`, you can get the required support by setting the character encoding to UTF-8. This is visible below Terminal on the menu bar. Not every terminal supports UTF-8; check your documentation.

By default, `strings` limits the output to strings of four characters or more; anything smaller is ignored. You can override this with the `-n` option, which indicates the smallest string to look for. To see the binary data in your file, you will need other tools.

8.3.2 The xxd Command

`xxd` is part of Vim and produces output very similar to `bvi`. The difference is that `xxd` is not a text editor. Like `bvi`, `xxd` shows data in hexadecimal and shows only ASCII characters:

```
$ xxd floats-ints.dat
0000000: 4865 6c6c 6f20 576f 726c 640d 2020 2020  Hello World.
0000010: 2020 2020 2020 2020 0abf e381 93e3 8293          ........
0000020: e381 abe3 81a1 e381 af0d 2020 2020 2020  ..........
0000030: 2020 2020 0a0a 0000 0000 0000 0000 0000      ............
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000130: 0090 5f01 0091 5f01 0092 5f01 0093 5f01  ..._..._..._..._.
0000140: 0000 c8af 4780 c8af 4700 c9af 4780 c9af  ....G...G...G...
0000150: 47                                       G
```

1. Hiragana is one of three sets of characters required to render Japanese text.

`xxd` defaults to 16-bit words, but you can adjust this with the `-g` option. To see the data in groups of 4 bytes, for example, use `-g4`. Make sure, however, that the groups preserve the byte order in the file. This means that 32-bit words printed on an IA32 will be incorrect. IA32 stores words with the least significant byte first, which is the reverse of the byte order in memory. This is sometimes called *Little Endian* byte order. To display the correct words, you must reverse the order of the bytes, which `xxd` does not do.

This can come in handy on some occasions. If you need to look at *Big Endian* data on a Little Endian machine, for example, you do not want to rearrange the bytes. Network protocols use the so-called network byte order for data transfer, which happens to be the same as Big Endian. So if you happen to be looking at a file that contains protocol headers from a socket, you would want a tool like `xxd` that does not swap the bytes.

8.3.3 The hexdump Command

As the name suggests, `hexdump` allows you to dump a file's contents in hexadecimal. As with `xxd`, the default format from `hexdump` is 16-bit hexadecimal, however, the byte order is adjusted on Little Endian architectures, so the output can differ between `xxd` and `hexdump`.

`hexdump` is better suited for terminal output than `xxd` because `hexdump` eliminates duplicate lines of data skipped to avoid cluttering the screen. `hexdump` can produce many other output formats besides 16-bit hexadecimal, but using them can difficult. Because the `hexdump(1)` `man` page does such a rotten job of explaining this feature, here's an example using 32-bit hexadecimal output:

```
$ hexdump -e '6/4 "%8X "' -e '"\n"'  floats-ints.dat
6C6C6548 6F57206F  D646C72 20202020 20202020 20202020
81E3BF0A 9382E393 E3AB81E3 81E3A181 20200DAF 20202020
20202020      A0A        0        0        0        0
       0        0        0        0        0        0
*
       0        0        0        0  15F9000  15F9100
 15F9200  15F9300 AFC80000 AFC88047 AFC90047 AFC98047
      47
```

Notice that I included two `-e` options. The first tells `hexdump` that I want 6 values per line, each with a width of 4 bytes (32 bits). Then I included a space, followed by the `printf`-like format in double quotes. `hexdump` looks for the double

quotes and spaces in the format arguments, and will complain if it does not find them. That is why I needed to enclose the entire expression in single quotes.

Still looking at this first argument, I had to include a space following the `%8X` to separate the values. I could have used a comma or semicolon or whatever, but `hexdump` interprets this format verbatim. If you neglect to include a separator, all the digits will appear as one long string.

Finally, I told `hexdump` how to separate each line of output (every six words) by including a second `-e` option, which for some reason must be enclosed in double quotes. If you can't tell, I find `hexdump` to be a nuisance to use, but many programmers use it. The alternatives to `hexdump` are `xxd` and `od`.

8.3.4 The od Command

`od` is the traditional UNIX *octal dump* command. Despite the name, `od` is capable of representing data in many other formats and word sizes. The `-t` option is the general-purpose switch for changing the output data type and element size (although there are aliases based on legacy options). You can see the earlier text file as follows:

```
$ od -tc floats-ints.dat                          Output data as ASCII characters
0000000   H   e   l   l   o       W   o   r   l   d  \r
0000020                                       \n 277 343 201 223 343 202 223
0000040 343 201 253 343 201 241 343 201 257  \r
0000060                 \n  \n  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0

*         Duplicate lines are skipped (indicated with "*").

0000460  \0 220   _ 001  \0 221   _ 001  \0 222   _ 001  \0 223   _ 001
0000500  \0  \0 310 257   G 200 310 257   G  \0 311 257   G 200 311 257
0000520   G
0000521
```

This output is comparable to what you've already seen with other tools. By default, the offsets on the left are printed in octal (in keeping with the name). You can change the base of the offsets with the `-A` option. `-Ax`, for example, prints the offsets in hexadecimal.

`od`'s treatment of strings is similar to that of `xxd` and `bvi`. It recognizes ASCII for display on the terminal but treats everything else as raw binary. What `od` can do that the others can't is rearrange the bytes *when necessary* to represent data in native format. Recall that the data from Listing 8-1 has IEEE floats and integers in the

data. To see the integers in decimal, you can use the `-td` option, but you must tell `od` where the data starts. In this case, the float data starts at offset `0x131` in the file, so use the `-j` option as follows:

```
$ od -td4 -j0x131 floats-ints.dat                          Show me 4-byte words in decimal.
0000461       90000       90001       90002       90003
0000501  1202702336  1202702464  1202702592  1202702720  Float data (gibberish)
0000521
```

Now you can see the four consecutive decimal numbers we stored, starting with `90000`. If you do not specify the offset to the data, the output will be incorrect. The float and integer data in this case starts on an odd boundary. The float data starts at offset `0x141`, so you must use the `-j` option again to see your floats:

```
$ od -tf4 -j0x141 floats-ints.dat
0000501   9.000000e+04   9.000100e+04   9.000200e+04   9.000300e+04
0000521
```

I stored four consecutive float values starting with `90000`. Notice that in this case, I qualified the type as `-tf4`. I used IEEE floats in the program, which are 4 bytes each. The default for the `-tf` option is to display IEEE doubles, which are 8 bytes each. If you do not specify IEEE floats, you would see garbage.

Note that `od` adjusts the byte order only when necessary. As long as your data is in native byte order, `od` will produce correct results. If you are looking at data that you know is in network byte order (that is, Big Endian), `od` will show you incorrect answers on a Little Endian machine such as IA32.

8.5 Tools for Working with POSIX IPC

POSIX IPC uses file descriptors for every object. The POSIX pattern is that every file descriptor has a file or device associated with it, and Linux extends this with special file systems for IPC. Because each IPC object can be traced to a plain file, the tools we use for working with plain files are often sufficient for working with POSIX IPC objects.

8.5.1 POSIX Shared Memory

There are no tools specifically for POSIX shared memory. In Linux, POSIX shared memory objects reside on the `tmpfs` pseudo file system, which typically is mounted on `/dev/shm`. That means that you can use all the normal file-handling tools at

your disposal to debug these objects. Everything that I mentioned in the section on working with open files applies here. The only difference is that all the files you will need to look at are on a single file system.

As a result of the Linux implementation, it is possible to create and use shared memory with only standard system calls: `open`, `close`, `mmap`, `unlink`, and so on. Just keep in mind that this is all Linux specific. The POSIX standard seems to encourage this particular implementation, but it does not require it, so portable code should stick to the POSIX shared memory system calls.

Just to illustrate this point, let's walk through an example of some shell commands mixed with a little pseudocode. I'll create a shared memory segment from the shell that a POSIX program can map:

```
$ dd if=/dev/zero of=/dev/shm/foo.shm count=100       Create /foo.shm
100+0 records in
100+0 records out
$ ls -lh /dev/shm/foo.shm
-rw-rw-r--  1 john john 50K Apr  9 21:01 /dev/shm/foo.shm
```

Now a POSIX shared memory program can attach to this shared memory, using the name `/foo.shm`:[2]

```
int fd = shm_open("/foo.shm",O_RDWR,0);
```

Creating a shared memory segment this way is not portable but can be very useful for unit testing and debugging. One idea for a unit test environment is to create a wrapper script that creates required shared memory segments to simulate other running processes while running the process under test.

8.5.2 POSIX Message Queues

Linux shows POSIX message queues via the `mqueue` pseudo file system. Unfortunately, there is no standard mount point for this file system. If you need to debug POSIX message queues from the shell, you will have to mount the file system manually. To mount this on a directory named `/mnt/mqs`, for example, you can use the following command:

```
$ mkdir /mnt/mqs
$ mount -t mqueue none /mnt/mqs                    Must be the root user to use mount
```

2. The leading slash is not strictly required, but it is recommended.

When the file system is mounted, you can see an entry for each POSIX message queue in the system. These are not regular files, however. If you `cat` the file, you will see not messages, but a summary of the queue properties. For example:

```
$ ls -l /mnt/mqs
total 0
-rw-------  1 john john 80 Apr  9 00:20 myq

$ cat /mnt/mqs/myq
QSIZE:6          NOTIFY:0     SIGNO:0     NOTIFY_PID:0
```

The `QSIZE` field tells you how many bytes are in the queue. A nonzero value here may be indication of a deadlock or some other problem. The fields `NOTIFY`, `SIGNO`, and `NOTIFY_PID` are used with the `mq_notify` function, which I do not cover in this book.

To remove a POSIX message queue from the system using the shell, simply use the `rm` command from the shell and remove it from the `mqueue` file system by name.

8.5.3 POSIX Semaphores

Named POSIX semaphores in Linux are implemented as files in `tmpfs`, just like shared memory. Unlike in the System V API, there is no system call in Linux to create a POSIX semaphore. Semaphores are implemented mostly in user space, using existing system calls. That means that the implementation is determined largely by the GNU real-time library (`librt`) that comes with the `glibc` package.

Fortunately, the real-time library makes some fairly predictable choices that are easy to follow. In `glibc` 2.3.5, named semaphores are created as files in `/dev/shm`. A semaphore named `mysem` shows up as `/dev/shm/sem.mysem`. Because the POSIX API uses file descriptors, you can see semaphores in use as open files in `procfs`; therefore, tools such as `lsof` and `fuser` can see them as well.

You can't see the count of a POSIX semaphore directly. The `sem_t` type that GNU exposes to the application contains no useful data elements—just an array of `int`s. It's reasonable to assume, however, that the semaphore count is embedded in this data. Using the `posix_sem.c` program from Listing 7-14 in Chapter 7, for example:

```
$ ./posix_sem 1                                  Create and increment the semaphore.
created new semaphore
incrementing semaphore
semaphore value is 1
```

```
$ ./posix_sem 1                                    Increment the semaphore again.
semaphore exists
incrementing semaphore
semaphore value is 2
$ od -tx4 /dev/shm/sem.mysem                       Dump the file to dump the count.
0000000 00000002 ...
```

Although you can use tools like `lsof` to find processes using a semaphore, remember that just because a process is *using* a semaphore doesn't mean that it's blocking on it. One way to determine whether a process is blocking on a particular semaphore is to use `ltrace`. For example:

```
$ lsof /dev/shm/sem.mysem                          Identify the process using a named semaphore...
COMMAND PID USER   FD   TYPE DEVICE SIZE    NODE NAME
pdecr   661 john mem    REG   0,16   16 1138124 /dev/shm/sem.mysem

$ ltrace -p 661                                    Find out what it is doing...
__errno_location()                                     = 0xb7f95b60
sem_wait(0xb7fa1000, 0x503268, 0xbffa3968, 0x804852f, 0x613ff4 <unfinished ...>
```

Process is blocking in a `sem_wait` call on a semaphore located at `0xb7a1000`...

```
$ pmap -d 661 | grep mysem
b7fa1000       4 rw-s- 0000000000000000 000:00010 sem.mysem
```

This address is mapped to a file named `sem.mysem`! This process is blocking on our semaphore.

This is a bit of work, but you get your answer. Note that for this to work, your program must handle interrupted system calls. I did not do that in the examples, but the pattern looks like this:

```
do {
        r = sem_wait(mysem);                       Returns -1 with errno == EINTR if interrupted
} while ( r == -1 && errno == EINTR );
```

This is required because tools like `ltrace` and `strace` stop your process with `SIGSTOP`. This results in a semaphore function returning with `-1` and `errno` set to `EINTR`.

8.6 Tools for Working with Signals

One useful command for debugging signals from the shell is the `ps` command, which allows you to examine a process's signal mask as well as any pending (unhandled) signals. You can also see which signals have user-defined handlers and which don't.

By now, you may have guessed that the `-o` option can be used to view the signal masks as follows:

```
$ ps -o pending,blocked,ignored,caught
         PENDING           BLOCKED           IGNORED           CAUGHT
0000000000000000 0000000000010000 0000000000384004 000000004b813efb
0000000000000000 0000000000000000 0000000000000000 0000000073d3fef9
```

A more concise equivalent uses the BSD syntax, which is a little unconventional because it does not use a dash to denote arguments. Nevertheless, it's easy to use and provides more output for you:

```
$ ps s          Notice there's no dash before the s.
 UID   PID   PENDING   BLOCKED   IGNORED    CAUGHT ...

 500  6487  00000000  00000000  00384004  4b813efb ...
 500  6549  00000000  00000000  00384004  4b813efb ...
 500 12121  00000000  00010000  00384004  4b813efb ...
 500 17027  00000000  00000000  00000000  08080002 ...
 500 17814  00000000  00010000  00384004  4b813efb ...
 500 17851  00000000  00000000  00000000  73d3fef9 ...
```

The four values shown for each process are referred to as *masks,* although the kernel stores only one mask, which is listed here under the `BLOCKED` signals. The other masks are, in fact, derived from other data in the system. Each mask contains `1` or `0` for each signal `N` in bit position `N-1`, as follows:

- Caught—Signals that have a nondefault handler
- Ignored—Signals that are explicitly ignored via `signal(N,SIG_IGN)`
- Blocked—Signals that are explicitly blocked via `sigprocmask`
- Pending—Signals that were sent to the process but have not yet been handled

Let's spawn a shell that ignores `SIGILL (4)` and look at the results:

```
$ bash -c 'trap "" SIGILL; read '&
[1] 4697
$ jobs -x ps s %1
  UID   PID   PENDING   BLOCKED   IGNORED    CAUGHT STAT ...
  500  4692  00000000  00000000  0000000c  00010000 T    ...
```

You ignore `SIGILL` by using the built-in `trap` command in Bash. The value for `SIGILL` is 4, so you expect to see bit 3 set under the `IGNORED` heading. There,

indeed, you see a value of `0xc`—bits 2 and 3. Now this job is stopped, and you know that if you send a `SIGINT` to a stopped job, it won't wake up, so see what happens:

```
$ kill -INT %1
[1]+  Stopped                 bash -c 'trap "" SIGILL; read '
$ jobs -x ps s %1
  UID   PID   PENDING   BLOCKED   IGNORED    CAUGHT STAT ...
  500  5084  00000002  00000000  0000000c  00010000 T    ...
```

Now you can see a value of `2` (bit 1) under the `PENDING` heading. This is the `SIGINT (2)` you just sent. The handler will not be called until the process is restarted.

Another useful tool for working with signals is the `strace` command. `strace` shows transitions from user mode to kernel mode in a running process while listing the system call or signal that caused the transition. `strace` is a very flexible tool, but it is a bit limited in what it can tell you about signals.

For one thing, `strace` can only inform you when the user/kernel transition takes place. Therefore, it can only tell you when a signal is delivered, not when it was sent. Also, queued signals look exactly like regular signals; none of the sender's information is available from `strace`. To get a taste of what `strace` is capable of, look at the `rt-sig` program from Listing 76 in Chapter 7 when you run it with `strace`.

```
$ strace -f -e trace=signal ./rt-sig > /dev/null
rt_sigaction(SIGRT_2, {0x8048628, [RT_2], SA_RESTART}, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_BLOCK, ~[RTMIN RT_1], [], 8) = 0
Process 18460 attached
[pid 18459] rt_sigprocmask(SIG_BLOCK, [CHLD], ~[KILL STOP RTMIN RT_1], 8) = 0
[pid 18460] kill(18459, SIGRT_2)        = 0
[pid 18460] kill(18459, SIGRT_2)        = 0
[pid 18460] kill(18459, SIGRT_2)        = 0
Process 18460 detached
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
--- SIGCHLD (Child exited) @ 0 (0) ---
--- SIGRT_2 (Real-time signal 0) @ 0 (0) ---
sigreturn()                             = ? (mask now [])
--- SIGRT_2 (Real-time signal 0) @ 0 (0) ---
sigreturn()                             = ? (mask now [])
--- SIGRT_2 (Real-time signal 0) @ 0 (0) ---
sigreturn()                             = ? (mask now [])
```

I cheated a little here. Because `rt-sig` forks, I can trace both processes with the `-f` option, which follows forks. This allows me to see the sender and receiver in one trace.

`strace` normally produces a great deal of output that has little to do with what you are interested in. It is common to use a filter, specified with the `-e` option, to limit the output to what you are interested in. In this case, you would use the `trace=signal` filter to limit the output to the results of signals and signal-related system calls.

8.7 Tools for Working with Pipes and Sockets

The preferred user-space tool for debugging sockets is `netstat`, which relies heavily on the information in the `/proc/net` directory. Pipes and FIFOs are trickier, because there is no single location you can look at to track down their existence. The only indication of a pipe's or FIFO's existence is given by the `/proc/pid/fd` directory of the process using the pipe or FIFO.

8.7.1 Pipes and FIFOs

The `/proc/pid/fd` directory lists pipes and FIFOs by inode number. Here is a running program that has called `pipe` to create a pair of file descriptors (one write-only and one read-only):

```
$ ls -l !$
ls -l /proc/19991/fd
total 5
lrwx------  1 john john 64 Apr 12 23:33 0 -> /dev/pts/4
lrwx------  1 john john 64 Apr 12 23:33 1 -> /dev/pts/4
lrwx------  1 john john 64 Apr 12 23:33 2 -> /dev/pts/4
lr-x------  1 john john 64 Apr 12 23:33 3 -> pipe:[318960]
l-wx------  1 john john 64 Apr 12 23:33 4 -> pipe:[318960]
```

The name of the "file" in this case is `pipe:[318960]`, where `318960` is the inode number of the pipe. Notice that although two file descriptors are returned by the `pipe` function, there is only one inode number, which identifies the pipe. I discuss inodes in more detail later in this chapter.

The `lsof` function can be helpful for tracking down processes with pipes. In this case, if you want to know what other process has this pipe open, you can search for the inode number:

```
$ lsof | head -1 && lsof | grep 318960
COMMAND  PID    USER   FD      TYPE     DEVICE     SIZE       NODE NAME
ppipe  19991    john    3r      FIFO        0,5              318960 pipe
ppipe  19991    john    4w      FIFO        0,5              318960 pipe
ppipe  19992    john    3r      FIFO        0,5              318960 pipe
ppipe  19992    john    4w      FIFO        0,5              318960 pipe
```

As of `lsof` version 4.76, there is no command-line option to search for pipes and FIFOs, so you resort to `grep`. Notice that in the `TYPE` column, `lsof` does not distinguish between pipes and FIFOs; both are listed as `FIFO`. Likewise, in the `NAME` column, both are listed as `pipe`.

8.7.2 Sockets

Two of the most useful user tools for debugging sockets are `netstat` and `lsof`. `netstat` is most useful for the big-picture view of the system use of sockets. To get a view of all TCP connections in the system, for example:

```
$ netstat --tcp -n
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0     48 ::ffff:192.168.163.128:22   ::ffff:192.168.163.1:1344   ESTABLISHED
```

Following is the same command using `lsof`:

```
$ lsof -n -i tcp
COMMAND     PID    USER   FD   TYPE DEVICE SIZE NODE NAME
portmap    1853     rpc    4u  IPv4   4847       TCP *:sunrpc (LISTEN)
rpc.statd  1871 rpcuser    6u  IPv4   4881       TCP *:32769 (LISTEN)
smbd       2120    root   20u  IPv4   5410       TCP *:microsoft-ds (LISTEN)
smbd       2120    root   21u  IPv4   5411       TCP *:netbios-ssn (LISTEN)
X          2371    root    1u  IPv6   6310       TCP *:x11 (LISTEN)
X          2371    root    3u  IPv4   6311       TCP *:x11 (LISTEN)
xinetd    20338    root    5u  IPv4 341172       TCP *:telnet (LISTEN)
sshd      23444    root    3u  IPv6 487790       TCP *:ssh (LISTEN)
sshd      23555    root    3u  IPv6 502673       ...
       TCP 192.168.163.128:ssh->192.168.163.1:1344 (ESTABLISHED)
sshd      23557    john    3u  IPv6 502673       ...
       TCP 192.168.163.128:ssh->192.168.163.1:1344 (ESTABLISHED)
```

The `lsof` output contains PIDs for each socket listed. It shows the same socket twice, because two `sshd` processes are sharing a file descriptor. Notice that the default output of `lsof` includes listening sockets, whereas by default, `netstat` does not.

`lsof` does not show sockets that don't belong to any process. These are TCP sockets that are in one of the so-called *wait* states that occur when sockets are closed. When a process dies, for example, its connections may enter the `TIME_WAIT` state. In this case, `lsof` will not show this socket because it no longer belongs to a process. `netstat` on the other hand, will show it. To see all TCP sockets, use the `--tcp` option to `netstat` as follows:

```
$ netstat -n --tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address     Foreign Address       State
tcp        0      0 127.0.0.1:60526   127.0.0.1:5000        TIME_WAIT
```

When using these tools to look at sockets, note that every socket has an inode number, just like a file. This is true for both network sockets and local sockets, but it is more important for local sockets, because the inode often is the only unique identifier for the socket. Consider this output from `netstat` for local sockets:

```
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node Path
unix  2      [ ]         DGRAM                    3478   @udevd
unix  2      [ ]         DGRAM                    5448   @/var/run/...
unix  8      [ ]         DGRAM                    4819   /dev/log
unix  3      [ ]         STREAM     CONNECTED     642738
unix  3      [ ]         STREAM     CONNECTED     642737
unix  2      [ ]         DGRAM                    487450
unix  2      [ ]         DGRAM                    341168
unix  3      [ ]         STREAM     CONNECTED     7633
unix  3      [ ]         STREAM     CONNECTED     7632
```

This is just a small piece of the output. I'll zoom in on something specific that I can talk about in more detail. The GNOME session manager, for example, creates a listen socket in the `/tmp/.ICE-unix` directory. The name of the socket is the process ID of the `gnome-session` process. A look at this file with `lsof` shows that this file is open by several processes:

```
lsof /tmp/.ICE-unix/*
COMMAND    PID USER   FD   TYPE     DEVICE SIZE NODE NAME
gnome-ses 2408 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
gnome-ses 2408 john   19u  unix 0xc2709cc0      7036 /tmp/.ICE-unix/2408
gnome-ses 2408 john   20u  unix 0xc27094c0      7054 /tmp/.ICE-unix/2408
gnome-ses 2408 john   22u  unix 0xc2193100      7072 /tmp/.ICE-unix/2408
gnome-ses 2408 john   23u  unix 0xc1d3ddc0      7103 /tmp/.ICE-unix/2408
gnome-ses 2408 john   24u  unix 0xc1831840      7138 /tmp/.ICE-unix/2408
gnome-ses 2408 john   25u  unix 0xc069b1c0      7437 /tmp/.ICE-unix/2408
gnome-ses 2408 john   26u  unix 0xc3567880      7600 /tmp/.ICE-unix/2408
bonobo-ac 2471 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
gnome-set 2473 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
wnck-appl 2528 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
gnome-vfs 2531 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
notificat 2537 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
clock-app 2541 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
mixer_app 2543 john   15u  unix 0xc3562540      6830 /tmp/.ICE-unix/2408
```

The first thing to notice is that most of these have unique inodes, although they all point to the same file on disk. Each time the server accepts a connection, a new file descriptor is allocated. This file descriptor continues to point to the same file (the listen socket), although it has a unique inode number.

A little intuition and some corroborating evidence tell you that the server is the `gnome-session` process—`PID 2408`. In this case, the filename of the socket is a dead giveaway as well. The server is listening on file descriptor `15` (inode number `6830`). Several other processes are using file descriptor `15` and inode number `6830`. Based on what you know about `fork`, these processes appear to be children or grandchildren of `gnome-session`. Most likely, they inherited the file descriptor and neglected to close it.

To locate the server using `netstat`, try using `-l` to restrict the output to listen sockets and `-p` to print the process identification, as follows:

```
$ netstat --unix -lp | grep /tmp/.ICE-unix/
unix 2 [ACC] STREAM LISTENING 7600 2408/gnome-session /tmp/.ICE-unix/2408
```

Notice that the duplicate file descriptors are omitted, and only one server is shown. To see the accepted connections, omit the `-l` option (by default, `netstat` omits listen sockets):

```
netstat -n --unix -p | grep /tmp/.ICE-unix/2408
Proto RefCnt/Flags/Type/State/I-Node/PID/Program name Path
unix  3 [ ] STREAM CONNECTED 7600 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7437 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7138 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7103 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7072 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7054 2408/gnome-session  /tmp/.ICEunix/2408
unix  3 [ ] STREAM CONNECTED 7036 2408/gnome-session  /tmp/.ICEunix/2408
```

Unlike `lsof`, the `netstat` command does not show the inherited file descriptors that are unused.

8.8 Using Inodes to Identify Files and IPC Objects

Linux provides a virtual file system (`vfs`) that is common to all file systems. It enables file systems that are not associated with a physical device (such as `tmpfs` and `procfs`) and at the same time provides an API for physical disks. As a result, virtual files are indistinguishable from files that reside on a disk.

The term *inode* comes from UNIX file-system terminology. It refers to the structure saved on disk that contains a file's accounting data—the file-size permissions and so on. Each object in a file system has a unique inode, which you see in user space as a unique integer. In general, you can assume that anything in Linux that has a file descriptor has an inode.

Inode numbers can be useful for objects that don't have filenames, including network sockets and pipes. Inode numbers are unique within a file system but are not guaranteed to be unique across different file systems. Although network sockets can be identified uniquely by their port numbers and IP addresses, pipes cannot. To identify two processes that are using the same pipe, you need to match the inode number.

`lsof` prints the inode number for all the file descriptors it reports. For most files and other objects, this is reported in the `NODE` column. `netstat` also prints inode numbers for UNIX domain sockets only. This is natural, because UNIX-domain listen sockets are represented by files on disk.

Network sockets are treated differently, however. In Linux, network sockets have inodes, although `lsof` and `netstat` (which run under operating systems in addition to Linux) pretend that they don't. Although `netstat` will not show you an inode number for a network socket, `lsof` does show the `inode` number in the `DEVICE` column. Look at the TCP sockets open by the `xinetd` daemon (you must be the `root` user to do this):

```
$ lsof -i tcp -a -p $(pgrep xinetd)
COMMAND  PID USER   FD   TYPE DEVICE SIZE NODE NAME
xinetd  2838 root    5u  IPv4  28178       TCP *:telnet (LISTEN)
```

Here, you can see that `xinetd` is listening on the telnet socket (port 23). Although the `NODE` column contains only the word `TCP`, the `DEVICE` column contains the inode number. You also can find the inode for network sockets listed in various places in `procfs`. For example:

```
$ ls -l /proc/$(pgrep xinetd)/fd
total 7
lr-x------  1 root root 64 Oct 22 22:24 0 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 1 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 2 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 3 -> pipe:[28172]
l-wx------  1 root root 64 Oct 22 22:24 4 -> pipe:[28172]
lrwx------  1 root root 64 Oct 22 22:24 5 -> socket:[28178]
lrwx------  1 root root 64 Oct 22 22:24 7 -> socket:[28175]
```

Now `procfs` uses the same number for file descriptor `5` as `lsof`, although it appears inside the filename between brackets. It's still not obvious that this is the inode, however, because both `lsof` and `procfs` are pretty cryptic about reporting it. To prove that this is really the inode, use the `stat` command, which is a wrapper for the `stat` system call:

```
$ stat -L /proc/$(pgrep xinetd)/fd/5
  File: `/proc/2838/fd/5'
  Size: 0               Blocks: 0          IO Block: 1024   socket
Device: 4h/4d   Inode: 28178       Links: 1
Access: (0777/srwxrwxrwx)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 1969-12-31 18:00:00.000000000 -0600
Modify: 1969-12-31 18:00:00.000000000 -0600
Change: 1969-12-31 18:00:00.000000000 -0600
```

Finally, the inode is unambiguously indicated in the output.[3] Notice that I used the `-L` option to the `stat` command, because the file-descriptor files in `procfs` are symbolic links. This tells `stat` to use the `lstat` system call instead of `stat`.

8.9 Summary

This chapter introduced several tools and techniques for debugging various IPC mechanisms, including plain files. Although System V IPC requires special tools, POSIX IPC lends itself to debugging with the same tools used for plain files.

8.9.1 Tools Used in This Chapter

- `ipcs`, `ipcrm`—command-line utilities for System V IPC
- `lsof`, `fuser`—tools for looking for open files and file descriptor usage
- `ltrace`—traces a process's calls to functions in shared objects
- `pmap`—user-friendly look at a process's memory map
- `strace`—traces the system call usage of a process

---

3. Another place to look is `/proc/net/tcp`. The inode is unambiguous, but the rest of the output is not very user friendly.

8.9.2 Online Resources

- http://procps.sourceforge.net—the `procps` home page, source of the `pmap` command
- http://sourceforge.net/projects/strace—the `strace` home page