A veteran Unix and Linux enthusiast, Chris Brown has written and delivered open source training from New Delhi to San Francisco, though not on the same day.

CORE TECHNOLOGY

Dive under the skin of your Linux system to find out what really makes it tick.

Filesystem: what's going on?

Take a programmer's-eye view of the Linux filesystem.

Over the last three months our look at core Linux technology has focussed mostly on inter-process communications – pipes and sockets. This month we're going to turn our attention to the filesystem. My interest here is not about how to access and manage files from the command line (**ls**, **mv**, **rm**, **cp**, **chmod**... that kind of thing). I'm assuming you know all that. Rather, I want to take you behind the scenes of the filesystem and view it through the eyes of a programmer.

The lowest level at which you can read and write files is by using the four system calls **open()**, **read()**, **write()** and **close()**. Let's dive straight in with an example. This simple file copy program is written in C:

```
1. #include <fcntl.h>
2. #define BSIZE 1024
3.
4. void main()
5. {
6.   int fin, fout;
7.   char buf[BSIZE];
8.   int count;
9.
10.  fin  = open("foo", O_RDONLY);
11.  fout = open("bar", O_WRONLY | O_CREAT, 0644);
12.
13.  while ((count = read(fin, buf, BSIZE)) > 0)
14.    write(fout, buf, count);
15.
16.  close(fin);
17.  close(fout);
18. }
```

Down at the system call level, file descriptors (or file handles – call them what you will) are plain integers. We declare two of them (one for input, one for output) at line 6. We allocate a modest buffer at line 7; this will be used to store the data as it is being copied across. At lines 10 and 11 we open our input and output files. In each case we get back descriptors that refer to the open files. For simplicity we've just hard-coded the filenames here; more realistically, you'd take them from the command line. The parameters passed at line 11 say that we want to write to the file and that we want to create it if it doesn't exist. The mysterious octal value **0644** specifies the permissions that will be assigned to the file as it is created. You may recognise them more easily written as **rw-r--r--**. Notice that you don't get to specify the owner of the file – it will be owned by whoever runs the program. You don't get a choice.

Coding back to front

All the real work happens in the loop at lines 13 and 14, and there's a lot packed into these two lines of code. Line 13 needs reading 'inside-out'; it goes something like this: Read the next BSIZE bytes from the input file into the buffer. Record the number of bytes you read in the variable **count**. Test the value of **count**: if it's greater than zero, write however many bytes you got back out to the output file (line 14). To illustrate how this works, suppose the input file was 2500 bytes long. Then line 13 would execute 4 times, returning count values of 1024, 1024, 452 and 0. The zero means we've reached the end of the file. This 'perform an action, capture the result, and test it' is a common idiom in C; indeed, any C programmer worth his salt hides all the really important parts of his programs inside the test predicates for **if()** and **while()** loops in this way.

After we fall out of the loop (line 15) we are careful to close both file descriptors. This will ensure that any data buffered by the kernel is actually written to the disk. In this example the progam terminates immediately afterwards and any open descriptors will be implicitly closed. But if the program went on to process lots of other files we would eventually run out of file descriptors if we failed to close the ones we'd finished with.

Now I realise that some of you may think that this system-level code looks like awfully hard work. Well, maybe it's because I was weaned on a diet of assembly languages as a youngster, but I actually quite enjoy programming at this level. Short of micro-miniaturising yourself and crawling out over

"Short of crawling over the disk with a tiny magnet, this is as close as you can get to the metal."

A Ken Thompson quote

There was originally a system call named **creat()** that created a new file. Indeed there still is, but it's seldom used since you don't usually want to create a file unless you're about to write to it, and files can be created by the **open()** call, as our file copy example shows. But there's a nice story about **creat**. Apparently Ken Thompson was once asked what he would do differently if he were redesigning the Unix system. His reply: "I'd spell **creat** with an e". (See *The Unix programming environment* by Kernighan and Pike, p204). The implication being, of course, that he'd got everything else right.

the disk's surface with a tiny magnet, this is the closest you can get to the metal when it comes to file I/O.

Moving up a level

Let's move up a level and re-write the program using the standard I/O library instead of direct system calls:

```
#include <stdio.h>
#define BSIZE 1024

void main()
{
 FILE *fin, *fout; /* Input and output handles */
 char buf[BSIZE];
 int count;

 fin = fopen("foo", "r");
 fout = fopen("bar", "w");

 while ((count = fread(buf, 1, BSIZE, fin)) > 0)
   fwrite(buf, 1, count, fout);

 fclose(fin);
 fclose(fout);
}
```

It doesn't look too much different, does it? File descriptors are now of type **FILE *** instead of just integers, and the calls are renamed – **open()** becomes **fopen()** and so on. But there's an important distinction. The first program used Unix-specific calls; the second uses routines from the standard I/O library, so it should run anywhere that C is supported.

The I/O calls we've just seen – **read()**, **write()**, **fread()** and **fwrite()** – just do binary I/O. There's no sort of format conversion; they just shovel bytes between a file and an in-memory buffer. In contrast, **fprintf()** does formatted output of strings and numeric data, something like this:

```
fprintf(fout, "Answer is %f\n", 22.0/7.0);
```

Random access

By default, the contents of a file are read sequentially. There's a "file position pointer" maintained for each open file, which points to a specific byte offset within the file and determines where the next read or write will start. If I read 1024 bytes, the pointer advances by that much so that the next read continues where the last left off. Our file copy program relies on this behaviour for both the input and output files.

However, it's possible to explicitly manage this file position pointer, moving it to any desired position within the file. This gives us 'random access', as opposed to 'sequential access', into the data. (The use of the word 'random' here has always struck me as rather odd. It shows up again in the common abbreviation RAM – Random Access Memory – and seems to suggest that we have no control over which piece of the data we actually get. But I digress.)Here's an example that swaps the first and last lines in a text file. I confess it's slightly contrived; in particular it assumes that the first and last lines are the same length. But it illustrates random access quite well. This example is in PHP, though since PHP is just providing its own language binding to the same standard I/O library, the code would not look that much different in C:

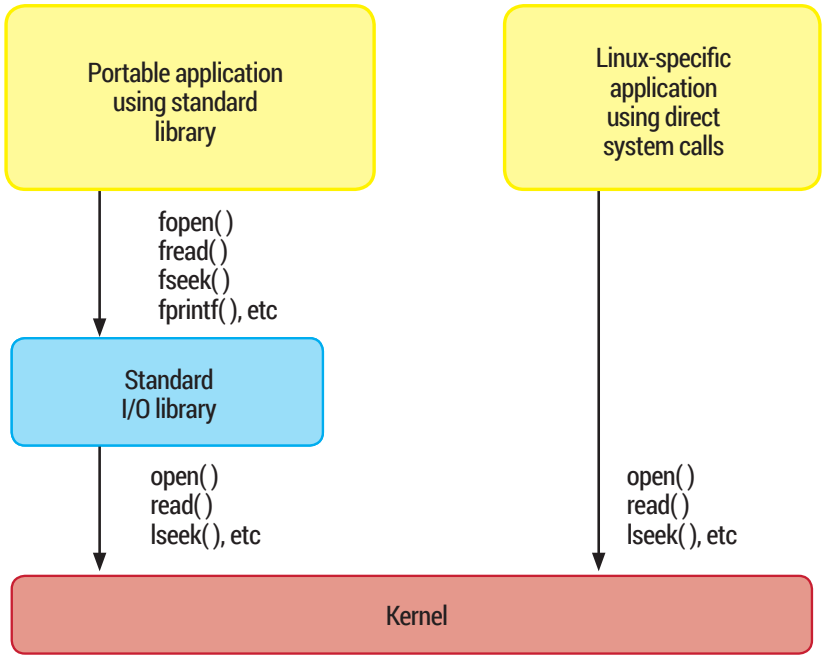
```
1. #!/usr/bin/php
2. <?php
3.  $f = fopen("foo", "r+");
4.  /* walk to the first newline */
5.  while (fread($f, 1) != "\n") ;
6.
7.  /* get current file position */
8.  $n = ftell($f);
9.
10.  /* Read and save the first line */
11.  rewind($f);
12.  $alpha = fread($f, $n);
13.
14.  /* Read and save the last line */
15.  fseek($f, -$n, SEEK_END);
16.  $omega = fread($f, $n);
17.
18.  /* Replace the first line */
19.  fseek($f, 0, SEEK_SET);
20.  fwrite($f, $omega, $n);
21.
22.  /* Replace the last line */
23.  fseek($f, -$n, SEEK_END);
24.  fwrite($f, $alpha, $n);
25.  fclose($f);
26. ?>
```

Here's the scoop. We open the file at line 3. The parameter **r+** is important – it says that we want to both read and write the file. The loop at line 5 (with an empty body) just walks along the file a byte at a time until we reach the first newline character. We are trying to figure out how long the line is. The **ftell()** call at line 8 gets the current file pointer position; this gives us the line length. Line 11 resets the file position pointer to the beginning. The call

```
fseek($f, 0, SEEK_SET)
```

would do the same. Then at line 12 we re-read that first line all in one go, saving it for later. Line 15 is interesting. It positions the file pointer one line before the end of the file. (This is where our assumption that the first and last lines are the same length kicks in.) At line 16 we read in that last line. At line 19 we rewind to the beginning of the file again then overwrite the first line of text.

Portability and the standard I/O library



Applications can choose to access files through the Standard I/O library, or use direct system calls .

mmap

The **mmap()** system call provides a very different approach to random access into a file's data. It allows a file's contents to be mapped into the address space of a process and accessed like an array. Random access is achieved simply by indexing into the array. The **mmap** call itself is a little complicated, but if you're looking for an efficient way to dive into a file, **mmap** may be worth a look.

Finally, at lines 23 and 24 we scoot along to the start of the last line of the file and overwrite that, too.

Well, that's a little tricky to follow, so I've drawn a diagram that might help (see below). And if you want to explore this in more detail, the man page for **fseek** will show you the C language bindings for these functions, or browse to **http://php.net/manual/en/function.fseek.php to see the PHP bindings**.

Listing directories, deleting files

So far we've concentrated on accessing the data within a file, with code that does things broadly equivalent to commands like **cat** and **cp**. Let's shift focus a little and look at the management of the filesystem itself; something more analogous to commands like **cd**, **ls**, and **rm**. Here's a program that will delete all the files in a directory (passed as a command line argument). To add variety, this one's in Perl; it even has some error checking built in!

```
1. #!/usr/bin/perl
2.
3. if (@ARGV != 1) {
4.   warn "usage: empty dirname\n";
5.   exit(1);
6. }
7.
8. if (!chdir($ARGV[0])) {
9.   warn "$ARGV[0]: $!\n";
10.  exit(1);
11. }
12.
13. opendir($d, ".");
14.
15. foreach $info (readdir($d)) {
16.   if ($info ne "." && $info ne "..") {
17.     print "removing $info\n";
18.     if (unlink($info) != 1) {
19.       warn "$info: $!\n";
20.       exit(2);
21.     }
22.   }
23. }
```

Let's talk through this. Lines 3–6 verify that the user provided a command-line argument, printing an error message and bailing out if not. Lines 8–11 change into the directory specified on the command line (equivalent to **cd** in a shell script), printing an error if this fails. Line 13 opens the directory; the handle is returned in **$d**. Line 15 is the start of a loop, calling **readdir()** repeatedly to enumerate the files in the directory. There is an explicit check at line 16 to ignore the entries **.** and **..**; otherwise the file is deleted (unlinked) at line 18. Notice that the program will fail ungracefully if there's a subdirectory in the directory you're emptying. Do be careful if you run this example – it really will remove all the files in the directory you specify, so beware!

My reason for providing examples in different languages is not just to add variety, but to make the point that although different languages have different syntax, they are all providing language bindings to the same library routines – in this case **chdir()**, **opendir()**, **readdir()** and **unlink()**.

Everything looks like a file

As we reach the end of this discussion we're in a good position to answer the question "what is a file?" Well, the traditional answer is that it's information stored on a disk, referenced by a name. But there's a broader view... anything that responds to the classic system calls such as **open()**, **read()** and **write()** in the appropriate way is going to look like a file, and can be accessed by the usual command line tools like **cat** or **cp**. This perhaps makes a little more sense of the 'files' in the procfs and sysfs virtual filesystems, usually mounted onto **/proc** and **/sys**. These files are purely a figment of the kernel's imagination, providing a view from userspace into internal kernel data. For example, the following command:

```
$ cat /proc/cpuinfo
```

will provide details of the kernel's view of the processor on which it's running. Most parts of these filesystems are read-only – you can't upgrade your processor by writing to **/proc/cpuinfo** or get more memory by writing to **/proc/meminfo**. But some parameters can be tweaked by writing to the appropriate 'file'. A classic example is **/proc/sys/net/ipv4/ip_forward**, which determines whether the Linux kernel will forward (route) IP traffic. By default this is disabled, (zero) as you'll see if you examine the file:

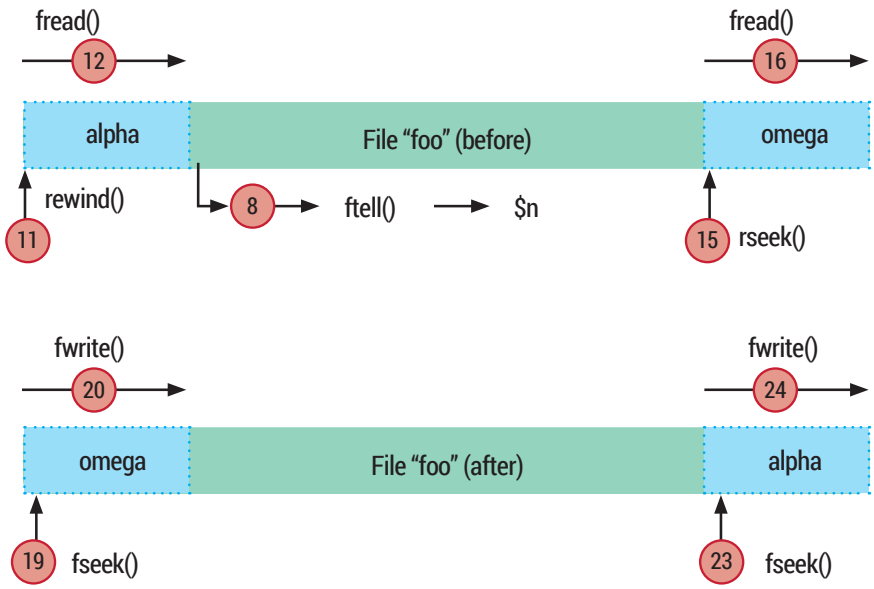
```
$ cat /proc/sys/net/ipv4/ip_forward
0
```

but you can enable it by writing to the 'file' (you'll need to do this as root):

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

There are lots more parameters you can interrogate and adjust in this way; my purpose here is not to survey them all but simply to point out that we are able to treat these things like files because they respond to the file I/O system calls in the the usual way.

Swapping records in a file



Linux supports random access into a file's data. Numbered circles correspond to numbered code lines within the main text.

How to become invisible

Would you like to learn how to write to a file that has no name from a program that doesn't exist? Here's how! There's a well-known (but slightly weird) feature of Linux that if a program opens a file then deletes it (keeping it open) the file will continue to exist. It will have a valid inode but no entry in the filesystem. Here's a program that does exactly that (this one's in C again):

```
1. #include <fcntl.h>
2.
3. main()
4. {
5.   int fout;
6.   char buf[10];
7.   fout = open("/tmp/topsecret", O_WRONLY | O_
CREAT, 0600);
8.   unlink("/tmp/topsecret");
9.   write(fout, "attack at dawn\n", 16);
10.  pause();
11. }
```

The **pause()** at line 10 is there simply to keep the process alive.

To compile this program, place the code into a file called **secret.c** and compile it with:

```
$ gcc -o secret secret.c
```

If we run this program with the **unlink()** call at line 8 commented out, we can of course list and examine the output file in the usual way:

```
$ ./secret &
$ ls -l /tmp/topsecret
-rw------- 1 chris chris 16 Aug  6 15:06 /tmp/topsecret
$ cat /tmp/topsecret
attack at dawn
```

But if we re-run it with line 8 in place, things get more interesting. There will be no entry in the filesystem for **/tmp/topsecret**. It won't show up on the output of **ls** and you certainly can't examine it with **cat**.

```
$ ls -l /tmp/topsecret
ls: cannot access /tmp/topsecret: No such file or
directory
```

We can even delete the executable:

```
$ rm secret
```

Now, neither the file we're writing nor the program that's writing it has an entry in the filesystem. Is this weird or what? And why do we care? Well, let's pin on our "Paranoid About Security" badges and imagine that a hacker of evil intent has managed to plant a program on our machine that is collecting important information in a file that it later intends to transmit back to the bad guy. Using this trick, our villain remains pretty well hidden. But not entirely. We can ask **lsof** (my command of the month in LV005) to show unlinked files like this:

```
$ sudo lsof +L1
secret   8632 chris   3w  REG   8,1     16    0
1573121 /tmp/topsecret (deleted)
```

The option **+L1** tells **lsof** to only show files that have a link count less than 1. If you run this command you will almost certainly see lines of output in addition to the one shown here from programs like **init** (among others).

OK, so we have some evidence that the file still exists. From this output we know its size (16 bytes) and we know the PID of the process that has it open (8632). But given that it has no name, can we see its contents? It turns out we can! You may be aware that **/proc** contains directories named after each process ID, and within each of these is a subdirectory called **fd**. Here you'll find symbolic links (named after the file descriptor) to each file that the process has open. In this case, file descriptor 3 is the one we're interested in:

```
$ cd /proc/8632/fd
$ ls
0  1  2  3
$ cat 3
attack at dawn
```

and – hey presto! – we see the contents of our invisible file.

Similarly, most of the things in **/dev** present a file-like view to userspace. Pseudo-devices like **/dev/null**, **/dev/random**, and **/dev/zero** deliver data streams (or not, in the case of **/dev/null**). Disk partitions have names like **/dev/sda3** (these are linked to more complex names in modern linux kernels) and can be written to like a file, so that a command like:

```
$ echo "Kilroy was here" > /dev/sda3
```

is perfectly legal, though probably not at all a good idea if there is a filesystem on **sda3**.

This "everything looks like a file" view of things, which is such a fundamental part of Linux, provides a very consistent picture of the world, with disk partitions having owners, timestamps and access permissions just like regular files. The only things that aren't part of this world (for reasons I have never really understood) are the network interfaces. There's no **/dev/eth0** for example.

Next month I'm planning to look at the system calls that examine and modify a file's attributes, and to examine the **inotify()** API, which lets you monitor the filesystem for changes. See you then!

Command of the month: dd

My command of the month is **dd**. It's basically a file copy program. A simple invocation is:

```
$ dd if=foo of=bar
```

which copies the file **foo** to **bar**. Of course you could do it more easily with **cp**.

But **dd** supports various conversions that will be applied to the file as it is copied. For example,

```
$ dd if=foo of=bar conv=ucase
```

will convert the file to upper case. Or:

```
$ dd if=foo of=bar conv=swab
```

will swap each pair of bytes in the file (historically useful if you were moving data between "little-endian" and "big-endian" machines).

The **dd** command also lets you control how much data is copied, and in what size chunks. For example:

```
$ dd if=/dev/zero of=zeros bs=1MB count=10
```

copies the pseudo-device **/dev/zero** (an endless source of zeros) into the file **zeros**, copying 1MB (1 million bytes) at a time, and continuing for 10 records. So we end up with a file exactly 10,000,000 bytes long.

Occasionally **dd** is used to image disk partitions. For example,

```
# dd if=/dev/sda3 of=sda3copy
```

will make a direct bit-for-bit copy of a complete disk partition into the file **sda3copy**. Or you can restore a partition by doing it the other way round:

```
# dd if=sda3copy of=/dev/sda3
```

though please don't try this at home, folks, unless you know what you are doing! Also beware that copying disk partitions in this way may not be the most efficient approach, because **dd** will blindly copy the partition byte by byte, whereas tools like *Partimage* and *Clonezilla*, which understand the filesystem structure, will only copy the blocks that are actually in use. This can result in a much smaller image if the file system isn't very full.

The name **dd**, and to some extent its command syntax (which is decidedly not Unix-like) are a reference to an old job control language used on IBM mainframes. Nowadays we take the ease and elegance of the Unix command line for granted. If you think it's arcane, please believe an old-timer: the job control language needed to persuade an IBM mainframe to to anything at all was breathtaking in its obscurity.