On the T(r)ail of Open Files

By Jerry Peek

If you're more of a Linux user than a programmer, you may not have given much thought to how the operating system handles files. As a user, you simply give a filename to a program (from the "Open" command on a menu, from the command line, and so on) and the file is (hopefully) accessed however it's supposed to be.

However, programmers who perform low-level access of files — for instance, to seek to a particular point in a file or rewinding the file to its start — have to understand more about Linux file handling. That's where we're headed this month: seeing how Linux handles files that have been opened by a program, and learning how you can take advantage of this even if you don't usually write programs that handle files.

Along the way, we'll look at `tail -f`, *MultiTail*, and `less +F`. These three programs can show you what's happening to a file as it grows (as data is added to end of the file). They're handy for viewing log files and monitoring a long-running process.

Open Files

Let's start with a quote from the Linux *man* page for `open()`, the low-level system call that programmers use to open a file:

> *The* `open()` *system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the* `fork()` *system call.) The new file descriptor is set to remain open across exec functions (see* `fcntl()`*). The file offset is set to the beginning of the file.*

As in most *man* pages, there's a lot of information packed into that paragraph!

When you give a file's pathname (like */a/b/afile*, *../afile*, or simply *afile*) to a program, the program opens the file to access its contents. A file can be opened for reading, for writing, or (in some cases) for both. When the Linux kernel opens a file, it returns a *file descriptor* — one of the numbers 3, 4, 5, and so on — which your program uses to refer to that file. The file stays open until the program closes it or until the process ends. (This rule — that a file stays open until it's closed or the process ends — can be taken to extremes. See the sidebar "Removing Open Files?")

Every process has at least three file descriptors assigned to it: *standard input* (*stdin*) is file descriptor (*fd*) 0; *standard output* (*stdout*) is *fd* 1, and *standard error* (*stderr*) is *fd* 2. So, if a program issues an `open()` call on file *foo*, and no other files were open so far, the contents of *foo* are accessible through *fd* 3.

After a file is opened, the kernel also tracks its *file offset*. This is the point in the file where the process is currently reading or writing. It's kind of like putting a bookmark in a book to hold your place. Each time you read more and then stop reading for a while, you move the bookmark along toward the end of the book. Later, when you want to read some more, the bookmark has held your place. The file offset works the same way. You can move the file offset ahead by reading or writing data, or you can move it more furtively with a system call like `seek()`.

Bourne-type shells — *bash*, for instance — let you open files and access them by their *fd*. We've seen this in the June 2004 column "Execution and Redirection," available online at http://www.linux-mag.com/2004-06/power_01.html.

File This Under Linux

In Linux, basically all input and output (I/O) is done via "files" — streams of characters, accessed through a file descriptor with a file offset pointer — although many of those "files" are actually pipes, disk drives, and other character sources or sinks.

With that in mind, let's look again at a redirected-input *while* loop from the May 2004 column, "Great Command-line Combinations," which you can read at http://www.linux-mag.com/2004-05/power_01.html:

```
find /proj -type d -print |
while read dir
```

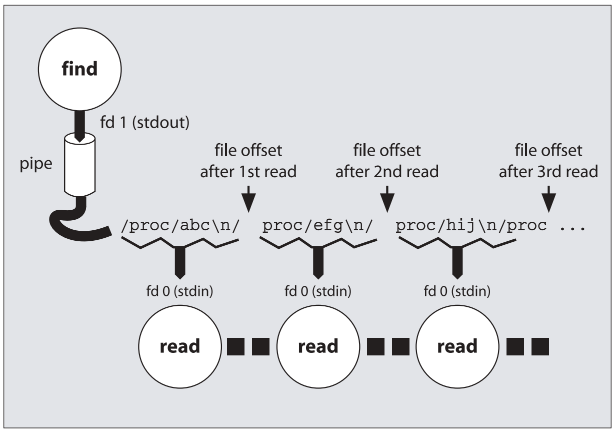

find
fd 1 (stdout)
pipe
file offset after 1st read
file offset after 2nd read
file offset after 3rd read
/proc/abc\n/ proc/efg\n/ proc/hij\n/proc ...
fd 0 (stdin)
fd 0 (stdin)
fd 0 (stdin)
read
read
read

FIGURE ONE: Reading data from an open file

```
do
  ...commands...
done ...
```

Figure One shows what's happening at the input side of the loop. The *find* process is writing a string of characters (actually a series of pathnames and newlines) onto its standard output, which feeds a pipe. On the other side of the pipe, those characters are available to read in the order they were written.

(This isn't an exact picture. The real story depends on how the shell implements the redirected-input loop. For instance, the loop is often run in a subprocess which would get its standard input from the pipe. The effect, though, is the same.) Each *read* process takes text from stdin (*fd* 0) until it reads a newline. Then it stops reading, and the file offset stays the same until the next *read* process comes along later on the next pass of the loop.

In all processes, the file offset tracks where the next read (or write) takes place. An application like *split* can read data from an open file megabyte by megabyte. A pager like *more* can read data in chunks that precisely fill your screen, waiting to read the next chunk until you press the spacebar.

When the file offset reaches the end of the file, a program can see that and quit reading. (`read()`, for instance, returns a nonzero exit status when that happens.)

Another interesting choice is to keep trying to read, waiting for the file to grow.

Watching a File Grow

Several Linux utilities don't quit when they reach the end of a file. As was mentioned before and in previous columns, those utilities are very handy when you're viewing a log file, like */var/log/messages*, that grows over time.

One of these utilities, *less*, is the "Swiss Army Knife of file-viewing programs." With the command-line option `+F` (or, if you have multiple files, use `++F` instead), *less* shows a file as it grows. Other commands are ignored until you interrupt

REMOVING OPEN FILES?

As we discussed in the September 2004 column, "Think Links" (available online at http://www.linux-mag.com/2004-09/power_01.html), a filename in a directory isn't actually the file itself; it's simply a reference (a hard link) to the inode that contains file information. When a program opens a file, the filename is used to get the inode number, then to get the file information, and (finally) to open the file.

When you remove a file (with the Linux *rm* utility, for instance) that doesn't actually remove the file. It removes a hard link to the file's inode. Once a file has no hard links (its hard link count is zero), its data is then finally freed for re-use.

But that's actually not quite true: if a file is open, its data won't be removed until it's closed — even if it has no hard links remaining!

So, for example, here's a shell script that removes itself but keeps running. If you'd like to try this, save the following in a file, add execute permission, and run it:

```
#!/bin/sh
echo "hello from $0:"
ls -l $0
echo "Now removing myself..."
rm $0
echo "My file is still open, but it's gone:"
ls -l $0
echo "Goodbye."
```

(The shell parameter `$0` expands into the name of the script file.)

This technique is more useful when a program needs a temporary file that must be removed when the process terminates. To do this, use the *exec* command (described in the June 2004 column, "Execution and Redirection", which you can read online at http://www.linux-mag.com/2004-06/power_01.html) to open the file and associate a file descriptor with it.

Let's finish with a sample script that runs a program (for demonstration, we'll use *who*, which lists all logged-in users) and saves its output to a temporary file. Then it opens the temporary file for reading on the shell's standard input (*fd* 0) and removes the file's link. After that, the *read* command reads from its standard input, line by line, parsing each line of *who* output into three shell variables. At this point, the file can't be found in */tmp*, which gives the script some extra security and helps to avoid name conflicts.

```
#!/bin/sh
temp=/tmp/DELME$$     # name of temp file
umask 077             # set -rw-------
who > $temp           # fill temp file
exec < $temp          # open it on stdin
rm $temp              # remove its link

# Read temp file content line by line:
while read user tty login
do
  echo "processing $user on $tty..."
done
```

One problem with this technique is what happens over a network filesystem like *NFS*. Because the file doesn't actually have a name anymore, NFS has to create a special temporary file and negotiate the tricky stuff needed to make the file's contents accessible over the network.

Also, if the "removed" temporary file is huge and you need to do some filesystem cleanup, you can't use *rm* to remove the file because its contents are basically "untouchable" at this point! The only way to actually remove the data blocks from the disk is to kill the shell process which has the file open. If you're using this technique for a production script or one that handles a lot of data, you might talk with your system administrator first.

less (typically with CTRL-C); then you can mark and return to places in the file, jump between multiple files, and more. To resume viewing new lines, type the command `F` while using *less*.

Another utility, MultiTail, can open multiple windows on a graphical display. This is similar to opening multiple terminal windows with a *less* process running in each, but it has several advantages. One is that text displayed can be color-coded and filtered so that only some lines are shown — eliminating "noise" lines from a log file, for instance. MultiTail can also run a program like *netstat* or *w* over and over, highlighting and filtering the program's output in the same way as for files. MultiTail can be downloaded from its home page at http://vanheusden.com/multitail/.

Let's dig into the the granddaddy of these programs, *tail*. By default, *tail* shows the last ten lines of a file and quits. Adding the `-f` option tells *tail* to keep trying once per second, outputting anything added to the end of the file.

The original *tail* (on old BSD systems, at least, where this author had the most experience) could only "watch" one file. Another problem was what happened if a file was *replaced* while *tail* was waiting for it to grow. For instance, at midnight each day, a cron job could rename */var/log/messages* to *messages.0* and create a new empty *messages* to hold the new day's log messages. If you were running `tail -f /var/log/messages`, *tail* would show no new lines after midnight.

The GNU version of *tail* solved that problem by adding two options: `--follow=name` and `--retry`. Armed with our knowledge of open files and file descriptors, let's learn how it works.

We've seen that, in general, a program opens a file by its pathname and the kernel returns a file descriptor. From then on, the program doesn't need to know or care about the file's pathname because it has direct access to the file's contents. (In fact, as explained in the sidebar "Removing Open Files?", an opened file can even be "removed" and the process will still have access to it.) So, in our example two paragraphs before, when the *cron* job executed `mv messages messages.0`, that changed the file's pathname, but the file was still open. So, *tail* continues to watch the current open file — which, because it's been renamed, no longer receives input from the system logging daemon.

The GNU *tail* options `-f`, `--follow`, and `--follow=descriptor` all act like the "classic" `tail -f`: opening a file and tracking changes through the open file descriptor. If you use `--follow=name`, though, *tail* will periodically reopen the file by name to see if it has been removed and recreated. In this case, you may also want to use `--retry`, which keeps trying to open a file even if it's inaccessible when *tail* starts or at some later time. The `-F` option is a shortcut for `--follow=name --retry`.

Another handy GNU *tail* option is `--pid=PID`, which makes *tail* exit if the process with ID number *PID* dies. For instance, if you want to watch the log file from a program that's running in the background, you could monitor it with a little Bourne-type shell script like this:

```
#!/bin/sh
someprog > someprog.log &
tail -f --pid=$! someprog.log
```

The shell operator `$!` holds the PID of the last background job.

POWER TIP: Tunnel your POP Connection

If your email setup uses a standard POP connection and the network connection isn't secure, your POP username and password — as well as all of your email — is sent "in the clear," open to snoopers. If the remote POP server also runs an SSH daemon, though, you can tunnel the POP connection via *ssh*. Here's the command to run from your local host (with your mail client):

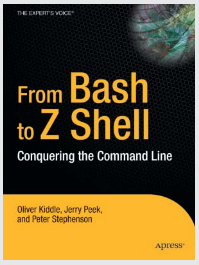
```
ssh -L 12345:127.0.0.1:110 remuser@pophost
```

Then configure your email client to fetch mail from *localhost* port 12345. When your email client connects, the connection is forwarded to `pophost`, where a connection is made to `127.0.0.1` (which is the remote machine's loopback interface) port 110 (the POP server).

You can also specify this setup in the *ssh* configuration file. (Remember, though, that only the superuser can forward privileged ports — those port numbers under 1024.)

CONQUER THE COMMAND LINE

For the past few years, this column has shown how you can combine a shell with *Linux* utilities into flexible and powerful combinations. We've focused on the utilities, though, and barely scratched the surface of shells. If you'd like to read much more about shells, a new book by Jerry Peek and two co-authors digs into the *bash* and *zsh* shells. "From Bash to Z Shell: Conquering the Command Line" (ISBN 1-59059-376-6, Apress, http://www.apress.com) is unique for two reasons: it emphasizes using shells interactively (from the command line) and has extensive and unique coverage of *zsh*. The book covers shell scripting, but its focus is on the literally hundreds of features hidden behind the shell prompt.


THE EXPERT'S VOICE
From Bash to Z Shell
Conquering the Command Line
Oliver Kiddle, Jerry Peek, and Peter Stephenson
Apress

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.