Wizard Boot Camp, Part Seven

Jerry Peek

This month, as a part of our long-running series on obscure Linux topics that wizards should know, we'll wrap up the discussion of Linux processes with a look into the twisty (virtual) corners of the */proc* pseudo-filesystem. This little-known directory is a gold mine of information about your system and its processes.

If you typically call utilities like `uptime` or `ps` to get system information from scripts, you may start using */proc* from now on: your script can read */proc* without invoking a new process, so it can be more efficient. One warning, though: */proc* isn't necessarily the same on every Linux system, and non-Linux systems may not have it at all. If you use */proc* in a script that should be portable to other systems, check the other systems — or stick to the old standby utilities like `uptime`.

Introducing /proc

If you haven't looked in */proc* before, that's a good place to start. See *Listing One*.

We won't describe every part of */proc* here; doing that would fill most of this article's three pages! (And, to save space, we've omitted a lot of the entries from *Listing One*.) You can get details from the *proc* manual page. Let's hit some highlights.

You can treat the virtual filesystem entries in */proc* as if they're on an actual hard disk: for instance, read the files with `cat` or `less`; list symbolic links (like */proc/self*) with `ls -l`; `cd` into directories or run `ls` on them. The sidebar *Reading /proc "Files" Efficiently* explains an efficient way to get the contents of */proc* "files."

Most of the names are self-explanatory. The numbered directories correspond to the processes running on your system; the number is the process PID. We'll look at those, the the special symlink named *self*, in the next section.

LISTING ONE: Top Level of /proc Directory

```
  cd /proc
$ ls -F
1/           dma          self@
10/          driver/      slabinfo
1043/        loadavg      stat
11037/       locks        swaps
11041/       meminfo      sys/
11042/       misc         sysrq-trigger
cmdline      modules      sysvipc/
cpuinfo      mounts@      uptime
crypto       mtrr         version
devices      partitions   vmstat
diskstats    scsi/        zoneinfo
$
```

READING /PROC "FILES" EFFICIENTLY

From a shell script, it's more efficient to read a file with the bash operator `$(<file)`, which opens a file directly without starting a new process. (Using a utility like cat starts a new process to run the program.) For instance, in a shell script that's monitoring the system load average, you could read */proc/loadavg* into the array named *loadavg* like this:

```
loadavg=( $(</proc/loadavg) )
```

Then `${loadavg[0]}` has the first load average (the one-minute value), and so on.

- The file *cpuinfo* gives detailed information on the machine's processor(s).
- The file *loadavg* gives the 1, 5, and 15-minute load average, the number of processes currently executing, and the last PID created.
- *partitions* lists the current disk partitions, including major and minor device numbers and the number of blocks.
- *sys* gives detailed system performance information in a series of subdirectories such as *fs* (filesystem), *kernel*, and *net*.

Per-process Directories

As we said, the numbered directories have information about each process on the system. (Or you may see only your own processes — and many other named entries may have permissions that only allow superusers to read them.) These make a nice alternative to the Byzantine options and output formats of `ps`. For instance, if you're trying to find the PPID of process 11037 (that is, the PID of the parent that started process 11037), look in */proc/11037/ppid*:

```
$ cat /proc/11037/ppid
1043
```

Soon we'll see more of what's in these directories. By the

way, one of those numeric directories in the `ls -F` output from *Listing One* is guaranteed not to exist anymore. Which one? It's the directory that was created with information about that *ls* process itself. Once the *ls* process finished listing */proc*, the *ls* process terminated, so its virtual directory in */proc* vanished.

A process that needs to get information about itself can look in the numeric directory pointed to by the symbolic link */proc/self*. This is worth a moment of thought before you use it. Consider this example:

```
$ ls /proc/self
attr       exe           oom_adj     status
auxv       fd            oom_score   task
cmdline    maps          root        wchan
cpuset     mem           smaps
cwd        mounts        stat
environ    mountstats    statm
```

(If you have *ls* aliased to run `ls -F`, you'll get a result like `/proc/self@` instead of the directory entries shown above. In that case, try `/bin/ls /proc/self`, or `\ls /proc/self`, to get a listing of the directory's contents.)

Which process is that listing for: the shell that's running `ls`, or for the `ls` process itself? Think: which process is actually reading */proc/self*? Right: the `ls` process is reading */proc*, so you'll see information about `ls` in the listing.

LISTING TWO: A shell's Own Process Information

```
$ echo /proc/$$
/proc/2588
$ ls -l /proc/$$
total 0
dr-xr-xr-x 2 0 2008-02-12 12:26 attr
-r——— 1 0 2008-02-12 12:26 auxv
-r—r—r— 1 0 2008-02-12 12:26 cmdline
-r—r—r— 1 0 2008-02-12 12:26 cpuset
lrwxrwxrwx 1 0 2008-02-12 12:26 cwd -> /home/jpeek
-r——— 1 0 2008-02-12 12:26 environ
lrwxrwxrwx 1 0 2008-02-12 12:26 exe -> /bin/bash
dr-x—— 2 0 2008-02-12 12:26 fd
-r—r—r— 1 0 2008-02-12 12:26 maps
-rw——- 1 0 2008-02-12 12:26 mem
-r—r—r— 1 0 2008-02-12 12:26 mounts
-r——— 1 0 2008-02-12 12:26 mountstats
-rw-r—r— 1 0 2008-02-12 12:26 oom_adj
-r—r—r— 1 0 2008-02-12 12:26 oom_score
lrwxrwxrwx 1 0 2008-02-12 12:26 root -> /
-r—r—r— 1 0 2008-02-12 12:26 smaps
-r—r—r— 1 0 2008-02-12 12:26 stat
-r—r—r— 1 0 2008-02-12 12:26 statm
-r—r—r— 1 0 2008-02-12 12:26 status
dr-xr-xr-x 3 0 2008-02-12 12:26 task
-r—r—r— 1 0 2008-02-12 12:26 wchan
```

To get information on your shell, use the `$$` parameter. It expands into the current shell's PID number. There's an example in *Listing Two* for the shell whose PID happens to be 2588.

A process that needs to get information about itself can look in the numeric dictionary pointed to by the symbolic link */proc/self*, which seems appropriate

Although the sizes list as 0 bytes, that's deceptive: The files output whatever the current value is at the time you read them. For instance, the *status* "file" gives the current status of the process:

```
$ cat /proc/self/status
Name:   cat
State:  R (running)
SleepAVG:        88%
Pid:    22383
PPid:   22010
Groups: 1007
VmSize:     2748 kB
...
```

The contents of *status* are a handy alternative to reading many of the other files in the directory — which give the same information in smaller chunks.

Several of the entries are symbolic links. Reading the directory with `ls -l` shows each link's target. For instance, the process' current directory, pointed to by *cwd*, is */home/jpeek*. (The shell's current directory was */home/jpeek*, which *cat* inherited when the shell started it.)

The *root* entry points to the process' root directory. That's typically /, as you see here — but it can be different for a process run with chroot (2).

The *fd* subdirectory lists open file descriptors for the process... which leads us neatly into the next section.

The /proc/#### /fd and /dev/std* Subdirectories

I've talked before in this column about open files and file descriptor numbers. Two handy virtual parts of the Linux filesystem, the */proc/nnnn/fd* and */dev/std** subdirectories, make it easy to explore these.

Let's start with some special entries in */dev*. The entries */dev/stdin*, */dev/stdout*, and */dev/stderr* point to those open standard I/O files in the current process. These entries are actually symlinks pointing into the (virtual) */dev/fd* subdirectory, as you can see by listing them:

```
$ ls -l /dev/std*
lrwxrwxrwx ... /dev/stderr -> fd/2
lrwxrwxrwx ... /dev/stdin -> fd/0
lrwxrwxrwx ... /dev/stdout -> fd/1
```

When you experiment with file descriptors, it may be best to do from a shell script, or from an interactive subshell, just in case of errors

What's in the *fd* subdirectory? It's a list of the currently-open file descriptors in the process. It's actually a symlink to the */proc/self/fd* directory, which has the real information:

```
$ ls /dev/fd
0  1  2  3
$ ls -l /dev/fd
lrwxrwxrwx ... /dev/fd -> /proc/self/fd
```

Let's look in that directory:

```
$ ls -l /proc/self/fd
total 0
lrwx—— ... 0 -> /dev/pts/5
lrwx—— ... 1 -> /dev/pts/5
lrwx—— ... 2 -> /dev/pts/5
lr-x—— ... 3 -> /proc/26055/fd
$ tty
/dev/pts/5
```

When you list that directory, you're actually seeing the open files for the `ls` process — as explained earlier in this column. But, since `ls` inherits the open files from the process that started it — in this case, the open files from the shell that ran `ls` — what you see are the shell's open files plus any other files that `ls` might have opened.

The standard input, output, and error all point to */dev/pts/5*, which is our current terminal device — as `tty` confirms. So, another way to write to the standard error of your current process — instead of using the Bourne shells' operator `1>2` — is by writing to */dev/stderr*. This is a great help to C-shell scripts, since they don't have an easy way to write arbitrary text to the standard error (which is where error messages should be written):

```
echo an error > /dev/stderr
```

File descriptor 3 is also open in this process; it points to the *fd* subdirectory of process 26055. As we said, it's for `bash` or `ls`.

This leads to a nice technique for exploring how open files are used in a shell: by listing */proc/self/fd* after you change the shell's open files.

Fiddling with File Descriptors

When you experiment with file descriptors, it may be best to do from a shell script, or from an interactive subshell. That way, if you do something you didn't mean to do (such as redirecting the standard output to a file, so you can't see the outputs of commands), it's easy to put things back to normal: simply terminate the subshell. Because changes to a child process don't affect its parent process, the parent shell retains its original standard input and output after the subshell exits.

Let's start a child *bash* shell. When we're done playing — or, if something goes wrong — we can get back to a sane state by typing CTRL-D or *exit* to terminate the child shell. We'll set the shell prompt to `sub$` as a reminder that this is a subshell.

To save typing, we'll store a temporary filename in an environment variable with the arbitrary name *T*. (Environment variables are copied to child processes.) We'll also make an alias that lists `/proc/self/fd`.

LISTING THREE: Watching Open Files in /proc/self/fd

```
$ export T=/tmp/myfile
$ bash
$ PS1='sub$ '
sub$ alias ck='ls -l /proc/self/fd'
sub$ ck
total 0
lrwxrwxrwx ... 0 -> /dev/pts/5
lrwxrwxrwx ... 1 -> /dev/pts/5
lrwxrwxrwx ... 2 -> /dev/pts/5
sub$ exec 3> $T
sub$ ck
total 0
lrwxrwxrwx ... 0 -> /dev/pts/5
lrwxrwxrwx ... 1 -> /dev/pts/5
lrwxrwxrwx ... 2 -> /dev/pts/5
lrwxrwxrwx ... 3 -> /tmp/myfile
sub$ echo a test message 1>3
sub$ cat $T
a test message
sub$ cat /proc/self/fd/3
a test message
sub$ ls garbage
ls: cannot access garbage
sub$ ls garbage 2>3
sub$ cat $T
a test message
ls: cannot access garbage
sub$ exit
$ rm $T; unset T
```

Listing Three shows some examples. (Try them yourself!) To avoid confusion here, we'll omit listings for file descriptors that `bash` and `ls` may open. Here's what we do:

- After defining the `ck` alias and running it, we can see the usual three standard I/O file descriptors.
- Running `exec 3> $T` opens the file `/tmp/myfile` for writing and associates file descriptor 3 with it.
- The shell operator `1>3` makes the standard output of `echo` (file descriptor 1) go to file descriptor 3 — which is the file in */tmp*. We write three words there.
- Reading the file with `cat $T` shows the words we wrote there.
- As an example that's somewhat opaque but also illustrative, `cat /proc/self/fd/3` does the same thing! (Although the file */tmp/myfile* is only open for writing from the shell, don't let that confuse you. */proc/self/fd/3* is just a symbolic link pointing to the file that was opened by the shell. The command `cat /proc/self/fd/3` is completely independent of the shell; `cat` is simply reading a file in the filesystem — which it finds via the symbolic link at */proc/self/fd/3*.)
- We run `ls garbage` to generate an error message on the standard output. Then we re-run the command with the operator `2>3`, which sends standard error (fd 2) to the file in */tmp* via fd 3. Running `cat` shows the two lines in */tmp/myfile*.

  This illustrates another important reason to use open files and file descriptors instead of constantly re-opening a file from a script: the file stays open, and you can keep adding text to it, until you close the open file or end the shell process that's holding the file open.
- We end the shell subprocess with *exit*. That automatically closes the open file */tmp/myfile*. Then we remove the file and the environment variable that held its name.

Experiment!

There's a lot more to see in */proc*, and a lot to learn from experimenting with */proc/self*. Until next month, try exploring and see what you find.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for more than 25 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.