What's in a Pathname?

By Jerry Peek

How can you locate something in the filesystem with the least amount of work? You might be tempted to use a graphical (GUI) file manager, but in many cases the command line is faster. If you do a lot of work with your system, learning some pathname power tools can save you a lot of time.

This is the second in a pair of columns about locating things in the filesystem. Last month's column, "A Very Valuable Find," (available online at http://www.linux-mag.com/2001-09/power_01.html) explained how to use the *find* utility to perform sophisticated searches and operations on a collection of files. This month we'll see what's really behind pathnames and the current directory, and see tips for working smarter with the Linux filesystem.

Two Paths to the Very Same Place

Let's start with a few basic definitions.

- The directory that holds all other directories (as its subdirectories) is called the *root* directory, also referred to as / ("slash").
- Every Linux user has a *home* directory, which typically stores the user's files.
- Every Linux process — including your login shell process — has its own *current directory*, which it can change at any time. You can think of the current directory as the directory where the process is "located" in the filesystem hierarchy.
- A *pathname* is the location of something (a file, a directory, a FIFO, etc.) in the filesystem. There are two types of pathnames: *absolute pathnames* (also called *full pathnames*) and *relative pathnames*. An absolute pathname always starts at / and is the direct route from the root directory to a filesystem entry. On the other hand, a relative pathname starts at the current directory, and it does *not* start with a slash. If you're trying to find something in the filesystem, you want to choose the shortest possible pathname.

Using Pathnames

Pathnames can mystify users — even experienced pros who've used Linux or Unix for years. Let's look at some examples to show how pathnames work.

Figure One shows a sample filesystem. / contains four subdirectories, *prj*, *bin*, *etc*, and *home*, where the latter stores users' home directories named *al*, *fox*, *jo*, and *root*. *al* contains a file named *core* (a debugging file saved as a program crashes), and *jo* contains two subdirectories, *bin* and *prj*, and two files *afile* and *core*. The directory *home* is shown in bold to indicate that it's the *current directory*.

When you give a pathname to a Linux program, it follows the pathname to the end and opens whatever is there. If the pathname starts with a slash, it's an absolute path, so Linux starts searching at /; otherwise it starts at the current directory. Indeed, the main reason for having a current directory is to make relative pathnames conveniently short.

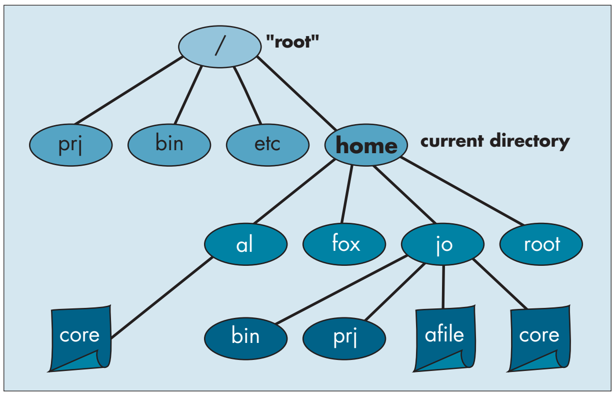
For example, if you type *less afile*, *less* (actually, a library routine that *less* uses) interprets `afile` as a relative pathname, opens the current directory, and looks for an entry named *afile*. If there's an entry for *afile*, that file is opened; otherwise, you get an error like `afile: No such file or directory`.

In our sample filesystem, if you ran the command in the current directory, */home*, you'd get an error because there's no *afile* in it. However, you can read the file *afile* in *jo*'s home directory by typing `less jo/afile`. What happens? `jo/afile` is also a relative pathname, so Linux opens the current directory to look for an entry named *jo*. That's a subdirectory, so it opens that directory to look for an entry named *afile*. In this case, *afile* exists, so *less* opens and shows it.

What other pathname could you have used? You could use an absolute pathname in the command, like `less /home/jo/afile`. In this case, Linux starts at the root directory, finds the *home* entry there, finds the *jo* entry in *home*, and then opens *afile*. In this case, a relative pathname was shorter and easier.

Note that simply using a pathname doesn't change your current directory. To change your current directory you have to give a pathname to the *cd* command.

FIGURE ONE: A simplified filesystem tree



If you'll be doing a lot of work with entries in a particular directory, use the *cd* command to make that directory your current directory. Then you can use short relative pathnames instead of typing long absolute paths. But, there's no rule that says you have to change your current directory! If you want to access something in another directory, you can always keep your same current directory and type an absolute path or a long relative path.

For instance, if you're currently in your home directory (you can always get to your home directory by typing `cd ~` or just `cd`) and want to edit a file in the */prj* directory, you can type `vi /prj/somefile` — and never change your current directory. When you leave *vi*, you'll still be in your home directory.

Using Many Paths at Once

Most non-GUI Linux utilities can open a series of files — just name them in series on the command line, separated by spaces. (GUI applications typically make you select files one by one.) A *shell* reads the Linux command line, and a *wildcard* like `*` tells the shell to *build a list of pathnames* — absolute or relative — that match the pattern.

For instance, if the system administrator is cleaning out users' home directories, he could type `cd /home` (making */home* the current directory) and then:

```
$ file */core
al/core: ELF 32-bit LSB core file of 'prog'
jo/core: ELF 32-bit LSB core file of 'sgorp'
```

What happened? The shell saw the wildcard, looked in the current directory, and built a list of all relative pathnames that have a single slash and end with *core*. It passed that list of paths (`al/core jo/core`) to the *file* program, which tells what kind of file each pathname points to. If the system administrator wanted an *ls* listing of the files, he (or she) could type `ls -l */core`. Note that he didn't need to *cd* to those users' home directories; a wildcard and some relative pathnames did the job more quickly.

Of course, *core* files don't get dumped just in home directories. Here's where *find* comes in:

```
$ find . -type f -name core -print
./al/core
./jo/core
./jo/prj/core
```

The system administrator told *find* to start in the current directory (`.`) and look downward in its hierarchy for all files named *core*. *find* generated a series of relative pathnames starting with `./`. (As the sidebar "What's in a directory?" explains, the leading `.` is the relative path to the current directory. *find* always starts its pathnames with the directory you specify. You can ignore this "no-op" `./`.)

If the system administrator wanted to remove the *core* files, he could type the following command, using `-i` to be prompted before each file:

```
$ rm -i al/core jo/core jo/prj/core
rm: remove al/core? y
rm: remove jo/core? y
rm: remove jo/prj/core? y
```

But there's an easier way. The shell operator `$( )` replaces a command with its output. So you can tell the shell to use the output of *find* as arguments to `rm -i` like this:

```
$ rm -i $(find . -type f -name core -print)
rm: remove ./al/core? y
rm: remove ./jo/core? y
rm: remove ./jo/prj/core? y
```

That's called *command substitution*, and it's one of the powerful shell features that are hard to duplicate in a GUI. (In *tcsh*, *csh* and early Bourne shells, command substitution uses backquotes `' '` instead of `$( )`. In fact, backquotes work in *all* shells.)

Of course, if the system administator wanted to remove *all* of the *core* files (in the first two levels of subdirectories), he could have used `rm */core */*/core`.

Keeping Track of Many Pathnames

What if the *find* had found many more files? What if the system administrator wanted to run *file* first, to be sure all *core* files were really program debugging output, but then not need to re-enter all of those filenames that *are* files to remove?

The command substitution example above hints at the powerful ways that shells have for building command lines with pathnames. There are plenty of other ways — we'll see some in this article and more later.

For example, instead of running *find* over and over to get a list of pathnames for each command, you can use command substitution to store the pathnames in a shell variable. In the next example, the first command stores the pathnames in a shell variable named `paths`. The second command runs *file* to see what sort of file each of the pathnames leads to. The third runs *rm* after the system administrator is sure they're all safe to remove:

```
$ paths=$(find . -type f -name core -print)
$ file $paths
./al/core: ELF 32-bit LSB core file of 'prog'
 ...
$ rm $paths
```

The system administrator might find some *core* files that should *not* be removed. (Maybe those files have core concepts for a new product.) How can he make a list of files that *should* be removed? One great trick is to redirect *find*'s output into a temporary file, then edit the file to include pathnames you want to remove. (This trick also works well with commands like `ls -lt`, which produces lists of files with lots of info, sorted by date.)

```
$ file $(find . -type f  -name core -print)
  > /tmp/x
$ vi /tmp/x
$ rm $(cat /tmp/x)
```

As *find* runs, the pathnames it prints are gathered by command substitution and passed to the *file* program. The output of *file* is redirected to a temporary file named */tmp/x*. A quick session in *vi* leaves only the pathnames of files to remove. The third command uses command substitution again: this time, *cat* emits the edited list of pathnames onto the command line of the *rm* program, which removes them. (The first few times you do this, add an *echo* before the *rm* command — `echo rm $(cat /tmp/x)` — to see what *rm* would do without actually doing it. *echo* simply outputs its command-line arguments.)

Another great pathname-cruncher is *xargs*. It's similar to command substitution, but the GNU version (which probably comes with your Linux system) has an advantage. Let's look at the previous example. What if one of the pathnames had a space in it, like *./al/my files/core*? That's a legal pathname, but when command substitution sees that pathname, it can treat the space as an argument separator and break it into two separate paths, *./al/my* and *files/core*. You can spot that while editing the */tmp/x* file, but automated use of *rm* on broken pathnames can cause big problems!

The solution is to use the *find* operator `-print0` and the *xargs* option `-0`, as shown below (both of those are the digit zero). Instead of outputting a newline character after each pathname, as `-print` does, the `-print0` operator outputs an ASCII `NUL` character after each pathname. *xargs* reads the NUL-separated pathnames and feeds them to *rm* without the dangerous pathname-breaking at space and newline characters. Here's a command that searches for all *core* files and removes them without asking:

```
$ find . -type f -name core -print0 |
xargs -0 rm
```

xargs reads pathnames from *find* and executes *rm* with those arguments. This is more efficient than running one *rm* command for each pathname, as the *find* operator `-exec rm {} \;` would do.

Well, there's much more to cover, but we're out of room for this month! To summarize briefly: a pathname comes in *relative* and *absolute* styles. The main purpose of a current directory is to make relative paths shorter. But you don't need to *cd* everywhere — you can use paths to other directories whenever it's convenient. Finally, Linux has some powerful ways to use many paths on one command line, which can save you a lot of work.

And that's what power tools are all about: saving you work and giving you more time to do the good stuff. (Now, what's the pathname to my copy of Quake?)

Jerry Peek is a freelance writer and instructor who has used Unixn and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

WHAT'S IN A DIRECTORY?

When you say that a file is "in a directory," what does that really mean? It means that the file's *name* is listed in a special kind of file called a *directory file*. On traditional filesystems, at least, a directory holds a series of filenames and *i-numbers*. Here's a simplified picture of what our */home* directory file could contain:

```
. 3335
.. 264
root 4261
al 13256
jo 43600
fox 50133
```

People use the names to locate a file. Deep down, though, Linux uses the i-numbers. If your current directory is */home* and you type the command `cd al`, Linux sees that *al* is a relative pathname (no slash at the start), so it checks the current directory for an entry named *al*. As you can see, that's i-number 13256. So Linux opens the *inode* (filesystem index node) with i-number 13256 and locates the actual *al* directory on the disk.

Notice the special entries named `.` and `..`. Every directory has those entries. The `.` is a link to the current directory, and `..` is a link to the parent directory (the directory that contains this directory). So, for instance, if you type `cd ..`, Linux opens the directory at i-number 264.

"So," you might ask, "if the root directory has i-number 264, what's that entry named *root?*" The answer is that the directory named *root*, here with i-number 4261, is the home directory for the user named *root*, the superuser. Its absolute pathname is */home/root* — very different than the root directory, with absolute pathname */*.)

Note that our current directory is named *home*, but you can't tell that from looking at the directory entries. Where is its name? A directory's name is kept in its parent directory! But *all* directories have a link named ., which is a handy way to refer to the current directory — from the *find* program, for instance.