Wizard Boot Camp, Part One

Jerry Peek

Over the past five years, this column has covered a lot of not-so-obvious features of *Linux* — mostly techniques that you'd use from the shell or another non-graphical environment. The start of your columnist's sixth year seemed like a good time to try something a little different: a series of columns on features of Linux and shells that many books about Linux basically ignore, but that wizards should know. Studying these obscure features isn't just for trivia buffs. Understanding how each feature works can help you grok "the big picture" of some pieces that make your favorite operating system the power platform that it is.

This month, let's start with a quick overview of Linux and shell concepts that you should know, then dig into some shell features that might be new to you.

Linux Overview in Ten Paragraphs

Let's review some of the fundamentals to start from the same place. (Please note: this isn't an exact, complete description. Some details are simplified or omitted.)

A Linux system manages a set of *processes*, letting them share system resources (the processors, disks, networks, and so on). Basically, a process is a running program, a program waiting to run, or a suspended program. A process has a name (often the name of the program file that it's executing) and a list of arguments (for instance, some filenames). Each process has a unique process ID, or PID, that's assigned when the process starts. The *ps* utility lists processes.

A process runs under the user ID, or UID, of one of the accounts on the system. In most cases, a process runs with the UID of the account it was started under. So, if user *joe* starts a text editor, that process inherits the abilities that *joe* has: for instance, it can read and write his files. The superuser, *root*, has UID 0. A process running under UID 0 is *privileged*: it can change its UID (actually, its *effective UID*) to any other UID on the system. The *su* and *sudo* programs, which run under UID 0, allow any user to run processes as another user.

A parent process can start other processes called *child processes*. A child process inherits many of its parent's attributes — as you'll see next. A child process can also start its own child processes. The process named *init*, with PID 1, is the parent (or "grandparent", or great-grandparent, and so on) of all processes. After a parent process starts a child process, it typically waits for the child to finish. However, if the parent doesn't wait, it can "disown" the child, and that "orphaned" process becomes a child of *init*.

A process has attributes including a *current directory* (the starting place for *relative pathnames*); a list of *environment variables* (for example, `EDITOR=emacs` or `EDITOR=vi`); and a list of *open files* for input/output.

The open files include three as *standard input and output:* the standard input, abbreviated *stdin*, file descriptor number 0; standard output, *stdout*, with descriptor 1; and standard error, *stderr*, with file descriptor 2.

A child process inherits attributes from its parent. For instance, the child starts in the same current directory as its parent, and has the same open files and environment variables. Once the child begins running, it can manage these attributes, and any changes made in the child process don't affect its parent.

When a process starts, its standard I/O files are often connected to a terminal device, such as an *X Window System* terminal or the system console. So, for instance, if a process prints a message to its *stdout*, the text appears on the terminal. Because a child process inherits its parent's open files, it too writes to and reads from the same terminal (by default) as its parent.

When a process exits, it returns a numeric *exit status* to its parent. By convention, a zero status signals success and a non-zero status means there was some kind of error.

If a process was started under an X Window System, it can also open windows there. Then, although that process also has standard I/O available, it generally won't use them.

Shells in Five Easy Steps

One particularly interesting kind of program is the *shell*. A shell is a program designed especially to run other programs (usually, non-shell programs — though a shell can start a child shell). Here's how a shell works:

1. The shell outputs a prompt and waits for input (unless the shell is reading a *shell script* file).

2. It reads the command-line, parses the command-line to find the command names, handles operators like | and > (which tell the shell to redirect the child's standard I/O), and interprets any arguments (and special characters in them, such as *wildcards*).

3. The shell then starts one or more child processes to run the specified program(s), unless the command you specified is *built-in* to the shell. (For example, the built-in command *cd* changes the shell's current directory. It doesn't

start a child process because the command needs to affect the current process.)

4. It waits for the program to finish (unless you suspend the program after it's started running by typing Control-Z, or if you started the program *in the background* by typing `8-;` at the end of the command-line).

5. The shell then repeats step 1, unless the script file ends or you used the built-in command *exit*, in which case the shell process terminates.

There are several different shells. This series covers *bash*.

How Shells Get Commands

A shell is a *command interpreter*. Commands can come from several places:

- A shell can run a command provided as an argument. You might do this from a non-shell program that needs to run a command-line entered by a user. Most shells have a –c option for this purpose. For example, to cd to the user's subdirectory named lib, list its contents, and rename a file, another program could execute:

```
$ sh -c "cd ~/lib 8-8-; ls; mv afile afile.old"
```

The shell expands ~ into the absolute pathname of the user's home directory; the result is passed to *cd*. The `8-8-` operator executes *ls* only if *cd* succeeded. The `;` operator executes *mv* after *ls* finishes.

- If you pass an argument that isn't associated with an option, the shell opens that argument as a file and reads command-lines from it. This is how shell scripts are read.

This method and the previous one make the shell *non-interactive*: the shell doesn't prompt a user for input, it simply reads one or more commands and executes them. After the last child process exits, the shell process itself exits.

- A shell invoked without any arguments or only with certain options runs in *interactive* mode. After reading initialization files like *.bashrc*, the shell outputs a prompt string and waits to read input from *stdin*.

The latter is the most complex (and interesting) way that commands are input. Let's look into it some more.

Linux newbies think of a shell reading everything they type on the keyboard until the Return (Enter) key is pressed. That's the simplest case. Shells like *bash* with built-in command line editing can edit multiple-line commands like `while` and `for` loops. You might be wondering how multi-line statements like these are entered from a prompt.

As you're entering a command-line, the shell reads the line you've typed (actually, the text from its standard input) when you press Enter. The shell checks the line to be sure it's a complete statement. If it's not complete, the shell prints its *secondary prompt*, `>`, and keeps reading input until it finds the end of the statement.

What's an incomplete statement? Here are some examples:

- A line with unmatched single quotes (`'`), double quotes (`"`), and backquotes (`` ` ``); unquoted opening parentheses `(` and curly braces `{`, and so on.

For instance, to make a shell script output a multi-line message, you don't need to use one *echo* command for each line. Instead, pass a multi-line argument (containing newline characters) to *echo*, surrounded by quote marks.

```
$ echo 'This is the first line
and this is the second line.'
```

Here's how that example looks when it's typed at the primary (`$`) and secondary (`>`) shell prompts:

```
$ echo 'This is the first line
> and this is the second line.'
This is the first line
and this is the second line.
```

- A loop before you type `done`. Here's a `for` loop that finds all *TIFF* images (all filenames ending with *.tif*) in the current directory), strips off the `.tif`, and uses the *ImageMagick convert* utility to create a *JPEG* image:

```
$ for file in *.tif
> do
>   echo "doing $file..."
>   base=${file%.tif}
>   convert "$file" "$base.jpg"
> done
```

- An unfinished pipeline. Typing a pipe symbol at the end of a line tells the shell that there's another command to be read. Writing pipelines this way can make a shell script easier to read. You can intersperse comments, too:

```
# List the 20 largest files:
ls -S | head -20 |
# open and search for errors:
xargs grep ERROR
```

Problematic Ways that Shells Get Commands

The previous section discussed shells reading commands from standard input. A shell can also read commands from its standard input non-interactively via redirection:

```
$ bash < script-file
$ command-generator | bash
```

script-file is a shell script file, and *command-generator* is a program that outputs shell command-lines for *bash* to execute.

The problem here is that the standard input of *bash* is redirected. So, if any of the commands in *script-file*, or in the output of *command-generator*, try to read from the standard input, they'll take input from *script-file* or *command-generator*, which is almost certainly not the right place!

This is a big "gotcha" in redirecting a shell's standard input. You may be able to work around this with the shell operator `m<8n` (which we'll cover in November), but it's likely to be painful at best.

The Shell Expands Wildcards

A wildcard is a character like `*` or `?` that stands for other characters in a filename or pathname. Linux shells interpret wildcards. They replace arguments containing wildcards with a list of one or more matching names. However, shells don't replace quoted wildcards.

If an unquoted wildcarded argument doesn't match any pathnames, what happens depends on the shell you're using.

Shell Quoting

It's common knowledge that quoting special characters — that is, preceding a special character with a backslash (like `afile\*`) or surrounding its argument with quotes (like `"afile*"` or `'afile*'`) — disables those characters' special meanings. The shell strips off the quoting character (s) and passes the special character, as-is, as part of the argument. Hence, `cp afile*somedir` copies all filenames starting with *afile* into *somedir*, and `cp afile\ *somedir` copies the file named `afile*` to *somedir*.

A pair of single quotes (`'`) is "stronger" than a pair of double quotes (`"`). Within double quotes, a few special characters keep their meanings, including `$` for variables (like `cp"$var"somedir`), and backquotes, as in `message="The files are: `ls`."`. Knowing this helps you decide what kind of quotes to use.

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.