COMPUTER POWER USER

Tips & Tricks

Click To Print

Tips & Tricks
July 2008 • Vol.8 Issue 7
Page(s) 100-101 in print issue

Warm Up To Penguins

Shell Games

"Hi, this is Andy Rooney. Have you ever wondered just what happens when you type a command into a Linux shell? I know I sure have."

OK, so we're unlikely to see this as an essay topic on "60 Minutes" anytime soon, but understanding how a command you enter turns into a running program is a topic worth understanding. There are basically three types of commands (or programs) that you can invoke from the command line: built-ins, binaries, and scripts.

Built-ins are just that: commands that are built into the shell itself. For example, the "set" command is an intrinsic part of shells such as bash, csh, and sh. There's no separate program that runs when you set a shell variable. You're just using a different piece of the same program that's already giving you your shell prompt.

Binaries and scripts exist in files separate from the shell itself. When you type the command **echo test**, for example, you're actually running a program called echo. The shell figures out where to find the command by looking at a shell variable called PATH. For example, here's a standard path under bash:

james@james-ubuntu-test:~$ **echo $PATH**
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

This says that if we type a command, the first place to look for the file that matches the name of the command in the /usr/local/sbin directory, then /usr/local/bin, then /usr/sbin, and so on. If the shell gets to the end of the list and hasn't found a directory that contains a file with that name, it will return an error saying it can't find that command.

In file listings, executable files will show up in green.

Binaries are programs that have been compiled from a programming language, such as C or C++, into the native instruction set of your computer's processor. Scripts, on the other hand, are human-readable code that a program interprets when you invoke them. (At least you can hope that they're human-readable; we've seen some pretty unintelligible scripts.) In Linux, you'll commonly see scripts written in Perl, Python, Ruby, and the various shells (sh, csh, bash.)

One way to tell if a command is a script or a binary is to use the "more" command on it. For example, if we try doing a more on /bin/echo, we get:

james@vm2:~$ **which echo**
/bin/echo
james@vm2:~$ **more /bin/echo**
******* /bin/echo: Not a text file ********

This tells us that /bin/echo is a binary. If you want to make sure, you can use the "file" command, which gives you lots of useful information about a file. For example, when we used file on the same command:

james@vm2:~$ file /bin/echo

/bin/echo: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.8, dynamically linked (uses shared libs), stripped

For the curious, this tells us the following: "echo" is stored using ELF (Executable and Linkable Format) for binaries; it is compiled for x86 processors; it requires the use of libraries in /usr/lib to run; and it has had all debugging information stripped out

of it.

For comparision, let's look at the "file" output for a pair of scripts.
james@vm2:~$ **file /usr/bin/gdialog**
/usr/bin/gdialog: perl script text executable
james@vm2:~$ **file /usr/bin/ldd**
/usr/bin/ldd: Bourne-Again shell script text executable

Looking at this, we can see that gdialog is a Perl script, while ldd is written using bash. (And yes, bash stands for Bourne Again Shell, a pun on the fact that it's based on /bin/sh, the Bourne Shell.)

How does Linux know whether a command is a script or binary? And if it's a script, how does Linux know what type of script? It knows by looking at the beginning of the file. For example, if the file is an ELF binary, the first characters in the file (in hexadecimal) will be 7F 45 4C 46. All scripts can be recognized by the fact that "#!" are the first two characters in the file name. In the case of scripts, the path to the program that can interpret the script must immediately follow the "#!." For example, a Perl script would start "#!/usr/bin/perl." A bash script, on the other hand, would begin with "#!/bin/bash."

Roll Your Own Script

With this information under our belt, we're ready to try our hand at writing a script. All of the shell scripting languages (sh, csh, and bash) are a little arcane to use, but using Perl to write scripts may be easier. Perl's very similar to C or Java. But for this example, we'll stick to something simple, so bash will do fine.

Let's start by writing a command that honors a great programming tradition, one that prints out "Hello, World!" The very first examples of this program can be found in the Brian Kernighan's book "The C Programming Language," which has been a tradition to use as a first programming example for a language ever since. So, here's our "Hello, World!" program, written in bash.

```
#!/bin/bash
echo "Hello, World!"
```

There's not much to look at, is there? The first line identifies the script as a bash script and tells Linux where to find the program that can interpret the commands. The second line is a normal command you could type into a bash command line prompt. Assuming that we saved these lines into a file called "helloworld," we could then try to run it. After typing **helloworld** at the prompt, we get the following output:

```
bash: helloworld: command not found
```

Oops, that didn't work so well. There are two reasons we weren't successful: the path and the permissions on the file. Remember when we looked at the PATH variable? There was a list of directories that the shell would search when you typed in a command name. What was missing from that list was the "current" directory, which is accessed in Linux using the "." character. So "./helloworld" would mean "the file helloworld in the current directory." You could add "." to the end of the list of directories in the PATH variable, but there are security reasons not to do that. Instead, we can just rerun the command specifying the directory.

```
james@vm2:~$ ./helloworld
bash: ./helloworld: Permission denied
```

Well, that's different but still not what we want. The other thing that a file has to have to be considered a run-able script is to have the "execute bit" turned on. Every file in Linux has three types of permission: read, write, and execute. Furthermore, these permissions can be different for the owner of the file, people in the same group as the file, and the world in general. For the purposes of this example, all you need to know is that helloworld doesn't have execute rights set for anyone, something you can see by using the "ls" command with the "-l" (long) argument.

```
james@vm2:~$ ls -l helloworld
-rw-r--r-- 1 james james 33 2008-04-20 21:27 helloworld
```

The important part of the output is the first part, which tells us that we have read and write permission as owner and that group and "world" have read permission only. We can change that using the chmod (command):

```
james@vm2:~$ chmod +x helloworld
james@vm2:~$ ls -l helloworld
-rwxr-xr-x 1 james james 33 2008-04-20 21:27 helloworld
james@vm2:~$ ./helloworld

Hello, World!
```

To finish off our look at scripts, let's write a script that actually does something. The file /etc/passwd has an entry for every user that the system knows about. If we want to get the entry for a given user, we can do it by typing the following:

james@vm2:~$ **grep james /etc/passwd**
james:x:1000:1000:James Turner,,,:/home/james:/bin/bash

The grep command re-turns any lines in a file that match the specified string. Let's make a command, finduser, that will do it for any user we pick. Here's the script:

```
#!/bin/bash
grep $1 /etc/passwd
```

As you can see, "$1" is the only new thing we added. Any time you use something that starts with a $ in a shell, you're referring to a variable. For example, "$PATH" at the beginning to the article is the way to get the value of the PATH variable. In any script, $1 is the first argument used on the command line that invokes the script. (The $0 variable, by the way, is the name of the script itself.) So, once we add the execute bit to the file, we can take it for a test drive.

james@vm2:~$ **chmod +x finduser**
james@vm2:~$ **./finduser root**
root:x:0:0:root:/root:/bin/bash

And there we go, the $1 variable is replaced by the argument "root," and the command outputs the results of the grep command.

We've spent the last few months digging around the guts of Linux from the command line. Next month, we'll get back to looking at some cool Linux applications, starting with a peek at audio editing with Audacity. ■

by James Turner

I Am Iron Man . . . Sort Of

Looking to relieve its overburdened troops, Canada's Defence Department solicited bids for military contractors to build "exoskeletons and other mobility devices" to "reduce the load burden for the soldier of tomorrow." The article would have you believe such a creation would be an Iron Man-type suit, which might be an accurate comparison except for the fact that it doesn't fly, has no armaments, and is completely free of an 18-year Glenlivet aroma.

Source: ca.news.yahoo.com/s/capress/080505/koddities/iron_man_canuck