The Bourne Again Shell

What Is A Shell?

Yeah, what exactly is a shell? Well, a shell is basically a command-line user environment. In essence, it is an application that runs when the user logs in and allows him to run additional applications. In some ways it is very similar to a graphical user interface, in that it provides a framework for executing commands and launching programs. There are many shells included with a full install of Slackware, but in this book we're only going to discuss ***bash***(1), the Bourne Again Shell. Advanced users might want to consider using the powerful ***zsh***(1), and users familiar with older UNIX systems might appreciate ***ksh***. The truly masochistic might choose the ***csh***, but new users should stick to ***bash***.

Environment Variables

All shells make certain tasks easier for the user by keeping track of things in environment variables. An environment variable is simply a shorter name for some bit of information that the user wishes to store and make use of later. For example, the environment variable PS1 tells ***bash*** how to format its prompt. Other variables may tell applications how to run. For example, the LESSOPEN variable tells ***less*** to run that handy `lesspipe.sh` preprocessor we talked about, and LS_OPTIONS tuns on color for ***ls***.

Setting your own envirtonment variables is easy. ***bash*** includes two built-in functions for handling this: ***set*** and ***export***. Additionally, an environment variable can be removed by using ***unset***. (Don't panic if you accidently unset an environment variable and don't know what it would do. You can reset all the default variables by logging out of your terminal and logging back in.) You can reference a variable by placing a dollar sign ($) in front of it.

```
darkstar:~$ set FOO=bar
darkstar:~$ echo $FOO
bar
```

The primary difference between ***set*** and ***export*** is that ***export*** will (naturally) export the variable to any sub-shells. (A sub-shell is simply another shell running inside a parent shell.) You can easily see this behavior when working with the PS1 variable that controls the ***bash*** prompt.

```
darkstar:~$ set PS1='FOO '
darkstar:~$ export PS1='FOO '
FOO
```

There are many important environment variables that ***bash*** and other shells use, but one of the most important ones you will run across is PATH. PATH is simply a list of directories to search through for applications. For example, ***top***(1) is located at ***/usr/bin/top***. You could run it simply by specifying the complete path to it, but if `/usr/bin` is in your PATH variable, ***bash*** will check there if you don't specify a complete path one your own. You will most likely first notice this when you attempt to run a program that is not in your PATH as a normal user, for instance, ***ifconfig***(8).

```
darkstar:~$ ifconfig
bash: ifconfig: command not found
darkstar:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/games:/opt/www/htdig/bin:.
```

Above, you see a typical PATH for a mortal user. You can change it on your own the same as any other environment variable. If you login as root however, you'll see that root has a different PATH.

```
darkstar:~$ su -
Password:
darkstar:~# echo $PATH
/usr/local/sbin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:
/usr/games:/opt/www/htdig/bin
```

Wildcards

Wildcards are special characters that tell the shell to match certain criteria. If you have experience with DOS, you'll recognize * as a wildcard that matches anything. ***bash*** makes use of this wildcard and several others to enable you to easily define exactly what you want to do.

This first and most common of these is, of course, *. The asterisk matches any character or combination of characters, including none. Thus b* would match any files named b, ba, bab, babc, bcdb, and so forth. Slightly less common is the ?. This wildcard matches one instance of any character, so b? would match ba and bb, but not b or bab.

```
darkstar:~$ touch b ba bab
darkstar:~$ ls *
b ba bab
darkstar:~$ ls b?
ba
```

No, the fun doesn't stop there! In addition to these two we also have the bracket pair *"[ ]"* which allows us to fine tune exactly what we want to match. Whenever ***bash*** see the bracket pair, it substitutes the contents of the bracket. Any combination of letters or numbers may be specified in the bracket as long as they are comma seperated. Additionally, ranges of numbers and letters may be specified as well. This is probably best shown by example.

```
darkstar:~$ ls a[1-4,9]
a1 a2 a3 a4 a9
```

Since Linux is case-sensitive, capital and lower-case letters are treated differently. All capital letters come before all lower-case letters in *"alphabetical"* order, so when using ranges of capital and lower-case letters, make sure to get them right.

```
darkstar:~$ ls 1[W-b]
1W 1X 1Y 1Z 1a 1b
darkstar:~$ ls 1[w-B]
/bin/ls: cannot access 1[b-W]: No such file or directory
```

In the second example, 1[b-W] isn't a valid range, so the shell treats it as a filename, and since that file doesn't exist, ***ls*** tells you so.

Tab Completion

Still think there's entirely too much work involved with using wildcards? You're right. There's an even easier way when you're dealing with long filenames: tab completion. Tab completion enables you to type just enough of the filename to uniquely identify it, then by hitting the TAB key, ***bash*** will fill in the rest for you. Even if you haven't typed in enough text to uniquely identify a filename, the shell will fill in as much as it can for you. Hitting TAB a second time will make it display a list of all possible matches for you.

Input and Output Redirection

One of the defining features of Linux and other UNIX-like operating systems is the number of small, relatively simple applications and the ability to stack them together to create complex systems. This is achieved by redirecting the output of one program to another, or by drawing input from a file or second program.

To get started, we're going to show you how to redirect the output of a program to a file. This is easily done with the '>' character. When ***bash*** sees the '>' character, it redirects all of the standard output (also known as stdout) to whatever file name follows.

```
darkstar:~$ echo foo
foo
darkstar:~$ echo foo > /tmp/bar
darkstar:~$ cat /tmp/bar
foo
```

In this example, we show you what ***echo*** would do if its stdout was not redirected to a file, then we re-direct it to the `/tmp/bar` file. If `/tmp/bar` does not exist, it is created and the output from ***echo*** is placed within it. If `/tmp/bar` did exist, then its contents are over-written. This might not be the best idea if you want to keep those contents in place. Thankfully, ***bash*** supports '»' which will append the output to the file.

```
darkstar:~$ echo foo
foo
darkstar:~$ echo foo > /tmp/bar
darkstar:~$ cat /tmp/bar
foo
darkstar:~$ echo foo2 >> /tmp/bar
darkstar:~$ cat /tmp/bar
foo
foo2
```

You can also re-direct the standard error (or stderr) to a file. This is slightly different in that you must use '2>' instead of just '>'. (Since ***bash*** can re-direct input, stdout, and stderr, each must be uniquely

identifiable. 0 is input, 1 is stdout, and 2 is stderr. Unless one of these is specified, ***bash*** will make its best guess as to what you actually meant, and assumed anytime you use '>' you only want to redirect stdout. 1> would have worked just as well.)

```
darkstar:~$ rm bar
rm: cannot remove `bar': No such file or directory
darkstar:~$ rm bar 2> /tmp/foo
darkstar:~$ cat /tmp/foo
rm: cannot remove `bar': No such file or directory
```

You may also redirect the standard input (known as stdin) with the '<' character, though it's not used very often.

```
darkstar:~$ fromdos < dosfile
```

Finally, you can actually redirect the output of one program as input to another. This is perhaps the most useful feature of ***bash*** and other shells, and is accomplished using the '|' character. (This character is referred to as 'pipe'. If you here some one talk of piping one program to another, this is exactly what they mean.)

```
darkstar:~$ ps auxw | grep getty
root      2632  0.0  0.0   1656   532 tty2     Ss+  Feb21   0:00
/sbin/agetty 38400 tty2 linux
root      3199  0.0  0.0   1656   528 tty3     Ss+  Feb15   0:00
/sbin/agetty 38400 tty3 linux
root      3200  0.0  0.0   1656   532 tty4     Ss+  Feb15   0:00
/sbin/agetty 38400 tty4 linux
root      3201  0.0  0.0   1656   532 tty5     Ss+  Feb15   0:00
/sbin/agetty 38400 tty5 linux
root      3202  0.0  0.0   1660   536 tty6     Ss+  Feb15   0:00
/sbin/agetty 38400 tty6 linux
```

Task Management

bash has yet another cool feature to offer, the ability to suspend and resume tasks. This allows you to temporarily halt a running process, perform some other task, then resume it or optionally make it run in the background. Upon pressing CTRL+Z, ***bash*** will suspend the running process and return you to a prompt. You can return to that process later. Additionally, you can suspend multiple processes in this way indefinitely. The ***jobs*** built-in command will display a list of suspended tasks.

```
darkstar:~$ jobs
[1]-  Stopped                 vi TODO
[2]+  Stopped                 vi chapter_05.xml
```

In order to return to a suspended task, run the ***fg*** built-in to bring the the most recently suspended task back into the foreground. If you have mutiple suspended tasks, you can specify a number as well to bring one of them to the foreground.

```
darkstar:~$ fg # "vi TODO"
```

```
darkstar:~$ fg 1 # "vi chapter_05.xml"
```

You can also background a task with (surprize) ***bg***. This will allow the process to continue running without maintaining control of your shell. You can bring it back to the foreground with ***fg*** in the same way as suspended tasks.

Terminals

Slackware Linux and other UNIX-like operating systems allow users to interact with them in many ways, but the most common, and arguably the most useful, is the terminal. In the old days, terminals were keyboards and monitors (sometimes even mice) wired into a mainframe or server via serial connections. Today however, most terminals are virtual; that is, they exist only in software. Virtual terminals allow users to connect to the computer without requiring expensive and often incompatible hardware. Rather, a user needs only to run the software and they are presented with a (usually) highly customizable virtual terminal.

The most common virtual terminals (in that every Slackware Linux machine is going to have at least one) are the gettys. ***agetty***(8) runs six instances by default on Slackware, and allows local users (those who can physically sit down in front of the computer and type at the keyboard) to login and run applications. Each of these gettys is available on different tty devices that are accessible seperately by pressing the ALT key and one of the function keys from F1 through F6. Using these gettys allows you to login multiple times, perhaps as different users, and run applications in those users' shells silmutaneously. This is most commonly done with servers which do not have ***X*** installed, but can be done on any machine.

On desktops, laptops, and other workstations where the user prefers a graphical interface provided by ***X***, most terminals are graphical. Slackware includes many different graphical terminals, but the most commonly used are KDE's ***konsole*** and XFCE's ***Terminal***(1) as well as the old standby, xterm(1). If you are using a graphical interface, check your tool bars or menus. Each desktop environment or window manager has a virtual terminal (often called a terminal emulater), and they are all labelled differently. Typically though, you will find them under a *"System"* sub-menu in desktop environments. Executing any of these will give you a graphical terminal and automatically run your default shell.

Customization

By now you should be pretty familiar with ***bash*** and you may have even noticed some odd behavior. For example, when you login at the console, you're presented with a prompt that looks a bit like this.

```
alan@darkstar:~$
```

However, sometimes you'll see a much less helpful prompt like this one.

```
bash-3.1$
```

The cause here is a special environment variable that controls the ***bash*** prompt. Some shells are considered *"login"* shells and others are *"interactive"* shells, and both types read different configuration files when started. Login shells read `/etc/profile` and `~/.bash_profile` when

executed. Interactive shells read `~/.bashrc` instead. This has some advantages for power users, but is a common annoyance for many new users who want the same environment anytime they execute ***bash*** and don't care about the difference between login and interactive shells. If this applies to you, simply edit your own ~/.bashrc file and include the following lines. (For more information on the different configuration files used, read the INVOCATION section of the ***bash*** man page.)

```
# ~/.bashrc
. /etc/profile
. ~/.bash_profile
```

When using the above, all your login and interactive shells will have the same environment settings and behave identically. Now, anytime we wish to customize a shell setting, we only have to edit `~/.bash_profile` for user-specific changes and `/etc/profile` for global settings. Let's start by configuring the prompt.

bash prompts come in all shapes, colors, and sizes, and every user has their own preferances. Personally, I prefer short and simple prompts that take up a minimum of space, but I've seen and used mutli-line prompts many times. One personal friend of mine even included ASCII-art in his bash prompt. To change your prompt you need only to change your PS1 variable. By default, Slackware attempts to configure your PS1 variable thusly:

```
darkstar:~$ echo $PS1
\u@\h:\w\$
```

Yes, this tiny piece of funny-looking figures controls your ***bash*** prompt. Basicaly, every character in the PS1 variable is included in the prompt, unless it is a escaped by a \, which tells ***bash*** to interpret it. There are many different escape sequences and we can't discuss them all, but I'll explain these. The first *"\u"* translates to the username of the current user. *"\h"* is the hostname of the machine the terminal is attached to. *"\w"* is the current working directory, and *"\$"* displays either a # or a $ sign, depending on whether or not the current user is root. A complete listing of all prompt escape sequences is listed in the ***bash*** man page under the PROMPTING section.

Since we've gone through all this trouble to discuss the default prompt, I thought I'd take some time to show you a couple example prompts and the PS1 variable values needed to use them.

```
Wed Jan 14 12:08 AM
alan@raven:~$ echo $PS1
\d \@\n\u@\h:\w$
HOST: raven - JOBS: 0 - TTY: 3
alan@~/Desktop/sb_3.0:$ echo $PS1
HOST: \H - JOBS: \j - TTY: \l\n\u@\w:\$
```

For even more information on configuring your bash prompt, including information on setting up colored prompts, refer to `/usr/doc/Linux-HOWTOs/Bash-Prompt-HOWTO`. After reading that for a short while, you'll get an idea of just how powerful your ***bash*** prompts can be. I once even had a prompt that gave me up to date weather information such as temperature and barometric pressure!

Chapter Navigation

Previous Chapter: Basic Shell Commands

Next Chapter: Process Control

Sources

- Original source: http://www.slackbook.org/beta
- Originally written by Alan Hicks, Chris Lumens, David Cantrell, Logan Johnson

slackbook, bash, task management, terminals