Mastering Linux, part 10

In this month's instalment of the Mastering Linux series, *Jarrod Spiga* outlines some essential shell commands.

By now, you should be familiar with many shell commands. But it's only when you get into shell scripting that you can truly understand how these commands can be tied together to easily and quickly perform very complicated tasks. Shell scripting even allows you to design your own commands and macros — all on the command line.

However, there are a few other commands which need to be discussed as they are invaluable for creating scripts.

MORE THAN ECHOES TALK ALONG THE WALLS

The `echo` command simply takes what has been supplied as an argument, and prints it to the standard output (the next line of the shell).

When used by itself, the echo command almost seems trivial. Why would you have the shell return a string of text you've just given it? However, echo is invaluable to a script writer, as it allows the script to display information onscreen while it's being executed. For instance, it could be used to ask a question of the user while an interactive script is run.

When using echo, it's a good idea to encapsulate the text that you want to appear in quotation marks. This way, special characters such as asterisks and question marks appear as intended and aren't used for other purposes such as pattern matching.

A CALCULATED EXPRESSION

1 There may be times when a script needs to perform a basic mathematical calculation. For instance, use the following command to add 21 to 42:

```
expr 42 + 21
```

The `expr` command doesn't support floating point operations, so the numbers you work with must be integers. The numbers and operands in your command should always be separated with a space, as each of these elements needs to be treated as its own argument to the command.

Addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and modulus (`%`) operations are possible with the command. When multiplying, be sure to enclose the asterisk in quotation marks to prevent pattern matching.

It can also be used to perform comparison functions, but this is outside the scope of this article. For more information, type `man expr` at the shell.

KNOWING YOUR HEAD FROM TAIL

2 Often, you'll be required to parse the first or last few lines of a file through a script in order to perform a task. The `head` command returns data from the beginning of a file, while `tail` returns data from the end. To display the first five lines of a file, use:

```
head -5 filename.txt
```

To see the last 5 lines of the file, substitute the head command for tail. If you leave out the numeric switch, both commands will return a total of 10 lines.

The tail command has a few more switches than head, `-f` (follow) is one of the most common. When used alone, the last 10 lines of the specified file are displayed, but you won't be taken back to a command prompt. The command continues to run and displays any lines added to the file. This is handy when processing a stream of data.

WHEN ALL IS SED AND DONE

3 The `sed` (stream editor) command performs single-pass transformation of streams of data which is usually piped through to the command. It's often used to filter, substitute and/or insert data into various parts of a stream of data.

For instance, imagine you need to list the contents of a configuration file, but don't want the comment lines in the file to get in the way. In this case, you could use the sed command as follows:

```
cat config.txt | sed '/^#/d'
```

To understand what this command does, break it

Bonus DVD software

PDFs of every instalment of the Mastering Linux series.

Skill level

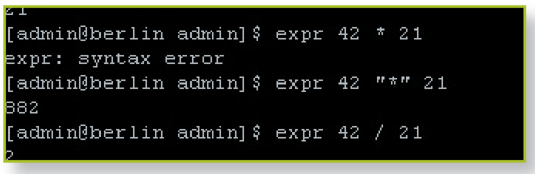
Intermediate

Requirements

An installation of Linux (Fedora Core 3 was used in the writing of this article).

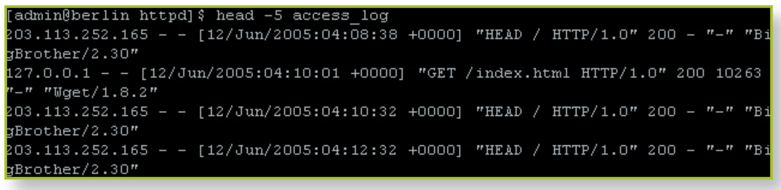
Time to complete

2 hours (approx)



```
21
[admin@berlin admin]$ expr 42 * 21
expr: syntax error
[admin@berlin admin]$ expr 42 "*" 21
882
[admin@berlin admin]$ expr 42 / 21
2
```

1 ***Expression in action:*** *you'll need to surround the multiplication (*) symbol with quotes for it to work. And remember, expr only works with integers.*



```
[admin@berlin httpd]$ head -5 access_log
203.113.252.165 - - [12/Jun/2005:04:08:38 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
127.0.0.1 - - [12/Jun/2005:04:10:01 +0000] "GET /index.html HTTP/1.0" 200 10263
"-" "Wget/1.8.2"
203.113.252.165 - - [12/Jun/2005:04:10:32 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:04:12:32 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
```

2 ***Heads or tails:*** *if you need to find out the contents of the beginning or end of a file, use the head or tail command.*

down into parts. The portion before the pipe lists the contents of the file in question — the output is then passed on to the second half of the command. The d tells sed to delete the lines that match the prefixed filter.

The filter is the complicated part. All filter expressions must be surrounded by a forward slash (/). The caret (^) is a special character used to match the beginning of a line. The table below details what all of these special characters are, and some common usages. The hash (#) is used as the next character in the filter because all comment lines in a configuration file will start with one.

Special characters	
`^`	Matches the beginning of a line.
`$`	Matches the end of a line.
`.`	Matches any single character.
`*`	Matches zero or more occurrences of the previous character.
`[]`	Matches any of the characters between the brackets.
Frequently used combinations	
`/./`	Matches all lines containing at least one character.
`/^.$/`	Matches lines containing only one character.
`/^#/`	Matches all lines starting with a hash.
`/^$/`	Matches all blank lines.
`/) *$/`	Matches any line ending with a closed-parenthesis followed by zero or more spaces.
`/^[xyz]/`	Matches any line beginning with a lower-case x, y or z

SEARCH AND REPLACE

4 The example above is fine for deleting entire lines of output, but what if you need to remove a few words from any given line in a file? The search and replace function of sed is useful for this.

Imagine a text file containing a large amount of data and a few confidential email addresses. The following command could remove those email addresses from the file:

```
cat document.txt | sed 's/[:blank:
][:graph:]*@[:graph:]*.[:graph:]*
[:blank:]/<confidential>/g'
```

Dissecting the sed command, the first character to address is s (for substitute). What appears between the first two forward slashes is the search string — [:blank:] matches any space or tab character, while [:graph:] matches any visible character. The search term looks for a space or tab, followed by zero or more visible characters (the alias), the @ character, then zero or more characters (the domain name, minus the top-level domain suffix), then a period, then zero or more characters (making up the suffix) and finally, another space or tab. The string appearing between the second and third forward slashes is what is substituted in to the area matched by the search string.

Lastly, the g (global) option replaces all instances of the search string on all lines. If you don't use this option, only the first email address on each line will be replaced, leaving the rest visible.

Overall, sed is a very powerful command line tool capable of much more than has been demonstrated in this Workshop. A more in-depth sed tutorial can be found at **http://www-106.ibm.com/developerworks/linux/library/l-sed1.html**. Of course, the sed man page is also very useful, albeit brief.

SHELL VARIABLES

Variables are a fundamental part of most shell scripts. In the same way as variables are used in algebra, values can be assigned to shell variables, each defined by a unique name. There are two methods to using variables:

1. Declaring the variable is usually done by assigning a value to a unique keyword using the equals sign. For example, to abbreviate the location of where the web root on your system is located, you could define:

```
www=/var/www/html
```

2. Using variable substitution to access the contents of a variable by preceding the variable name with a dollar sign. Using the example above, change the working directory to your web root by entering:

```
cd $www
```

You can find out what the value of a variable is at any time by echo-ing it to the shell:

```
echo $www
```

ENVIRONMENT VARIABLES

Another special class of variables exist on a Linux system. Environment variables provide the shell with instructions on how it should behave under certain conditions. The most frequently used environment variable is PATH.

PATH contains a list of directories that hold executable commands or scripts. When you enter a command into the shell, it first searches the present working directory for the command. If the directory can't locate it, it searches the locations defined in the PATH variable in order until the command is found. If the command isn't found in any of the PATH locations, an error message will appear.

Consider the echo command. It's actually located in the /bin/ directory on Fedora systems. Because of the PATH variable, you can run the echo command regardless of your location in the directory tree. You can find out what the PATH variable on your system contains using the method described above:

```
echo $PATH
```

When viewing the PATH, note that the /home/<user name>/bin directory appears in it. This means the shell will look in the bin directory inside your home directory when searching for commands. This location is one of the best places to store any shell scripts you create.

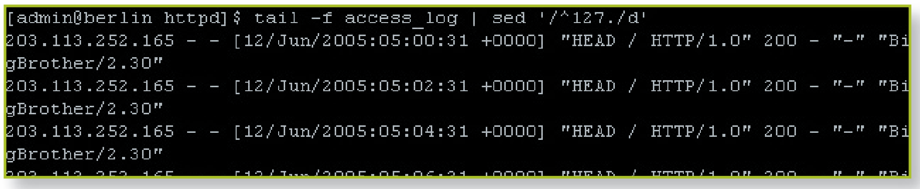
Editing the PATH variable is as simple as re-declaring it. Remember to consider the notation of the variable (use the output from echo as a guide); the order of execution; and to keep the existing locations in the PATH available. You can refer to the PATH variable while re-declaring it:

```
PATH="/home/<user name>/scripts:
$PATH"
```

USING QUOTES AND BRACES

5 One of the most common problems encountered when writing scripts results from not using (or incorrectly using) quotation marks. For instance:

```
NAME=APC Magazine
```



```
[admin@berlin httpd]$ tail -f access_log | sed '/^127./d'
203.113.252.165 - - [12/Jun/2005:05:00:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:02:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:04:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
```

3 ***Take it back:*** *sed can be used to filter out sections of screen output. For instance, to remove Web server log entries relating to local addresses.*

attempts to assign APC to the NAME variable, and then execute the Magazine command (which probably doesn't exist on your Linux system). To get a whole name assigned to the variable, you need to surround it with single or double quotation marks — that way, the shell doesn't treat what appears after the space as an argument.

Double quotation marks should be used when you need to group words together, but still want some level of command and variable substitution to occur. Single quotation marks should be used to assign data to variables literally. Consider the commands below:

```
LINE1="I read $NAME."
LINE2='I read $NAME.'
```

Because double quotation marks are on the first line, variable substitution is used during the assignment operation, and the value assigned will be `I read APC Magazine.` A literal assignment is used in `LINE2`, and its value is `"I read $NAME."` The same principle applies for command substitution.

It's especially important to remember the distinction between single and double quotation marks. In most cases, use double quotation marks to allow substitution. Single quotation marks should be used when you're dealing with strings containing dollar signs.

Braces can also be used to reduce ambiguity in scripts and are used by the shell to discern exactly where a variable name begins and finishes.

```
BRAND="Coca"
echo ${BRAND}Cola
```

Using the above example, the shell would attempt to echo a variable named `BRAND-Cola` if the braces weren't included.

However, the braces inform the shell that the variable that you're using is a name brand, and to append the `-Cola` string to the variable in the output.

A BASIC SCRIPT

A shell script is a plaintext file that contains commands that the shell executes in a procedural fashion. Since they are plaintext, they can be created by any text editor, including vi or pico. The first line of your shell scripts should contain the `#!` characters followed by the path to the interpreter that is used to execute the script. Since you're using pure shell commands, your shell scripts should start with:

```
#!/bin/sh
```

You may have seen a similar line at the start of Perl scripts, which informs the shell that the Perl interpreter needs to be used to process the script. The syntax used in those files is different, but a syntax that Perl understands.

After you've defined the interpreter, you can start entering the commands that make up your script:

```
ORIGINAL="$1"
ARCHIVE= "$2"
lpr "$ORIGINAL"
cp "$ORIGINAL" ~/backup/mastering_
linux_"$ARCHIVE"
```

The `"$1"` and `"$2"` variables are substituted for the first and second arguments to the script automatically. The first two lines assign these arguments to more user-friendly variable names. The third line prints the contents of the file parsed as the first argument. The last line copies the file specified as the first argument to a backup location, and names it in accordance with the second argument.

EXECUTION

Once you've finished writing the script, save it (remember, keep your scripts in ~/bin) and exit from your editor. You can now run the script by entering:

```
sh ~/bin/<script name> <arg1>
<arg2>
```

where `<script name>` is the name of the file that you saved your script as. This method of running your script isn't the most optimal. By passing the script location to the shell command, you're not taking advantage of the contents of the PATH variable.

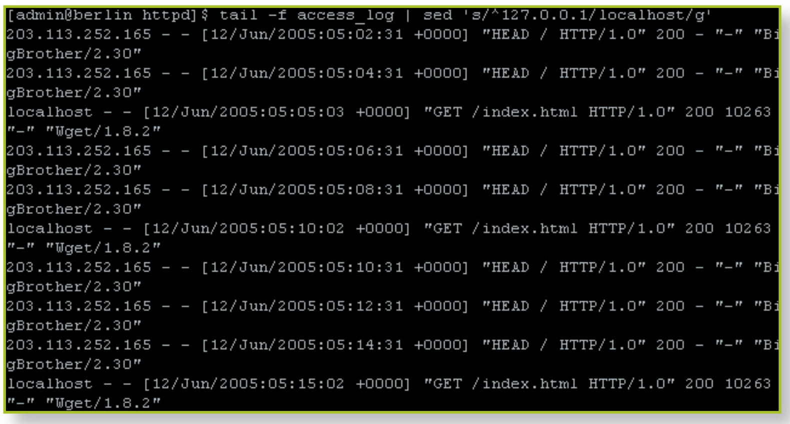
To make it easier to run your script, use `chmod` to change the permissions of the file so that it's executable:

```
chmod u+x ~/bin/<script name>
```

Once you've done this, you can run your script from any location in the file system by calling its name as a command. And since the script is located in your home bin directory, the shell will be able to find the script by using the PATH variable. apc

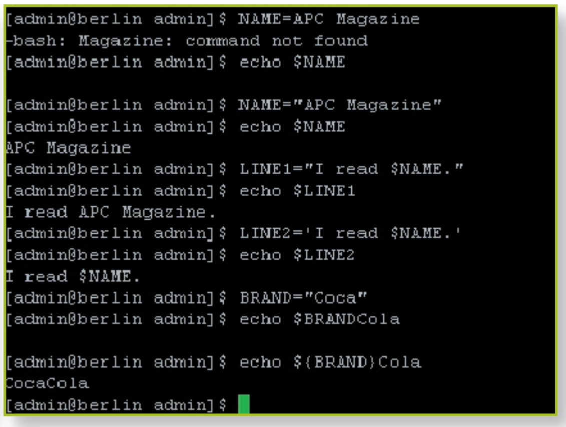
Next month . . .

Part 11 of the Mastering Linux series will cover conditional statements, control structures and other script coding methods. It will also briefly demonstrate some techniques for customising your X-Windows experience.



```
[admin@berlin httpd]$ tail -f access_log | sed 's/^127.0.0.1/localhost/g'
203.113.252.165 - - [12/Jun/2005:05:02:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:04:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
localhost - - [12/Jun/2005:05:05:03 +0000] "GET /index.html HTTP/1.0" 200 10263
"-" "Wget/1.8.2"
203.113.252.165 - - [12/Jun/2005:05:06:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:08:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
localhost - - [12/Jun/2005:05:10:02 +0000] "GET /index.html HTTP/1.0" 200 10263
"-" "Wget/1.8.2"
203.113.252.165 - - [12/Jun/2005:05:10:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:12:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
203.113.252.165 - - [12/Jun/2005:05:14:31 +0000] "HEAD / HTTP/1.0" 200 - "-" "Bi
gBrother/2.30"
localhost - - [12/Jun/2005:05:15:02 +0000] "GET /index.html HTTP/1.0" 200 10263
"-" "Wget/1.8.2"
```

4 ***Search and replace:*** *alternatively, sed can be used to replace information to make things easier to read.*



```
[admin@berlin admin]$ NAME=APC Magazine
-bash: Magazine: command not found
[admin@berlin admin]$ echo $NAME

[admin@berlin admin]$ NAME="APC Magazine"
[admin@berlin admin]$ echo $NAME
APC Magazine
[admin@berlin admin]$ LINE1="I read $NAME."
[admin@berlin admin]$ echo $LINE1
I read APC Magazine.
[admin@berlin admin]$ LINE2='I read $NAME.'
[admin@berlin admin]$ echo $LINE2
I read $NAME.
[admin@berlin admin]$ BRAND="Coca"
[admin@berlin admin]$ echo $BRANDCola

[admin@berlin admin]$ echo ${BRAND}Cola
CocaCola
[admin@berlin admin]$
```

5 ***Variable care:*** *the proper use of no, single or double quotation marks is imperative to creating useful scripts, as is the use of braces.*