Introduction to Shell Scripting

By Ben Okopnik

"You wouldn't believe how many managers believe that you can get a baby in one month by making nine women pregnant."
-- *Marc Wilson*

Random Wanderings

Well, this should be the last article in the "Introduction to Shell Scripting" series - I've had great feedback from a number of readers (and thank you all for your kind comments!), but we've covered most of the **basics** of shell scripting; that was the original purpose of the series. I may yet pop up at some point in the future ("Oh, rats, I forgot to explain XYZ!"), but those of you who've been following along should now consider yourselves Big-Time Experts, qualified to carry a briefcase and sound important... <grin> Well, at least you should have a pretty good idea of how to write a script and make it work - and that's a handy skill.

A Valued Assistant

Quite a while ago, I found myself in a quandary while writing a script (NO-O-O! How unusual! <grins>); I had an array that contained a list of command lines that I needed to execute based on certain conditions. I could read the array easily enough, or print out any of the variables - but what I needed was to execute them! What to do, what to do... as I remember, I gave up for lack of that one capability, and rewrote the whole (quite large) script (it was not a joyful experience). "eval" would have been the solution.

Here's how it works - create a variable called $cmd, like so:

```
cmd='cat .bashrc|sort'
```

It's just an example - you could use any valid command(s). Now, you can echo the thing -

```
Odin:~$ echo $cmd
cat .bashrc|sort
Odin:~$
```

- but how do you execute it? Just running "cmd" produces an error:

```
Odin:~$ $cmd
cat: .bashrc|sort: No such file or directory
Odin:~$
```

This is where "eval" comes into its own: "`eval $cmd`" would evaluate the content of the variable as if it had been entered at the command line. This is not something that comes up too often... but it is a capability of the shell that you need to be aware of.

Note that "bash" has no problem executing a single command that is stored as a variable, something like:

```
Odin:~$ N="cat .bashrc"
Odin:~$ $N
# ~/.bashrc: executed by bash(1) for non-login shells.

export PS1='\h:\w\$ '
umask 022
```

works fine. It's only when more complex commands, e.g., those that involve +aliases or operators ("|", ">", ">>", etc.) are used that you would encounter problems - and for those times, "eval" is the answer.

Trapped Like a Rat

One of the standard techniques in scripting (and in programming in general) is that of writing data to temporary files - there are many reasons to do this. But, and this is a big one, what happens when your users interrupt that script halfway through execution? (For those of you who have scripts like that and haven't thought of the issue, sorry to give you material for nightmares. At least I'll show you the solution as well.)

You guessed it: a mess. Lots of files in "/tmp", perhaps important data left hanging in the breeze (to be deleted at next reboot), files thought to be updated that are not... Yuck. How about a way for us to exit gracefully, despite a frantic keyboard-pounding user who just has to run "Quake" RIGHT NOW?

The "trap" command provides an answer of sorts (shooting said user is far more effective and enjoyable, but may get you talked about).

```
#!/bin/bash
```

```
function cleanup ()
{
    stty intr "" # Ignore 'Ctrl-C'; let him pound away...
    echo "Wake up, Neo."
    sleep 2; clear
    echo "The Matrix has you."

    echo "He's at it again."|mail admin -s "Update
stopped by $USER"

    # Restore the original data
    tar xvzf /mnt/backup/accts_recvbl -C /usr/local/acct
    # Delete 'tmp' stuff
    rm -rf /tmp/in_process/

    # OK, we've taken care of the cleanup. Now, it's
REVENGE time!!!
    rm /usr/games/[xs]quake

    # Give him a nice new easy-to-remember password...
    chpasswd
$USER:~X%y!Z@zF%HG72F8b@Idiot&(~64sfgrnntQwvff########^

    # We'll back up all his stuff... Oh, what's "--
remove-files" do?
    tar cvz --remove-files -f /mnt/timbuktu/bye-bye.tgz
/home/$USER

    # Heh-heh-heh...
    umount /mnt/timbuktu

    stty intr ^C # Back to normal
    exit         # Yep, I meant to do that... Kill/hang
the shell.
}

trap 'cleanup' 2

...
```

There's a little of the BOfH inside every admin. <grin> (For those of you not familiar with the "BOfH Saga", this is a **must** read for every Unix admin; appalling and hideously funny. Search the Web.)

DON'T run this script... yes, I know it's tempting. The point of "trap" is, we can define a behavior whenever the user hits `Ctrl-Break' (or for that matter, any time the script exits or is killed) that is much more useful to us than just crashing out of the program; it gives us a chance to clean up, generate warnings, etc.

"trap" can also catch other signals; the fact is that even "kill", despite its name, does not of itself `kill' a process - it sends it a signal. The process then decides what to do with that signal (a crude description, but generally correct). If you wish to see the entire list of signals, just type "`trap -l`" or "`kill -l`" or even "`killall -l`" (which does not list the signal numbers, just names). The ones most commonly used are 1)SIGHUP, 2)SIGINT, 3)SIGQUIT, 9)SIGKILL, and 15)SIGTERM.

[But SIGKILL is untrappable. -Ed.]

There are also the `special' signals. They are: 0)EXIT, which traps on any exit from the shell, and DEBUG (no number assigned), which can - here's a nifty thing! - be used to troubleshoot shell scripts (it traps every time a simple command is executed). DEBUG is actually more of an "info only" item: you can have this exact action without writing any "trap"s, simply by adding "-x" to your "hash-bang" (see "IN CASE OF TROUBLE..." below).

"trap" is a powerful little tool. In LG#37, Jim Dennis had a short script fragment that created a secure directory under "/tmp" for just this sort of thing - temp files that you don't want exposed to the world. Pretty cool gadget; I've used it a number of times already.

In Case of Trouble, Break Glass

Speaking of troubleshooting, "bash" provides several very useful tools that can help you find the errors in your script. These are switches - part of the "set" command syntax - that are used in the "hash-bang" line of the script itself. These switches are:

```
-n      Read the shell script lines, but do not execute
-v      Print the lines as they're read
-x      Prints $PS4 (the "level of indirection" prompt)
and the command just executed.
```

I've found that "-nv" and "-x" are the most useful invocations: one gives you the exact location of a "bad" line (you can see where the script would crash); the other, `noisy' though it is, is handy for seeing where things aren't happening quite the right way (when, even though the syntax is right, the action is not what you want). Good troubleshooting tools both. As time passes and you get used to the quirks of error reporting, you'll probably use them less and less, but they're invaluable to a new shell script writer.
To use them, simply modify the initial "hash-bang":

```
#!/bin/bash -nv
...
```

Use the Source, Luke

Here's a line familiar to every "C" programmer:

```
#include <"stdio.h">
```

- a very useful concept, that of sourcing external files. What that means is that a "C" programmer can write routines (functions) that he'll use over and over again, store them in a `library' (an external file), and bring them in as he needs them. Well - have I not said that shell scripting is a mature, capable programming language? - we can do the same thing! The file doesn't even have to be executable; the syntax that we use in bringing it in takes care of that. The example below is a snippet of the top of my function library, "Funky". Currently, it is a single file, a couple of kB long, and growing apace. I try to keep it down to the most useful functions, as I don't want to garbage up the environment space (is the concept even applicable in Linux? Must find out...)

There's a tricky little bit of "bash" maneuvering that's worth knowing: if you create a variable called `BASH_ENV` in your .bash_profile, like so:

```
export BASH_ENV="~/.bash_env"
```

then create a file called ".bash_env" in your home directory, that file will be re-read every time you start a `non-login non-interactive shell', i.e., a shell script. A good place to put initialization stuff that is shell-script specific; that's where I source "Funky" from - that way, any changes in it are immediately available to any shell script.

```
func_list ()     # lists all the functions in Funky
{
    # Any time I need a reminder of what functions I
have, what
    # they're called, and what they do, I just type
"func_list".
    # A cute example of recursion - a func that lists all
funcs,
    # including itself.

    cat /usr/local/bin/Funky|grep \(\)
}

getch ()     # gets one char from kbd, no "Enter"
necessary
{
    OLD_STTY=`stty -g`
    stty cbreak -echo
    GETCH=`dd if=/dev/tty bs=1 count=1 2>/dev/null`
    stty $OLD_STTY
}

...
```

Not too different from a script, is it? No "hash-bang" is necessary, since this file does not get executed by itself. So, how do we use it in a script? Here it is (we'll pretend that I don't source "Funky" in ".bash_env"):

```
#!/bin/bash

. Funky

declare -i Total=0

leave ()
{
    echo "So youse are done shoppin'?"
    [ $Total -ne 0 ] && echo "Dat'll be $Total bucks,
pal."
```

```
    echo "Have a nice day."
    exit
}

trap 'leave' 0
clear

while [ 1 ]
do
    echo
    echo "Whaddaya want? I got Cucumbers, Tomatoes,
Lettuce, Onions,"
    echo "and Radishes today."
    echo

    # Here's where we call a sourced function...
    getch

    # ...and reference a variable created by that
function.
    case $GETCH
    in
       C|c) Total=$Total+1; echo "Them are good cukes."
;;
       T|t) Total=$Total+2; echo "Ripe tomatoes, huh?" ;;
       L|l) Total=$Total+2; echo "I picked da lettuce
myself." ;;
       O|o) Total=$Total+1; echo "Fresh enough to make
youse cry!" ;;
       R|r) Total=$Total+2; echo "Real crispy radishes."
;;
       *) echo "Ain't got nuttin' like that today, mebbe
tomorra." ;;
    esac

    sleep 2
    clear

done
```

Note the period before "Funky": that's an alias for the "source" command. When sourced, "Funky" acquires an interesting property: just as if we had asked "bash" to execute a file, it goes out and searches the path listed in $PATH. Since I keep "Funky" in "/usr/local/bin" (part of my $PATH), I don't need to give an explicit path to it.

If you're going to be writing shell scripts, I strongly suggest that you start your own `library' of functions. (HINT: Steal the functions from the above example!) Rather than typing them over and over again, a single "source" argument will get you lots and lots of `canned' goodies.

Wrapping Up the Series

Well - overall, lots of topics covered, some "quirks" explained; all good stuff, useful shell scripting info. There's a lot more to it - remember, this series was only an introduction to shell scripting - but anyone who's stuck with me from the beginning and persevered in following my brand of pretzel-bending logic (poor fellows! irretrievably damaged, not even the best psychologist in the world can help you now... :) should now be able to design, write, and troubleshoot a fairly decent shell script. The rest of it - understanding and writing the more complex, more involved scripts - can only come with practice, otherwise known as "making lots of mistakes". In that spirit, I wish you all lots of "mistakes"!

Happy Linuxing!

Linux Quote of the Month:

```
``The words "community" and "communication" have the same
root.
Wherever you put a communications network, you put a
community as
well. And whenever you take away that network --
confiscate it, outlaw it, crash it, raise its price
beyond affordability -- then you hurt that community.

Communities  will fight to defend themselves.  People
will fight harder and more bitterly to defend their
communities, than they will fight to defend their own
individual selves.''
 -- Bruce Sterling, "Hacker Crackdown"
```
