

Basic Shell Scripting

Class presented to TriLUG
Saturday, August 13, 2004
9:00 AM (or thereabouts)

Jeremy Portzer
jeremyp@pobox.com

Shells

- sh = “Bourne Shell”
 - Named after author, old Unix standby shell
- csh = c-shell, C like syntax structures
 - tcsh = modern somewhat updated version
- ksh = Korn Shell
 - Introduced job control and other popular features now in bash
- bash = “Bourne Again Shell”
 - Variant of ksh, Popularized by Linux
 - By far most common shell used today
- Perl/Python = interpreted programming languages with many more features than shell, but can be used as more advanced platform for same tasks as shell scripts

Shell Concepts

- Basic script is equivalent of “Batch File” in DOS – list of commands to be executed in order, but many better features
- Ideal for cross-platform, quick jobs, not overly complex programs
- Shell scripts commonly used in boot scripts and OS-related automation tasks
- Not ideal for complex data structures or variable types, random file I/O, or many other things found in full programming languages. Perl is a good alternative.
- Shell scripts rely heavily on external utilities and tools present in the OS. Compatibility can be a big issue.

Hello, World

- Basic script:

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

```
echo "Hello world"
```

```
#commented line, this does nothing
```

- Note blank line after shebang.
- Give permission mode +x, such as 755, to be run from command line (man chmod for explanation of modes)
- Recommend personal scripts be stored in ~/bin, system-wide in /usr/local/bin
- Edit PATH setting in ~/.bashrc if needed
- Shell scripts cannot be setuid in Linux. Use sudo.

First things first: Standard files

- Standard Output
 - Output of a shell command, e.g. default location of echo or print statements. Can be redirected to a file with > operator, which overwrites its destination
 - echo “Hello, World!”
echo “Hello, World!” > file.txt
cat file.txt > file2.txt
>> operator appends instead of overwrites
- Standard Input
 - Where many commands get their data from.
 - If not provided, bash reads stdin from the terminal
 - One line processed at a time
 - use ^D on a line by itself to indicate end of file.
 - cat > file3.txt
 - Redirect with < operator, e.g. cat < file2.txt
- What does the “cat” command when run alone? Why?

Standard files, con't

- Standard Error
 - Similar to standard output, but used for error messages. This way, if standard output is re-directed, the error message is still seen (or at least it goes somewhere else)
 - Redirect with `2>`; note difference in these commands:
 - `cat /no/file/here > file3.txt`
 - `cat /no/file/here 2>/dev/null`
 - Print to standard error with `>&2`
 - `echo "An error has occurred! Blame jtate!" >&2`
- Other file descriptors
 - There is a system for using other file descriptors above 2, for your own purposes. I have no clue how it works, but I needed to put something in this space.

Pipelines

- A pipe redirects standard output of one command to the standard input of the next one
- Explain these pipelines
 - `tac /var/log/messages | less` (or “more”)
 - `who | grep jeremy`
 - `cat /var/log/messages | tail`
- Can include multiple commands, various input/output redirections, etc.
 - `uniq < file.txt | sort | uniq`
 - `sort -n -k 3 -t : < /etc/passwd | cut -f 1 -d : | xargs echo`
- “Filters” are commands that take stdin, do something with it, and output the data on stdout.
- Can span multiple lines for readability

Super-Simple Scripts

- Many scripts simply run a few commands that you would want to do frequently. For example:

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

```
cat /dev/null > /var/log/messages  
cat /dev/null > ~/root/.bash_history  
cat /dev/null > /var/log/wtmp
```

- Note, sh is a symlink to bash and invokes a supposedly [Bourne] sh-portable environment
- Run this from cron every N hours (real motd in /etc/motd.in):

```
cat /etc/motd.in > /etc/motd  
/usr/games/fortune -s >> /etc/motd
```

Variables

- Variables are essentially always strings, traditionally named in all caps, expanded with \$ or \${}, and are set without \$ at the start of a line:

```
SPAM=$HOME/mail/spam
mv $TENSAM ${SPAM}.tmp
sa-learn --showdots --spam --mbox $SPAM.tmp
rm ${SPAM}.tmp
```

- Export command makes available to sub-programs, ie, "export SPAM=..." to set and export, or "export SPAM" to just export. Needed in ~/.bashrc, e.g. with PATH
- "set" and "env" commands list environment variables currently set; some already set like UID, HOME.
- Variables expanded within double-quotes, but not within single quoted strings (compare echo "\$HOME" with echo '\$HOME')

Commands and Operators

- Many built-in commands available. “man bash” contains complete list. “help <command>” gives usage info for each command, or just “help” to list built-ins.
- Some commands exist as both built-in and separate binaries, i.e.: true, test, time (use “type” command)
- There aren't really “operators” in bash, but the test command, abbreviated as [, gives illusion

```
MHZ=$(grep "cpu MHz" /proc/cpuinfo | cut -d : -f2)
MHZINT=`awk -F. '{print $0}`
if [ $MHZINT -lt 500 ]
then
    echo "Your machine is slow. Blame jtate."
fi
```
- `$()` and ``` converts commands' standard out into args.
 - `grep :`id -g` : /etc/passwd`

Exit Status

- One "char" of information is passed between Unix processes in the exit status field, which is used extensively by the shell. Zero is "successful" or true, nonzero is false, backwards from standard

programming

```
jeremy@jeremy ~ $ /bin/true
```

```
jeremy@jeremy ~ $ echo $?
```

```
0
```

```
jeremy@jeremy ~ $ /bin/false
```

```
jeremy@jeremy ~ $ echo $?
```

```
1
```

- Control structures use command exit status to determine flow.
- "exit" command, as in C, exits your script with given status

Control Structures

- **if/then/elif/else/fi**

```
if [ $UID != 0 ]
then
    echo "you are not root"
    logger -p security.alert "$USER is running $0 at `date`"
elif [ $HOME != /root ]
then
    echo "home directory is wrong"
else
    # do something
fi
```

- **while/do/done**

```
while true
do
    #something
done
```

Control Structures, con't

- `case/;/esac` -- demonstrated in "apt-yum" script :

```
case "$1" in
  install)
    shift
    yum install "$@"
    ;;
  update)
    shift
    yum list >/dev/null
    yum clean oldheaders
    ;;
  upgrade)
    shift
    yum update "$@"
    ;;
  dist-upgrade)
    shift
    yum upgrade
    ;;
  *)
    echo "I don't know how to yumify that!"
    exit 1
    ;;
esac
```

- Note use of positional paremeters \$1, \$2, etc, shift, \$@ for entire line
- See `getopt(1)` for argument parsing "library"

Control Structures, con't

- `for/in/do/done`

```
for i in foo*
do
    mv $i ${i}.bak
done
```
- For loop list whitespace separated (foreach in perl)
- A C-like "for" is available but rarely used
- Conditional control operators `||` and `&&`
 - `touch /etc/motd || echo "Error" >&2`
 - `touch /etc/motd && echo "wrote to motd" >&2`
- Loops and branches can be nested as expected. Use whitespace and CRs to make your scripts look sane
- More complex arithmetic and conditionals are allowed, but less used; see `man bash`, "let" command, "test" command help, etc.

File I/O Goodies

- Review - Dealing with standard error
 - `echo "error" 2>&1` # prints stdout on stderr
 - `touch /etc/motd 2>/dev/null` #sends stderr to file
 - `ls /nowhere >/dev/null 2>&1` #sends stderr same place as stdout, which in this case is /dev/null

- Safe temporary files:

```
NAME=`basename $0`  
TMPFILE=`mktmp /tmp/$NAME.XXXXXX`  
if [ ! -w $TMPFILE ]  
then  
    echo "Temporary File Creation Failed" >&2  
    exit 1  
fi  
# do stuff, using $TMPFILE variable  
rm -f $TMPFILE
```

- "Here" documents, multiple lines to stdin:

```
cat <<EOF  
This is some  
multi-line output.  
EOF
```

Other I/O

- "read" builtin to prompt user for variable (or to read one line from an input file)
 - Use with caution due to security implications
- "logger" to send syslog messages
- Send things to /bin/mail to output things to email
 - Simple log analysis script example:

```
TODAY=`date "+%b %d"`  
COUNT = $(  
    grep $TODAY /var/log/messages |  
    grep "authentication failure" |  
    wc -l  
)  
if [ $COUNT -gt 4 ]  
then  
    echo -e "There have been $COUNT authentication failures  
today. \nPlease investigate." |  
    /bin/mail -s "Security Alert from `hostname`" jeremy@
```
 - Note, requires properly configured mail subsystem

Useful Commands

- Grep -- learn regular expressions! Careful with options
- Awk
 - Scripting/programming language in its own right. I mostly use it to parse fields delimited by whitespace like so:
 - `ps auxw|grep bash | awk '{print $2}'`
 - Say no to advanced awk scripts -- choose Perl (or Python)
- Cut
 - Useful for parsing files with a given delimiter
 - `grep -v /home /etc/passwd | cut -d : -f 1`
- Sed
 - Stream editor, make changes based on regular expressions
 - Strip comments from input files:
 - `/bin/sed -e 's/^#.*$//g'`

Useful Commands, con't

- `Rpm -ql coreutils | grep bin`
 - fileutils, textutils, sh-utils on older systems
- `Sort, uniq, tr, join, pr, expr, seq, basename`

```
INFILE=trilug.mbox
grep '^From ' < $INFILE |
    awk '{print $2}' |
    sort |
    uniq -c |
    sort -rn > $EMAILCOUNTFILE
```

```
grep '^From .*@.*' < $INFILE |
    awk '{print $2}' |
    uniq |
    sort |
    uniq > $EMAILOUTFILE
```

```
for i in `seq 1 100`
do
    touch $i
done
```

- Many others – audience suggestions?

find my xargs!

- Extremely versatile command for finding files and doing stuff to/with them
- `find /path/to/directory -print`
- `find -name '*.txt' -exec echo rm {} \;`
 - Note single quotes to avoid shell wildcard expansion
- `find -name '*.txt' | xargs echo rm`
- `find -name '*.mp3' -print0 | xargs -0 mpg123`
- `find -not -newer /tmp/basefile 2>/dev/null | xargs ls -l`
- For traversing the whole filesystem, consider “locate” instead but understand database requirements (updatedb, etc.)

Other shell niftiness

- Learn about job control. Very handy to have around the shell and with care, can be used in scripts too.
- Using perl one-liners for regular expression stuff that is easier in perl. Good for quick things, but maybe consider writing the script in perl to start with
 - `perl -ple 's/^s/:/g' < file2.txt`
 - `man perlre`; `man perlrun`
- Learn special variables
 - `$?` - exit status of last command
 - `$$` - process ID of current shell
 - `$*` or `$@` - all positional paremeters
 - `$1`, `$2`, `$3`, etc - positional paremeters
 - `$#` - number of positional parameters
- **RANDOM** special variable (see `man bash`)

cron-ological

- I can't ever remember the crontab format, so I always put this in the top of mine (copied from the man page, which is “man 5 crontab”)

```
- #           minute           0-59
  #           hour             0-23
  #           day of month     1-31
  #           month            1-12 (or names, see below)
  #           day of week      0-7 (0 or 7 is Sun, or use names)
```

- Example lines

```
- 0 */6 * * * ~/public_html/pisg/run-pisg >/dev/null 2>&1
  30 */2 * * * ~/public_html/pisg/topic.sh
  0 0 * * * cat /dev/null > ~/procmail.log
  2 2 * * * ~/bin/delspam
```

- crontab -e, crontab -l
- Keep in mind /etc/cron* directorys, plus root's crontab
- If your cron generates output, it will be mailed to you. So check those mailboxes!
- Put commands to run at boot-time in /etc/rc.d/rc.local

Security Ideas

- Never do much with user input unless it's carefully parsed and you have thought through it
- Use full paths to binaries, perhaps setting them in variables at top of script
- If not, at least overwrite PATH with known safe values
- Always use full path for input and output files, or chdir to working area before starting
- Safe temporary files! (see previous example)
- Lots of error detection code; check exit status of commands frequently and respond appropriately
- What else?

Resources

- Advanced Bash Scripting HOWTO
<http://www.tldp.org/LDP/abs/html/>
- O'Reilly & Associates:
 - Learning the Bash Shell, 2nd Edition
 - Essential System Administration, 3rd edition
- Documentation:
 - Bash man page
 - "help" output for shell built-ins,
 - command --help for quick usage
 - man and info pages for other commands (I recommed "pinfo" for info viewing)
 - /usr/share/doc/* for certain commands (gawk, bash, etc)
- Read scripts in your distro (example, /etc/rc.d stuff)
- Google!

