Linux Shell Scripting

Linux System Administration
COMP2018 Summer 2017

What is Scripting?

- Commands can be given to a computer by entering them into a command interpreter program, commonly called a shell
- Scripting is the act of saving one or more commands to a file for automated execution
- Many interactive command interpreters can be run using saved scripts
- Commands in scripts are executed exactly the same as if they were typed by hand into the interpreter

Why Script?

- Insulate a script user from the details of a task they are performing; create tools for non-administrative users
- Communicate task activities and requirements to the user in a way the user can understand
- Perform checks to ensure the task is properly run and deal with problems that may arise in carrying out the task as requested by the user

Script Execution

- Scripts can be run as commands or by specifying them as an argument to the bash command
- Either way, they run in a new child process, not the current shell process, so they inherit your environment but do not have access to your local variables
- Running a script as a command requires execute permission for the script file and that the shell can find the script file
- To execute a script file in the current shell process instead of a child process, source it using the source command or its alias, the . (dot) command - this is equivalent to copying and pasting the script into the current shell
- Scripts can be copied and pasted onto a bash command line if you want them to run in the current process, be careful if you try to do this between Windows and any other operating system

```
bash scriptfile.sh

chmod u+x scriptfile.sh
./scriptfile.sh

mv scriptfile.sh ~/bin
scriptfile.sh

source ~/bin/scriptfile.sh
. ~/bin/scriptfile.sh
```

Script Content

- Scripts can contain commands, blank space, comments, and inline data
- Scripts are a minimum of one line, with no practical limits on length
- Commands in scripts are the exact same commands you could use on the command line interactively
- Scripts end when they encounter a fatal bash error, or the exit command, or run out of commands

```
#!/bin/bash
# My first script

echo 'Hello World!'
echo "I am process # $$"
```

```
#!/bin/bash
# My second script

cat <<EOF
Hello World!
I am process # $$
EOF
```

Comments

- A comment is any text beginning with #
- They provide the reader of the script with useful information
- They can also be used as part of the process of debugging scripts

```
# This is a comment

# Comments are ignored by the interpreter

echo “Hello World” # this is a comment on the same line as a command

# funky-command-that-might-be-causing-trouble
```

Command Pipeline

- A command pipeline is a sequence of commands separated by the | character
- The | character causes the output (/dev/stdout) of the command on the left of the | to be connected to the input (/dev/stdin) of the command on the right
- The exit status of a pipeline is the exit status of the last command in the pipeline

```
ls | wc –l
ps –ef | awk ‘{print $1}’ |sort | uniq -c
```

Command Lists

- A command list is a sequence of commands separated by the operators ; & && and ||
- ; is used to simply execute commands in order with no dependence on each other
- & is used to put a command into the background
- && and || cause the command on the right to be run only on the success and failure respectively of the command on the left
- We can use the exit status of test or [, and [[commands to perform simple command lists based on the results of evaluating expressions

Variables

- Every process has memory-based storage to hold named data
- A named data item is referred to as a variable (sometimes called a parameter), all of them together create a simple table with names and values
- Variables can hold text or binary data as their value
- Variables are typically created by assigning data to them, using an assignment operator such as =

```
myvar=3
variable2="string data"
vowels="aeiou"
```

Environment Variables

- By default variables are created in local process memory, not in the process environment
- Environment variables are inherited by child processes
- Environment variables are normally named using capital letters and numbers only
- Variables can be exported to the environment or removed from the environment

```
VARNAME=“Some data for this variable”
export VARNAME
export -n VARNAME
export VARNAME=“Data”
```

Accessing Variables

- Variable content is accessed using the $ symbol prefixed to the variable name
- Non-trivial variable names must be surrounded by { }

```
echo $myvar
echo ${myvar}
echo ${myarray[32]}
echo ${frank-n["beans"]}
```

Using Variables

- Variables have many uses
- Command line substitution is a very common use, using a variable to provide data used on a command line
- Variables are deleted using the unset command

```
echo $SHELL
ls $HOME
for file in $FILES; do
if [ “$USER” != “$LOGNAME” ]
mypid=$$

unset VAR
```

Shell Data - Numbers

- Data is often found inline in scripts
- Inline data can be simple numbers or strings of characters
- Numbers are simply entered as digits and can be signed, but must be integers
- bash can do basic arithmetic +, -, *, /, % on integers by putting arithmetic statements inside $(( ))
- Leaving off the $ allows you to test if the result is zero

```
echo 32
echo $(( 3 + 4 ))
echo "5 divided by 2 leaves a remainder of $(( 5 % 2 ))"
(( $loot % $raiders )) || echo "Uh-oh, doesn't divide evenly"
```

Shell Data - Strings

- Strings are normally entered as single words, or surrounded by quotes for strings containing special characters such as spaces

- Single quotes turn off all special characters

- Double quotes turn off most special characters, $ is still special inside ""

- Special characters can also be preceded with \ to turn off their special meaning

```
cd
touch My File
ls
touch "My File"
ls
touch My\ Other\ File
ls
touch 'Terrible"Name"'
ls
touch -- '-worse\ name'
ls
```

Special Variables

- $ - current process id
- # - number of parameters on the command line of a script
- 0-n - command line parameters
- RANDOM - a random 15-bit integer
- REPLY - default variable name for the read command
- ? - exit status of most recent command

Exit Status

- Every process that runs, produces an exit status when it ends, either intentionally using the exit [status] command, or automatically due to script bailout, or end of file, or signal reception
- The shell can access that status using the special variable ?
- Exit status 0 normally means successful completion
- Any time a command might fail and this would cause problems or be a problem for the script user, your script should be doing something to deal with the failure

```
echo $?
[ $? = 0 ] || handle error
```

Testing Data

- We can do unary tests to determine if strings have data -v string or -n string, could also use -z string for zero length
- We can do binary tests comparing string data to other values using =,<,>,!= including static values
- We can do binary tests comparing integer data using -eq, -ne, -lt, -le, -gt, -ge
- This can be used to validate user input as well as test data retrieved from elsewhere in the system

Testing Files

- We can do unary tests for file existence (-e), type (-f,-d,-h,-p,-b,-c,-S), permissions (-r,-w,-x,-k,-u), ownership (-O,-G), size (-s), modification (-N), and whether a file is an open terminal device (-t fd)

- We can do binary tests on files based on their dates (-nt, -ot), and determine if two filenames are hard linked (-ef)

If Command

- Action can be taken, or not taken, based on the exit status of a command list
- The test command can evaluate unary or binary expressions, so it can be a very useful command for the list
- For a list of available expression operators, refer to the man page

```
if list; then
    list
else
    list
fi

if [ expr ]; then
    list
fi
```

While Command

- A list can be executed repeatedly based on the exit status of another list
- The break or continue or exit commands can be used in the *do list* to get out of a loop early

```
while list; do
    list
done
```

For Command

- The for command allows repeated execution of a list either substituting values from a word list in a variable or by evaluating expressions

```
for varname in wordlist; do
    list
done


for (( initial expression; test expression; loop expression )); do
    list
done
```