Server Operating Systems

Lecture 16
Shell Scripting

Introduction to Shell Scripts

- Shell Script Basics
- Advanced Shell Scripts

Overview of Shell Scripts

- At their simplest, shell scripts are are just an ASCII file with Unix commands in them.
- *Comments* can be in a shell script.
  - A comment is a '#' anywhere on a line and everything following to the end of the line.
- Shell scripts are fed, one line at a time, to a particular shell and *interpreted* by that shell as it sees the commands.

Basic Shell Script Example

```
$ vi simplescript1


#!/bin/bash
# simplescript1
# displays current user status
echo "Today's date is:"
date
echo " "
echo "Users currently on the system are: "
who
```

Shell Script Development Cycle

1. Decide what the script will do.
2. Make a list of commands.
3. Create a new file for the script.
4. Identify the shell the script will use.
5. Add commands and comments.
6. Save the script file.

Creation of the Shell Script.

7. Make the script file executable.
8. Type the name of the script to execute it.

Executing the Shell Script

9. Debug and modify the script if errors occur.

Debugging the Shell Script

Creating the Shell Script

- '.sh' is the conventional filename extension.

- Use **`#!/path/to/shell`** on the first line of the script to execute the script with the desired shell.

  Otherwise, the parent shell is used.
  - **`#!/bin/bash`**
  - **`#!/bin/ksh`**

- Avoid the names of Unix commands for your script filenames.

Executing a Shell Script

- A shell script is always run in a sub-shell (new process spawned).
- You can run a shell script in two ways:
  - ***`shell script_name`*** at the command line.
  - Make the script executable, then just use the name of the script like a Unix command.

    (Provided script is in a folder listed in $PATH environment variable).

Example of two ways of running a script

```
$ ksh simplescript1

The number of users logged on is: 3
Today's date is: Sat May 25 2002
```

```
$ chmod 755 simplescript1
$
$ simplescript1

The number of users logged on is: 3
Today's date is: Sat May 25 2002
```

Debugging a Script

- There are two *shell* options that are helpful in debugging: -x (echo) and –v (verbose).

- Echo
  - Displays each line of the script *after* the shell interprets it.
  - A ‘+’ sign is placed in front of each line to differentiate it from actual script output.

- Verbose
  - Displays each line of the script as it appears in the script. That is, *before* interpretation.
  - No ‘+’ is printed out.

'Quotes', "Quotes" or `Quotes`?

Quote Character	Meaning	Example

Single quote (')	Display contents - including metacharacters - literally.
	Does **not** allow variable expansion.
		`$ echo ' *** $LOGNAME *** '`
		
		`*** $LOGNAME ***`

Double quote (")	Display contents - including metacharacters - literally.
	Does allow variable expansion.
		`$ echo " *** $LOGNAME *** "`
		
		`*** user10 ***`

Back quotes (`)	Execute command and display output.	`$ echo `uname -n``
		
		`user5`

vi editor commands

vi filename	Start editor
i	to go into insert mode
esc	come out of insert mode
:wq	save and quit
:w filename	save to filename
:q	quit

Or you could do what most people do and use a GUI editor.

Advanced Shell Scripting

- A shell script can do more than hold a list of commands to be executed sequentially.

- The built-in shell programming languages include logic statements, flow control statements and variables.

Variables

You define a variable as follows: **`X="hello"`**

and refer to it as follows: **`$X`**

bash gets unhappy if you leave a space on either side of the = sign.

```
X = hello    ##error
```

While I have quotes in my example, they are not always necessary.
Where you need quotes is when your variable names include spaces.

```
X=hello world                    # error

X="hello world"                  # OK
```

Three Types of Variable.

Global or Environment Variables

- Available in all shells. Use env to display them.
- $PATH, $PS1, $PWD

User defined Variables

- (These can be typed in as direct commands)

```
name=Kevin
echo $name
```

Command line (positional) Parameters

Positional Parameters

- Information can be passed in to a shell script on the command line in the form of parameters or *arguments*.

- These arguments are stored in special variables.

Parameter	**Meaning**
$0	Name of script
$1 - $9	Command line argument number
$*	All arguments entered on command line
$#	Number of arguments entered.

Example of Arguments

```
$ addto /home/fruit  /usr/bowl
```

```
$0      addto
$1      /home/fruit
$2      /usr/bowl
$*      /home/fruit /usr/bowl
$#      2
```

Interactive Input

- The `echo` and `read` commands provide a way to prompt for and obtain user input from a shell script.
  - **`echo` *`string`*** is used to print text out.
  - **`read [`*`variable(s)`*`]`** is used to obtain input typed by the user.

- Escape characters:
  \t (tab), \n (newline), \c (carriage return).
  - In Bash, **`echo -e`** must be used to recognise escape chars.

The if-then Command Format

`if` *condition is true*

`then`

Execute block of code…

`fi`

The if-then-else Command Format

`if` *condition is true*

`then`

Execute block of code…

`else`

Execute block of code…

`fi`

The if-then-elif Command Format

if *condition is true*

then

Execute block of code…

elif *condition is true*

then

Execute block of code …

else

Execute block of code…

fi

Exit Status

- A command run from the command line or a shell script returns a value to the parent process indicating its success or failure called an *Exit Status*.
- A command defines what a given exit status means. The convention is that a return value of 0 (zero) is a success, and a non-zero value is failure.
- This is how if statements determine which blocks of code to execute.
- The variable `$?` is defined automatically by the shell to hold the exit status of the last command executed.
- Use `exit 0` as the last line of a shell script to indicate successful completion.

Example of Exit Status

```
$ mkdir collegework
$ echo $?
0

$ mkdir dir1
$ mkdir: Failed to make directory "dir1"; file exists
$ echo $?
1
```

The `test` Command

- The `test` command evaluates an expression, and if the result is true, it returns an exit status of 0. Otherwise, with a false result, a non-zero exit status is returned.

- String or variable comparison can be performed with the `test` command as well as testing file status.

The test Command Syntax

```
if test expression
then
        Execute command(s)
fi
```

---------- or ----------

```
if [ expression ]
then
        Execute command(s)
fi
```

`test` Command Operators

Operator	Returns TRUE (exit status of 0) if
-e file	File exists
-s file	File is not empty
-d file	File exists and is a directory
-f file	File exists and is a plain file
-r file	File exists and is readable
-w file	File exists and is writeable
-x file	File exists and is executable

- The test command can be used with files to test: file type, file permissions, and whether a file contains data.
- '!' can be used to negate a test.

Example of test command operators

```
if [ -d $1 ]
then
  echo $1 is a directory
elif [ -c $1 ]
  then
    echo $1 is a character file
  else
    echo I do not know what it is
fi
```

Save as filetester.sh and use by typing in:

`filetester` *`filename`*

The `case` Command

```
case value in
value1 )
        Execute command(s)
;;
value2 )
        Execute command(s)
;;
*)
        Execute command(s)
;;
esac
```

Example of Case

```
#!/bin/ksh
answer=$1
case "answer" in
y)
echo "You selected Yes"
;;
n)
echo "You selected No"
;;
*)
echo "Invalid Selection"
;;
esac
```

save as tester.sh and use by typing in

tester y or tester n

`for` Loop

`for` *variable* **`in`** *list*

`do`

command(s)

`done`

- The loop executes once for each value in *list*.
- *variable* is set to a new value in *list* each time through the loop.
- When no more values exist in *list*, the loop terminates.

```
#!/bin/ksh
dir=/home/user10/backup
for file in chapter1 chapter2 chapter3
do
cp $file $dir/$file.back
echo $file has been backed up in directory $dir
done
```

while Loop

while *condition*

do

command(s)

done

- This loop executes a block of commands until the expression tested becomes false.

`until` Loop

`until` *condition*

`do`

command(s)

`done`

- A block of code is executed as long as the *condition* returns unsuccessfully.