UNIVERSITY OF THE WEST of SCOTLAND

COMP09024 Unix System Administration

Laboratory 5: The Shell and Shell Scripting

The Shell

Learning Outcomes

- **Understanding the concept of shell variables**
- **Commands: printenv, export**
- **Understanding the principle of command execution within a shell**

In this lab we will examine the basic functionality of the Unix shell and also write some simple shell scripts.

The Shell

The shell is a fundamental component of the Linux OS. As the interpreter for commands it acts precisely according to the inputs provided. The actual outcome of a shell command can look quite strange to novice users and expertise is needed to interpret the full meaning. Keep in mind that the shell does things in a simple and efficient way; this simplicity makes it a fast interface.

Working within the shell, you should always keep in mind:

- The shell is a crude interpreter, it does not think nor ask for confirmation and above all it does not attempt to fix any *odd* input. How could it possibly anticipate ALL potential mistakes?
- The shell clears the output channel of a command first before executing the command.
- The shell works on and expands Shell *metacharacters* and *wildcards* (e.g. '*****') before executing commands.

Environment Variables

The characteristics of a shell environment are specified by setting special (shell) variables. In order to find out about the current settings, use the **printenv** command. This command will give a list of all currently defined shell variables. (These are normally all defined using capital letters only!).

```
[root@UWS /]# printenv
```

Questions

- **Q5.1)** How many variables are currently defined in your shell? (Tip: use **wc -l**)
- **Q5.2)** What are the values for the shell variables: **DISPLAY, LOGNAME, PATH**

The **HOME** variable defines the home-directory of the current user. Of course you can redefine any variable to any value that you like. To do so, you would first overwrite the existing entry and then export the new setting to the shell environment.

The current setting of the **HOME** variable should be **/root** since you are the root user. You can display the contents (value) of any variable by echoing it in the following way:

```
[root@UWS /]# echo $HOME
```

The **$** sign tells the shell to display the value of the variable. Remember that there is a distinct difference between the variable name (e.g **HOME**) and its value (e.g. **$HOME**).

$HOME **IS NOT** HOME

If you type **cd** on its own, the shell responds by going to the directory specified by the **HOME** variable. Please type the following commands (to see this for yourself):

```
[root@UWS /]# cd
[root@UWS /]# pwd
```

Now let us change the **HOME** variable to see the difference this makes:

```
[root@UWS /]# HOME=/etc
```

There should be no space between any of the characters in the above line! Continue with:

```
[root@UWS /]# export HOME
```

Now check the new setting by typing:

```
[root@UWS /]# cd
[root@UWS /]# pwd
```

Question

- **Q5.3)** To which directory does the **`cd`** command set you now?

Revert the current setting back to normal by:

```
[root@UWS /]# HOME=/root
[root@UWS /]# export HOME
```

Another very important variable is the **PATH** variable, since it defines where the shell will search for commands/programs. When you invoke a command, the shell scans the directories specified in the **PATH**, variable working from left to right, and attempts to execute the first matching command. The directories to be read are separated by a colon sign '**:**' .To check in which directory the shell actually finds a program or command for execution, use the **`which`** command e.g.:

```
[root@UWS /]# which ls
```

Moreover, the command **`whereis`** checks for all instances where a given command is found via the **PATH** variable.

Questions

- **Q5.4)** In which directories are the **`fsck`**, **`which`** and **`whereis`** commands found?
- **Q5.5)** How many directories are searched for the occurrence of a program or command?
- **Q5.6)** What is the output if you try to run **`whereis cp`**?
- **Q5.7)** What is the **`whereis`** argument used to search only manuals?

Different users may have different **PATH** variables. To verify this, login as **`student`** with the command:

```
[root@UWS /]# su student
```

provide the appropriate password and then type:

```
[student@UWS /]# echo $HOME
```

Question

- **Q5.8)** Which binary directories are omitted for **`student`** as opposed to **`root`**? Explain why this might be the case.

As a matter of fact the current directory is normally not included in the **PATH**, as well as the home directory of any user. To demonstrate this please copy the **`ip`** command to your **`/root`** home directory giving it the name **`ipmine`**:

```
[root@UWS /]# cp /sbin/ip /root/ipmine
```

Next make sure you are in the **/root** directory and try to run **ipmine**

```
[root@UWS ~]# ipmine
```

Question

- **Q5.9)** What is the output if you try to run **ipmine**?

Now let us include the local directory in the **PATH** variable by:

```
[root@UWS /]# PATH=$PATH:.
[root@UWS /]# export PATH
```

Question

- **Q5.10)** Explain the effect of the two commands **PATH=$PATH:.** and **export PATH** in your own words. (Tip: Check the **PATH** variable again)

Try to run **ipmine** now.

Question

- **Q5.11)** Does **ipmine** run now? If 'yes' explain why.

Wildcards and Globbing

The shell provides some *wildcard* characters for the purpose of matching filenames - this is known as *globbing*. The most important wildcards are the asterisk ***** and the question mark **?**. The asterisk matches *any sequence of characters* the question mark matches *any single character*. Let us create the following empty files:

```
[root@UWS /]# touch one.dat two.dat three.dat four.dat five.dat
```

and then please try:

```
[root@UWS /]# echo *
[root@UWS /]# echo *.dat
```

Questions

- **Q5.12)** What command provides the same output as **echo *** ?
- **Q5.13)** Explain the two command lines above in respect to the fact that the shell expands wildcards

Now (still being in **/root**) go on with typing:

```
[root@UWS /]# echo ???.dat
[root@UWS /]# echo ????.dat
```

```
[root@UWS /]# echo ?????.dat
[root@UWS /]# echo ??????.dat
```

Questions

- **Q5.14)** Explain the output produced by the FOUR command lines in detail.
- **Q5.15)** What does the shell force the **echo** command 'to echo' if the expansion of wildcards does not produce any results, as in the case of the last command line?

Other important metacharacters include: **[A-Z]** representing any uppercase character. **[a-z]** representing any lowercase character and **[0-9]** representing any digit. If you put just one character or one digit in the square bracket the shell will look for the occurrence of this character: e.g. **[S]** will look for all uppercase **S** in the filename and **[S,s]** will look for the occurrence of an uppercase or lowercase **s**.
Please type:

```
[root@UWS /]# find / -name [K][0-9][0-9]*
```

(make sure that you are logged in as **root** for this command!)

Question

- **Q5.16)** Do you know what type of files are found by the above command? Explain the function of the 3 metacharacters and the wildcard given within: **[K][0-9][0-9]***

The Shell is Literal
The shell follows instructions literally and does not bother that much about things that may go wrong. To illustrate this consider the following:

```
[root@UWS /]# echo 'abcdefg' > test.dat
[root@UWS /]# more test.dat
```

We just created a file named **test.dat** with some data. Now let us deliberately type a mistake by trying to invoke a non-existent command, namely **echa** (which could be seen as a typo for **echo**), that appears to be intended to overwrite **test.dat**:

```
[root@UWS /]# echo '12345678' > test.dat
[root@UWS /]# more test.dat
```

Questions

- **Q5.17)** What actually happened to the existing file **test.dat**? What was the 1st task the shell performed once the above command line was issued?
- **Q5.18)** Reflect on some potential dangers of this default behaviour?

The Shell and Regular Expressions

There are 3 different kinds of quotation marks that deal with the interplay of the shell and regular expressions. Each of them has a different meaning:

The single quote:	**‘ ’**	shields all entries fully from the shell!
The double quote:	**“ ”**	allows the replacement of variables by their contents
The back quote:	**` `**	will perform any command in the quotation before handing the result over to the shell

The understanding and use of these different quotation marks defines a real Linux expert! It allows you to write and read almost all kinds of complex shell script.

To see the difference type:

```
[root@UWS /]# echo ‘$HOME’
[root@UWS /]# echo “$HOME”
```

Question

- **Q5.19)** Explain the differences in the output of the above command lines

Next see the importance of the back quotes by:

```
[root@UWS /]# echo `echo $HOME`
[root@UWS /]# echo `hostname` ; echo `hostname –i`
```

Questions

- **Q5.20)** Explain the output of the above command lines, if necessary use the man pages to understand the different output.
- **Q5.21)** What is the function of the semicolon **;** in the above command line?

Shell Scripting

Shell scripts are simply programs that use the shell as a programming language. They are fast in execution, because they are in the ‘local’ language of the shell and don’t require compilation. Of course they also inherit the crudeness of the shell.

The best shell for writing scripts is the bash shell. Bash scripts must always begin with the line: **`#!/bin/bash`**. Let us create a simple script called **`envdisplay`**:

```
[root@UWS /]# nano envdisplay
```

Enter the following code in your editor of choice (e.g. **`nano`**) and save as **`/root/envdisplay`**:

```
#!/bin/bash

echo $HOME
echo $PWD
echo $HOSTNAME
```

Question

- **Q5.22)** Would the script **`envdisplay`** run without a change of the permission settings? Note the current permission settings in octal representation.

Now change the permissions by typing:

```
[root@UWS /]# chmod +x envdisplay
```

Question

- **Q5.23)** Note the new permission settings of **`envdisplay`** , after the **`chmod`** command was issued, in octal representation. Why is the script executable now?

Since your current directory is not by default included in the shell's **`PATH`** variable you must preface the script with a dot. Continue to run the script from your current directory:

```
[root@UWS /]# ./envdisplay || echo 'I did something wrong'
```

Questions

- **Q5.24)** Explain the output of **`envdisplay.`**
- **Q5.25)** Explain why you should only see the line `'I did something wrong'`, if your script does not perform as intended. Please refer to the list of metacharacters as shown in the appendix to understand the meaning of the conditional pipe: '**||**'. (If you see the quoted line on your display, please go back and do some troubleshooting!

Input variables are treated as **`$1, $2`**, … etc within a shell. Extend the above shell-script by adding the following line at the end of **`envdisplay`**:

```
#!/bin/bash

echo $HOME
echo $PWD
echo $HOSTNAME
echo $1
```

Set a new shell variable as follows:

```
[root@UWS /]# MYVAR=test ; export MYVAR
```

Now run **`envdisplay`** again like:

```
[root@UWS /]# ./envdisplay MYVAR  || echo 'I did something wrong'
```

and then type:

```
[root@UWS /]# ./envdisplay $MYVAR || echo 'I did something wrong'
```

Question

- **Q5.26)** Explain the output of **envdisplay**. In particular, why does the first invocation of **envdisplay** not produce the string **test**, yet the second one does?

Now create a shell script called **newhome** in the **/root** directory that allows you to override the current setting of the **HOME** variable with a named directory provided during execution.

```
#!/bin/bash

export HOME=$1
echo 'Home directory during script execution is:'  "$HOME"
```

The script should take the new **HOME** directory as external variable **$1.**
Before you run the script don't forget: **chmod + x newhome**

Now perform

```
[root@UWS /]# ./newhome  /etc
```

if you want to set **/etc** to your home directory.

After you run **newhome** check the contents of the **$HOME** variable
Question

- **Q5.27)** What is the value of **$HOME** after the script has finished running? What has happened?

The result is not as odd as you may think. As previously mentioned the shell only allows the temporary resetting of variables while a shell or program is executing. You are by default not allowed to change any important settings. This strategy makes Linux a more stable system and minimises crashes. Again if you really want, you may overrule the default. Try:

```
[root@UWS /]# . /root/newhome  /etc
```

(Notice the leading dot at the beginning of the command **and** the space between this leading dot and the expression **/root/newhome**. This small change makes all the difference!)

Question

- **Q5.28)** What is the value of **$HOME** now? What happened?

As you should have just seen, the shell allows you to override the resetting of variables by putting a space between the dot and the shell script (referred to as dot command). This trick allows you to

penetrate the different onion-layers of the Linux system. Sometimes this is exactly what you need, but be careful when using it, since there is great potential for confusion.

Shell Scripting Exercises

In laboratory 2 we noted that directories with permissions such as: **777**, **766**, **733** and **722** can be a serious security hazard, since they permit operations indiscriminately. In order to find such directories the **find** command in conjunction with the -**perm** qualifier can be very useful:

To demonstrate this, make the following directories (as root):

```
[root@UWS /]# mkdir /insecure
[root@UWS /]# chmod 777 /insecure
[root@UWS /]# mkdir /home/crazy
[root@UWS /]# chmod 777 /home/crazy
```

and continue with:

```
[root@UWS /]# cd /
[root@UWS /]# find ./ -type d –perm 777  –print
```

Questions

- **Q5.29)** Besides **/insecure** and **/home/crazy** are there any other unprotected directories? Try with at least one other insecure permission (766, 733 or 722).
- **Q5.30)** What does the -**perm** in the **find** qualifier do?

Considering the fact that the command **chmod 755** would render these directories more secure, try to answer the following question:

Question

- **Q5.31)** What would be the **find** command line to change the directories from **777** to **755**? (Tip: besides the -**perm** qualifier, the **–exec** qualifier will come in handy as well). **Do NOT experiment by running potentially malformed commands as root in your / directory.**

Now let's write a simple shell script, called **dir_checker**, to find any insecure directories automatically, taking the starting directory as an input variable ($1). Create the following script, in an editor of your choice, and save it in /root:

```
#!/bin/bash

# remember $1 refers to the input, which should be the start
# directory, also remember that the hash introduces a comment
echo 'Checking for sub-directories with insecure permissions in the
directory:' $1
```

```
find $1 -type d -perm 777 -print
find $1 -type d -perm 766 -print
find $1 -type d -perm 733 -print
find $1 -type d -perm 722 -print
exit 0
```

Before you can run the script remember to type:

```
[root@UWS /]# chmod +x dir_checker
```

this will allow the shell script to be executed (without it you will get an error message: *permission denied* simply because the execute permission is not set per default).

After this you can run your shell script twice with different arguments ('**/**' and '**/home**') for the directory starting point the script should look at:

```
so_comp_e113a-01# ./dir_checker /
so_comp_e113a-01# ./dir_checker /home
```

Questions

- **Q5.32)** Does your shell script work?
- **Q5.33)** Comment on the different output.

By replacing the -**print** predicate at the end of each find-command line in the shell script with: -**exec chmod 755 {} \;** you can alter the shell script to automatically change the permission of each insecure directory that was identified. (Note there should be no space between the braces {}). Name your new shell script **dir_protector** and run the script as before giving the starting directory as /home.

Question

- **Q5.34)** Check whether your shell-script works by running it twice in a row (the 2nd time no directories should be found) or by checking the permissions using the **ls -l** command on the directories that were found to have the permission **777**. What is the output of the script after the 1st and 2nd run?

In laboratory 4 we used the **renice** command to change the priority of an executing process. In order to do this we had to know the PID of the process in question. Write a shell script that …

- **Q5.35)** Given a **<PID>** and nice number will **renice** a process with the given **<PID>** to the given nice number. This should be a one line shell-script using two input variables **$1** and **$2**, representing the nice number and the **PID** of the process. Try it out with your cruncher script.
- **Q5.36) Advanced: Try to get the script to find the required <PID> if provided with only a process name only and a nice number.** (There are several possible ways of doing this and you may wish to spend some time investigating a few possible solutions.)

– END OF LAB –

Metacharacters

A shell *metacharacter* is a character that has a special meaning (instead of a literal meaning) to a computer program, such as a **shell interpreter** or a **regular expression.** The following table presents the most important ALL-Shell Metacharacters. (All-Shell means: sh, bash, ksh, csh, tcsh, …). Also appended are some metacharacters for the C-Shell csh.

`>`	redirect output
`>&`	redirect output and error log
`>>`	append output
`>>&`	append output and error log
`<`	redirect input
`|`	pipe commands
`<<str`	take input from following lines up to the line containing *str* only
`*`	match any string
`?`	match any character
`[ccc]`	match any char in *ccc*
`[1-4]`	match any char in the range *1* to *4*
`;`	run commands sequentially
`;;`	separate elements of a case construct
`&`	run commands in the background
`` `...` ``	define a command
`(...)`	concatenate commands in a sub-shell
`{...}`	concatenate commands in current shell

`$0`	command name
`$n`	shell argument *n* with *n* ranging from *1* to *9*
`$var`	value of shell variable var
`\`	stop shell from expanding character; continue command on new line
`\c`	add special character to command line (see *C* conventions)
`'...'`	define string
`"..."`	define string
`#`	Comment
`=`	assign value, no blanks allowed (ex. v=var)
`&&`	run conditionally AND (ex. p1 && p2 means run p2 if p1 ok)
`||`	run conditionally OR (ex. p1 || p2 means run p2 if p1 fails)

C shell Metacharacters (C shell ONLY)

`~`	home (*tilde*)
`>!`	redirect output and ignore *noclobber*
`!`	specify history command (!n means *redo command n*)
`:`	precede substitution modifiers
`^`	special history substitution(*circumfle*