1 Lab 12

Date: Nov 7, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

In the listings which follow, comments are any text extending from a `#` character to end-of-line.

1.1 Aims

The aim of this lab is to introduce you to some advanced features of the Unix command-line. After completing this lab, you should be familiar with the following topics:

- Using the output of Unix commands within other commands.
- The difference between shell and environmental variables and the use of each.
- Control constructs.

1.2 Background

Recall from lab1, that there are many Unix shells, usually belonging to one of two main families: `sh`-based shells and `csh`-based shells. In this lab, we will be using `bash` which is a `sh`-based shell.

Unix shells are programming languages in which it is possible to write programs with minimal red-tape (such programming languages are referred to as *scripting languages*). Hence they have variables as well as control constructs like conditionals and loops.

Since shells have minimal excise, there are no variable declarations. Variables simply come to life when they are assigned to. In `sh`-based shells a variable assignment of *value* to some variable named *name* simply looks like *name*=*value*. There cannot be any space around the `=`. Hence `dir=tmp` assigns the value `tmp` to the shell variable `dir`.

In both families of shells, the value of an existing variable can be accessed by preceeding its name by a `$`. Hence after the assignment above, `$dir` would result in `tmp`.

A shell is setup to run in a *read-eval-loop* where it reads a *command* from its input and evaluates it. The input is line-oriented with successive commands usually separated by newline characters. Before executing each line the shell parses it in several phases; during the parse, several characters are regarded as

special characters (AKA *metacharacters*). Most non-alphanumeric characters except a very few (comma (‘,‘), forward-slash (‘/‘), colon (‘:‘), period (‘.‘), at-sign or ampersat (‘@‘), hyphen (‘-‘) and underscore (‘_‘)) should be regarded as metacharacters and quoted if they are not to have their special meaning. A special character can be quoted by preceeding it by a backslash character \ or by being enclosed within single quotes '. Double-quotes function as a weak-form of quoting (variables, backslash-escape sequences and some other special characters are still interpreted within double-quotes).

The syntax of a shell command is complex, but one can think of shell commands as operands separated by operators with the following operators (in decreasing order of precedence):

Binary | The *pipe* operator used for redirecting the standard-output of one command to the standard-input of the other.

Binary && and || Control operators which evaluate their second operand based on whether or not the evaluation of the first operand succeeded (similar to C's && and || operators).

Postfix ; and & Command terminators/separators. & will evaluate its operand in the *background*.

As mentioned earlier, Unix shells are full programming languages and have programming constructs. This lab is restricted to those constructs which are useful within a single line. However, there are many other facilities include `if` and `case` conditionals as well as loops and functions. Refer to a `bash` manual for details.

1.3 Exercises

1.3.1 Starting up

Use the startup directions from the earlier labs to create a `work/lab11` directory and fire up a terminal whose output you are logging using the `script` command. Make sure that your `lab11` directory contains a copy of the files directory.

The exercises all assume the use of a `bash`-shell. So if you are using another shell, please start a `bash`-shell by typing `bash` at your command-prompt.

1.4 Exercise 1: Quoting

In Unix, a filename can contain any character other than a forward-slash `/` (which is used as a *path-separator* character) and an ASCII `NUL` character `\0`. However, since many non-alphanumeric characters are special to the shell, specifying the names of files which contain special characters requires quoting. This exercise deals with that. Change over to the quote-files directory. Do an `ls` and

you will notice filenames containing characters which are special to most Unix shells.

To understand how quotes work within `bash`, let's first play with using backslash as the quote character. Try the following commands:

```
$ echo \\
$ echo \'
$ echo \*
$ echo \$HOME\"\\
```

(Recall that `echo` is a command which merely echoes its arguments). The above examples should show you that \ can always be used to quote the following character.

Now play with characters enclosed within single quotes (''). You will see that no character within single-quotes is special in any way.

```
$ echo '\'
$ echo '\\'
$ echo '*$~'
```

Since no character within single-quotes is special, there is no way to specify a single-quote within single-quotes. If you try something like `echo '\''`, you will get a secondary shell prompt > as the shell terminates the first string at the second occurrence of ' and then starts looking for a terminating quote for the third '. If you type in a ' at the > prompt, you should see output containing a \ and a newline character.

```
$ echo '\''
> '
```

Now try using " as your quote character:

```
$ echo "\"\\"
$ echo "$HOME"
```

You will see that some characters are still interpreted specially within the double-quotes.

It is possible to spread a single command over multiple physical lines by quoting the newline character (usually by a backslash). For example,

```
$ echo \
> abc \
> 123
```

This is useful for splitting up long commands over multiple physical lines. After processing the quoted newlines, the shell sees only a single logical line.

Recall the file-globbing patterns from lab1. Now use those along with your knowledge of quoting and the `ls` command to:

1. List precisely those files in the current directory which contain a dollar sign `$` in their names.
2. List precisely those files in the current directory which contain a single-quote `'` in their names.
3. List precisely those files in the current directory which start with a back-slash \.
4. List precisely those files in the current directory which contain a name consisting of exactly 2 characters.

Is the result for the last question consistent with what you would expect after looking at the output of a simple `ls` command? According to `ls`, there are two filenames `**` and `-l` consisting of exactly 2 characters. If you succeeded in picking those out using a globbing pattern, the `ls` program would have seen `-l` and `**` as its arguments. It would regard `-l` as a command-line option specifying a *long listing* for its remaining argument `**`.

The problem is that globbing is implemented entirely within the shell. This is often advantageous but the flip side is that a command like `ls` cannot distinguish the origin of an argument `-l` as originating from a glob pattern or actually typed by a user as an option. You cannot fix by any quoting within the shell.

Fortunately, many modern Unix commands allow a special option `--` which guarantees that the following arguments are not treated as command-line options. Now use `--` to fix your solution above.

1.5 Exercise 2: Shell and Environment Variables

Every process runs in an environment which is a collection of *NAME*=*VALUE* pairs. The `env` command prints out the current environment. This environment is provided to all programs which are launched by the shell. Use the `env` command to list out your current environment.

```
$ env
$ echo $HOME
```

```
$ echo $PATH
```

The environment variable `HOME` contains the path to your home directory, and the `PATH` variable contains a `:` separated list of directories which are searched for matching programs when you attempt to run a command.

Now let's add a shell variable:

```
$ echo $xxx          #not currently defined
$ xxx=123            #set a value for xxx; no space around =
$ echo $xxx          #it now has a value
$ env | grep xxx     #it is only a shell variable; not in env
```

You should see that you now have a shell variable `xxx` but it has not been added to the environment. To add it to the environment, you will need to *export* it to the environment. Try the following:

```
$ export xxx
$ env | grep xx
$ export xx=456
$ env | grep xx
```

From the above it should be clear that all environmental variables are available as shell variables, but shell variables become environment variables only if they are explicitly exported to the environment.

[For `csh`-based shells, the analog to `export` is `setenv`.]

Perform the following exercises:

1. Create a shell variable `p` which contains a copy of your current `PATH` environment variable. Verify by echoing `$p`.
2. Make your PATH environmental variable empty. Verify by `echo $PATH`.
3. Try a `ls`. You should get an error as the system cannot find any `ls` program along your (currently empty) `PATH`. (Note that since `echo` continues to work, it must be built-in to the shell).
4. Restore your environment `PATH` variable from the `p` shell variable. Verify by confirming that a `ls` command is once again working.

[Note: Environmental variables were involved in the first bug revealed by the shellshock security exploits for `bash`.]

1.6 Exercise 3: Multiple Commands per Line and Background Commands

Recall that it was possible to split a single command over multiple lines by quoting the newline character. It is also possible to type multiple commands in a single physical line by terminating each command by a `;`:

```
$ ls ; wc *
```

Usually, when the shell launches a program corresponding to a command, the shell does not regain control until **after** the program has terminated. However, if you terminate the command with an `&`, then the shell launches the program corresponding to the command and returns immediately ready to handle the next command. The launched program continues to run in the background.

Background execution is useful when you want to run a command which takes a long time to complete. For example, if you would like to find all `.c` files in the `cs220` directory, you can use the `find` command which recursively searches specified directories for all paths which meet certain conditions:

```
$ find -L ~/cs220 -name '*.c'
$ find -L ~/cs220 -name '*.c' | wc -l
```

[By now, the reason for quoting the `*.c` should be clear].

The `-L` option above tells `find` to follow symbolic links; try the command without the option to see the difference.

[The `find` command is extremely useful and it is a good idea to get an idea of its capabilities by looking at its *man page*.]

But if you want to do the same thing over the entire system, it would typically take quite a few minutes:

```
$ find / -name '*.c' 2>/dev/null
```

(the `2>/dev/null` redirects standard-error to a black-hole bit-bucket).

This will take a while to run and you would rather not wait. So you can run it in the background by terminating the command with a `&` character:

```
$ find / -name '*.c' 2>/dev/null >c-files.lst &
```

The shell returns immediately while the `find` command continues to run. Note that since we would prefer not to have `find` interrupt us with its output we have redirected its standard output to a file `c-files.lst`.

[If you let the above run around a few 10s of seconds, you should see that it has started appending names of C files to `c-files.lst` by either `cat`'ing it or doing a `ls -l` on it; alternatively, in a different terminal do a `tail -f c-files.lst`.]

You can see what jobs are currently running in the background by using the `jobs` shell built-in:

```
$ jobs
```

The above will print a small integer for each currently running background job. You can manipulate a job using its job number by typing specific commands followed by the job number preceeded by a `%` character:

```
$ kill %1
```

would kill job 1.

1.7 Exercise 4: Sub-Shells

Any commands which you type within parentheses are run in a separate sub-shell. Hence changes which are local to a shell (like shell variables, and the current directory) are not visible outside the sub-shell.

```
$ cd ~
$ pwd
$ xx=123
$ pwd; echo $xx; (xx=abc; cd /; pwd; echo $xx); pwd; echo $xx
```

The last line shows that changes made in the sub-shell does not affect the current shell.

1.8 Exercise 5: Using the Output of a Command within Another Command

If a shell command contains a sequence of characters within back-quotes `‘`, or `$(` and `)` delimiters, then that sequence of characters is executed as a shell command and the output of that command is pasted into the current command. So, for example:

```
$ grep -i 'MAIN(' `find ~/cs220/projects -name '*.[ch]'`
```

would print out all lines from `.c` and `.h` files from the `~/cs220/projects` directory which contain the sequence of characters `MAIN(` (irrespective of case).

Use the `wc -l` command to print out the number of lines in each `.c` and `.h` file in the `~/cs220/projects` directory.

[Many modern shells like `bash` allow the use of `$(...)` instead of `` `...` ``, thus allowing nesting.]

1.9 Exercise 6: Conditional Execution of Commands

Every shell command has an exit status (the return value from `main()`). The command is said to have *succeeded* if the exit status is `0` and *failed* otherwise. The exit status of the last shell command is always available in the shell variable `?`:

```
$ echo $?
```

The command `false` is a NOP except that it always fails; similarly, the command `true` is a NOP except that it always succeeds. Now try:

```
$ false; echo 123
$ true; echo 123
```

You should see that when we use `;` to separate commands, the next command runs irrespective of whether or not the preceeding command succeeded.

But now try:

```
~~
$ false && echo 123
$ true && echo 123
$ false || echo 123
$ true || echo 123
~~~
```

You will see that `&&` evaluates it's second command iff its first command succeeded (i.e. returned 0) and `||` evaluates it's second command iff its first command failed.

The `sleep` command sleeps for the number of seconds specified by its first argument. For example, `sleep 10` will sleep for 10 seconds. Use the sleep command and the above facilities to run a command in the **background** which will echo `123` to the terminal after a delay of 5 seconds. Note that you should get the shell prompt **immediately** after typing in your command, and then after a delay of approximately 5 seconds, the `123` should appear on your terminal.

1.10 References

Brian W. Kernighan, Rob Pike, *The Unix Programming Environment*, Prentice-Hall, 1984.

Web shell tutorials: do a google search on *bourne shell* or *bash tutorials*.

GNU bash Manual.