1 Lab 8

Date: Mar 26, 2020

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 lab6, 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 (underscore `_`, forward-slash `/`, colon `:`, period `.`, at-sign or ampersat `@`, hyphen `-` and comma `,`) 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

Follow the *provided directions* for starting up this lab in a new git `lab8` branch and a new `submit/lab8` directory. Start a `script` session to log your interaction into a `lab8.LOG` file.

You will be doing all your work in your `submit/lab8` directory:

```
$ cd ~/i240?/submit/lab8
```

Copy over the exercises directory:

```
$ cp -r ~/cs240/labs/lab8/exercises .
```

1.4 Exercise 1: 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`.]

1.5 Exercise 2: 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 `cs240` directory, you can use the `find` command which recursively searches specified directories for all paths which meet certain conditions:

```
$ find -L ~/cs240 -name '*.c'
$ find -L ~/cs240 -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. Unfortunately, running `find` over the entire system is not friendly on a shared system; so we simulate the `find` taking 100 seconds by `sleep`'ing for 100 seconds:

```
$ sleep 100 &
```

The shell returns immediately while the `sleep` command continues to run.

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.

1.6 Exercise 3: 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.

Change over to the 3-subshells directory. It contains a Makefile with the intent that running `make` should produce a `ls` listing of the `~/cs240` directory. However, if you run `make`, you will get a `ls` listing of the current directory, i.e. the `3-subshells` directory.

Given the fact that `make` runs each logical line using a separate subshell, can you understand the problem? Fix the problem while keeping the `cd` and `ls` commands unchanged.

1.7 Exercise 4: 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 ~/cs240/projects -name '*.cc'‘
```

would print out all lines from `.cc` files from the `~/cs240/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 `.cc` file in the `~/cs240/projects` directory.

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

1.8 Exercise 5: 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.

1.8.1 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab8.LOG` in your `lab8` directory. Add all your files to git and commit. Then merge your `lab8` branch into the `master` branch and commit your changes.

1.9 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.