1 Lab 6

Date: Mar 12, 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 the Unix command-line. After completing this lab, you should be familiar with the following topics:

- Simple Unix commands like `ls`, `cat`, and `wc`.
- The typical syntax of Unix commands.
- The use of wildcards on the command-line.
- The standard I/O streams.
- How to redirect the standard I/O streams to files and commands.
- How to get help on a Unix system.

1.2 Background

The Unix shell is a command-line program which allows users to interact with a Unix system. Even though we colloquially use the term *the Unix shell*, there are many Unix shells with names like `sh`, `csh`, `bash`, `tcsh`, etc. The first popular shell was `sh` written by Stephen Bourne.

All information within a Unix system is stored within *files*. As far as Unix is concerned a *file* is nothing but a unstructured sequence of bytes. Unlike other OS's, Unix itself does not require that the files have any kind of record structure. The structure of a file is dictated only by the application programs which manipulate the file.

The collection of files on a unix system is organized in a hierarchy of *folders* or *directories*; i.e., a directory consists of a collection of files and other directories. The root directory of the entire hierarchy is denoted as `/`. Any Unix process always has a current directory denoted as `.` (the character period), and can refer to the parent of the current directory using `..` (two period characters).

A file or directory **name** can consist of a sequence of any characters other than `/` or the ASCII `NUL` character (''\0'' in C syntax). A **pathname** is a sequence of directory names, optionally followed by a filename separated by `/` characters.

If the pathname begins with a /, then it is an absolute pathname, otherwise it is a relative pathname interpreted relative to the *current directory*.

Example names:

```
/bin/ls #an absolute path name
bin/hello #a relative path name
./hello #the file hello in the current directory
../hello #the file hello in the parent of the current directory
```

A shell **command** consists of a *command-name* followed by *options* and *arguments* separated by blanks. The arguments specify the information needed by the command whereas the options control how the command works. Usually, options begin with a - or --, but since each command defines the exact syntax of options and arguments there can be subtle differences in the syntax of options between different commands.

Example commands:

```
$ ls        #a simple command without arguments or options
$ ls -l    #list the contents of a directory in a long format (option -
l).
$ ls dir    #list contents of directory dir (an argument).
$ ls -l dir #list contents of dir (argument) in long format (option -
l).
$ ls -d -l dir #list information about dir (not contents)
$ ls --directory -l dir #long option --directory equivalent to -
d above
$ ls -dl dir #like -d -l; many commands allow short options to be combined
```

To get help for a command, use the `man` command. Example, `man ls`.

1.3 Exercises

1.3.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab6` branch and a new `submit/lab6` directory. Start a `script` session to log your interaction into a `lab6.LOG` file.

You will be doing all your work in your `submit/lab6` directory:

```
$ cd ~/i240?/submit/lab6
```

Copy over the exercises directory:

```
$ cp -r ~/cs240/labs/lab6/exercises .
```

1.3.2 Exercise 1: Some Basic Commands

In this exercise we will create some empty files using the `touch` command. (On an existing file, the `touch` command modifies its last-modified timestamp; however, if the file does not exist, then the `touch` command will create it)

Type the following commands:

```
$ mkdir dir  #create a directory for this exercise
$ touch t1.c T1.s v.c v1.s x.c dir/t2.c dir/T2.s1 dir/t3
$ ls      #see contents within current directory sorted by name
$ ls -l      #long contents of current directory: the columns will
           #show permissions (r = read allowed; w = write allowed;
     #x = execute allowed, 3 groups for owner, group and other;
     #initial d for directories), owner name, group name, size in
     #bytes, last modification time, name.
$ ls -tl     #sort by last modification time instead of by name
```

Now type in commands to list:

1. The contents of `dir` (only the names), sorted by name.
2. The long contents of `dir`, sorted by name.
3. The long contents of `dir`, sorted by last modification time.
4. The contents of `dir` (only the names), sorted by last modification time.

To familiarize yourself with how Unix `man` pages are setup, do a `man touch`, followed by `man ls` in the auxiliary terminal window. If your terminal is setup correctly you should be able to use the `PgUp` and `PgDn` keys on your keyboard to move back and forth within the man page. Minimally, the spacebar should page forward.

1.3.3 Exercise 2: Wildcards

The following characters are interpreted specially by the shell to allow specification of file-"globbing" patterns:

`*` matches any sequence of file-name characters (including the empty sequence).

`?` matches any single character.

`[`***XY ... Z***`]` matches any one of the characters *XY ... Z*. For example, `[aeiou¬ ]` matches a vowel character.

`[`***X*`-`***Y***`]` match character *X* through *Y*. For example, `[0-9]` matches a digit.

The special interpretation occurs only when there are files which match the pattern. To prevent the special interpretation, you can quote the special char-

acter by preceeding with a backspace or by enclosing within single ' or double " quotes. This is covered in a subsequent exercise,

Type the following commands:

```
$ ls T* #should only list T1.s
$ ls dir/*.? #should not list dir/T2.s1 or dir/t3
$ ls [t-v]*.c #should only list t1.c, v.c
```

Now type in the commands to list:

1. All the names in directory `/bin` whose second character is `p` through `s`.
2. All the names in `/etc` which end in the two characters `rc`.
3. All the names in `/bin` which contain exactly 3 letters.

1.3.4 Exercise 3: Standard Streams and I/O Redirection

Every process on a Unix system has access to 3 standard input-output (I/O) streams:

Standard input The process can read its textual input data from this stream.

Standard output The process can write its normal textual output to this stream.

Standard error The process can write its error messages to this stream.

By default, all 3 streams refer to the terminal. However, it is possible to redirect the streams to files or commands.

If a command is followed by a `>` character followed by a filename, then the standard output of the command is **redirected** to the file.

Try `ls > ls.log`. No output should be produced on the terminal; instead the output should have been redirected to `ls.log`. Do `cat ls.log` to see its contents (the `cat` command con*cat*enates the files specified by its command-line arguments onto standard output; if there are no files specified, then it merely copies standard input to standard output).

Try `cat >cat.log`. There should be no output. Type a couple of lines of garbage text on the terminal and terminate with a `control-D` character. Then type `cat cat.log` and you should see your garbage text displayed on the terminal (note this is a handy trick when you happen to get onto a barely working computer which does not have a functioning text editor).

If a command is followed by `>>` characters followed by a filename, then the standard output of the command will be **appended** to the filename.

Try `ls dir >>ls.log`, followed by `cat ls.log`. You should see the output of both redirections.

If a command is followed by the `<` character followed by a filename, then the standard input of the command is redirected from filename.

Try `cat <ls.log`; i.e., the `cat` command is run without any arguments, hence it will copy its standard input to standard output. Since the standard input is being redirected from `ls.log`, this command should do exactly the same as `cat ls.log` without the input redirection.

Finally, if two command are separated by the `|` character, then the standard output of the first command is fed into the standard input of the second command. The combined command is known as a *pipeline*.

For example, `wc -l` will print out the number of lines on its standard input (do `man wc` in your auxiliary terminal for other options). So try `ls | wc -l` to get a count of the number of files in the current directory.

Now type in the commands to achieve the following:

1. Produce on the terminal a count of the number of filenames in the `/bin` directory which are exactly 4 letters in length.
2. Create a file `bin-c.log` containing all the filenames in the `/bin` directory which start with the character `c`.
3. Print out the number of lines in the file `bin-c.log` created by the previous command.
4. Do a `man tr` in the auxiliary terminal to understand how to translate all lowercase characters to uppercase. Then list out all the names in the current directory with all names in uppercase.

1.3.5 Exercise 4: 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 4-quoting 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\"\\
```

The `echo` command 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 the earlier exercise. 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.

1.3.6 Winding Up

Follow the *provided directions* for winding up this lab. Terminate your `script` session producing the log file `lab6.LOG` in your `lab6` directory. Add all your files to git and commit. Then merge your `lab6` branch into the `master` branch and commit your changes.

1.4 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.

Rob Pike and Brian Kernighan, *Program design in the UNIX environment*, AKA *cat -v Considered Harmful*, AT&T Bell Laboratories Technical Journal, October 1984, Vol. 63, No. 8, Pt 2. Available as ps/pdf at *<http://harmful.cat-v.org/cat-v/>*.