1 Unix Command Line Introduction

Date: Sep 23, 2021

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 an 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 -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 `lab3` branch and a new `submit/lab3` directory. You should have copied over the contents of `~/cs220/labs/lab3/exercises` over to your directory.

Run through the following exercises in the `exercises` directory (note that the provided directory only contains a single sub-directory for the last exercise).

1.3.2 Exercise 1: Some Basic Commands

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 another directory
$ 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 an 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. Alternately, you can do a web search, though there is a chance that any resulting manual pages may not match the system you are currently using.

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 $XY \ldots Z$. 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, quote the special character by preceeding with a backspace or by enclosing within single ' or double " quotes.

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 an 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. Print out a approximate count of the total number of files and directories whose pathnames start with `/etc` (note that the `-R` option to `ls` produces a recursive directory listing where `ls` will list out the contents of the directories specified on the command-line as well as all their direct or indirect subdirectories).

   You may get permission errors while performing this step, you may ignore those errors.
5. Do a `man tr` in an 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: Standard Streams in C

Change over to the c-stdin-stdout directory. It contains a trivial Makefile and a simple C program sum-ints.c which sums pairs of integers read from either a file or standard input. It also contains a trivial test1.in test data file.

Study the code provided in sum-ints.c. Use the Unix man pages to understand the library functions which are used. In particular, be sure to look at the man pages for fopen(3), strcmp(3) and the return value for fscanf().

Build the program using `make`.

Note that the program requires the first argument to be a file containing pairs of integers to be added. However, if the first argument is specified as `"-"` then it will read the input integers from stdin.

Run the program:

```
# show a usage message
$ ./sum-ints
usage: ./sum-ints IN_FILE [OUT_FILE]

# show contents of test data
$ cat test1.in
3 5
3 -3

# sum pairs of ints from test1.in
$ ./sum-ints test1.in
8
0

# sum pairs of ints from test1.in, write results to test1.out
$ ./sum-ints test1.in test1.out

# look at result file
$ cat test1.out
8
0

# specify input file as - to read from stdin
$ ./sum-ints -
7 4
11
8 2
10
-3 5
2
# type ^D to indicate EOF

# read from stdin but write to test2.out file
$ ./sum-ints - test2.out
3 6
```

```
9 7
# type ^D to indicate EOF

# check results
$ cat test2.out
9
16
$
```

To test your understanding, do the following:

1. Use the `./sum-ints -` command in such a way so as to force the command to read the input integers from `test1.in` even though the first argument is specified as `-`.
2. Modify `sum-ints.c` so that if `-` is specified for the second argument then the resuts are written `stdout`. Make sure to modify the initial comment in the file as well as the usage message. Also ensure that your tests for this change are recorded into your `lab3.LOG` file and to also submit the modified `sum-ints.c` file.
3. Run your modified program using `./sum-ints - -` but set up the command in such a way that all input is read from `test1.in` and all output is written to `test1.out`.

1.4 Winding Up

Wind up your lab by using the *provided directions* to terminate your log in a `lab3.LOG` file and merging your `lab3` branch into the `master` branch. Once you have the lab on your `master` branch, commit and push your changes to github. Be sure to include your `lab3.LOG` file as well as the content of your `exercises` directory.

1.5 References

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

Web shell tutorials: do a google search on `bourne shell`.

GNU bash Manual at *<http://www.gnu.org/software/bash/manual/html_node/index.html#Top>*.

Mendel Cooper, *Advanced Bash-Scripting Guide* at *<http://tldp.org/LDP/abs/html/>*.

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/*>.