CS320
Lab 0
Introductory Linux

Srinibas Swain (srinibas@iiitg.ac.in)

Welcome to Lab 0 of CS320. This entire lab is designed to introduce you to the Linux operating system, some of its tools, and to help you become accustomed to the lab environment. You shall be using Linux for all practical work in CS320. Although this lab is not assessed, it is possibly the most crucial lab of the course as far as building your understanding of Linux is concerned. Please take it seriously: work carefully, and think critically about what you are doing at each step.

Linux may be unfamiliar territory for you. Even if you already have some Linux experience, you should come into the computer labs and familiarize yourself with the Linux environment before your scheduled lab session. In particular, you should treat the first section of this lab sheet as preparation work: **be sure to get through all of it before you arrive for class**.

This lab is only the beginning, however, as you will continue to learn about Linux and its tools over the semester. Here is some helpful, general advice:

- Take the time to experiment with the work that you do. Ask yourself questions as you go along. Play with the commands and tools that you meet.
- When you run into difficulties, try to work through them on your own or with a fellow student before seeking the help of your tutor. Any help that you eventually receive will be of greater value if you have already thoroughly thought about the problem you are tackling.

We hope you enjoy the lab and, more generally, the unit!

1 Preparation: The CS320 lab environment

This part of the lab will help you get started in your Linux work environment[1] and introduce you to the fundamental tasks that you'll need to be able to carry out while doing the unit. You should treat this section as preparation work: try to get through as much of it as possible before you arrive at your scheduled lab.

1.1 Orientation

For our work in Linux, we use the *Bash* shell via the desktop terminal program.[2] Bash is an example of a shell, which is a program that acts as your text-based interface to the operating system. It takes your typed commands as input and displays text in response. This program is already running, and waiting for you, when you first start using Linux; you don't need to do anything to start it.

Open up a terminal in your Linux desktop environment (look under "Accessories" in the "Applications" menu on the task panel). You should see the $ prompt of the Bash shell as it awaits your command.

Spend a minute or so *experimenting*. Try pressing Return; what happens? What if you enter some spaces (then Return)? What if you enter some random text? Try the commands `pwd` and `ls`.

Files in Linux are organised into *directories*. These are arranged in a tree: every directory has one parent directory, and may have one or more *subdirectories*.[3]

At any given time, as you interact with Linux via Bash, you have a "current location" within this directory hierarchy. This is known as your *current directory* or *working directory*. You can find this out using the command pwd, for *print working directory*.

[1] If you already have Linux installed on a laptop that you can bring to class, you're welcome to work within that environment. However, CS320 teaching staff are under no obligation to assist you with any problems that arise through the use of your own computer for classwork

[2] Although the GUI interface would work just as well, treat this as an exercise in getting to know the basic commands and behaviour of Bash — an extremely powerful programming tool in its own right. Also, many programs are run from the terminal, including compilers, interpreters, and powerful data-processing tools. Learning to work in the terminal efficiently is essential to becoming a skilled Linux programmer.

[3] Think about the logical implications of this statement. Is it entirely correct? If not, how would you correct it?

```
$ pwd
```

The output from this command is a *path*. Read from left to right, a path pinpoints a location in the filesystem by specifying the route to that location from a source reference point. In this case, the root directory / is used, which is the top-most location in the filesystem's tree.

You have a special directory called your *home directory*, which is the default location of many things you do, and the usual starting point for your own subtree of directories containing your own files. It is also, by default, your initial working directory, when you first login. So, if you entered `pwd` as above, just after logging in, Linux should respond with the path to your home directory. In the current Linux system in labs, this should be

/srv/home/username

Whereas your working directory changes as you move around in Linux, your home directory is always the same.

You can find out what's in your current directory using the *list directory* command:

```
$ ls
```

You will see a list of names of all the files and subdirectories that belong to the current directory. This list is usually in multicolumn format (i.e., several names per line). If you prefer one name per line, you can use the -1 option:

```
$ ls -l
```

This list includes the `Documents` directory. You will need to store your CS320 work there, since files can be stored in this directory from week to week. To this end, you will shortly make an CS320 *subdirectory* here.

You can move from one directory to another using the change directory command. For example, to go to the Documents directory, enter

```
$ cd Documents
```

You can go to the parent directory using

```
$ cd ..
```

You can go to the home directory using

```
$ cd ∼
```

In fact, ∼ is a standard Linux shorthand for the home directory. So, if you are elsewhere in the filesystem and want to go to your home directory, you can do `cd ∼` which is easier than typing out the whole path to the home directory, as in `cd /srv/home/username`. Similarly, . (a single full stop) is an abbreviation for the path to the current directory, and .. is an abbreviation for the path to the parent directory of the current directory.

You can find out what's in the root directory by going there. Try

```
$ cd /
$ pwd
$ ls
```

Now go back to your `Documents` directory. (How can you do this with as few keystrokes as possible?)

2 Some useful commands and tools

Try out, and play with, each of the following commands. The `echo` command just repeats any strings you give it, followed by a newline. For example:

```
$ echo Panini
Panini
$ echo Alan Turing
Alan Turing
$ echo “Alan Turing”
Alan Turing
```

Although this command doesn't do much, it can be useful when playing with other commands. You can copy a file using the `cp` command:

```
$ cp filename1 filename2
```

creates a new file which is an exact copy of the first file. Try this, and then use ls to see that the new file does indeed now exist.

If you don't want the copy after all, you can remove it with rm.
If you want to rename a file, you can move it:

```
$ mv filename1 filename2
```

You can cause the entire contents of a file to be output for the user to see:

```
$ cat filename
```

This is useful for displaying small files. But for longer files, the contents may pass in front of you too quickly. If the file is too large to fit in one terminal window, try more, which displays one screenfull at a time (press the space bar to move forwards to the next screenfull), or less, which is like more except that it also allows you to move backwards by pressing “b”.
You can also use cat to concatenate several files and display them all:

```
    $ cat filename1 filename2 filename3 ···
```

The wordcount command gives a one-line output containing the numbers of lines, words, and characters (including newline characters) in the file. For the purposes of this command, words are delimited by spaces, tabs, or newlines.

```
    $wc filename
```

Now, try wc without any filename:

```
    $ wc
```

It seems nothing happens, and you don't get a $ prompt. In fact, the system is waiting for your input, from the keyboard. Such input is known as `standard input`. Type a few words, then Return, then a few more words, and so on. The system accepts your input but does nothing with it; you see no output. After a few lines, type Control-D. This terminates the standard input to this program.
Now, `wc` gives output to the screen, giving counts of the lines, words and characters you typed in standard input.

You can sort the lines of a file:

```
$ sort filename
```

Note that this will not change the original file. A sorted version is shown to the user, but then disappears (in the sense that it is not stored in any file). Fortunately, we will shortly see how to retain it.

The command `uniq` removes repeated lines. So, if you have two or more identical lines in a row, all but one is removed. Lines that are identical but have other, different lines in between them are kept; neither is removed, as they are not repeats. Again, the output (with repeated lines removed) goes to the screen; the original file is unchanged. The option -c causes each line of the output file to have, at its start, the number of repeats of that line that there were in the input file.

3 Input, output, redirection, pipes

By default, the output from a Linux command is to the screen where it is displayed in the terminal window (standard output). But you can change this.

Some commands will have a specific way of nominating an output file. Examples:

- `sort` can take the name of an output file given as part of the `-o` option. So

  ```
  $ sort inputFile -o outputFile
  ```

  places the sorted output into file *outputFile*.

- `uniq` can take the name of an output file as a second argument. So

  ```
  $ uniq inputFile outputFile
  ```

  places the output in outputFile. Here, there is no `-o` option.

In each case, any existing file of the same name is overwritten — so take care.
To determine how output files are specified for a particular command, you need to look up its documentation (e.g., its manpage). It is instructive to try to use `sort` as above but without the `-o`, and explain what happens.

If a command or program in Linux gives output to the screen (standard output), then you can *redirect* this output to a file, say *outputFile*, by using > *outputFile*. Whenever output goes to a file, the output is not seen by the user at that time. To see the output, the user must inspect the contents of that output file (for example, by `cat` *outputFile*, or by opening it with an editor). For example,

```
$ ls > directoryList
```

places a directory listing in the file `directoryList`, rather than showing it on the screen.
You can *append* output to a file using >> *outputFile*. This means that the original contents of outputFile are still there, but the file is now enlarged by having the new output at the end as well. For example,

```
$ echo ”The End” >> outputFile
```

puts the string “The End” (without quotation marks) at the end of outputFile.
Exercise: Suggest two ways to concatenate two files, placing the combination into a single new file.

You can also do *input redirection*. Some commands may wait for you to type some input to them (*standard input*). We have not seen much of this yet; all our commands so far either take no user input at all, or take input from a nominated file (although we did briefly try `wc` with standard input). If a command uses standard input, you can use < *inputFile*, after a command, to specify that the input is taken from the file *inputFile*. The command will then execute without expecting anything further to be typed by the user.

Sometimes, you want the output from one program to be used as the input to the next. You can make this happen using *pipes*, represented by a vertical bar, |, as follows.

```
$ command1 | command2
```

Here, *command1* is executed first, and the standard output from it (which would normally be displayed to the user) instead becomes the standard input for *command2*, which is executed second. For example,

```
$ ls | wc
```

can be used to give information on the number of files. You can have a whole series of commands, linked by pipes in this way.

4 The character translator tr

The command `tr` is useful when you want to translate characters according to a mapping you specify.

```
$ tr string1 string2
```

The meaning is that, for each position *i* in *string1*, the character there is mapped to the character at the corresponding position in *string2*. It's like specifying a function, in mathematics, with *string1* listing the characters in the domain, and *string2* specifying what the function does to each of those characters.

If *string2* is omitted, then `tr` simply removes all characters appearing in *string1*, without replacing them by anything.

The command does not specify an input or output file. Input is from standard input, and output is to standard output.

```
$ tr abc 123
abracadabra
12r131d12r1
open sesame
open ses1me
Control-D
$
```

One typical application is to convert all letters to lower case.

A handy option is `-s`, for squeeze, which removes consecutive repeats of nominated characters. There are also special ways of defining ranges of characters and special names for commonly used character sets. Check out the manpage for details of these and other features.

You can use redirection if you want to use files for your input and/or output.

Exercise: How would you use tr to concatenate all the lines of a file so that the file just contains one single long line (with all line breaks removed)?

5 The stream editor sed

The stream *editor* `sed` enables you to transform a file, line by line, according to rules you specify. A rule specifies a pattern to be replaced, and what the *pattern* is to be replaced with. `sed` is very powerful; for a full description, see its manpage, or one of the many introductions to it such as https://www.gnu.org/software/sed/manual/sed.html and http://www.grymoire.com/Unix/Sed.html. The most basic way to use `sed` is as follows.

```
$ sed 's/pattern/replacement/' filename
```

The text between single-quotes is called the *script*, and this gives the rule to be used on the input file, *filename*. A *pattern* can be a simple string of characters, and the *replacement* could be the string that you want it replaced by. In that case, running `sed` as above causes the first occurrence of the pattern string in each line (if such exists) to be replaced by the replacement string. Lines without the pattern string are not changed. The output goes to the screen, as usual. If you want all occurrences of the pattern in every line to be replaced, you can append "`g`" to the script:

```
$ sed 's/pattern/replacement/g' filename
```

For example, the following command replaces every occurrence of `A Lady` by `Jane Austen`:[4]

```
$ sed 's/A Lady/Jane Austen/g' filename
```

You can include special characters in these pattern strings. For example, the tab, newline and forward-slash character are represented by \t, \n and \/, respectively. (The latter is necessary if you want to include a forward-slash character in your pattern, since that character is also used in the script to delimit the pattern.)[5]

Patterns can be more general than just one specific string. To match any one of a set of characters, use a list within square brackets (with no commas, or other separators, between the items in the list). For example, [aeiou] matches any lower-case vowel. So the pattern b[aeiou]t matches any of the words bat, bet, bit, bot, but. To match a range of characters, use $[\alpha_1 - \alpha_2]$, where α_1 and α_2 are the first and last characters in the range. The order of characters in the range is alphabetic – or, more precisely, in order of ASCII values. So the range [a – z] matches any lower-case letter; [A – Z] matches any upper-case letter in the second half of the alphabet; and [# – &] matches any of #, $, %, &. If you want to match any character that is *not* in some list, or range, of characters, you can put just after [. For example, [^aeiou] matches any character that is not a lower-case vowel, and [^a-zA-Z&] matches any character that is not a letter or an ampersand.

Exercise: How would you use sed to remove all characters that are not letters?

Replacement strings can also be used to specify a variety of ways of replacing a string that matches the pattern. If you put \(· · · \) around some part of the pattern, then the corresponding portion of the matching string can be used, using \1, in the replacement string. For example, suppose you have a file of postcodes (which are six-digit numbers in India), one per line, whose first digits have been erroneously recorded as 4 (for Mumbai) when they should be 7 (for Guwahati), but are otherwise correct. Then you could correct your file by

```
$ sed 's/4\([0-9][0-9][0-9][0-9][0-9]\)/7\1/' filename
```

Here, the pattern matches every erroneous postcode, and the *subpattern* in \(· · · \) matches the last five digits of the postcode. The matched string (the entire erroneous postcode) will be replaced by 7 followed by the portion of the original matched string that matches the subpattern, i.e., by the last five digits of the original postcode.

Exercise: Suppose you have a file of prices in dollars and cents, all under $100. Use sed to remove all cents from the prices. Then consider how, instead, to round all prices to the nearest dollar.

The "1" in \1 indicates the string that matches the *first* subpattern (counting from left to right in the pattern). You can have up to nine subpatterns, referred to in the replacement string by \1, \2, · · · , \9. For example,

```
$ sed 's/ \([0-9]\)\/\([0-9]\) / \2\/\1 /' filename
```

[4] Jane Austen (1775–1817) is one of the most popular authors in English. This year is the 200th anniversary of her death. Her first novel was published under the pseudonym, "A Lady".

[5] An apostrophe in the pattern is trickier. To match it, instead of just using a single apostrophe ', use '\' '

takes a fraction, with single-digit numerator and denominator and a space on each side, and swaps the numerator and denominator.

Exercise: Use sed to insert a space between every pair of adjacent letters in a file.

6 Frequency count

Let us now combine some of the tools and skills we have learned, in order to determine frequencies of words and letters in English text. This is a common task in studying the statistical properties of human language, and is useful in data compression, machine learning and cryptography.

First, find a good source of reasonably long English text files that have little or no formatting or mark-up. For example, look at one of Project Gutenberg's lists, https://www.gutenberg.org/browse/scores/top, choose a book, then choose Plain Text, and download it.

Excercises:

1. From your input file, derive one with the same words, in the same order, but with each word on a separate line.
2. From this file, find one in which each word appears only once, and is accompanied by its frequency (i.e., the number of times it occurs in the file).
3. Sort the file of word frequencies in order of decreasing frequency.
4. Now do a frequency count of *letters* in the file. You should give the frequency count in two separate files: one with the letters in alphabetical order, the other with the letters ranked by frequency.
5. A *digraph* is a pair of consecutive letters. Do a frequency count of all digraphs in a file. (Overlapping digraphs are still counted separately. For example, if the file just consists of the single word dodo, then we have three digraphs, namely do, od, and do, so the frequency count should show that do has frequency 2 and od has frequency 1.)

Once your method works on these small examples, try it on the large textfile you downloaded earlier.

7 Challenges

1. Suppose you represent each word of a file by its position in the list of words ranked by frequency. So the most frequent word (which might be "the", say) is represented by the number 1, and for all i, the i-th most frequent word is represented by the number i. If each word in your file is replaced by its corresponding number (with spaces, newlines and punctuation unchanged), how much compression of the file is achieved? It is possible to do a good estimate of this without actually implementing this compression.
2. Suppose we do `ls | wc` as above, followed by a sequence of further applications of `| wc`.

```
$ ls | wc
...
$ ls | wc | wc
...
$ ls | wc | wc | wc
...
...
```

Before trying it out, can you predict how many pipes are required before the output ceases to change, and what that output will be? Having made your prediction, try it out.

What if we had started out with some other command, instead of `ls`? How would you prove that, for `every` Linux command, continued application of | `wc` eventually produces this same fixed output?

3. Read about how *Simple Substitution* cyphers work, and show how to implement Simple Substitution using `tr`.

8 List of Linux commands for this lab

`cat`
`cd`
`cp`
`cpdir`
`echo`
`ls`
`make`
`man`
`mkdir`
`mv`
`mvdir`
`pwd`
`rm`
`rmdir`
`sed`
`sort`
`tr`
`uniq`
`wc`