CS 346 – Lab 1 – Introductory Practice with Linux

With any computer, the operating system is the most important piece of software to use, because this is the manager of the machine, giving you access to your files, plus other resources such as the CPU and printer. The labs in this course will make use of the Linux operating system, which is a member of the UNIX family of operating systems. UNIX was originally developed at Bell Labs in the early 1970s and has many similar versions on various platforms today. Versions of UNIX include Linux, Solaris and BSD. I may use the words "Linux" and "UNIX" interchangeably.

Logging in

We will log in remotely to a Linux server called cs.furman.edu, which we generally call the "CS server." If you are off campus, you will need to use Citrix to connect to campus first.

On your machine, use PuTTY to remotely log in. When the PuTTY window appears, enter the machine name in the box labeled "Host Name." There are a couple of default settings that we probably should verify: The port number should be 22, and the Connection type should be SSH (i.e. secure shell). Then, click on the Open button to initiate the connection.

The remote shell window appears, and you are ready to log in! Enter your username and password. After logging into the CS server, if you were given a default password by me or by Dr. Treu, I urge you to change your password to something that only you know. Use the passwd command to accomplish this.

I have provided an appendix that lists the more commonly used Linux commands. But it's likely that after just two hours you will not be fully acquainted with the computing environment. I encourage you to come back frequently to continue your exploration of the various commands.

Part I – Essential commands

1. If you have not done so already, the first thing to do is to *log in* to the Linux server. Use Putty to log in remotely. When prompted, enter your username and password.

   Now you should see a command prompt, which means the operating system is ready for your commands. You will notice that the command prompt gives you some information concerning who and where you are: it tells you your username, the name of the machine you are logged onto, and your current directory.

2. Most of your work is done inside the terminal window. The part of the operating system that is in charge of interacting with you is called the shell. We communicate with the Linux shell through a number of commands. To tell Linux to do something, you just type in the appropriate command and hit enter. On the PCs, before people used Microsoft Windows, there was DOS. In fact, even today you can access a DOS prompt when working on a PC. Entering Linux commands on the terminal screen is just like using the DOS prompt, but of course the commands are different here since it is a different operating system.

3. As a reminder: If necessary, don't forget to change your original password! Assume that it could be compromised. Enter the command `passwd` and hit return. Follow the directions to update your password.

4. Next, in the terminal window, type in the command `who` and then hit return. This command asks the system who is logged into the machine. In the response, you should see your user-name listed, along with what time you logged in. Sometimes you may see multiple names listed; it's possible for several people to be using the same machine at the same time.

5. At this point you should carefully look over sections 1-9 in the appendix, to review the basic commands covering basic file management (creating, renaming, copying, etc.). However, at this point we probably don't have any files to manipulate! Let's create some. There are basically 2 ways to create a file: either by using an editor to type out the contents we want; or by running some command/program and redirecting the output to a (new) file. Let's try both of these techniques.

   Read sections 1 and 9 of the Linux appendix. Section 1 deals with an editor called nano, which is fairly simple to use. Section 9 discusses compilers, and there are instructions for creating an example Java program that you can compile and run.

   Now, the 2nd way to create a file. Anytime you run a program that has output, you can *redirect* the output to a file. To accomplish this, you need to use the special ">" symbol and give the name of the output file you want. For example, you can type `java Hello > hello.out`. This command says that we want to run "java Hello", and write its output to the file called "hello.out". **Caution** – if this file already exists you will be overwriting it. Linux will not warn you about this.

   Also in the appendix, you'll see a short section on the cal command. If you type "cal 2020" you'll see this year's calendar to the screen. And it's not hard to write this calendar to a file. Of course, just like any other file, you could use the "more" or "less" commands to look at it.

6. We'll do much more with Linux later (e.g. there is such a thing as "input redirection"), but for the time being, it would be a good idea to experiment with renaming, copying and moving files. I recommend you create a new directory called lab01, and put the files you created today into this folder. In future weeks you'll be creating new projects, and they'll each go into separate folders for good organization.

   To continue with the lab you will need to copy some files from the class Web site directory into your lab01 folder. Navigate into your lab01 folder (cd ~/lab01), and execute this copy command: `cp –r ~chealy/www/cs346/lab01/* .`
   Be sure to type a space between the * and the period (dot), and note that the period is part of the command.

Part II – I/O Redirection and Shell Scripts

Next, the focus is on using the Linux operating system more effectively. I hope that you will be able to use some of these techniques when running and debugging the code for future labs, and even other CS courses.

Up until now, all of the I/O for our programs has been interactive: you enter input at the keyboard, and the output immediately goes to the screen. But this is not a feasible approach if there is a lot of I/O, or if you want to run your program many times for testing purposes. The simple techniques we'll explore today will get us over the hurdle.

Within your lab01 directory, use the cd command to enter the subdirectory called part2.

Redirecting I/O

The Java program Count.java is short and simple. It reads characters from standard input and counts how many are capital letters. Once there is no more input, it prints the total for the user. The input loop will terminate when it reaches end of file. Compile this program using the Java compiler javac like this: "javac Count.java". If there are no errors, this command will produce the Java byte code in a file called Count.class. Run the program interactively using the command "java Count". You can signal the end of file by hitting return and then ctrl-D. Then the program should then give you the answer.

Note: When you use ctrl-D to halt inputting, be sure to hit ctrl-D just once. If you type ctrl-D twice, the 1st ctrl-D will exit your program back to the command line, and the 2nd ctrl-D will then close your window. In general, ctrl-D is a signal that you are done with a program. You can think of the command line itself as a program, which happens to be called the "shell" whose job is to allow you to communicate with the operating system. Hitting ctrl-D in the shell means you are done with the shell, logging you out.

But let's say you have a lot of input. Consider the file input.txt. How many capital letters does this file have? It is *not* necessary to rewrite the program. And there is certainly a faster way than typing the whole thing by hand. We basically want to put this input file into standard input. This is called redirecting, because standard input is usually the keyboard. We accomplish input redirection using the symbol <. So type this command: "java Count < input.txt". You have just told the operating system to run the count program, but use input.txt as the standard input instead of the keyboard.

Similarly, we can also redirect the *output* so that it goes into a file rather than to the screen. This is extremely useful in case there is a lot of output that cannot fit on the screen. The output redirection symbol is >. So if you type the command "java Count < input.txt > output.txt", it will do the input redirection as before, and it will also create a new file called output.txt, and put the program's output there.

WARNING: Be very careful about the redirection operators < and >. Do not confuse them. In particular, when you use >, the file name that follows will be erased if it happens to exist already.

And it is possible to redirect the output only, if you are interested in a little interactive (keyboard) input, but expect a lot of output. In this case, we would type:

"java Count > output.txt". Feel free to experiment with these combinations of redirection. You may also call your files anything you wish – I was simply using the .txt extension for clarity.

How many capital letters are in the file input.txt? _____

Writing and Running a Shell Script

A UNIX saleslady, Lenore,
Enjoys work, but she likes the beach more.
She found a good way
To combine work and play:
She sells C shells by the seashore.

Redirecting I/O can make program testing more convenient. But what if you want to run your program many times, each on different input files? Very often we have a situation where we have a working program, but we need to test it thoroughly to make sure we've handled all the special cases. This is where a shell script is handy.

A shell script (also called a batch file) is simply a text file that contains multiple Linux commands that you want to run consecutively. For example, if you want to run a program 10 times, then the shell script would contain commands that explicitly run your program 10 times. If you find a bug in the program, you can re-edit the program, and then invoke the shell script to re-run all your test cases.

A good name for a shell script is "run" or something more descriptive. To invoke the shell script, type the dot, slash and then the name of the shell script, as in "`./run`". The dot and slash are necessary so that Linux knows exactly where the shell script is (the current directory). Make sure that you have execute permission on the file: you can do this by typing the command "chmod 755 run" – this makes the file accessible and executable by anybody, but only you can modify it.

We are now going to use a shell script to help us track down and isolate bugs in our program's output.

1. Examine the program Convert.java. This program will ask the user to enter a military time, such as 0730, and then it will print it in civilian format, e.g. 7:30 am. Basically, the algorithm has to determine the hour number, the minute number, and whether it is am or pm.

2. Unfortunately, as you will find out, the program does not always work properly. It contains a few small bugs. Run the program interactively with example input like 0730 and 1420.

3. It's tedious to run the program interactively, so let's use a shell script. Examine the file called "run". Please call me over if you do not understand what all these commands mean. It turns out that the 24 input files are called 0, 1, 2, …, 23. Ultimately, the output file created is called output.txt. The temporary output files like 5.out are actually deleted to save space. To run this shell script just type "run". It may take out

15 seconds to finish. (Sometimes when you know that a shell script is going to take a long time, it's a good idea to insert "echo" commands at certain points in the shell script just to reassure the user that there's no infinite loop.)

4. Sometimes when you run a program or a shell script, it can be disconcerting to have no output appear. Did anything happen at all? If you see no interactive output, then what has likely happened is that your output has found its way to a file. Which file? Let's print out a full directory listing, in chronological order, to see if any file has just been created. Type this command: `ls -latr`. This is the command I like to use when I want to see a full directory listing, and the last file printed is the newest one. You should verify that output.txt is your newest file, and you can check its modification time against the clock with the `date` command.

5. Next, examine the file output.txt. Does the output make sense? Do you see some evidence of bugs in the program? Modify the Convert.java program so that it will fix these bugs revealed in this test set. √

6. Let's add some test cases. Create some new input files 24, 25 and 26 (more if you like) that test other possible input cases. It turns out that there is another bug that is not uncovered by cases 0-23. See if you can generate such a test situation, and then fix the bug in the program. Here's a hint: don't worry about any error checking of bad input, but consider the range of values for the minute. You will also need to modify the "run" script so that it runs the program with these extra input cases. Diagnose the bug and fix convert.java so that it will work correctly in all cases. √

Part III – Using the Command Line

A program usually gets its input from 2 sources: either interactively with the user, or from a file. But there is a 3rd place we can provide input: at the command line when we invoke the program. This is helpful if the amount of input is going to be small. In fact, every Linux/UNIX command works this way. For example, the cp program is designed to copy files. It takes two file names as input: the source file and destination file. The cp program does not interactively ask us "What files do you wish to copy?" We specify these file names at the command line.

We will now investigate the power of the command line. Now, navigate to your part3 directory: cd ~/lab01/part3.

Consider the program Add.java. Every Java program you've written has had a main() function, and it's always taken a mysterious parameter "String [] args". Today we finally get to use this parameter. If the main() function is taking parameters, this would imply that some other "function" is calling main(). Who? Actually, when you invoke a program, it is basically the operating system (the loader program) that is calling main(). Similarly, when main() returns, it returns control to the operating system.

The argument to main() is an array of String objects, customarily called `args`. (short for "arguments", and argument is another name for parameter) And because array indexing begins at zero, args[0] will be the string representing the first argument, args[1] is the 2nd argument, etc. How do we know how many arguments there are? The size of any array in

Java can be found by using the "length" attribute like this: args.length. Thus, the last argument will be args[args.length – 1].

Use the javac command to compile, and run the add program by typing "java Add". This means that we are not providing any command line arguments. The sum by default is zero.

Notice that the program is written with a loop so that it can accept as many integers as you want. Experiment with this program with various command-line input. Don't use plus signs between arguments. Try running the program like this:

```
java Add 3 5 9
java Add 7 –5 0 –3
```

Note that this program uses the built-in Integer.parseInt() function defined in the java.io package. This function (method) belongs to the Integer wrapper class, and you can look it up at the API Web address, http://docs.oracle.com/javase/8/docs/api/. The parseInt() function takes a String as a parameter, and returns an int – this is perfect for what we want to do because our command-line arguments are Strings.

But the parseInt() function can get nasty on you if you don't give it a genuine integer to parse. Try this: "java Add 6 xyz 5". What kind of error message do you get?

You'll need to handle the exception using try and catch. When your program detects an input error, have it print some helpful error message (such as "Argument 2 is not an integer. Continuing…") telling the user which argument is wrong. Have the program skip this argument (since it doesn't have a numerical value anyway) and continue with the next argument. For example, "java Add 6 xyz 5" should still report an answer of 11. √

Next, take a look at the program Range.java. Compile and experiment with this program. When you enter 2 numbers, it is supposed to print all the numbers in the range you specify (inclusive).

But what happens if you forget to provide the command line arguments, or if you only give one integer at the command line? Uh-oh! Another exception!

Modify the Range.java program so that it is robust against this type of deficient input. For example, when you run the cp command but you don't specify the files to copy, you automatically get an error message. The range program should print some informative error stating what is wrong, and what the proper usage is. The easiest way to do this is to first check the length of the args array, to make sure it has enough command-line arguments. √

Command Line Arguments as Options

Remember the "ls" command? This lists the files in a directory. Back in Part I, we saw that we could provide various options to get more information about our files. For example, "ls –t" lists the files in chronological order with the most recent file first. "ls –l" prints a long-form list of information about each file.

We can extend the range program so that it accepts options from the command line that tell the program to do different things. For example, we may want to print the numbers in ascending order or descending order.

Modify the range program so that it will take 3 command line parameters instead of two: args[0] will be either '-a' for ascending or '-d' for descending order. The arguments args[1] and args[2] will be the extremes of the range. But to be fully flexible, your program should handle the case where the two values are out of order. For example, "java Range –a 1 10" and "java Range –a 10 1" should give the same output.

When the number of command-line arguments is 3, then in the case of args[0], the program should also make sure that the option being selected is either a or d. Anything else is an error. Print an appropriate error message if the input is invalid in either respect – and then halt the program without printing any numbers.

Most of the time we will be interested in an ascending list, so let's make this the default option. Only force the user to specify an option if it happens to be the descending one. So if the user types in '–a' as the option, or no option at all, this will represent ascending. Modify the range program so that it can handle a default option in this manner. Early in the program you will need to determine if args.length equals 2 or 3 to decide whether the default option is in effect. √

Here are some example test cases:

java Range 5 20	prints 5 through 20
java Range –a 20 5	prints 5 through 20
java Range –d 20 5	prints 20 through 5
java Range –d 5 20	prints 20 through 5
java Range –u 5 20	prints error
java Range –a 5	prints error

Running Two Related Programs

Earlier we experimented with the shell operators < and > for redirecting standard input and output. There is another operator |, called the "pipe" that takes the output of one program, and feeds it as the input into a second program.

The pipe operator is often used in Linux. See if you can figure out the meaning of the following commands. Write your answers in the spaces provided. In case there are specific commands that you have forgotten about, use the "man" command to read their descriptions. √

ls | wc _____

java Add 1 3 5 7 9 | wc _____

java Range 1 100 | grep 8 _____

java Range 1 100 | more _____

java Range 1 100 | tail _____

java Add 1 3 5 | java Add _____

What to turn in

The deadline for completing this lab is six days from today, which is August 31. Please send me the following files that you modified:

Convert.java
Add.java
Range.java

Also include a text file containing the answers to the questions on this lab paper, in other words, where you had to fill in a blank.

Congratulations, you're done with your first lab! ☺ Future labs will generally not be as lengthy as this one.

APPENDIX: Handy Reference of Linux Commands

Some of the more useful Linux commands are described here. You should keep this lab handout handy until you feel more comfortable with the commands. Just one word of caution: Linux was developed primarily for computer professionals. It was designed to get work done quickly and powerfully. Consequently, it is easy to shoot yourself in the foot if you are not careful – Linux will do exactly what you tell it to do. For example, when you tell it to delete a file, it won't prompt you for a confirmation – that file is gone forever! In just a few keystrokes it is possible to destroy all your files. So be especially careful when using special wild-card characters like the '?' and '*'. (The question-mark will match any single character in a file name, and the star will match any number of characters in a file name.)

Commands for files

1. The `nano` command

   `nano` is a text editor, which allows us to create a text file. Most often we will use an editor to write a program's source code, or some other input file. Since we will be using Linux as a programming environment, it is important to become comfortable with a text editor. Another text editor on this system is vi, but it is more difficult to learn.

   To edit a file, simply type in `nano` at the command prompt, followed by the name of the file you want to edit. For example, type in `nano test1`. Since this file does not already exist, you will see it empty inside `nano`. At this point, type in the following sentence:

   ```
   This is my first text file created with nano.
   ```

   Also put in your name and date, and anything else you wish. The arrow keys allow you to navigate through the file, and the backspace and delete keys do what you would expect. Notice that along the bottom of the window, there is a list of control commands. For example, to save your file, hit control-O and verify the file name. You also have the opportunity to save the current file under a new name as well. Then, to quit from `nano`, hit control-X. If you enter control-X without saving, you will be warned and given a chance to save the file before you leave.

   If you happen to have a bigger file, using the arrow keys to go up and down can be quite slow, so to move up and down faster, there are control keys for this too: control-V goes down and control-Y goes up. (We consider the "top" of the file to be the beginning.)

   Note that it is possible to start `nano` without specifying a file name.

2. The `ls` command

   This is like the `dir` command in DOS. The ls command is saying, "Please list the files that are in this directory". Various options can give you additional information.

At the command prompt, type `ls` . (The period is not part of the command!) Not much of a list, is it? Now try `ls -a` . When we add that option `-a`, this stands for "all" files, including those whose names begin with a dot. The dot-files are special files that deal with the particular characteristics of your computer account.

Now type ls `-l` . Here, the `-l` option stands for "long listing". (That option is the lowercase letter L not the number one.) This provides more detailed information about each file, including its size and when it was last modified.

If you type `ls -la` , this combines both options: print all files, and give a long listing.

Now type `ls -lat` . The extra option t means to list the files in chronological order according to the date and time of their last modification.

Now type `ls -latr` . The option r means to list in reverse (chronological) order.

3. The `pwd` and `mkdir` commands
   `pwd` stands for "present working directory". If you type `pwd`, the system will tell you in which directory you are currently working. Since you have just recently logged in, you should still be in your home directory.

   If you want to create a new directory (for example, a subdirectory within your home directory), we use the `mkdir` "make directory" command. So to create a directory called lab01, simply type `mkdir lab01`. Note that when a directory is created, it is initially empty: there are no files inside. (You can create as many directories as you wish to hold files, and directories can themselves contain directories: this is called a hierarchical file system.)

   Now type `ls` to see that a new directory is in fact listed.

4. The `cd` command
   cd stands for "change directory". This is like the `cd` command from DOS, except that the convention in Linux is to separate directory names with a forward slash instead of a backslash. With this command, you can navigate the directory hierarchy, or give a specific destination. Here are some examples:

Command	*Meaning*
`cd`	changes to your home directory no matter where you are
`cd ~`	does the same thing as just typing "cd"
`cd lab01`	changes to the current directory's subdirectory lab01
`cd ~/lab01`	no matter where you are, go to the lab01 directory that is located right under your account's main level

`cd ..`	goes back to the next higher directory in the hierarchy
`cd /usr/share/dict`	changes to the specific directory /usr/share/dict

5. The `cp`, `mv` and `rm` commands
   `cp` stands for "copy", `mv` stands for "move" and `rm` stands for "remove".

   We use the `cp` command in order to create another copy of the same file into the same or a different directory. First, type `cd` to go to your home directory. Let's try some examples of copying files:

`cp test1 first_file`	creates a copy of the file with a new name
`cp first_file lab01/.`	creates a copy of this file and puts it in the lab01 directory Note: the "." means to keep the same name as the original.

   Now use the `cd` command to go into the lab01 directory, and then type `ls .` Do you see the first_file there?

   Now that we are in the lab01 directory, let's copy the other text file `test2` here as well. Type `cp ../test2 second_file.` This literally says "copy the file test2 from the parent directory and put it in here under the name second_file".

   (Please ask me any time you don't understand what's going on.)

   The `mv` (move) command is used either to rename a file or to move it into another directory. The big difference between `cp` and `mv` is that `mv` is not making a copy.

   Make sure you are in the test1 directory. Try these examples:

`mv first_file 1st_file`	change the name of this file
`mv second_file ..`	move this file back up to the home directory

   Now type ls to see the changes. Do you understand what happened? Go to the home directory (either by typing `cd` or `cd ..`) and type `ls` there. You should notice that second_file has just been moved here.

   The `rm` (remove) command deletes the file(s) you indicate. Once deleted, a file cannot be restored, so be very careful! Try this sequence of commands inside the lab01 directory:

`ls`	list the contents of the directory
`cp 1st_file new_file`	create a copy of the file

`ls`	list the contents of the directory
`rm new_file`	remove the file
`ls`	list the contents of the directory

6. A little help from the shell: filename completion and history.
   This is not really a command, but I wanted to share with you a couple of feature of the operating system shell. (Note: the "shell" is the part of the OS that we directly communicate with at the terminal.) Anytime you are typing some command, and you need to enter the name of some file, directory or command, it is normally not necessary to type every character. Often, all you need to do is type the first few letters, and Linux can figure out the rest. This happens when those first few letters already uniquely identify what you are talking about. For example, if you have a directory called homework, and it is the only thing that starts with the letter h, then typing the characters "cd h" and then hitting tab will complete the rest of the file name.

   If you ever need to repeat a command you recently issued, you can use the up and down arrow keys to navigate your history of recent commands. If you actually want to see a list of the commands you have typed, you can simply type "history" and hit enter.

7. The `scp` and `ssh` commands
   These are useful commands for communicating with another machine. `scp` means "secure copy" and `ssh` means "secure shell."

   `scp` is just a generalization of `cp`, and the syntax is similar. The idea is that you have an account on another machine, and somewhere in your directory structure over there, you have a file you wish to retrieve. A typical command may look like this:
   `scp mmouse@server.disneyworld.edu:menu/dinner.txt .`

   The @ sign separates the name of the user from the hostname of the machine. Following the : we have the name of the file, which in this case lies inside some directory. That final '.' means to copy the file to the current working directory on this machine, just like with the `cp` command. As soon as you enter your `scp` command, you may be prompted for a password before the file transfer can take place.

   The `ssh` command allows you to remotely log in to another machine.

8. The `more` and `less` commands
   This is one of the most commonly used commands. With the `more` command, you can view the contents of one or more text files, one screenful at a time. To proceed through a file, press the *space bar* to continue to the next screenful, or the *return* key to advance just a single line. You can exit from more when you reach the end of the file, or by hitting **q** at any time. Typing **h** gives a concise help menu for using more.

If you are looking at a small file that can fit completely on the screen, then more will immediately exit, although you can still see the file. Experiment with these:

`more /etc/motd`	Show me the message of the day.
`more /usr/share/dict/words`	Show me the spell checker's dictionary.

Believe it or not, `less` is similar to `more`, but allows a little more flexibility in navigating a file. For example, when you reach the end of a file, less does not automatically quit. Within less, you can use instant single-letter commands such as "g" to go to the beginning of a file, or "G" to go to the end. The space bar goes to the next page, and the "u" command goes up. To search for text within the file, preface your search with the slash key. To exit, hit "q".

If you simply want to dump the contents of a text file to the screen without stopping page by page, you would use the cat command.

9. Compilers
   `gcc` is the C compiler. It is customary for a C program to have a file-name extension of lowercase c. One note about using C compilers – by default, the executable file is called `a.out`, which is sometimes not the name you want. You can specify your own name for the executable by using the `–o` option, like this: `gcc –o hello hello.c`. In Linux it's customary for executable filenames not to have an extension.

   Warning! Be careful when using the `–o` option. The file name that immediately follows –o will become the executable file. You will not be warned if this file already exists. In particular, the following command would be a fatal mistake: `gcc hello.c –o hello.c`. This will have the effect of compiling your program and overwriting your source code with the executable. Your source code would be destroyed! In short, when typing –o, be sure that what follows is simply "`hello`" and not "`hello.c`".

   On the CS department Linux servers, it turns out that other compilers exist such as `g++` for C++, and `javac` for Java. Similarly, the Python interpreter is `python3`. In the case of Java, let's say you are ready to compile and run a Java program called Driver.java. Then you would enter these two commands, one to compile, and then another to run:

```
javac Driver.java
java Driver
```

<u>Commands for general information</u>

1. The `date` command
   This command prints the current date and time.

2. The `cal` command
   If you type `cal`, it will give you the calendar for the current month. If you want the calendar for some other month, then you need also to give the desired month and year, such as `cal 4 1861`. If you want to see the calendar for an entire year, you only need to give the year number, as in `cal 2002`.

   Try the calendar for September 1752. Notice anything funny about this?

3. The `man` command
   `man` stands for "manual". This may become one of your best friends in the Linux/UNIX world. If you ever come across some new command you don't understand or just want to learn more about it, the man command can tell you what it does. This command works like the less command. When you are finished reading the documentation, hit "q" to exit. Try these examples:

`man cal`	tell me what the cal command does
`man mv`	tell me what the mv command does
`man man`	tell me what the man command does ☺

   Also, if you want to look for a command, but you don't know what it's called, use the `-k` option to initiate a "keyword search".

`man -k file`	list commands having to do with files
`man -k compiler`	list commands having to do with a compiler

A few miscellaneous commands

You should probably just skim this section for now. Some of these commands are a bit advanced or not available on the machines we are currently working on. Some commands will only make sense if you are on a multi-user system. And some commands are simply not needed until you have done more work in the Linux environment. For example, you don't need to worry about running out of disk space until you start generating some big output files, which probably won't happen in this class.

1. The `look` command
   When writing some (English) document, sometimes you need to know how to spell a word, and you don't have time to get to a dictionary. With the look command, give just the first few letters and the system will tell you all the words it knows that begin with those letters. The word list is somewhat abridged, so many English words are not included, but it does contain

about 250,000 words. It does include many proper names and some abbreviations. Try look aqua to see what words begin with these letters.

2. The `diff` command
   If you have two files that are nearly the same, diff can tell you on which lines they differ. The response you will get from diff is the line numbers corresponding to the differing lines followed by what those lines look like. If `diff` doesn't say anything, then the two files are identical.

   For example, `diff one.c two.c` might give output like this:

```
23c23
<   int floor, dest, duration;
---
>   int floor = 1, dest, duration;
41c41
<       duration = SPEED;
---
>       duration = SPEED * (dest - floor);
```

   The output shows that the two files differ at lines 23 and 41. The "<" at the beginning of the line refers to the first file you specified, while the lines beginning with ">" refer to lines within the second file.

   As you can see, diff is useful when you are working with two versions of the same program, and you want to see what has changed.

   Here's another example output for a command like `diff one.C two.C`:

```
39a40,41
>       printf("going up\n\n");
>       duration = SPEED * (dest - floor);
```

   This output indicates that the second file two.C contains two new lines (40 and 41) that are not present in the first file one.C.

   If you are comparing two files that are very large and you know that there aren't many line-by-line differences, then you can use the `-h` option to make diff go much faster.

   If it turns out that there are large differences between the two files, it may be helpful to slow down the output by sending the output through more, as in:

```
diff one.c two.c | more
```

   The vertical bar (|) tells the system to send the output of diff through more. This is a useful trick in general, anytime you have a lot of output coming onto the screen.

3. The `cat`, `head` and `tail` commands
   `cat` dumps a file to the screen, which is not helpful if the file is large. More commonly cat is used for concatenating several files into one. If you still have the files test1 and test2 in your home directory, you can try concatenating them. Try this:

   `cat test1 test2 > combined`. This command creates a new file called `combined`. Take a look at this file.

   Note the '>' sign when concatenating files. This symbol is used to separate the list of files to be concatenated from the name of the new file.

   By the way, here is a general rule in Linux – any time you ask the system to create a new file for you, be sure it does not already exist, or else it will be automatically overwritten!

   `head` displays the first 10 lines of a file. Similarly, tail shows the last 10 lines of a file. You can specify a different number of lines as an option like this:

   `head -25 time.c` show me the first 25 lines of time.c

   `tail -40 prog2.c` show me the last 40 lines of prog2.c

4. The `wc` command
   wc stands for "word count". It can tell you how many lines, words and characters are contained in the files you specify. Suppose we have a file called "prog2.c", and we type wc prog2.c . The output would look something like this:

```
    162     411     2845 prog2.c
```

5. The `grep` command
   This command is often used to look for a word, phrase or some other text pattern in a file. The output from this command consists of lines from the file that contain what you are looking for. Note that if you are searching for a phrase, or a "word" containing special symbols, you should enclose it in quotes before you give the file name. Some examples:

   `grep file wordgame.c` displays lines from wordgame.c containing "file"
   `grep Eight /etc/motd` displays lines of motd containing "Eight"

   Actually, it would be even more useful if you could find out which line numbers are being printed, especially if it you are looking in some big file. To do this, pass the option –n, as in grep -n file unix . This will print all the lines from the file unix containing the word "file", but at the beginning of each line will be its line number. For example, the output may look like this:

```
105:   name of a file
220:   file name.
319:      does not get sent, but appears in your home directory
as a file
```

```
337:   name of the file you wish to create/edit.
371:   executable file will be called "a.out".  You can specify a
different name
```

6. The `du` and `df` commands
   The `du` command can tell you how you disk space is distributed among all of your directories or any directory you specify. Each line of output tells how much space (measured in blocks of 1024 bytes = 1KB) is taken up by that particular directory including all of its subdirectories.

   The `df` command gives you an overview of the entire file system. It shows how many kilobytes are in use and available on the memory device(s).

7. The `chmod` command
   If you are ever in a multi-user system, this command may be useful. It will give you some control over the security of your files. Depending on how your account is set up, by default, when you create a new file or directory, it may be readable by anyone who knows where your file is. The command `chmod` stands for "change mode". To protect your files from eavesdropping, here is what you do:

   If you only want to turn off the permissions on a single file, enter this command:

   ```
   chmod 600 <filename>
   ```

   The number 600 means that you have read and write privileges, and everyone else has no access. If you are trying to protect a directory or executable, use the number 700 instead in order to preserve execute access for yourself.

   To protect an entire directory, here is how to do it:

   Go up one directory level by typing the command "cd ..". Then enter the command:

   ```
   chmod 700 <name of directory>
   ```

8. The `ps` command
   This command will tell you what commands you have running on the machine. Each process will be printed one per line. At the beginning of the line is the process number, and at the end of the line is the name of the command. To get a list of all the processes running on the machine at this instant, use `ps -ef`. Similarly, to retrieve a list of all processes running by you, then you can use `grep` to filter the entire list. For example, if your username is dduck, you would say:

   ```
   ps -ef | grep dduck
   ```

9. The `kill` command
   In case you need to terminate a (runaway) process, try ctrl-C. If this doesn't help, you need to kill the process. That's right: in Linux you have the power to kill (a running program, that is). Look up the process number with the `ps` command, and then type kill followed by the process number. For example, let's say that `ps` shows you:

```
  PID TTY      TIME CMD
 6641 pts/1    0:01 ksh
 6735 pts/1    0:00 sleep
```

   In this case, to terminate the sleep process, we type `kill -9 6735`.