1. Semester and Year	: 2012 FALL
2. Course Number	: CS-227
3. Course Title	: Linux & System Software
4. Work Number	: LA-03
5. Work Name	: Lab 03
6. Work Version	: Version1
7. Long Date	: Sunday, 16, September, 2012
8. Author(s) Name(s)	: Jake Waffle

Task 1.5: Edit Files

In this task, we are using Vi to edit text files within a terminal. Vi has three modes that it runs in. Each mode contains a different set of commands.

The command mode is one of them and our book says that the commands are usually in the form of single letters (page24.) When I first used Vi, I would try to add text while in the command mode, but Vi doesn't work that way. The command mode is more for navigating through the text and doing simple things like removing a character/line or adding a new line.

The ex mode is for manipulating the file itself instead of the text within the file. And it allows the user to exit out of the program (with or without saving) and open new text files. To enter a command in ex mode, the user needs to type a colon (which brings the user into ex mode) and then enter the command.

The insert mode is for inserting text into a text file. And it is entered from the command mode by typing either 'i' or 'a'. After entering the insert mode, navigation is no longer possible and the user can only insert text and exit insert mode (which can be done by hitting escape and brings the user to the command mode.)

Now I'm going to edit a text file with Vi. I'll use 'sudo vi textName' to enter vi with the appropriate text file. And upon entering a random text file, I navigated around with the arrow keys (not available in non-GUI terminal 'h', 'j', 'k' and 'l' are used instead.) I removed some characters with 'x, added a new line with 'o' and then proceeded to insert mode with 'i' after putting the cursor in the new line. Then I typed in some random string of characters and used 'esc' to exit insert mode. And finally I left the text file (without saving) by entering ':q!'. I could have entered ':wq' to save and quit, but my changes weren't coherent enough to even make me want to save it.

Task 1.6: Manage Accounts

For this task I will need to add an account, delete an already existing account and modify an account. And before I can do this, the book says that I need to understand how Linux stores its passwords.

Linux stores its information on accounts within /etc/passwd and /etc/shadow. /etc/passwd is "world-readable" (pg30) so that programs can access information on accounts. But the account's password field will not contain the actual password in /etc/passwd, but it will contain a code that means it's stored elsewhere, /etc/shadow. /etc/shadow is only accessible by root (to disallow common users from seeing the passwords) and it contains account names, encrypted passwords as well as other advanced information.

Adding an account is done with the 'useradd' command (unless the user prefers to do it manually through a text-editor) followed by the username of the account (pg 31.) Doing this will setup an account with default arguments, but one can add arguments to change the defaults if needed. Arguments need an option (i.e. useradd username –d /homeDir) placed before them, indicating what the argument is actually for.

Now I'm going to add a user to my linux distro. I used command "useradd –d /home/Jake Jake" to add an account with a specified home directory (because the task needs the home directory I think.) I checked to see if /home/Jake directory exists and it does. Then I went ahead and add a password to my new account with command 'passwd Jake'.

Deleting a user's account looks very simple from the way the book describes it. All that is needed is command 'userdel' followed by the username to delete an account (pg 32.) But this does not delete the home directory or mail spool according to the book. Adding the –r option to 'userdel' will delete the home directory and mail spool along with the account. I will not delete my newly added account, until after I'm done using it for this task though.

Modifying a user's account is much like adding an account and is done with the command 'usermod' followed by options and the username of the account you'd like to modify. I tested the usermod command by changing the username of the account I previously made with 'usermod –l newName Jake'. The book does not explain that the username of the account must come after the options and that stumped me for a bit, but I got the correct information from the man pages. Then I checked to see if the account's name actually changed by reverting it back to its original name with a similar usermod command.

The book uses the example of changing an account's home directory for the usermod command (pg 32.) But changing an account's home directory with usermod will not transfer the files that existed in the previous directory. So copying the home directory to its new location and then using usermod to alter the home directory was used to get around this.

Task 1.7: Use Streams, Pipes and Redirection

For this task I will be learning about how to use streams, pipes and redirection as a Linux user and system administrator.

Understanding Streams
it goes over Standard input (stdin), Standard output (stdout) and Standard error (stderr) -- these all remind me of cin, cout and cer in the C++ programming language.

Standard input – This is something that can be read by programs as if it were a file that gives the program access to the user's input.

Standard output – Programs can write data to this as if it were like a file and the data will get printed on a terminal-like window.

Standard error – Programs can use this to display error messages on a terminal-like window and it's separate from stdout so that error messages can be handled in the background.

Redirecting Output
I used command “netstat -p > net-connections.txt” to redirect the output of “netstat -p” to the text-file “net-connections.txt.” The outputs of the prior command said that not all of the processes could be identified and that root access is needed to see everything (sudo prefix is needed.)

In response to the processes not being able to be identified, the book says that each redirection operator only redirects certain things (i.e. the (2>) redirection operator redirects standard error, while the (>) redirection operator redirects standard ouput and the &> redirection operator redirects both standard output and error.) So then the unidentifiable processes were probably related to the standard error, because the (>) redirection operator only deals with standard output.

These redirection operators also by default overwrite the file they are being saved in. But if the greater than sign is doubled (>>), the redirected content will be added to the file.

Redirecting Input
After creating error-messages.txt with command “netstat -p 2> error-messages.txt”. And installing text2gif with command “sudo apt-get install giflib-tools”. I used command “text2gif < error-messages.txt > error-graphics.gif” to redirect input from error-messages.txt to the text2gif program. Then the last redirection operator redirects the standard output into error-graphics.gif. After opening the resulting gif image said exactly what the command “netstat -p 2> error-messages.txt” outputed to error-messages.txt, but is represented as an image instead of text.

Piping Data between Programs
A tool known as a pipeline (|) is introduced. It allows standard output to be passed from one program to another.

It can be used with commands “netstat -p” and “less” to allow someone at a terminal to scroll through (with less) what would have been outputed by “netstat -p” (netstat -p | less.)

Or it can be used with commands “netstat -p” and “grep firefox” to search for lines containing firefox in the “netstat -p” output (netstat -p | grep firefox.)

And multiple pipes can be used to do even more (netstat -p | grep firefox | text2gif > error-message.gif)

Additional Pipe and Redirection Tricks
I get that the &1 and &1 string allow redirection into the standard input and standard error. But I don't really understand the rest of the given example command.

Task 3.5: Manage Running Processes

For this task I will learn how to check and terminate running processes.

Checking System Load
The system load is commonly measured as a load average. A load average can be represented as a ratio between the demanded CPU time and as much as the CPU can offer.
The system load can be checked over the last 1, 5 and 15 minutes with the command “uptime”. My computer's load averages are 0.13, 0.07 and 0.10. And that means that I'm not even using half of the CPU time that my CPU is capable of.

Obtaining Static Process Listings
The book states, “The main process-monitering tool in Linux is ps”(page 139.) And its many options are split into three categories.
Unix98 options – Single-character options preceded by a dash.
BSD options – Single-character options not preceded by a dash(-).
GNU long options – Multi-character options preceded by two dashes(--).

The -U option can be used to list a particular user's processes.
The -a option lists all running processes.
The -u option adds a column that shows the percentage of total CPU time that goes to a process.
Command “ps U user” will ouput a list of processes being run by a user.
Command “ps au” will output a list of all running processes with an extra column showing percentage of total CPU time.
And command “ps au | grep glxgears” will output lines piped in from “ps au” that contain glxgears within them.

Locating Processes Dynamically
There is also other process-identification tools amidst the Linux community. One being called top, a dynamic variant of ps (it updates its display every three seconds.)
Top allows the user to sort processes by various criteria and attain various bits of information on the running processes (such as the PID.)

Terminating Processes
To terminate a process that is hung or for some other reason, the “kill PID” command can be used. The PID refers to the process' PID and can be found with command ps and top. And sometimes the -9 option needs to be added to the kill command to force the process to terminate.

Task 3.6: Adjust Process Priorities

This task involves using nice and renice commands to adjust process priorities.

Understanding Process Priorities
The book says, “Every Linux process has an associated priority number. . .”(page 144) and this priority number ranges from -20 to 19 (where -20 is the highest priority and 19 is the lowest.) Process priorities only come into effect when CPU time becomes scarce.

Launching with Specified Priorities
The program nice allows commands to be launched with an initial priority number different to that of the default.
After using nice on glxgears, I found out that I don't have a glxgears. So I used “nice -n 10 bash” (bash was listed by “ps au”) to run a second bash program with different priorities. And after looking up the bash processes with “ps au”, I noticed that the %CPU value for the bash process implemented with nice is higher than the prior bash process (0.8 versus 0.1.)

Adjusting Existing Processes' Priorities
The renice command allows a user to change the priorities of a process that is already being run. It takes in a priority number and PID as inputs.
I used renice to make both priorities of my bash processes the same, with “renice 10 -p 4536”.