Computer Science Department

Linux OS Laboratory Manual (V 1.2)

COMP311

Nael I. Qaraeen

Approved By:

Computer Science department

May 2015

Table of Contents

Lab. #	Title	Page #
1	Introduction to Linux environment	2
2	Text editing (vi editor)	7
3	File Systems (I) (Structure and File Types)	11
4	File Systems (II) (File Metadata)	16
5	Shell Usage and Configuration (I)	22
6	Shell Usage and Configuration (II)	28
7	Job and Process Management	34
8	Text Processing Tools and Regular Expressions	41
9	Shell Scripts (I) –Introduction	47
10	Shell Scripts (II)- Programming (Selection Constructs)	53
11	Shell Scripts (III)- Programming (Looping Constructs)	60
12	Security and Networking Concepts	66

Lab1. Introduction to Linux environment

Objectives

After completing this lab, the student should be able to:

- Log into and use a Linux system
- Learn the Linux command basic structure
- Use basic Linux commands to get familiar with the Linux environment
- Get help information on Linux commands
- Identify some important Linux Files

Logging in to Linux

Login directly to your Linux system by using your username (login name) and password (provided to you by the instructor). If you are using a client machine (machine running another Operating System such as Windows) then you need to use a remote access command such as telnet to login to the Linux server as follows:

Open the start menu from Windows
Type the following command in the search field:

telnet IP address of server (e.g. telnet 172.16.2.90)

This should provide you with a terminal where you can enter your login name and password to access the Linux system.
At the terminal prompt (e.g. [username@localhost ~]$) you can type your commands to interact with the system.

Linux Commands General Format

A Linux command may contain one or more of the following three categories specified in the following order:

Command ***Option(s) Argument(s)***

a- Command name (such as ls, cp, or finger)

Type the command:
ls
Describe the result?

_____

b- Command options which indicate optional preferences on how you want the command to run. Those are usually specified using a – followed by a letter to

specify the option(s) (e.g. ls –al where the ls command has two options a (all) and l (long format)).

Type the command:
ls –a
Describe the result now?
_____.

Why did the files get listed in this case and not the previous case (hint: notice that all files names newly listed start with a dot (.))?
_____.

Type the command:
ls -al
Describe the result?
_____.

c- Command Arguments which specify the parameters that you want your command to use while executing (e.g. ls –al /etc where /etc is an argument to the ls command specifying that you want to list the files in the /etc directory).

Type the command:
ls –al /etc
Describe the result?
_____.

To be able to display the result one line (or one page at a time), you need to filter that result using filtering commands like ***more*** or ***less***.

Type the command:
ls -al /etc | more.
Press the space bar to get one page at a time.
Press Enter to get one line at a time.
Press letter 'q' to quit

An option (as the name specifies) is not needed to run a command, but arguments may sometimes be necessary to run certain commands (e.g. cp, rm, …)

Type the command:
cp
What happened?
_____.

Now type it with a source and destination file names (i.e. ***cp srcfile destfile***).
What happened? (Note: you can simply create a ***srcfile*** by running the following command before the ***cp*** command is run***: touch srcfile***).

_____.

Getting Familiar with Linux Environment

Try the following commands to familiarize yourself with Linux:

1- ***who*** (displays information about who is logged in to the system at this time)
2- ***finger username*** (you may select any user name from the who command output)
What is displayed? How do you think the system knows all that information?

_____

_____

3- ***finger***
4- **w** (displays **what (w)** the users are doing on the system)

To be able to communicate with another user try the command:

5- write username
Type your message
Type ctrl-d to stop and exit

Such commands as write and talk may be distracting for users on the receiving end. Linux gives the user the ability to allow or deny such commands from disrupting his/her work. To do this a user uses the ***mesg*** command as follows:

mesg n (prevents others from sending messages using commands like write)
mesg y (allows other users to send messages)
mesg (displays whether messages are allowed (y-yes) or denied (n-no))

Help on Commands

There are several methods to get help on a Linux system. One of the most useful is the man (manual) pages.
To get detailed information about a command type:

man command name (e.g. man ls)

This will provide you with information on the syntax of the command and a short description. It will also present the different options that may be used with that command. The ***man*** command uses the ***less*** command (similar to the more command) for displaying information. This means you can use the space-bar (one page at a time), Enter (one line at a time), and q (quit) to browse the given information.

The man pages are divided into sections. Each section gives information about the command in a different context as shown below:

Try the command:
man passwd

Gives information about ***passwd*** as a command (section 1 of the man pages) used to change your password. ***Read through it and use it to set a new password to your account***.

Did it work?_____.

Try the command:

man 5 passwd (5 is section 5 of the man pages)

This will give you information about ***passwd*** as a file on the system.
When you browse through the man pages of any command make sure you check out the see also section which provides you with the names of related commands to the one your exploring. The names of the commands are usually specified as:

command name(section number) (e.g. ***passwd(5)*** – meaning ***passwd*** in section 5 of the man pages).

Using the man pages do the following:

1- ***Find what the command du is used for with the options -h and -s;***

du command: _____.

-h:_____.

-s: _____.

2- ***What is the clear command used for? Try it.***

_____.

3- ***List any two new commands that you haven't seen before and find out what they do. <u>Hint: check the directories /usr/bin and /usr/sbin</u>.***

Command 1: _____.
Function of command:
_____.

Command 2: _____.
Function of command:
_____.

Important Linux Files

There are two important Linux files that contain information regarding user and group data. The first is **/etc/passwd** and the second is **/etc/group**.

/**etc/passwd file:**

Type the command:

more /etc/passwd

(hit the space bar until you see a line that starts with your user name)

The line will be similar to something like:

u1112233:x:520:66:Ahmad_Hamdan:/home/students/comp311/u1112233:/bin/bash

use the man pages (section 5 for passwd to see what the fields in this line represent).

u1112233:_____.

x: _____ .

520:_____.

66:_____.

Ahmad_Hamdan:_____.

/home/students/comp311/u1112233:_____.

/bin/bash:_____.

Where do you think the finger command gets the information it displays about users?

_____.

/etc/group file:

Type the command:

more /etc/group

You will find lines similar to the following:

students:x:66:ahmad,u123456

Use the man pages (section 5 for group to see what the fields in this line represent)

students:_____.

x:_____.

66:_____.

Ahmad,u123456:_____.

Lab2. Text editing (vi editor)

Objectives

After completing this lab, the student should be able to:

- Become familiar with the visual (vi) editor modes
- Use the vi editor to manipulate simple files
- Write and use simple vi macros

Linux Editors

There are several text editors on Linux but one of the most commonly used is the ***vi*** (visual) editor which was recently upgraded to ***vim*** (vi modified). Using the ***vi*** editor may seem strange at the beginning, but once you get used to it you will rarely use any of the other editors. This editor is a default on most (if not all) Linux and UNIX based systems. It provides the user with very powerful editing capabilities using very simple tools (a keyboard and a simple terminal).

Vi Modes

To get started with ***vi***, you need to understand that when using this editor you are usually in one of two modes:

1- Insert (input) mode: which is used to type characters in a file.
2- Command mode: which is used to run commands (edit) file content

When you first start a vi editing session, you are in the command mode by default. To switch to insert mode you use one of the following letters:

i- Insert before current character
I- Insert at the beginning of current line
a- Append after current character.
A- Append at the end of current line
o- Open new line below current line.
O- Open new line above current line.

To switch back to the command mode you can use the ESC (Escape) key.

Editing A Simple File

Let us go ahead and start editing a simple file.

Type the command:

vi myfirst

Now press the letter ***i*** to change to insert mode and type the following (on three separate lines):

Your full name
Your student id number
Your major

To finish your vi session, you need to switch back to command mode (***Press ESC***).
Type a colon (:). This will move your curser to the bottom left side of your page. This is called the ex mode (which is an extension to the command mode). To quit your file type one of the following:

q! - to quit without saving recent changes
w - to save recent changes and not quit
wq - to save recent changes and quit session

Save and quit the file and then open it again using the command: ***vi myfirst***

To move around in a vi session you can use the following letters (in command mode):

h - move left j- move down k - move up l (el) - move right

To delete any mistakes you make in a file use one of the following (in command mode):

x – delete current character
#x – delete # of characters (e.g. 4x – deletes 4 characters starting with current)
dw - delete current word
#dw – delete # of words (e.g. 3dw - deletes 3 words starting with current word)
d$ (or dd) – delete current line
#dd- delete # of lines (e.g. 5dd – deletes 5 lines starting with current line)

Try deleting your last name, the last number of your student id and your major line from the file myfirst.

The content deleted in the above commands is actually saved in the vi buffer (like the cut command). To be able to restore that content back you may use the commands:

p – paste left or below
P - paste right or above

If you want to copy/paste instead of cut/paste then use a y (yank) instead of a d(delete) as follows:

yw – yank word
#yw – yank # words
y$ (or yy) – yank line
yy – yank # lines

To modify your file content use one of the following (in command mode):

r- replace current character (e.g. re changes the current character to e)
cw - change word (e.g. cwnew changes current word to new)
#cw – used to change # words with new words.
c$ - change line
#c$- change # of lines with new lines.

If you decide to undo any of the changes you made to your file, use one of the following:

u – undo last change
U – undo all changes in current line
:e! – undo all changes since last save

To search a file forward for a certain pattern, you use the / (slash) followed by the pattern. For example to search for the word hello in a file you type: ***/hello***. To get the next position of the pattern type ***n*** (next).
To search a file backwards use a ***?*** instead of a /.

Vi Macros

You can create and use macros in ***vi***.
To create a macro in command mode use the map command in the ex mode as follows:

:map S 1GddGp:wq

which will create a command mode macro called ***S*** which when pressed will go to the first line (1G) cut the line (dd) move to the end of file (G) and paste the line after (p) and then will save file (:w) and quit (q).

To create a macro in input mode you use the ***map!*** command rather than map. To include control characters such as ESC or ENTER in your macro, you do the following:
Press the ctrl key then v key then the char you want. i.e. to include the ESC key in my macro I press Ctrl then v then ESC and it will be saved as the control char ^] .
To undo a macro, you use the ***unmap*** command followed by the macro's name (e.g. : ***unmap S***).

Macros will disappear after you exit your vi session. To make your macros permanent, you need to include them in a file called ***.exrc*** or ***.vimrc*** under your home directory.

Create the following macros and put them in file .exrc under your home directory:

Macro 1: called H which is an input mode macro that copies the last three lines of the file to the beginning of the file.

Macro 2: called S which is a command mode macro that replaces the current word with the word new.

Save the .exrc file and then do the following:

Create a new file called testmacros and fill it up with any five lines.

open the file testmacros and apply your macros (H and S) to it. Did they work?_____.

More vi Practice

In the brackets, write the command(s) that you would use to do each of the following:

1. Type vi linux
2. Change to input mode (_____)
3. Type the following text

Linux was first created by Linus Torvalds
from Finland.
He based it on the UNIX OS
which was first created by Ken Thompson.
Linux is open source.
There are different Linux distributions,
such as RedHat
and Ubuntu.
Linux is a very stable and popular Operating System.
Using Linux is fun.

Go back to command mode (_____).
4. Go to the beginning of the first line in the file linux (_____)
5. Go to the beginning of the last line (_____).
6. Go to the ninth line (_____)
7. Use the letters h, j, k, l, and w to move the cursor to the beginning of the word "Red" and then delete three chars. (_____)
8. Insert the word "Black" (_____)
9. Search for word open and move there (_____).
10. Delete the word "open" by using a single command (_____)
11 Undo this last deletion (_____)
12 Using the search facility move your cursor to the sentence "Linux is a very stable …".
(_____).
13. Using one command, change the word "is" to "was" (_____)
14. Move back to the first line of the file (_____)
15. Copy four lines into the buffer (_____)
16. Move to the end of the file (_____) and put the copied lines there (____)
17. Write a command mode macro called B that will save the file and quit.
(_____)
18. Write (save) the file and quit using the macro in 17.

Lab3. File Systems (I) (Structure and File Types)

Objectives

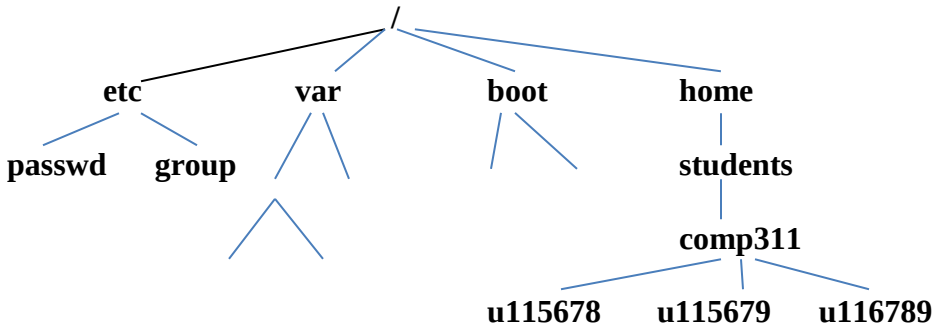
After completing this lab, the student should be able to:

- Understand the structure of the Linux file system
- Build tree structures using absolute and relative paths
- Recognize and create the different main Linux file types

Linux File Systems

A File System is a data structure that contains data about files and how they are organized. It is a logical name that represents a physical device such as a partition on disk.

The File Systems on Linux usually have directory names such as /home or /usr/local. Linux must at least have one file system which is the root file system (/).
Linux file systems are hierarchical tree structures similar to the following:



There are several common directories that usually exist in most Linux systems such as:
/etc: includes most administration related files.
/var: includes data that changes such as log files and user mail boxes.
/boot: includes system boot files.
/home: includes user home directories

To display the file systems, use the command ***df*** (display file systems) as follows:

df -h (-h human readable format).

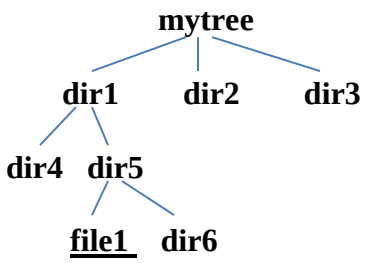
What are the physical device names as well as the mount points (directory names where file systems are mounted) for the file systems on your system?

_____

_____.

To Manipulate directories under a file system, you can use the following commands:
mkdir newdir (creates a new directory called newdir)
cd newdir (changes your position to newdir)
rmdir newdir (removes directory new directory only if newdir is empty)
rm –rf newdir (removes non-empty or empty directory newdir)
pwd (displays present working directory)

Using the commands above, create the following tree under your home directory:



All the above are directories except **file1** which is a regular file.

Commands used to create *mytree* in order:

To display your tree use the command:
ls –R mytree

Absolute and Relative Paths

Any Linux file may be referenced by either its absolute path name or relative path name. Each file has one and only one absolute path name while it may have an infinite number of relative names.

The absolute path name for file1 in the previous tree is:
/home/students/comp311/username/mytree/dir1/dir5/file1

While it has several relative names such as:
./mytree/dir1/dir5/file1
./mytree/../mytree/dir1/dir4/../dir5/file1

and so forth. Notice that (**.)** stands for current directory while (**..)** stands for previous (parent) directory.

Remove the sub tree ***mytree*** created in the previous section (***rm –rf mytree***)
Now try creating the whole tree again using relative paths from your home directory (i.e. you are not allowed to use the ***cd*** command to create any parts of the tree).

Commands used:

So far, we have mentioned two types of Linux files as follows:

1- Regular files which include scripts, binaries, as well as text files. These files contain data and are identified by any empty first slot when we list them using the ***ls –al filename*** command. These are created using the ***vi*** editor.
2- Directories which are simply containers that include the mappings between filenames and subdirectory names and their unique inode (index) number in the file system. These are identified by the letter ***d*** in the first slot when we list them using the ***ls –al dirname*** command. These are created using the ***mkdir*** command.

The third type of Linux files are the special (device) files which are usually located under the ***/dev*** directory. There are two types of device files:

1- Character device files: which are used to read and write from/to devices one character at a time (e.g. keyboard device files). These are identified with the character ***c*** when we list them with the ls –al command.
2- Block device files: which are used to read and write from/to devices one block at a time (e.g. disk device files).). These are identified with the character ***b*** when we list them with the ls –al command.

Run the command ls –al on the /dev directory. List three character device files and three block device files.

The fourth type of Linux files are the links.
The original links in Linux are called the hard links and are created using the command ***ln***.
Create a directory to try some links (***mkdir links; cd links***)
Create a file called original and put the phrase "this is original" inside then save and quit (***vi original***)
Now create a directory called mydir (***mkdir mydir***)

Let us now start working with links:

Create a hard link called filehlink to file original:

ln original filehlink

List the files in your directory with ***ls –ali*** (the i option displays the inode numbers)
Notice that all the properties of file original and link filehlink (except the name) are exactly the same (even the inode number). Hard links basically give a new name to the same inode number.
Hard links have two limitations:

1- Not allowed on directories

Try the command:

ln mydir dirhlink

What was the result?

_____.

2- Not allowed across different devices (file systems)

Try the command:

ln /etc/passwd passwdhlink

What was the result?

_____.

Those limitations are solved using a different type of links called symbolic (soft) links.
To create a symbolic link simply use the option (-s) with the ***ln*** command.

Try the following commands and see what happens:

rm filehlink
ln -s original fileslink

ls –ali (what are the differences between original and fileslink properties?)

_____.

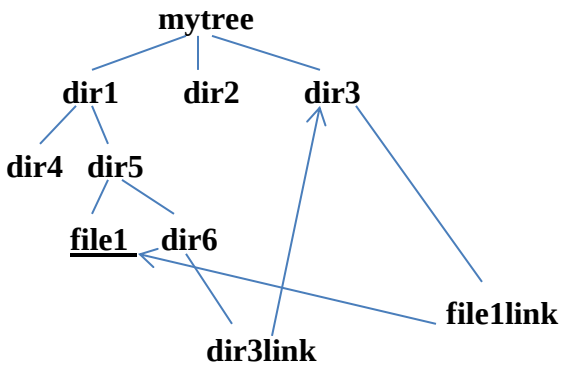
ln –s mydir dirslink (what happened now?)

_____.

ln –s /etc/passwd passwdslink (what happened now?)

_____.

Now go back to the tree you created earlier (***mytree***) and add the links shown below:



What commands did you use:

Although there are few more types of Linux files such as pipe files or socket files. They are rarely used or seen by users.

Lab 4. File Systems (II) (File Metadata)

Objectives

After completing this lab, the student should be able to:

- Understand and manipulate permissions (mode) on different Linux files
- Set the default permissions for files and directories
- Identify and handle file properties such as ownership, groups, size, and timestamps.

Permissions (Mode)

Each file has nine characters that represent the permissions on that file. Those are divided into three equal parts:
user (u)= user (owner) permissions on the file.
group (g) = permissions of members of the group name stamped on the file (except owner).
other (o)= all system users other than the group and owner.

The main three permissions that may exist on the file are ***read (r), write (w),*** and ***execute (x)***. These mean different things for files than they do for directories as follows:

Read: for files it means the user can view content of file (using vi, more, cat, …) while for directories it means the user can view content of directory (using ls).
Write: for files it means the user can modify the content of the file, but is not allowed to remove it while for directories it means the user can modify the content of the directory (i.e. can create or remove files and subdirectories).
Execute: for files it means the user can run the file (scripts or binaries) while for directories it means the user can access the directory (use cd).

To change the mode, a user may use the ***chmod*** (change mode) command. This command can specify the new permissions using a relative or absolute method.

Chmod using relative method

Using this method the user can modify the permissions on a file (or directory) relative to the already existing permission as follows:

Assume that we start with the following permissions on a file called myfile ***r_xrw_r___***

The command: ***chmod u+w,g-rw,o+ x myfile***
Will change the permissions on myfile to ***rwx____r_x***
If we continue with the command: ***chmod u=rw,g+w myfile***
The permissions will now become ***rw___w_r_x***

Check the man pages on command ***chmod*** for more examples and then do the following:

Create a directory called ***mode*** and move inside it (***mkdir mode; cd mode***).
Create a file called ***myfile*** and a directory called ***mydir*** inside directory ***mode***.
Using the ***chmod*** command with relative mode, change the permissions on both ***myfile*** and mydir as follows:

rwxr_xrw_ ***commands= _____;_____.***

r__rw____x ***commands= _____;_____.***

___rwx_wx ***commands=_____;_____.***

The second method is the absolute method which does not depend on the permissions that already exist on the file.
This method uses a binary 1 where you want a permission to be set and a binary 0 where you want it unset as follows:

The command: ***chmod 734 file*** will set the permissions on ***file*** to 111 (7) for user = rwx and 011(3) for group = _wx and 100(4) for other = r__ so the permissions on the file will be= rwx_wxr__

Using the ***chmod*** command with absolute mode, change the permissions on both ***myfile*** and ***mydir*** as follows:

rwxr_xrw_ ***commands= _____;_____.***

r__rw____x ***commands= _____;_____.***

___rwx_wx ***commands=_____;_____.***

Default Mode

The default permissions that are set on newly created files and directories are set using the ***umask*** command.

Run the command:

umask

What number did you get:_____.

This number decides the permissions set on newly created files or directories.
Read the manual page for umask (***man 2 umask***) and try to figure out what permissions would you get on files and directories after you run the command: ***umask 123***
Expected permissions on a new file= _____.
Expected permissions on a new directory= _____.

Check to see if you understood how ***umask*** works by creating a new file and a new directory and checking the set permissions. Did you get it right? _____.

What permissions would you expect after the command: ***umask 625*** is executed. Try it to see the results. Did it work?_____.

Now let us try and do the reverse:

If you want a newly created directory to have the permissions rwxr___wx what umask command would you run: _____.
Try it. Did it work? _____.
To have the following permissions on a newly created file: r__rw___w_ what umask command would you run:_____.
Try it. Did it work?_____.
What about if you wanted a newly created file to have permissions: rwxr___wx. What umask command would you run: _____.
Try it. Did it work? _____. Why? _____.

Changing Link Properties

Since we can modify the mode property of a file we can do more testing to see how links work. Go back and create two files called ***file1*** and ***file2*** and then create a hard link called ***hlink*** to ***file1*** and a symbolic link called ***slink*** to ***file2***. List the commands you used:

Now try changing the permissions on file1 to rwxrwxr__.
Command:_____

What happened to the permissions on hlink? Why?

_____.

Now change the permissions on hlink to rwx_____x.

Command:_____.

What happened to the permissions on file1? Why?

_____.

Now try changing the permissions on file2 to rw_r_xr__.

Command:_____

What happened to the permissions on slink? Why?

_____.

Now change the permissions on slink to r__rwxr_x.

Command:_____.

What happened to the permissions on file2? Why?

_____.

What happened to the permissions on slink?_____.

Ownership and Groups

The next file property is the name of the owner of the file. The owner is the only user (other than root) that can modify the properties of a file. The root is the only one that can change a file ownership using the command ***chown*** as follows:

chown newuser filename

Try changing the ownership of any of your files. Did it work?_____.

The following file property is the group name on the file. This group name may be modified by the owner if he/she is a member of the new group he/she wants to put on the file. To change the group, a user uses the command ***chgrp*** as follows:

chgrp newgroup file

Try to change a group on any of your files. What happened?_____.

Size

The next property shows the size (in bytes) of a file. Try creating a file and putting the phrase "how are you" inside then save and quit. **What is the size of the file?_____.**

Why?_____.

Change to directory /dev. Command:_____.

Check out the size property on device files. What did you find?

_____.

What are the two numbers that exist instead of the size?

_____

_____.

Go back to your home directory. Command:_____.

Go back and display the size of the symbolic link (slink) you created earlier. Can you figure out how that size was calculated?_____.

Try creating a new symbolic link and see if you are able to figure out how the size on a symbolic link is set. What did you find?

_____.

Time Stamps

A file has several time stamps. The main two are:

1- Last modification time: which is the time the file was last modified and saved. This is the default time displayed by ls –al command.
2- Last access time: which is the time the file was last accessed or viewed. What ls option is used to display that time. _____ (Check the man pages).

Check the times on file ***myfile*** and record them.
Now view the file using the ***more*** command***. What happened to the times?***

_____.

Now open the file ***myfile***, modify it and then save and quit.
What happened to the times now?

_____.

Another way to display file properties in detail is to use the ***stat*** command. Run the ***stat*** command on file ***myfile*** as follows:

stat myfile

What information can you see:

_____

_____

For more information on the output, you can read the man pages on the ***stat***.

File name

A Linux file name can be up to 255 characters long and is made of any characters. A dot has no special meaning in a file name except if it is the first character then the file is a hidden file. ***Create a hidden file called .hidden.***
Command:_____.
Try to list your files using the command ls. Can you see .hidden?
_____.
Now try to list the files using the command ls with the –a (all) option? Can you see it now?_____.

Lab5. Shell Usage and Configuration (I)

Objectives

After completing this lab, the student should be able to:

- Become familiar with common Linux shells.
- Recognize and manipulate system and user defined shell variables.
- Identify and use shell functions like command substitution, aliasing, command line editing and file name completion.

Linux Shells

A shell is a command interpreter. It is the main program used by the user to access and use the Linux operating system. When the user enters a command and hits the Return key the shell checks the command and rejects or accepts it. Accepted commands are then passed on to the Kernel part of the OS for execution and the result is displayed by the shell to the user.
There are different shells on many Linux and UNIX based systems such as:
sh (bourne shell)
csh (C shell)
ksh (korn shell)
bash (bourne again shell)
To change from one shell to another simply type the name of the shell on the command line and hit Enter.

Shells have many features and functions which allow them to perform their work. The following are some of those features or functionalities:

Variable Substitution

A shell is a program that has several variables that help the shell do its work. Many of those variables are system defined variables (usually written using upper case letters) and some may be user defined variables.

Let us consider some of the system defined variables to see how the shell uses them.

PATH variable:
Run the command:
echo PATH
What did you get?_____.
Now run the command:
echo $PATH
What did you get?_____.

To display the value of a variable you need to precede it with the ($) character.
The **PATH** variable is used by the shell to locate commands for execution. Let us see how the shell is affected by modifying that variable.
Run the following commands:

SAVEPATH=$PATH (saves the value of PATH in variable SAVEPATH)
ls Did it work? _____
PATH=/etc
ls Does it work now?_____.
Why?_____.
Restore the original value for variable PATH.
Command?_____.

Now try the command:
ls Does it work now? _____.
You can add directories to your PATH. To add the dir /etc to the end of the PATH use the command:
PATH=$PATH:/etc
Try it and then use the command:
echo $PATH
Was it added as expected? _____.

PWD and PS1 Variables:

Display the value of the PWD variable.
Command?_____.
Change your directory to /etc. Command?_____.
What is the value of PWD now?_____.
How do you think the pwd command works?_____.

Now run the following command:
PS1="hello >"
What happened to your prompt?_____.
Now run the command:
PS1='$PWD >'
What happened?_____.

There are several other variables such as HOME, PS2, SHELL, MAIL and so forth. To display the variables in your shell run the command:

set | more

List three more variables other than the ones mentioned above and their values:

1- _____.

2- _____.

3- _____.

Run the command:

env | more

Is the output the same as the set command or different? _____.
What is the difference between set and env? (hint: Check the man pages).

_____.

User-Defined Variables

Users can define their own shell variables to simplify their work or store values for later use.
Under your home directory (***cd***) create the following structure:

mkdir project
mkdir project/myfiles
touch project/myfiles/firstfile

Now create a new variable called **myprojfiles** as follows:

myprojfiles=$HOME/project/myfiles

Now you can use the new variable to manipulate your project directory. Try the following commands and write what each does:

vi $myprojfiles/firstfile

_____.

cp /etc/passwd $myprojfiles

_____.

touch good; mv $HOME/good $myprojfiles

_____.

To summarize, a shell checks a command for any variables (words starting with $) and substitutes them with their values before executing the command. E.g. in the command ***echo $PWD*** the shell first substitutes the variable **PWD** for its value and then executes the echo command on that value.

Command Substitution

Another important shell function is command substitution where the shell substitutes commands with their results before executing the main command.

Try the command:

date

What is the result?_____.

Now try to command:

echo $(date)

What is the result?_____.

The result of both commands is the same, but for different reasons. In the first case, command ***date*** is executed and the result of the command is displayed. In the second case, the shell first substitutes the result of the command ***date*** (which is indicated using the ***$(command)*** notation) and then executes the main command echo on that result. Thus, the output of the ***date*** command is used as an argument for command ***echo***.

Command substitution is very useful for saving command outputs in variables for later use.

Run the command:

grep yourusername /etc/passwd | cut -d: -f5 | cut -d_ -f1

What is the result? _____.

To get that result again you need to run the same command each time. You can save the result of that command in a variable for later use using command substitution as follows:

firstname=$(grep yourusername /etc/passwd | cut -d: -f5 | cut -d_ -f1)

Now you can use the variable **firstname** whenever you need it. This is especially useful in shell scripts. You can run the following command for example:

echo how are you doing $firstname?

The notation ***$(command)*** is common to many shells, but not all. The **csh** shell does not use that notation. There is another older notation which is understood by most if not all shells. Instead of ***$(command)*** the notation used is ***`command`*** (The single quote used here is the one below the ESC key on the keyboard).

Try the new notation to get your last name and save it in a variable called ***lastname.***

Command:_____.

Aliasing

Another function of the shell is aliasing which is basically used to give new simple names to complicated or long commands. For example:

alias dir="ls -ali"

dir

The new alias **dir** will now behave exactly as ***"ls –ali"*** when executed.

To display the aliases you already have on your system, run the command:

alias

List three aliases that you have and their values:

1- _____.

2- _____.

3- _____.

To cancel an alias, use the ***unalias*** command. For example:
unalias dir (cancels the ***dir*** alias)

Always be careful of aliases that have the same names as commonly used commands. An alias such as the following may be very dangerous. Do NOT try it.
alias ls="rm -f *"

Command Line Editing

The commands you enter on the command line are stored by the shell in a history file called ***.bash_history*** under the bash shell. To use or modify commands you executed earlier you can use the arrow keys. The up and down keys are used to get commands and the left and right arrows are used to move and modify a command if needed.

Rename (use the mv command) the file .bash_history to .save_history.
Command: _____.
Exit from the system and log back in.
Check the commands stored in .bash_history. What did you find? Why?

_____

_____.

What can you do to restore all your previous commands?

_____.

File Name Completion

Another useful shell function is file name completion where the shell completes long file names for you when you type commands as follows:
Suppose I have a file called : abcdefghijklmnopqrstuvwxyz and I need to copy it.
All I have to do is type:

cp abcESCESC newfilename

If there are no other files starting with abc then the shell will complete the long name for me. If there are other files that start with abc then the shell will display them and I need to specify the first different character and then press ESCESC for the shell to complete the name.

Making changes permanent

Many of the changes mentioned above such as creating new variables, changing existing variables, or creating aliases will disappear after exiting and logging back into the system. To make those changes permanent, they need to be added to your environment file **(.bash_profile**). Be very careful when modifying this file and always copy it first before making modifications.

Copy the file .bash_profile to file .save_bash_profile

Command:_____.

Add the following to the end of your .bash_profile file:

1. ***Add the . (current directory) to your PATH variable***
2. ***Add a variable called myproj with the name of a project directory under your home directory.***

Save the file and quit.
Exit the system and then log back in.

Check to see if the changes still exist on the system. Do they? _____.

Lab6. Shell Usage and Configuration (II)

Objectives

After completing this lab, the student should be able to:

- Understand and use shell input, output, and error redirection.
- Use pipes to join several Linux commands into single powerful commands.

I/O Redirection

Commands (and programs) usually receive input and then produce output and error. By default the input is usually received from the keyboard and the output and error are usually both directed to the screen. Linux shells allow us to change those defaults and redirect input, output, and errors.

Input Redirection

To understand input redirection, let us first use the ***mail*** command. The mail command is the default command used to send and receive mail on most UNIX based systems. To send email to another user simply use the command:

mail username (username@system if on another system)

You can try sending yourself an email by typing:

mail yourusername
subject:hello
This is my mail message
Goodbye
.
cc:

As you can see the mail asks you for a subject (title of message) and you end the mail by typing a dot **(.)** by itself on a line and then pressing enter.

To read your email, you can simply type:

mail

You will get the & sign. Type ? for help on how to use (read/delete/save/reply/forward/…) the mail program. To quit just type **q** and Enter.

The input for the mail command was received from the keyboard (default). You can redirect the input such that it is received from a file. To do that use the (<) character as follows:

Create a file called message and type the following two lines inside:

This is my message file
Goodbye

Then save and quit

Now run the following command:

mail -s hello yourusername < message

The input in this case was redirected to come from file ***message*** instead of the keyboard.

Another example is the ***tr*** (translate) command. This is a useful command used to change input characters and may be used to encrypt characters.

Run the command

tr "a-z" "A-Z"
how are you

The result is "HOW ARE YOU". As you can see the input was received from the keyboard. You may redirect the input to come from the file message you created earlier as follows:

tr "a-z" "A-Z" < message

What was the output?

_____

_____

You can append the redirected input using the here text (<<). Run the following command:

tr "a-z" "A-Z" << !

- ***hello***
- ***how are you***
- ***hope well***
- ***bye***
- ***!***
-

What did you get as output?

Output Redirection

The output of commands is sent to the screen by default. You may redirect the output by using the (>) character. Run the command:

ls –al

The output will be shown on the screen.
Now run the command:

ls –al > lsfile

No output will be displayed on the screen. View the file **lsfile** using the ***more*** command. It should contain the output of the "***ls -al***" command.

Using the (>) character will create a new file or overwrite an existing file.
To append the output to a file, you can use the (>>) character as follows:

ls -al >> lsfile
who >> lsfile
echo hi >> lsfile

One of the main Linux philosophies is that everything is treated as a file including hardware devices. To interact with hardware devices, Linux interacts with device files which represent those hardware devices. This means that if we are able to redirect input or output from/to files then we actually do the same with devices. We can try this with device files that represent our terminals (screens).

Open two terminals (if using telnet then do two telnet connections).
Run the command:

who

Record the ***pts*** numbers (you should have two, one form each terminal).
Assume the terminal you are working on has ***pts/4*** and the other terminal has ***pts/5*** (you need to use your numbers when testing).
Type the following command:

echo hello

This will display the word hello on your current terminal (i.e. pts/4) which is the default.
Now type the following command:

echo hello > /dev/pts/5

What happened? Explain.

_____

_____

Error Redirection

Command output is sometimes mixed up with command errors since they are both sent to the screen by default. Run the following command:

cp
What did you get displayed?
_____.

Is that output or error? _____.
Now run the command:
cp > cpfile
What happened? _____.
Since the same message got displayed on the screen and was not sent to file ***cpfile*** then it must not be output. It is error.
To understand how to redirect errors, we should learn about file descriptors. There are three file descriptors used by programs to specify input, output, and error.
Standard input has file descriptor 0
Standard output has file descriptor 1
Standard error has file descriptor 2

There is no need to use the file descriptors 0 and 1 when redirecting input and output respectively since they use two different characters namely < and >.
To redirect error we need to use the (>) character so to distinguish it from redirecting error, we must specify the file descriptor before the > character as follows:
cp 2> cpfile

What happened now?_____.

Check the contents of file cpfile. What did you find?

_____.

Redirecting output and error to different places may be very useful especially when dealing with commands that produce both at the same time. Try the following command:

find / -name passwd -print
What did you get? Was that output or error?_____.

Now run the command as follows:
find / -name passwd -print 2> errors
What did you get now? _____.

Check file errors content.
Now run the command as follows:
find / -name passwd -print > output 2> error
What happened?_____.

Check both files output and error.

To append errors use (**2>>**).

Pipes

One of the main Linux philosophies is to have commands where each does one thing very well. For example, the ls command has so many options to display file information in so many different ways. Another philosophy that complements that is the ability to join different commands together in a chain to produce more powerful commands. This is usually done using pipes.

Run the following command:

cat /etc/passwd | grep yourusername | cut -d: -f5 | cut -d_ -f1

What did you get? _____.

This command is made up of four different commands joined together using pipes (|). Pipes usually work with commands we call **filters**. They take input and filter it to produce a certain output. They usually do not change the original input source.
This is how the above command works:

"***cat /etc/passwd***" produces the passwd file (many lines) as output.
The **passwd** file is passed as input to the "***grep yourusername***" command which in turn filters that into a single line that contains your username. This line is then passed to the command "***cut -d: -f5***" which filters it to one -field (field five) (-f5) based on dividing fields by delimiter : (-d:). This output is then passed as input to the next cut command "***cut -d_ -f1***" which filters it to get the first field (your first name) by cutting based on delimiter underscore (-d_). The output (your first name) is then displayed on the screen.

What command would you use to get your group number from /etc/passwd:

_____.

What command would you use to get your login time from the who command?
(Hint: use the tr command with the squeeze option)

_____.

What command would you use to get the default group name for any given user?

_____.

Try the following command:

find / -name passwd -print | more

What happened? Why is the result of the command not filtered by more?

_____

_____.

How can we fix this?

_____

_____.

Lab7. Job and Process Management

Objectives

After completing this lab, the student should be able to:
- Manage several jobs running in the background.
- Understand how processes are created using the fork and exec steps.
- Control the priority of newly created processes using the nice command.
- Identify and use signals for manipulating processes.

Job Control

Sometimes we need to execute more than one job on the same terminal, but we are forced to wait until one command is done executing and getting the shell prompt back before we can execute the next command. This is especially a problem if the one of the jobs we are executing takes a long time such as a backup job. To get around this, Linux allows us to run several jobs at the same time in the background. This is called job control.
To be able to understand job control, we need to create and use a command that will take a long time. To do this, we do the following steps:

1- Create a new file called forever using vi as follows:

```
vi  forever
        while   true
        do
        echo  running  >  myfile
        done
:wq
```

This is basically a script file with an infinite loop.

2- Now we have to make sure that our PATH variable includes the current directory (.). This step is important for the shell to locate our newly created command forever. This is done as follows:

```
PATH=$PATH:.
```

3- The third step is adding the execute (x) permission to the command to make it executable. This is done by adding x to all parts of the mode as follows:

```
chmod   +x   forever
```

Now we have a command called ***forever*** that runs for a long time and that can be used to understand job control.
To run a job in the background, we follow the command with an ampersand (**&**). In our case we are going to run three forever jobs in the background as follows:

```
forever&
 [1][2000]
```

forever&
[2][2500]
forever&
[3][2503]

Each time we run a job in the background the system displays two numbers. The first is the job id number and the second is the job process number. These numbers are important to be able to reference the job later on for manipulation.

We can display our background job by using the command:

jobs

This will display an output similar to the following:

[1] Running forever
[2] - Running forever
[3] + Running forever

The number is the job id number. The plus and minus signs reference the last and the one before last jobs. The status of all jobs is running. The last column is the name of the command used to create the job.

We can manipulate the jobs in several ways, as follows:
To get a job back to the foreground we use the ***fg*** (foreground) command followed by the job id number. E.g. to get job 2 to the foreground, we run the command:

fg %2

This brings the job to the foreground. To send the job to the background, we press ***ctrl-z***. The job is moved back to the background.
Run the following command:

jobs

What do you notice different about job # 2?

_____.

To resume a stopped or suspended job, we use the ***bg*** (background) command followed by the job id number. To resume job 2 (change its status to running) we use the command:

bg %2

Run the command:

jobs

What is the status of job # 2 now? _____.

To terminate a job we use the ***kill*** command followed by the job id number. E.g. to kill job 3 we issue the following command:

kill %3

If we type the command: ***jobs*** quickly enough we will see the status of job 3 changing to Terminated and if we check again it will disappear.

Do the following:
kill all remaining jobs such that none are in the background.
Write the sequence of commands needed to have the following output displayed when the command "jobs" is issued:

[1] Stopped forever
[2] - Terminated forever
[3] + Running forever

Commands:

Process Control

A process is simply a program in execution. Each command we run results in one or more processes. There are several processes running in the background that allow us to use the system and provide us with different services. Interacting and manipulating processes is called process control.

When a command is run, a duplicate copy of the parent process is created using the fork function. This copy is similar to the original except for its process id number (pid). After that the system executes the command using the exec step which basically loads the new command on top of the copy created as follows:

When we run the command ls under the bash shell, a copy of bash is created and which is replaced by ls.

pid (100) ***Fork →*** ***pid (200)*** ***Exec →*** ***pid (200)***

bash (local variable)	bash (local variable)	ls (local variable)
bash code	bash code	ls code
bash (env. variable)	bash (env. variable)	bash (env. variable)

Notice that the environment variables are passed from the parent process (**bash**) to the child process **(ls**).

Let us now run through to see the some details on what happens above.
To view process information, we can use the ps (process status) command. To see our running processes we use ps with the –f option as follows:

ps -f

Describe the output?

_____.

Let us create two variables called var1 and var2 respectively.

var1=first
var2=second

When new variables are created they are defined as local variables. To change a variable from local to environment, we export it (use the export command). Let us make var2 an environment variable as follows:

export var2

The ***set*** command is used to display both local and environment variables. The command ***env*** is used to check the environment variables only. Let us check for **var1** and **var2** in our main process (bash shell):
Run the command:

set | grep var

Which of the two variables (var1 and var2) do you see in the output? Why?

_____.

Now run the command:

env | grep var

Which do you see now? Why?

_____.

Now run a child processes (ksh) as follows:

ksh

Run the command:

ps -f

What is the output now?

_____.

Notice the numbers pid (process id) and ppid (parent process id). Those should tell you that bash is the parent process and ksh is the child process.
You are now in the child process. Let us check for the variables **var1** and **var2** in the child process (ksh).
Run the command:

set | grep var

Which of the two variables (var1 and var2) do you see in the output? Why?

_____.

Now run the command:

env | grep var

Which do you see now? Why?

_____.

This shows that only environment variables are passed from parent processes to child processes.

As shown above any created process goes through the **fork** and **exec** steps explained above. We can use the ***exec*** command to skip the **fork** step and just do the exec step and see what happens, as follows:
Run the command:

ps –f

You should have three processes (bash, ksh, and ps –f). ps –f does not exist anymore. Now register the pid number for the ksh process. Now instead of running the "ps –f" command as before, run is as follows:

exec ps –f

What processes do you see now? What happened to ksh (hint: note the pid number for the ps –f process)

_____

_____

What would you expect to happen if you run the command "exec ps –f" again?

_____.

Try it. What happened?

_____.

This shows that processes do go through both the fork and the exec steps, otherwise a new child process will take over its parent process and destroy it.

Nice command

Users may decrease the priority of their processes (especially those that take a long time and are not of high priority such as backups) to allow other users to run their processes at a higher priority. When they do that, they are nice and to do that they use the nice command. The only user that can both decrease and increase the priority of his/her processes is the root (system administrator). Let us see how the nice command is used.
Run the command:

ps -l

Note the two new columns displayed namely:

PRI (which refers to the priority of the process)

NI (which refers to the nice value of the process)

Now run the above command as follows:

nice -6 ps -l

Notice what happened to the **PRI** and **NI** values for process "**ps -l**". They increased. Increasing the priority number actually makes the priority for that process less.
Now try to run the command:

nice --8 ps –l (--8 = two dashes then 8)

What happened? Why?

_____.

Signals

Users can control their processes through sending signals using the "kill" command. There are many signals that may be sent to a process. To get a list you may use the following command:

man 7 signal

There are three interesting signals that stand out. Those are namely **TERM** (also called SIGTERM) which has the number 15, **HUP** (also called SIGHUP) which has the number 1, and **KILL** (also called SIGKILL) which has the number 9. The default signal is the TERM signal.
The TERM signal tries to terminate signals cleanly and may be blocked by processes such as shells. The HUP signal is used to restart a process to have it upload any changes in its configuration files. The KILL signal is used to kill a process uncleanly and cannot be blocked. Let us try the TERM and KILL signals:
Run the command we created in the beginning of this lab (***forever***) in the background and note the process id number given (let us assume it is **1234**). Check to see that the process is running in the background (use the ***jobs*** command).
Try the following command:

kill 1234 (use the number shown for your process)

Now recheck if the process is running with the ***jobs*** command. What did you find?

_____.

Now repeat the same steps (i.e. create the ***forever*** job in the background and check its PID (we are assuming its 1234, but it could be anything)).

For ***each time*** you create the ***forever*** job try killing it with one of the following commands:

kill -15 1234 (specify the correct PID, we are assuming its 1234)
kill -TERM 1234
kill -SIGTERM 1234

What did you notice about each of the three commands above?

_____.

Open two terminals (if you are using telnet then open two telnet connections)
Use the ***ps*** command to determine the process id number of the terminal you are not using, as follows:

ps –f

What is the pid number for the bash process running on the pts number different from the pts number that your ps –f process is running. That is the pid you need. Now try running the following command:

kill pidofbash (or kill -15 pidbash)

What happened? Why?

_____.

Now try the following kill command:

Kill -9 pidofbash (-9 is equivalent to –KILL or -SIGKILL)

Now what happened?

_____.

Lab8. Text Processing Tools and Regular Expressions

Objectives

After completing this lab, the student should be able to:

- Identify and use filters as valuable text processing tools.
- Use simple regular expressions to make text processing more efficient.

Text Processing using Filters

In the pipes lab, we mentioned a group of commands called filters. These are basically commands that take some input and then filter it to produce the requested output without changing the original source of input. In this lab we will practice how to use some filters as useful tools for text processing.

The filters we will use are (***use the manual pages (man command) to get more information on those filters and their available options***):

head and tail: used to display lines from the beginning or end of a given input respectively.
cat: used to view or concatenate files.
grep: used to extract certain rows (lines) from a given input. We will concentrate on the options -i, -l (EL), -v.
cut: used to extract certain columns from a given input. We will use the options -d, -f, and –c.
tr: translates (changes) a given input to a specified output
wc: used to count lines, words, or characters in a given input.
sort: used for sorting a given input. We will present the options -i, -o, -u, -n, -k, and –t.
sed: used for stream editing (changing parts of an input to a specified output)

Create the following file called ***students*** using the ***vi*** command and then save and quit:

ah6:506:Ahmad_Hamdan
sh5:345:Suha_HAMDAN
rd7:427:Ribhi_ahmad
hr4:234:hamdan_ribhi
ad6:386:Arwa_Ahmad
ad5:285:ahmadi_Ahmad

Each line of the file ***students*** contain three fields: ***student user name*** (e.g. ah6), ***student id number*** (e.g. 506), and ***student full name*** (first and last names separated by an underscore e.g. Ahmad_Hamdan), respectively.

Let us now try our filters on the file ***students***. Write down what each of the following commands does. Make sure you understand why each command behaves that way.

head -2 students

_____.

tail -3 students

_____.

What command would you use to get the fourth line only from file students (hint: mix head and tail with pipes):

_____.

cat students

_____.

grep ahmad students

_____.

Join both cat and grep with pipes to get the same result as the previous grep command:

_____.

grep -i Ahmad students

_____.

grep -l Ribhi *

_____
.

grep -v Ribhi students

_____.

grep -iv hamdan students

_____.

cut -d: -f2 students

_____.

What command would you use to get the last names for all users in file students:

_____.

What command would you use to get the first names of all users with last name hamdan (all cases):

_____.

cut -c2,3 students

_____.

What command would you use to get the middle digit in the id numbers for all users with last name hamdan:

_____.

tr "a-z" "A-Z" < students (Describe output)

_____.

What command would you use to get the first names (all in lower case) of all users that have the word ahmad (all cases) as part of their full name:

_____.

wc -l students

_____.

head -1 students | cut -d: -f3 | cut -d_ -f2 | wc -c

_____.

What command would you use to count the number of files in your home directory? Hint: use the ls and wc commands:

_____.

sort students (Describe output)

_____.

sort -o result students (What happened?)

_____.

sort -k2 -t: -n students (Describe output)

_____.

What command would you use to list all the first names of users in file students sorted based on lower case letters and without repetition (hint: check the –f and –u options for sort):

_____.

sed 's/ahmad/damha/' students

_____.

What is different when we run the same command with the i (ignore case) option, as follows:

sed 's/ahmad/damha/i' students

_____.

What is different when we run the same command with the g (global) option, as follows:

sed 's/ahmad/damha/ig' students

_____.

Regular Expressions

Some of the filters mentioned above such as ***grep*** and ***sed*** may use what we call regular expressions to be more powerful and precise. To get more information about the power and extent of regular expressions, you can read the man pages using the command:

man regex

We will just give a very basic introduction (a simple taste) to how regex may be used with some filters. The following are some common regular expressions:

pattern$: applied to a pattern if it is at the end of a given line.
^pattern: applied to a pattern if it is at the beginning of a given line.
[abc]: means a or b or c
[^abc]: means all characters except a, b, or c.

Let us try some commands with ***regex***. Write down what each command does:

grep -i 'hamdan$' students

_____.

cut -d: -f3 students | grep -i '^ahmad'

_____.

cut -d: -f3 students | cut -d_ -f1 | grep -i '^ahmad$'

_____.

cut -d: -f1 students | grep a[dh][^6]

_____.

cut -d: -f3 students | sed 's/^ahmad/sameer/ig'

_____.

sed 's/ahmad$/Sameer/i' students

_____.

Using what you learned above, write the commands that are needed to extract and display the following information from the /etc/passwd file:

Display the first names of all users whose last names end with the letter 'n' or 'm':

_____

Display the last names of all the users sorted by their user id numbers (ascending order):

_____

List the login names for all users with the bash as their default shell.

Display the default shell used by user root.

Display the number of files in directory /etc that end with the word .conf

---

Lab9. Shell Scripts (I) –Introduction

Objectives

After completing this lab, the student should be able to:

- Create and execute simple shell scripts.
- Use positional parameters and shifting to pass command line arguments to scripts.

Introduction to Shell Scripts

One of the most powerful tools in Linux is the ability to group several commands into scripts in order to automate manual tasks. Scripts are also used as configuration and setup files in different areas of the Linux File System. Let us start by writing and executing our first simple script. To create and setup a script you need to do the following:

1- Using the vi editor open a new file and write your script as follows:

vi myfirst

echo this is my first Linux script
echo I like it
echo bye

:wq (save and quit)

2- You now need to add the x (execute) permission to your script to run it. This is done only the first time you create your script. If you try to run your script and get the error "permission denied" then the reason would most likely be that you did not do this step. To add the x permission run the following command:

chmod +x myfirst

3- You must make sure that you have the following line added to the end of your environment setup file (.bash_profile):

PATH=$PATH:.

To make this step take effect you either exit the system and log back in or you run the following command:

. .bash_profile

This adds the current directory (.) to your search path which would make the shell search for your script in the current directory. If you try to run your script and get the error "permission denied" then the reason would most likely be that you are missing this step.

This third step is only done once for all your future script to run without trouble.

Now you are ready to run your first script by typing its name on the command line as follows:

myfirst

What was the result of running the script?

_____.

As you can see the echo command is used to print output messages from your script to the output device (screen or file).

Let us now see how to create another script that has both input and output as follows:

```
vi   greetings
        echo What is your name
        read  name
        echo hello  $name
:wq

chmod   +x   greetings

greetings
```

What do you think is the purpose of the read command?

_____.

Notice that when you read a value in your variable you just put the name of the variable (e..g name) while when you print the value, you need to put the $ sign at the beginning (e.g. $name).

By default, shell scripts treat all variables as strings.

Now let us write our own script for deleting a file:

```
vi   delete
        echo Enter file name:
        read  filename
        rm   $filename
        echo   File $filename has been deleted
:wq
```

Now run your script. Did you forget to add the (x) permission?_____.

Now it is your turn to write a complete script and run it.

Write a script called copy that asks the user to enter a source filename and a destination filename and then copies the source to the destination. Your script should work as follows:

```
copy
Enter source file name:
one
Enter destination file name:
two
File one is copied to file two
```

Your script:

Try to run your copy script. Did it work? _____.

You probably realize that programs like delete and copy would behave more like similar commands if they took their input from the command line instead of asking the user to enter those after running the program. To do this we need to use positional parameters. Let us write a simple script to understand how those are used:

```
vi  params
        echo  $1
        echo  $3  $2
        echo  $#
        echo $0
        echo $5
        echo  $*
:wq
```

Now run the script params as follows:

params one two 3 four 5 6 bye

What was the output?

Now what do you think are the values of each of the following variables:

$1: _____

$3:_____

$*:_____

$#:_____

$0: _____

Now that you understand how positional parameters work, ***rewrite both the delete and copy scripts above to run as follows:***

delete thefile
thefile has been deleted

Answer:

copy file1 file2
File file1 has been copied to file2

Answer:

Now try your new delete and copy scripts? Did they work? _____.

Notice that since you already had the ***x*** permission on the previous scripts (***delete*** and ***copy***), you did not have to do that step again in order to run them.

Practice:
Write a script called whoisuser that takes the login name of a user as a parameter and then uses the /etc/passwd file to get and print the full name of that user as follows:

whoisuser u1122334
u1122334 = Ahmad Hamdan
hint: use variable and command substitution.
Answer:

Shifting parameters

To shift script command line parameters to the left, we use the shift command as follows:
shift number of shifts (e.g. shift 2 for 2 shifts)
shift (no number shifts one)

To understand how shift works, let us rewrite and run the params script above as follows:

```
vi params
        echo $1
        shift 2
        echo $2 $3
        echo $#
        shift
        echo $0
        shift 3
        echo $1
        echo $*
:wq
```

Now run the script as follows and notice the effects of shifting:

params one two three 4 5 6 seven 8 9 ten bye

Which parameter is not effected by the shift command? _____.

Comments

You can add comments to your scripts by using the # sign followed by the comment anywhere in your script. Lines that start with (#) are interpreted as comments except in one case where shells have **(#!)** followed by the name of a shell as the first line of a script. In that case that line is interpreted as the name of the shell to be used for executing that script.

Example:
If your script starts with the line:
#!/bin/bash
Then the script is meant to be executed using the **/bin/bash** shell.

Check out the following system scripts:

more /etc/rc.sysinit
more /etc/rc.local

What is the first line in those files (scripts)?

_____

What is the difference between the first line and the few lines that come after it?

_____.

Lab10. Shell Scripts (II)- Programming
(Selection Constructs)

Objectives

After completing this lab, the student should be able to:
- Include programming selection constructs in shell scripts.
- Use the if/else statement to manipulate integer and string values as well as file properties.
- Apply the case statement programming construct for efficient selections as well as creating menus.

Script Selection Constructs

In the previous lab, you have noticed that in our scripts we made several assumptions that files and user names already existed and that we have permissions to remove, copy, or view files and that the correct number of command line arguments where given to our scripts. This is not always the case. Our scripts should be able to check for values and properties before executing what is required. To do this, we need to use selection statements (the If and Case statements).

Unix commands return a value (success = zero and failure or error = non-zero) to the shell. This value is stored in the variable (?) as follows:

Run the command:

ls –al

Now run the command:

echo $?

What result did you get? _____ Why? _____.

Now run the command:

cp

followed by the command:

echo $?

What result did you get? _____ Why? _____.

The value returned by Linux commands may be checked in scripts using the if/else structure.

Write the following script:

```
vi checkcommand
        if $1 > out 2> err
        then
                echo Command $1 succeeded
        else
```

```
                echo  Command $1 failed
        fi
:wq
```

Now run the script as follows:

checkcommand date

What result did you get? _____ Why? _____.

Now run the command:

checkcommand mv

What result did you get? _____ Why? _____.

This is one way to use the if/else structure. Still, many scripts do not check commands, but rather check for variable values, file properties, and number of arguments. To do that we need to use one of two syntaxes:

if test condition (e.g. if test $# -eq 2)
or
if [condition] (e.g. if [$# -eq 2])

The general syntax for the if/else statement is as follows:

```
if  condition
then
        statements
elif  condition
then
        statements
else
        statements
fi
```

To compare integer values, we use the following relational operators:

-lt (less than), -gt (greater than) -eq (equal)
-le (less than or equal) -ge (greater than or equal), -ne (not equal).

Let us rewrite the delete script we wrote in the previous lab to check for the correct number of arguments as follows:

```
vi  delete
        if [  $#  -eq  1  ]
        then
          rm  $1
          echo  $1 is deleted
          exit  0                # This line returns 0 from the script (success)
        else
```

```
        echo   Usage: delete filename
        exit   1
    fi
:wq
```

Now try the above script as follows:

delete myfile (assuming myfile exists and is a regular file)

Then run the command:

echo $?

Did it work?_____.

What is the value of variable (?) ?_____

Now try it as follows:

delete

Then run the command:

echo $?

What happened? _____ Why?_____.

What is the value of variable (?) ?_____

To check file values we use the following operators:

-f filename (to check if file exists and is of type file)

-d filename (to check if directory exists and is of type directory)

-x,-r,-w (to check if a user has execute, read, or write permissions on a file)

Let us rewrite our delete script to include those:

```
vi delete
    if [  $#  -ne   1  ]
    then
        echo   Usage:  delete filename
        exit   1
    else
        if  [  -f  $1  ]       #  $1 exists and is a file name
        then
            rm  $1
            echo   File   $1 is deleted
            exit   0
        elif  [  -d  $1  ]
        then
            rm   -r    $1     #  $1 exists and is a directory
            echo  Directory   $1 is deleted
            exit 0
        else
            echo  $1: No such file or directory
```

```
                    exit  2
                fi
            fi
        :wq
```

Now create a file and a directory using the following commands:

touch myfile; mkdir mydir

No try the updated delete script in the following ways:

delete

What happened? _____.

delete myfile (myfile exists and is a file)

What happened?_____.

delete mydir (mydir exists and is a directory)

What happened?_____.

delete wrong (wrong does not exist)

What happened?_____.

Now rewrite the copy script to act as follows:

```
copy
        Usage:  copy  src  dest
copy  myfile  newfile
        File myfile is copied to file newfile
copy  mydir  newdir
        Directory  mydir  is copied to newdir
copy  wrong  good
        wrong:  No such file or directory
```

Try the new copy script and make sure it works as above?

Did it work correctly? _____.

Sometimes our scripts need to check string values. To do that we need to use the following operators:

= (equal), != (not equal) -n (none null string) -z (zero string (null))

Let us try some of those. let us write a script to check the value of the name entered by the user:

vi checkname

```
if [ $# -ne 1 ]
then
        echo Usage: checkname name
        exit 1
else
        if [ "$1" = "ahmad" ]
        then
                echo $1: Hello
                exit 0
        else
                echo $1: Goodbye
                exit 0
        fi
fi
:wq
```

try it as follows:

checkname ahmad

What happened?_____.

checkname suha

What happened?_____.

checkname

What happened?_____.

Write a script called checkusername which works as follows:

checkusername
No names were entered
checkusername u1112233
u1112233 = Ahmad Hamdan
checkusername u11
u11 = No such user name
checkusername bash
bash = No such user name

Script:

Now try it with similar cases to those written above.
What happened?_____.

Case Statement

We can also use a case statement (similar to switch in c) to check for values. The syntax is as follows:

```
case    value   in
pattern1)  statements
                ;;      # ;;  is the break statement
pattern2)  statements
                ;;
*)   statements          #   * stands for anything which is the default case
esac
```

The patterns may be strings or parts of strings. Those can include the * wild card, the (|) OR operator, as well as ranges (e.g [0-9] or [a-f]) as follows:

s* | S* | good)
: means any pattern that starts with s or S or the word good.

[A-Z]*[0-5])
: means any pattern with any size that starts with a capital letter and ends with a number between 0 and 5

[a-z][0-9][0-9][0-9] | [0-9][A-Z][A-Z][A-Z][a-f])
: means the accepted pattern must consist of exactly four characters the first is a small letter and the next three are numbers or the pattern must be exactly five characters with the first being a number followed by three capital letters and then one small letter between a and f.

Write a script that uses case statement with patterns similar to the above.
Did they work?_____.

Case statements are usually used for handling menus and menu options. Let us try a simple example that uses a menu to call different scripts (modular programming):

Create three different scripts called ***script1***, ***script2***, and ***script3*** respectively. In each script put one line to display which script you're in (e.g in script1 put the line "echo this is script 1").
Now create a script called ***mainscript*** that displays the following menu:

Please select your choice (1-4):
1 - Run script1
2- Run script2
3- Run script3
4- Exit main script

Using a case statement, have your script run the suitable script (1,2, or 3) or exit based on the user's selection.

vi mainscript

Now try ***mainscrip***t. ***Did it work?_____.***

Lab11. Shell Scripts (III)- Programming
(Looping Constructs)

Objectives

After completing this lab, the student should be able to:
- Include programming looping constructs in shell scripts.
- Understand and use the while, until, and for loops constructs.
- Learn how to make for loops more efficient by using command outputs as lists.

Shell Script Loops

In order to create useful scripts that can automate real jobs, we need to learn how to include loops in those scripts. There are different loop constructs that may be used in shell scripts which include:

while loops
until loops
for loops

Each has its own useful features that make it useful in certain situations.

While Loop

Let us first start with the while loop. The structure of the while loop is as follows:

```
while   condition
do
        statement(s)
done
```

example:

```
vi listarguments
        while   [ $# -ne   0 ]
        do
                echo  $1
                shift
        done
:wq
```

Run the above script as follows:

listarguments a hello 7 x

Check the output.

Note: Rules that apply to conditions used in selection statements are exactly the same as those that apply to conditions in loop statements.

After making sure you understand the above example do the following:

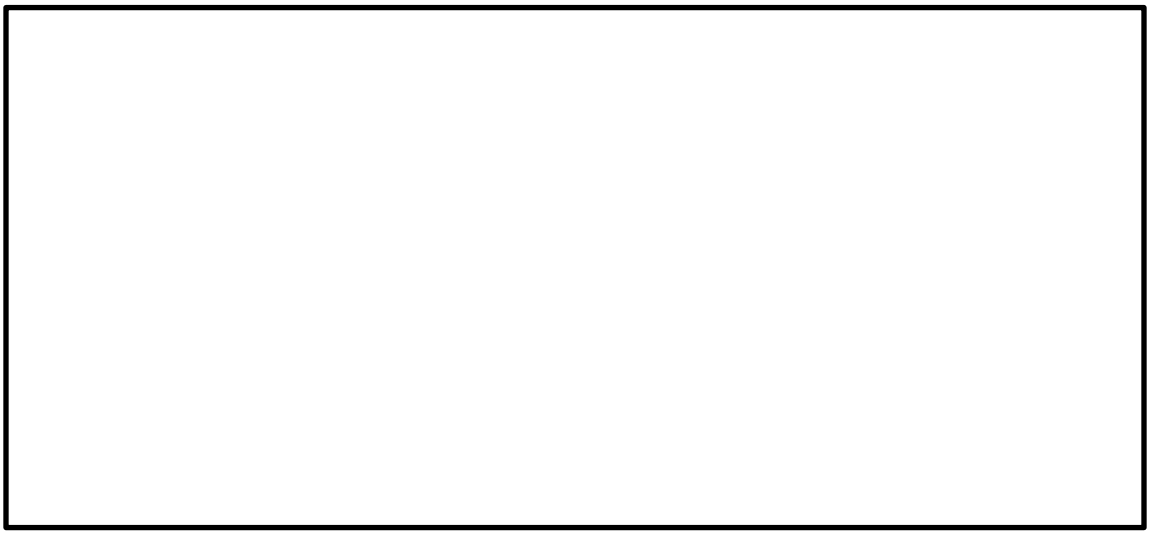
Rewrite the delete script we wrote in the last lab such that it works as follows:

delete file1 wrong dir1 file2

File file1 is deleted
wrong: No such file or directory
Directory dir1 is deleted
File file2 is deleted

Answer:

vi delete



:wq

Now try it with existing file and directory names as arguments. ***Does it work? _____.***

Sometimes the loop will stop executing based on the user input, as follows:

```
vi findahmad
    echo Enter name
    read name
    while [ "$name" != "ahmad" ]
    do
        echo $name: wrong name. Try again.
        echo Enter name
        read name
    done
:wq
```

Now modify the ***checkusername*** script from the previous lab such that it is called ***checkusernames*** instead and works as follows:

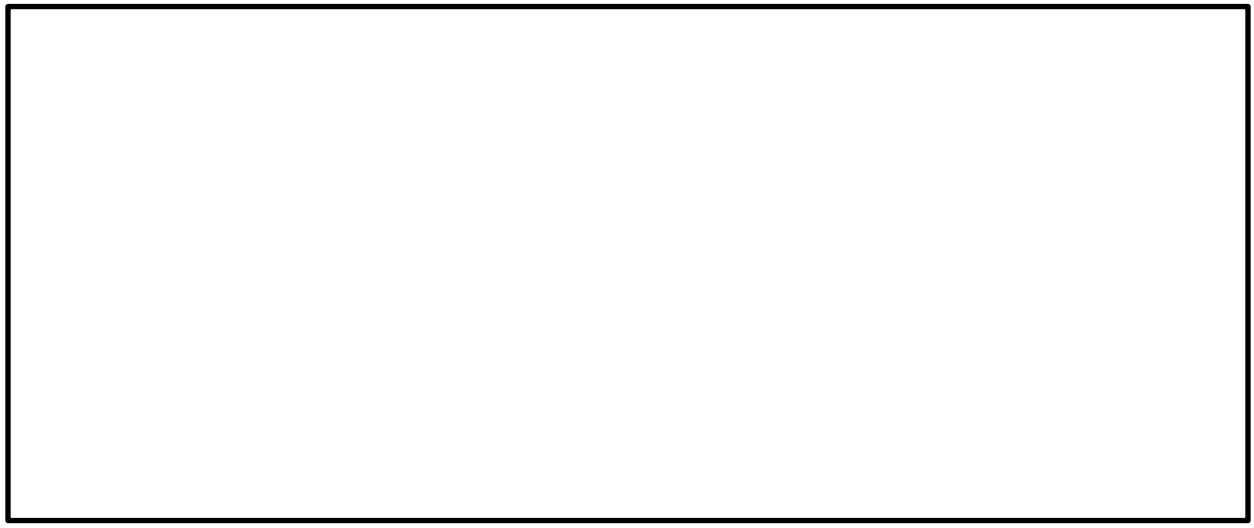
checkusernames

Enter user name to check or word "enough" to stop
u1112345
Enter user name to check or word "enough" to stop
u11
Enter user name to check or word "enough" to stop
u1123456
Enter user name to check or word "enough" to stop
enough

u1112345 = Salem Hamdi
u11 = No such user name
u1123456 = Sabah Khaled

Answer:

vi checkusernames



:wq

Break and Continue Statements

The programmer can use ***break*** and ***continue*** statements inside shell script loops which mean the same as they do in the C language:
break - exit the loop immediately.
continue – stop running the current cycle but go back and check the condition.

In addition they can use ***break*** and ***continu***e followed by a number to specify how many loop levels they want them to work for. For example:
break 2
Will exit out of two nested loops if they exist.

until loop

The until loop is similar to the while loop, but stops when the condition becomes true.

```
until  false
do
        statements
done
```

Modify the above two programs such that they use the until construct instead of the while construct and try them out. Did they work? _____.

For loop

In shell scripts, the for loop is very powerful and useful. The general structure of the for loop is as follows:

```
for   item   in  list of items
do
        statement(s)
done
```

What makes a for loop powerful is the different ways a list of items may be specified. Let us start with a simple example:

```
vi  names
        for   name  in  ahmad  hamdan  subha  khaled
        do
                echo  $name
        done
:wq
```

Run the script names. It should display the names given in the list.

Now change the first line in script names to the following:

for name in $* (remember that $* holds all the arguments as a list)

and run the modified script as follows:

```
names   ahmad  subha  khaled
```

What happened? _____.

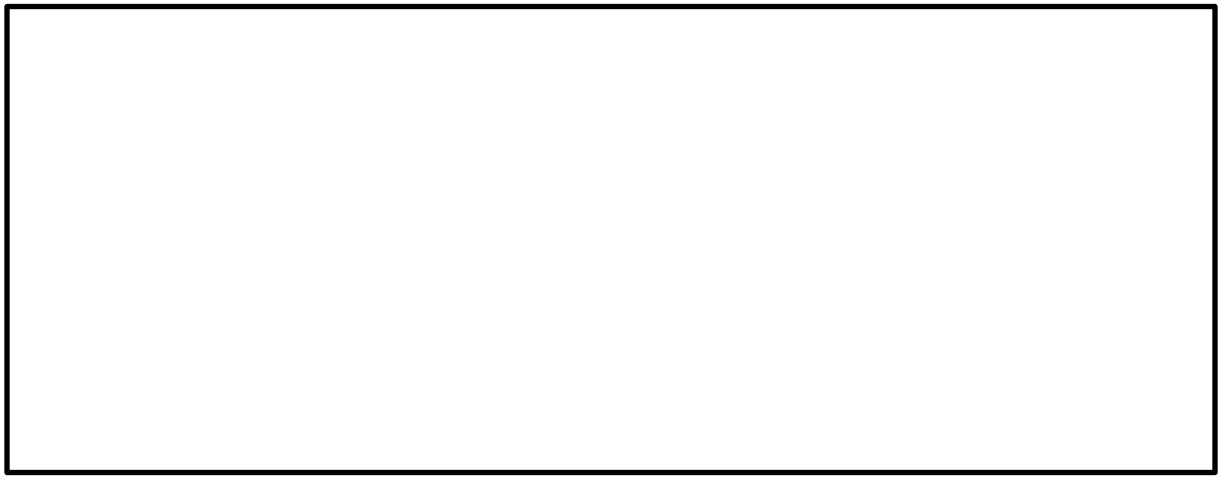
Rewrite the delete script we wrote at the beginning of this lab such that it uses a for loop instead of a while loop. Did it work? _____.

The best feature about the for loop is that we can treat the output of a command as a list of items as follows:

```
vi lines
        for line in $(cat /etc/passwd)
        do
                echo $line
        done
```

Using a for loop, write a script called ***comp311*** that lists the full names of all the users that are registered in the comp311 course.
Answer:

Now rewrite the script ***comp311*** such that it will display only the names of the users that are currently logged in to the system. (hint: use the output of the who command)
Answer:



The for loop can also be applied to a directory of files as follows:

```
vi myfiles
for file in *
do
        echo $file
done
```

Write a script called ***filetypes*** that uses a for loop to type the name and type (file, dir, or unknown) for each file in a given directory as follows:
Assume that I use the script in the following way:

filetypes /etc

then the script should display the names of all the files under directory /etc and the type of each of those files:

Answer:

The which command displays the directory in the PATH that contains the command. Try it as follows:

which ls

What is the result? _____ .

Write a script called ***mywhich*** that simulates the which command. You are not allowed to use the which command in your script. (***hint: use the for loop and the sed command***).

Answer:

Lab12. Security and Networking Concepts

Objectives

After completing this lab, the student should be able to:

- Understand through example the importance and usage of set user id (suid) and set group id (permissions) in Linux.
- Set and modify suid and sgid values on Linux files.
- Identify and learn some Linux networking tool basics.

Suid and Sgid

Linux systems are very secure and have multiple levels of security that takes volumes to discuss. We have already talked about a part of one of those security levels which is file security where we explained the permissions (mode) and how they are used to control who can access and use files and directories. The permissions we talked about were the read (r), write (w), and execute (x). In this lab we will present a less obvious, but very powerful permission called the setuid (set user id) and setgid (set group id) permission usually referenced with an (s) permission.

Set User Id (suid) Permission

To understand how the suid permission is used let's take an example based on the passwd command which we use to change our passwords.

run the command

which passwd

This gives you the absolute path name of the passwd command (usually /usr/sbin/passwd).

now run the command ***ls –al*** on that file as follows:

ls -al /usr/bin/passwd (or whatever the which command produced)

Notice the permissions on that file.

What are they? **_____.**

The (s) on the user part of the mode is the suid. This (s) is very important and without it a user will not be able to change his/her password.

When a command such as passwd is executed, a process is created as explained in lab 6. That process has many properties. Four of those are:

real uid (real user id)
real gid (real group id)
effective uid (effective user id)
effective gid (effective group id)

The real uid and gid are the same as the username and group of the user executing that command. The effective uid is the same as the real uid and the effective gid is the same as the real gid except when there is an (s) permission. In this case the effective uid will be same as the owner of the file and the effective gid will be same as the group name on the file.

The process resulting from running the passwd command has an effective uid as root (owner of the file passwd) which is why this command is able to open and modify files (e.g. /etc/passwd and /etc/shadow files) which the user running the command is not allowed to.
This gives great flexibility by giving regular users the ability to access files through running commands which they cannot access normally.

Set Group id (sgid) Permission

Let us now see how the same idea is applied to the sgid.

run the command:

which write

what are the permissions on the write command? _____.

What is the name of the group name on the write command ? _____.

Using the ***who*** command find the pts file for your neighbor. Now run the ls –al command on that pts file in the dev directory as follows:

Assume the pts file is 5 then you run:

ls -al /dev/pts/5

what is the group on that file and what permission does the group have?

_____

Now to see how that helps try to write a message directly to your neighbor's terminal as follows:

echo hello > /dev/pts/5

What happened? _____.

Now using the ***write*** command write the same message to your neighbor's terminal as follows:

write u1112233
hello
ctrl-d

What happened?_____.

In both cases, it was you (same user with same permissions) that was trying to write a message to the other user's terminal. ***Why did it not work when you tried to do it directly while it worked using the write command? (hint the (s) permission on the write command).***

_____

_____.

Adding (s) Permission

To add the s permission to your files, use the chmod command with four digits instead of three as before, for example:
create a file called newfile (touch newfile).

chmod 2777 newfile

What permissions are now on file newfile? _____.

chmod 4777 newfile

What permissions are now on file newfile? _____

chmod 6777 newfile

What permissions are now on file newfile? _____

As you can see adding an even digit (2 or 4 or 6) will put (s) on group, user, or both respectively.

What command would you use to set the permissions on newfile to:

1. ***r_s_wxrwx*** _____
2. ***r_xrwsr___*** _____
3. ***rwSrwsr___*** _____

How do you get a capital s (S) and a small s (s)?

_____.

Networking

As users we are not allowed to modify network setups, but we can view some information on how networks are configured on Linux.

Run the command:

/sbin/ifconfig

What is the ip address (inet) of your machine?

_____.

What is the MAC (HWaddr) address of your machine?

_____.

What is the netmask used by your machine?

_____.

Run the command
/sbin/route

What is the default gateway?

_____.

Run the command:
/bin/netstat -n | grep 23
This will give information about the telnet connections made to/from the system.
List the quad (Socket Connection) for your telnet connection:

_____.

Use the ftp tool to copy files from windows to Linux and vice versa.
Show your work to the instructor.