CIS-350
INFRASTRUCTURE TECHNOLOGIES
Unix/Linux – Advanced Command Line Interface (CLI) & Scripts
LAB #5

This Lab is worth 50 points. The questions in the Lab 5 Report that you need to answer are worth 25 points. You **must** log in to Ubuntu Linux and actually work the lab commands on the Ubuntu Linux system. The hands-on work is worth remaining 25 points.

Due date: **See Blackboard**

Objectives: Learn more about **Linux commands and script files**

Login to Ubuntu Linux to learn several extra Unix/Linux commands and develop shell scripts. In part I of the lab, you will learn some additional Unix/Linux commands such as **cancel**, **PS1**, **sleep**, **clear**, **head**, **tail**, **set**, **env**, **less**, **cal**, **date**, **echo**, **history**, **grep**, **wc**, **spell**, and **tee**. In part II of the lab, you will create and execute several shell script files. Each script file will contain several commands. For some commands or script programs to operate properly, you may need to create some dummy directories and files below your home directory.

All the commands and script programs that you encounter in this lab should work well in all four shells [Bash, Korn, Bourne Again (Bash), and C shell].

At the end type your short evaluation of the lab and suggest how it could be improved to make it clearer. What commands worked and what did not? What commands or script files need an extra explanation? You will need to submit this evaluation of the lab as well as Lab 5 Report.

Conventions Used in the Lab

(1) If a part of a line to enter appears in **bold** text, it is a part of the name of a Unix/Linux program or instruction and has precise meaning to the system. It must be entered exactly as is.

(2) Portions of commands that you enter are in *italics*. These are the portions of the instructions that are modifiable by users. For instance, I include suggested filenames in the commands I ask you to enter. You will later use the same command in conjunction with your own files. The italics indicate the portions of the command line you can change and still have it work.

(3) If a system file name or system variable is used in a command, it is in ***bold-italics*** indicating that it can be replaced with other system file names or system variables, but they must be spelled exactly as shown.

(4) A distinctive type of font (Arial, pt 11) is used for screen display output that is included in the lab. For example,

```
This is screen display output.
```

(5) Parts of the commands appear in **bold**, *italic,* and/or ***bold-italics*** - see points (1) through (3) above. To execute a command, type a command in lower- and/or upper-case exactly as you see it on the command line and press the Enter/Return key.

Now, let's start the lab. The script command allows you to record all your activities in a file. To record this interactive session with Unix/Linux, type

script *Lab5*

From now on all commands that you will type and the output from these commands/script programs will be recorded in file *Lab5*.

PART I - Unix/Linux Commands

1. The **set** and **env** commands

These commands allow you to explore your Unix/Linux environment. The shell program that interprets your commands is started when you log on. Several pieces of information are given to your particular shell process, so that your computing environment is appropriate. You can examine how the environmental variables are set up. Type one of these two commands.

set or **env**

Because the listing may scroll off of the screen, you may pipe the output from the **set** or **env** command as input to the **more** or **less** command. Type

set | more or **env | less** (less has similar effect to more)

The **env** command may be more convenient to use than the **set** command as it generates less output. The output is listing of some of the variables that are currently set for your shell. Among the many displayed, you should find something like the following in this order:

```
SHELL=/bin/bash
USER=jmzura01
PATH= ……. /usr/bin:/etc: ……..
PWD=/home/jmzura01
HOME=/home/jmzura01
LOGNAME=jmzura01
```

The user id jmzura01 will be replaced by your_user_id. The ***LOGNAME*** variable is your account name that you entered when you logged on. The ***SHELL*** line indicates which of several shell programs is started at login to interpret the commands that you enter: ***csh*** is the C shell, ***bash*** is the Bourne Again shell, and ***ksh*** is the Korn shell. The ***HOME*** variable is the location of your work space or home directory. The ***PATH*** variable is very long and lists the directories where the shell looks to find Unix/Linux utilities you request.

2. The **echo** command

The command allows you to evaluate variables. The shell maintains a series of variables and their values. You can ask the shell to evaluate the specific variables. Type

echo $*USER*
echo $*SHELL*
echo $*LOGNAME*
echo $*USER* $*SHELL* $*LOGNAME*

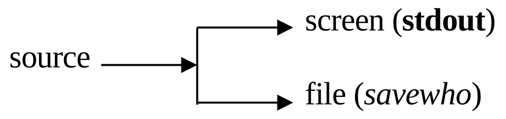
The variable name should be typed in uppercase. The **$** character has special meaning to the shell. It tells the shell to locate the variable whose name follows, and replace this string with the variable's value. The **echo *$SHELL*** command allows you to find out which shell is the login shell.

3. The **tee** command

You can combine commands so that the response of a command can come both to the monitor and a file. The Unix/Linux command to do this is called **tee**. This name is derived from the physical device called T-joint attached to a water pipe, for example. In this case the T-joint lets water out from one source to two outlets. This is exactly what the *tee* command in Unix/Linux does. It lets the output go to **stdout** and also go to a designated file, for example, *savewho*.

Type

who | tee *savewho*



The command sends the output of the *who* command to the monitor as well as creates a file called *savewho* and places the result in that file. Note that the *tee* command always comes after a main command (for example, *who*) is given. It always comes after a pipe (|).

Type

ls | tee *savels* **| sort -r**

This command saves in file *savels* the result of **ls** command and also shows on the monitor the same result, but in a reverse sorted order.

The *tee* command takes two options.

i	option tells the system to ignore the interrupts such as CTRL-C
a	option causes the output to be appended to the end of the file.

Type

ls -l | tee -a *savels*

Type:

cat *savels* **| more** or just **more** *savels*

to check the contents of the file *savels*.

4. The **PS1** command

Note that in PS1 command, a '1' is the digit not a letter 'l' in lowercase. To change the system prompt, use the **PS1** command (in uppercase). This allows one to change the level 1 prompt. To change level 2 prompt one can use the **PS2** command. For example, type

PS1=*'bash> '*

Hereafter the system prompt will be *bash>* instead of *your_user_id@mercury:~$*. For example, *jmzura01@mercury:~$*. This remains in effect until the end of login session. If you want to have the $ prompt back, type

PS1=' *jmzura01@mercury:~* **'**

5. The **history** command.

Unix/Linux has a facility to recall and see what the most recent commands that you had given are. It is called **history**. Type

history or **history | more** or **history | less** **(**if the output scrolls of the screen)

This recalls the recent commands that you had given. The number of the commands shown may differ from system to system.

6. The **wc** command

It is often useful to count the lines, words, or characters in a file. The Unix/Linux utility **wc** (word count) accomplishes this task. Type

wc *savewho*

The output from the **wc** utility will be similar to the following:

```
2       10      111     savewho
```

This output consists of the number of lines (2), words (10), and characters (111) read from the file, followed by the filename that was read (here *savewho*). When you run this command, it is very likely that you will not get the same output for the number of lines, words, and characters.

The **l**, **w**, and **c** options in the **wc** command display the number of lines, words, and characters, respectively. Type

wc -l *savewho*
wc -w *savewho*
wc -c *savewho*

7. The **grep** command

It is similar to the Windows **find** command and allows one to locate specific lines in a file. The file *users_on* contains a record for each user who is logged on when you create the file. Type the following commands, replacing *yourlogin* with your actual login name.
who > *users_on*
grep *yourlogin users_on*

The **grep** utility selects the line in the file *users_on* that contains the string of characters that is *your login*, and outputs it. In the command, replace *yourlogin* with your actual *user id*.

Now look for all lines in the file that contain the string *pts*. Type

grep *pts users_on*

8. The **date** and **cal** commands

To obtain the date and time, type

date

To display the calendar for the current month in the current year, type

cal

To display the calendar for year 2017, type

cal *2017*

To display the calendar for November 2017, type

cal *11 2017*

9. The **sleep** utility

The **sleep** command is one of the ways to affect scheduling. The command does nothing for a specified time. You can have a shell script suspend operation for a period by using **sleep**. The command **sleep** *time* will delay for *time* seconds. You can use sleep to control when a command is run, and to repeat a command at regular intervals. For example, type the command:

(**sleep** *3600*; **who >>** ***log***) **&**

provides a record of the number of users on a system in an hour. It creates a process in the background that sleeps (suspends operation) for 3600 seconds; then wakes up, runs command **who**, and places (appends) its output in a file named *log*. The "*;*" is a command separator. It allows one to specify multiple commands in parenthesis ( ) on the same command line.

You can also use **sleep** within a shell program to regularly execute a command. The script,

```
$ (while true
> do
> sleep 3600
> finger userid
> done) &
```

can be used to watch whether the user *userid* is logged on every hour. For the script to work, the *userid* on the **finger** command should be replaced by an actual *userid*.

Type the command

ps

too see that the two previous commands (processes) are running in background.

Try also to execute these simple commands:
sleep 30

It will put the shell to sleep for 30 seconds and you will not be able to do any other activities.

sleep 30 &

The command will be executed in the background. The shell prompt will reappear immediately after you submit the command and you will be able type other commands. When the sleep command executing in background is over, the message will appear on the terminal.

10. The **clear** command

To clear the screen, type

clear

11. The **head** and **tail** commands

To display a few lines from the beginning or the end of a file, one can use the **head** or **tail** commands, respectively. Type

head *prog1.c*

Note that in Lab 4, you should create file *prog1.c*. If you do not have this file in your account, type the name of any other file that contains more than 10 lines. The **head** utility reads the first ten lines of file(s) named as arguments.

To display the first four lines from a file, type

head *-4 prog1.c*

To view the last 10 lines from a file, type

tail *prog1.c*

12. The **spell** command (I am not sure if it will work. I may have to install this command)

Now using a nano or pico editor, create a file *test1* with a few misspelled words. Type

spell *test1*

You should have all misspelled words in this file displayed on the screen.

PART II - Programming with the Shell

In this section of the lab you will get acquainted with several Bourne Again shell scripts containing programming statements that control operation of the shell and Unix/Linux utilities. These scripts can be used to perform repetitive, complex, and routine tasks. The detailed understanding of the shell language and its syntax is beyond the scope of this course. The lab will show you only some limited features of the Bourne Again shell programming language. The scripts are simple and they should also run in the C, Korn, and Bourne shells. I will try to explain the major programming statements used in the script files.

1. Creating Simple Script Files with the Commands You Already Know

A. Using nano editor, create this script file named *simple1*:

```
echo 'This is a simple script file'
ls -al | more
ls prog*.c
mkdir jack don adam
cat prog1.c
```

B. Close the file. After already working Labs 3 and 4 as well as part of Lab 5, you should understand all the statements used in this file. Grant yourself execute permission by typing:

```
chmod u+x simple1
```

Now you can execute this script file by typing

./*simple1* and pressing Enter.

The characters **./** on the command line tell the shell exactly where to find the shell script *simple1*, in your current directory known as dot. Check using the **ls -al** command if the directories *jack*, *don*, and *adam* have been created. To remove these directories use the **rmdir** command.

Type:

```
rmdir jack don adam
```

Creating Simple Script Files with the Commands You Already Know, Continued

Using nano or pico editor create and run this simple script file after granting yourself an execute permission. Name the file *simple2*.

```
echo Your files are
ls
echo Today is
date
```

Create another script called *arguments* with the following contents.

```
echo hello
echo $1
echo $2
echo $3
echo $*
echo good bye
```

Use command **chmod** to make the file executable and run by entering:

./arguments Butch Kaye Jason Amber Brandon

./arguments Helen Joe Cathy Isaac

The shell interprets **$***1* as the value of the first argument (*Butch*) on the command line, **$***2* is the second argument (*Kaye*), $3 as the third argument (*Jason*), and **$*** is all arguments. User information can be given to a process by including the information as arguments. Arguments may affect a script behavior during execution, obviously not in this simple script.

You should see the following output for the 1st run :

```
hello
Butch
Kaye
Jason                    (Amber and Brandon are omitted because there is no echo $4 and echo $5)
Butch Kaye Jason Amber Brandon
good bye
```

2. Creating More Complex Script Files

Features Shown: Echoing Fixed Strings, Passing User Input into a Script

Multiple logins. A user may be logged on to Unix/Linux on multiple ports. This script assists in determining the number of times you are logged on.

A. Using nano editor create a script name *checkuser* containing the following two lines:

```
echo The number of times you are currently logged on is:
who | grep $LOGNAME | wc -l
```

Command	Interpretation
echo	The **echo** utility reads all its arguments and writes what it reads to standard output, usually connected to the workstation display. Programmers often use the **echo** commands in shell scripts to send information to the user.
who	Instruction to the shell to execute the **who** utility.
|	Instruction to the shell to connect the output of *who* to the input of the next utility, **grep**.
grep	Instruction to the shell to execute the **grep** utility.
$*LOGNAME*	Instruction to the shell to evaluate the variable ***LOGNAME*** and place its value on the command line, replacing the string **$*LOGNAME***. The shell then passes the value of the variable (your login id) to *grep* as an argument. To **grep**, the first argument, your login id, is interpreted as the target string to look for in the input. Only lines that contain your login id are output by **grep**.

|	Instruction to the shell to connect the output of **grep** to the input of the next utility, **wc**.
wc	Instruction to the shell to execute the **wc** utility.
-l	Instruction to the shell to pass the argument **–l** to **wc**. The **wc** utility interprets **–l** as instruction to output only the count of lines it receives as input, not characters or words.

B. Start two more terminal windows at your workstation, i.e., log on twice. You should have 3 terminal sessions open now.

C. Make *checkuser* executable by entering

chmod *755 checkuser*

Note that the above command replaced the command **chmod u+x** *checkuser* used earlier. The octal "*755*" in the command stands for the user permission "*rwxr-xr-x*". Also, the user grant himself/herself a permission without explicitly specifying "*u+x*".

D. Run the script by entering

checkuser

You should get the message: *checkuser*: not found

Note that the current directory is not in the **PATH** variable that the shell uses to locate utilities. Add the current directory to your shell's search path by entering the following commands:

PATH=$PATH::
export PATH

E. After you have added your current directory to the search path, you can run the script by entering

checkuser

The number of times that the user running the script (in this case, you) is logged on to the system is displayed (should be 3). Note that you do not have to type **./***checkuser*.

F. Modify all lines in the *checkuser* script as follows, so that it includes reading input from the user:

echo *Please enter the login name of the user you want to check*
read *name*
echo *The number of times $name is currently logged on is:*
who | grep $*name* **| wc -l**

G. Run the modified script by typing

checkuser

or

./checkuser

H. When prompted, enter the login id of a user who is currently logged on and then press Return.

Originally, *checkuser* produced a count of current logins for the user who ran the script. Now, it will do the same for any login name entered by the user when asked by the script. It is an interactive shell script.

Examining the code

Command	Interpretation
read *name*	Instruction to create a new variable called *name* and assign it the value of whatever is read from the keyboard.
echo *The …$name … is:*	Information line written to the screen, including the value of the variable **$***name*.
who | grep $*name*	Instruction to the shell to run **who**, pass its input to **grep**, and give **grep** the evaluated variable name as an argument. Hence, **grep** searches its input for the specified user and outputs only lines that include the user string.
| wc –l	The output of **grep** is passed to **wc**, which counts the number of lines.

3. Creating Complex Script Files

Features Shown: Handling Multiple Choices with the case Statement and Making the Script Loop Continually Using while

Let's try to design our own menu-driven interface using a shell script. The script syntax should be relatively easy to understand, especially after taking the C++, C#, or Java class.

A. Using one of the Linux editors, preferably *nano* or *pico*, create carefully a new script named *designmenu*. The script is on the next page. If, after several attempts, your program still does <u>not</u> compile, copy *designmenu* from the Linux *tmp* directory to your home/login directory by performing the steps below. (If you do not see the program in the *tmp* directory, please send me e-mail.)

```
cd  /              and press  <Enter>            - to bring you to the root directory
cd  tmp                       <Enter>            - to move you to tmp directory
ls -al                        <Enter>            - to check that file designmenu is there
cp  designmenu  /home/youruserid    <Enter>- to copy designmenu to your login directory
cd                            <Enter>            - to bring you to your login directory
ls  –al  designmenu           <Enter>            - to check that designmenu is there
```

The *designmenu* script is on the next page.

```
#!/bin/ksh
leave = no

while [$leave = no]

  clear
  do
  cat <<++
                      MAIN MENU
        1) Print current working directory
        2) List all files in current directory
        3) Print today's date and time
        4) Exit

Please enter your selection $LOGNAME:
++
  read selection

  case $selection in

    1)
      pwd
      echo "Press Enter to continue."
      read junk
    ;;

    2)
      ls -al | more
      echo "Press Enter to continue."
      read junk
    ;;

    3)
      date
      echo "Press Enter to continue."
      read junk
    ;;

    4)
      exit 0
    ;;

    *)
      echo "Invalid choice.  Try again."
      echo "Press Enter to continue."
      read junk
    ;;
  esac

done
exit 0
```

B. Explaining the code

Command	Interpretation
#!/bin/ksh	Runs the script in the Korn shell. However, if this command is absent, it should also run in the bash shell.
leave=no	Assigns *no* to the variable *leave*.

while [$*leave = no***]**
do
	While the variable *leave* has the value *no*, keep repeating the actions called for in the code following the **do** statement to the **done** statement.

clear	Clears the screen before each display of the menu.
cat <<++	Instruction to read the lines following this command line in the script as input To *cat*. The file is read as input to *cat* until a line is reached that has ++ characters located at the beginning of the line.
++	The ++ is the tag indicating the end of the text to read as input to *cat*.
case $*selection* **in**	Based on the value of the variable selection, do one of the following:
*1***)** **pwd** **echo** "Press Enter to continue." r**ead** junk	In the case where *selection* has the value 1: Run **pwd** to print working directory. The dummy statements **echo** and **read** are to prevent the screen from clearing immediately after displaying the results.
;;	End of course of action. Like **break** in C++
*2***)** **ls -al | more** **;;**	In the case where *selection* has the value 2: Run **ls -al | more** to list all files in this directory, one screen at a time. End of course of action.
*3***)** **date** **;;**	In the case where *selection* has the value 3: Run **date** to print the date and time. End of course of action.
*4***)** **exit 0** **;;**	In the case where *selection* has the value of *4:* Exit the program. End of course of action.
*) **echo** "Invalid choice. Try again." **echo** "Press Enter to continue"	In the case where *selection* has any other value:

;;	Print error message.
esac	End of course of action.
	End of this **case** control structure.
exit 0	Similar to **return 0** in C++. The statement indicates that the shell script ended OK.

C. Type

ls -al *designmenu*

and note that you do not have execute permission.

D. Make the file executable and then run it by entering

chmod ***755*** *designmenu*

./*designmenu*

Note that the octal number '***755'*** in the **chmod** command is equivalent to '*rwxr-xr-x*'. The command adds execute permission to the user although '*u*' and '+' is not explicitly written in the command.

E. When you are prompted for input, type either a 1, 2, 3, 4, x, X, or any other character followed by the Return. Note that only typing 4, you exit the loop and the script program.

To save the script in a file named *Lab5*, type *^D* (*CTRL-D*) at the beginning of the command line. It will terminate the script session. Look up the contents of the *Lab5* file using one of the editors and verify whether it contains the log of your entire Unix/Linux session with *Lab5* activities.

4. Type *exit* or *logout* to log out from Linux.

This is the end of Lab 5.

Submit Lab 5 Report. (See the Assignments/Labs folder on Blackboard)