Tutorials on the Unix Shell & vim editor

Objective:

In this lab session you are to continue learning about the Unix shell commands, through three tutorials, and an interactive tutorial on *vim*, the powerful Unix text editor. Also, you will learn how shell commands can be used in an executable shell script file (as a shell program) to accomplish more complicated tasks. By the end of this lab you should be able to write a complete shell program.

Your TA may give you additional exercises on this subject.

Introduction:

The Unix *shell* provides users with a set of commands for the operating system. Some basic commands and constructs were introduced in Lab1. Here you will learn about the following additional commands:

`&`	Creates a new foreground process sending the current one to the background
`|`	Creates a pipe that makes the output stream of one command as the input stream of the next
`>`	Redirects the output of a command to go into a file instead of the screen, replacing its contents
`>>`	Same as > but it adds to the output file instead of replacing its contents
`<`	Redirects the input of a command to be taken from a file instead of the keyboard
`*`	A wildcard character that matches any number of characters in filename searches
`?`	A wildcard character that matches any one character in filename searches
`bg`	Sends the current job to execute in the background
`chmod`	Changes the protection information associated with a file
`fg`	Brings a background job to execute in the foreground
`jobs`	Lists current jobs
`kill`	Terminates execution of a process
`man`	Displays manual pages describing Unix commands and library routines
`printf`	Writes ASCII (text) output to "standard output" or to a file
`ps`	Gives information about the status of processes
`sleep`	Suspends the execution of a process
`sort`	Sorts the input data lines
`vi/vim`	A well-known Unix text editor. Type "`vi FILENAME`" to create and edit files See the vi tutorial at the course site references page
`who`	Lists currently logged-in users

The *vim* Text Editor: (1 point)

Go to the following link and start your *vim* tutorial to learn its most basic operations.

Tutorial *vim* Link: http://www.openvim.com/tutorial.html.

The TA will give some exercises to practice what you have learned.

More Unix Commands: (1 point)

1. Go to item #6 titled "*Unix Tutorial for Beginners*" in the references page linked from the main page of this course and do tutorials 3, 4 and 5. Use the skills of *vi* / *vim* you learned from the tutorial above to write and apply the examples on your Linux system. Take snapshots of your work.

2. Do the exercises of the tutorials in your Linux system and take snapshots of your work.

3. Put all the snapshots you took above in one .pdf file and submit to your TA as part #1.

Shell Script Programming: (2 points)

Introduction:

The Unix "shell" is a command language provided by the operating system. Shell commands can be placed in a file, known as a *shell script*. The shell interface is a complete programming language, with variables, mathematical expressions, conditional statements, loop constructs, and so on. These advanced shell features are covered in the tutorial. You can find out more about them through the "**man**" command. Note: You should do steps 1 to 7 below *before* going to the lab. Use the Linux virtual machine you installed before. Step 8 should be done during the lab session for credit.

You can skip the discussions and go directly to the numbered instructions written in RED

Step 1: Placing Unix/Linux commands into a file (called a *shell script*)

4. Create a file named "**work**" in your home directory, which has the following contents:

```
ls                # display the names of files in the current directory
sleep 1           # suspend execution for one second
printf "Hello\n"   # display text
sleep 2           # suspend execution for two seconds
ls -l             # display file information in long format. (That's a small L, not a one.)
```

5. The file "**work**" contains a sequence of Unix/Linux shell commands. To execute these commands:

   - Type: "**work**" at a Unix prompt.

   If you get an error message about "`work not found`", you can type: "**./work**" instead, or you can fix this problem once and for all by adding the current directory to your search path, i.e. to the value stored in the **$PATH** environment variable. To fix this problem:

   - Type: "**vim .bashrc.d/user-path**", to edit the specified resource extension file.
   - Add the following line: [**PATH=".:$PATH"**]. Where, the period is the symbol for the current directory, the colon is a separator, and `$PATH` is the old value of the **$PATH** variable.
   - Save the file, log-out and log-in again then type, "**work**" again.

6. You may probably get the response "`work: permission denied`". Fix the permission problem as follows:

   - Type: "**`ls -l`**" to see the permission bits of your files. By default, the "**`work`**" file has the bits "`-rw-r--r--`". This means that the owner of the file (which is you) has permission to read and write this file (but not to execute it).
   - Type: "**`chmod u+x work`**".
   - Retype: "**`ls -l`**" to see that the permission bits are changed to "`-rwxr--r--`". This means that the owner of the file has permission to execute it.
   - Type: "**`work`**". You should see three bursts of output, separated by sleep times of one second and two seconds.

7. The commands "**`ls`**" and "**`printf`**" produce output that goes to the stream called **"stdout"**, standard output. By default, standard output is displayed at your terminal. Redirect this output to a file by typing:

   ```
   work > result
   ```

   This time the script creates a new file named "`result`", executes the commands in "**`work`**", and puts the output into "`result`".

8. Type: "**`cat result`**" to see the output.

9. Retyping "`work > result`" gets the error message "`result: file exists`". But the following sequence should work:

   ```
   rm result      # remove (i.e. delete) the file named "result"
   work > result
   ```

10. Retype: "**`work >> result`**" without deleting "`result`" first. It should work this time. What happened?

    Type: "**`cat result`**" to see that the "`result`" file has in it, the output of the first and the second runs.

Step 2: Using the man command

The Unix/Linux "**`man`**" command, short for "manual", provides information about Unix commands.

11. Type the following to read about two of the Unix commands that were used in step 1:

    ```
    man ls
    man sleep
    ```

    In step 1, you noticed the difference in output between "**`ls`**" and "**`ls -l`**". Now read about the `-l` option on the manual page for **`ls`**. The description shows "use a long listing format". Make sure that you can find the description of the "`-l`" option among the many other options that "**`ls`**" has. Effective use of man requires that you be able to sift through and find the parts of the **`man`** entry that are of interest to you. All man pages have a standardized layout, so with practice you can become fast at finding what you need.

12. Sometimes you need information about a topic, but do not know the name of the relevant Unix commands. In that case, use **`man`**'s `-k` option to get a list of all Unix commands that mention a certain keyword. (For more details about `-k` and other options, type: "**`man man`**" to see the manual page for the **`man`** command.) For example, suppose you are interested in finding out about protection in Unix.

Type:

```
man -k protect
```

You see a list of a few items related to memory protection and making variables read-only.

13. Suppose you actually wanted to learn about file protection.

    Type:

    ```
    man -k file
    ```

    A huge number of Unix commands come up in response to this query.

14. Type: **control-c** to stop this list. (Hold down the control key and type "**c**".)
    This terminates the process that is currently in the *foreground*. In this case, our *foreground* process is the "`man -k file`" one. If you want to look over a long list such as this, you may redirect it to a file using: "`man -k file > info`". Then you can use a text editor or other tools to search the "`info`" file.

Step 3: Creating concurrent processes via "&"

If you place **&** at the end of a Unix command, this command is executed as a separate process. The new process runs concurrently with the process that is handling your terminal session.

15. Type in quick succession:

    ```
    work &
    work &
    ```

    The output from both processes is mixed together on your screen. There is an additional line of output for each process, of the form: "`[1] 5896`". This tells you that a new process has been created, with process ID 5896.

16. The "**ps**" command (short for "process status") can be used to see a list of processes that you have running. The TIME column tells you how much CPU time has been used by this process. In quick succession, type the following two commands:

    ```
    work &
    ps
    ```

    The output from the "**ps**" should show you at least two processes: one is the process called "`bash`" which is managing your current Unix window. The other is the process called "`bash`" which is running the "**work**" script. Depending on when you issue "**ps**", you might see a third process called "`sleep`"; this one was started up by the "**work**" process.

17. It's confusing to have the output from "**ps**" interleave with the output from "**work**". To avoid this, redirect the "**work**" output as follows:

    ```
    work > out &      # now the output produced by work goes to a file named "out"
    ps
    ```

Step 4: Observing the output from concurrent processes

18. Create *three* new files, with the following contents:

    1. Contents of file "**messageOne**" (type the first three lines; repeat them several hundred more times using copy-paste):

       ```
       printf "O"
       printf "N"
       ```

```
printf "E"
printf "O"
printf "N"
printf "E"
printf "O"
printf "N"
printf "E"
…
```

2. Contents of file "**messageTwo**" (type the first three lines; repeat them several hundred more times using copy-paste):

```
printf "T"
printf "W"
printf "O"
printf "T"
printf "W"
printf "O"
…
```

3. contents of file "**messages**":

```
MessageOne &
MessageTwo &
```

19. Type: "**chmod u+x messageOne messageTwo messages**".

20. Type: "**messages | less**" to execute the Unix commands in this file. Use the space bar to page through the output, and the "q" key to exit. You will learn about the "|" symbol in step 5 below.

    You see that the output from the processes "messageOne" and "messageTwo" get mixed together. Type: "**messages**" a few more times. You can get **different** output each time, depending on the exact timing with which these processes execute. Other processes on the system (belonging to other users, or to the operating system) can affect the timing of your processes.

A note about printf:

If file "messageOne" had used a single `printf "ONEONEONEONEONE"` command (instead of hundreds of separate "printf" commands) then its output would stay together. This is because Unix provides *mutual exclusion* for the duration of a "printf" command. If several processes start a "printf" to the same file at the same time, then one process goes first, printing all its characters, before the other process goes. For example, concurrent execution of "`printf "Yes"`" and "`printf "No"`" produces one of the following outputs: YesNo or NoYes. It never produces an output such as YNoes or NYeso. The situation is different if the printing is split into "`printf "Y"`", "`printf "e"`", `printf "s"`", "`printf "N"`", `printf "o"`".

Step 5: Using pipes "|"

The command, "**ps -a**", lists the status of most processes on the machine, and "**ps -ef**" or "**ps -ax**" (depending on the version of ps you get) lists the status of all processes. Often this is a long list.

21. Suppose you are interested in seeing only bash processes (these are shell processes; one gets created each time someone logs on to the machine, and each time a new "terminal" window is opened.) To do this, Type:

```
ps -ax | grep bash  # if you get an error message, type "ps -ef" instead of "ps -ax"
```

    The vertical bar establishes a *pipe*: Two concurrent processes in which the output of one process is connected to the input of the other process. In this case, the output of "**ps**" is connected to the input of "**grep**". Both of these processes ("**ps**" and "**grep**") run in parallel.

The pipe has a certain amount of buffer space associated with it. So, if "**ps**" produces output quickly (relative to the rate at which "**grep**" uses up its input) then the buffer fills up. Once the buffer is full, the operating system makes the "**ps**" process wait until the "**grep**" process reads some input. This frees up buffer space which "**ps**" can fill again.

On the other hand, if "**ps**" is slow at producing its output, and the pipe's buffer is empty when "**grep**" wants input, then the operating system makes the "**grep**" process wait. In many cases, the two processes are fairly well matched in speed, so the buffers stay partly full and neither process has to wait for I/O.

The "**grep**" command searches a file for a pattern. For example, "**grep bash**" checks each line of input to see whether it contains the substring "bash". All lines that contain "bash" are produced as output of "**grep bash**". Further information can be found from "man grep".

Step 6: Killing processes

For debugging, it is important to know how to kill processes. The following simple C program is used to illustrate this.

22. Create a file called "**main.c**" with the following contents: (You may use cut-and-paste.)

```
// This is a simple c program with an infinite loop.
void main () {
   while ( 1 < 2 ) { } // do nothing

} // end of main
```

1. Compile this program with: "**gcc main.c**". By default, the compiler puts the result into an executable file called "**a.out**".
2. Type: "**a.out**" to execute this program. Now the program is executing its infinite loop. You cannot type any standard shell commands because you do not get a prompt from the shell.
3. Type: **^C** (control-c) to terminate this program, hold down the "control" key and then type "c". After typing **^C** once or twice, you should get a prompt from the Unix shell.
4. Now run the **a.out** program again as a separate process. Do this by typing: "**a.out &**". You continue to receive a prompt from Unix, while **a.out** runs in the *background*.
5. Type: "**ps**" to see the status of the **a.out** process. Included in the output of **ps**, there will be a line that looks something like this:

   ```
   3078 pts/0 R      0:04 a.out
   ```

   This tells you that the command "**a.out**" is being executed by a process with process id (PID) equal to 3078. Currently this process has used 0:04 seconds of CPU time.

6. Type: "**ps**" again a few seconds later. You will see that the process has now used more CPU time.
7. You cannot kill this process with **^C**, because that command only works for the *foreground* process. Instead, to get rid of this process, use **ps** to find the PID number. Then type:

   ```
   kill 3078     # replace 3078 by the PID number of your a.out process.
   ```

8. Type: "**ps**" again to verify that you no longer have a process called **a.out**. (Note that, logging out does ***not*** kill this process.)

Step 7: Shell Script Example

Here is an example of a shell script to show various system configurations, like:

1) Currently logged user and his log name
2) Your current shell
3) Your home directory
4) Your operating system type
5) Your current path setting
6) Your current working directory
7) Show Currently logged number of users
8) About your os and version ,release number , kernel version
9) Show all available shells
10) Show mouse settings
11) Show computer cpu information like processor type, speed etc.
12) Show memory information
13) Show hard disk information like size of hard-disk, cache memory, model etc.
14) File system (Mounted)

23. Create a text file that contains the following, then run it and show the result to your TA:

```
#!/bin/bash
#
# Written by Vivek G. Gite <vivek@nixcraft.com>
#
nouser=`who | wc -l`
echo -e "User name: $USER (Login name: $LOGNAME)" >> /tmp/info.tmp.01.$$$
echo -e "Current Shell: $SHELL"  >> /tmp/info.tmp.01.$$$
echo -e "Home Directory: $HOME" >> /tmp/info.tmp.01.$$$
echo -e "Your O/s Type: $OSTYPE" >> /tmp/info.tmp.01.$$$
echo -e "PATH: $PATH" >> /tmp/info.tmp.01.$$$
echo -e "Current directory: `pwd`" >> /tmp/info.tmp.01.$$$
echo -e "Currently Logged: $nouser user(s)" >> /tmp/info.tmp.01.$$$

if [ -f /etc/redhat-release ]
then
    echo -e "OS: `cat /etc/redhat-release`" >> /tmp/info.tmp.01.$$$
fi

if [ -f /etc/shells ]
then
    echo -e "Available Shells: " >> /tmp/info.tmp.01.$$$
    echo -e "`cat /etc/shells`"  >> /tmp/info.tmp.01.$$$
fi

if [ -f /etc/sysconfig/mouse ]
then
    echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
    echo -e "Computer Mouse Information: " >> /tmp/info.tmp.01.$$$
    echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
    echo -e "`cat /etc/sysconfig/mouse`" >> /tmp/info.tmp.01.$$$
fi
echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
echo -e "Computer CPU Information:" >> /tmp/info.tmp.01.$$$
echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
cat /proc/cpuinfo >> /tmp/info.tmp.01.$$$

echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
echo -e "Computer Memory Information:" >> /tmp/info.tmp.01.$$$
echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
cat /proc/meminfo >> /tmp/info.tmp.01.$$$

if [ -d /proc/ide/hda ]
then
    echo -e "------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
    echo -e "Hard disk information:" >> /tmp/info.tmp.01.$$$
```

```
    echo -e "----------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
    echo -e "Model: `cat /proc/ide/hda/model` " >> /tmp/info.tmp.01.$$$
    echo -e "Driver: `cat /proc/ide/hda/driver` " >> /tmp/info.tmp.01.$$$
    echo -e "Cache size: `cat /proc/ide/hda/cache` " >> /tmp/info.tmp.01.$$$
fi
echo -e "----------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
echo -e "File System (Mount):" >> /tmp/info.tmp.01.$$$
echo -e "----------------------------------------------------------------------" >>
/tmp/info.tmp.01.$$$
cat /proc/mounts >> /tmp/info.tmp.01.$$$

if which dialog > /dev/null
then
    dialog  --backtitle "Linux Software Diagnostics (LSD) Shell Script Ver.1.0" --title
"Press Up/Down Keys to move" --textbox  /tmp/info.tmp.01.$$$ 21 70
else
    cat /tmp/info.tmp.01.$$$ |more
fi

rm -f /tmp/info.tmp.01.$$$
```

Step 8: A Shell Script Exercise: (2 Points)

24. Write a script to determine whether the given command line argument ($1) contains "*" symbol or not, if $1 does not contain "*" symbol then add it to $1, otherwise show the message "**Symbol is not required**".

    For example, if we called this script **sc2** then after giving the command:

    ```
    $sc2 /bin
    ```

    Here $1 is **/bin**, the script should check whether "*" symbol is present or not. If not, then the script should print "**"*" is required**" and add the "*" to $1, i.e. **/bin/***. And if the "*" symbol is present in $1, then, the script should print "**Symbol is not required**".

25. Test your script as:

    ```
    $sc2 /bin
    ```

    and as:

    ```
    $sc2 /bin/*
    ```

26. When you finish, submit your script and its runs as part #2 to your TA.