

# CIS-350 INFRASTRUCTURE TECHNOLOGIES

## LAB #5

Due: See Blackboard

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

Login to your Athena account 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**, **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 the Korn shell. If you switch to the Bourne shell or C shell during the course of this lab, some of the commands will not work because these shells use a slightly different command language.

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? Which commands or script files need an extra explanation? You will need to submit this evaluation of the lab as well as a log of your interactive session - printed file *lab5*.

### 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) To execute a command, you need to type a command in **bold**, *italic* or ***bold-italics*** 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 a 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** command. Type

**set | more**                      or                      **env | more**

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:

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

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 shell, and **ksh** is the Korn shell. The **HOME** variable is the location of your work space or home directory. The **PATH** variable 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
```

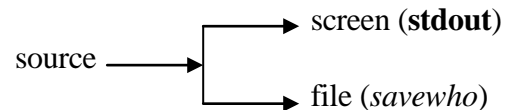
The **\$** character has special meaning to the shell. The variable name should be typed in uppercase. It tells the shell to "locate the variable whose name follows, and replace this string with the variable's value." The **echo \$SHELL** command allows you to find out which shell is the login shell.

### 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 *jmzura01@user1\$*. This remains in effect until the end of login session. If you want to have the \$ prompt back, type **PS1**=*"jmzura01@user1\$ "*.

## 5. The **history** command.

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

### **history**

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 is similar to the following:

```
28    139   390   total
```

This output consists of the number of lines (28), words (139), and characters (390) read from the file, followed by the filename that was read (here *savewho*).

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 MS\_DOS **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 *yourlogin*, 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
```

**cal** command. To display the calendar for the current year, type

```
cal
```

To display the calendar for year 2002, type

```
cal 2002
```

## 9. The **sleep** utility

The **sleep** command is one of the ways to effect 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 the **who** command, and places (appends) its output in a file named *log*. The ";" is a command separator. It allows one to specify multiple commands 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*.

Try also to execute these simple commands:

```
sleep 20
```

It will put the shell to sleep for 20 seconds and you will not be able to do any other activities.

*sleep 20 &*

The command will be executed in the background. The shell prompt will reappear immediately after you submit the command and you will be able to 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 lab4, you should create a 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

Now using a nano 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 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 shell programming language. 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. 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.

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

Using nano 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 by
```

make the file executable and run by entering:

*./arguments Butch Kaye Jason Amber Brandon*

The shell interprets **\$1** as the value of the first arguments on the command line, **\$2** is the second, ... **\$\*** is all arguments. User information can be given to a process by including the information as arguments.

## 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
<b>echo</b>	The <b>echo</b> utility reads all its arguments and writes what it reads to standard output, usually connected to the workstation display. Programmers often use the <b>echo</b> commands in shell scripts to send information to the user.
<b>who</b>	Instruction to the shell to execute the <b>who</b> utility.
	Instruction to the shell to connect the output of <i>who</i> to the input of the next utility, <b>grep</b> .
<b>grep</b>	Instruction to the shell to execute the <b>grep</b> utility.
<b>\$LOGNAME</b>	Instruction to the shell to evaluate the variable <b>LOGNAME</b> and place its value on the command line, replacing the string <b>\$LOGNAME</b> . The shell then passes the value of the variable (your login id) to <i>grep</i> as an argument. To <b>grep</b> , 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 <b>grep</b> .
	Instruction to the shell to connect the output of <b>grep</b> to the input of the next utility, <b>wc</b> .
<b>wc</b>	Instruction to the shell to execute the <b>wc</b> utility.
<b>-l</b>	Instruction to the shell to pass the argument <b>-l</b> to <b>wc</b> . The <b>wc</b> utility interprets <b>-l</b> as instruction to output only the count of lines it receives as input, not characters or words.



B. Start two 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
<b>read</b> <i>name</i>	Instruction to create a new variable called <i>name</i> and assign it the value of whatever is read from the keyboard.
<b>echo</b> <i>The ...\$name ... is:</i>	Information line written to the screen, including the value of the variable <i>\$name</i> .
<b>who</b>   <b>grep</b> <i>\$name</i>	Instruction to the shell to run <b>who</b> , pass its input to <b>grep</b> , and give <b>grep</b> the evaluated variable name as an argument. Hence, <b>grep</b> searches its input for the specified user and outputs only lines that include the user string.
<b>wc</b> -l	The output of <b>grep</b> is passed to <b>wc</b> , 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++ or Java class.

A. Create carefully a new script named *designmenu* containing the following lines. If the script below does not compile/execute, copy the exact script from this lab and paste it in to your Unix/Linux editor.

```
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
<i>leave=no</i>	Assigns <i>no</i> to the variable <i>leave</i> .
<b>while</b> [ <i>\$leave = no</i> ] <b>do</b>	While the variable <i>leave</i> has the value <i>no</i> , keep repeating the actions called for in the code

<b>clear</b>	following the <b>do</b> statement to the <b>done</b> statement.  Clears the screen before each display of the menu.
<b>cat &lt;&lt;++</b>	Instruction to read the lines following this command line in the script as input To <i>cat</i> . The file is read as input to <i>cat</i> 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 <i>cat</i> .
<b>case \$selection in</b>	Based on the value of the variable <i>selection</i> , do one of the following:
1) <b>pwd</b> <b>echo</b> "Press Enter to continue." <b>read</b> junk	In the case where <i>selection</i> has the value 1: Run <b>pwd</b> to print working directory. The dummy statements <b>echo</b> and <b>read</b> are to prevent the screen from clearing immediately after displaying the results.
;;	End of course of action. Like <b>break</b> in C++
2) <b>ls -al   more</b> ;;	In the case where <i>selection</i> has the value 2: Run <b>ls -al   more</b> to list all files in this directory, one screen at a time. End of course of action.
3) <b>date</b> ;;	In the case where <i>selection</i> has the value 3: Run <b>date</b> to print the date and time. End of course of action.
4) <b>exit 0</b> ;;	In the case where <i>selection</i> has the value of 4: Exit the program. End of course of action.
*) <b>echo</b> "Invalid choice. Try again." <b>echo</b> "Press Enter to continue" ;;	In the case where <i>selection</i> has any other value: Print error message.  End of course of action.
<b>esac</b>	End of this <b>case</b> control structure.
<b>exit 0</b>	Similar to <b>return 0</b> 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 a 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 line. It will terminate the script session.

This is the end of lab 5. **Turn in hardcopy of your short typed *evaluation of the lab* and print *lab5* file** in which your session is recorded. (If lab5 is too large, reduce its size to 4-5 pages.)

Note: Ftp the files from the CRC RedHat Linux to your laptop or local desktop and print them from there.