ECE 364 Software Engineering Tools Lab

Prelab 01

Introduction to Bash

Instructions

1. Work in your PrelabXX directory, where XX is your prelab number, i.e. 01, 02 ...etc.
2. Copy all files from **~ee364/labfiles/PrelabXX** into your **PrelabXX** directory. You may use the following command:
   `cp ~ee364/labfiles/PrelabXX/* .`
   (Note there is a dot at the end of the command. The dot represents current working directory.)
3. Remember to add and commit the file into SVN! Use SVN keywords for all scripts. You can simply copy the header file, that you created in Lab00, for your scripts. Always ensure that your most recent version is checked into SVN. **We will ONLY grade the scripts present in the SVN.**
4. Name and spell your scripts exactly as instructed. Your scripts may be graded by an automated process.
5. When writing scripts, you may accidentally introduce an infinite loop. To kill your script, use the keyboard shortcut `Ctrl+C`.
6. **Do NOT submit a hard copy. We will *only* look at the submitted scripts.**
7. **Review Lecture slides before attempting this prelab.**

Part I: `echo` and `printf`

1. a. What is the output of:
   `echo "Price: $0.99"`

   Price: /bin/bash.99

   b. Why?

   Because the echo just print the following and $ is used for variable

   c. What would you need to do to make the output appear as: Price: $0.99?

   echo "Price: \$0.99"

2. Write a `printf` statement to print each of the following:
   a. 3.14159 rounded to the nearest hundredth.

   printf "%.2f\n" 3.14159

b. A string called **`String_Var`** padded to at least 5 characters wide.

printf "%s\n" $String_Var

3. What is the output of each of the following commands? If the output is just whitespace, please write a textual description of it.
   a. `echo "${HOME}"`

   /home/yara/ee364/ee364c01

   b. `echo '${HOME}'`

   ${HOME}

   Explain the difference in output between "a" and "b".

   The first one is treat "${HOME}" as a vaiable and read the value, the second one treat '${HOME}' as

   c. `printf $USER`

   ee364c01

   Does `printf` append a newline to its output?

   no

   d. `printf "%15s\n" $SHELL`

   /bin/bash

   e. `echo They said, "Hello, World."`

   They said, Hello, World.

4. How would you `echo` the following?
   a. $2.679

   echo \$2.679

   b. The name of the currently running script

   echo "$0"

Part II: Command line arguments and working with files

One important feature of Bash scripts is that the programmer has full control over what to do with arguments that are passed in by the user. Like C or Java, Bash has variables for the number of command-line arguments as well as the arguments themselves.

1. Write a script called **`sum.bash`** that sums all values passed to it and simply prints the sum. You can assume that at least two values will be passed on the command line, and you can also assume that all passed-in values will be integers. Use the `shift` builtin combined with a loop to iterate through the command line parameters. Remember to add the script to **`SVN`** and commit it when you are finished.
2. Bash makes working with text files easy. While you will always be able to reference lecture slides during labs and practical exams, it would be useful to have most of conditional checks memorized. Answer the following questions:
   a. Write a conditional that checks if **`test.txt`** exists

   [[-e test.txt]]

   b. Write a conditional that checks if **`file.in`** is readable AND writable.

(HINT: You can combine checks the same way as in C - use && between them.)

[[-r file.in && -w file.in]]

c. Write a conditional that checks if the value stored in the variable **`FILENAME`** is executable

[[-x $FILENAME]]

d. Write a conditional that checks if **`foo`** is a directory

[[-d foo]]

3. Write a short script called **`exist.bash`** that takes a list of filenames as an argument. For each given filename, print a message that says
`File <filename> is readable!`
if the file can be read. If the file cannot be read or does not exist, create an empty file with that name. (HINT: Look up the `touch` command.)
NOTE: Whenever a value in a prelab or lab is presented as **`<variable>`** , it means to print the value of that variable, not the string '<variable>'.
4. Write a script called **`line_num.bash`** that uses a loop and the `read` command which takes a single filename as an argument and prints out the file with line numbers added to the beginning of the lines. You should first check that the user provided a single argument. If not, print an error message and exit. You should next check to see if the file is readable and exit with the message
`Cannot read <filename>`
if it is not. **Almost all scripts you write that work with command line parameters or files should perform these checks unless we tell you otherwise.**
Example output:

```
$ line_num.bash
Usage: line_num.bash <filename>
$ line_num.bash unreadable
Cannot read unreadable
$ line_num.bash 1.txt
1:15
2:37
3:23
4:9
5:21
```

Part III: Basic Commands

1. **The `head` command:**
The `head` command can be used to print out the start of a file. To view more information on the `head` command, enter `man head` at the command prompt. How would you use `head` to do the following?

a. Display the first 10 lines of the file **`file.in`**

head -n 10 file.in

b. Display the first 20 lines of the file **`file.in`**

head -n 20 file.in

c. Display the first 17 characters of the file **`file.in`**

head -c 17 file.in

2. **The `tail` command:**
   The `tail` command can be used to print out the end of a file. To view more information on the `tail` command, enter `man tail` at the command prompt. How would you use `head` to do the following?
   a. Display the last 10 lines of the file **`file.in`**

      tail -n 10 file.in

   b. Display every line after (and including) line 27 of the file **`file.in`**

      tail -n +27 file.in

   c. Display the last 88 characters of the file **`file.in`**

      tail -c 88 file.in

   d. Display the last 5 lines of the file **`file.in`**

      tail -n 5 file.in

3. **The `cut` command**
   The `cut` command can be used to parse tabular data. To view more information on the `cut` command, enter `man cut` at the command prompt. You have been given a file named **`football.txt`**. Open it in a text editor to get an idea of its contents. The first column contains players' numbers, the second column lists positions, and the third column gives the players' names. Use the `cut` command to achieve the following:
   a. Display only the players' names

      cut -f2 football.txt

   b. Display the players' numbers and positions (but not names)

      cut -f1,3 football.txt

   c. Display the first digit of each player's number

      cut -c1 football.txt

   d. What would you need to add to your cut commands in parts a-c if the file were formatted with colons as the field separator instead of tabs?

      add -d:

4. **The `paste` command**
   The `paste` command can be used to join data files line by line. To view more information on the `paste` command, enter `man paste` at the command prompt. You have been given three files named **`1.txt`**, **`2.txt`**, and **`3.txt`**. Each contains one of the columns from **`football.txt`** explored with the `cut` command. The first file contains players' numbers, the second file lists positions, and the third file gives the players' names. Use the `paste` command to achieve the following:
   a. Recreate the contents of **`football.txt`** (join all 3 files using tabs)

      paste 1.txt 2.txt 3.txt

   b. Print a list of data where each line has the form position:player

      paste -d: 2.txt 3.txt

5. **The `wc` command**
   The `wc` command can be used to obtain counts of words, characters, and lines in a file. To view more information on the `wc` command, enter `man wc` at the command prompt. You have been given a long text file named **`book.txt`**. Use the `wc` command to achieve the following:
   a. Display **only** the number of words in the file (along with the filename)

wc -w book.txt

b. Display the number of lines, words, and characters in the file (along with the filename)

wc -lwc book.txt

c. Display the length of the longest line in the file (along with the filename)

wc -L book.txt

6. **The `diff` command**
   The `diff` command can be used to view differences between files. To view more information on the `diff` command, enter `man diff` at the command prompt. You have been given two files named **`a.txt`** and **`b.txt`**. Use the `diff` command to answer the following:
   a. What is the output of `diff a.txt a.txt` ?

   nothing

   b. How many lines are different between **`a.txt`** and **`b.txt`**?

   1

   c. What differences exist between **`a.txt`** and **`b.txt`**?

   3c3

Part IV: Combining commands using pipes

You can combine commands in Bash using the pipe `|` operator. For example:

```
$ cat 1.txt | wc -w
5
```

The output of `cat 1.txt` is sent to **`STDIN`** of the `wc -w` command.

1. Write a command that pipes the output of `wc -w` to a `cut` command to achieve the exact same output as above

   cat 1.txt | wc -w | cut -c1

2. Use a `cut` command piped to another `cut` command that prints only the first character of each entry in the second field of **`football.txt`**

   cut -f2 football.txt | cut -c1

3. From here on, use any combination of commands you know joined by pipes (if needed) to achieve the goal. Display only the first 5 of the last 10 lines of **`book.txt`** (e.g. lines 16-20 if a file had 25 lines)

   tail -n10 book.txt | head -n5

4. Count the number of words in the 6th line of **`book.txt`**

   sed '6q;d' book.txt | wc -w

5. Display the first 10 characters of each line of (**`1.txt`** joined with **`2.txt`**)

   paste 1.txt 2.txt | cut -c 1-10

6. Count the number of characters in the first 4 lines of a listing of the differences between **`a.txt`** and **`b.txt`**

diff a.txt b.txt | head -n4 | wc -c

7. Count the number of files that start with the letter **a** in the current directory.

   echo a* | wc -w

8. Same as (7), except output 5 if there are 5 or more files that begin with **a**.

   if [[$(echo a* | wc -w) > 5]]; then echo 5; else echo a* | wc -w; fi

Part V: Capturing command output in a script

in order to write useful scripts, you will often need to examine the output of a command that is run from your script. Say, for example, that you are writing a script to examine whether the files in a directory have been added to **SVN**. You would like to use the `svn status` command in your code. The output is empty (or equal to "") if a file in the current directory is up-to-date with the **SVN** copy. So if you could capture the output of `svn status`, you would have an easy way to determine which files are up to date. Fortunately, Bash provides an easy way to do this. Simply enclose the command in parentheses and place a **$** before it. For example:

```
DATA=$(head -1 test_file)
```

would set the value of the variable **$DATA** to the first line of **`test_file`**. Complete the following exercises:

1. Write a line in a script that would set the value of the variable **$FILENAMES** to a list of all the files ending in **`.c`** in the current directory.

   FILENAME=$(echo *.c)

2. What does the following code do?

   ```
   STATUS=$(svn status exist.bash | head -c 1)
   ```

   get the first charater of the svn status for exist.bash

3. You can use variables in combination with command output capturing. Try the following and report what it does:

   ```
   FILETYPE=c
   FILES=$(ls *.$FILETYPE)
   echo $FILES
   ```

   it prints the the files that have .c extension

Part VI: Putting it all together (Or: Getting ready for lab)

The scripts you are asked to write in this section will be similar to what you will need to do in lab. Labs typically involve the creation of 3-4 scripts that do something useful. Often times, these scripts will be similar to those that you complete in the prelab. Completion of the prelabs correlates strongly with student performance in this course.

SCRIPT NAME:	`check_file.bash`
ARGUMENTS:	`filename` to check.

INPUT: None.
OUTPUT: Information about various attributes of the file.

Before you begin, type the following commands:

```
touch File_1
chmod 600 File_1
```

Note: It would be in your best interest to create other files in a similar fashion for testing purposes. Your script checks the following attributes in the exact order as given below and prints out the results.

-e exists
-d directory
-e exists
-f ordinary file
-r readable
-w writable
-x executable

You must also do error checking with regard to the number of arguments being passed to the script. Your output should look EXACTLY as follows:

```
$> ./check_file.bash File_1
File_1 exists
File_1 is not a directory
File_1 is an ordinary file
File_1 is readable
File_1 is writable
File_1 is not executable
$> ./check_file.bash .
. exists
. is a directory
. is not an ordinary file
. is readable
. is writable
. is executable
$> ./check_file.bash
Usage: ./check_file.bash <filename>
```

Note: `$>` denotes the command prompt and there is no newline after the output. Incidentally, just to be clear, there should always be 6 lines of ouput, each indicating whether or not that particular attribute is true. In other words, here is the breakdown (replace **`<filename>`** with the actual file name being checked):

Line 1 should either say:
`<filename> exists`
OR
`<filename> does not exist`

Line 2 should either say:
`<filename> is a directory`
OR

```
<filename> is not a directory
```

2. and so on....

SCRIPT NAME: `svncheck.bash`
ARGUMENTS: None.
INPUT: From file.
OUTPUT: `STDOUT`

You are provided with a file called **`file_list`**. This file contains a list of all files that need to be added to the **SVN** repository. Your task in this section is to write a script that reads this file line-by-line, checks whether the file is in **SVN** (using the `svn status` or `svn list` command), and also checks whether it is executable. For each file: If the file is not in **SVN** but exists, you should check to make sure it is executable. If it is not executable, you should ask the user if he or she would like to make it executable. If the user responds with **`y`**, then make the file executable using `chmod` , otherwise, leave it as non-executable. Then add the file to **SVN** (but do not commit it yet.)
If the file is in **SVN** but is not executable, perform the following command:

```
svn propset svn:executable ON <filename>
```

where **`<filename>`** represents the name of the file to modify.

Finally, if the file is not in **SVN** and does not exist, print an error message of the form
`Error: File <filename> appears to not exist here or in svn`

You do not need to handle the case where the file is in **SVN** but not in the current directory. You also do not need to handle the case where the file in **SVN** and the file in the working directory are different (though it may be helpful to do so.) Once the script has gone through all the files, it should execute a single `svn commit` with the message

3. `Auto-committing code`

SCRIPT NAME: `sensor_sum.sh`
ARGUMENTS: `filename` to check.
INPUT: From file.
OUTPUT: `STDOUT`

You are given a log file of sensor measurements with the following format:

```
        SensorID        N0      N1      N2
```

The **`SensorID`** field identifies a particular sensor device and has the following format: **`<Sensor#>-<HardwareID>`**
The remaining fields **`N0`** through **`N2`** are integer values measured by the sensor.

Your job is to write a bash script that takes the **`filename`** of a sensor log and prints the sum of the measurements made by each sensor. When printing the results, only the Sensor # should be displayed. You have been given an example file called **`sensors.log`** that you should use to test your script.

An example of the output your script should produce is given below:

```
> ./sensor_sum.sh sensors.log
S0 112
S1 75
S2 50
S3 83
S4 73
S5 76
S6 68
S7 117
S8 81
S9 112
```

HINTS:

i. Use the `read` command to extract the columns.
ii. The `cut` command can be used to cut from standard input!
iii. You can pipe the output of an `echo` command to use `cut` on a bash variable. (e.g. `echo $myvar | cut -d',' -f2`)
iv. Your script should also perform the appropriate file and argument tests to make sure the input file is valid.

```
> ./sensor_sum.sh
usage: sensor_sum.sh log

> ./sensor_sum.sh sdfsdf
error: sdfsdf is not a readable file!
```
