Lab 4: Bash Script and Bitwise Operations

Objectives

- Gain experience with writing bash scripts
- Learn and practice bitwise C operations

You may complete this lab individually or with one colleague from this class using pair programming rules (see the course Canvas page).

0 Introduction

So far, we've been issuing simple commands into terminals to navigate file systems, compile code, run programs, etc. We've only just scratched the surface on the power the shell gives us. The shell, as you know, interprets commands given to it and performs those actions, but it allows for much more than single, simple commands. Shells provide full-blown languages with variables, loops, and functions. There are several shells, including Bourne, Bash (Bourne-again), Korn, C Shell, and Z shell. The most ubiquitous shell is Bash. In this lab, you will continue to explore the Bash shell, learning to write programs in it. Programs written in the Bash language are called Bash scripts.

1 Writing Bash Scripts

Frequently a developer needs to execute same action on multiple files at once, e.g. rename a bunch of files or test a program on many test files at once. The easiest way to do this is with a bit of scripting. A shell script is a file containing a sequence of commands that can be invoked by "executing the file" as a command at the prompt. These scripts can be run at the prompt even if you use Eclipse or some other development environment for the actual code editing. Everything you write in your script could have been typed in the terminal window by hand, but it would be much slower.

Create a file and name it `myScript`. At the top of your file, type

```
#!/bin/bash
```

This tells your operating system to use Bash to parse and run the script.

The (optional) last line in your script should be

```
exit
```

Lab 4: Bash Script and Bitwise Operations

Printing to the terminal window

The easiest thing to put in script is the `echo` command to print to the terminal. The `echo` command simply passes all command line arguments it receives to the standard out of the shell.

```
echo "Hi there"
```

You can also combine this with the redirects to print to your `handin.txt` file.
(**Warning!!!** Use >> to append or you will delete your original `handin.txt` file!!)

```
echo "Hi there" >> handin.txt
```

Running a script

You can only run a command at a prompt if it is in your current executable search path. You can see all of the directories on your path if you run the command `echo $PATH`. Unless you see the directory where you stored your `myScript` file, you will not be able to run it by simply typing `myScript`.

To run your script, you need to specify either an absolute or relative path to `myScript`. For example, if you are in the same working directory as your script, you add the path "`./`" to the front of the script you want to run. For example: `./myScript`. The alternative to explicitly writing out the path to the script file is to update your `PATH` variable to contain the directory containing the script. (We will see variables shortly.)

Comments

The comment marker in any UNIX shell variant is #. Any text on a line to the right of a # is commented out.

```
echo "Hi there" >> handin.txt # this part is a comment
# this whole line is a comment
```

Variables

You can create variables in your Bash script and set their values.

```
x="my favorite string"
y=10
```

It is very important that you do not add whitespace between the variable name and the =, or between the = and the right-hand side expression. To use these variables, you will need add a $ to access their values.

Lab 4: Bash Script and Bitwise Operations

```
echo "$x"
```

It is common to enclose a variable name using the `${…}` notation to access it. One use case for this notation is where you want to concatenate its value with some other expression within some string. A good rule of thumb is to always enclose variable accesses in curly braces to avoid future problems.

```
echo "${y}0" # will print 100
echo "$y0"   # will print nothing
```

Arguments from the command line

Some commands take parameters (officially named flags or options if they start with a dash, and command line arguments otherwise). For example, the `rm` command takes `-i`, `-r`, `-f`, and various combinations of these flags. These parameters are accessible inside your script using the names `$1 $2 $3` etc. Note that these are predefined variables that you can access directly in your script.

```
echo "The first parameter was ${1}"
```

To call your script, add the parameters after the name (with spaces). For example "`./myScript 3`" or "`./myScript eep`". Bash defines a # variable for determining the number of command line arguments provided when the script was run.

```
echo "You gave me ${#} command line arguments"
```

Selection (if then else fi)

Selection uses the words **`if`**, **`then`**, **`else`**, and **`fi`**.

```
if [[ 3 != 2 ]]; then
  echo "not equal"
else
  echo "equal"
fi
```

The word **`then`** may either be on the same line as the **`if`**, in which case there needs to be a semicolon after the test, or it can appear on the next line. The indentation is not enforced but makes it readable.

Options inside the if-test include:

`[[3 != 2]]`	
`[[ 3 != 2]]`	*# inequality*
`[[ 3 -gt 2]]`	*# greater than*

Lab 4: Bash Script and Bitwise Operations

`[[ 3 -lt 2]]`	*`# less than`*
`[[ -d $variable]]`	*`# is the variable's value a directory?`*
`[[ -f $variable]]`	*`# is the variable's value a regular file?`*
`[[ -e $variable]]`	*`# is the variable's value an existing directory/file?`*
`[[ ! -d $variable]]`	*`# is it`* ***`not`*** *`a directory?`*
`[[ -d $variable || -e $variable]]`	*`# the boolean`* ***`or`***
`[[ -d $variable && -e $variable]]`	*`# the boolean`* ***`and`***
`[[ ${#} -gt 2 ]]`	*`# is the number of parameters greater than 2?`*

There are many more comparison operators. You can find them on the web by searching for "bash script" or by visiting the sites mentioned later in this document.

For loops (for in do done)

For loops in Bash use the output of a "shell expansion". This can either be a variable access, use of an expansion pattern like `*` (any file), or execution of a command specified between two backquote characters (for example, `` `ls` ``). The `for` loop will assign each whitespace separated string in this expansion to the variable named after the `for` keyword on successive iterations of the loop.

```
for file in *; do
  echo ${file}
done
```

This example means, for each file in the current folder, print its name. The loop ends with `done`. The following is an equivalent piece of code:

```
for file in `ls`; do
  echo ${file}
done
```

Using variables with expansions, you can do some fancy expansions. For example, you can also list all of the files found in a directory provided as a command line argument.

```
for file in ${1}/*; do
  echo ${file}
done
```

The indenting is, as in Java, optional. The line breaks act as semicolons in Java and are less optional.

This example means, for each file matching the pattern `${1}/*` (all files in the folder given by the first parameter), print its name.

You can also look for files with some name characteristics.

Lab 4: Bash Script and Bitwise Operations

```
for file in *.class; do
  echo ${file}
done
```

This example prints the name of all Java class files found in the working directory. A word of warning with pattern shell expansions: if there are no matches to the pattern, then Bash will assume that you meant to include the `*` as a non-expanded character. So, if there are no class files in your directory, the for loop will still iterate once assigning the `file` variable the value "`*.class`".

How is pattern expansion useful? Let's say you have a folder full of executables (Java class files).

```
#! /bin/bash
date > results          # save the date to my results
for file in `ls *.jar 2> /dev/null`; do # for each file here named
                                        # something.jar
  echo ${file}          # echo its name
  java -jar ${file} >> results # run the jar and save the results done
exit
```

The `2> /dev/null` means that the errors (stderr) produced by the command `ls *.jar` are sent to `/dev/null`. This program runs all the files and saves them to a dated results file. I only need one command to run the program even if the program does a lot of work.

More information on Bash

This is just a simple introduction to Bash. It is actually much more powerful and elegant than this tutorial describes. More information can be found (with clearer writing and examples) on websites such as

- http://www.linuxconfig.org/Bash_scripting_tutorial
- http://en.wikipedia.org/wiki/Bash_(Unix_shell)
- http://tldp.org/LDP/Bash-Beginners-Guide/html
- http://www.gnu.org/software/bash/manual/
- http://www.ibm.com/developerworks/library/l-bash.html

2 Bash Scripting Problems

Problem 1 (2 pts)

1. Create a Bash script named `script1`.
2. Have it print one line to the terminal window and another line into a results file.
3. Make your script take 4 command line arguments.

Lab 4: Bash Script and Bitwise Operations

4. Add comments to tell us which arguments are what (2 need to be numbers, 1 needs to be file name, 1 needs to be a folder name).

5. Make variables for each command line argument.

6. Print the value of the second variable.

7. Write conditional statements to tell which number is bigger. Print a sentence telling us which one it was. Write conditionals to tell us if the file/folder names exist, and when they exist tell us if the names correspond to files or folders. (These can be any number of selections, you don't need one uber-complex selection).

8. Write a loop to search for some kind of files inside your folder parameter. Print their names.

9. Run your script a few times with various inputs, concatenating both the command line you ran and the output of the script into your `handin.txt` file.

10. Submit `script1` and `handin.txt`.

Problem 2 (2 points)

In this and subsequent problems, follow the same step-by-step approach as in Problem 1. You will need to search the web and read more Bash tutorials to accomplish these tasks. In these two problems you will be writing scripts to help instructors and TAs grade work in Introduction to Computer Science courses. In the Intro II class, many students turn in a whole bunch of files (`.class` files and other files created by Eclipse) even though they are asked to turn in only their `.java` files. On top of that, for each submission, Canvas creates a `.txt` file describing that submission. Write a Bash script called `keepjava` that takes in an absolute directory path as a parameter and removes all files in that directory that are not `.java` files. Test your script.

Problem 3 (3 points)

In addition to the above file clutter problem, there is a naming problem. When I download a file from Canvas that was submitted as "`lab1pr1.py`" the name becomes

```
lab0pr0.py_amaus_attempt_2020-02-02-14-40-17_lab1pr1.py
```

Write a Bash script called `nameonly` that takes in an absolute directory path as a parameter and renames all files in that directory to just contain the username of the user that submitted them. So,

```
lab0pr0.py_amaus_attempt_2020-02-02-14-40-17_lab1pr1.py
```

should become `amaus.py`.

Lab 4: Bash Script and Bitwise Operations

Test your script.

3 C bitwise operations

C provides several bitwise operators to allow you to interact with values at the level of the bit. Geeks for Geeks has a good tutorial on them. In a file named `bitwise.c`, write the following three functions (utilizing bitwise operations) along with a main method to test them.

prefix (4 points)

Write a function `prefix(x,n)` that returns the first `n` bits of the number `x` using bitwise operations (anything other than bitwise operations will receive 0 credit). Both input numbers should be input as `unsigned ints`. The main function should output these bits in binary and in decimal format.

suffix (4 points)

Write a function `suffix(x,n)` that returns the last `n` bits of the number `x` using bitwise operations (anything other than bitwise operations will receive 0 credit). Both input numbers should be input as `unsigned ints`. The main function should output these bits in binary and in decimal format.

toggle (5 points)

Write a function `toggle(x,n)` that toggles the `n`th bit of a number `x`. As input, it would take a number `x`, and the position of the bit `n` (starting from the rightmost bit 0). Both input numbers should be input as `unsigned ints`. The resulting number with bit toggled should be returned: if a bit at position `n` is 0, it should be changed to 1 (and vice versa). The main function should output this number in binary and in decimal format.

Submission

Submit `script1`, `handin.txt`, `keepjava`, `nameonly`, and `bitwise.c` to Canvas.

Acknowledgement: Thanks to Prof. Lea Wittie of Bucknell University for developing the Bash scripting lab from which this lab stems.