

Lab B: Bash

1 Introduction

Here are more Bash commands. As a side note, UNIX/Linux folks often refer to folders as directories.

2 Zipping and Tarring

When you develop or want to distribute a large collection of files, it is often much more convenient to just package them up and compress them in what is called a file archive. There are a number of utilities that accomplish this task. In fact, you've already made extensive use of archives when you write Java programs, as the libraries are stored in various jar (Java ARchive) files.

Such utilities can be extremely useful if you have a bunch of folders for cs203, cs204, and cs206 that you might want to look at again some day but right now they are just eating up your quota. In fact, now is probably a good time to check your quota to find out if you are likely to run out of space during this semester.

Run the command `quota -s` and compare the first number you see to the second number you see. An M character stands for megabyte (or, possibly, mebibyte) while an m character is short for million. If you are within 200 megabytes of your quota, send an email to `ecst@bucknell.edu` and request that your quota be expanded.

The UNIX command `tar` will be used frequently this term. The term `tar` stands for "Tape ARchive" and comes from a time when it was common to copy a set of files and/or folders to a tape-based backup system. Today the command has the same name but is more commonly used to package up a set of files to send as an attachment or just for easier transfer from one place to another. An extension of this command is often available on a UNIX derived systems (like Linux) under the same name, `tar`. Modern `tar` commands are more advanced than their predecessors and are capable of both creating and compressing the archives. You might also hear archives created with `tar` called tarballs. The `tar` utility is an ideal mechanism for packaging up old folders – the compression can result in considerable space savings.

The command has two forms, one for packaging (and compressing) files and the other for (decompressing and) unpackaging the same set of files. To package and compress your folder `cs208` you would issue the following command:

```
tar cjvf cs208.tbz cs208
```

The "cjvf" portion of the command is a sequence of arguments specifying that `tar` should create the tarball (c), compress it with the `bzip2` utility (j), display information on what the command is doing (v), and use a specified name for the resulting archive (f). The name `cs208.tbz` is the name referred to by the `f` argument and the `cs208` is the name of the folder to be archived and compressed. That's a lot. The important thing to remember is the command. The `tar` command is actually quite configurable. You might want to explore the man pages (`man tar`) and info pages (`info tar`) to learn more about all of the other compressions schemes available.

To uncompress and unpackage an archive one issues the following command: `tar xjvf cs208.tbz`

The argument `x` (eXtract) indicates to unpackage or "untar" the specified archive. When this is done a new folder will appear named `cs208` containing all the files which were there at the time the packaging took place.

2.1 Problem

Pick an old folder (not `cs208`). Use `tar` on it to both tar and compress it. Remove the folder (not the archive!). Use `tar` again to untar and uncompress the just created tarball. If you accidentally obliterate your folder, see the end of this lab ASAP for a lesson in undeleting files. (Undelete only works for a day or two). Save the commands you used to your `handin.txt` file for this lab.

You might consider using `tar` to archive some of your stuff - to decrease the amount of disk space you are using. Of course the archive doesn't save any space unless you also delete the folder you archive!

3 A Bash program

During this course, you will frequently need to 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 basically 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 Linux to use the Bash program to run the script.

The (optional) last line in your script should be

```
exit
```

3.1 Printing to the terminal window

The easiest thing to put in a script is a command to print something using the `echo` command. 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
```

3.2 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.)

3.3 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
```

3.4 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 a `$` to access their values.

```
echo "$x"
```

It is common to enclose a variable name in curly braces after \$ before accessing it in a situation where you want to concatenate its value with some other expression without following the access with a space. A good rule of thumb it to always enclose variable accesses in curly braces to avoid future problems.

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

3.5 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!"
```

3.6 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]]	inequality
[[3 -gt 2]]	greater than
[[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.

3.7 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 ` . 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.

```
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 this possibly 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
```

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.

3.8 More information on Bash

This is just a very simple introduction to Bash. It is actually much more powerful and elegant than this simple 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://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>

3.9 Problem

1. Make a script (if you haven't already).
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.
4. Add comments to tell us which arguments are what (2 need to be numbers, 2 need to be files or folders).
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/or are 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 directory 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. Commit your script and your `handin.txt` file into SVN under the directory `labs/Bash_B`.

4 Optional: Undeleting files

Bucknell is running a program called `snapshot` on the Linux system. `Snapshot` makes a copy of all of your files every 4 hours. These copies are saved for a day or two.

If you delete a file or folder and want it back (or mess it up and want to turn back the clock), go to where the problem is. Type "`cd .snapshot`". If you do an `ls`, you will see folders with dates and times.

```
01092010-1200am/  01122010-1200am/  01142010-0800pm/  01152010-0400am/  
01102010-1200am/  01132010-1200am/  01142010-1200am/  01152010-0800am/  
01112010-1200am/  01142010-0400pm/  01142010-1200pm/  01152010-1200am/
```

You can go into these folders to see what files existed at that time. Go into one of those folders. Look at the files and folders in it. If you see one you need to undelete, copy it back into the original folder. To copy a folder, use "`cp -r`" so it copies any subfolders as well.