50.005 CSE

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Lab 1: OS Shell

Natalie Agus

1 Overview: OS Shell

As we have learned in class, a shell is an interface to allow user to access OS services. These services are those you encounter daily when using a computer, for example: file management (rename, create, list files, delete, etc), process management (run and terminate), OS configuration, I/O operations like printing, reading from disk, communication via network, resource management (overclock speed, VM size), protection (security settings), etc. It is named as 'shell' because it is the outermost layer around the OS kernel. OS shells are usually either in a form of command-line interface (CLI, also known as terminal) or graphical user interface (GUI) a.k.a things you see on your desktop, depending on your computer and OS. In this lab, we are going to learn a little bit about CLI, command lines, and Bash (Bourne Again SHell), which is one of the most commonly used unix shells.

2 Shell

What is shell? To know this better, open the terminal window. The terminal window in front of you **contains shell**, which enables you to use commands to access OS services.

2.1 Basic Commands [12m]

2.1.1 Commands Without Options / Arguments

Checkout the following basic commands:

1. **[1m]** Type `date` and press enter. Record your answer.
2. **[4m]** Do the same thing with `cal`, `pwd`, `ls`, `who`. Record your answer.

3. **[5m]** Based on your observation, whats the purpose of each of the five commands above?

2.1.2 Commands With Options / Arguments

The commands you have typed in step 1 and 2 are those that do not require arguments. Some commands require input arguments. Try to do the following:

1. **[1m]** The command `cd` changes your current working directory. Type `cd /path` where `/path` is the path to your desktop, and press enter. Has your current directory change? *Hint: you can confirm this by typing* `ls + enter` *and you will see that the output is the lists of files in your desktop*.

2. **[1m]** Type `echo hello world`. Record the output. What does this command do?

There is a whole lot of other UNIX commands thats just impossible to cover in a single lab. Below is the summary of the popular commands:

1. `mkdir dirname` when you want to create a folder or a directory and `rmdir dirname` when you want to delete an *empty* directory.

2. `rm dirname` can remove a directory that contain files

3. `touch newfilename.txt` creates a new file. It doesn't have to be just .txt, it can create any new file type.

4. `man commandname` is used to know more about the command and how to use it

5. `cp copyfromloc copytoloc` copies a file from a location to another

6. `locate filename` is used to locate a particular filename. You can also locate files containing specific words using `locate -i *word1*word2`, each word separated by an asterisk, the option -i is to ignore lower/upper case.

7. `cat filename` is used to display the contents of a file, and `wc filename` prints a count of newlines, words, and bytes for each input file.

8. `sudo command` stands for SuperUser Do, which is to execute command with administrative privileges

9. `df` shows the available disk space in each partition

10. `sudo apt-get install packageName` is used to install packages

11. `chmod -x filename` is used to make a filename executable. For example, if you have a file such as helloworld.py, each time you want to run it you have to call python with argument helloworld.py on command line: `python helloworld.py`. This can be easily reduced into just `helloworld.py` once you make it executable using the `chmod` command.

12. `hostname` and `hostname -I` returns your name in your host or network, and the latter gives your IP address in your network

13. `ping servername` checks connection to a server. For example `ping google.com` tells you whether your connection is active or not.

2.2 I/O redirection [7m]

2.2.1 Standard output

As you know from the previous part, `echo hello` is a command that means ”output hello to standard output”. A **standard output** is a default place for output to go, also known as **stdout**. Your shell is constantly watching that output place, and whenever there's something there, it will print it to your screen.

2.2.2 Standard input

The standard input (**stdin**) is a default place where commands listen for information. For example, all the commands above with no arguments listens for input on stdin. Try typing `cat` on the command line and press `enter`. Notice you can type any character from your keyboard, because it listens for input on stdin, outputting what you type to stdout (and your shell is watching that output place so it is being printed on your screen), until you type an `EOF` (end of line) character : `CTRL + d`.

2.2.3 Standard error

The standard error (**stderr**) is the place where error messages go. Try this command that will prompt an error `cat inexistent-file`. What is the output that you see? Similar to stdout, stderr is printed directly to your screen by your shell.

2.2.4 I/O Redirection

We can instantly see stdout and stderr because they are by default, printed to your terminal screen. We can redirect stdout using the > operator. Try the following tasks:

1. `echo hello > textfile.txt` + `enter`

2. `ls` + `enter`

3. **[1m]** What new file do you see?

4. `echo hello its me > textfile.txt` + `enter`

5. **[1m]** `cat textfile.txt`. What is the content of textfile.txt?

6. The > operator redirects the stdout to the text file (creating it if it doesnt exist yet) replaces the content of the file.

7. Now type `echo hello from the other side >> textfile.txt` + `enter`

8. **[1m]** `cat textfile.txt`. What is the content of textfile.txt?
9. **[1m]** What does >> do differently than >?

Similarly, we can redirect stdin using < operator. If we do `command < filename`, it means that we use the *content* of filename as an input to command. This is particularly useful for commands that only takes in input stream, and is unable to read a content of a file given a filename. One example is `tr`, a command line utility for translating or deleting characters. Do the following:

1. Create a text file with the following content: ”hello, have a good day today!”, give it a filename and save it to your current directory.
2. Type the following in command line (dont forget to navigate to your current directory first), `tr "[a-z]" "[A-Z]" yourfilename.txt`.
3. **[1m]** What is the output?
4. **[1m]** Now try `tr "[a-z]" "[A-Z]" < yourfilename.txt`. What is the output?
5. **[1m]** What is the difference between `tr "[a-z]" "[A-Z]" < yourfilename.txt` and `tr "[a-z]" "[A-Z]" yourfilename.txt`?

3 Bash Scripting [2m]

Now we have learned a bit about shell, lets move on to bash. Bash is a default interpreter on many system, so you have basically used it when you typed all the commands in the previous section. To see your default shell interpreter, type `echo $SHELL`. So what is scripting? It is simply lines of codes for the command lines. Imagine you want to create a bunch text files (using `touch`). You wouldn't want to type the command one by one, but rather just run a script that do this task in one shot.
Do the following:

1. Open a text editor and type the following:

   ```
   #!//bin/bash

   echo "Hello world"
   ```

   Save it as helloworld.sh in a directory that you know.
2. `cd` to that directory
3. Type `chmod +x helloworld.sh` and press `enter`
4. Execute the script by typing `./helloworld.sh` and press `enter`

5. **[1m]** Record your answer

You have just created and run a super simple bash script. Note that the # sign basically means a comment. Similar like coding in any other language, you can use variables, functions, conditional statements, loops, comparisons, etc in your bash script. We do not have enough time to run them all, but the following bash script shows some of the common operations:

1. Open a text editor and type the following:

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is
the best day of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"

string_a="UNIX"
string_b="GNU"

echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?

num_a=100
num_b=200

echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?

if [ $num_a -lt $num_b ]; then
    echo "$num_a is less than $num_b!"
fi

for i in 1 2 3; do
    echo $i
done
```

2. Make it executable using `chmod +x filename.sh` command and execute it by typing `./filename.sh`

3. **[1m]** Write down the output that you see.

Do not worry about Bash programming language. The purpose of this lab is just to expose you to fundamental basic concepts about shell and bash. You will not be tested with materials that are out of the scope of the lab.

4 Unix Makefile [9m]

Download and look at the files inside `Lab1_makeFileDemo`. Read all the .c and .h files and get an understanding of what each file is supposed to do. To compile the files and run the executable:

1. Type the command `gcc -o myexecoutprog.o main.c hello.c factorial.c binary` and press `enter`

2. Type the command `./myexecoutprog.o`

3. **[1m]** What is the output?

Basically, `gcc` compiles all the input argument files: `main.c, hello.c, factorial.c, binary.c` and produce an output (this is what `-o` means) named `myexecoutprog.o`. The next line runs the executable, which is `myexecoutprog.o` binary file. In this context, it is feasible to type out the source file one by one each time you want to compile your program. However in a large scale project with thousands of files, it is very tedious to do this all the time. Hence, the `make` command allows us to compile these files more easily. It requires a special file called the `makefile`.

1. In your terminal, navigate to the directory of `Lab1_makeFileDemo`, and type `make`

2. **[1m]** What is the output?

3. After executing `make`, realise that `myexecoutprog.o` is made. You can run the executable in the terminal by typing `./myexecoutprog.p` or by simply clicking that executable in your shell GUI (your desktop).

4. Open `makefile` with your text editor. The first four lines are the MACROS, which is a convenient shorthands you can make to make your life easier when typing these codes. Afterwards, there's a bunch of explicit rules that you can call using `make`. So in this makefile, you can try calling these in sequence and observe what each rule do: `make myexecout`, `make myexecoutFromMacro`, `make clean`, and then finally `make cleanFromMacro`.

4.1 Recompiling

Run this command: `make myexecout2`. You should see the following output on your terminal:

```
gcc    -c -o main.o main.c
gcc    -c -o hello.o hello.c
gcc    -c -o factorial.o factorial.c
gcc    -c -o binary.o binary.c
gcc -o myexecoutprog2 main.o hello.o factorial.o binary.o
```

Now open `binary.c` in text editor and add another function in it. You can write any function you want, the point is just to modify the file. Save your changes, and run `make myexecout2`. **[1m]** What is the output? **[1m]** Is it the same as above? **[1m]** Why?

Scroll down to the end of the `makefile`, and notice there's implicit rules there to determine dependency. This gives more efficient compilation. It only recompiles parts that are changed. The Figure below shows the data dependency between files that are specified in the `makefile`,

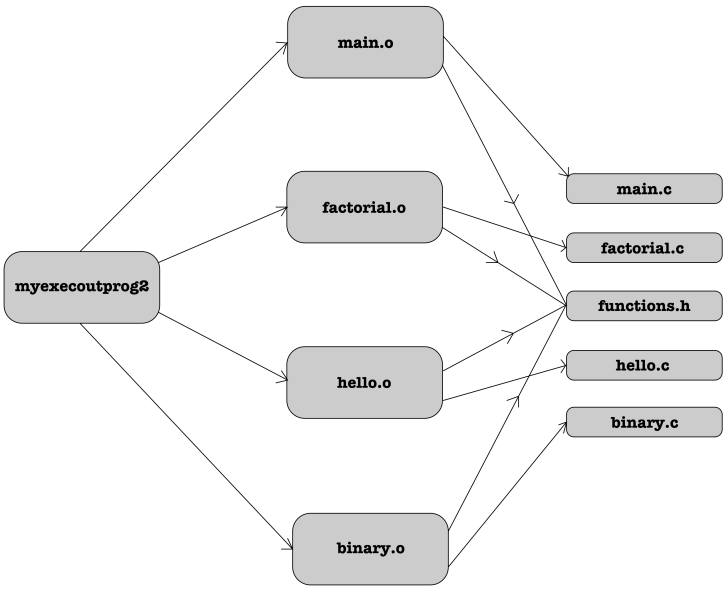

Figure 1

[1m] Which file you should change in order to force gcc to recompile everything? Now, create another .c file containing the implementation of any function that you want that can print something on the terminal. Add the function to `functions.h`, and call it from `main.c`. **[3m]** Modify the makefile to include this new .c file during compilation. Zip all the .c, .h, and the makefile.

5 Submission

Type out all your answers in the document Lab1_Answers.pdf. For the last part where you are tasked to modify the .c, .h, and makefile, upload all these files together with Lab1_Answers.pdf and upload to e-dimension (Submission link in Week 2). **Do not zip the files together**.