1 Unix Shell 1: Introduction

Lab Objective: *Unix is a popular operating system that is commonly used for servers and the basis for most open source software. Using Unix for writing and submitting labs will develop a foundation for future software development. In this lab we explore the basics of the Unix shell, including how to navigate and manipulate files, access remote machines with Secure Shell, and use Git for basic version control.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of the Linux and MacOSX operating systems. Most servers are Linux-based, so knowing how to use Unix shells allows us to interact with servers and other Unix-based machines.

A *Unix shell* is a program that takes commands from a user and executes those commands on the operating system. We interact with the shell through a *terminal* (also called a command line), a program that lets you type in commands and gives those commands to the shell for execution.

> **Note**
>
> Windows is not built off of Unix, but it does come with a terminal called PowerShell. This terminal uses a different command syntax. We will not cover the equivalent commands in the Windows terminal, but you could download a Unix-based terminal such as Git Bash or Cygwin to complete this lab on a Windows machine (you will still lose out on certain commands). Alternatively, Windows 10 now offers a Windows Subsystem for Linux, WSL, which is a Linux operating system downloaded onto Windows.

> **Note**
>
> For this lab we will be working in the `UnixShell1` directory provided with the lab materials. If you have not yet downloaded the code repository, follow steps 1 through 6 in the `Getting Started` guide found at `https://foundations-of-applied-mathematics.github.io/` before proceeding with this lab. Make sure to run the `download_data.sh` script as described in step 5 of `Getting Started`; otherwise you will not have the necessary files to complete this lab.

Basic Unix Shell

Shell Scripting

The following sections of the lab will explore several shell commands. You can execute these commands by typing these commands directly into a terminal. Sometimes, though, you will want to execute a more complicated sequence of commands, or make it easy to execute the same set of commands over and over again. In those cases, it is useful to create a *script*, which is a sequence of shell commands saved in a file. Then, instead of typing the commands in individually, you simply have to run the script, and it takes care of running all the commands.

In this lab we will be running and editing a bash script. Bash is the most commonly used Unix shell and is the default shell installed on most Unix-based systems.

The following is a very simple bash script. The command `echo <string>` prints `<string>` in the terminal.

```
#!/bin/bash
echo "Hello World!"
```

The first line, `#!bin/bash`, tells the computer to use the bash interpreter to run the script, and where this interpreter is located. The `#!` is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the bash interpreter.

To run a bash script, type `bash <script name>` into the terminal. Alternatively, you can execute any script by typing `./<script name>`, but note that the script must contain executable permissions for this to work. (We will learn more about permissions later in the lab.)

```
$ bash hello_world.sh
Hello World!
```

Navigation

Typically, people navigate computers by clicking on icons to open folders and programs. In the terminal, instead of point and click we use typed commands to move from folder to folder. In the Unix shell, we call folders *directories*. The file system is a set of nested directories containing files and other directories.

You can picture the file system as an tree, with directories as branches. Smaller branches stem from bigger branches, and all bigger branches eventually stem from the root of the tree. Similarly, in the Unix file system we have a "root directory", where all other directories are nested in. We denote it by using a single slash (/). All absolute paths originate at the root directory, which means all absolute path strings begin with the / character.

Begin by opening a terminal. The text you see in the upper left of the terminal is called the *prompt*. Before you start creating or deleting files, you'll want to know where you are. To see what directory you are currently working in, type `pwd` into the prompt. This command stands for **p**rint **w**orking **d**irectory, and it prints out a string telling you your current location.

```
~$ pwd
/home/username
```

To see the all the contents of your current directory, type the command `ls`, **l**ist **s**egments.

```
~$ ls
Desktop     Downloads       Public      Videos
Documents   Pictures
```

The command cd, **c**hange **d**irectory, allows you to navigate directories. To change to a new directory, type the cd command followed by the name of the directory to which you want to move (if you cd into a file, you will get an error). You can move up one directory by typing cd ...

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing cd ~ or just cd. You can navigate to root by typing cd /.

Problem 1. To begin, open a terminal and navigate to the UnixShell1/ directory provided with this lab. Use ls to list the contents. There should be a file called Shell1.zip and a script called unixshell1.sh. [a]

Run unixshell1.sh. This script will do the following:

1. Unzip Shell1.zip, creating a directory called Shell1/
2. Remove any previously unzipped copies of Shell1/
3. Execute various shell commands, to be added in the next few problems in this lab
4. Create a compressed version of Shell1/ called UnixShell1.tar.gz.
5. Remove any old copies of UnixShell1.tar.gz

Now, open the unixshell1.sh script in a text editor. Add commands to the script, within the section for Problem 1, to do the following:

- Change into the Shell1/ directory.
- Print a string (without using echo) telling you the directory you are currently working in.

Test your commands by running the script again and checking that it prints a string ending in the location Shell1/.

[a] If the necessary data files are not in your directory, cd one directory up by typing cd .. and type bash download_data.sh to download the data files for each lab.

Documentation and Help

When you encounter an unfamiliar command, the terminal has several tools that can help you understand what it does and how to use it. Most commands have manual pages, which give information about what the command does, the syntax required to use it, and different options to modify the command. To open the manual page for a command, type man <command>. Some commands also have an option called --help, which will print out information similar to what is contained in the manual page. To use this option, type <command> --help.

```
$ man ls
```

```
LS(1)                         User Commands                         LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).

       -a, --all
              do not ignore entries starting with .
```

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags

When you use `man`, you will see a list of options such as `-a`, `-A`, `--author`, etc. that modify how a command functions. These are called *flags*. You can use one flag on a command by typing `<command> -<flag>`, like `ls -a`, or combine multiple flags by typing `<command> -<flag1><flag2>`, etc. as in `ls -alt`.

For example, sometimes directories contain hidden files, which are files whose names begin with a dot character like `.bash`. The `ls` command, by default, does not list hidden files. Using the `-a` flag specifies that `ls` should not ignore hidden files. Find more common flags for `ls` in Table 1.1.

Flags	Description
`-a`	Do not ignore hidden files and folders
`-l`	List files and folders in long format
`-r`	Reverse order while sorting
`-R`	Print files and subdirectories recursively
`-s`	Print item name and size
`-S`	Sort by size
`-t`	Sort output by date modified

Table 1.1: Common flags of the `ls` command.

```
$ ls
file1.py  file2.py

$ ls -a
.  ..  file1.py  file2.py  .hiddenfile.py

$ ls -alt    # Multiple flags can be combined into one flag
total 8
```

```
drwxr-xr-x  2 c c 4096 Aug 14 10:08 .
-rw-r--r--  1 c c    0 Aug 14 10:08 .hiddenfile.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file2.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file1.py
drwxr-xr-x 38 c c 4096 Aug 14 10:08 ..
```

Problem 2. Within the script, add a command using `ls` to print one list of the contents of `Shell1/` with the following criteria:

- Include hidden files and folders
- List the files and folders in long format (include the permissions, date last modified, etc.)
- Sort the output by file size (largest files first)

Test your command by entering it into the terminal within `Shell1/` or by running the script and checking for the desired output.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

```
~$ cd Test/                 # navigate to test directory

~/Test$ ls                  # list contents of directory
file1.py

~/Test$ mkdir NewDirectory  # create a new empty directory

~/Test$ touch newfile.py    # create a new empty file

~/Test$ ls
file1.py  NewDirectory  newfile.py
```

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, use the `-r` flag to recursively copy files contained in the directory. If you try to copy a directory without the `-r`, the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second.

If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

```
~/Test$ ls
file1.py  NewDirectory  newfile.py

~/Test$ mv newfile.py NewDirectory/         # move file into directory
```

```
~/Test$ cp file1.py NewDirectory/          # make a copy of file1 in directory
~/Test$ cd NewDirectory/
~/Test/NewDirectory$ mv file1.py newname.py # rename file1.py
~/Test/NewDirectory$ ls
newfile.py  newname.py
```

When deleting files, use `rm <filename>`, and when deleting a directory, use `rm -r <dir_name>`. The `-r` flag tells the terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to print strings in the terminal describing what it is doing.

When your terminal gets too cluttered, use `clear` to clean it up.

```
~/Test/NewDirectory$ cd ..          # move one directory up
~/Test$ rm -rv NewDirectory/        # remove a directory and its contents
removed 'NewDirectory/newname.py'
removed 'NewDirectory/newfile.py'
removed directory 'NewDirectory/'

~/Test$ rm file1.py                 # remove a file
~/Test$ ls                          # directory is now empty
~/Test$
```

Commands	Description
`clear`	Clear the terminal screen
`cp file1 dir1`	Create a copy of `file1` and move it to `dir1/`
`cp file1 file2`	Create a copy of `file1` and name it `file2`
`cp -r dir1 dir2`	Create a copy of `dir1/` and all its contents into `dir2/`
`mkdir dir1`	Create a new directory named `dir1/`
`mkdir -p path/to/new/dir1`	Create `dir1/` and all intermediate directories
`mv file1 dir1`	Move `file1` to `dir1/`
`mv file1 file2`	Rename `file1` as `file2`
`rm file1`	Delete `file1` [`-i`, `-v`]
`rm -r dir1`	Delete `dir1/` and all items within `dir1/` [`-i`, `-v`]
`touch file1`	Create an empty file named `file1`

Table 1.2: File Manipulation Commands

Table 1.2 contains all the commands we have discussed so far. Commonly used flags for some commands are contained in square brackets; use `man` or `--help` to see what these mean.

Problem 3. Add commands to the `unixshell1.sh` script to make the following changes in `Shell1/`:

- Delete the `Audio/` directory along with all its contents
- Create `Documents/`, `Photos/`, and `Python/` directories

- Change the name of the `Random/` directory to `Files/`

Test your commands by running the script and then using `ls` within `Shell1/` to check that each directory was deleted, created, or changed correctly.

Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, you may need to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using *wildcards*. We will use the `*` and `?` wildcards. The `*` wildcard represents any string and the `?` wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```
$ ls
File1.txt   File2.txt   File3.jpg   text_files

$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt

$ ls
File3.jpg   text_files
```

See Table 1.3 for examples of common wildcard usage.

Command	Description
`*.txt`	All files that end with `.txt`.
`image*`	All files that have `image` as the first 5 characters.
`*py*`	All files that contain `py` in the name.
`doc*.txt`	All files of the form `doc1.txt`, `doc2.txt`, `docA.txt`, etc.

Table 1.3: Common uses for wildcards.

Problem 4. Within the `Shell1/` directory, there are many files. Add commands to the script to organize these files into directories using wildcards. Organize by completing the following:

- Move all the `.jpg` files to the `Photos/` directory
- Move all the `.txt` files to the `Documents/` directory
- Move all the `.py` files to the `Python/` directory

Working With Files

Searching the File System

There are two commands we can use for searching through our directories. The `find` command is used to find files or directories with a certain name; the `grep` command is used to find lines within files matching a certain string. When searching for a specific string, both commands allow wildcards within the string. You can use wildcards so that your search string matches a broader set of strings.

```
# Find all files or directories in Shell1/ called "final"
# -type f,d specifies to look for files and directories
# . specifies to look in the current directory

$ find . -name "final" -type f,d
$            # There are no files with the exact name "final" in Shell1/

$ find . -name "*final*" -type f,d
./Files/May/finals
./Files/May/finals/finalproject.py
```

```
# Find all within files in Documents/ containing "Mary"
# -r tells grep to search all files with Documents/
# -n tells grep to print out the line number (2)

$ Shell1$ grep -nr "Mary" Documents/
Documents/people.txt:2:female,Mary,31
```

Command	Description
`find dir1 -type f -name "word"`	Find all files in `dir1/` (and its subdirectories) called `word` (`-type f` is for files; `-type d` is for directories)
`grep "word" filename`	Find all occurrences of `word` within `filename`
`grep -nr "word" dir1`	Find all occurrences of `word` within the files inside `dir1/` (`-n` lists the line number; `-r` performs a recursive search)

Table 1.4: Commands using `find` and `grep`.

Table 1.4 contains basic syntax for using these two commands. There are many more variations of syntax for `grep` and `find`, however. You can use `man grep` and `man find` to explore other options for using these commands.

File Security and Permissions

A file has three levels of permissions associated with it: the permission to read the file, to write (modify) the file, and to execute the file. There are also three categories of people who are assigned permissions: the user (the owner), the group, and others.

You can check the permissions for `file1` using the command `ls -l <file1>`. Note that your output will differ from that printed below; this is purely an example.

```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
drw-rw-r-- 1 username groupname 373 Aug  5 21:16 Documents
-rwxr-x--x 1 username groupname  27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

The first character of each line denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The next nine characters denote the permissions associated with that file.

For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rwx`, tell us the owner can read, write, and execute the file. The next three characters, `r-x`, tell us members of the same group can read and execute the file, but not edit it. The final three characters, `--x`, tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are multiple notations used to modify permissions, but the easiest to use when we want to make small modifications to a file's permissions is *symbolic permissions* notation. See Table 1.5 for more examples of using symbolic permissions notation, as well as other useful commands for working with permissions.

```
$ ls -l script1.sh
total 0
-rw-r--r-- 1 c c 0 Aug 21 13:06 script1.sh

$ chmod u+x script1.sh      # add permission for user to execute
$ chmod o-r script1.sh      # remove permission for others to read
$ ls -l script1.sh
total 0
-rwxr----- 1 c c 0 Aug 21 13:06 script1.sh
```

Command	Description
`chmod u+x file1`	Add executing (`x`) permissions to user (`u`)
`chmod g-w file1`	Remove writing (`w`) permissions from group (`g`)
`chmod o-r file1`	Remove reading (`r`) permissions from other other users (`o`)
`chmod a+w file1`	Add writing permissions to everyone (`a`)
`chown`	change owner
`chgrp`	change group
`getfacl`	view all permissions of a file in a readable format.

Table 1.5: Symbolic permissions notation and other useful commands

Running Files

To run a file for which you have execution permissions, type the file name preceded by `./`.

```
$ ./hello.sh
bash: ./hello.sh: Permission denied

$ ls -l hello.sh
```

```
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh

$ chmod u+x hello.sh        # You can now execute the file

$ ./hello.sh
Hello World!
```

Problem 5. Within `Shell1/`, there is a script called `organize_photos.sh`. First, use `find` to locate the script. Once you know the file location, add commands to your script so that it completes the following tasks:

- Moves `organize_photos.sh` to `Scripts/`
- Adds executable permissions to the script for the user
- Runs the script

Test that the script has been executed by checking that additional files have been moved into the `Photos/` directory. Check that permissions have been updated on the script by using `ls -l`.

Accessing Remote Machines

At times you will find it useful to perform tasks on a remote computer or server, such as running a script that requires a large amount of computing power on a supercomputer or accessing a data file stored on another machine.

Secure Shell

Secure Shell (SSH) allows you to remotely access other computers or servers securely. SSH is a network protocol encrypted using public-key cryptography. It ensures that all communication between your computer and the remote server is secure and encrypted.

The system you are connecting to is called the *host*, and the system you are connecting from is called the *client*. The first time you connect to a host, you will receive a warning saying the authenticity of the host can't be established. This warning is a default, and appears when you are connecting to a host you have not connected to before. When asked if you would like to continue connecting, select yes.

When prompted for your password, type your password as normal and press enter. No characters will appear on the screen, but they are still being logged. Once the connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type exit.

```
alice@mycomputer:~$ ssh alice27@acme01.byu.edu

alice27@acme01.byu.edu password:# Type password as normal
last login 7 Sept 11
```

```
[alice27@byu.local@acme01 ~]$ ls    # Commands are executed on the host
myacmeshare/

[alice27@byu.local@acme01 ~]$ exit  # End a secure connection
logout
Connection to acme01.byu.edu closed.

alice@mycomputer:~$                 # Commands are executed on the client
```

Secure Copy

To copy files from one computer to another, you can use the Unix command `scp`, which stands for **s**ecure **c**opy **p**rotocol. The syntax for `scp` is essentially the same as the syntax for `cp`.

To copy a file from your computer to a specific location on on a remote machine, use the syntax `scp <file1> <user@remote_host:file_path>`. As with `cp`, to copy a directory and all of its contents, use the `-r` flag.

```
# Make copies of file1 and dir2 in the home directory on acme01.byu.edu
alice@mycomputer:~$ scp file1 alice27@acme01.byu.edu:~/
alice@mycomputer:~$ scp -r dir1/dir2 alice27@acme01.byu.edu:~/
```

Use the syntax `scp -r <user@remote_host:file_path/dir1> <file_path>` to copy `dir1` from a remote machine to the location specified by `file_path` on your current machine.

```
# Make a local copy of dir1 (from acme01.byu.edu) in the home directory
alice@mycomputer:~$ scp -r alice27@acme01.byu.edu:~/dir1  ~
```

Commands	Description
`ssh username@remote_host`	Establish a secure connection with `remote_host`
`scp file1 user@remote_host:file_path/`	Create a copy of `file1` on host
`scp -r dir1 user@remote_host:file_path/`	Create a copy of `dir1` and its contents on host
`scp user@remote_host:file_path/file1 file_path2`	Create a local copy of file on client

Table 1.6: Basic syntax for `ssh` and `scp`.

Problem 6. On a computer with the host name `acme20.byu.edu` or `acme21.byu.edu`, there is a file called `img_649.jpg`. Secure copy this file to your `UnixShell1/` directory. (Do not add the `scp` command to the script).

To `ssh` or `scp` on this computer, your username is your Net ID, and your password is your typical Net ID password. To use `scp` or `ssh` for this computer, you will have to be on campus using BYU Wifi.

Hint: To use `scp`, you will need to know the location of the file on the remote computer. Consider using `ssh` to access the machine and using `find`. The file is located somewhere in the directory `/sshlab`. Sometimes after logging onto the machine with `ssh`, there will appear to not be any directories you can access. Using the command `cd /` will fix this. When logging on initially, you also may get a message about not having a `myacmeshare`; this is not needed for this lab and the message may be ignored safely.

After secure copying, add a command to your script to copy the file from `UnixShell1/` into the directory `Shell1/Photos/`. (Make sure to leave a copy of the file in `UnixShell1/`, otherwise the file will be deleted when you run the script again.)

Git

Git is a version control system, meaning that it keeps a record of changes in a file. Git also facilitates collaboration between people working on the same code. It does both these things by managing updates between an online code repository and copies of the repository, called *clones*, stored locally on computers.

We will be using git to submit labs and return feedback on those labs. If git is not already installed on your computer, download it at `http://git-scm.com/downloads`.

Using Git

Git manages the history of a file system through *commits*, or checkpoints. Each time a new commit is added to the online repository, a checkpoint is created so that if need be, you can use or look back at an older version of the repository. You can use `git log` to see a list of previous commits. You can also use `git status` to see the files that have been changed in your local repository since the last commit.

Before making your own changes, you'll want to add any commits from other clones into your local repository. To do this, use the command `git pull origin master`.

Once you have made changes and want to make a new commit, there are normally three steps. To save these changes to the online repository, first add the changed files to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. If you want to add all changes that you have made to tracked files (files that are already included in the online repository), use `git add -u`.

Next, save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

Finally, add the changes in this commit to the online repository with `git push origin master`.

```
$ cd MyDirectory/                # Navigate into a cloned repository
$ git pull origin master         # Pull new commits from online repository

### Make changes to file1.py ###

$ git add file1.py               # Add file to staging area
$ git commit -m "Made changes"   # Commit changes in staging area
$ git push origin master         # Push changes to online repository
```

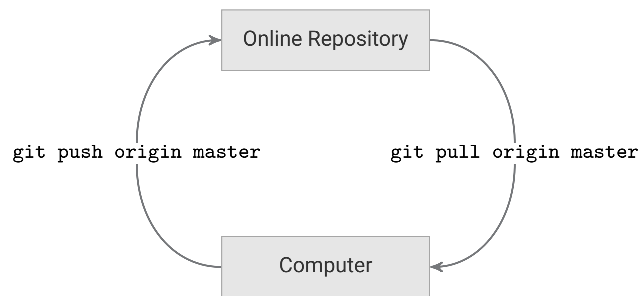

Figure 1.1: Exchanging git commits between the repository and a local clone.

Merge Conflicts

Git maintains order by raising an alert when changes are made to the same file in different clones and neither clone contains the changes made in the other. This is called a *merge conflict*, which happens when someone else has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have.

Achtung!

When pulling updates with `git pull origin master`, your terminal may sometimes display the following merge conflict message.

```
Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
```

This screen, displayed in `vim` (`https://en.wikipedia.org/wiki/Vim_(text_editor)`), is asking you to enter a message to create a *merge commit* that will reconcile both changes. If you do not enter a message, a default message is used. To close this screen and create the merge commit with the default message, type `:wq` (the characters will appear in the bottom left corner of the terminal) and press `enter`.

Note

`Vim` is a terminal text editor available on essentially any computer you will use. When working with remote machines through `ssh`, vim is often the only text editor available to use. To exit vim, press `esc:wq` To learn more about vim, visit the official documentation at `https://vimhelp.org`.

Command	Explanation
`git status`	Display the staging area and untracked changes.
`git pull origin master`	Pull changes from the online repository.
`git push origin master`	Push changes to the online repository.
`git add <filename(s)>`	Add a file or files to the staging area.
`git add -u`	Add all modified, tracked files to the staging area.
`git commit -m "<message>"`	Save the changes in the staging area with a given message.
`git checkout <filename>`	Revert changes to an unstaged file since the last commit.
`git reset HEAD <filename>`	Remove a file from the staging area, but keep changes.
`git diff <filename>`	See the changes to an unstaged file since the last commit.
`git diff --cached <filename>`	See the changes to a staged file since the last commit.
`git config --local <option>`	Record your credentials (`user.name`, `user.email`, etc.).

Table 1.7: Common git commands.

Problem 7. Using git commands, push `unixshell1.sh` and `UnixShell1.tar.gz` to your online git repository. Do not add anything else in the `UnixShell1/` directory to the online repository.