An Abridged Introduction to the UNIX Command Line

CSCI Student Mentors – Western Washington University

Contents

Revision History

Version 1.0 from 4/24/16

Authors: Brandon Tarquinio(BT), Robin Cosbey(RC) Ted Weber (TW), Sarah Gunderson (SG), Serena Bowen (SB)

Revision	Date	Author(s)	Description
1.0	4/24/16	TW	created
1.1	4/27/16	TW, SG, SB	revisions

Chapter 1

The Basics

1.1 Introduction

In this workshop we will run through the basics of using *bash* or some other Unix-like command line environment. Bash is one of many different *shells* (it is the default on most Linux distros and OSX), but most things we discuss will be universal to all shells for Unix-like computers.[1]

1.2 What is a shell?

In general, a shell is a user interface for accessing the operating system's services. This includes command line interfaces (like *bash*), as well as graphical shells that provide graphical user interface built on top of a windowing system.[2]

[1] For example Linux, OSX, FreeBSD. Also Cygwin provides a Unix-like environment for Windows computers.

[2] More information about shells here `https://en.wikipedia.org/wiki/Shell_(computing)` and the history of shells here `https://www.ibm.com/developerworks/library/l-linux-shells/`

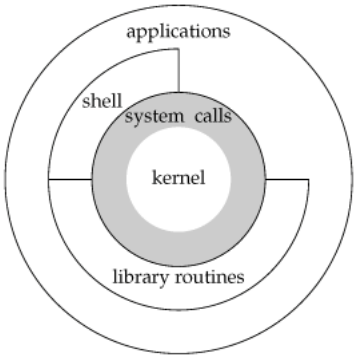

Figure 1.1: The shell sits above the kernel, which is generally the lowest-level, or "innermost" component of most operating systems.

1.3 Getting started

We will be focusing on using a command line interface shell, *bash* in particular. We will discuss how to manipulate files and directories, use the built-in manual pages, compile/execute programs, and build your command repertoire in general. Let's start by opening the terminal:[3]

```
ctrl + alt + t
```

1.4 Running a Command

The general form of any command, for the most part, is as follows:

```
$ command_name -arg1 ... --longerargs ... input1 input2 ...
```

- **$** is what's known as your command *prompt*, and it is displayed on the console after every command has finished executing.
- **command_name** refers to the name of the user command.
- **-arg** is an example of a *short-option*, which is a single dash followed by a single character.
- **--longerargs** is an example of a multi-character *long-option*, and these are preceded by two dashed. *Long and short-options are used inform the user command program to perform or not to perform additional tasks.*
- **input1** and **input2** represent additional arguments, or parameters, that the user command may allow.

[3]Most Linux desktop environments use this keyboard shortcut by default, but it does vary. If this shortcut doesn't work you can use the search functionality in your desktop environment and open the shell from there.

- The "..." above are used to denote the fact that we can have multiple arguments and options for a single user command.[4]

Lets look at an example:

```
echo hello world
```

echo is a simple command that displays a line of text. Now let's try adding an optional short argument.

```
echo -n hello world
```

the **-n** was interpreted by the shell as a *short-option* telling the shell to refrain from putting a newline character (\n) at the end of the input string.

1.5 Directories and Moving Around

1.5.1 Directory trees and pwd

We will start by learning some commands to move around the directory tree. It is called a directory tree because it can be visualized as follows:

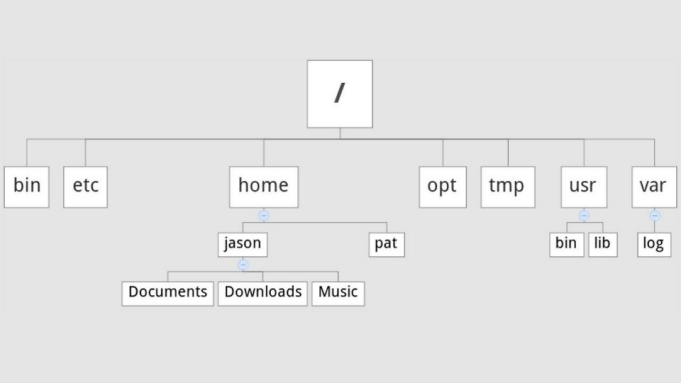

Figure 1.2: An Example Linux Directory Tree.

Each of the boxes in Figure 1 are directories and we call the directory "/" *root* since it is the root of our tree. In a command line enter:

```
pwd
```

This will **p**rint your current **w**orking **d**irectory. For example if we were in the home directory of the user *jason* and typed **pwd** we would get the following:

```
/home/jason
```

[4]In actuality every command has a finite number of options it understands and you normally use a small subset of those. There might also be a limit on the number of arguments allowed.

A user's *home directory* is the default current working directory when you open up a terminal. You can think of it as the root of your personal directory tree, different than the root of the entire file system.

Exercise 1.1. *Lets explore the concept of home by running the following commands:*

```
pwd
```

```
echo ~
```

What did you find? Note that "~", the tilde symbol, is normally located near the top-left of your keyboard. Remember "~", since it will be used a lot as a shortcut to your home directory.

1.5.2 Listing the contents of a Directory

To **lis**t all the files in your current working directory enter:

```
ls
```

We can also specify to **ls** what directory we want to list the contents of with the following generic form:

```
ls dirname
```

For example:

```
ls ~
```

will display the contents in your home directory.

To display more information about the files, use the **-l flag**:

```
ls -l
```

Example output:

```
-rw-r–r– 1 webert3 grp.csci.Students 0 Apr 27 12:26 file1
```

```
drwxr-xr-x 4 webert3 grp.csci.Students 4 Apr 25 14:11 java_programs
```

There is a lot more information here! So what does it all mean? Lets use the file listing for the directory *java_programs* as an example:

1	2	3	4	5	6	7	8
d	rwxr-xr-x	4	webert3	grp.csci.Students	4	Apr 25 14:11	java_programs

1. File type. For example, **-** for a file, **d** for a directory, or **l** for a link.
2. File permissions.
3. Number of hardlinks to this file.
4. Owner of the file.
5. Group that the file belongs to.
6. The size of the file in bytes.

7. The date in which the file was last modified.
8. The name of the file.

We won't go into detail on the specific meaning of these fields in this workshop, but there is a link in the footnotes for more information.[5]

1.5.3 Changing your Current Working Directory

Your *current working directory* is the directory your process is currently associated with (i.e. the directory your are currently operating inside of). We can **c**hange the **d**irectory with **cd** which has the following form:

```
cd dirToChangeTo
```

For example we can run:

```
cd /
```

And then:

```
pwd
```

and we will see:

```
/
```

To change back to our previous working directory with this shortcut:

```
cd -
```

To change to our home directory regardless of where we are we can simply type:

```
cd
```

The default directory **cd** will change to is the user's home directory.
Lastly, to move back one directory in the directory tree enter this command:

```
cd ../
```

Exercise 1.2. *Use **cd**, **pwd**, **ls** to explore the directory tree a bit. Becoming familiar and quick with these commands is essential to becoming efficient with the command line. **cd** and **ls** are easily the two most used commands in Unix-like shells.*

1.5.4 Relative vs Absolute Path Names

We need to learn one more important concept about UNIX files and directories before moving on.
Suppose we are in the directory */home* in our directory tree and we wanted to list the contents of the jason directory. We can do this in two ways:

[5] Advanced bash workshop: `https://github.com/wwu-mentors/bash/tree/master/Bash_II`

```
ls jason
```

or

```
ls /home/jason
```

In the latter we used the **absolute path name** which is the name of the file or directory prefixed by the path from "/" to it. When we use absolute path names it no longer matters what are current working directory is (e.g. we could have run the second command from any directory). In contrast, the first command will only list the contents of /home/jason if are current working directory is /home. This is because jason is a **relative path name** since we assume the path begins in our current working directory. Relative path names allows the user to remember and type less but require careful consideration of what your working directory currently is.

1.5.5 Special Directory Entries and Hidden Files

Hidden files and directories are generally configuration files that are not shown unless you specify that you would like to see them. These entries are hidden to de-clutter the file system. Now we will review two important hidden directories that are useful shortcuts.

- ".": This represents the current directory and is an actual entry in every directory.
- "..": This represents the parent of the current directory and is also an actual entry in the current directory.

Exercise 1.3. *Try the following commands:*

```
ls -a
```

```
ls -a .
```

To view all files in your home directory:

```
ls -a ~
```

Does the output make sense? As a challenge question what will the following command do? (Try to think it out before running it!):

```
ls ./../../
```

1.6 Grab the Demo Files from GitHub!

Take a moment to download a few files for this workshop from our GitHub. Each lab machine has the version control system Git installed, so if you enter the following command you will download our demo files into your current working directory.

```
git clone https://github.com/wwu-mentors/bash.git
```

Then enter the following command to change your current working directory to the directory pertaining to our demo.

```
cd bash/Bash_Combined/demo
```

The files you download from GitHub also include a couple .pdf documents with the write-ups for previous bash workshops run by the CS mentors. Take a look at those files if you are curious!

1.6.1 Making a Directory

Now that we understand the directory tree and how to move around it we are ready to learn how to add to it. The general form of the command to **ma**k**e** a **dir**ectory is:

```
mkdir newdirname1 newdirname2 ...
```

Exercise 1.4. *Run the following sequence of commands. Before the second and third calls to* ***ls*** *think about what you would expect to see:*

```
ls
```

```
mkdir FirstDir SecondDir
```

```
ls
```

```
mkdir FirstDir/Foo FirstDir/Foo2
```

```
ls FirstDir
```

clear *is a useful command that clears the screen.*

1.6.2 Creating a File

Unix is very flexible regarding file names. A valid Unix file name can contain any other character than "\". Also, Unix does not add any special significance to characters like "." or "_" in file names; they are treated like any other character. You can create an empty file using the command **touch**.

Exercise 1.5. *In this exercise you will create an empty hidden file.*

1. *Create a file called* ***.f1****.*
2. *Try finding the file using* ***ls****.*

1.6.3 Deleting Files & Directories

The program to **re**m**o**ve a file or directory is **rm**, and it has the following form:

```
rm [OPTION] ... [FILE]...
```

IMPORTANT NOTE: With great power comes great responsibility. This command will remove a file from your file system permanently. THIS COMMAND DOES NOT PUT FILES INTO A TRASH/RECYCLING BIN. There may be some way to recover files after

using **rm**, but it will not be easy, especially for an inexperienced user.[6] Be cautious!

Also, by default this command only works if the directories you are trying to delete are **empty**. An empty directory is a directory which only has no other entries but "." and "..".

If you would like to delete a directory and all of it's contents, use the **-r** flag.

```
rm -r FirstDir
```

This will silently remove FirstDir/Foo, FirstDir/myfile, and FirstDir/Foo2 forever. **AGAIN**, this is a powerful tool, don't blow away your entire file system but running this command on your home directory![7]

If you want to play it safe, use the **-i** flag and a prompt will come up before removal.

```
rm -i file1
```

1.7 Basic Shortcuts

1.7.1 clear

Is your terminal covered in lines of code and difficult to determine where one previous command ends and another begins? Typing "clear" into the terminal will wipe away all of the old commands (command shown below).

```
clear
```

It is possible to refer back to those deleted commands by scrolling up in the terminal.

1.7.2 ctrl ($\wedge$)

Sometimes within bash ctrl will be denoted by " $\wedge$ ". Ctrl is included with several commands, one of the more relatable being ctrl-C. This will terminate the process.

1.7.3 tab completion

When we enter tab at the prompt it will try to finish whatever we are currently typing as long as there are no ambiguity. For example if we only had "**file1**" in our current working directory we could view the file listing by tying "**ls f**" and hitting tab. It will fill out the rest of the word so that you now see:

```
ls file1
```

If instead we had both "file1" and "file2" then when we pressed tab it would have only filled in this much:

[6] `https://unix.stackexchange.com/questions/101237/how-to-recover-files-i-deleted-now-by-running-rm`
[7] The WWU CS Mentors are not liable for any misuse of this command. Sorry folks!

```
ls file
```

This is because it does not know if you want a 1 or a 2 to follow next. To see all option press tab twice and the list of all options will be displayed on the screen. You can test out this functionality in the directory *demo/dirs*.

Exercise 1.6. *Tab completion can also be used to complete program names.*

- *Type*

```
pyt
```

- *Use tab completion to complete the command "python" and launch an interactive python shell.*
- *Exit the python shell by typing*

```
exit()
```

1.7.4 Wildcard expansion (*)

The ***** symbol is known as a wildcard. This is a useful tool in various situations, but is commonly used to perform actions on more than one file at a time, or to find part of a phrase in a text file.[8]

Lets give it a try by navigating to the directory *demo/dirs* and typing this command:

```
ls -l f*
```

You should see the extended file listings for all three files!

1.8 Editing Files

In Computer Science courses we will use files for a lot of reasons:

- Your program source code: Whether you are taking a class that uses Python, Java, C, or some other language, you will need a file that is your program.
- Programs themselves: In Unix, all of the programs, the ones you write and the ones that are installed, are themselves a file. For example type the following commands:

```
which ls
```

```
which python
```

```
which which
```

  which is a command that tells you where the program is located on the computer.
- Plain text files: For example, input to or output from your program will generally be stored in text files.

[8] Standard wildcards are also known as globbing patterns. For more information see **man 7 glob**.

1.8.1 Text Editors

There are several pure command line editors like *vim* and *emacs*, but we will refrain from using these in our demo because there is some overhead to learn these tools. Instead we can use the command line to open up one of our favorite gui[9] text editors with commands like:

```
gedit myfile
```

or

```
atom myfile
```

Note that when you run one of the above commands it may open a gui window and you will no longer be able to use the command line until you close the newly opened window. You can get around this by doing the following:

```
gedit myfile &
```

This will open gedit (or whatever program you choose) in the *background* and allow you to continue using the command line.

Exercise 1.7. *Create a file and write to it.*

1. *Open up a gui text editor, or command line editor if you have experience.*
2. *Write some characters in the file.*
3. *Save your changes (i.e. write your data to the disk).*
4. *Find out how large your file is in bytes. Do you remember what command you could use?*

1.9 More on Files

1.9.1 Reading Files

There are many ways to read files; we will discuss a few now. For starters you can always use your text editor to read a file. If you don't want to edit a file you could use a viewer such as **less** or **more** instead. For example, to open the file $myfile$ with **less** type:

```
less myfile
```

This will open the file full screen in the command line and allow you to move up and down the document line by line. You can do this by pressing the up and down arrow keys. For those familar with Vim you can use Vim's way of moving around in less (which is why less is so cool!). Also some terminals are set up so you can use the scroll wheel on your mouse.

more is used in a similar fashion:

```
more myfile
```

[9] gui stands for **G**raphical **U**ser **I**nterface

This program is more primitive and only moves one page at a time. Hit space to move to the next page until you are done or press **q** to quit at any time.[10] So yes, **less** actually has *more* functionality than the command **more**.

Some other ways of viewing include **cat**, **head**, and **tail**. cat's main purpose is for con**cat**ination but it can also be used to dump a file to the screen by typing:

```
cat myfile
```

This is great for small files but horrible for larger ones. **head** and **tail** will grab the first or last 10 lines, respectively.[11].

Exercise 1.8. *Try running these commands on the file "numbers" in the directory "demo"*

```
cat numbers
```

```
more numbers
```

```
less numbers
```

1.9.2 Copying a File

We can **co**p**y** a file by **cp** command which has the following generic form:

```
cp sourceFileName destinationFileName
```

For example:

```
cp myfile myfile.backup
```

will make it so the file name *myfile* in my current working directory is copied into a new file called *myfile.backup* which will also be in my current working directory.

1.9.3 Moving a File

We can **mo**v**e** a file by the **mv** command which has the following generic form:

```
mv sourceFileName distinationFileName
```

Exercise 1.9. *Copy a file, then move the copy elsewhere.*

1. *Create a file using **touch**.*
2. *Make a copy of the file using **cp**.*
3. *Use **mv** to move the copied file to a new file within the same directory. That is, run a command of the following form:*

```
mv file.copy file.newfile
```

*Tip: Run **ls** before and after the command and observe the result.*

[10] Pressing **q** to quit is also used in the command **less**.

[11] These commands has several other options as well allowing you to grab a variable number of lines. See **man 1 tail**

1.9.4 Rename?

There is no rename command since the effect of such a command can already be achieved by something we already learned.

Exercise 1.10. *Try renaming a file using one of the commands we've already learned.* ***SPOILER****: We renamed a file in the previous exercise!*

1.10 Manuals

In this section we will figure out how to learn new things about the command line without ever having to leave it! This includes how to learn more about commands we already know and how to find new commands.

1.10.1 A help message

Many Unix programs will use one or both of the following to give you a brief help message:

```
program_name -h
```

or

```
program_name --help
```

This is optional so some programs will have both, one, or neither of the above arguments. Also some programs that have both will have different messages for each. Note that some programs will also use -h to do something other then give a help message.

Try them with **vim**:

```
vim -h
```

```
vim --help
```

Now try them with **ls** and **which**.

We have learned that -h and –help sometimes give brief help messages but we also found flaws with consistency. A more consistent system with verbose messages are the programs **man** and **info**.

1.10.2 Man Pages

The **man** program gives you **man**ual pages about what every program you specify. Its general form is:

```
man program_name
```

For example:

```
man cat
```

will give you information on the program **cat** which we have seen a few times now. You can even learn more about **man** by running the following:

```
man man
```

The interface for reading these pages is the same as **less** so this should feel familiar by now. For those who use C note that you the system calls and libc functions have manual pages in section 2 and 3 of the man pages! There are nine sections to the man pages and sometimes two things with the same name will exist in different sections. For example:

```
man printf
```

will open a manual page for a program called **printf** which is not an important command to learn but merely a good example. We can see that this is from the first section due to the top left corner having this:

```
PRINTF(1)
```

The number in parenthesis is the section the page is in. Thus we get the same page if we instead type:

```
man 1 printf
```

Now those who have ever written a line of **C** will no that this is the name of a library function to print text to the screen. To view the page for that function enter the following:

```
man 3 printf
```

See how it is a different page? The manual is generally split into eight number sections:

1	User commands
2	System calls
3	Library functions
4	Special files
5	File formats & conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

We have been mostly dealing with *user commands*, but if you were writing a **C** program you would likely be visiting section 2 and section 3 quite frequently.

Exercise 1.11. *Using the man pages.*

1. *Try opening the man page for a command we have previously used.*
2. *Try using this command with one of the listed long or short-options.*

1.10.3 Info Pages

man pages are the standard way of learning how use software on Unix systems and it cannot be stressed enough how important it is to learn how to use. As stated before **man** uses the **less** program to view the pages and thus it has a way of interacting that is similar to **vim**. There is a newer program and set of **info**rmational pages called **info** which instead uses the

text editor **emacs** style of interacting. There are some programs that only have both **man** and **info** pages, only one, or neither.
View an info page by running:

```
info program_name
```

You will get a full screen viewer like **man** but arrow keys and vim movements wont work! You can move forward pages with "ctrl+v" and backward pages with "alt+v" to view pages similar to **more**. You can move line by line by using "ctrl+n" and "ctrl+p" to move to the **n**ext and **p**revious lines respectively. You can leave by typing "q" just like we did with **more**, **less**, and **man**.

Exercise 1.12. *View the info page of some of the commands we have learned so far. Are there any differences?*

Just like Vim, Emacs is a very powerful editor with a sharp learning curve that is loved by many programmers. We will offer workshops on both Emacs and Vim this year and encourage everyone to learn a bit about both editors.

1.10.4 Programs to help you find commands: apropos and whatis

We can use the **whatis** program to get quick descriptions about other program. The general form is:

```
whatis programname
```

For example we can type:

```
whatis ls
```

and see:

```
ls (1) - list directory contents
```

The one should be familiar, it is the section of the man page that **ls** is in. If all of these descriptions are in the computer it would be cool to search for commands like you would use google to search for webpages. That is what **apropos** is for, it has the following general form:

```
apropos whatToSearchFor
```

For example if we wanted to find other editors to use we could try:

```
apropos editor
```

This will produce a list of every command whose name either contains "editor" or whose **whatis** description contains "editor". Clearly this will not always contain the results you want and therefor you can try different searches to refine or expand your results such as:

```
apropos text editor
```

Also note that you can group words with quotation marks:

```
apropos "text editor"
```

Will only return results that have *text* followed by *editor* in the **whatis** description.

1.10.5 When you just want to use your default

The user command **xdg-open** will open a file or URL in the user's preferred application. For example:

```
xdg-open 'https://www.google.com'
```

This will open `www.google.com` in your default web browser.

1.11 Running Your Programs

In this section we will show how to use the command line to run the programs we write. Since no assumptions are made as to how far you are in Western's Computer Science Department's curriculum we will show how to run programs in Python & Java. All code will be provided so you can follow along without needing to know the language. For each language we will run the iconic Hello World program.

1.11.1 Python

Change your current working directory to *demo/python_programs*, which should be already created for this demo.

Use a text editor to create and edit the following file called "HelloWorld.py". For example:

```
atom HelloWorld.py
```

Write this in your file:

```
print("Hello World!")
```

and save the file. Now on the command line enter the following to run your program:

```
python HelloWorld.py
```

This will open the python interpreter with the file "HelloWorld.py" running on it. You should see the following output:

```
Hello World!
```

1.11.2 Java

Change your current working directory to somewhere you would like to save your java programs. For our demo you could do this in the directory *java_programs*.

Create a file called "HelloWorld.java" using your favorite text editor.:

```
[some text editor] HelloWorld.java
```

Write the following:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

and save the file. Now compile the program with the following:

```
javac HelloWorld.java
```

If errors occurred during compilation they would be displayed during the last step. You would need to fix the lines in the file causing the errors and rerun the previous line until your program compiles correctly. If the program compiles without errors you should see "HelloWorld.class" when you enter:

```
ls
```

Now we can run the program with the following command:

```
java HelloWorld
```

Which, as you might expect, will display the following:

```
Hello World!
```

Exercise 1.13. *To see example of how errors will be displayed edit "HelloWorld.java" to no longer have class on the first line. Recompile with the* ***javac*** *command we just learned. See how the top line that is returned informs us that java was expecting the world class but couldn't find it?*

1.12 More Shortcuts

There are a number of shortcuts that save typing time and allow you to remember less. We will discuss a few now.

1.12.1 Reuse previous commands

You will notice the more you use the command line that there the same sequence of commands are used a lot. For example when you are writing a program you will edit it using some text editor, and then compile or run the program. You will then notice a bug and open the editor again and then test your changes by compiling or running your program again. Such as:

```
vim HelloWorld.c
```

```
gcc HelloWorld.c
```

```
./a.out
```

```
vim HelloWorld.c
```

And this will cycle will keep going. Wouldn't it be nice to not have to keep retyping these same commands? There are a few tools for this and we will discuss two of them.

The simplest is to use the **up and down keys** to look through your history of commands. By pressing up once you will see the last command you entered. You can continue hitting up to see later and later commands. In our example instead of retyping "vim HelloWorld.c" we could have hit up three times at the prompt.

The only issue with this method is sometimes the command you want is far back in your history and the amount of time to press up till you find it is longer than if you would have just typed it again! This is a common mistake even by experienced programmers! A solution to this is the following:

```
!!
```

will also run the last command and is equivalent to pressing up once. More importantly though is this feature:

```
!str
```

Will find the last command you entered that began with str. In our previous example we could have entered:

```
!v
```

which will run:

```
vim HelloWorld.c
```

since it was the last command we entered that started with a "v". Sometimes you will need to use more than one character such as this situation:

```
vim HelloWorld.c
```

```
view HelloWorld.c
```

```
!v
```

In this situation, the command **view HelloWorld.c** would be executed instead of **vim HelloWorld.c**.[12]

1.12.2 reverse search

Another way to go back to previously entered commands is by pressing "**Ctrl + r**" while at the command line prompt. This will search for any past commands you've entered matching the string that you type in.
As you type, you will see the first command that matches appear. If that is not the command you are looking for, you can either continue typing in hopes that another previous command will appear or press "**Ctrl + r**" again to move to the next matching command in history.

Exercise 1.14. *Try it out: Press "Ctrl+r" and then start typing "cd" to see the last change directory command you entered. To search through all of your past change directory commands, continue pressing "Ctrl+r".*

1.12.3 Running commands sequentially in one line

Say I wanted to navigate to a particular directory where I store my C files and open a file in a text editor. I would need to use the command **cd** to change my current working directory followed by another command to open the file I want. We can do this all in one line using the **;** token!

```
cd demo/c_programs ; vim HelloWorld.c
```

Now when I'm done editing the file I'll be right where I want to be to compile and run the executable. Pretty neat.

[12] **view** opens files in **vim** in read-only mode. See **man view** or **man vim** for more information

Chapter 2

Slightly More Advanced Topics

2.1 Redirection and Pipes

Now that we have a solid repertoire of commands we are ready to learn about redirection and pipes. Redirection allows us to change (or *redirect*) the input and output files to any program. Pipes allow us to chain programs together. This is where we start to see the true power of using the command line. First let's learn a little bit about file descriptors.

2.1.1 Files in UNIX

During the course of this workshop we have referred to programs that have their default modes set to print to the console or get input from the keyboard. This has been an over-simplification which we need to now elaborate on.

In a UNIX or UNIX-like environment, everything is a file.[1] This includes a wide range of input/output resources including keyboards, printers, and even some channels for inter-process and network communications. This is handy because it allows the same set of tools, utilities, and APIs to be used on a wide variety of resources.

2.1.2 File Descriptors: stdin, stdout, stderr

Each open file in a process is represented by a non-negative integer known as a *file descriptor*. There are three file descriptors open by default for each process, 0, 1, and 2, which correspond to **stdin**, **stdout**, and **stderr**, respectively. By default, stdout and stderr are set to write to the console, and stdin is set to read input from the keyboard. Figure 2.1 illustrates this concept:

[1] Well, it might be more accurate to say that everything is a file descriptor... `https://en.wikipedia.org/wiki/File_descriptor`

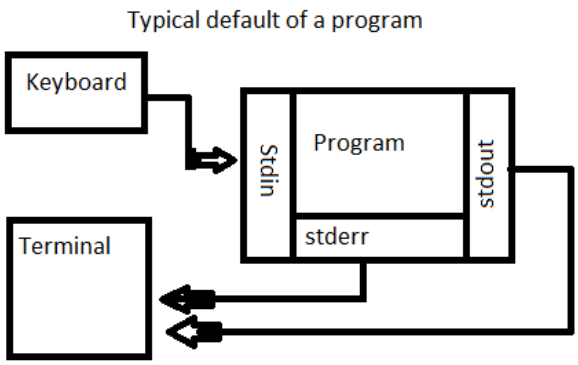

Figure 2.1: Default configuration.

2.1.3 Redirection

File redirection is telling the shell to set stdin, stdout, or stderr to something other than their default (i.e. some other file). This is achieved by using the '>' and '<' characters as part of your command. For example, a common use would be to redirect a program's output to a file when the program typically writes to stdout:

```
java HelloWold.java > output.txt
```

Here is a table with some more examples:

program < file1	stdin is set to file1.
program > file1	stdout is set to file1. If file1 does not exist it is created. If it does exist it is overwritten.
program >> file1	Same as above but appends to the file instead of overwriting.
program 2 > file1	Same as > but for stderr.
program 2 >> file1	Same as >> but for stderr.

Exercise 2.1. *Let's redirect stdout to another file.*

1. *Navigate to demo/java_programs/large_output*
2. *Compile and run this program once without redirection.*
3. *Now redirect stdout to a file called 'output'. NOTE: Using the redirection character will create and open this file for you!*

2.2 grep

grep is used to find occurrences of anything that matches a given regular expression in a file. A regular expression, put simply, is a sequence of characters that define a search pattern.

Wildcard expansion[2], for example, uses a regular expression to match file names.

The general form of **grep** is:

```
grep options regularExpression file1 file2 ...
```

The most common options are:

-r	recursive
-i	case insensitive
-n	line numbers

grep is a very useful tool and has a variety of applications in the UNIX environment. Let's give it a try:

Exercise 2.2. *Find the word 'hello' in the output of LargeOutput.java*

1. *Compile and run LargeOutput.java and redirect it's output to a file called 'output'.*
2. *Use grep to search for the pattern 'hello'. Try writing the search expression in a variety of ways to obtain the same result.*
3. *Use the command options to find out the line numbers that 'hello' is printed on.*

2.3 Pipes

Pipes are similar to redirection except that we are changing stdout of one program to now be stdin of another. We can string together as many programs as we would like. Here is the general form:

```
prog1 | prog2 | prog3 ... | progN
```

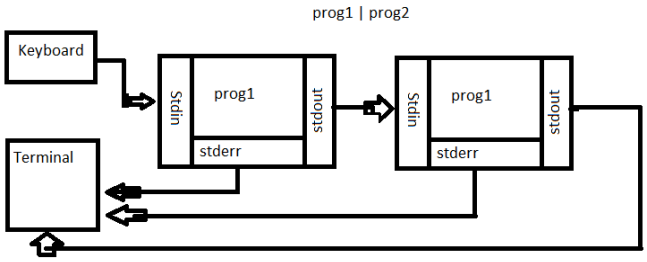

Figure 2.2: The output of prog1 is being piped to the stdin of prog2.

Pipes also have a wide variety of applications in the UNIX environment, but let's take a look at some concrete examples.

[2]An example of wildcard expansion can be seen on page 12

Suppose I wanted to view the file listings of a directory, but the directory happens to contain a lot of files. We could pipe the output of **ls** to another handy program **less**, and we could view the files in a more reasonable fashion without flooding our console with output.

```
ls -al ~ | less
```

This is very helpful when using shells with no scrolling feature.

An extremely powerful command to use with pipes is one we just learned about, **grep**. We can pipe the output of our programs into grep and search for patterns. Say I had written a command awhile back that was rather long (e.g. using **ssh**[3] to log into another machine remotely). To avoid having to type it in again, and potentially make mistakes in the process, we can use bash's builtin command[4] **history**. By default this command will output entries in our shell's history file, which contains several commands we had previously entered.

```
history | grep [search expression]
```

2.4 Unlock the true power of your shell: Configuring your .bashrc file.

.bashrc is a ordinary file with extraordinary capabilities. It's a shell script that Bash runs whenever it is started interactively[5]. This allows us to run a bunch of useful commands right when the shell boots up. A useful builtin command commonly found in .bashrc files is **alias**.

```
alias [OPTIONS]  [name[=value]...]
```

This command allows a string to be substituted for a word when it is used as the first word of a simple command. If we use this command in our .bashrc file we can ensure that these shortcuts exist whenever we open a new shell!

You can print out all of your aliases using the **-p** option. You can also unset aliases using the **unalias** command.

2.4.1 Creating a .bashrc file

Navigate to your home directory and create a new file **.bashrc**. Now open up a text editor and start writing out the commands you want to run when the shell boots up.

2.4.2 Configuring your .bashrc file

This is where you can be as creative as you would like. If you want to create a bunch of aliases for long/tedious command sequences, that's a great start. If you want to change the

[3] We will be doing another workshop on this tool and other related tools, but if you're eager to try it out see **man 1 ssh**.

[4] Builtin commands are executed directly in the shell itself, instead of an external executable program.

[5] Difference between interactive and non-interactive sessions `https://unix.stackexchange.com/questions/50665/what-is-the-difference-between-interactive-shells-login-shells-non-login-shell`

color of your prompt, or display a calendar when you boot up bash, this is possible as well. We will take a look at an example .bashrc file to get you started, but there are all sorts of interesting ways you can configure your .bashrc file to enhance your command-line interface. Here are a few links that you may find helpful:

example .bashrc files:
`http://crunchbang.org/forums/viewtopic.php?id=1093`
`http://tldp.org/LDP/abs/html/sample-bashrc.html`

Tutorials on bash scripting:
`http://ryanstutorials.net/bash-scripting-tutorial/bash-script.php`
`http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`

2.5 Retrospective and Closing Comments

If you understand all the content that we just covered then you can now do all of your programming and file managing from the command line. Try to use the command line on your next few projects to master these skills. We have barely scratched the surface, however, and we encourage you to use the **man** pages and your favorite search engine to look into the tools and topics that we just outlined. Also note that you will learn a lot from friends, teachers, and colleagues in the years to come if you continue in the CSCI program. Good luck out there!

Bibliography

[1] Figure 1.1 found at http://homepage.cs.uri.edu/faculty/hamel/courses/2016/fall2016/csc301/bootcamp/sessions/session-1a.html

[2] Figure 2 found at http://www.linuxtrainingacademy.com/wp-content/uploads/2014/03/linux-directory-tree.jpg