Unix and Computer Science Skills Tutorial
Workbook 1:
The Command Line and How to Use It

January 6, 2016

This workbook gives a tutorial on how to use the basic tools of the *nix command line. I recommend that you take the time to work through examples provided. There are also questions asked to help prompt you to look investigate and experiment (and sometimes to reinforce points). Some of the answers may be easy to see, and sometimes you may need to do a little research. If you'd like some suggestions on how to better find things you're looking for with Google I like the cheat sheet located at `http://mashable.com/2011/11/24/google-search-infographic/#7KEPYUXTxskY`.

1 Where are we? and Finding Ourselves

The first question we have is, "what can we see?" and in this case it's the *command prompt*.

1.1 The Prompt

The command prompt in your terminal should look something like this:

```
[user@zone1234 ~]$ □
```

This is what the default prompt should look like on all of the CPSC computers. It provides the place where we can type commands and it can give us some handy information. The first part of the prompt above is the *username* (who is logged in here), followed by the *hostname* (the name of the computer). This can be helpful, especially in the lab environment, to make sure that 1) you're using your own account and 2) you're using the computer you thought you were (although this shouldn't matter for most things).

The next thing this prompt shows is the current directory, which we'll talk more about in a second.

Finally, the prompt ends with a '$'. In most systems you'll see two types of prompts, the first is the '$', which, by convention, indicates that the shell is waiting for you to enter a command on the command line. The second prompt is '>', which indicates that a command is waiting for you to give more input.

> One of the places where you'll often see the '>' prompt is when you've included an unmatched quote in your command, when that happens the OS will give you the '>' prompt since it thinks you're not finished. You can always use `Ctrl-C` to quit out of that situation and try your command again.

On rare occasions, you might see a prompt '#'. It looks different to remind you to be very careful. This prompt is usually shown to administrators who need to make serious changes to the system. If you see one, be very careful.

The prompt is configurable, so on any given *nix computer it may be set up completely differently and anyone can configure their own prompt to be useful for them. We'll talk a little bit about customizing the prompt later.

> For all examples in the rest of this course when talking about the command line, I'm going to simply show a '$' prompt to keep things clean and simple.

1.1.1 Questions

1. What is the *hostname*?

1.2 Commands

At the prompt we enter commands. We'll talk about what the commands in the next section (and really for the rest of the tutorial), before that we'll take a quick look at the parts of a command.

```
$ <command> <options> <arguments>
```

> We use text in angle brackets '<' and '>' to show generic text that we'll replace with specific text later. So we would replace <command>with the actual name of the command and <options>with the actual options we would like to use. You'll see this convention in a lot of places in computer science. In other places you may see text to replace *italicized*.

- The command is the name of the thing that we would like to computer to do, such as list all files with the command `ls` or copy a file with the command `cp`. This workbook will introduce you to the important commands you will need to know, but if you're curious the command `compgen -c` will show you all the commands you can run.

- The options tell the command how you would like it to run. Most commands are complicated and can do a lot of different things different ways. Most options come in the form of flags, either short-flags which are a dash followed by a letter (`-v`) for example or long-flags that are usually two dashes followed by a word (`--version`.)

  Sometimes (most of the time) you won't need to specify options to a command, so they're optional (hence the name). Other times you may need to use several options together with a command to make it work.

  Different commands will have different options and you'll learn those commands looking in the command's manual entry.

- The arguments specify what you would like the command to work on. For some commands, like `ls`, the command doesn't need an argument to work, so we can leave them out. For other commands like, `cp`, the command doesn't know what you want it to do unless you tell it. When you call the `cp` command you need to tell it which file to copy and where to copy it to. Different commands will need to take different arguments (and quite a few take variable numbers of arguments), so you will need to read the manual to see what arguments the command expects and how it will handle them.

1.3 The manual

Before we can really start looking around we should figure out what we have in our tool box. *nix has a large number of standard programs that are common pretty much everywhere. This may seem a little overwhelming, but conveniently there is a manual system built in for all of them. You can call the manual by entering the command `man` and then telling it what command you would like to know more about.

```
$ man pwd
```

If `man` has an entry for the command then it will display it. Each `man page` is written by the maintainers of the software and looks a little different depending on what information they need you to know about the command.

A `man page` is divided into several parts including `NAME`, `SYNOPSIS`, `DESCRIPTION` and `SEE ALSO`. These are the four parts that are required for any `man page` but you will see several others that provide useful information, especially `EXAMPLES`.

- `NAME` provides a the name and brief description of the command.

- `SYNOPSIS` tells you how to use the command on the command line.
- `DESCRIPTION` is usually the most important part, it tells you in detail how the command works and what all of the different options are for the command and how to use them.
- `SEE ALSO` lets you know which other `man pages` you might need to read for a better understanding of what's going on.
- `EXAMPLES` provides examples of how to do common things with the command and can often be very helpful when learning a command for the first time.

All `man pages` are arranged into different sections, you can see in a `man page` that whenever a man page is referenced there is a number in parenthesis behind it. This shows which section a `man page` is in. All of the `man pages` we care about are in Section 1: User Commands. Other sections include system calls, library calls special files and system administration tools.

`man` is great when you know what command you need to learn about, but if you don't know the name of the command then you'll need a way to search. Fortunately `man` has a built in system to help you search, using the command line option `-k`.

```
$ man -k copy
bcopy (3) - copy byte sequence
BIO_ssl_copy_session_id (3ssl) - SSL BIO
BN_copy (3ssl) - copy BIGNUMs
BN_dup (3ssl) - copy BIGNUMs
BN_MONT_CTX_copy (3ssl) - Montgomery multiplication
ccopy (3) - (unknown subject)
ccopy.f (3) - (unknown subject)
copysign (3) - copy sign of a number
copysign (3p) - number manipulation function
copysignf (3) - copy sign of a number
copysignl (3) - copy sign of a number
copywin (3x) - overlay and manipulate overlapped curses
windows
cp (1) - copy files and directories
cp (1p) - copy files
...
```

When you use `man -k` to search, `man` will return a list of all `man pages` related to the term you give it. Here I searched for man entries related to copying. We can see that there are a lot of results (try it yourself, I only included a small portion here), but we're able to scan quickly through the list to see that most of

them aren't important to us. We can tell that based on the descriptions given, but more importantly that most of the `man pages` are in section 3, which is filled with library calls. If we know we want a user command we just have to find the entries in section 1.

If you want to know more about the `man` command you can run `man` on it.

```
$ man man
```

Some times you'll find that a `man page` doesn't do the best job of answering your questions, in which case you'll have to look to other places (by which I mean the Internet). But for an overview (and especially a list of options) the `man page` is the best place to start.

1.4 The Directory Structure

Everything in a *nix computer is contained of a directory. A directory is a place that can hold files and other directories (and actually, directories are special types of files). Every directory has a name, and all directory names end with '/' (though these slashes are sometimes left out, when we're sure we're talking about a directory). The command line environment allows us to look at one directory at a time. The directory we are currently looking at is called the *current directory*. We can see the name of the current directory by using the command, `pwd`.

```
$ pwd
/home/grads/tjkendon
```

`pwd` shows you the 'full' name of the directory you are currently working in. You can remember `pwd` by thinking **p**rint **w**orking **d**irectory. In the example above we can see that I'm in the directory `/tjkendon`, that directory is in another directory called `/grads` and that directory is in yet another directory called `/home`. Your directory will be different.

On a *nix computer there is one special directory, called the root directory, named '/', which contains everything else. There are several other special directories in a *nix computer which we will investigate later.

Every user gets a special directory of their own called their *home directory*. The home directory is the space given to that user to store and organize on their own and to put any files they want. *nix systems use permissions to control what files users can read, write and access and in your home directory everything is yours!

Since your home directory is where you'll do most of your work, *nix systems have a shortcut to make it easy to refer to. The tilde symbol '~' is the shortcut for the home directory and that should be the symbol you see in your prompt telling you were you are. You're at home!

1.4.1 Questions

2. What is the current working directory?

3. What is a home directory?

2 Moving Around and Looking at Things

We know where we are (in our home directory, somewhere in the file structure). Next let's look at how to move around the file structure.

2.1 Listing Around

The first thing we should find out is what's in the current directory. We can list the directory's contents using the `ls` command.

```
$ ls
Desktop Documents Downloads
```

Here we can see some default directories that the operating system created when your account was made. If you want to go look at one, we can use the `cd` command to change directories.

```
$ cd Desktop
```

If you run `pwd` you'll see that the current directory has changed and if you run `ls` you'll see that the contents are different. Now, what do we do if we want to go back to where we were?

The first thing I'll point out is that `ls` doesn't always tell you about every file in a directory. By default, `ls` doesn't tell you about any file that starts with a '.'. I'll also point out that there is an *option* to 'ls' that will let you see *all* the files. Use the `man` command to read the manual for ls and find the option to see **all** the entries.

Now you can see two important hidden directories. The first is '.', which is actually the shortcut name of the current directory. We can go:

```
$ cd .
```

and `cd` will change the directory to the current one. This is useful for things we'll look at later, but for right now we want to know how to get to the directory *above* this one and the shortcut name for that is '..'.

The directory structure looks like a tree if you draw a line from each directory to the one that contains it, with the root of the tree being '/', root. One important fact to know about computer scientists is that we draw all our trees with the roots at the **top**. There are reasons for this, but for now just know that the directory **above** contains the current one. We also often call this the parent directory.

So to go to the parent directory all we need to do is:

```
$ cd ..
```

and we should be back where we started.

We can change more than one directory at a time. For example if we want to go to the `examples` directory in your `Desktop` directory, then we can combine them into one and use the command:

```
$ cd Desktop/examples
```

from your home directory.

You can also use this along with '..' to go back up the file structure. If you wanted to go from `Desktop` to `Downloads`, then you could use the command:

```
$ cd ../Downloads
```

Finally anytime you're not in your home directory and you want to go there, you can use the '~' shortcut to go home again:

```
$ cd ~
```

.

As a quick note, in a lot of *nix systems, the command `cd` on its own also returns you to your home directory. This isn't true everywhere so it's worth checking what happens when you use it.
It's also good to know that if you want to go back to the directory you were just in you can use the command `cd -`.

2.1.1 Questions

4. What option lists all hidden files in `ls`?

5. What is the shortcut name for the current directory?

2.2 Going for a Wander

Now you can use `cd` and `ls` to move around the file structure and see what things look like, in your home directory and above it.

2.2.1 Relative and Absolute Paths

Just before you start looking around, there's one more thing you need to know, *relative* and *absolute* paths. A *path* is a full name for a file that tells you where the file is in the file structure.

An absolute path describes where the file is relative to the root directory '/' (the directory that holds everything else). For example, the Desktop directory in your home folder might have the absolute path `/home/you/Desktop`. This lets you know that the `Desktop` directory is in the `you` directory, which is in the `home` directory which is in the `/` directory.

Absolute path names are necessary because we need to be able to uniquely identify every file in the file structure. However it can be a lot of work to type the whole absolute path every time you need to change directories. Relative paths describe the location of a file relative to the **working directory**.

We've already seen this when we used `cd` to go into the Desktop directory. We used the command `cd Desktop` and the shell looked in the working directory to find the `Desktop` directory and changed to it. We did the same thing when we moved back to the home directory, when we used `cd ..`, the shell moved to the parent directory of the working directory.

From your home directory, the commands:

```
$ cd /home/you/Desktop
$ cd Desktop
```

are the same.

You can tell the difference between an absolute path and a relative path because the absolute path always begins with a '/'.

2.2.2 Visiting the *nix File Structure

Let's start our tour by going to root. Root is the same as any other directory, so all we need to do is:

```
$ cd /
```

Now let's list it to see what's here:

```
$ ls
bin dev home lib64 lost+found mnt proc run scripts srv tmp
var boot etc lib libexec media opt root sbin share sys usr
```

These directories are the common directories that hold all the files (and programs) that are needed to make a *nix operating system work. Let's start by taking a quick look at the `/bin` directory.

```
$ cd /bin
$ ls
```

You'll notice that there are a lot of files to print here, so ls leaves us at the bottom of a very long list. Using the terminal window, we can scroll back up through the list to see all the files, but sometimes we might not have that option, and even with scrolling it can be too hard to see all of the output of a program (and ls is actually a program). We have another option to make it easier to see however and that is to use *piping* and the reader `less`. Try this command:

```
$ ls | less
```

Where that vertical line is the 'pipe' character '|', which shares the backslash '\' key. Piping is a form of I/0 redirection, which allows us to send the output of one program to another program, so here we have piped the output of `ls` into `less` which lets us scroll through the output using the spacebar to read a whole page and the up and down keys to go up and down line by line. **You can quit out of `less` using the 'q' key**. You'll notice that the files aren't colour coded this way, but that's the trade off for being able to see the output one page at a time. We'll look a little bit more about piping and file redirection later.

`/bin` holds the core programs that allow us to do all the things we need to do. `cd` and `ls` live here (I told you it was a program). Take a minute to look at see if you recognize any of the other programs in this directory.

The shell has a special variable called the PATH. The PATH is a list of all the places the shell should look to find a program to run. One of the places in the PATH is normally the current working directory, which means if you have an executable program in your current directory you can just write its name on the command line. `/bin` is also in the PATH, so when you want to run a command line `cd` the shell knows to run `/bin/cd`. Otherwise we'd have to write out `/bin/cd` every time we wanted to change directories.

The name `/bin` refers to the fact that this directory holds the *executable binary files*. Binary files are files that are designed to be computer readable, but not human readable. We call human readable files *text files*. Most files that contain programs we can run are binaries because they contain the machine code that the operating system needs to run the program.

Let's take a look at another directory now, `/etc`. Change directories to `/etc` and take a look at its content now. `/etc` holds many of the configuration files that tell the operating system and other programs vital information.

Take a few minutes to look around at all of the different directories that live in '/' and research online to figure out what these directories are usually used for and how they got their names. When you're ready go back to your home directory (`cd ~`) and we'll look at making our own directories.

2.2.3 Questions

6. Does the absolute pathname change when you change the working directory?
7. Does the relative pathname change when you change the working directory?
8. What is unusual about /dev and /proc?
9. What is the difference between /bin, /usr/bin and, /usr/local/bin?

2.3 Making our own File Structure

Making directories is quite simple, we just need to use the command `mkdir` and give it a name for the directory we want to create. First, make sure you're in your home directory, then start by making a new directory for the tutoria , since we want to avoid spaces, call it `unix_tutorial`.

```
$ mkdir unix_tutorial
```

Now you can list and see the new directory and you can use `cd` to change directories into it.

If you put a space between unix and tutorial (or anywhere in your new directory name) `mkdir` will make two directories. It's possible to put spaces into directory names but generally it makes things much harder to do, so stick to alpha-numeric characters and underscores for your file names.

If you want to make a hierarchy of directories all at once you can do that too. However if you just tell `mkdir` to make a directory with a hierarchical path it will get confused and give you an error.

```
$ mkdir workbook1/solutions
mkdir:  cannot create directory workbook1/solutions: No
such file or directory
```

`mkdir` went looking to find a directory called `workbook1` and couldn't find one, so it stopped and reported an error. Instead we can use an option to tell `mkdir` to make all the parent directories needed. Take a moment to look at the man page for mkdir and find the necessary option and make the directory `workbook1/solutions`.

Removing directories is also simple. We use the `rmdir` command, which takes a directory name the same as `mkdir`. Since there are no solutions let's remove that directory. Change directories to workbook1 and use the command:

```
$ rmdir solutions
```

`rmdir` works so long as the directory is empty. If you return to your home directory and try to remove your `unix_tutorial` directory `rmdir` will *throw* an error.

```
$ rmdir unix_tutorial
rmdir:  failed to remove unix_tutorial: Directory not empty
```

Next we should get some files to put into our directories.

2.3.1 Questions

10. What is the option needed to make mkdir create a hierarchy of directories?

3 Files and Things to Do With Them

Now let's start by making some files. To keep things organized, let's go to the unix_tutorial/workbook1 directory we created earlier.

There are a lot of different ways to create files, but since for right now, we don't want to put anything in them (more on that soon) we can use a command called `touch`. Let's start by making a file called **test**.

```
$ touch test
```

Now we can list and see that our new file exists.

The next thing we might want to do is rename the file. The `mv` command allows us to do this. `mv` takes the existing name of the file and then the name you would like it to change to.

```
$ mv test bettertest
```

Now we can list again and see that we've changed our file's name.

Next, let's try copying the file. The command `cp` allows us to copy files. Like `mv` we have to give it the name of the original file and the name of the file we would like to copy.

```
$ cp bettertest secondtest
```

Listing again, we can see that there are now two files in the directory.

`mv` and `cp` also work when you want to move or copy file between directories. Let's make a directory called `inner` and then we'll copy a file and move a file into it.

```
$ cp bettertest inner/innertest
$ mv bettertest inner
```

Now if we change directories to `inner` and list the contents we will see two files, `innertest` and `bettertest` (and you'll see that better test isn't in the parent directory any more). With both `cp` and `mv` if you supply a file name after the directory name, the moved or copied fill will have that name. If you don't supply a name it will use the same name it has in the current directory.

You should use `cp` a lot as you work on your assignments. Once you have a part of an assignment finished, make a copy to make sure you have an old version in case something goes wrong. That way you always have something to go back to.

`mv` works on directories the same way it works on files. You can rename `inner` to something else and if you create a second directory you can move inner into it. Take a moment to test this on your own.

`cp` doesn't copy directories unless you tell it explicitly to.

```
$ cp inner newinner
cp:  omitting directory inner
```

We can tell it explicitly to copy the directory by using the `'-r'` option.

```
$ cp -r inner newinner
```

Deleting files is also simple. We can remove files by giving the command `rm` the name of the file we want to remove. For example to remove the file `secondtest`:

```
$ rm secondtest
```

completely removes the `secondtest` file. This is important to remember: **There is no undo from deleteing a file with rm!** There's no trash or recycle bin, your file is simply **gone**. So use `rm` with the respect and caution it deserves. Many installations of `rm` now ask you to confirm before you delete a file, just to give you a chance to change your mind.

3.0.2 Questions

11. What option do you need to use to make cp copy a directory?
12. When you delete a file with rm, is there a way to get it back?

4 Reading Things and Writing Things Down

Now that we can create directories and manage files, let's do something a little more exciting. To start off with let's go get some files to work with.

The files are stored in the course directory at `/home/courses/unix` in a directory called `text_files`. Use the `cp` command to copy those files to your `unix_tutorial/workbook1` directory.

Your `text_files` directory should have 3 files, `short_file.txt`, `medium_file.txt` and `long_file.txt`.

4.1 Reading Files

We're going to start by using the command `cat` to see what's in each of these files. `cat` does a lot of things, but one of the things it does is print text files out on the command line.

```
$ cat short_file.txt

TEA

By FRANCIS SALTUS SALTUS

From what enchanted Eden came thy leaves
That hide such subtle spirits of perfume?
Did eyes preadamite first see the bloom,
Luscious nepenthe of the soul that grieves?

By thee the tired and torpid mind conceives
Fairer than roses brightening life's gloom,
Thy protean charm can every form assume
And turn December nights to April eves.

Thy amber-tinted drops bring back to me
Fantastic shapes of great Mongolian towers,
Emblazoned banners, and the booming gong;
I hear the sound of feast and revelry,
And smell, far sweeter than the sweetest flowers,
The kiosks of Pekin, fragrant of Oolong!
```

We can also use `cat` to print out the `medium_file.txt`, although we have to scroll back a little way (if we can). The `long_file.txt` is long enough that it fills up the terminal buffer and we can't actually scroll back through it all.

`long_file.txt` is a data set for machine learning. It's about 250,000 words long, which would be a fairly long novel. Although because these sentences were taken from comments on the Internet, it wouldn't be a `good` novel.

As with when we had to deal with too much output from `ls` earlier, `less` can help us. `less` breaks the file up into pages and shows us one page at a time. We can use the space bar to go down one page a time and the b key to go back one page, we can go up or down one line at a time using the arrow keys. We can quit out of `less` using the q key.

> `less` isn't the only *nix utility that can let you read a file one page at a time. `more` was the predecessor to `less` that did the same thing. `more` has two big problems though, it crashes when it see certain characters in files and it won't let you go backwards. `less` was created to address both of these problems and was named `less` because less is more than `more`. (You'll find that programmers and computer scientists tend to make really, really awful jokes in ways that last forever.)

4.1.1 Questions

13. Why is it better to use less rather than cat to read a text file?

4.2 Writing Files

For all the things we've covered so far, there has really only been one program for each of the things we need to do. Now that we want to start editing files, things become more complicated.

There are two editors which are present on every *nix operating system, `emacs` and `vim`. These are the two oldest (usable) text editors and in order to do anything on a *nix system you will need to be comfortable working in at least one of them. You don't necessarily have to master them, although both are incredibly powerful tools, but you do need to be able to at least open, edit and save files with one of them.

That being said, in computer science, you will spend most of your time editing text files. So, while you don't have to master `emacs` or `vim` you do need to find a favourite text editor and get very, very good at using it. A simple graphical text editor that is available on all of the department machines is `gedit` and a more advanced editor is `Kate`.

You might notice that if you invoke a graphical editor, like `gedit` or `Kate`, that you can't do anything else with your terminal until you close the window. That's because the shell is running your editor in the *foreground* and it's waiting until your editor is done before it does anything else.
Sometimes you might not want to lock up a terminal window while you're editing a files (for example if you want to run or compile the program you're working on). In these cases you can tell the shell not to wait for the command to finish before going on (to run in the *background*) using a `&` at the end of your command. For example:

```
$ gedit myfile.txt &
```

This will unhook your terminal from your editor while you edit and you can keep using the command line while you have the file open in another window.

When we're talking about mastering a program we mean that you should be able to do most of your normal, everyday work without needing to stop and think about the tool you're using. At the lowest level, this means not needing to look at the manual (although you might want a cheat sheet, while you're learning and even afterwards). It also means that you're able to do things quickly, without having to hunt through menus for the command you need, and that you're familiar with all of the commands/utilties that your tool can offer that will help you do your job.

In computing there's a long history of two alternate solutions to a problem being developed and people having long, unending arguments about why one is better than the other. One of these arguments has been between `emacs` and `vim` and when you work with one or the other people will tell you that you're using the wrong one. Just remember: **as long as you can do the things you want to get done then you're using the right tool**.

There's nothing wrong with working in the Windows environment, but it's important to be sure that your using the right tools. In particular you should find a good text editor. Notepad is a very simple program and while it produces plain text files, it doesn't have a lot of the nice features that we would like. There are several different programs you can download, but one of the most commonly used is called Notepad++ (Notepad, plus, plus) and it provides the syntax highlighting and other features that make it easier to work effectively in a programmer's environment.

You might wonder why we aren't talking about programs like Microsoft Word. While there are good uses for Word, it and programs like it like OpenOffice, LibreOffice or Google Docs are *word processors*.
Word processors are designed to let you produce nice, readable documents. Their files are full of information about how to organize and display information and we don't want that. We want plain text files that only include what we put in them. Word and its cousins are the wrong tool for the job for us.

4.2.1 Questions

14. **Find a cheat sheet for your favourite editor. (Don't limit yourself to `emacs` or `vim`, but do think carefully when choosing the editor that makes the most sense to you). Provide a link to the cheat sheet.**