Compiled on 12/01/2021 at 1:33pm

This is a draft for a module on the Linux Command Line Shell being prepared for a new course, CS136-Lab. The document is provided to you with the hope that you will provide feedback on the content including any typos/mistakes to nanaeem@uwaterloo.ca

The Shell

The Shell is the user interface to the Operating System (OS), i.e., it is a program running on the OS that enables a user to interact with the OS. It is through the shell that a user can run other programs, manipulate files etc. Graphical shells are more popular. Like any other program with a graphical interface, a Graphical Shell provides controls such as menus and buttons that can be manipulated with a mouse. Features such as clicking on icons and the ability to drag and drop stuff makes using a graphical shell intuitive. All mainstream OSes provide graphical shells and you are likely already comfortable using the graphical shell supported by your machine's OS.

An alternative to using a graphical user interface is to use a text-based interface called a command shell. A user opens a window that provides a *command prompt*. The user types in commands to perform the desired action. Those not accustomed to using command-line shells often think of it as archaic and cumbersome. After all, why learn a large number of commands and deal with syntax and typing errors when you can simply use your mouse and button clicks to achieve the desired effect? However, many, especially developers, find command-line interfaces to be much easier to use and often prefer them for their development work over the graphical interface. It is difficult to pinpoint any one reason for this preference. For one thing, many developers believe that they can perform tasks much faster by typing the commands rather than moving the mouse around and going through graphical menus. For another, a graphical interface is always going to be limited by the tasks it can conveniently perform, since each task the graphical interface can perform is a feature that had to be built into the graphical interface. A common example is tasks that might need to be performed on multiple files, such as exporting a document to PDF format. While the graphical interface might allow you to perform the task on individual files through a sequence of clicks (open the file, select export to PDF, assign the destination name and location, save the file), it is quite cumbersome to do this for hundreds of files. On the other hand, such a task would be quite straightforward to perform through the command-line interface if you knew the command to convert one file.

Our aim in this module is to not force you to like using the command-line interface. It is simply to introduce you to this alternate way of interacting with the Operating System. The learning curve can be steep as a small set of commands need to be learnt upfront before reaping the benefits of the command-line shell. We encourage the reader to give learning to use the command-line interface an honest try. You will not regret it.

1 The Linux Shell

We choose to use the Linux command-line shell. Linux is a freely distributable version of the Unix Operating System and was originally developed by Linus Torvalds. There are a number of popular command-line shells available for Linux. While the history of some of these shells is fascinating, we consider it beyond the scope of this document. One of the oldest, if not the oldest, Unix shells was

developed by Stephen Bourne at AT&T Bell Labs in the 70s. At that time, this shell, being the only shell was simply referred to as the shell. Over time as Unix evolved, more shells were created. C shell (csh) and Korn Shell (ksh) are two such notably old shells. As the Unix Operating System evolved in the 70s and 80s, so did the shells. In 1989, the Bourne-Again Shell (`bash`) was written by Brian Fox. This shell supported all the features of the original Bourne Shell and made a number of useful extensions that have become ubiquitous in command-line shells nowadays, such as piping and globbing.

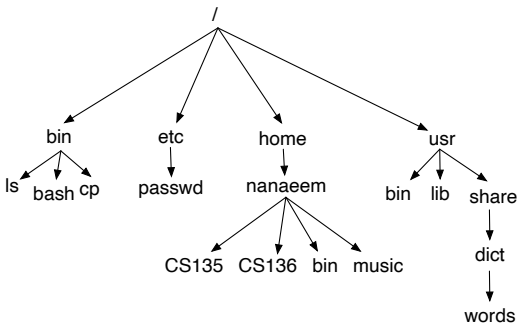
We will use the Bourne-Again Shell in this module. Other shells are similar, though minor differences in commands and syntax do exist. To ensure that you are running `bash`, open your chosen command-line shell and type the command shown in blue below:

```
$ echo $0
```

The first `$` symbol shown above is the command prompt. The `bash` shell often uses `$` to represent the command prompt. The command you typed in was `echo $0` followed by enter. The shell should respond with the type of shell you are running, i.e., if you are running `bash`, it should print `bash`. If you happen to not be running `bash`, you can start a `bash` shell by simply typing in `bash` and pressing enter.

2 Linux File System

Operating systems must have a way to organize the information stored on the machine. The smallest entity in which information is stored is called a file. Certain files are special in that they can hold other files. Such files are called *directories* in the Unix world (as opposed to Folders in Windows). The Linux File System, forms a tree structure with a special directory named the `root` directory, forming the root of this tree of files. The symbol `/` is used to refer to the root directory. The following diagram shows a possible organization of some of the common files found in a typical Linux installation.



An advantage of the single tree structure is that we can uniquely specify the location of any file by giving the `path` taken to reach that file from the root directory. For example, from the sample file system shown above, the CS135 file (which could be a directory) has the path `/home/nanaeem/CS135`, i.e., within the `root` directory, there is a directory named `home` which has a directory named `nanaeem`. This directory has a file named `CS135`. One thing you will notice is that the `/` symbol serves a dual purpose in a path; while the first `/` represents the `root` directory, the same symbol is used to separate the different files in a path. As an example consider the path `/usr/share/dict/words`, i.e., within the `root` directory, there is a directory named `usr` which

contains a directory named `share` which contains a directory named `dict`. We know that these files are directories since they contain other files. The path ends with a file `words`. This is in fact a newline-delimited text file containing dictionary words.

3 Commands

In this section we will cover some basic commands and their typical usage. This is by no means an exhaustive list nor does it cover the multiple options that can be supplied to each command. The hope is that knowledge of these commands will serve as a good starting point for a reader to comfortably navigate the command-line shell.

- **ls: list directory contents.** Typing the command `ls` will list all non-hidden files in the directory where the command is executed. To include hidden files (those whose filename begins with a period/dot) in the listing, the argument `-a` can be provided.

  ```
  $ ls -a
  ```

  It is worth noting the syntax at this point; the command executed is `ls`. The string `-a` is an argument to the command. One or more command-line arguments can be provided to commands by delimiting them with one or more space.

- **man: manual page**. A very useful command to know is `man` which is short for manual. It can be used as a way to quickly find the manual page for a program or command. You can use the command by simply providing, as an argument, the name of the command whose manual entry you would like to see.

  ```
  $ man ls
  ```

  The command above will show the manual page for the `ls` command.

  Read through the manual entry for `ls` to find the command-line argument needed to sort the listing in reverse lexicographical order.[1]

- **pwd: present working directory**. A user can navigate the file system by going in and out of directories. At any time, the user is always in some directory which can be called the `current directory` or the `present working directory`. The `pwd` command returns the name of the current directory. In the section on configuring the command line shell, we discuss how you can configure your terminal to always show you present working directory; a significant convenience knowing where you are at all times!

- **cd: change directory**. As the name suggests the `cd` command is used to change the current directory. The command takes one argument, a path to the directory which will become the current directory once the command is executed.

  ```
  $ cd /usr/share/dict
  ```

  The command above will change the user's current directory to become `dict`. Recall that `cd` is the command and `/usr/share/dict` is the argument to the command.

[1] Using `man ls` we can search through the manual page for the `ls` command. The `-r` argument reverses the order in which the listing would have been displayed.

The paths we have looked at so far, e.g. `/home/nanaeem/CS135` or `/usr/share/dict/words` have been examples of `absolute paths`; paths that start at the root directory as indicated by the presence of `/` at the start of the path. Another type of paths, `relative paths`, also exist. These paths specify a path relative to the current directory. Using relative paths can be a great convenience. Consider the situation where the user's current directory is `/home/nanaeem` and the user wants to navigate to the directory given by the absolute path `/home/nanaeem/cs136/a1`. The user could type `cd /home/nanaeem/cs136/a1`, i.e., provide an absolute path as an argument to `cd`. However, providing a relative path is more convenient. From within the `/home/nanaeem` directory, the command `cd cs136/a1` has the same effect. Notice that the relative path does not begin at the root directory. The relative path `cs136/a1` from within `/home/nanaeem` is the same as the absolute path `/home/nanaeem/cs136/a1`.

Special Directories:
While relative paths are convenient, another convenience is the presence of shortcuts for specific directories in the file system. We list four such special directories below:

1. **Dot** (.) : The dot symbol is used to refer to the current directory. For instance the path `./cs136/a1` says in the current directory there is a directory `cs136` which contains a file a1. Note that this makes dot a relative directory since it is by definition relative to the current directory.
2. **DotDot** (..): The dotdot shortcut is another relative directory and refers to the parent directory of the current directory. If the current directory was `/home/nanaeem/cs136`, then the command `cd ..` would change the directory to the parent of `cs136`, i.e., the current directory would become `/home/nanaeem`. From within the `cs136` directory, the shortcut `../..` can be used to refer to the parent of the parent, i.e., the grandparent directory, which in this case is `/home`.
3. **tilde**(~): The symbol ~refers to the current user's home directory. This is an absolute path, i.e., the symbol refers to the home directory of the user irrespective of where they are in the file system. Irrespective of where the user is in the file system, the path `~/CS136` is the same as the path `/home/nanaeem/CS136` given that the user's home directory is `/home/nanaeem`. A common use is to use the command `cd ~` to change to the home directory.[2]
4. **~userid**: This shortcut is used to refer to any user's home directory. Of course files in that user's home directory would only be accessible if the current user has the appropriate permissions (discussed later).

The following table summarizes some of the directories and paths we just discussed:

Directory	Meaning
.	Current directory
..	Parent of current directory
~	Your home directory
/	Root directory
Starts with / or ~	Absolute path
Does not start with / or ~	Relative path

[2]Interestingly, if the command `cd` is used without any arguments, it will take the user to their home directory, providing yet another shortcut.

- **mkdir: make directories**. As the name suggests the `mkdir` command is used to make, i.e., create, new directories. The command takes one or more directory paths as arguments. The most common usage of the `mkdir` command looks as follows:

  ```
  $ mkdir testing
  ```

  The command above creates a directory named `testing` within the current directory.

- **cat: concatenate and print files**. The command `cat`, short for concatenate, is used to concatenate files. One can also use the `cat` command with just one argument, the name of a single file as shown in the example below:

  ```
  $ cat /usr/share/dict/words
  ```

  Recall that `/usr/share/dict/words` is a newline-delimited text file containing dictionary words. Running the command above will display the contents of the file on the user's screen. (Note that if this file exists on your system, it is likely several hundred thousand lines long and a lot of output will be produced!)

  To use the concatenation functionality of the command, the names of two or more files can be provided to the command as command-line arguments. The command will display, on the screen, the sequential concatenation of the contents of the files in the order the names were provided as arguments, i.e., the command

  ```
  $ cat file1.txt file2.txt
  ```

  will display the contents of `file1.txt` followed by the contents of `file2.txt` on the screen. One can capture the output generated by this command, and all others. We discuss this in the next section.

- **cp: copy files**. The `cp` command is used to copy files from a source location to a destination. For example, the following command copies the file `main.c` in the current directory into a directory named `backup` which also exists in the current directory, i.e., the first command-line argument is the path to the source of the copy and the second command-line argument is the path to the destination.

  ```
  $ cp main.c backup/main.c
  ```

  The following two commands have the same effect as the command above:

  ```
  $ cp main.c backup/
  $ cp main.c backup/.
  ```

- **mv: move files**. The `mv` command moves the source file (first command-line argument) to the destination (second command-line argument). If the source and destination directory are the same, the `mv` command acts as a command that renames a file.

  ```
  $ mv test.c backup/test.c
  ```

  The command above moves the file `test.c` from the current directory into the `backup` directory. The command below renames the file `test.c` to `main.c`.

  ```
  $ mv test.c main.c
  ```

- **rm: remove files**. The `rm` command is used to remove files. Aside from removing ordinary files, it can also completely remove directories by recursively deleting all the files and subdirectories contained within them.

One must be very careful using this command as **there is no "recycle bin" or "trash" directory** where removed files are moved. This means that once the file is removed, it becomes unlinked and, while the data still resides on disk, there is no easy way to retrieve it (unless you are a professional with expertise in data recovery). Beginners are therefore advised to always use the `-i` command line argument with the `rm` command. This argument will cause `rm` to prompt the user whether they actually want to remove the file; only typing y/Y will cause the file to be removed.

By default, `rm` will produce an error message if used on a directory as opposed to an ordinary file. To remove an empty directory, use the `-d` argument with `rm`, or the separate command `rmdir`.

To remove a directory that still has files in it by recursively deleting all the files and sub-directories within, use the `-r` argument with `rm`. Be extremely careful with this option as you can destroy a large amount of data with it!

- **ssh: OpenSSH SSH client for remote command-line logins**. The `ssh` command is used to establish a secure SSH connection to a remote host. The simplest form of the command is as follows:

  ```
  $ ssh nanaeem@linux.student.cs.uwaterloo.ca
  ```

  In the example above, we are connecting to the host `linux.student.cs.uwaterloo.ca` as the user `nanaeem`. If the host is reachable, and the provided user exists on the host, you will be prompted for a password for that host. Note that typing the password does not show any characters appearing on the screen as an added security feature; you are probably used to stars or dots appearing for each character, but when using SSH nothing at all will appear. Once a correct password is provided, you will have initiated an SSH session on the host machine, i.e., any commands executed from hereon will be executed on that machine and not your local machine. The command `logout` terminates an SSH session.

  If the userid is not provided, then the userid of the user currently logged on is used. It is often convenient to create a user on your local machine with the same userid as the one you often use to connect to external accounts.

  Specifying userids, long host names and passwords can get annoying. This can be simplified by setting up an SSH configuration file. If a configuration file already exists it will have the following path `~/.ssh/config`, i.e., the name of the file is `config` and it is in the `.ssh` directory (note the . in the directory name) in the user's home directory. You can edit this file using your favourite text editor and add new configurations. We show a simple configuration entry below:

  ```
  Host student
    Hostname linux.student.cs.uwaterloo.ca
    User nanaeem
  ```

  With the above configuration added to the ssh config file, we can now simply run the command:

  ```
  $ ssh student
  ```

  Notice that `student` was the name we gave to this host which had the hostname `linux.student.cs.uwaterloo.ca` and userid `nanaeem`.

This overcomes the annoyance of having to remember your userid and hostname for different servers or having to type long addresses. However, it still does not remove the annoyance of having to input your password every time. The `ssh-keygen` tool can be used to automatically authenticate hosts and users using public key cryptography. Briefly, the command

```
$ ssh-keygen
```

will generate a public/private key pair by prompting the user for a location for the key pair and a passphrase. If the computer being used to generate the key pair is only accessed by the user, it is common practice to leave the passphrase empty. A word of caution, leaving an empty passphrase is akin to having no password, i.e., this would enable anyone to log in as the user without being prompted for a password. By default the public/private rsa key pair is stored within the `.ssh` directory under the user's home directory (notice the dot, which makes the directory hidden). The public key (default filename `id_rsa.pub`) can then be copied to the `.ssh/authorized_keys` text file on the host. The following command will copy the public key to the host and user's account discussed above:

```
$ ssh-copy-id nanaeem@linux.student.cs.uwaterloo.ca
```

Alternately, the public key text file can be manually copied to the appropriate location. Once this is configured, the user will no longer be prompted for a password when initiating an SSH session with that specific host.

- **scp: secure remote copying**. The `scp` command is similar to the `cp` command discussed above but enables copying files to and from remote (external) machines over a network. The command has the same format as the copy and move commands, i.e., the first command-line argument is the source and the second command-line argument is the target/destination. The only difference is that both the source and target can be remote hosts.

  While using `scp` one can use relative or absolute paths for local paths but must use an absolute path for a remote path. Additionally, the remote path must be prefixed with the userid and host specification needed to access the remote host. Here is an example:

  ```
  $ scp test.c nanaeem@linux.student.cs.uwaterloo.ca:~/cs136/a1/main.c
  ```

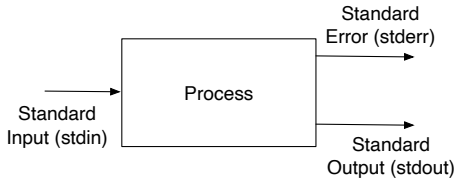
  The command above attempts to copy the file `test.c` from the current directory on the local machine (hence the relative path) to the machine `linux.student.cs.uwaterloo.ca`. The user must specify the userid for that machine (`nanaeem` in the example above). If a userid is not provided, the command will automatically use the userid of the current machine. A colon separates the host name with the absolute path to the destination. In this example, the file `test.c` is being copied from the local machine to the remote directory `~/cs136/a1` and being given the name `main.c`. If the target directory does not exist, the command will not succeed. If a host has been configured in the ssh configuration file as discussed above, that configuration can be utilized in uses of the `scp` command:

  ```
  $ scp student:~/cs136/a3/solution.c .
  ```

  The command above is copying a file named `solution.c` from within the directory `~/cs136/a3` on the remote machine that we previously configured as `student` (from above, the hostname was `linux.student.cs.uwaterloo.ca` and the user `nanaeem`). The destination path is specified using just a dot, representing the current directory.

4 Standard Streams for Linux Processes

Each Linux process is attached to three standard streams, illustrated by the following diagram:



The standard input stream is the source of input to the program. When the programs reads input, it is the standard input stream that it reads data from. For example, when a C program uses the `scanf` function, it is reading input available on the standard input stream. By default, this stream (or `stdin` for short) is connected to the keyboard. However, the shell supports the ability to use `input redirection` to channel information from a file into the standard input stream. This is achieved using the `<` operator.

```
$ myprogram < /usr/share/dict/words
```

The command above is redirecting the contents of the text file `/usr/share/dict/words` and making it available to the program `myprogram` on the standard input stream. If this program was a C program, when the program uses the `scanf` function it would be receiving information that was once in the text file. It is however important to realize that the program is not itself reading the contents of the file; input redirection is a feature of the Linux shell. The Linux shell opens and reads the file specified and makes the contents available on the input stream for the program. This is different from the following command:

```
$ myprogram /usr/share/dict/words
```

which provides the file name as a command-line argument to the program (notice the absence of the input redirection operator, `<`). In this case, the program would need to have the ability to open and read the contents of a file when given the filename.

As another example, consider the following two executions of the `wc` command (we also display the output generated by each execution):

```
$ wc /usr/share/dict/words
235886  235886 2493109 /usr/share/dict/words
$ wc < /usr/share/dict/words
235886  235886 2493109
```

The command `wc`, when given a file name as a command-line argument in the first example above, will output the number of lines, words and characters in the given file. Notice that the output also shows the file name. The second example above uses input redirection, i.e., instead of providing the file name to `wc` we make the contents of the file available on the standard input stream through input redirection. The output shows the same counts for the number of lines, words and characters. However, the file name is missing. It is missing since the name was never provided to the command, only the contents of the file were.

Of course, a command might use both command-line arguments and input redirection together:

```
$ myprogram file1.txt < file2.txt
```

The command above provides the string `file1.txt` to the program as a command-line argument and also makes available the contents of the file `file2.txt` on the standard input stream.

Let's now direct our attention to the two output streams. The standard output stream (or `stdout` for short) is the preferred output stream for sending the output generated by a process during its normal processing, i.e., expected output is produced on stdout. For example, the `printf` function from C sends the output to standard out. The second output stream, standard error, is intended to be used for outputting error messages or unexpected output. The `fprintf` function in C can be used to generate output on the standard error stream (`stderr` for short). One reason for a separate output stream for errors is so that the standard output stream is not cluttered by unwanted error or logging information. Another reason is that while the standard output stream is buffered for efficiency, the error stream is not, i.e., data made available by a process on the standard error stream becomes available right away.

By default, both output streams are connected to the computer's display, so any data sent to these streams becomes visible on the screen. However, this can be changed by using the Linux shell's `output redirection` capabilities; for `stdout` use `>` and for `stderr` use `2>` to redirect the output generated on these streams to a file. Of course, input and output redirection of different streams can be combined. We look at some examples below:

```
$ myprogram > out.txt
```

The command above redirects any output produced by the program on the standard output stream to the file `out.txt`. If a file with that name already existed, it is overwritten. Any output produced on the standard error stream is sent to the default; the screen.

To append to a file, rather than overwrite it, one can use `>>`. For example:

```
$ echo "Good morning" > out.txt
$ echo "Good afternoon" > out.txt
$ echo "Good evening" >> out.txt
$ cat out.txt
Good afternoon
Good evening
```

The `echo` command simply outputs the strings that are given to it as command line arguments. The first `echo` command above creates a new file containing the string `Good morning`. The second `echo` will overwrite this file with a file containing `Good afternoon`. However, because `>>` is used in the third `echo`, the string `Good evening` is appended to the end of the existing `out.txt`, instead of overwriting the file. The result is a file that contains `Good afternoon` followed by `Good evening`.

```
$ myprogram > out.txt 2> errorlog.txt
```

The command above redirects the standard output stream to the file `out.txt` as in the previous example, but also redirects the standard error stream to the file `errorlog.txt`.

```
$ myprogram < in.txt > out.txt 2> errorlog.txt
```
In the example above, the standard output stream is redirected to the file `out.txt`, the standard error stream to the file `errorlog.txt` and the standard input stream is redirected from the file `in.txt`.

It is possible to tie both streams to the same location:
```
$ myprogram > out.txt 2>&1
```
This command sets the standard output stream to the file `out.txt` and **then** sets the standard error stream to `&1` which is the descriptor for the first output stream, i.e., wherever the standard output stream has been redirected. In other words, output produced on either of the output streams is being redirected to the same file. In previous examples, the order in which the redirections were done did not matter. In this case it does matter: first the output stream is redirected to the file, and then the error stream is redirected to the same place as the output stream. The shorthand `&>` can also be used to combine standard output and standard error. This command is equivalent to the command above:
```
$ myprogram &> out.txt
```

Finally we mention the special file `/dev/null`. Data redirected to this file is simply discarded, which allows you to supress the output of commands.
```
$ myprogram 2> /dev/null
```
This command discards all output sent to the standard error stream. All output sent to standard output will appear on the screen but standard error will be surpressed.

5 Wildcard Matching / Globbing Patterns

The Linux shell supports "globbing patterns" as a means of matching filenames. Note that globbing patterns are different from regular expressions, which have become a common feature in programming languages and are used by some Linux commands. Globbing patterns support a number of wildcard characters and syntax to create patterns that are then checked against filenames. We discuss the different wildcard characters with examples below:

? symbol is used to match any one character. For example ?.txt will match a file name that has any one character and is then followed by .txt. Of course, to match any two characters, one can use two ? symbols. The globbing pattern CS???.txt will match any filenames that begin with CS followed by any three characters followed by .txt.

*** symbol** is used to match zero or more characters. For example, *.txt will match any file name that ends with a .txt.

[ ] symbols can be used to specify one character from a set of characters. For example, [aeiou].txt would match any filename that has a name with one of the vowels followed by .txt. Character ranges can also be specified e.g. [1-9], [a-z] etc.

[!] symbols can be used similar to [ ] with the difference being that it will match any one character other than those specified in the set. For example [!aeiou].txt will match filenames where the single character filename is not a vowel followed by .txt.

{ } symbols are technically not a globbing pattern but are supported as a means of saying "or". For example *.{cc,cpp} will match all filenames with a .cc or .cpp extension (the two common extensions for C++ files).

It is worth understanding the process behind globbing patterns. When a globbing pattern is used on the command-line, the Linux shell intercepts the pattern and finds all (non-hidden) files in the current directory that matches the pattern. These filenames are then substituted, delimited by a single whitespace, in place of the globbing pattern. If the globbing pattern was used as a command-line argument to a command, the shell will then execute the command using the substituted filenames as arguments to the command. For example, suppose the current directory contains a number of files, three of which are a.txt, b.txt and c.txt. If we execute the command

```
$ cat *.txt
```

the Linux shell will first replace the globbing pattern with the names of the three files that match the pattern. At that point, the cat command will be executed. This command will have the format:

```
$ cat a.txt b.txt c.txt
```

Notice how the globbing patten has been replaced by the filenames that match the pattern. The resulting command will produce the concatenation of these three text files.

It is important to realize the convenience that comes about from the fact that globbing pattens are a feature of the shell itself, not a feature of individual commands. The implication is that we can use globbing patterns with many different commands as long as the resulting substitution makes sense for the command. The following are some such examples:

```
$ echo *.txt (print the names of the files that end with .txt)
$ rm *.txt (remove/delete the files that end with .txt)
$ scp *.txt nanaeem@linux.student.cs.uwaterloo.ca:~/cs136/.
(secure copy the files that end with .txt to the remote machine's cs136 directory)
```

One interesting issue that comes about is what if we do not want an input to be interpreted as a globbing pattern. For example, what if we wanted to print the literal string *.txt. Typically, we can print strings by using the `echo` command and providing it with the string we want to print as a command-line argument. However, if we run the command

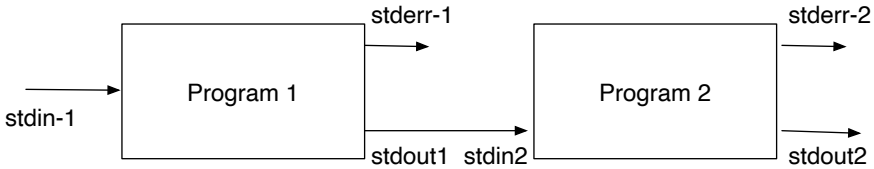
```
$ echo *.txt
```

as we discussed above, the Linux shell will notice the globbing pattern, substitute it with filenames that match the pattern and then echo those arguments. To suppress the shell's globbing pattern capabilities, we can use single or double quotes around the string. The following two commands will both print the literal string, *.txt.

```
$ echo '*.txt'
$ echo "*.txt"
```

6 Linux Pipes

Earlier we discussed output redirection; the ability to redirect the output generated by a program to a file. The Linux shell also supports the ability to redirect the output generated by a program to the input of another program. This is achieved by using a "pipe". A Linux pipe is used to connect the standard output stream of a program to the standard input stream of a program that executes right after the termination of the first program.



A key feature of the command-line shell is the ability to connect together basic commands to achieve more complex tasks. As an example, consider the task of counting the number of words in the first 20 lines of a file named `sample.txt`. There are two tasks at hand (1) getting the first 20 lines of the file (2) counting the number of words in these lines.

To achieve the first task, we can use the command named `head`.

Run the command `man head` and read the description.

By reading the manual, we learn that task 1 can be achieved by running the command
`$ head -n 20 sample.txt`

We already know a command that counts the number of words in a file. `wc` will print out the number of lines, words and characters in a file. But, we just want the number of words, not any additional information.

Run the command `man wc` to find a command-line argument to wc that will force wc to just print the number of words.

We learn that task 2 can be achieved by running the command
`$ wc -w 20lines.txt`
if the first 20 lines are placed in the file named 20lines.txt.

One solution is to execute the following sequence of commands:

```
$ head -n 20 sample.txt > 20lines.txt
$ wc -w < 20lines.txt
```

In words, run the `head` command which would output the first 20 lines on the standard output stream. Capture this output, through output redirection, into the text file `20lines.txt`. Then, run the command `wc` with the `-w` argument to just print the number of words. Give the input to the command using input redirection from the file we created in the earlier command (notice we could also have given the filename as an argument to `wc`).

While this solution works, we had to create a temporary file to store the output of Command 1 and then make it available to Command 2. The Linux pipe functionality allows us to directly

connect the standard output stream of Command 1 to the standard input stream of Command 2. The same result can therefore be achieved as follows (the Linux shell uses the | as the pipe symbol):

```
$ head -n 20 sample.txt | wc -w
```

Alternately, we could also write
`$ cat sample.txt | head -n 20 | wc -w`
In words, run the command `cat` which produces the contents of sample.txt on the standard output stream which are piped into the standard input stream of head which produces the first 20 lines on its standard output stream which are piped into the standard input stream of wc which produces the final result.

As mentioned earlier, this touches upon the core philosophies of the Linux command-line shell; there are a basic set of building blocks, in the form of commands that do simple things, which can then be composed together to do complex tasks.

Suppose files words1.txt, words2.txt etc., contain list of words, one per line. Print a duplicate free list of all the words that occur in any of the files, words*.txt

Hint: lookup the manual descriptions of the commands: uniq and sort.

Since we want a single list, we can begin with concatenating all the files using the `cat` command. However, this concatenated list of words is going to have duplicates. The Linux shell comes with the `uniq` command which can be used to eliminate duplicates from a list. The problem is, the `uniq` utility will only eliminate duplicates from adjacent lines. To solve this issue, we can first `sort` the lines in our concatenated list of words and then run the `uniq` command. As a Linux command pipeline, this can be achieved as follows:

```
$ cat words*.txt | sort | uniq
```

While the above is a good example of how multiple commands can be connected to each other using Linux pipes, another solution also exists. The `sort` command has a command line argument, `-u`, which will suppress additional lines that are duplicates of lines already processed.

You are running a contest and you have a file called "entrants.csv" with all the entrants. Each line in the file has the form `full_name,email_address,phone_number` and you can assume there are no duplicate entrants. You want to randomly select 10 entrants, and output a list of just their email addresses.

Hint: look up the manual pages for the `shuf` and `cut` commands.

As is obvious from the name of the command, `shuf` shuffles (creates random permutations of) the input lines. Since we only want 10 random entrants, we begin with the pipeline:
`$ shuf entrants.csv | head -n 10`

The only task left is to retrieve just the email addresses from the 10 randomly selected entrants. If you read the manual entry for `cut`, the command can be used to remove sections from each line of

files (or standard input). The `-d` option can be used to specify how the line is split up into "fields". For example, `-d,` (note the comma) tells the command that the fields are separated by commas. The `-f` option is used to specify which field(s) to extract.

```
$ shuf entrants.csv | head -n 10 | cut -f2 -d,
```

In words, the above Linux pipeline will shuffle the entrants, generating the shuffled output on standard output, which is then fed to the `head` command that outputs the first 10 lines to standard output, which is then fed into the `cut` command that cuts out the second field (where fields are delimited by commas).

There are other possible orders for the commands, but `shuf` should come before `head`, or else you are only shuffling a portion of the entrants. The `shuf` command itself has an option to limit the number of lines it outputs, so piping into `head` is not technically needed.

```
$ shuf -n 10 entrants.csv | cut -f2 -d,
```

7 Embedding commands

The Linux pipe allows us to connect the output of one program/command to the standard input of another. We can also embed the output of a command in a string, allowing us to provide the output of a command **as an argument** to another program/command. As an example, consider the following execution of the `echo` command:

```
$ echo "Today is $(date) and I am $(whoami)"
```

The `echo` command above is being executed with a single argument, by using the double quotes to indicate that everything within the quotes are a single argument to echo. More interestingly though, is the use of the `$( )` notation to embed the `date` and `whoami` commands. When the Linux Shell receives this input on the command-line, it executes the embedded `date` and `whoami` commands first and replaces these commands by the outputs produced by the commands. Assuming that the `date` command produces the string `Fri 25 Sep 2020 15:56:35 EDT` and the `whoami` command produces `nanaeem`, the resulting command and the output is:

```
$ echo "Today is Fri 25 Sep 2020 15:56:35 EDT and I am nanaeem"
Today is Fri 25 Sep 2020 15:56:35 EDT and I am nanaeem
```

Can you think of a difference between running the command:
`$ echo "Today is $(date) and I am $(whoami)"`
and
`$ echo Today is $(date) and I am $(whoami)`?

In this case, there is no difference; the end result is the same. However, from an execution perspective, the first of the two commands, the one which uses the double quotes, provides a single command-line argument to the `echo` command. The second command, the one without the quotes, provides multiple arguments, each argument separated by whitespace. While there is no difference for these particular invocations of the `echo` command, this is not always the case. Consider these two commands:

```
$ echo "Lots   of     spaces"
Lots   of     spaces
$ echo  Lots   of     spaces
Lots of spaces
```

The first command will preserve the spaces perfectly, since `echo` is given a single argument, which is a string containing many spaces. However, in the second command, `echo` is given three arguments, "Lots", "of", and "spaces". The behaviour of `echo` in this case is to output each argument in sequence, leaving a single space between them.

We have seen the `$( )` syntax for embedding a command within another. But what if we wanted the text to contain `$( )`? This can be achieved by placing the text in single quotes. For example, take a look at the following command and the output it produces:

```
$ echo 'Today is $(date) and I am $(whoami)'
Today is $(date) and I am $(whoami)
```

Because of the use of single quotes, the execution of the embedded commands was suppressed. Recall from before that both single and double quotes also suppress globbing patterns.

There is an alternate syntax for embedding commands within other commands:
`$ echo Today is` `` `date` `` `and I am` `` `whoami` ``
The commands `date` and `whoami` above are surrounded with backtick characters. The `$( )` syntax is more modern and is preferred for several reasons (for example, nesting is easier).

8 Pattern Matching in Text Files

It is often useful to search the contents within text files. The Linux Shell provides the `egrep`[3] utility. Paraphrased from the manual entry for `egrep`, the tool searches given files and outputs lines that match one or more patterns. Patterns are provided as (extended) regular expressions. It is important to realize that regular expressions are not the same as globbing patterns even though they sometimes use the same symbols.

We will not address the formal theory that underlies regular expressions. Instead, we will learn how to write regular expressions by example:

```
$ egrep CS136 test.txt
```

The `egrep` command above has been provided the regular expression `CS136` as the first command line argument and the filename `test.txt` as the second argument. The command will independently try to match the regular expression on **each line of** the provided file. Consider the following contents for the file:

```
The cs136 course provides an introduction to the imperative programming paradigm.
In CS136, we will discuss how the imperative paradigm provides an alternate way of
thinking to the functional paradigm you learned in cs135. CS 136 will provide an
introduction to imperative programming using a procedural language, C. In CS246,
you will learn Object Oriented Programming using C++, another imperative language.
```

Carefully look at the text above. Can you determine which lines the command will print out?

The regular expression `CS136` is an example of concatenation of the characters C, S, 1, 3 and 6. In other words, we are specifying that we want to find the lines that contain a substring with a C followed by S followed by 1 followed by 3 and then 6. The command above prints out only the second line from the text file since only this line contains the exact substring that matches our regular expression. The first line contain cs136 (with lowercase c and s) whereas the third line contains an occurrence where the CS and 136 is separate by the space character. In other words, `egrep`, by default, is case sensitive and respects whitespace.

Search through the manual entry for `egrep` and find a command-line argument that would make `egrep` case insensitive.[4]

One way for us to match the lowercase cs136 would therefore be to use the appropriate command-line argument. Alternately, we can enhance the regular expression to say match CS136 OR cs136.

```
$ egrep "CS136|cs136" test.txt
```

[3] You may have heard of the `grep` (Global Regular Expression Print) command before. The `egrep` command is an "extended `grep`" which allows using Extended Regular Expressions and has simpler syntax than `grep`.

[4] The -i command line argument makes egrep case insensitive.

The vertical bar symbol, when used within a regular expression is used to specify choice. In the regular expression above, we specify that we are looking for substrings that match CS136 or match cs136. Of course a line that contains both strings is also a match. An observant reader would have noticed that we placed the regular expression within quotations. These are necessary to indicate that the entire sequence of characters between the quotations is an argument intended for the `egrep` command. Without the quotes, the shell might interpret characters within the regular expression to have special meaning. Recall that the vertical pipe is the Linux Shell's symbol for piping the standard output stream of one program into the standard input stream of another; that of course was not the intent here. Double quotes will prevent the shell from interpreting any of the characters within the regular expression.

Parts of a regular expression can be grouped into a subexpression using parentheses. For example, the regular expression `cs136|CS136` can also be written as `(cs|CS)136`. The latter regular expression matches strings consisting of either the substring `cs` or `CS`, followed immediately by the substring 136.

In a regular expression, any one single character can be represented using a period. For example, `cs13.` (note the period) would match any substring that begins with `cs13` and has one additional character. Notice that this is different from a globbing pattern where the ? character represents any one symbol.

When the choice is between a single character from a set of characters, square brackets can be used as a shorthand. For example, the regular expression `[abc]` is shorthand for the regular expression `a|b|c`. Similarly, the regular expression `[cC][sS]136` is shorthand for `(c|C)(s|S)136`. Note that this regular expression is not equivalent to the regular expression `(cs|CS)136` from before since it matches the substrings `cS136` or `Cs136`. Inside `[ ]`, most characters with special meanings lose their meaning and are treated as regular characters.

Ranges can be used: `[0-9]` is equivalent to `[0123456789]`. There are also special "character classes" for commonly used sets of characters. For example, `[[:alpha:]]` is equivalent to `[A-Za-z]`.

To indicate any character *other* than those from a particular set, we can use the syntax `[^ ]`. For example, `[^abc]` would match any single character as long as it is not `a`, `b`, or `c`.

The * symbol can be used in a regular expression to indicate that the preceding (sub)expression can be repeated zero or more times. For example, `(cs)*136` would match the substrings `136` (0 repetitions of `cs`), `cs136` (1 repetition of `cs`), `cscs136` (2 repetitions of `cs`) and so on. Recalling that a period represents any one character, the expression `.*` can be used to represent arbitrary-length sequences of arbitrary characters. An important observation is to note the difference between what the * characters means in a regular expression vs. what it means in a globbing pattern.

The extended regular expressions that `egrep` supports also allow for using `+` after a (sub)expression to represent 1 or more repetitions of the preceding pattern, i.e., it disallows the "0 occurrences" that would be allowed by the `*` symbol. In other words, `a+` is equivalent to `aa*`. There is also the `?` symbol which represents zero or one occurrences of the preceding (sub)expression. For example, the regular expression `cs ?136` (notice the space between the `cs` and `?`) allows matching the substring `cs136` and `cs 136`. It is important to notice that these symbols apply to the preceding

(sub)expression. For example, the regular expression `(cs)?136` would indicate that the substring `cs` is optional, i.e., this expression would match just `136` or `cs136`.

As discussed at the beginning of this section, `egrep` looks for substrings within each line of a text file for matches to the provided regular expression. In other words, if the matched substring is anywhere within a line, `egrep` will produce that line as an output of the tool. However, we can force the tool to only consider matches if the matched string begins at the start of the line. The command

```
$ egrep "^CS136" test.txt
```

will only match lines in `test.txt` that begin with `CS136`. Notice that this does not mean that the line may not contain anything else; it must begin with `CS136` and can be followed by anything. Similarly, we can use the $ symbol to force that the matched substring must go till the end of a line. For example, the command:

```
$ egrep "CS136$" test.txt
```

will match lines where the string CS136 appears at the end of each line. Of course the two symbols can also be used within the same regular expression.

Since a number of characters have special meanings when used within a regular expression, it is useful to be able to specify when we do not want a character to be treated as special. Characters with special meanings can be `escaped` using a backslash. For example, to match an actual period, one must write `\.` in their expression.

We conclude this section with a few examples.

Search all words in the file `/usr/share/dict/words` that start with e and consist of 5 characters.

```
$ egrep "^e....$" /usr/share/dict/words
```

Notice that it is important to start the regular expression with ^and end it with $ as that insists that a line should only be matched if all the characters in the line satisfy the pattern. Without it, the regular pattern could match a substring of a longer string which is not what the questions asks.

Give the regular expression for fetching lines in a file that have even length.

The answer is `"^(..)*$"` where once again we see the importance of requiring that all characters in each line be part of the pattern. Without it, i.e., the regular expression `(..)*` searches for **substrings** with 0 or more pairs of characters within a longer string. That is not what we want. For example, the regular expression `(..)*` would match `abc` as it could treat `ab` or `bc` as the substring which does have even characters.

List files in the current directory whose names contain exactly one a.

In this question we are not searching within a file, instead we want to search the current directory. While we could use a globbing pattern to do that, we can also use `egrep` with an appropriate regular expression. First, we get the list of all files in the current directory simply through the `ls` command. The `ls` command generates output on the standard output stream. We can pipe this output to become the input that `egrep` searches; instead of searching a file provided as a command-line argument, `egrep` can also search content coming in on the standard input stream. The following command-line pipeline achieves the task:

```
$ ls | egrep "^[^a]*a[^a]*$"
```

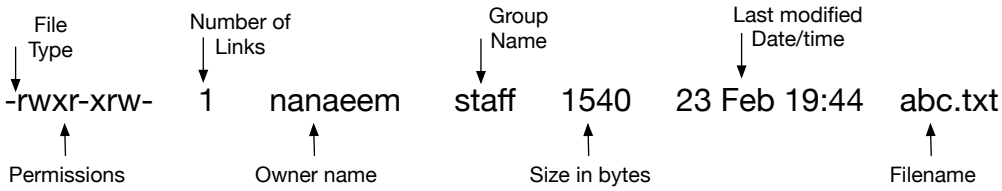
The regular expression itself is also worth exploring in more detail. Notice the use of `^` and `$` to indicate that the regular expression will look at all the characters in the line and not a substring. The regular expression can be read as matching 0 or more characters other than `a` (via `[^a]*`), followed by a single `a`, and then followed by 0 or more characters other than `a` (via another `[^a]*`).

9 File Permissions

Every file in the file system comes with some permissions that can be changed by the owner of the file. To see the permissions currently assigned to a file we can use the `-l` command-line argument to the `ls` command. The command

```
$ ls -l
```

will provide the `long form` listing of all non-hidden files in the current directory.



While the long-form listing provides some other useful information such as the group the file belongs to (a group is a collection of users who are able to share certain files or resources), its size, and its last modified date/time stamp, we are interested in the file permissions. These are represented by nine characters in the first column of the output of the long-form listing.[5]

Let's take a closer look at the 9 characters that represent file permissions:

$$\underbrace{rwx}_{\text{user bits}}\ \underbrace{r-x}_{\text{group bits}}\ \underbrace{rw-}_{\text{other bits}}$$

[5]The first column actually has 10 characters; the first character is used to indicate the type of the file e.g. - is used to represent an ordinary file, d is used to indicate a directory. The remaining 9 characters are the file permissions.

As noted above, these 9 characters are actually 3 sequences of 3 characters each. Each represents permissions for specific subsets of users. The `user bits` are permissions for the owner of the file, the `group bits` are permissions assigned to users (other than the owner) who are in the group to which this file belongs, and the `other bits` are permissions for all other users.

Within each sequence, the first character can either be a r (read permission granted) or - (read permission not granted). The second character within a sequence can either be a w (write permission granted) or - (write permission not granted). The third character in each sequence can be x (execute permission granted) or - (execute permission not granted).

Based on this, the file `abc.txt` above grants the owner read, write and execute permissions. It grants members who are in the `staff` group (other than the user `nanaeem` who is the owner) read and execute permissions but not write permissions. All other users are granted read and write access but not execute.

Of interest is of course what it means for a file, and in particular directories, to actually have some of these permissions. Recall that we use the term "ordinary file" for files which are not directories (since directories are considered a type of file that contains other files). For an ordinary file, permissions are fairly intuitive. The read permission means that the ordinary file's contents can be read, e.g., you can run the `cat` command, or open the file in a text editor. The write permission allows the contents of the ordinary file to be modified, e.g., edited in a text editor. The execute permission allows the ordinary file to be executed as a program (if it is actually a program). If you write a custom program or script you will need to make sure the execute permission is set before you can run it (though compilers often do this automatically).

Let's now talk about how file permissions pertain to directories. If one has a read permission for a directory, the contents of the directory can be read, e.g. we can run the `ls` command, or use a globbing pattern. If a write permission has been given, then the contents of the directory can be modified, i.e., we can add/remove files. The execute permission on a directory controls whether a user can navigate into a directory, e.g. change to that directory using the `cd` command. Directory permissions make for some interesting cases. For example, if the execute permission is not available, one cannot enter that directory, which means one cannot access any file or subdirectory within. Alternately, if the execute permission is granted but the read permission is not, one can enter the directory but not see the contents. But, if the user happens to know that a particular file exists in this directory, they can access that file (of course subject to the permissions of that file).

9.1 Changing Permissions

The owner of a file is the only user that can change a file's permissions.[6] The `chmod` command is used to change file permissions. The first argument to the command is the `mode`, i.e., the permissions to be applied. After this, the command can be given one or more files to which the permissions will be applied. The mode has three components: (1) the ownership class (2) the operator and (3) the permission. The following tables specify different combinations that can comprise the mode:

[6] The `chown` command can be used to change ownership.

Ownership	
u	owner
g	group
o	other
a	all

Operator	
+	add permission
−	revoke permission
=	set permission exactly

Permission	
r	read
w	write
x	execute

Let's take a look at some examples. The command:
`$ chmod o+r file.txt`
adds the read permission for others to the file `file.txt`. Recall, that others means those who are not in the group that the file belongs to.

The command:
`$ chmod g-x *.sh`
revokes the execute permission for all group members (other than the owner) for all files that match the globbing pattern `*.sh`, i.e., all files that end with `.sh`.

The command:
`$ chmod a=rx file`
sets the permissions for all users to be exactly read and execute. Note the distinction between setting the permission using the = operator as opposed to adding the read and execute permissions using the + operator. By setting the permissions, we are implicitly revoking the write permissions from users who previously had the permission.

There is an alternate way of specifying the mode for the `chmod` command. It is to use a three digit value with each digit ranging between 0 and 7. The idea behind using digits to specify the mode is that when each of the three digits is independently converted to its binary equivalent, a 0 would indicate that the corresponding permission is not given whereas a 1 would indicate the permission is granted. For example, 756 would translate to 111101110 since 7 is 111, 5 is 101 and 6 is 110 in binary. This corresponds to the permissions rwxr-xrw- as per the way permissions are displayed by the long listing command. While the syntax above is arguably much more intuitive, this numeric syntax is still used and you may see it around.

10 Terminal shortcuts

Over time, the more you use the command line shell, the faster you will get at it. You can further improve your experience and can significantly improve your productivity by using some shortcuts. We discuss a few here:

Tab completion: Those who know about tab completion can no longer live without it! Anytime while typing a command, a file path, directory name or even command options, hitting the tab key will either automatically complete what you were typing or will show all the possible results that could complete what you have already typed.
Ctrl + c: A number of shortcuts are actually control sequences, i.e., you must keep the Ctrl button pressed and then in addition press another key. The Ctrl+c control sequence is a way to tell the shell to terminate the currently active process/program. Say you are executing a program and it is taking too long e.g. stuck in an infinite loop, or stuck for some other reason. You can press Ctrl+c to terminate/kill the process.

Ctrl + d: While we are listing this as a shortcut it is more than that. The Ctrl+d control sequence is used to communicate to the currently executing process that the standard input stream has been exhausted, i.e., there is no more input expected on the standard input stream. Often this is referred to as sending an End-Of-File signal to the process. This will allow programs that are waiting for input to terminate cleanly. If you are running a program that takes standard input from the keyboard, you should enter Ctrl+d when you are done typing so that the program can start processing the input.
The following table lists some other shortcuts with brief descriptions:

Ctrl+Z	Sends a running process to the background thereby freeing the shell to execute other commands. The process moved to the background can be brought to the foreground by using the `fg` command.
Ctrl+L	This is a shortcut for the `clear` command that clears the screen.
Ctrl+A	This is a shortcut for keeping the left arrow key pressed to move the cursor to the start of the command already typed on the prompt. With one control sequence the cursor will move to the start.
Ctrl+E	Move the cursor to the end of the command already typed on the prompt. This is often used after a user has used Ctrl+A to move to the start of the command to make a change and then needs to move back to the end.
Ctrl+U	If you have typed the wrong command, you have the option to use the backspace button repeatedly to erase it or you can use this shortcut to clear whatever you have typed on the command prompt.
Up arrow or Ctrl+P or PgUp	The Up arrow key or Ctrl+P or PageUp can be used to view the previously executed command. Repeatedly using the shortcut goes backwards through the history of executed commands.
Down arrow or Ctrl+N or PgDown	This is used in conjunction with the previous shortcut to go forwards through the history of commands.
!!	This executes the most recently issued command.
!string	This executes the most recently issued command that begins with `string`.
Ctrl+R	This lets you search the history of executed commands for a command and re-execute it. Once the user presses Ctrl+R anything the user types afterwards will be used to search the history and the best match will be displayed. The user can also press Ctrl+R to cycle through commands that match.

11 Terminal Configuration

The Command Line Shell can be customized. One common customization is to change what is shown as the command prompt. We have been showing the command prompt with just the `$` symbol. Your command prompt might look different. For example, a common default is `username@hostname$`. You can configure it to display whatever you like and even choose your own colours. To do this we must update a shell variable named `PS1`. We will discuss shell variables in more detail in another module on shell scripting. For now, we dive right into assigning the `PS1` variable to customize our command prompt. Consider the following assignment to the variable:

```
$ PS1="Laptop Prompt $"
```

(Note that there is no whitespace between `PS1`, `=` and `Laptop Prompt $`. This is actually necessary; it is a requirement of the shell's syntax for variable assignment.)

If you execute the command above, your command prompt would change to:

```
Laptop Prompt $
```

Now that we know how to change our prompt, we try the following popular configuration:

```
$ PS1="\u@\h:\w\$"
```

The \u indicates the user's username, the character @ appears as itself, \h represents the hostname, i.e., the computer name, the character : appears as itself, \w is for the present working directory and the character $ appears as itself. Performing the above assignment on my laptop results in the following command prompt:

```
nanaeem@Nomairs-Laptop:~$
```

Which makes sense since my userid is `nanaeem`, the computer name is indeed "Nomairs-Laptop" and since I am currently in the home directory, it is represented by the `~` after the colon which is used as a separator. Now, if I was to use the cd command to enter into the CS136 directory, let's see what happens:

```
nanaeem@Nomairs-Laptop:~$ cd CS136
nanaeem@Nomairs-Laptop:~/CS136$
```

Notice that only one command was executed, `cd CS136` which has the expected effect that our working directory is changed to `~/CS136`. But what is interesting to note is that this present working directory is now displayed as part of our command prompt; we have achieved the goal of always knowing where we are within the file system. We will let the interested reader research how they can colour code different parts of the command prompt differently.

One issue with our above discussion is that the way we have customized the command prompt is only applicable for our current session. As soon as we close the Command Line Shell, our customized command prompt would go away. There is a straightforward way to make the change permanent.

11.1 Command Line Shell Configuration files

When a new command line shell is started, part of the startup process includes the shell executing configuration commands stored in a hidden text file in the user's home directory.[7] Which configuration file is chosen is dependent on a number of variables including how the Linux system was configured and also whether another shell is already running. However, it is safe to assume that one of the following files will be executed: `~/.profile`, `~/.bash_profile`, `~/.bash_login` or `~/.bashrc`. If your search reveals that you have more than one of these files, then an easy way to find out which file is used is to edit each of these files and place a command such as `echo "In file <name of file here>"`. Then depending on the message that gets printed when you start a new shell, you know which configuration file executes for you.

Going back to our example of customizing the command prompt, we can place the assignment of the `PS1` variable in one of these files, so that this customization happens every time a session is started. In what follows, we assume that the `~/.profile` file is executed at startup. If your environment causes some other file to be used, you should substitute that file in the examples below.

A good first step is to make a backup of the file we are about to edit using the `cp` command:

```
$ cp ~/.profile ~/.profile_backup
```

Now, we can edit `~/.profile` and if we mess up, we can always replace our messed up version by copying from the backed up file. At the very end of the file, we can add the line that assigned to the `PS1` variable that we discussed in the previous section to customize the command prompt:

```
PS1="\u@\h:\w\$"
```

(We are adding this line at the end of the file, not typing it on the command prompt.)

Once we have permanently placed the above in the file `~/.profile` which runs every time the shell starts, the customization will apply every time and we will see our customized prompt.

[7] Recall that hidden files are simply files that start with a period. They do not show up by default when the `ls` command is executed, unless the `-a` argument is used.

11.2 Running other shell scripts as part of configuration

We will discuss writing our own shell scripts in a later module. Very briefly, shell scripts are text files that contain a collection of Linux commands that can be executed as a program. Since we are at the topic of configuring the Command Line Shell, it is worth discussing one other command:

source: `source` is a built-in command that allows users to read and execute the content of a text file that is passed as an argument to `source`. Typically, the text file contains other commands that are executed.

As an example of how this command is often put to use, imagine the case where a user adds their customizations in a separate file called `my_customizations`. Then, in their `~/.profile` file (or another one of the files that might run on startup) they can add the command:

```
source my_customizations
```

When the Linux Command Shell starts up, it runs the user's `~/.profile` which in turn runs the `source` command that the user has added, which in turn runs whatever is in the file provided as an argument.

This is a convenient way for software programs to provide environment configuration commands to their users. All the configuration is made available in a file and all the user has to do is to manually `source` that file or alternately place it in their `~/.profile` to have it "sourced" automatically.

12 Concluding Remarks

Our intent in this module was to give you a brief introduction to the Linux Command Line Shell. We discussed the Linux File System and some basic commands to navigate through the file system and manipulate files. We also looked at commands to connect to remote systems and copy files from such systems. We discussed streams attached to a Linux process, input/output redirection, and Linux pipes. We covered pattern matching using globbing patterns, and searching using regular expressions with the `egrep` command. Finally, we discussed shortcuts that make using the command line easier, and some techniques for configuring the shell to your liking.

We briefly alluded to the possibility of creating custom command line shell scripts. This is very useful for automating tasks, but is quite a complicated subject, and will be discussed in separate module.

On the next two pages, we provide a summary of a number of commands, tools and programs. Some of them were discussed in this module. Others are left for the reader to explore on their own.

Commands

Command	Meaning	Options
exit	log out	
passwd	change your password	
clear	clear screen	
man *command*	show the manual page for *command*	man -k *word* show a list of man pages that mention *word*
history	show all previously-issued commands	
whoami	display your login name	
date	display current date and time	
pwd	display current directory	
ls	list contents of current directory	ls -a show all files, including hidden files ls -l show in long format
cp *file1 file2*	copy *file1* to *file2*	cp -r *dir1 dir2* recursively copy *dir1* to *dir2*
mv *file1 file2*	move *file1* to *file2* (also use to rename)	
rm *file*	remove *file*	can be used to recursively remove a directory, if -r option is used
touch *file*	update *file*'s last modified time to current time	can be used to create an empty file if *file* does not exist
cd *dir*	change directory to *dir*	cd - return to most recently visited directory
mkdir *dir*	create new directory *dir* in current directory	can specify more than one directory at once
rmdir *dir*	remove directory *dir*	only works if *dir* is empty; if not empty, use rm -r *dir*; can specify more than directory at once
echo *string*	display *string* to screen	
chmod *perms file*	set permissions on *file* to *perms*	
ps	display current processes	ps -a show all users' processes ps -A show ALL processes (incl. system processes)
kill *pid*	kill process with number *pid*	kill -9 *pid* more forceful kill, for stubborn processes
who	show who is logged into this machine	
time *command*	show amount of time taken executing *command*	
fg	bring background job to the foreground	useful if you accidentally ran vim or emacs with an & (causing it to run in the background)
find *dir* -name "*pattern*"	find all files whose names match *pattern* in *dir* and its subdirectories	

Tools

Tool	Purpose	Options
cat *f1 f2 ...*	display files *f1*, *f2*, ... one after the other	cat -n *f1 f2 ...* attaches line numbers
less *file*	display *file* one screen at a time	
diff *f1 f2*	compare files *f1* and *f2*; outputs instructions for converting *f1* to *f2*	diff -w *f1 f2* ignores whitespace
cmp *f1 f2*	compare files *f1* and *f2*; outputs the first position where they differ	
wc *file*	count the number of words, lines, and characters in *file*	wc -c *file* show just the number of characters wc -l *file* show just the number of lines wc -w *file* show just the number of words
egrep *pat file*	print all lines in *file* that contain pattern *pat*	egrep -n *pat file* print matching lines with line numbers egrep -v *pat file* print lines that do *not* match *pat*
head *file*	print first 10 lines of *file*	-*num* prints *num* lines (e.g. head -5 *file*)
tail *file*	like head, but prints last 10 lines of *file*	tail -n +N instead of last 10 lines, output starting from line N
sort *file*	sorts the lines of *file*	sort -n *file* sorts strings of digits in numerical order
uniq *file*	removes consecutive duplicate lines from *file*	removes all duplicates if *file* is sorted
shuf *file*	randomly shuffles the lines in *file*	shuf -n *num* outputs *num* random lines from *file*
cut *options file*	extract portions from each line in *file*	cut -fN -dD extract field N from each line, where fields are separated by a delimiter D

Programs

Program	Purpose	Options
vim *file*	invoke vi IMproved text editor on *file*	
vi *file*	invoke vi text editor on *file* (often redirects to vim)	
emacs *file*	invoke emacs text editor on *file*	
nano *file*	invoke nano text editor on *file*	
wget *url*	fetch file from the web at *url*	
xpdf *file*	display pdf file (requires the ability to launch graphical programs)	
ssh *machine*	make SSH connection to *machine*; opens a secure shell on remote *machine*; type exit to end SSH connection	ssh -Y (or -X) *machine* enable X forwarding to allow use of graphical programs (must have X server running on local machine)
scp *mach1*:*file1* *mach2*:*file2*	securely copy *file1* on *mach1* to *file2* on *mach2*	can omit *mach1* if it is the local machine; similarly for *mach2*