Chapter 3 The Shell

From Chapter 1, we've seen a bit of the Shell already. But that raises the question of what even is the shell in the first place. Ultimately, the shell is just another program that reads data from an input (the keyboard), and interprets that keyboard input to run as commands.

You might wonder what the point of a shell is. Why don't we just use a GUI? It's very well that in many cases we could just use a GUI. They are often times significantly more user friendly (or at least they try to have a great user experience), however, there are some drawbacks:

- GUIs are computationally expensive. Rendering a program on the screen can cost a lot of extra power.
- When logging into a remote system, it is much simpler to use a CLI. Remote GUIs take more time to run.

3.1 Bash:

If you remember from the Chapter 1.3, we know there are a ton of different shells out there. Bash, the *Bourne Again Shell* is a shell from 1987. It has been improved over the years, but still holds to what it was back. To verify that you're using bash, type:

```
1 echo $0
```

> bash

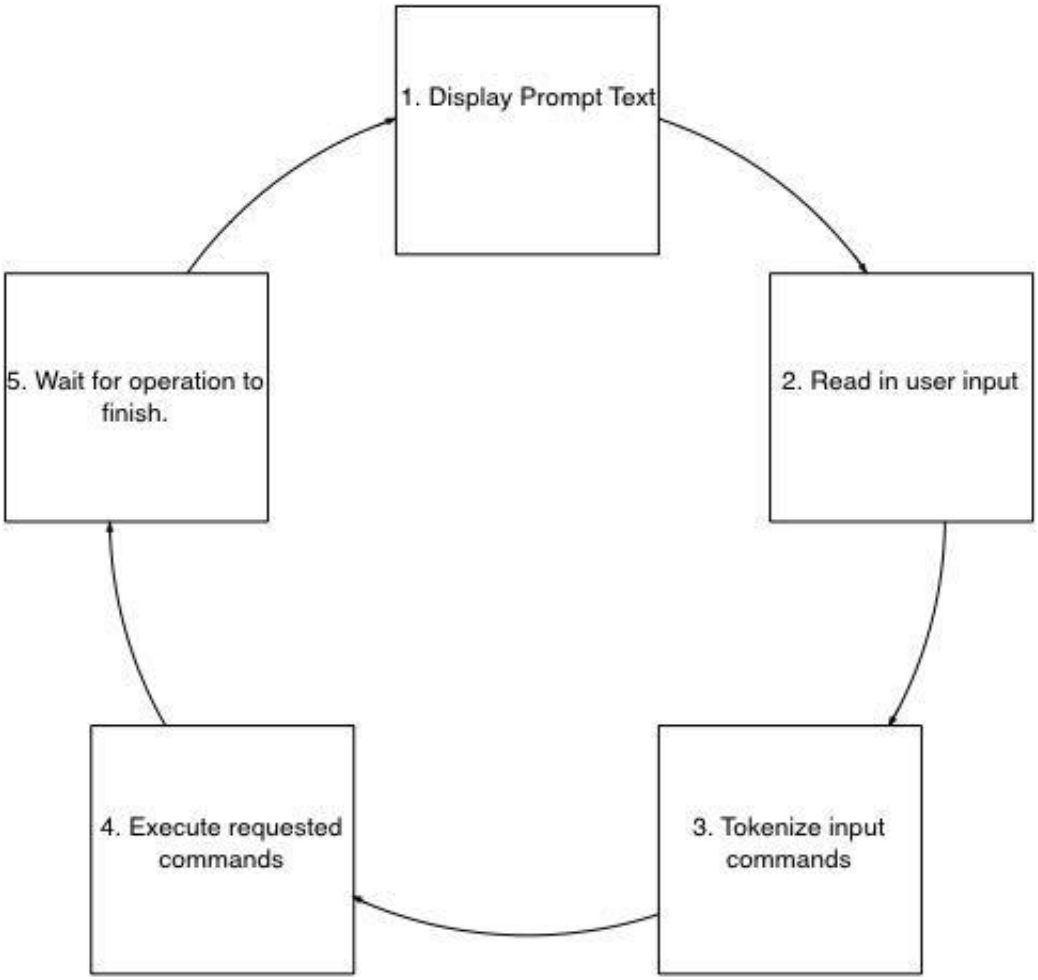
If you are not using Bash, use the change shell command `chsh`:

```
1 chsh -s /bin/bash
```

By using the above command, you're telling your system to set your shell to the bash shell (as opposed to any other).

3.2 Interacting with Bash

Without diminishing the power of bash, it is ultimately a fancy loop:



A command in the command line is one or more words separated by white space. A command is completed with a new line (unless escaped with a back slash).

```
command argumentOne argumentTwo
```

Commands

There are two different types of shell commands:

- **Shell Built In:** These commands invoke a routine that is part of the shell's program itself (e.g. `cd`) or any function / alias defined by the user (we'll get to those shortly). When executing a built in command, the shell just calls whatever subroutine, and moves on.
- **Regular Command:** Any other executable program in the linux environment not built into the shell, such as `ls`, `rm`, `cp`, or any executable file that has been compiled

Processes

All executing programs are called "processes". All processes are managed by the OS. The shell itself is a process. Whenever the shell runs another program (running a command, opening a new window, etc), the shell creates a new process, often called a "child process".

Simple Commands

A simple command is quite literally any command followed by arguments or options. Even though a new line character denotes the end of a command, a semicolon does the same thing. Several simple commands can all be on the same lien so long as they are separated by a semicolon.

```
command1 argument; command2; command3 anotherArgument
```

The shell reads commands from left to right. The above code is the same as writing:

```
command1 argument
command2
command3 anotherArgument
```

Compound Commands

It's not always the case that you'll have one command right after the other. There are ways to make our command structure a bit more complex. A *compound command* is any two or more commands combined together. What does that mean?

Compound commands can be combined by:

Example	Operator Name	Description
cmd1 | cmd2	Pipe **|**	Controls the flow of input / output (discussed in section 5)
cmd1 || cmd2	Or operator **||**	Executes cmd2 only if cmd1 fails
cmd1 && cmd2	And operator **&&**	Executes cmd2 only if cmd1 succeeds

How does the shel know if a command fails or succeeds? Linux commands have exit statuses. If a program succeeds, it returns 0, otherwise, any other returned value is a failure.

Type Ahead

Sometimes your commands may take a moment to complete. Bash allows for us to continue typing the next command while we wait for the shell to complete the previous command. Once the shell completes command and re-prompts you, what you were typing the entire time is already there.

Detaching

Suppose you wanted to run a program, but didn't want to do it in the foreground. You can do exactly that by detaching. This will kick off a job, and then run that job in the background while you do work on other processes! In order to do this, you only need to add an ampersand at the end of the command:

```
cmd argument &
```

Then Shell will then run the command detached from the shell that you're currently working in.

Using a text editor, create a file called `takeforever.sh`, and type:

```
#!/bin/bash

echo starting to work!
sleep 5
echo doing more work!
sleep 3
echo finishing up
sleep 20
echo done!
```

Once you've saved the file, make sure to change your file permissions so you can execute the file `chmod 700 takeforever.sh`. Now, run it in the background:

```
./takeforever.sh &
```

> [1] 29870

The output here is the job number and the job id! Yours will likely look different (though the number will be the same). By now, you've likely noticed that, even though we've put the process into the background, it's still printing to the console! Background processes will still print to the foreground console!

In the event you want to bring the background process back, you can either type:

```
fg jobID
```

or

```
fg jobNumber
```

We'll discuss more on this in section 6.

3.3 Command Line Editing

You can set how you use the command line (that is, how the command line is edited), however, I would recommend against not doing this. Getting used to the default settings first is always a good thing before you start to update your command line.

When you're working in the command line, suppose you know which program you would like to use, but you forgot exactly how it was spelled. You know it started with a letter. You can start typing, and then immediately press `tab` twice, and it will display the options for autocompleting:

```
di          #[tab] + [tab]
```

> diff diff-jars dig dircolors dirs diff3 diffstat dir dirname disown

The command line can complete not only command names, ,but also file names, user names, host names, and variable names (more on variables later).

3.4 Command Line Execution

Bash commands can invoke a procedure within the shell, an alias, a built in command, or an executable program independent of the shell. The first word of a command is the command name itself:

```
command argument1 argument2
```

The command name is always some executable file. The command name can be:

- The absolute path to the file.
- The relative path to the file.
- An executable file on the search path (where the file name is the command)

How does the shell know where to look? The shell follows the following steps in order:

- Looks for the absolute/relative path to the executable file.
- If no file found, the shell takes it on itself to search the "command search path".

    - The first matching filename it finds along the search path is the one it uses.
    - If no file is found, or the user doesn't have permissions to execute the file, an error is thrown.

What is the command search path? The command search path can be found as the variable `$PATH`:

```
1 echo $PATH
```

> /usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/mjlny2/.local/bin:/home/mjlny2/bin

What are we looking at? This is a collection of paths to directories holding executable files. The paths themselves are delimited by `:`, so in the response we can see we have:

- `/usr/local/bin`
- `/usr/bin`
- `/usr/local/sbin`
- `/usr/sbin`
- `/home/mjlny2/.local/bin`
- `/home/mjlny2/bin`

In the event the shell can't find the command from absolute or relative path, it will one by one search each of those directories until it finds a matching command. These directories will be searched from left to right, and the first matching command it comes across will be the command it executes.

Exporting A New Path

It is possible to modify the search path (though, not recommended to delete anything, only add to it). Typically, adding a new path to the `PATH` variable looks like:

```
1 export PATH=$PATH:/new/path/to/search
```

What's happening above is that the export is overwriting `PATH`, but by include `$PATH` as a prefix to our new path, we're taking the old path, and just adding the new search path of `/new/path/to/search`.

Let's try this for real. In your home directory, create a directory `mybinaries`. Inside, create a file called `history2`. Inside that file, write:

```
#!/bin/bash
echo "Did you ever hear the tragedy of Darth Plagueis the Wise? I thought not. It's not a story the Jedi would tell you. It's a Sith legend. Darth Plagueis was a Dark Lord of the Sith, so powerful and so wise he could use the Force to influence the midichlorians to create life... He had such a knowledge of the dark side that he could even keep the ones he cared about from dying. The dark side of the Force is a pathway to many abilities some consider to be unnatural. He became so powerful... the only thing he was afraid of was losing his power, which eventually, of course, he did. Unfortunately, he taught his apprentice everything he knew, then his apprentice killed him in his sleep. It's ironic he could save others from death, but not himself."
```

Now, if you change directories back to your home directory and type "history2", nothing ought to happen. However, if you add your new `mybinaries` to your path:

```
export PATH=$PATH:/home/ENTER_YOUR_USERNAME_HERE/mybinaries
```

Now, if you type `history2` you don't need to specify relative path or anything of the like! The shell searches through the paths to find the matching command!

Note: Make sure that you add your username in the above code where it says `ENTER_YOUR_USERNAME_HERE`

What should be also be noted is that when you `export`, that only lasts as long as your shell session. If you want to keep your path forever, you'll need to update your `.bash_profile`. Open your favorite text editor in your home directory and edit your `.bash_profile`.

Once inside, add the above line of code to your `.bash_profile`. This file gets run every time you open up a new session, so it will always export the new path right as you start your session up! Granted, right now, we didn't start a new session, so we'll need to source the `.bash_profile` to update our environment:

```
source .bash_profile
```

Now when you log back in, you'll be able to use the `history2` command.

Which Command

Because there are a number of different paths that we search with the shell, it's possible that you're going to be using the wrong command. If things feel off, or if you're just curious, you can use the `which` command to figure out if you're using the command from the proper binary files:

```
which cmd
```

So, if we were to try that with `history2`:

```
which history2
```

> ~/mybinaries/history2

When an Executable is Run

When the shell ultimately finds an executable file (like how it found our `history2`), the shell doesn't just "run" the program. The shell itself is a program doing its own thing. What it does instead is:

1. Creates a new child process that is a complete copy of the parent (i.e. they have the same environment variables).
2. The child process runs the command and takes in any of the arguments passed to it.
3. The main process waits for the child process to finish its job and retrieve the data (or prints to the command line).