The Unix Programming Environment

An operating system is the main software that your computer runs on, like Windows, macOS, or Linux.

The Unix programming environment is a set of similar operating systems which are commonly used for software development. Its development began about fifty years ago at Bell Labs, and it is still the foundation of most modern development workflows. This section will introduce you to the environment, and show you how to do some useful things.

The Unix Philosophy

First, let's discuss the "Unix Philosophy". Below is a set of four points which outline the design principles of unix operating systems, and software development in general.

1. Make each program do one thing well. To do a new job, build new rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet another unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled work, to help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

This is commonly summarized in this more digestible format:

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface

This advice is not meant to apply to all types of programs, but it will get you far, and it will help you understand Unix's approach to development.

Operating System, Terminal, Shell

First, let's clarify some terminology.

An **Operating System** is software on your computer that *other* software runs on top of. The operating system has special control over other programs, which is necessary for security and organization. For example, suppose there is some malicious or bugged piece of software on your computer. A good operating system will let you 'kill' that piece of software so it does not compromise all of

your other work. Similarly, your computer only has so much hardware available to the good programs which must be divvied up. This is another problem that the operating system solves.

A **Terminal** is any program running on your operating system that allows you to read text based output from programs.

A **Shell** is any program that runs on a terminal and allows you to interact with programs interactively via text commands. We will focus on learning a shell called `bash` but that being said, most of what we learn will transfer directly to other shells. Bash is by far the most commonly used, since it comes pre-installed as the default shell in the most popular Linux distributions and has some of the most expansive documentation.

From those definitions, a terminal and a shell might sounds pretty similar and they are. Put more simply: The terminal and shell both live on the operating system. The shell is what allows you to interact with programs. The terminal is simply a middleman that transcribes your input from the keyboard to the shell, and then takes the shell's output and and displays it to your monitor.

Directories and Files

Almost all operating systems use the concept of directories and files.

A file contains data stored on a hard drive.

A directory, also commonly referred to as a folder, contains a collection of directories and files.

Because directories may branch off into more directories, one may refer to a directory and its contents as a tree, with files and emtpy directories being leaves. For example, I could describe a directory visually like this:

```
/directory_1/
    directory_1/
        directory_1/
    directory_2/
        file_1
        directory_3/
    file_1
```

With this being its "English" representation

```
* A directory named directory_1, containing:
    * A directory named directory_1, containing:
        * A directory named directory_1 containing nothing
    * A directory named directory_2, containins:
        * A file named file_1
        * A directory named directory_3, containing nothing.
    * A file named file_1
```

Note that the names do not have to be unique if two things are in different directories. This is because when we specify a folder or file, we can specify it with its "path" from the "root" of the tree. For example one file could be specified by `/directory_1/directory_2/file_1` while the other could be specified by `/directory_1/file_1`. You could think of it like a family tree, referring to directories which come before as **parents**, and directories which come after as **children**. Keep in mind though that any child in this case has one and only one parent.

It is important to understand that a shell is always *inside* a **working directory**, from which you can specify files by their **relative** path. Going back to the previous example, if I were using a shell *inside* `directory_2`, I could take a shortcut and just write `file_1` instead of `/directory_1/directory_2/file_1`.

There are some other drectory shortcut symbols in most shells, including bash:

- `.`: The **working directory**
- `..`: The **parent** of the working directory
- `~`: The **home** directory.
- `-`: The **previous** working directory

Getting Setup

Getting setup with your terminal depends on your operating system, and some support it better than others. This section provides a short guide on getting set up in Linux, macOS, and (most painfully) Windows.

Linux

1. Press `Ctrl+Alt+T`.

macOS

1. Press `+space`.
2. Search for and open `terminal`.
3. (Optional) Install *iTerm2* for a better experience.

Windows

I recommend Cygwin which you can download directly here

1. Choose all default options.
2. When it asks you to choose a download site, pick any.
3. When it asks you to select packages, search for `gcc-g++`, navigate into `All` -> `Devel` -> `gcc-g++` and select the latest version from the dropdown
4. Now search for `make`, navigate to `All` -> `Devel` -> `make` and select the latest version

5. Now search `python`, navigate to `All -> Python`. Select the latest versions for `python38`, `python38-devel`, `python38-pip`, `python38-setuptools`, and `python38-virtualenv`.

If you miss one of these steps, you can always run the executable again to install it.

Using a Shell

Now that we've gotten all the definitions out of the way, let's take a look at what it actually means to use the shell.

> Interactive Demonstration

`pwd`

First, you might be able to tell your current directory by just looking at the prompt. But in case you can't, you can explicitly print the working directory with `pwd`

`man`

Format and display the on-line manual pages

Examples

- `man man`
- `man bash`
- `man ls`

`ls`

List all directory contents

Examples

- `ls`
- `ls .`
- `ls ..`
- `ls ~`

`mkdir`

Create a directory

Examples

- `mkdir MyDir`

`cd`

Change (navigate into a) directory

Examples

- `help cd`
- `cd MyDir`
- `cd .`
- `cd ..`
- `cd ~`
- `cd -`

`touch`

Create file or change timestamp on existing file

Examples

- `ls -ld .`
- `touch .`
- `ls -ld .`
- `touch NewFile`

`echo`

Print to standard output (your screen)

Examples

- `echo "hello"`

`cat`

Concatenate and print files

Examples

- `cat`
- `cat data/sample.txt`

`cp`

Copy files

Examples

- `cp NewFile CopiedFile`
- `cp -r Mydir CopiedDir`

`mv`

move (or rename) files

Examples

- `mv NewFile SameFile`

`rm`

Remove files

Examples

- `rm CopiedFile`
- `rm -r CopiedDir`

Be careful, these do not go to your trash so they are fully unrecoverable.

`chmod`

Change file modes (permissions)

Examples

- `chmod +x hello.sh`

Other Commands

You will likely come across other commands not mentioned here. Remember, `man` is your friend. Some other common commands are:

`ln`, `chown`, `find`, `du`, `df`, `less`, `head`, `tail`, `grep`, `sort`, `wc`, `diff`, `pkill`, `top`, `time`, `sudo`, `alias`

Wildcards

`man bash` (Search for **Pathname Expansion**)

Examples

- `echo *`
- `ls *.md`
- `ls *.??`

Redirection

Examples

- `echo "Hello, world" > hello.txt`
- `echo "Hello, you" >> hello.txt`

Pipe

Examples

- `ls | sort`
- `cat roster.csv | sort`
- `cat roster.csv | sort | cut -f1 -d ','`
- `cat names.csv | sort | cut -f1 -d ',' | cut -f2 -d ' '`
- `cat names.csv | sort | cut -f1 -d ',' | cut -f2 -d ' ' | rev`
- `cat names.csv | sort | cut -f1 -d ',' | cut -f2 -d ' ' | rev | sort --ignore-case`
- `cat names.csv | sort | cut -f1 -d ',' | cut -f2 -d ' ' | rev | sort --ignore-case | uniq -c`
- `sort roster.csv -R | cut -f1 | head -n1`
- `sort < names.csv | cut -f1 -d ','  | uniq -c`

Productivity Tips

- `up-arrow` lets your scroll through your history of commands.
- `ctrl+r` lets you search your history.
- `tab` autocompletes file names
- You can use `vim` or `emacs` to edit files from your terminal

Bash Scripting

All of the previous commands can be run directly on the terminal, however they can also be put into a "shell script." Shell scripts essentially allow you to write many shell commands, such as those in the examples section, and chain them together.

Example This is an example of a script that creates a new directory, creates a new file in that directory, and adds content to the file.

```
// snippets/script0.sh

mkdir newDir
touch newDir/newFile
echo "Here is some content." > newDir/newFile
```

Note that the file may not be "executable" by default meaning you may need be able to run the code. There are two ways around this.

1. Make the file executable by running `chmod +x snippets/script0.sh`
2. Run the file directly in your terminal with `source snippets/script0.sh`. This is equivalent to running every command in your current shell, as if they were copy pasted.

Variables

You can store data in "variables" for later use.

Example

```
myName="Ross Kaplan"
echo $myName
```

You can also store the output of commands in variables

Example

```
mydir=$(pwd)
echo $mydir
```

You can also read files into variables like this:

```
// snippets/script1.sh

myData=$(<names.csv)
echo "$myData"
```

You can concatenate variables as follows:

```
a="Ross"
b="Kaplan"
c=$a$b
echo $c
```

When running a script, you can pass values at run time like this:

```
// snippets/script2.sh

firstname=$1
lastname=$2
echo "Hello, first name: $firstname last name: $lastname"
```

Conclusion

From this lesson, you should be a little more familiar with how to use a terminal and a shell. However it takes a lot of practice to become proficient, so get out there and start working on it.