Introduction to using Linux

Cody Harrington
University of Canterbury Computer Society

May 23, 2013

1 The Linux philosophy

The idea of Linux is simple-To take the idea of UNIX, which is to have a collection of programs and modules that work cohesively together-and make a free, open and community driven operating system that can be edited and customised to your heart's content.

2 A Brief History

I will gloss over a brief history of the origin and etymology of Linux. For more information on this subject, there is much more detail on the internet.

2.1 The GNU coreutils

In 1983, a large hairy man by the name of Richard Stallman was part of a programmer hobbyist group (read: early hackers) at Massachusetts Institute of Technology (MIT). He had become unhappy with this new operating system that had been created at Bell Labs, called Unix, because it was closed source and had become proprietary software. He worked tirelessly to replicate the output of the Bell Labs programmers, but instead as a free, non-proprietary system which resulted in the GNU Coreutils: the main software which makes up Linux (GNU being a recursive acronym for "GNU's Not Unix"). However, the kernel he had written didn't catch on, so for now his project was unsuccessful.

2.2 The kernel

The linux kernel was written by Finnish programmer Linus Torvalds in 1991 as a hobby, which was a free and open-source alternative to a popular operating system at the time called MINIX. He wanted to name it Freax (Freak + Free + x), but eventually settled on Linux (Linus + x) at the advice of a friend. Once he had written the kernel, seeing the GNU project that had been written and released under the GNU General Public Licence (GPL), he took that work and combined it with his kernel, forming the first incarnation of what we know today as Linux.

3 Distributions

The base system of Linux comes in many different distributions which contain different packages and features written by different groups. These are referred to as distros for short, and they have a wide variety of different uses, purposes, system, features, and fanbases. This guide will attempt to be distro-independent, however, a few of the more popular distros are listed:

- Ubuntu

- Debian
- Fedora
- Linux Mint
- Red Hat
- CentOS
- Arch Linux
- Gentoo

A quick internet search can list more and describe the pros and cons of most distributions out there. Protip: There are thousands of them.

3.1 Package manager and repositories

Each distribution comes with a package manager, which handles software installed on the system and has a number of remote repositories from which it gets its software. For example, Ubuntu uses a package manager called aptitude (apt for short). You can type:

```
sudo apt-get install firefox
```

to install the Mozilla Firefox web browser. It will search the remote repositories (listed in /etc/apt/-sources.list) for the required packages and instructions to install them and once found, it will install the software on your computer for you. The package manager can also update and remove software, and manage your local package database. This is one of the brilliant things of a package manager: you can run a single command and you've installed new software–You can run a single command, and update all your packages, etc.

Note: `apt-get` is the package manager for Ubuntu. Fedora uses `yum`, Arch uses `pacman`, and Debian uses `aptitude`. For instance:

```
yum install firefox
```

Will install Firefox on a Fedora machine. You can also install multiple package managers on any one distro, but as they say: *too many cooks spoil the broth.*

4 The X Window System

4.1 Xorg server

Because at its core, Linux is just a text-based system, it has the Xorg server to provide graphical windows. Xorg is a program that runs in the background (called a *daemon*) that simply runs the windows for your display. It is a very simplistic server, with which clients communicate with to display your windows. If you're *especially* hardcore, you can install a Linux system without an Xorg server, and just interact with the terminal.

4.2 Window manager

A window manager is a piece of software that will change your window display and add more functionality, or allow you to customise it. It works as a client which interfaces with the Xorg server, and that's all a window manager really does-it manages windows. There are many different managers out there-some of them which work completely differently to how a window system would conventionally work on other systems.

Some people like to use what is called a 'Tiling Window Manager' (such as Awesome WM), which splits your screen space up into squares (tiles) and allows you to switch between them using they keyboard. They then extend these with scripts that they write, and, using `vim` or `emacs` they completely remove the need to use the mouse.

4.3 Desktop Environment

A desktop environment is a fully-fledged graphical display suite that (usually) adds a window manager, plus a whole lot of other software such as image editors, a music player and stuff which enhances the user experience. Because Linux is so flexible, some people usually just skip the Desktop environment and install a Window manager only because they want a lightweight system. A few examples of Desktop Environments are KDE, Gnome, Unity, XFCE, etc. Note that a Linux distro is not a desktop environment or window manager. In fact, most distros can be made to look quite similar by running the same desktop environments or window managers on them.

5 The Terminal

The terminal, also know as the Command Line, Shell, or just cold, emotionless text on a black background, gives you access to the real power and beating heart of Linux. This is what we will be focusing mostly on in this guide.

Linux by default uses BASH, or Bourne-Again Shell which is based on the shell program 'sh' from UNIX, written by Steven Bourne of Bell Labs. Bash will be all of our terminal interaction. To manage multiple user sessions on a single machine, Linux (and UNIX) uses what is called a TeleTYpeWriter (TTY) for each user to interface with the main kernel. Each tty is handled by its own special device file, located in the directory `/dev`. It also uses Pseudo Terminal Slave (pts) to handle other types of terminal interface, but this is beyond the scope of this guide.

You may be wondering what the difference between shell and TTY is. Shell is the command interpreter that runs everything you type in, and TTY is the connection that handles the data between Linux and the shell.

The shell uses files called stdout, stdin and stderr to handle text input and output to and from programs.

- **stdin** - Standard Input - All your typed text input goes into programs through here.
- **stdout** - Standard Output - All successful program output goes to here.
- **stderr** - Standard Error - All error messages and problem codes go to here.

For now we'll just focus on stdin and stdout.

5.1 Terminal commands

Firstly you should get familiar with the man pages, which are essentially the manual, and will display help pages on almost all commands.

```
Usage: man COMMAND
```

Where COMMAND is replaced with whatever command you want help on. Press 'q' to exit a manual page. Alternatively, most commands will allow you to add `--help` on the end to get their own personal help pages.

Here is a list of the essential commands that you should become familiar with (and you can use the manual to learn how to use them):

```
cd # Change directory
ls # Show contents of directory
echo # Print text to the screen
cat # Display contents of a file
nano # Edit a file via the command line
mv # Move (or rename) a file
cp # Copy a file
rm # Remove (delete) a file
```

These are some extra commands which aren't totally essential but are certainly helpful:

```
mkdir # Make a new directory
rmdir # Delete a directory
grep # Search for specific text within text
pwd # Print working (current) directory
whoami # Display user
ps # Display running processes
pstree # Display a tree showing running processes and processes they started
top # Display most intensive running programs
who # Display logged on users
w # Display logged on users
which # Display the path to a command's binary
df # Disk space free
du # Disk space used
passwd # Change user password (not to be confused with pwd)
more # Display text one screenful at a time
less # Display text one screenful at a time
wc # Word/letter/byte count
id # Display the uid, gid and groups of a user
su # Switch user
tty # Display which tty you are on
```

5.2 Keyboard shortcuts

Every key pressed ends a character to the terminal, and you can send different characters by holding down keys like [Ctrl] or [Alt]. This is how the shell can tell what key is pressed, and thus, allow shortcuts to be defined. Some of the more useful keyboard shortcuts are defined:

- Up arrow or Down arrow
  Scroll through typed commands
- Home or End
  Move to the start or end of a line, respectively
- Tab
  Autocomplete a file name, directory name or command name.
- Ctrl + C
  End a running process
- Ctrl + D
  End an End-Of-File (EOF) character (usually ends a process or signifies the end of input data)
- Ctrl + Z
  Send the currently running process to the background

- Ctrl + L
  Clear the screen, same as running the `clear` command

5.3 Piping and redirection

There are a number of little quirks that the shell has that gives it more functionality. Piping takes the stdout of the left program and connects it (i.e. *pipe*s it) into stdin of the right program with the pipe operator |. For example:

```
# Count number of words in helloworld.txt
cat helloworld.txt | wc -w
```

Redirection directs data in and out of files, i.e.

```
# Redirect stdout to file
echo "Hello world" > helloworld.txt

# Redirect stdout to the end of a file
echo "world." >> hello.txt

# Redirect a file to stdin
more < helloworld.txt
```

5.4 Wildcards

The shell uses a number of special characters called wildcards, similar to *regular expressions* or *regex*, which can be used to manipulate what is being dealt with on the command line. The standard wildcards are thus:

***** Match 0 or more characters. For example, `rm *.txt` will delete all files that end in .txt, and `cp somedirectory/* .` will copy all files from 'somedirectory' to the current directory.

? Match any single character. For example, `cp example.? somedir` will copy all files named 'example' with a single character extension, into the directory 'somedir'

[] Match any single character in the square brackets. You can even specify a range, i.e. `rm m[a-e]m` will delete any files starting and ending with m, and with any letter between 'a' and 'e' in between. `rm m[abc]m` will delete files 'mam', 'mbm', 'mcm'.

{} Match any item in the braces. For example, `cp {*.doc,*.pdf} ~` copies any files with the extension '.doc' or '.pdf' to the home directory.

5.5 Conditional execution

You can chain commands together on one line by separating them with a semicolon ';'.

```
cmd1; cmd2; cmd3
```

However, every program returns a number back to the OS once it has finished running to tell if it completed successfully or not, and we can use this to chain execute commands conditionally.
To run a command if and only if the last command completed successfully, we use `&&`:

```
cody@CODY-STUDIO-XPS:~$ mkdir foo && cd foo && echo "hooray" > somefile
cody@CODY-STUDIO-XPS:~/foo$ cat somefile
hooray
cody@CODY-STUDIO-XPS:~/foo$
```

To run a command if and only if the last command failed, we use `||`:

```
cody@CODY-STUDIO-XPS:~$ ls foo || cd foo || mkdir foo && ls -ld foo
ls: cannot access foo: No such file or directory
bash: cd: foo: No such file or directory
drwxrwxr-x 2 cody cody 4096 May 23 00:57 foo
```

5.6 Processes

Every program that runs, runs in virtual memory as a process, even the shell. You can list the currently running processes with the command `top`. When you run a command, the terminal session runs it on its process, waits for it to complete, then regains control once the command is finished. So, if you were to close the terminal window while a command was running, that would stop the command. Since this can be inconvenient, we can 'fork' the command into its own process to run in the background, and still use the shell while it runs (which is useful for commands that take a long time). To do this, we end the command with a single ampersand `&`. For example:

```
cody@CODY-STUDIO-XPS:~$ (sleep 15; date) & date
[1] 12186
Thu May 23 02:06:14 NZST 2013
cody@CODY-STUDIO-XPS:~$
cody@CODY-STUDIO-XPS:~$ Thu May 23 02:06:29 NZST 2013

[1]+  Done                    ( sleep 15; date )
cody@CODY-STUDIO-XPS:~$
```

`(sleep 15; date)` is sent to the background and returns the process ID (PID), then the next `date` is run and the shell is returned. After sleeping for 15 seconds, the `date` sent to the background outputs and the shell reports that the command completed.

There is a command, `nohup` (no hangup), which prevents a program from being forcefully terminated under normal circumstances. We can combine this with `&` to run programs that need to run uninterrupted for long periods of time.

Another way to list the processes running is with `ps`, and then end them with `pkill` or `killall` (kill by process name), or with `pkill` (kill by process ID), or even with they keyboard shortcut [Ctrl] + [C] as mentioned above.

You can also check what is running in the background and foreground with `bg` and `fg`, respectively.

6 File system structure

The file system is structured as a tree that flows down from the root directory, which is simply represented as /. I've taken a listing of my root directory as an example, using the ls command:

```
cody@CODY-STUDIO-XPS:~$ ls /
bin    etc         initrd.img.old  lost+found  proc  selinux  usr
boot   fixdm       lib             media       root  srv      var
cdrom  home        lib32           mnt         run   sys      vmlinuz
dev    initrd.img  lib64           opt         sbin  tmp      vmlinuz.old
```

The standard path is listed as all the directories to a file, separated by the / character. You can also use .. to represent the folder that the current folder is in, . to represent the current directory, and ~to represent your home directory. For example

```
# Move two folders up, then into dir1 and then dir2, then back into dir1, then back into dir2
cd /../../dir1/dir2/../dir2
# Get a listing of the current directory
ls .
# Change into your home directory
cd ~
```

Linux will automatically complete a command or filename if you are part-way through typing it; all you have to do is hit the [Tab] key. Press [Tab] enough times and it will list possible suggestions based on what you currently have typed in the terminal.

6.1 File operations

There are a number of useful programs that allow us to do file manipulation. To list some of the main operations:

cp Copy a file from one location to the other.

mv Move a file from one location to the other. Note, this is also used to rename files–you just 'move' the file to the directory it is already in but as a new name, for example `mv foo bar` would rename the file from 'foo' to 'bar', assuming you didn't have a directory named 'bar', in which case, the file would be moved to that directory instead.

rm Delete a file. Note that you can also delete empty directories this way, and you can delete a directory and its subdirectories by using `rm -r`. However, be **VERY** careful: if you were to run `rm -rf /`, you would erase every file on your whole computer, because it would delete the root directory and then every file and subdirectory below it and it wouldn't stop because the 'f' in '-rf' means 'force'. Use `rm -rf` with extreme caution, or even use `rmdir`, which removes a directory.

grep `grep` stands for Global Regular Expressions Parser, and can search through text for a match. For example, `grep foo bar` searches the file `bar` for the string `foo`, and you can also use `ls -l | grep "foo"`, which searches the file listing for a file called `foo`. When combined with `sed` and `awk`, you can do almost anything string related.

6.2 $PATH and the environment

Variables in the shell are defined using the `export` command, and when variables are used, they start with a $.

```
cody@CODY-STUDIO-XPS:~$ export FOO="This is a string"
cody@CODY-STUDIO-XPS:~$ echo $FOO
This is a string
```

You can see a list of the set environment variables by typing `set` by itself into the terminal.

Linux uses a global terminal variable to find programs. This is the $PATH variable and it consists of a list of file paths to search in for a specified program, in order, separated by the colon (:). For example, a listing of my path:

```
cody@CODY-STUDIO-XPS:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

This would mean that if I were to use the program 'ls', it would search for a binary file called 'ls' in /usr/local/sbin, then in /usr/local/bin, and so on until it found it. Then it would be executed. Because these are searched in order, if you were to prepend a directory path to the front of $PATH with your own copy of `ls` in it, and then run the `ls` command, then your copy would be run instead.

This is sometimes used as an exploit by modifying the user's $PATH variable so that a path containing malicious binaries with the same names as common commands is on the front. When the user goes to run these commands, then the malicious binaries are run instead.

7 Users and permissions

Every user has a user ID (uid) and a group ID (gid). Each user also has a list of groups they are a part of which give them the permissions that are assigned to those groups. You can see this by using the 'id' command:

```
cody@CODY-STUDIO-XPS:~$ id
uid=1000(cody) gid=1000(cody) groups=1000(cody),4(adm),24(cdrom),27(sudo),29(audio),
30(dip),44(video),46(plugdev),109(lpadmin),119(pulse),124(sambashare)
```

7.1 sudo and root

Now for the most powerful user on Linux: The root user. Root's uid and gid are both 0.

```
cody@CODY-STUDIO-XPS:~$ id root
uid=0(root) gid=0(root) groups=0(root)
```

Root is essentially god: root can do anything root likes, where other users would be denied due to the lack of permissions required. Root is the first account created on a newly installed Linux distro, and it is generally encouraged that you do not use the root account unless you absolutely have to because since root can do anything, then there's no stopping you from accidentally deleting something important, for example.

This is where the `sudo` command comes in (a.k.a "super-user do"). You can use `sudo` to execute commands that require elevated privileges without having to actually switch to root.

Say you want to edit the hostname file, which contains the name of your computer, but, by default, you need elevated privileges to edit it. You would type:

```
sudo nano /etc/hostname
```

to which it asks you for your password, and then opens nano with the extra privileges provided by sudo. There is a sudoers file which contains a list of users who can use sudo, and what privileges they get from using it.

7.2 File permissions

Linux inherits its file permissions system from Unix. You can use the command `ls -l` to display the permissions of a file or files:

```
cody@CODY-STUDIO-XPS:~/junk$ ls -l
total 8
-rw-rw-r-- 1 cody cody    9 Apr 25 21:28 junk1
drwxrwxr-x 2 cody cody 4096 Apr 25 21:29 other_junk
```

The first string consists of a sequence of letters, which represent the permissions on the file. The two names refer to the owner and group the file belongs to, respectively.

Lets take the file, `junk1`, as our example. The first character is the file type. This is a 'd' if the file is a directory (like `other_junk`). The next part should be read as three sets of permissions

```
-rw-rw-r--
(d)( u )( g )( o )
(-)(rw-)(rw-)(r--)
```

where the first set, u, refers to the permissions for the user who owns the file. The next set, g, refers to the permissions for the group that the file belongs to. The final set, o, refers to the permissions for any other user. Each set uses 'rwx' to specify the permission to (r)ead, (w)rite or e(x)ecute, or - if that permission is not set. Lets take a look at the folder `other_junk`.

```
drwxrwxr-x
```

This is a directory, the user has read/write/execute access, users belonging to the group of the file have read/write/execute access, and everyone else has read/execute access, but not write access.

7.3 Changing permissions

If you want to change a file's permissions, you can use `chmod`, meaning "change mode". There are two ways to do this: using u/g/o and +/- r/w/x:

```
chmod o+x junk1 # Add execute permission to others
chmod og+wx junk1 # Add execute and write permissions to other and group
chmod +x junk1 # Make junk1 executable for the user
chmod g-w junk1 # Remove write permissions from junk1 for group
etc...
```

or with a number that represents permissions, called a bitmask. This will set all the permissions at once for you.

```
cody@CODY-STUDIO-XPS:~/junk$ chmod 755 junk1
cody@CODY-STUDIO-XPS:~/junk$ ls -l
total 8
-rwxr-xr-x 1 cody cody    9 Apr 25 21:28 junk1
drwxrwxr-x 2 cody cody 4096 Apr 25 21:29 other_junk
```

The bit mask is three digits (sometimes four digits) between the numbers 0 and 7. Each digit represents what the read/write/execute permissions would be in binary. Take a look:

```
0 = 000 = ---
1 = 001 = --x
2 = 010 = -w-
3 = 011 = -wx
4 = 100 = r--
5 = 101 = r-x
6 = 110 = rw-
7 = 111 = rwx
```

So, as a few examples,

```
777 = rwxrwxrwx
755 = rwxr-xr-x
132 = --x-wx-w-
564 = r-xrw-r--
000 = ---------
etc...
```

So when we set our file junk1 to 755 earlier, we set it to rwxr-xr-x, which is a pretty good permission set on your average file. Realistically, you will usually always have your own user permissions set to rwx or rw-, otherwise you are just inconveniencing yourself. You can also use `chown` to change ownership of a file.

8 Secure Shell (SSH)

SSH gives you a secure, encrypted connection to a terminal session on a remote machine. It works exactly like a terminal session would on a local machine.

```
ssh username@address
ssh address # The username of the machine you are connecting from will be used
```

Where `username` is your username on the remote machine and `address` is the address of the machine to connect to.

You can also use what is called 'X forwarding' to run GUI applications on the remote machine, but have them display on the machine you are connecting from; you just use the -X or -Y switch.

```
ssh -X username@address
ssh -Y username@address # Trusted X Forwarding
```

8.1 Secure Copy

Secure copy is like `cp` but uses SSH to transfer files from one machine to another.

```
# Copy from remote machine to local machine
scp username@address:pathtosrcfile destonlocalmachine
# Copy from local machine to remote machine
scp fileonlocalmachine username@address:pathtodest
```

Where `pathtosrcfile` and `pathtodest` are paths on the remote machine.

8.2 SSH keys

Public Key Cryptography[1] is used to protect what information is sent over SSH. There are two keys: a public key (to encrypt data) and a corresponding private key (to decrypt the encrypted data). Think of the public key as a padlock, and the private key as the key to open the padlock.

With SSH, you store your public key on the computer you're connecting to, and keep your private key on the computer you want to connect from. When an SSH connection is initiated, the RSA[2] algorithm is used to check that the private key is correct for the public key, and if all is well you will be authenticated and given access.

All files pertaining to SSH can be found at `~/.ssh`. To generate a new SSH key, you run the command `ssh-keygen`:

```
cody@CODY-STUDIO-XPS:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/cody/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/cody/.ssh/id_rsa.
Your public key has been saved in /home/cody/.ssh/id_rsa.pub.
The key fingerprint is:
8c:09:0a:43:d5:b3:1e:28:e3:8d:d4:8c:84:4f:72:b0 cody@CODY-STUDIO-XPS
The key's randomart image is:
+--[ RSA 2048]----+
|oo...            |
|+oo  o           |
```

[1] http://en.wikipedia.org/wiki/Public-key_cryptography
[2] http://en.wikipedia.org/wiki/RSA_(algorithm)

```
|E=+ o o          |
| B.= + +         |
|o * . + S        |
| o . .           |
|                 |
|                 |
|                 |
+-----------------+
cody@CODY-STUDIO-XPS:~$ cd ~/.ssh
cody@CODY-STUDIO-XPS:~/.ssh$ ls
id_rsa  id_rsa.pub
```

Note: This is obviously not output from generating my real private/public keys. This data is essentially useless to me

If you decide to set a password, you will be prompted to enter that password to unlock your private key any time you want to connect via SSH.

8.3 SSH tunnelling

You can use what is known as SSH tunnelling, which is where you make all traffic pass through your SSH connection to protect your privacy and security, however that is beyond the scope of this guide.

9 Further information

- The Rute Users' Tutorial and Exposition (recursive acronym RUTE, like GNU).
  `http://linux.2038bug.com/rute/meaning.html`
- Local Linux user group wiki
  `http://wiki.linux.net.nz/CategoryBeginners`
- A good book which inspired and helped me write to write this document was "The Unix Programming Environment, 2nd edition" by Brian W. Kernighan and Rob Pike. I highly recommend it as it covers Unix systems far more in depth than this guide, plus it includes shell scripting, C programming, language parsing, and document typesetting.
- And as always, there is Google.

This document was typeset and compiled in LaTeX.
`http://www.latex-project.org/`
A tutorial on how to use it:
`http://en.wikibooks.org/wiki/LaTeX`
The editor that I use (available from most package managers):
`http://www.xm1math.net/texmaker/`