The GNU Linux Operating System

Sections:

Introduction

1.1: What is GNU?

In 1991, Linus Torvalds released his greatest feat to the world. This was none other than the Linux operating system. He designed it entirely himself as a successor to the more primitive Unix operating system, which was written by Dennis Ritchie (the developer of C) and Ken Thompson (it should be noted that Unix was entirely written in C, and therefore, both Unix and Linux share a lot of commonality with C). In 1983, Richard Stallman founded the GNU project which was an organization created with the intent of giving computer users freedom and control over their computers. Both the Linux operating system and the GNU project became heavily intertwined, whereas Apple and Microsoft went about their own business. GNU stands for "GNU Not Unix" (yes, the acronym is part of the acronym?). Linus Torvalds allowed the free distribution of the Linux Kernel for anyone to modify and distribute. This led to two major advancements in the Linux community. First, you had/have larger companies and very smart individuals modifying and adding to the kernel to create what are called Linux distributions (distros for short) which are varying versions of the central operating system. Second came the rise of major GNU packages such as the GCC (GNU Compiler Collection), a collection of compilers for various languages built into the kernel, as well as things like the X Window System which almost all distros use for their GUI. The GNU also owns almost all of the basic commands (called utils or utilities in Linux) which are mostly rewritten functions from Unix (as well as some new additions).

Figure 1 GNU Logo

1.2: Distributions

There are hundreds of distros you can pick from to date but only a handful are actually used by the majority of users. If it wasn't complex enough, there are larger distros which have been slightly modified to create families of distros. For example, there is Red Hat Linux which branched into distros such as Cent OS, Fedora, openSUSE, etc. A lot of those child distros also have their own families. The most common distros you'll hear of are some of the following: Debian, Ubuntu (Debian based), Red Hat, Fedora, Linux Mint, Arch Linux, Gentoo, Void, Manjaro, openSUSE, Cent OS, and perhaps a few others. You won't often hear many distros beyond these. It should be noted that a "minimalist" distribution refers to a distribution which has very little in terms of applications by default. Arch, Void, and Gentoo are examples of minimalist distributions. These often take more knowledge of Linux to setup but allow for much more customization/personalization. The most well-known beginner distro is Ubuntu and it is the one I would recommend starting with, since this guide is heavily based on it (though I do highly encourage you try others as well).

1.3: The Basic Components of a Linux Distribution

It is important to understand the main components of any functional distribution. This includes the boot manager to boot the OS, the Kernel which is the central code of the OS, applications (known as daemons in Linux), which are the primary applications required to do important tasks (ex. The package manager which installs, removes, queries packages, etc. or your network manager which manages your network configuration), utilities which are the primary commands included with the distro, at least one shell which is like a computer language with a syntax and commands unique from the utilities, and finally a terminal emulator which is the command line interpreter that interprets and executes commands. Because of the nature of the open source kernel, each update is heavily monitored by contributing users and therefore, drivers are not usually required, and bugs are very rare.

1.4: Installation

A Linux install will naturally vary depending on the distro, although there are a few common pointers which I can provide. As with any operating system, the OS will need to be placed onto a portable form of media such as a flash drive or cd. The distro should have a downloadable iso file which can be downloaded for free through torrenting or off a mirror server (no, this is not at all illegal, and is, in fact, the only way to do this). Typically, the kernel is installed on some bootable thumb drive or CD. For a dual install (having two operating systems on your computer), you can shrink the volume of one OS and then use the remainder of the disk for the distro. Alternatively, to install a distro on bare metal you only need to have a properly partitioned disk. Generally, I recommend the use of one of the many applications available for installing Linux. You will need to enter your BIOS or UEFI (some distros require that you know the difference), and then select the media which the distro was installed on as the primary boot device. If the boot loader GRUB is installed (which is almost always the case), then it will detect both OSs. There are many helpful resources online which will be more specific if further help is required.

The Environment

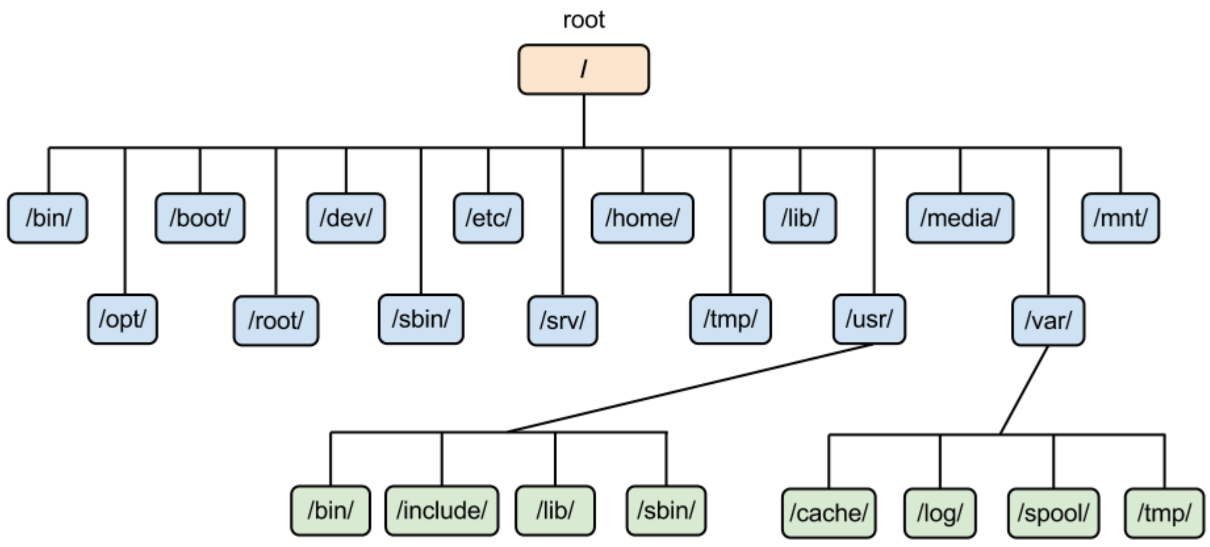
2.1: A Fresh Install

Assuming a Linux distro is running on your pc there are quite a few things to understand. Each distro will have a daemon known as the file manager which usually has an icon of a file cabinet or folder. This holds all your files and directories. In Ubuntu, the default file manager is called Nautilus. Another important thing to understand is the difference between a desktop environment and a window manager. Ubuntu uses a desktop environment called GNOME (currently on version 3). This is responsible for all your GUI elements. Whereas most people are used to desktop environments with icons that execute applications various other graphical elements, the alternative is to use a window manager. A window manager is still a graphical environment, but everything is run via terminals (called windows). While this may sound silly on paper, it is my preferred choice and is often much more efficient for a Linux work environment. It should be noted that desktop environments and window managers rarely get along. You should only choose one option so do your research because a distro like Ubuntu will automatically install GNOME, and you will not be able to easily change it. Anyways, the GNOME desktop environment, like most distros, uses the X Window manager (also called X11 or just X). We will discuss X later, in the services section. The default Terminal emulator for Ubuntu is the GNOME *terminal* and its default shell is bash. Aside from all this, there is not much more to a Ubuntu install (at least on face value). You are most likely eager to begin installing applications, I'm sure. Before we can do that though, packages and package managers must be understood. A package in Linux is the same as a package in any code base. It is a collection of files which are bundled to make a deliverable (an application). There are distro specific packages, though they are often cross compatible (especially if they originate from a parent distro). For example, a .deb package is a package made for Debian based distros. The Debian package manager is called dpkg. You may also see rpm (red hat package manager). Ubuntu's package manager is called apt. There are many ways of installing packages in Ubuntu, but the most common method is to open the terminal with Ctrl+Alt+t and type sudo apt install <package_name>. sudo is a command which will be explained later, though keep it in mind since it is very important. The apt command is actually short for apt-get (apt is very general so you may see apt-get which is more specific). I think the rest is self-explanatory. Other methods of installing software are to use the Ubuntu Store or to use a command called wget/curl (though curl is sort of legacy) which will install a package from a website. Usually you can guess the name of the package you're trying to download such as sudo apt install firefox, but if your not sure, you can either look it up online or type sudo apt search <similar_package_name> to find similar results. Often packages rely on other packages or programs which are called dependencies. Usually, these dependencies are downloaded for you, but sometimes, you will need to download them manually before you are able to download the intended package.

2.2: The File Hierarchy

The file hierarchy in Linux is far less complicated than in Windows despite not feeling like so at first. Essentially, you have the root directory which is like the directory at the very top of the pyramid.

Every file and folder flows from root. It is represented as the symbol '/'. The root folder has a few child directories, all of which are quite important and useful. In this section, we will go over each one.



/bin: This directory contains the important binary files required for the system to run. Binaries are executable files (typically utils), meaning that most of the most basic commands in Linux are stored here. Your installed shells are also located here, such as bash, sh, and the C shell.

/boot: This directory contains files pertinent to the boot process. The two most notable files are the grub.conf file which is the configuration file for the popular GRUB boot loader, and vmlinuz image file (also known as the linux kernel)!

/dev: This directory contains device information for things like hard drives and other SCSI devices.

/etc: Probably the most important directory to come from these next to /home is the etc folder which contains a lot of important files like the various *tab files (fstab, crontab, inittab, mtab, etc.), as well as the X11 files, dhcp, skel (I mean really, 90% of these files are important).

/home: This is where are the user home folders are located.

/lib: The library directory contains all the libraries required for application use.

/media: This is the directory used to mount removable media such as floppies, flash drives, and CD-ROMs.

/mnt: This is the directory for mounting permanent storage devices like your hard drive or solid-state drive.

/opt: This directory stands for optional. Optional software packages install and copy files from here.

/root: This is the home folder for the root user also known as the super user. This directory will make more sense later on.

/sbin: This directory is like /bin but for administrative binaries such as mount, umount, shutdown, etc.

/srv: This is the service folder which contains data for the system services like https or ftp, offered by the system.

/tmp: A temporary scratch directory with no special purpose. It is cleared each time the system boots.

/usr: This directory contains subdirectories used by many programs such as X. It is also common to store variables and files which will be accessible by every user in /usr/local.

/var: Contains various system files such as log, mail directories, print spool, etc.

2.3: Navigating Directories

Now we'll begin to use commands for navigating the file tree. This may sound mundane, but it is actually very useful and common, so it is important to understand. Linux was built under the assumption that users did not have access to a mouse since users often worked off a real terminal without any mouse. As you become more comfortable using Linux, I recommend straying away from using a mouse wherever possible since it saves a lot of time and allows you to become accustomed to the keyboard. The first thing to understand is that we can either specify what we call the absolute path or the relative path. The absolute path begins at root '/' and travels through every directory to get to the desired location. An example of an absolute path would be /home/neil/Downloads. Assuming the user is currently in their home directory (the default when opening the terminal), the relative path would be Downloads/ The slash at the end is not required, and only specifies that you are referring to a directory.

This is important when you have both a folder and file with the same name in the current directory for differentiation purposes. Speaking of the current directory, to check where you are in the tree, you can type pwd which stands for present working directory. This will display the absolute path to your current location. We move locations using the command cd which means change directory. So, if I would like to enter the Downloads folder, I could type cd Downloads or cd /home/neil/Downloads. You may notice that something changes at this point. To the left of your cursor, there are a few things which I should quickly explain. Typically, the current users user name will be displayed, followed by @ and the system's name, then the current directory and finally either $ or #, which simply represent the type of user logged in. $ will represent a regular user, and # will represent the super user. Another useful command is ls which will list everything in the pwd. Linux, much like Windows and OS X, uses hidden files which will not be shown using ls by itself. Instead, we must use what we call options. An option is an extra feature provided by a command. They are usually represented by a single letter, though there are options which can be full length words or sentences. To specify a single letter option the '-' symbol is used. Any proceeding single letter option can simply be written directly after the first. For example, ls -l -a can be written as ls -la. For an option that is represented as a string, we use '- -' (no space) instead. These types of options (--) are always specific to the GNU project, and therefore, are guaranteed not to works on Unix machines. Some generic options follow this trend as well (though double dashes are always Linux exclusive). One of the most useful commands is the man command which will display the manual page for commands or even important files. If we type man ls we can see a description of what the command does as well as its options and input order. For ls we can now see that [-l] formats the output into a long list, and [-a] displays all items, including hidden ones. Options vary for every command, though they often try to make them easy to remember (for example -a usually means all). So, we've discussed cd, ls and pwd. Let's now discuss the dot operator. The dot operator is represented by a period '.' and is the same thing as the pwd. Let's say that I wanted to move a file to my current directory. I could use the mv command which means move, followed by the dot operator to specify the pwd. This would look something like: mv Downloads/file . which would move the file named file in Downloads to the pwd. Now you may be wondering, can't we just type mv Downloads/file pwd instead of '.' if they are the same thing? Well no, and this is because of how bash works. There isn't actually a (re)name file command in bash. The mv command simply copies a file to another location and then deletes the original. What this means is that if we move a file, we can change its name by specifying a location that doesn't exist. For example, if we enter mv Downloads/file pwd, this would actually move the file 'file' to whatever directory we are in and give it the name pwd. I know that's kind of confusing, but just understand that bash only reads the first word entered as an actual command. Everything after that will either be an option, or a file. In other words, bash cannot read two commands in one line. It expects <command>[options][directory] in that order. Make sure you play with it yourself so that you understand. Anyways, going back to the dot operator, one . means pwd, but two .. means parent directory. This is useful for traveling up the file tree because we can type cd .. to go up one directory. Another strange thing about Linux, is that there is no command for running scripts. Yet, bash expects a command, and so we use the dot operator to tell bash that we'd like to run a script in our directory. This is done by typing ./<scriptname> In this case, the / indicates that . is referring to a directory, otherwise if we left it out, bash would only see .scriptname and search for a hidden file. If scriptname *was* a hidden file (which often times scripts are), we would instead type ./.scriptname The last two things I'll touch on for navigation is the tilde ~ The ~ refers to the users home directory. For this reason, you'll often see people type ~/filename which is just an abbreviation of the absolute path for

/home/user/filename. Believe it or not, these few commands we've learned will act as a crutch for most of the rest of the commands we'll learn.

Basic Commands

Since it is too difficult to explain every single command in detail, I will provide a brief summary of some commands which you will need to become accustomed to. A lot of them make sense, but some of them will be a bit harder to remember. Really take the time to practice these commands before we delve into the intermediate stages of Linux.

Information

help: This command is part of every Linux distro. It will provide information on basic utilities.

man: Stands for manual page. This command will come packaged with most distros but not all. It is the replacement for help and should be used often. The author must have written a manual page for their command which means it does not work 100% of the time, but it is very reliable.

whoami: Displays the username of the user currently logged in.

uname [-a]: uname provides information about the operating system. Specifically the -a option which prints all information including kernel version and install date.

pwd: Prints present working directory to console

File/Directory Manipulation

cd: Change directory

mkdir: Make a new directory

rm/rmdir: Remove a file or directory respectively. To remove a directory which is not empty, use rm -r (not rmdir -r since -r isn't an option for rmdir) to recursively remove all children files and the directory itself.

mv: Move a file or directory. Has the additional feature of changing file/dir name.

cp: Copy a file or directory to another location.

touch: This command updates the timestamp of a file but is most often used to create a new file. There is no command for creating a file in Linux, yet there are various ways of doing it. touch is the most common way of doing this since in order to update the timestamp of a file which doesn't exist, it must be created first.

Reading

grep: grep is an advanced tool which searches for patterns in a file. If no pattern is specified, the entire file is output.

cat: This command, while not its original intent, will display the contents of a specified file.

ls: lists contents of a directory.

tree: A very useful command which acts similar to ls but displays subdirectories as well. Must be installed before it can be used.

Writing

echo: Outputs a specified string.

> and >>: Redirect operators will redirect input to an output. > will write over everything, while >> will append input. In other words, If I have a file which contains the sentence "Hello my name is " and I type echo "Neil" > file, the file will now only contain "Neil", but if I use append echo "Neil >> file, then the output will be "Hello my name is Neil"

cat: As mentioned before, cat's original purpose was to concatenate input to an output similar to the >> operator. For example, you can concatenate multiple files into one. cat file1 file2 > file3

Process Management

Ctrl + c: Force quit a process.

Ctrl + z: Stop a process but don't terminate it.

Ctrl + d: Generates the end of file (EOF). This must be used after running a C script.

exit: Close a terminal.

q: While not an actual command, q typically quits out of a program, back to the terminal screen.

Other

tab: While also not a command, tab provides autocompletion in most shells and is extremely useful.

Ctrl + Shift + Plus: Enlarge text within shell (does not scale with window size).

Ctrl + Minus: Shrink text within shell.

;: Known as the command separator. A semicolon allows you to enter multiple commands on one line. The command which proceeds the command separator is ran whether or not the first command was successful. Essentially, it is as if the user wrote each line individually, one after the other.

|: The pipe command essentially concatenates two commands. For instance, if I wanted to grep the output of a command, I could do: ls | grep "Downloads" which would then search for the pattern downloads within the output from ls. This is often used with the more command to have only one page appear at a time if a mouse is not being used. Ex. man touch | more

shutdown: This Command required super user privileges. The -h option is generally used to poweroff the machine. By default, it will schedule shutdown for a minute prior to the command being entered. This can be changed by specifying now to shutdown immediately, or some other number to schedule it at a later time. Ex. sudo shutdown -h now

reboot: Reboots computer. shutdown is also capable of rebooting, but most people prefer reboot.

clear: Clears everything on screen.

history: Displays the last 1000 commands typed. This value can be changed in the .bashrc file

Permissions

4.1: The Purpose of Permissions

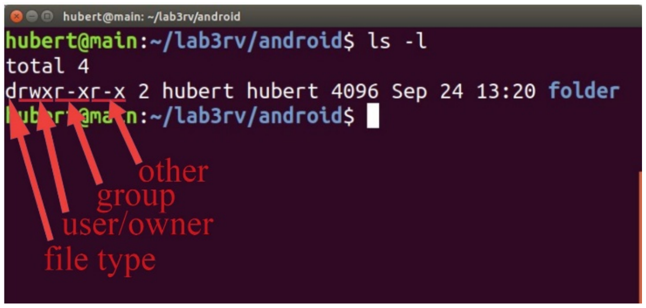
Linux is well known for its security and privacy. Apple's OS X operating system was originally a deviation of the Linux kernel similar to how Linux distributions are formed. Apple changed its OS so much that it is no longer considered a Linux distro, however, many remnants still exist such as its terminal feature. The reason I bring up OS X is because it is often praised for its lack of malware, while on the other hand, Windows is heavily criticized for the amount of viruses and malware that it allows. But really, Apple owes this to Linux because their heavy security makes malware nearly impossible. While Apple and Windows have permissions within their operating systems, they are not nearly as complex as Linux. Linux is very lenient with what the user is capable of doing. I mean for crying out loud, you can cd into the root directory and type sudo rm -r * to wipe the entire computer, and Linux will not give you any warnings. This is why there is such a heavy emphasis on permissions. When you have multiple users working from an SSH server, it is important for the administrator to restrict their employees in what files they can touch or which directories they can view.

4.2: Understanding Permissions

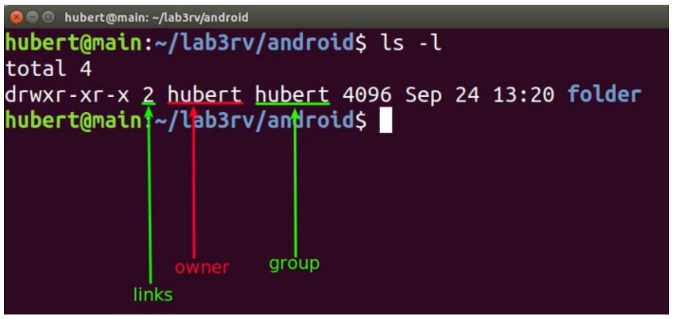
There are 3 kinds of permissions which every directory and file abide by. These are the read (r) write (w) and execute (x) permissions. These permissions mean different things for files and folders, and thus it becomes a bit difficult to remember how to read and understand these permissions. There are also three types of users that Linux groups its users into. First, there is the user group (u) which refers to the owner of the file (the one who created it). This user will generally have full permissions for the file. The next group is the group group (g) which refers to any user who is part of the group that owns the file or directory. By default the group owner will be the primary group of the file owner (something that will be explained in more detail later). Finally, is the others group (o) which refers to anyone who is not part of the first two groups. A fourth group, the all group(a), exists as an option for certain commands but does not exist in actuality. Keeping this is mind, each type of user has their own set of permissions for each existing file or directory. There are two ways in which we can represent permissions.

Visual

The simplest representation of permissions is the visual representation. This is displayed by doing an ls with the -l option to list files in a long list format. You will see to the left of each file will be a bunch of information that looks something like drwxr-xr-x 2 neil neil 4096 Sep 24 13:20. As was mentioned, rwx stands for read, write, execute permissions (sometimes referred to as bits). Each set of 3 bits (an octet) represents the user groups ugo, in that order. In this case, the user group has full permissions, and both the group group, as well as the others group only have read and execute permissions. The d in front of these permissions will vary in character depending on what type of file it is. Everything on a computer is a file, including directories; and so, in this case, the d means that this is a file of type directory (in other words, we're viewing a directory). After the permissions, the user who owns the file is displayed, as well as the group owner (which by default uses the same name as the user, hence neil neil in the example). After that is the link size in MB which we'll get into later, then the date



```
hubert@main:~/lab3rv/android$ ls -l
total 4
drwxr-xr-x 2 hubert hubert 4096 Sep 24 13:20 folder
hubert@main:~/lab3rv/android$
```



```
hubert@main:~/lab3rv/android$ ls -l
total 4
drwxr-xr-x 2 hubert hubert 4096 Sep 24 13:20 folder
hubert@main:~/lab3rv/android$
```

of creation, and finally the file name. You may be wondering then... What do these permissions actually mean. Well, as I mentioned, permissions mean different things for files than they do for directories.

Directories:

r: The read bit allows the user to view the contents within the folder ie. what is stored in any files or subdirectories.

w: This allows the user to delete or edit the directory or any of its files. This permission is essentially useless without the execute bit enabled simultaneously since the only thing you can do without it is modify empty directories.

x: The execute permission allows the user to cd into the directory. Without this permission, the user cannot modify anything within the directory since you first need to enter a directory to modify them.

Files:

r: In the case of files, the read bit also allows users to view the contents of the file.

w: Write allows the user to modify the contents of the file (no permissions are actually required to delete a file). In the case of directories, in order to delete a file the user requires both w and x to first enter the directory, and then delete the file (since the file is within the directory).

x: Execute is quite special for files since they typically don't need this bit set. Scripts cannot be run by the user without this bit being enabled. Typically, files with this bit set are colour coated to notify the user that it is executable.

<u>Octal</u>

The other method of understanding file permissions in Linux is through the octal format. While more difficult to wrap your head around, it is much easier to edit permissions in octal than it is visually. I have been referring to rwx as 'bits' and this is on purpose. While these characters are read as such to the user, the operating system reads them as octal values with r representing 11 or 3, w representing 10 or 2, and x representing 01 or 1. These are added together to form one number that describes all three permissions for each group of users. In visual, permissions might look something like rwxr-xr-x, but in octal, this can be represented as 744 since 3+2+1=7, 3+1=4, 3+1=4. Things get messy when we look at files however, because files don't actually have an execute bit (not in octal at least). It's simple enough when written out however, since it is the same as directories, only you don't write the x bit. In other words, the maximum permissions for a file is 666, and for a directory, is 777. Here's a chart for comparison.

Octal Values in Decimal	Directories	Files
0	---	---
1	--x	---
2	-w-	-w-
3	-wx	-w-
4	r--	r--
5	r-x	r--
6	rw-	rw-
7	rwx	rw-

<u>4.3: Default Values</u>

When a file or directory is created in Linux, its default permissions will be 666 or 777 respectively. However, this is not typically what we want since in general, we want to at least limit the write permissions of the group group and the others group. This is why Linux has what's called the umask. The umask is like a filter. It is always written as an octal value. Whichever corresponding permissions are set will be turned off by umask. In other words, the umask is like a kill switch which will disable permissions upon file creation. On every system, the umask is set to 022 by default. This means that the user group will be unaffected, but the write bit will be disabled on the group group and others group. Or as another example, a umask of 744 would mean that the permissions would be ----wx-wx or 033. You can see what the current umask is by typing umask. You can set it to another value by typing umask followed by the value you'd like to assign it.

<u>4.4: Setting permissions</u>

All this talk and I havn't yet explained how we actually set permissions for users. Well since there are two ways of representing permissions (visual and octal), we can also set them using two different methods. Typically, superuser privileges are required to set permissions. For visual, we specify which users we are modifying (ugoa) followed by + if we are adding permissions, – if we are removing

them, or = if we are setting them, followed by the bits we are adding/removing/setting. The command we use to actually modify the permissions is called chmod which stands for change mode. This might look something like the following: sudo chmod go=rx,u-w <directory/filename> This will set the group group and others group to r-w and remove the write bit from the user group. Assuming the umask was 022 upon the creation of the file or directory, and that permissions havn't been modified at an earlier time, the permissions for this file should now be r-xr-xr-x. To do this in octal, it is even simpler. We just type the octal values in decimal, followed by the filename. Ex. sudo chmod 555 <directory/filename> which will yield the same results as before. Additionally, we can change the owner of a file by using the chown command (change owner) which would look something like:
sudo chown <username>:[groupname] <filename>

User Management

5.1: User/Group Info

Similar to how Linux lead the way for security with its advanced file permissions, it was also the first to allow for multiple users on the same computer. This is called a multiuser environment. Users are assigned groups for easier management. In this section, I will finally be discussing the super user and sudo command in more detail, as well as how we manage users and groups. As I've mentioned, we can switch users using the su command. This command on its own defaults to logging in as the super user, or root user. This is also the user whose home directory resides in /root. The super user is very important and hard to fully explain. They do not abide by permissions and cannot be removed. The command sudo is one which allows a user to gain super user privileges granted that the user is part of the sudo group (in legacy systems, the sudo group might instead be called wheel). Settings for what a user is capable of doing if they are part of the sudo group are set within the /etc/sudoers file. Ironically, you will need to type sudo su to become the root user since you need sudo privileges to login as the super user. Once logged in, the $ in your prompt becomes #. You should never stay logged in as the super user. When a major system error occurs within a user's environment, the OS will switch its state to login as the super user for troubleshooting. Anyways, with that out of the way, let's get into user management.

5.2: shadow and gshadow

Most of the files in this section will be located within /etc since most of the systems config files are here. Typically, viewing these config files are how we find information about the system. For example, devices and users both have IDs which the OS uses. There are many kinds of IDs such as user id (UID), group id (GID), universally unique identifier (UUID) for devices, and process ids (PID). Most of these are located in files, and so if we ever need to refer to these, we will need to view the file they are located in. Of course, we can find more information beyond just IDs within these files. The first files we'll be looking at are /etc/shadow and /etc/gshadow. These are files which contain password information for users and groups respectively. These used to contain the actual raw passwords as plain text before they decided that maybe it would be better to encrypt them via a secure hashing algorithm (SHA). The hash is now stored in a database so that attackers can't undo the hash. Getting off track though, the shadow and gshadow files not only contain the encrypted password itself, but also other information about users and groups such as account expiry date, last password change, etc. Each field is separated by a colon. The tail and head commands are useful when you cat /etc/shadow since user information is always kept at the bottom of the file (head and tail display the first or last 10 lines of a file). You will also notice that Linux stores its services as users. This is because some processes must be run from a user ID, and others must be able to run on their own (ie. services). Here is a list of each field in detail, in the order which they appear.

1. username: The name of the user or process

2. encrypted password: This field contains the hashed password of the user. A ‘!’ means that the password hasn’t been set, ‘!!’ means that the account is locked or disabled, and ‘*’ means that passwords are not allowed to be set, which is the case for processes.

3. last password change: This field is confusing since it’s a number that does not resemble a date. This is because the number is actually the number of days since Jan 1, 1970, also known as the epoch date. To calculate this number to a date we can use the date command with the -d option by entering date -d “1970-01-01 <number> days” which will give us our last password change.

4. minimum number of days before password is allowed to be reset: Specifies whether or not the user has to wait before being able to reset password a second time.

5. maximum number of days a password is valid: The password will expire after this amount of days has passed.

6. number of days until a warning is issued about password expiration: A warning will be sent at this time to remind the user that their password will be expiring soon.

7. number of days until an account is disabled after a password expires: If the user does not reset their password within this time frame after it has expired, the users account will be disabled.

8. Amount of days a user account has been expired for.

For /etc/gshadow, the fields are a bit different:

1. group name: The name of the group

2. password: The encrypted password for the group (this isn’t usually set).

3. group administrators: Essentially the users who have authority over the group. These users can add and remove users from the group as they please.

4. group list: A list of every member who’s a part of the group

5.3: passwd and group

Whereas shadow and gshadow were mainly files for password information, /etc/passwd and /etc/group are the files for user and group information (yes, I know the file names make no sense). These files are a bit more interesting since they contain more relevant information. Before we look at these, we should probably understand some things first. For instance, every user has one and only one primary group. By default, the primary group for a user will be a group that the operating system will create with the same name as the user. This was demonstrated earlier in the permissions section where using ls -l displayed a file that was owned by the user neil (me) and had a group owner of neil (which is my primary group). Similarly, the root user has a primary group called root. As I mentioned, every user and group has an ID. The UID and GID of a user can be displayed by typing the command id. This will also display the groups that the user is a part of and their respective GIDs. For a cleaner output, you can use the command groups which will only output the *groups* the user is a part of (without the GIDs). It should be noted that the UID and GID will be the same for users by default. With that out of the way, let's tail passwd. Again, at the bottom, we can see the users. The fields go as follows:

1. username: The username of the user
2. password field: An x indicates that the user has a password located in /etc/shadow
3. UID: The user's UID
4. GID: The user's primary group's GID
5. comments: An optional comment which may be left by the user
6. home directory: The location of the user's home directory
7. shell: The shell which the user will use by default during their login session

The group file is very similar to gshadow, only the admin field is replaced by the GID.

1. group name
2. group password: An x indicates the group password is located in /etc/gshadow
3. GID: The groups GID
4. group members

5.4: Adding and Removing Users

Adding a user is done using the useradd command which is a somewhat complex command since you actually need to consider each of its options carefully. It is recommended that you use the man page if you can't remember all of the options off the top of your head. Here is a summary:

-d: Sets the location of the users home directory (if the location doesn't already exist you will need to use the -m option as well).

-g: Sets the primary group of the user (this must exist). If it is left blank, a new primary group is created with the same name as the user.

-G: Sets any groups the user would like to be a part of. Each group should be separated via comma.

-c: Add a comment which will be displayed in /etc/shadow

-e: Sets the expiration date of the user's account

-s: The default login shell of the user. The shell must exist in /bin

-m: Creates the home directory specified with the -d option

The user's username should be placed at the very end. This would typically look something like:
sudo useradd -d /home/username -m -e 2030-04-28 -s /bin/bash username

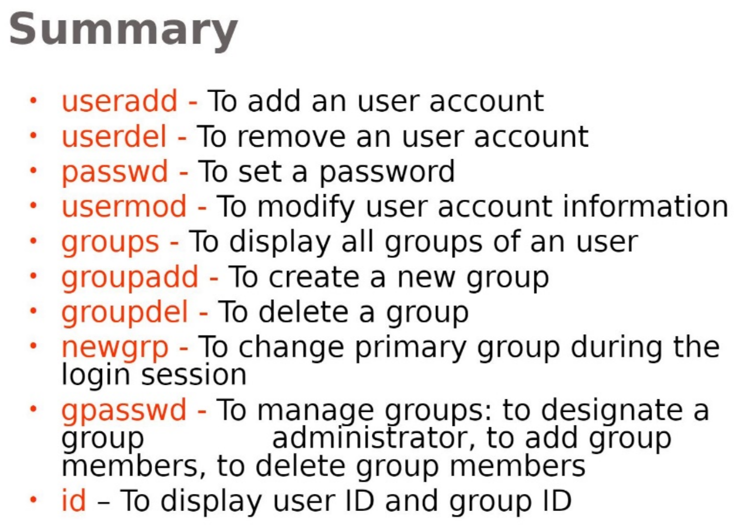
To remove users is a lot more simple; we just use the userdel command. The only option to really consider here is -r which will remove the user's home directory and everything inside.

5.5: User modification

To modify a user's settings (the options which were available to us with useradd and within /etc/passwd) we can use usermod. This command functions the same as useradd, only this time the options included are overwritten. There are also additional options such as -L and -U which lock and unlock user passwords. Another useful command is chsh which will change the default login shell of a specified user. This can also be done with usermod but chsh is simpler. The view a list of all the available shells in /bin, we can either ls it or cat /etc/shells which is a lot less cluttered. Additionally, to view the current shell we are using, we can echo $SHELL (though this command only works in certain shells like bash).

5.6: Group Management

The same commands can be applied for groups such as groupadd, groupdel and groupmod. There is also newgrp to change the primary group of a user and gpasswd to modify the settings in the /etc/group file. Here is a summary of all the most important commands for user management.



Summary

- useradd - To add an user account
- userdel - To remove an user account
- passwd - To set a password
- usermod - To modify user account information
- groups - To display all groups of an user
- groupadd - To create a new group
- groupdel - To delete a group
- newgrp - To change primary group during the login session
- gpasswd - To manage groups: to designate a group administrator, to add group members, to delete group members
- id – To display user ID and group ID

Disk Management

6.1: Mounting

Linux operates a bit differently than Windows in terms of managing partitions, but the core concepts remain the same. The directory tree which we looked near the beginning is the only tree Unix based OSs use. In an OS like Windows, each drive has its own tree. In Unix operating systems however, all drives share the same file tree. Each filesystem on a drive must be mounted in a specific location so that the system knows where to search for certain things. Without getting too in depth, I'll try to refresh you on some things. The basic input output system (BIOS) is low level program which resides on a flash memory chip on the motherboard. It is programed to search within the master boot record (MBR) of all drives for a boot loader program. The MBR is the first sector of a disk. The unified extended firmware interface (UEFI) is the modern version of the BIOS and is essentially the same, only it searches for an EFI partition rather than the first sector of the drive. Once the programs are finished ensuring everything is working correctly by running POST, they run the boot loader (in our case GRUB) which will then interact with the kernel. In order for UEFI to be able to detect the boot loader, the EFI partition must be mounted, which is to say added to the file tree. I'm sure you are familiar with an operating system's PATH variable, which is a long list of locations within a file tree that the system will search for certain things in. Mounting is very much the same thing. When we mount the EFI partition containing GRUB, we are adding it to our "PATH" which is a metaphor for our file tree, so that the system will look there for the boot loader. In other words, until we mount the EFI partition, the system will not be aware of its existence. This is also the case for external devices such as USBs. When a USB is inserted, the system identifies all of its contents/information and places it into a file within /dev for future reference. Let's say this drive is called /dev/sdc. After the drive is inserted, the system will mount it to a location within the file tree as a sort of extension. Usually removable media is mounted to /media, and thus, the system will run a command to place sdc in /media, resulting in a new location: /media/usb. When you perform a safe removal of the drive by ejecting it, the system runs a command to unmount the drive. You may have expected the new location to be called /media/sdc, however, the device file and the mount location are different. The mount location contains the files pertinent to the file tree, whereas the device file is for system-use, such as identifying partitions. Yet another example would be how your primary drive (most likely sda) contains an ext4 partition which is mounted to root (/) and contains the operating system itself. If you're still confused it's alright, just try to understand the difference between formatting (creating partitions) and mounting (adding partitions to the file tree). Some good commands to start with are lsblk, df and fdisk.

lsblk: You may remember that directories are a file type represented by d. There are many filetypes in Linux including links, block devices, character devices, FIFO, and UNIX domain sockets. A block device is one represented with b, and is just a fancy term for a type of drive. lsblk lists all of these block devices. It should also be noted that loop drives are virtual drives.

df: This command stands for disk filesystem, and searches for all of the filesystems located within all block devices. In Linux, a block device is represented as sd followed by a letter representing the drive. sd used to stand for scsi drive but I find it's easiest to remember it as standard drive. In other words, drives follow the order sda, sdb, sdc, sdd, etc. Each partition is identified via a number after the letter. Partition 1 of sda would be sda1, then sda2, sda3, etc. While lsblk retrieves all of the drives, df retrieves all of these filesystems and displays where they are mounted.

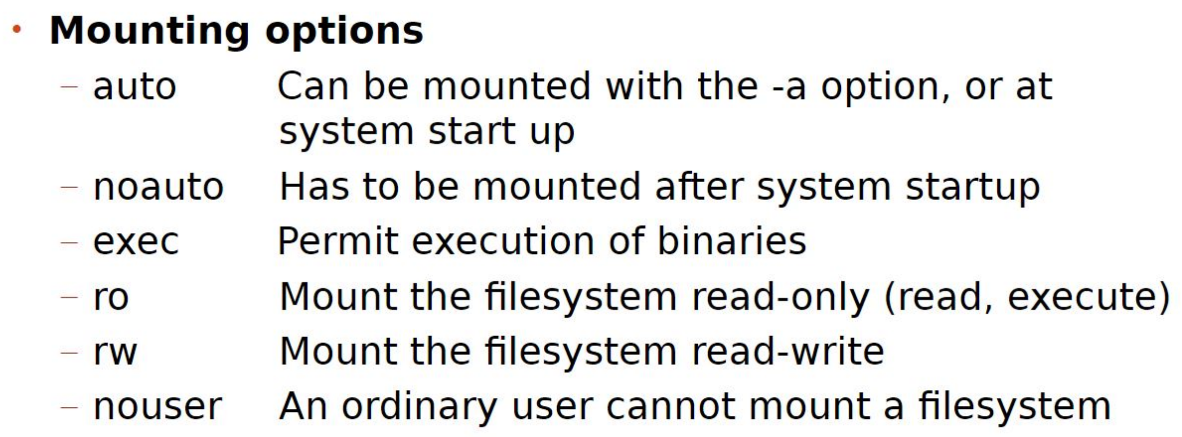
fdisk: This command is actually a program which stands for format disk. This command allows you to format drives, as well as create, delete and change partitions. The GUI alternative which I really like is gparted.

6.2: Creating Partitions

In order to function, Linux requires at least 1 partition containing the OS. The boot partition which was discussed earlier is actually optional in most modern distros (it was just helpful for explanation purposes), yet it is still recommended. It is also good to have a third partition for what we call a swap partition. This is a special partition which is left empty for the system to use. Essentially, if the system is low on RAM, it can use the physical storage space in this partition as virtual RAM. If you have the space, it is recommended to make the swap partition 25% the size of how much RAM you have (ie. if you have 8GB, it should be 2GB). Some people like to have a fourth partition for their home directory, but it's not really required. Anyways, let's pretend as though we didn't have any of these partitions. Well first we would need to find the drive that we'd like to format. This can be done with lsblk. You will have to go off drive size. If two drives have the same storage capacity, it's probably best to unplug one to determine which is which. After the drive you want to format is found, we will use gparted to format it (otherwise use fdisk or the dd command which will set the drive to all 0s). Make sure you back up the drive if there is valuable data on the disk and then delete all partitions. After that's done, we should create the 3 new partitions. The boot partition should be created according to the specifications of the boot loader. For GRUB 2, it's usually good to go with 400M(egabytes) and make it an EFI partition if using UEFI. After that make a linux swap partition which is 25% as big as your RAM. Finally, make an ext4 partition for the OS. ext4 is the standard for regular Linux files. It started with ext, then ext2, ext3 and as of now we're at ext4 (each iteration simply allows for bigger partitions than the last). If for whatever reason you are unable to use fdisk or gparted, you can also use the mkfs command by typing mkfs -t <partition_type> <device> Finally, we will need to mount to partition for the OS to recognize it. This can be done by typing mount <device> <location> such as mount /dev/sdc /media. umount is the option to unmount (umount not u**n**mount) a device. For swap partitions, we use a special command called mkswap instead of mount. Since we don't actually want the user to access our swap partition, we don't want to use mount. However, we still need the system to recognize the swap partition as such. This is where mkswap comes into play. In order to identify the partition as swap storage, you will need to use the command mkswap <device>. You will then need to actually activate this space using the command swapon <device>. To "unmount" a swap partition, we use swapoff <device>.

6.3: fstab

Really quick, we should touch on fstab. fstab is one of the many *tab files in /etc. It is the file which tells the system which partitions to mount on boot. fstab also has its own man page if you're having troubles. It contains quite a bit of info which I'll try to go over. The file contains a comment stating that we can use blkid to list the UUID of each block device. This is helpful if we want to mount partitions on boot which the system wouldn't normally do for us (or to change the location where they mount to) since we will need the UUID to make an entry. If you're not familiar with vim, you can use gedit or some other Windows like text editor to edit this file. The first order of business is to set the UUID similar to how the root partition is setup. Use the UUID of the device you want mounted by entering UUID=<UUID> on a new line. Next, enter the location where the partition should be mounted to. The third field will be the partition type of the partition we are mounting (ex. ext4). After that, things get a bit murkier. We will need to provide options (separated by commas) as to how the partition will be mounted. It must contain at least -ro or -rw which mean read only and read/write respectively, though others can be found in the man pages. The fifth field is not often used. It is used in tandem with the dump command which must be installed first and is used for backups. This will be set to 0 by default or 1 if you'd like to use it (see the dump man page for more info). The final field is used by the fsck command (which checks partition validity) to determine the order in which checks should be performed at boot. The man page states that the root partition should have a value of 1, and that all other partitions should have a value of 2.



- **Mounting options**

auto	Can be mounted with the -a option, or at system start up
noauto	Has to be mounted after system startup
exec	Permit execution of binaries
ro	Mount the filesystem read-only (read, execute)
rw	Mount the filesystem read-write
nouser	An ordinary user cannot mount a filesystem

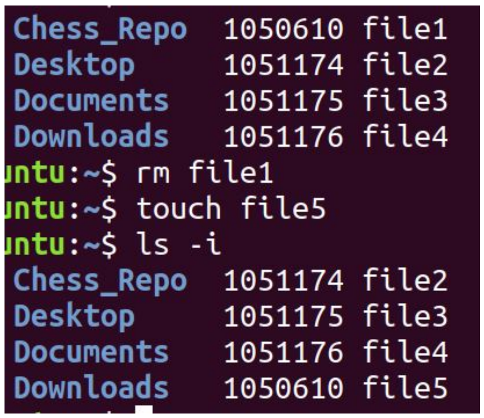
Figure 2-field 4

File-Types in Detail

At this point, I've mentioned filetypes a few times since it can't really be helped, but I'd like to go a bit more in depth with my explanation of each filetype and how it is used. First, to recount the different kinds of filetypes, we have regular files, directories, block devices, character devices, fifo files, links, and Unix domain socket files. Allow me to explain each:

Regular files: A normal file is pretty self explanatory. These make up the vast majority of files on your system and contain basic text. These can be hidden files, scripts, configurations, images, binaries, etc.

Directories (d): Again, you are familiar with directories, though you may not exactly understand how a directory functions. In order to understand exactly how a directory works, you will need to understand inodes. An inode is yet another identifier for Linux similar to IDs. inodes are specific to files and are just containers which contain the next available number. Let's say I continuously create files. Each file's inode will increment sequentially. However, if one file is deleted, the next file will break the sequence by borrowing the deleted files inode so that no available inodes are wasted. If all inodes are occupied, no more files can be created. This can be seen in this image:



```
Chess_Repo   1050610 file1
Desktop      1051174 file2
Documents    1051175 file3
Downloads    1051176 file4
ntu:~$ rm file1
ntu:~$ touch file5
ntu:~$ ls -i
Chess_Repo   1051174 file2
Desktop      1051175 file3
Documents    1051176 file4
Downloads    1050610 file5
```

Notice how the inode from file1 is adopted by the newly created file5 after it is deleted, thus breaking the sequence of files 2,3 and 4. The point here, is that a directory is simply a bunch of symbolic links (which we are about to discuss) pointing to the inodes of files.

Links (l): Links come in two primary forms: hard links, and symbolic links. These two types of links are both useful yet have fairly differing use-cases. A hard link is created with ln <targetfile> <linkname> where the target file is the file you'd like to create a link with and linkname is the name of the link file. Hard link can only be created with a file as the target, not a directory. A hard link is like a shadow of the target file. It is an exact replica, and any alterations to the hard link are applied to the file. This is useful if you want to always have a backup of a file. For instance, I could create a hard link for my .bashrc file by typing ln .bashrc bashrc which would then create an unhidden file called bashrc for ease of access. The hard link and the file are only bound by contents, and not by existence. What I mean is that they can exist independently of each other, so if one is deleted, the other will remain. A symbolic link, or symlink for short, is similar to a shortcut on a Windows or Mac computer. It is a pointer to another file or directory. This means that if I create a symlink with a directory as the target, I can cd into my symlink, and arrive at the same location as the target directory. This works with files too, as I can create a symlink with a file as my target, and when I cat the symlink, or use a text editor to edit it, I am altering the target file directly. A symlink is created using ln -s, and follows the same rules as hard links in terms of syntax. A symlink can be deleted without affecting the target, but removing the target will cause the symlink to become deprecated.

Block devices (b): We briefly talked about block device files such as the /dev/sd** files. As you are no doubt aware, block devices read and write in blocks, which are chunks of storage such as sectors on a HDD or NAND flash blocks in an SSD. This is why we must consider block size when creating a partition (because we need to know how much data to read or write at a time). A block file is simply a file which is capable of doing this reading and writing for us.

Character devices (c): The device file we haven't touched on is the character device. A character device is one which must constantly be monitored such as a mouse, keyboard, or any serial device which sends one bit at a time. As opposed to reading or writing data in blocks, these devices transmit data one character at a time.

FIFO files (p): A FIFO file is akin to using the pipe command, only between files. It stands for first in, first out, but is also referred to as a "known pipe". This is used for reading and writing data between two processes. The file must be open on both ends before data can be passed between them, similar to how a TCP connection works between a host and a server. FIFO files are created using the mkfifo command, though we won't be looking at these until much later on. It should be noted that pipes are a *one-way transaction.*

Unix domain socket files (s): A Unix socket domain file (also known as an inter-process communication or IPC file) is like a two-way pipe which allows data to be transmitted in both directions. If you are familiar with socket programming, you will be somewhat familiar with this type of file. To be more specific, the API of a Unix domain socket is similar to that of a TCP or UDP socket, only rather than connecting via the web, these sockets interface with the Kernel. Two applications with an established IPC connection refer to each other via their file inodes. An example of a Unix domain socket file would be the stdout stream which we will discuss soon.

Intermediate Commands

Permissions:

chmod: Change mode, ie. change file permissions in either octal or visual format.

sudo: Inherit super user privileges temporarily. You must be part of the sudo or wheel group (legacy) to use this command. Sudo settings are located in /etc/sudoers.

chown: Change the user owner and/or group owner of a file.

chgrp: Change group ownership of a file.

newgrp: Change primary group during the login session for a user.

umask: Check or set the active umask.

User Management:

adduser/useradd: Commands used to add a new user. Has many options which are good to memorize.

userdel: Delete a user.

addgroup/groupadd: Add a new group.

groupdel: Delete a group.

usermod: Modify user settings.

chsh: Change the default login shell of a user.

id: Check the UID and GID of a user, as well as the GID of each group they are part of.

groups: Check which groups the user is a member of.

passwd: Set the password for a specified user. If user field is left blank it assumes the current user.

su: Switch users. Defaults to root user/super user.

Drive Management

fdisk: A program which modifies, creates, and deletes partitions.

lsblk: Lists all block devices

blkid: List the UUID for each block device.

dd: Completely wipe a specified drive by overwriting it with all 0s.

mkswap: Sets the swap partition.

swap(on)(off): “Mount” or “unmount” the specified swap partition.

(u)mount: Mount or unmount a partition.

df: Stands for disk filesystem. Lists all mounted partitions and their mount location.

File-Types

ln (-s): Make a link file.

mkfifo: Make a FIFO file.

Other

alias: We did not discuss this command, but it allows you to create custom commands or abbreviate commands which are too long. This is done by typing alias <aliasname>=’<command>’.

find: Search for a file within a directory tree/hierarchy.

which: Searches the $PATH variable for a program executable.

wheris: Searches for binary, source code, and man page for a specified program.

tar: Compress files into a .tar file (Linux’ version of zip).

sleep: Pause for a given amount of seconds.

history: Display the last 1000 commands entered. This number can be changed in .bashrc.

!n: ! followed by a number will repeat the command executed on the nth line of the history output. By entering !!, the previously typed command will be executed. Typically, this is used to enter sudo !! if you forgot to specify it the first time.

Vim Scripting

9.1: What is Vim? What is a Script?

Vim is the go-to text editor for most people who work within Linux environments. Vim is a successor to vi which is a bit older. You can learn how to use vim by typing the command vimtutor which is very helpful. A script is a piece of code which is generally used to automate a process. The entire Linux OS is essentially run of scripts which are just a big list of commands (I mean this is true for all programs, but it is much more apparent in Linux). A script is ran top to bottom always, and does not stop if it reaches an error. This is different from a high end language like Java where the code is first compiled and methods of error handling exist. In this section, we will learn what all the fancy symbols mean in bash and what they do. A more accurate title for this section would be bash scripting since bash can be thought of as a programming language. Not everything I will be explaining here will function in another shell since syntax varies, however, generally shells share a lot of the same principles and are sometimes cross compatible.

9.2: String literals

That's right, we're gunna have to go back to basics for a little bit since bash doesn't handle strings the same as most programming languages. Bash uses Unicode by default. Most letters are recognized by bash as character literals, meaning of course that they should be interpreted either as their letter symbol, or their numeric value depending on circumstance. Metacharacters are special characters which do not function as normal characters. For example, >, *, ?, [], $, #, \, etc. We will be focusing on $ and \ right now. The $ character generally refers to a variable. You may remember earlier that we couldn't use the mv command in conjunction with the pwd command, and therefore, we had to use the dot operator instead. While you can't use the pwd *command*, you can use its variable $PWD which translates to the absolute path. We can set variables using the equals sign, and we don't have to specify any sort of data type since bash doesn't operate with them. This might look something like number=5. White space is not allowed since bash would read number as a command. This variable only exists in the terminal it was created in. This means exiting the terminal will remove the variable. We can use echo to echo $number and have it output 5. The "" and '' quotes can often get confused, yet it is important to distinguish between them. Double quotes preserve whitespace though metacharacters are still interpreted as such. Single quotes do not allow for interpretation of metacharacters; thus, everything is written as a character literal. echo "number = $number" outputs: number = 5, while echo 'number = $number' outputs: number = $number. Moving on, the backslash character '\' is the escape character as it is in most programming languages. On it's own it is printed as a null character meaning that it will be ignored. Also, by default, the echo command does not allow the use of escape characters, and thus you must use the -e option to enable them. For example, echo "\n" would output: n, while echo -e "\n" would output a blank newline. Without double quotes, both will output n since the double quotes sort of concatenate the \ and the n to be one entry recognized by bash. Let's look at one last example. Let's say that our $number variable is still present. We would like to output $5 but this presents a problem. You see, if we try echo "$$number", bash will read $$ as a command for the current process ID (PID), which will output the value of the PID followed by number ex. 7543number. An escape sequence will break the dollar signs, but anything following an escape sequence will be output as a character literal, meaning that echo “$\$number” will output: $$number. To get the desired output, we can separate the literal values and the metacharacters using single and double quotes ex. echo ‘$’”$number” or use the escape sequence to print $ as a literal ex. echo “\$$number” Knowing when to use what will come through practice.

9.3: IO Streams

IO streams are used by Linux to interpret strings. The three IO streams are stdin, stdout, and stderr which stand for standard input, standard output, and standard error. Similar to how programming languages handle errors using IO Exception objects, Linux also has a function for outputting error messages when an exception occurs. Each IO stream is represented by a number. stdin = 0, stdout = 1, and stderr = 2. We are able to manipulate these streams, such as redirecting them to create a log file. The redirect command > redirects stdout by default, meaning that we have to do a bit more work if we want to output stderr. The & operator is a good operator to become accustomed with, as it can join commands so that they are run one after the other. If the & operator is joined with the output operator, it becomes a pointer to both stderr and stdout. This means that > on its own will print stdout, but any combination of & and > will print stdout and stderr ex. 2>&1, >&1, &>, >&, >&21, etc. If we only wanted to output stderr, we could specify it using its associated number (2). This means that 2> or >2 will redirect stderr and not stdout. If it doesn’t completely make sense, play around, as this is an important concept to understand. One last note on this for now is that there is a file located in /dev/ called null. This file is actually a character device file which writes to nothing, and therefore, any input is immediately discarded to the void. This is useful for getting rid of unnecessary error messages or unwanted output.

9.4: Brackets, Brackets, Brackets

Brackets are well known in programming for separating blocks of code by a designated function. In Linux, they work relatively the same, but are a bit more intuitive in most areas. We will look at each type of bracket: (), (()), {}, and [].

Subshells (): A subshell can be thought of as an invisible terminal which can execute commands and return them to the parent shell. This is useful for when we need to run a command within a command. Since a subshell is a process, we can use it to do parallel processing. When a new terminal is opened, this in-and-of-itself is a subshell. We can see this if we run the command: echo “Current subshell: $BASH_SUBSHELL”; (echo “Current subshell: $BASH_SUBSHELL”) which will output Current subshell: 0 (\n) Current subshell: 1. A parent shell can set variables which are usable by the subshell, though this cannot be reversed. To set variables in the subshell and have them be accessible to the parent shell, we must use the export command to export our variables. This command places exported variables (also known as environment variables) in a file so that they can be accessed anywhere, even after the shell has been closed. The command env will list all of the *system* environment variables, and the set command will print all environment variables, including ones you’ve set.

Arithmetic Expansion (()): Double brackets indicate an arithmetic operation. This includes any of the arithmetic operators such as +, -, *, \, %, as well as equality operators like =, +=, -=, *=, \=, %=, ++, --, etc.

An example might be echo $((12 % 5)) output: 2. Notice that the dollar sign must be used to indicate we want the value of the operation. This is called *command substitution* and is used to retrieve the output of a command ran in a subshell. Arithmetic expansion can be used in conjunction with the test condition brackets to check whether or a not a variable is a number. This may be confusing, since we haven't yet covered it, but it should be understandable. Essentially, we test a variable to see if it is equal to its numeric counterpart. If this is true, then the variable is a number. Since everything within the arithmetic expansion brackets must be represented as a number, it can be used for this purpose. In other words, we are basically saying if $num = $((num)), then num is a number.

Lists {}: Curly braces are used to indicate a continuation of a trend. This can mean a sequence of letter or numbers, or it can mean a list or array. For a sequence, we can use a somewhat unique "command" which is to type {<number/letter>..<number/letter>} and this will fill the gaps between the two specified number or letters. For example, echo {1..10} will output each number starting from 1 and ending at 10 on the same line. You could do echo -e "\n"{1..10} to have each number output on a newline. By adding two more dots and another number, we can tell the sequence to iterate/jump by the given number. Ex. echo {a..z..3} output: a d g j m p s v y. Lists don't function in the same manner you'd expect them to work if you've worked with any sort of programming language. In the case of bash, lists can only contain commands. In order to create a grocery list in bash, we would need to use echo commands to make an entry for each item. Each command is separated using the command separator, and a semicolon is placed at the end of the final command like the following: { pwd; whoami; ls; }. Lists in curly braces execute in the current shell, though lists can also be created and executed in subshells using the syntax (pwd; whoami; ls). To set a list to a variable, we will need to use command substitution since there are no metacharacters to denote that a list should be printed apart from how we define it. This would appear as grocerylist=$(echo apple; echo banana; echo orange). Finally, when it comes to arrays, we use curly braces to indicate that more than one variable may be output. To define an array, we use the syntax arr=(item1 item2 item3) where each item can be anything. Indexes are automatically assigned to each element starting from 0. We can print the entire array by typing echo ${arr[*]} or echo ${arr[@]}. The @ symbol refers to "all parameters" (remember this for later), thus these commands are equivalent. We can also specify a specific element echo ${arr[n]}, print the size of the array echo ${#arr[*]}, the character length of a specific element echo ${#arr{n]} or index numbers echo ${!arr[*]}. Arrays are intuitive, yet complex, so play around with them. We can also set variables by doing arr[0]=firstelement

Condition Tests []: The square braces are unique from the rest of the brackets in that they are actually a command. This means that they must have spaces to separate themselves from their contents. Square braces may also be written as [[ ]]. They are related to the test command which is used to compare values. The test command is somewhat obsolete since square braces can be ran on their own. There are various operators used to perform checks between two values. For comparisons between strings it is best to use the standard comparison operators such as ==, !=, <(=), >(=), but for numeric comparisons it is best to use the special test operators which include -eq (equals), -ne (not equal), -gt (greater than), -ge (greater than or equal to), -lt (less than), -le (less than or equal to). There are additional operators such as -z which will test if a string has no characters or -x which checks if a file exists. A condition can be written as [5 -lt 6] which will evaluate to true or false. We will next discuss the $? variable which will contain the evaluation of the previously ran command. In other words, if the statement evaluates to true, this variable will be set to 0, and if the statement evaluates to false, it is set to 1. This is how the if statement along with other test cases function. I mentioned in the arithmetic expansion section that we could use square brackets in conjunction with double brackets to test whether or not a variable is a number. This would appear as [$var -eq (($var))].

9.5: Other Operators

At this point we have discussed all sorts of operators including arithmetic operators, comparison operators, equality operators, and the special test operators. However, there still remains Boolean operators, as well as some other special metacharacters and wildcard characters. I would like to briefly explain each one since most of them are important to know.

***:** The first of two wildcard characters is the * symbol which denotes “all” This can be used for many things such as grabbing all files and subdirectories or searching for items with a similar pattern. For example, if I had a file named dogbreeds and a file named dogtags, I could ls dog* and it would return anything which begins with dog, no matter what proceeds. This is also nice for finding files with the same extension by typing ls *.<extension>

?: The second of the two wildcard characters, the ? functions the same as the * character, only ? is limited to one character difference. For example, I could have a file named bug and a file named dug and then type ls ?ug to return both. If any characters before or after the ? differ, ls will not retrieve them.

$?: This variable is called the exit status. As I mentioned, it contains either 0 or 1 to represent the boolean exit status of the previously ran command. We could use it to write an if statement if we wanted. Ex. [6 -eq 21]; [$? -eq 0] && echo “How is this possible?!” || [$? -eq 1] && echo “6 doesn’t equal 21!”. This will make more sense as you get to the boolean operators, but for now just understand the all test conditions use similar logic to check the exit status of the previous command to make decisions as to what will happen next. The exit status does not typically need to be explicitly called like it was in this example, and therefor the example provided can be abbreviated (which I will demonstrate with the boolean operators).

$#: This command contains the number of parameters that were passed to a function or script. You see, in bash, we can pass variables to a script as we call it. If you create an extremely simple script which contains the command echo $# and then run it, it will return 0. However, if you call it using parameters, it will echo the number of parameters you entered. ex. ./paramscript.sh param1 param2 param3, will return 3.

$$: We looked at this command much earlier which prints the PID of the current process (useful for process management in top or htop).

&&: The boolean AND operator returns an exit status of 0 so long as every command evaluated to true. You know it well in the format of an if statement: if (condition && condition)... This is really useful in bash because it allows us to do what we did in my example for the exit status variable. If the previous test evaluates to true, or the previous command ran successfully, continue to the next test/command. This means that we could remove the need for the exit status in the previous example and write it as
[6 -eq 21] && echo “How is this possible” || echo “6 doesn’t equal 21”.

||: The boolean OR operator only takes action if all previous commands evaluated to false. In the example above, [6 -eq 21] returned false, and so it jumps all && operators and only runs whatever proceeds ||.

&: This operator will run a command in the background similar to using ctrl+z to stop a process and then entering bg for background. The & operator should be placed after the command you'd like to run in the background.

;: We have also discussed the semi colon which is the command separator. This will run the proceeding command whether or not the previous on succeeded/failed.

#: The hash symbol on it's own means a comment in bash. For example echo #hello will not print anything.

[]: We discussed the square brackets command, but that is different from the square brackets used for pattern searching. Without spaces, the square brackets will look for a certain range of characters or integers. It is definitely one of the more complicated operators. Let's say I have 3 files, file123, file125, and file223. I can search for numbers within a range of 0-9 for each 10s column. Since each file's highest 10s column is the 100s column, I will need 3 square brackets eg. file[][][]. Each bracket can contain numbers. These numbers cannot exceed 9 because we are looking at each column ie. I cannot have file[72][68][90] and expect it to search for file706890. Instead, this would pair each combination of numbers together ie. file769, file760, file789, file780, file269, file260, file289, file280. We can also include a range within a bracket as mentioned such as [0-5]. Remember, this can't go over 9! [0-72] will actually be seen as 0-7, 2 which is arbitrary since 2 is included in 0-7. So, long story short, is we were to ls file[1][0-4][35], the output would be file123 and file125.

9.6: Functions, Loops, and Data Structures

For the final preparation into vim scripting, you should understand functions, how if statements, while loops, and for loops are written, as well as case statements.

Functions: In bash, functions are written as <functionname>() {}. Unlike higher level OOP languages, there is no specific return type or access modifiers. Functions don't actually accept specific parameters, thus the () is obsolete (and can be left out if you'd like). Instead, parameters are passed as we discussed earlier. To call a function, it is as simple as typing its name. To pass parameters, you type them after the name of the function ie. function param1 param2... There are special variables to retrieve these parameters. These variables are the numbers 1-9 ie. $1, $2, $3... if I wrote a function:

```
rearrange() {

        echo "$3 $2 $4 $1 $5"

}

rearrange name my is hello neil
```

This would output: hello my name is neil. Remember, functions must be called after they are declared since bash reads top to bottom unlike compiled code. This means it will continue even if errors exist until it reaches the EOF. We also have to distinguish between local and global variables. When a variable is exported, it becomes global permanently. A variable is set to global by default while the shell is still active, but will be removed if not exported when the shell is terminated. A local variable must be declared as such using the keyword local. This will cause the variable to only exist within the block of code it is declared in such as a function. If a global and local variable exist with the same names, the local one takes precedence until the block is exited.

if Statements: If statements are pretty straightforward but arn't declared conventionally. For example, if is a command so we leave a space, then the test condition [ ], followed by a command separator and a then keyword for every test. Else-if is written as elif, and the entire if statement must end with fi. Here is an example since it's difficult to describe with words.

```
if [ -z $USER ]; then
        USER=neil
        echo "Welcome $USER"
elif [ $USER == 'John' ]; then
        sudo su - neil
else echo "Welcome $USER"
fi
```

While Loops: While loops also function normally but are declared a bit oddly as well.

```
while [ true ]; do
        echo "Press Ctrl+c !!!"
done
```

In this case, the string Press Ctrl+c !!! will be printed until the user force quits the application. Do while loops also exist but I shouldn't have to explain those.

For Loops: For loops function quite differently in bash. A for loop only really works like a for each loop would work in a language like java. The syntax follows the format for $var in <list>. Since this is also a loop, we begin with do and end with done.

```
for i in 1 2 3 4 5; do
        echo $i
```

```
done
```

The output will be the numbers 1 through 5, each on a newline.

Case Statements: Case statements are fairly unique as well. They begin as: case $var in. Each conditional check is done before a ')' bracket. You can have multiple checks by separating them with a pipe '|'. The "else" statement is replaced with *) and the case statement ends with the esac clause. Since case statements require breaks between each conditional check, each condition is ended with ;; This ends up looking something like:

```
x=4

case $x in

        1) echo 1

        ;;

        2 | 4) echo $x

        ;;

        *) echo 3

        ;;

esac
```

This pretty much covers it for vim scripting. Be sure that your files end in .sh and that the first line of the file is #!/bin/bash. Despite file extensions not mattering in Linux, certain extensions will check the first line of a file to know for sure how to interpret the file. Some Linux distros don't default to bash so we use #!/bin/bash (pronounced "she-bang" or "hash-bang") to ensure it is ran with the right shell since syntax varies.

systemd and services

10.1: Services

A service in Linux is an application/daemon running in the background of your system. For example, you will have a service controlling your desktop environment (X11) or window manager, or a service managing your internet connection (default is NetworkManager). systemd is the Primary service which boots your system (as PID 1) and runs all other default services. systemd is a replacement for sysvinit. It manages file types known as units. There are 11 kinds of units, for example .service, .mount, .socket, .device, etc. The top, htop and ps applications are there to help the user manage the .service processes. A service can have multiple processes running for performance. For general unit management on systemd devices we use the systemctl command (use service on init devices). This command allows us to manage all systemd units. It includes commands like systemctl start/stop/restart to temporarily start or stop a service, systemctl enable/disable have a service start or keep from starting at boot, systemctl list-units to view all system units, etc. The directory /usr/lib/systemd/system/ contains all units provided by installed packages, whereas /etc/systemd/system/ contains units installed by the admin. The aforementioned ps command is like top or htop, only it doesn't operate in real time. This is nice because it's annoying having processes shift around depending on their memory management. Specifically, the ps aux command is often used in conjunction with grep. ps aux is different from ps -aux so be careful. aux is an acronym. a=show processes for all users, u=display the processes user/owner, and x=show processes not related to a terminal. So, for example if chrome crashes, we can use the search feature in htop or ps aux | grep "chrome" to find its PID. Then we can use the kill command to kill the process. The kill command is a bit confusing since there are various signals we can send the process such as SIGHUP to restart the process, SIGKILL to kill it, SIGSTOP (equivalent to typing Ctrl+z), etc. (you can see all of the signals using kill -l). So, to restart chrome we'd enter killall -s SIGHUP brave. Since chrome uses multiple processes to run if we just entered kill -s SIGHUP brave we would get an error since we'd have to specify each individual PID being used. In theory, killall will kill all of the daemons processes though this is not guaranteed so use htop or ps again to be sure.

10.2: X Window System

We briefly covered the difference between window managers and desktop environments, as well as the X11/X.org communication protocol. And that's what X11 is... It is a server side application which clients may connect to. This doesn't even necessarily have to be anything graphical, though it usually has to do with drawing things to the screen. The X Window system is what we are concerned with. This is the process which actually manages the GUI of the system. systemd will enter the startx command (which you sometimes have to do yourself on minimalist installs) to enter your DE or WM. Without the X window system, you generally only have access to a single terminal with no resizing abilities or anything like that. This is how most POSIX (Unix and Linux) systems used to work back in the day. Of course, back then you had real terminals and not terminal emulators.

Networking in Linux

11.1: The Basics

The most basic command for networking is ifconfig which stands for interface configuration. This command will display all interfaces and their information. For example, the lo interface is the loopback interface. inet refers to IPv4, and naturally, inet6 is IPv6. ether refers to the MAC/physical address and there are also fields for RX packets and TX packets. RX refers to packets received and TX stands for transmitted. The bytes represent the total amount of data sent/received during a login session. The -a option will include disabled interfaces as well. To disable or enable interfaces, we can use the up or down specifiers after the interface. For example ifconfig vm02 down. You can also assign an ip address by entering ifconfig <interface> <new_address>, a new subnet mask by entering ifconfig <interface> netmask <new_subnetmask>, and/or the broadcast address by entering ifconfig <interface> broadcast <new_broadcast_address>. These can be strung together in one command ex. ifconfig eth0 192.168.4.56 netmask 255.255.255.0 broadcast 192.168.4.255. Along with this, you should definitely be familiar with the ping command for testing network connections and arp for discovering information about physical addresses.

11.2: More Networking Commands

For more advanced users, the following commands may come in handy when debugging network connections. The first command is the netstat command. Without any options, this command will display established port connections (TCP or UDP) with web servers, as well as connected ports with Unix domain sockets. There are various options which can display routing tables (-r), network interface info (-i) or multicast group membership info for IPv4 and IPv6 (-g). Personally, I also find the -p option useful, as it will show program information such as the PID and associated name of the application running over the port. The traceroute command will display each hop address between two nodes. This is typically between your router and a destination router. nslookup is a nice one if you need to find dns information about a particular web address. This command will display the public address of the website's dns server as well as the public ip address of the website itself. The final command which you should know is iwconfig for configuring wireless interfaces. Apart from these, there is not much more you will really need unless you are actually planning on being a network administrator.

Advanced Commands

Services and Service Processes

systemctl: Manages systemd units.

top/htop: Graphical interfaces for viewing process information such as name and PID.

ps (aux | grep <process_name>): A static version of top and htop, output as text.

kill: Force quit an application via PID or name. SIGNALS may be passed to the process as well.

Networking

ifconfig: The primary command for viewing information about network interfaces.

ping: An essential tool for testing network connectivity.

arp: Displays information pertaining to physical addresses within the broadcast domain.

netstat: Information on ports, routing tables, multicast group info, network interface info, etc.

traceroute: Displays the public address of each hop between two nodes.

nslookup: DNS information about a given host.

iwconfig: Wireless network interface info, similar to ifconfig.

Other

free: Provides information about available memory in bytes.

diff: Compares two files and outputs the differences between them. This function is often used in conjunction with the output operator (>) to create .diff files. .diff files can be used to 'patch' other files. The output will contain each individual line which differs with either a < or > denoting whether the line has been removed or added respectively. This can also be ran with the - -git option for a more legible output using + or – rather than < and >.

patch: Patch essentially updates existing files with pre-existing .diff files. The best example that I can think of is suckless software, which is an organization that writes cool code for Linux, often written in C for speed, and often very minimalist (check out their site). So if you were to patch dwm (the suckless dynamic window manager), you would install the .diff file with wget, and then (assuming the .diff and the file you want to patch are in the same directory), enter: patch -p1 <file_to_patch> <diff> where -p1 means that both files are in the same directory level. The file should have now updated to reflect the changes in the .diff file.

make: Arguably the most difficult util to understand if you are not familiar with C, make is how we build executables in C. Many low-level packages in Linux are written in C and will require you to build them, ie. compile each function or header file into one deliverable. This is done through a Makefile which I can't explain here. Typically, if you see a file called 'Makefile', you will be required to run sudo make install or sudo make to build the package.

Git and Github

13.1: The Difference

Git and Github are different things but function together to create a very helpful resource for developers. There is a lot of confusion for newcomers due to a lack of understanding of what Github is and how it functions. Hopefully this section will clear things up. The reason I have included Git and Github in my GNU Linux is because Git functions best in Linux hands down and I strongly recommend against using it on Windows.

As I mentioned, Git and Github are separate entities but coordinate to create essentially one service. Git should be installed in your Linux environment with whatever package manager you're using. For Ubuntu this will be sudo apt install git. There is no GUI for Git, just a man page with its various commands. Essentially, Git is your local repository manager and it will track all of the files you tell it to track. Every command begins with "git" followed by the parameters. Github on the other hand, is a cloud-based application which operates remotely and is free to use. You will need to make an account first, and then you will be able to create repositories which are just like directories for projects. You can really put anything in Git such as Linux config files, pictures, or what we will be using it for, code projects. Git is open source, which means that so long as the user has made their repository public, you may download, modify, and share it without copy right infringement. This is useful for teams, because it means that a project manager/tech lead can add source code to a repository and have each dev clone it so that they can modify it and merge their code with the source code located on Github. Git has a few other features which makes it worth while. We are able to create commits which is like a save state that we can revert back to if necessary. If we want to create multiple versions or builds of the same project, we can branch the timeline and have two separate versions with their own respective commits and modifications. These branches can also be merged back into one branch. Branches are often used in group environments so that additional features don't overlap or cause errors. The owner of the repository has the ability to view any modifications made to the code before approving a merge. Github will also notify them if there are major conflicts (more on this later).

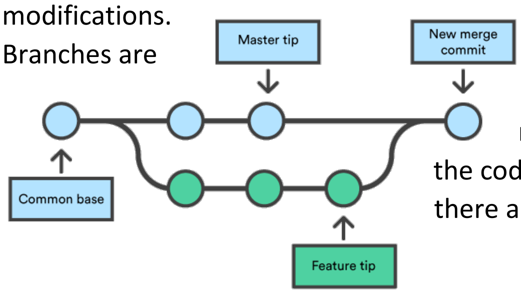


Figure 3 Dots=commits, lines=branches

13.2: Git Workflow

As I've alluded to, there are various states which the project must undergo. There are three primary states we refer to: Working directory, index, and Head.

Head: The main branch located on Github is called origin by default. The origin branch is also known as the *head* of the project. It is the 'real' code which is actually having changes made to it and being developed.

Working directory: The working directory refers to your local clone of the head code. It's name by default is master. Therefor, you will often see mention of the master branch and origin branch. Specifically, the working directory is the last commit you made to your code.

Index: The index is like the staging area refers to the code in production (the code being worked on before/after a commit). It is kind of like the code in limbo.

These states are not essential to understand, but they are helpful for describing workflow. The head is cloned into a local repository, then becomes the index as it is modified, and then becomes a working directory when a commit is made. The working directory is what will be pushed to Github to be merged with the origin branch.

13.3: How to Use Git

There are a few commands which must be executed in a consistent order for us to be able to actually get to the point of successfully pushing our code to Github. It will be easiest for me to make a list of the commands in their proper order where I will discuss them in further detail. Before we get into the primary commands however, if this is your first time using Git on your device, you will need to set your username and email for the Git program. This is done by entering the commands git config - -global user.[name][email] "[username][email@example.com]"

git init: The command git init is used to mark a directory as a Git repository. It will be applied to the parent directory, therefor, you will need to cd into the directory you want to initialize, and execute the command there. Just because Git recognizes the directory as a repository, doesn't mean it is actually tracking any files. This is where the next command comes in to play.

git add: This command can be run on files and subdirectories to have Git track them. Git will not operate on anything which has not been added. Changes be it additions, code deletion, or changes in added files/directories will be marked by Git. Typically, the * is used to add everything.

git remote add origin https://www.github.com/<Repo_Name> : This lengthy command is used to add our remote repository on Github to our local repository so that we know where to push our code. This command also sets the name of the remote branch that the user will be pushing to. It does not necessarily have to be called origin, but it is best to be left as such to avoid confusion.

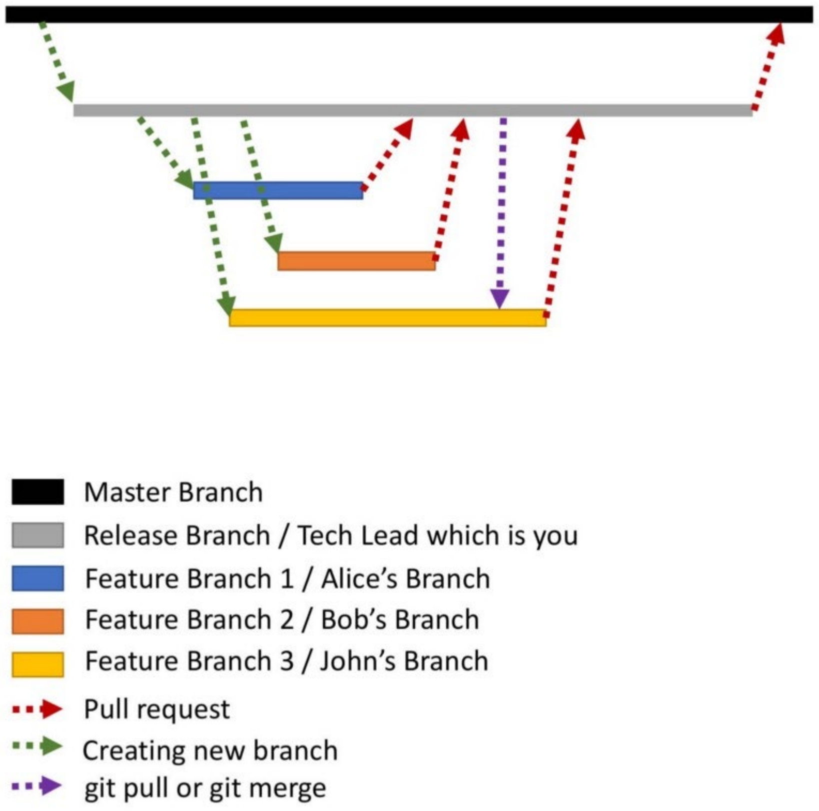
git commit [-m]: Commits can be made as often as needed and will set restore points for the current branch. Use it often, within reason, since pointless commits will only make it harder to backtrack. The -m option means message. If left out Git will still ask you to enter a message so it's really not optional and I would recommend using it. -m Accepts a string naturally. Ex. git commit -m "First Commit".

git push -u origin master: This command can also be written as git push –upstream origin master. This command is used to send our working directory code to the head (hence why we say origin master: because we are attempting to push master to origin).

git pull: I strongly recommend, as a final precaution, to use git pull directly after pushing to Github. While not necessary, I have had so many instances where I have had to delete my repositories because my local code in master did not update to the code located in origin (which it is supposed to do when you use push). If this happens, you will make a bunch of changes to your code, and upon the next push, you will encounter an error: code in master has no related history with origin. Git pull will ensure this doesn't happen. It is actually a combination of two commands, git fetch, and git merge. Git pull will first call git fetch, which will get the code located in the remote repository (Github) and bring it to our local repo. Then, git merge will attempt to merge the fetched code with the local code, essentially synchronizing the two together. This command will catch the conflict mentioned above.

Other helpful commands: Other helpful commands include git status, which will display tracked files, and git checkout <branch_name> which will switch the current branch.

13.4: Using Git Effectively


Master Branch
Release Branch / Tech Lead which is you
Feature Branch 1 / Alice's Branch
Feature Branch 2 / Bob's Branch
Feature Branch 3 / John's Branch
Pull request
Creating new branch
git pull or git merge

This section will be directed at group/team projects specifically, but is still helpful for the regular user. We will be using our applied knowledge to understand how a group project might function using Git and Github (this section is almost entirely stolen from here https://www.freecodecamp.org/news/how-to-use-git-efficiently-54320a236369/

We will begin with our master branch (which is confusingly orgin in this case) at the top in black which will most likely contain the outline for some project. The example given is a Facebook clone. Absolutely no one will touch the origin branch since we always want to have this copy ready for deployment. When the development begins, a new branch is created via git branch <branch_name>. In the example, this is called release. It will be handled by the tech lead. For each additional feature we want to add to the project, the developer creates a new branch called feature/featurename ex. Feauture/loginScreen. Each developer is assigned their own branch and they will work on that branch, making their own commits and changes. When they think their feature is finished, they will attempt a pull request. This is a button on Github, and is not to be confused with the git pull command. A pull request can be thought of as a formal push to the remote repo. If the pull request fails, either the tech lead can view the code and edit it, or the developer can do a git pull or git merge to see the dependencies and fix conflicts themselves. This can be seen in the image above for John's branch. Only at the end of the project is the release branch merged with origin.

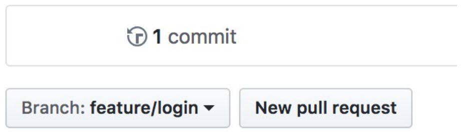
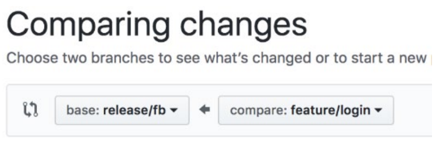


Figure 4-Pull request


Comparing changes

Choose two branches to see what's changed or to start a new

13.5 (Optional): Github and SSH

Secure Shell (SSH) can be used to have a central server host Github repositories, and have clients access them. This is kind of useless though, unless you're too lazy to login to Github on each new device. The best use of the SSH feature is to get rid of the required username and password entry upon every push you make. Within Github, locate Settings > SSH & GPG Keys > New SSH Key. Open a terminal and type ssh-keygen -t rsa -b 4096 -C "youremail@example.com" . This will generate a private key (in other words do not share this), tied to your account. When it asks you where you'd like to save it, it is best to hit enter and have it save in the default location (~/.ssh). Next, it will ask for you to set a password for SSH. Obviously, hit next since the whole point is to get rid of our password. This will not put you at a high security risk (to reiterate, an attacker would need access to your actual computer to pose any threat). The key must be managed by a third party trusted by Github. They will prevent attacks as best as they can. Type eval "$(ssh-agent -s)". This ensures that our device doesn't already have a key associated with it. Then, to give them access to the key, type ssh-add ~/.ssh/id-rsa which is where the key should have been saved if you followed the first step correctly. You will need to cat the file id_rsa.pub and then paste the key into the drop box we opened to have Github recognize the device as valid. If the remote repo has already been set, it will need to be reset to git@github.com:<Repo_Name>.git Every push you make will be to this repository and this removes the need to add credentials every time.

Reverse Engineering

I thought that as a sort of final addition to this beginner's guide on Linux that I would explain how to reverse engineer a basic program since I thought it was kind of neat. When we compile a program, the compiler obviously spits out a binary file which is the file that the computer reads. Within the GNU Compiler Collection (GCC), used to compile certain programs, a package called the GNU Debugger (GDB) is also included. This package can read a binary file and disassemble it, outputting the assembly instructions. Let's say we have a compiled C program with a main function. We should enter set disassembly-flavor intel to change the default AT&T syntax to Intel syntax. Now we'd enter disassemble main or whichever function we wanted to look at. To understand the output, you will need to investigate assembly a little bit, which may sound daunting, but at its core is pretty straightforward. For now, we want to focus on our control flow operations (jump [jmp], call [call], and compare [cmp]). We can set breakpoints using break *<function> to set the breakpoint at the start of a function or break *<address> to set it at a specific address address ex. break *main. It is useful to draw a map of the control flow or use a program to do it for you such as hopper, IDA, or radare2. Entering run will run our program and stop at any breakpoints we set. The command info registers will display information about all of our registers. The command ni allows us to step through the program 1 line at a time. Another cool thing we can do is set the value of registers by entering set $<reg>=<value> which can be used to bypass checksums or other security checks. Obviously this won't be a full guide on reverse engineering but I hope it gives you an idea of what capabilities Linux has.