The GNU + Linux Operating System

Introduction:

Although niche to most people, Linux is an extremely significant achievement in the world of computing (with a rich history to boot). It was created in 1991 by a man named Linus Torvalds, who was a computer science graduate from the university of Helsinki. We will come back to Linus momentarily, but first, it’s important to understand the history of Unix. What’s Unix you ask? Unix was invented in 1969, with team leads Ken Thompson and Dennis Richie. Unix was invented for use within the Bell System at Bell Laboratories, and is commonly well respected for its design philosophies, often referred to as “The Unix Philosophy”. It was a modular and portable operating system that was ported to many systems in the early 1970s under various vendors. You may know Dennis Richie as the author of the C programming language. Originally, Unix was written in assembly language, but was eventually rewritten in C after it was created by Richie. In the late 1980s, an open operating system standardization effort known as POSIX was released, which attempted to unify the multitude of distributions of Unix released under various vendors. POSIX was very successful, and we still have operating systems being built to comply with its standards to this day. Cut forward to 1983 - man named Richard Stallman, a programmer and freedom activist, launched a campaign called the GNU project. GNU stands for “GNU, not Unix”. Stallman’s goal was to write an operating system composed entirely of free software that would be based off Unix. At the time, the GNU Project had written most of the fundamental software required for an operating system such as the GNU compiler collection (GCC), a rewrite of libc called glibc, and rewrites of many of the Unix core utilities, now called the GNU core utilities (utils for short), however, they lacked a kernel to drive the OS. Most people think of “free” to mean $0.00, but by “free” we mean free as in freedom. Stallman defines his idea of “The Four Freedoms”:

1. The freedom to run any program as you wish, for any purpose

2. The freedom to study how the program works, and make modifications to you it as you wish. Open-source code is a pre-requisite for this.

3. The freedom to redistribute software

4. The freedom to redistribute modified copies of the software

Along with starting the GNU Project, Stallman is also noted for starting the Free Software Foundation (FSF), developing the famous text editor, EMACS, and for creating the GNU Public License (GPL). Cut back to Linus Torvalds - Linus attended one of Stallman’s lectures and was compelled to take on the task of developing a kernel that would satisfy the GNU Project. Thus Linux (often called GNU/Linux or GNU + Linux) was written, heavily based off Unix. The Linux kernel remains open source to this very day and accepts contributors on GitHub.

The Linux Kernel:

If you aren’t familiar with kernels, that is okay – they are not something that every developer will work on in their lifetime. A kernel is essentially the heart of the operating system. Its primary purpose is to be the bridge between the hardware of a device and the operating system. To be a bit more specific though, the kernel often deals with hardware interrupts, device drivers, process management, resource management, etc. The Linux kernel is regarded as a monolithic kernel, as opposed to a micro kernel. This means that it oversees extra functionalities which aren’t necessarily essential for the kernel to operate. The kernel can be modified by vendors and can have extra applications packaged alongside it to create a unique copy of Linux. These distributed copies of Linux are known as “distros” for short. There are hundreds, possibly thousands of Linux distros for you too choose from; each of which is different, but unified through the core functionality of the Linux kernel, and their compliance to the POSIX standard.

Some Linux Distros:

As I stated earlier, there are many distributions of Linux. Since the Linux kernel falls under the GPL license, anyone is free to modify it and redistribute modified copies. A few “major” distros gave rise to many child distros. These “major” distros include Slackware, Debian, RedHat, and Arch. If you look at the Linux timeline Wikipedia: <https://en.wikipedia.org/wiki/List_of_Linux_distributions> you can get a better visual. The only popular child distro that came from Slackware (in my opinion) is OpenSUSE. Popular children of Debian include Ubuntu, Raspbian (for Raspberry Pi), Kubuntu, Linux Mint, Pop OS, Elementary OS, Zorin, etc. Popular RedHat-based distros include CentOS, Fedora, Qubes OS, etc. And finally, popular Arch-based distros include: Parabola, Manjaro, Black Arch, and Artix.

Some of the Linux community gets very concerned over so-called “bloat-ware” i.e., software that is either monolithic (such as systemd, arguably), packed with extra (and more importantly, unnecessary) features, or that has poorly written and unoptimized code. You may hear certain distros referred to as “minimalist distros”. Generally, minimalist distros such as Arch Linux, Void Linux, and Gentoo come packaged with next to no applications, and expect the user to install and configure most applications by themselves. Stay away from these distributions if you are a beginner, but don’t disregard them as “too difficult” either. Installing a minimalist distro is usually not be as hard as some people assume, and even if you don’t stick with it as your preferred distro, it’s a really good learning experience, if nothing else.   
  
Components of a Linux Distro:

I’ve been sort of vague about the key differences between distributions up until now, but I aim to cover what you can reasonably expect to find in a typical Linux distribution. Of course, the heart of every distribution is the Linux kernel. The kernel may have minor changes from distro to distro, but for the most part, it usually remains unchanged. Most Linux distributions come packaged with a boot loader, which is launched by the system firmware (BIOS/UEFI), or possibly u-boot if you’re on ARM. The boot loader is responsible for booting the OS. By far the most common boot loader is GRUB, which stands for “Grand Unified Bootloader”. The distro will also come packaged with various services known as “daemons”. A daemon is essentially an application process that runs in the background. Daemons often provide some low-level functionality. For example, sshd is the Secure Shell (SSH) daemon, which provides ssh functionality. All Linux distros must come with certain ‘core’ utilities. When we talk about utilities, we’re referring to the commands that you can type into the command line to have your shell execute. Some basic utilities include cd, ls, cat, etc. Depending on the distro, you may have more or less utilities when you first install it, but each distro should have a package containing that distro’s core utilities, which are the collectively decided minimal utilities required for a base installation. Speaking of shells, every distro comes with some shell, which is a command line interpreter. It reads in shell code from the console and executes it as commands. Common shells are the Bourne Shell (sh), the Bourne Again Shell (bash), the Korn Shell (ksh), the C Shell (csh), and the Z Shell (zsh). Finally, a lot of distros come with a package manager. Packages in Linux refer to utilities + extras e.g., documentation, config files, etc. A package manager is in charge of downloading applications from some centralized repository (files on a server) and then installing and configuring them. It is also responsible for updating packages on your system and for uninstalling packages.

Installation Process:

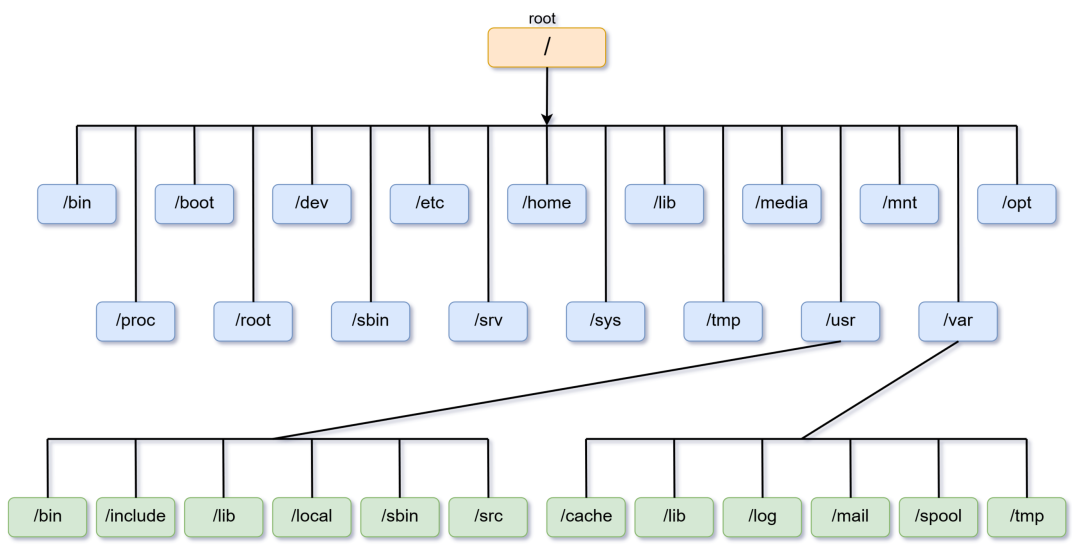
Installing a Linux distribution varies from distro to distro, but generally, the process remains similar. Because of the history of Linux, and of operating systems in general, Linux distros are often distributed as ISO files which represent optical disks of the past (CDs and DVDs) in a digital format. ISO is not actually an acronym for anything, it just refers to the ISO 9660 standard used for CD-ROM media. Anyways, these can often be torrented using qtorrent (which is not illegal unless the distribution costs money) or downloaded from an official website or mirror server depending on the distro. It is recommended that you compare the checksum that is often provided on the website with the checksum of the ISO file to make sure that it hasn’t been tampered with (if possible). You may want to use a tool such as Rufus (if on Windows) to “burn” the ISO onto your thumb drive. This will optionally format the drive at the same time, which is ideal. Once the ISO is on the thumb drive, entering your PCs BIOS/UEFI will allow you to select the thumb drive as your primary boot device. Upon saving the settings and rebooting your computer, the boot loader will launch the Linux image. This is often referred to as a “live CD”. The operating system is not yet installed on your computer’s hard drive, it’s just a live image that you can mess around in, sort of like a VM. This is good for testing that all your devices connect and that you enjoy the UI. Changes made to a live CD are volatile, meaning on reboot, they will no longer be saved. The live CD will sometimes come with a document that will explain the process of installing the OS, or sometimes, the installer is launched as soon as you boot the live CD, and other times, you just need to read the documentation online. It is possible to dual boot both Linux and Windows if you have enough hard drive space, but I will not cover how to do that here.

A Fresh Install:

Once Linux is installed to your hard drive, you will no longer require the live CD. There are some things to note about the environment that is executed on boot. First and foremost, you will almost always start in some login screen. The login screen is the basic authentication method for most operating systems. Authentication on Linux is usually controlled by a daemon known as PAM (Password Authentication Module). Presumably, the installation wizard for the OS requested a username and password, so this is where you’d enter that. Upon entry, you will most likely be greeted by a GUI (not always, but usually). The OS’s GUI is known as the Desktop Environment, or DE for short. Popular DEs include KDE Plasma, XFCE, Cinnamon, GNOME, MATE, etc. You will probably have some sort of application launcher, some sort of file explorer/manager, a terminal emulator, and a system info bar at the top of the screen. These all depend on the distro, as well as the DE. For example, Ubuntu uses GNOME as its DE. Nautilus is the GNOME file manager, Applications Menu is the GNOME application launcher, and GNOME Terminal is the GNOME terminal emulator. Speaking of the terminal emulator, Ctrl + Alt + t is usually the shortcut to open a terminal session. It’s always good to update your packages and your system on a fresh install. This is because your ISO could come with outdated packages (if you’re not on a rolling release distro such as Arch). Assuming that you are using a Debian-based distribution (e.g. Ubuntu), your package manager will be called “apt”. If you open a terminal, we can enter our first commands. The first command to run is “sudo apt update”, followed by “sudo apt upgrade”. The former command will download any updates to packages installed on your system, and upgrade will install those updates. This includes updates to the kernel. If you want to see what packages are installed on your system, you can run “apt list --installed”. apt has other features as well, such as searching the repositories for packages with apt search. These are summarized in the manual for apt. On nearly all distros, packages must come with documentation in the form of user manuals called man pages. These docs can be found with the “man” utility. For example, to learn more about apt, you can run “man apt” which will bring up the documentation for apt. The man command is one of the most useful and essential commands for becoming self-sufficient in Linux, otherwise you will be doing google searches all the time, which can be very time consuming.

The Linux Filesystem:

Linux has a very intuitive filesystem with a tree-like structure. On Windows, each hard drive is its own filesystem, but on Linux, there is only one filesystem which can be shared with external devices. Here is an example of what your filesystem might look like (give or take a few directories).



The star player here is the root directory. Root is just a regular folder from which all of our other directories stem. It is denoted by the forward slash symbol (/). Within root we have multiple directories which are each designed for some unique purpose. I will cover each in depth (not including the green ones).  
  
**/bin:** The bin directory stands for “binaries”. Certain utilities are stored here, such as ls, cat, mv, grep, any shells that came pre-installed, etc. A binary is just a compiled executable that can be run on your system. Note that /bin is a generic directory for binaries of any sort. On certain systems, /bin might point to /usr/bin which is specifically meant for user binaries.

**/proc:** Proc stands for process. It contains the Process IDs (PIDs) of all running processes on your system. Utilities like top can query these files for information about running processes.

**/boot:** The boot directory contains configuration files for the boot loader, as well as the Linux kernel itself. The kernel will be called vmlinuz-linux. If you are using GRUB as your boot loader (likely), then grub.cfg should be present, which is responsible for determining how your system boots (although you should not go tampering with that file unless you know what you’re doing).

**/root:** The root directory is the home directory for the root user. It is often empty since the root user usually doesn’t need to have any files in their home directory for the most part.

**/dev:** The dev folder is quite special and possibly difficult to grasp at first. Essentially, dev contains device files. These devices can range from hard drives to peripherals such as your mouse and keyboard, to even virtual devices such as a random number generator. It’s worth noting that hard drives (including SSDs) often start with the prefix sda, whereas NVMe drives often start with the nvme prefix. We will cover this more later.

**/sbin:** The sbin directory should almost certainly exist on your Linux install. Similar to the bin directory, sbin also contains binaries, although the s stands for system, meaning that it contains system binaries. These are executables that are essential for booting, restoring, and/or repairing the system. Note that on my system, /sbin also happens to point to /usr/bin, meaning that I do not really have a dedicated directory for system binaries, since it’s shared with /bin.

**/etc:** The etc directory (pronounced et-see) is assumed to stand for et cetera. This directory contains configuration files for various applications that will be applied to all users on the system. Notable files within /etc include the \*tab files (primarily fstab which tells the system which drives to mount on boot), locale.conf (information about your preferred language and character encoding), passwd (a file containing users on the system), shadow (contains user’s encrypted passwords), skel (a directory containing skeleton examples of frequently used configuration files), mime.types (maps file extensions to default application launchers), and sudoers (grants permissions to users and groups).

**/srv:** The srv directory stands for services. Services in this directory generally refer to “web” services. For example, httpd resides here, which is the Apache http web server. Other than that, this directory is not used much.

**/home:** The home directory contains the home directories for each user that exists on the system (aside from the root user). For instance, if you had three users, Ted, Bob, and Alice, then three directories with matching names would exist in /home, one for each user. When you open a shell, you will be placed within your user directory by default.

**/sys:** The sys directory is short for system as you may have guessed. This directory contains subdirectories with system information such as power usage, temperature, battery life (if using a laptop), firmware, etc. This is useful for writing scripts that monitor some system resource.

**/lib:** The lib folder contains system libraries and kernel modules. Libraries in Linux are often dynamically linked libraries (DLL) aka. shared objects. Shared object files end with a .so file extension. A kernel module is similar to a shared object but is specifically used to extend the capabilities of the kernel. Note that on my system, /lib points to /usr/lib

**/tmp:** The tmp directory is used for temporary files. Everything within it should get cleared after a reboot of the system, so it is volatile.  
  
**/media:** The media folder may or may not be present on your distro. Its sole purpose is to act as a mount point for external media such as thumb/flash drives or external hard drives.

**/usr:** The usr directory is for files that would be convenient for users. It contains a similar file structure to that of root. It contains its own version of bin, sbin, and lib which may or may not be pointed to by /bin, /sbin, and /lib (as is the case on my system). It also contains a local/share/ directory which is for files that are ideally meant to be shared between users such as font files. The history of how usr has been used is quite messy, and varies based on the distro that you’re using, so I recommend doing a bit of research for a more detailed explanation.  
  
**/mnt:** mnt is your mount directory. This is where you would create mount points for devices such as hard drives. A mount point is just a directory that acts as the root directory for a file system. For example, I could create a directory in /mnt called /mnt/HHD which would act as the root directory for my hard drive. Mounting my hard drive to /mnt/HHD would mean that all of the files on the drive would branch out from under the mount point.

**/var:** The var directory stands for “variable”. This directory is used for files that change often. Primarily this includes your spool directory which houses files that are queued for some task. For example, your print spool might contain tasks queued for your printer. The var directory also contains cache files under /var/cache for programs like your package manager.

**/opt:** Finally, the opt directory stands for optional. This is usually where you would place binaries that are not added to any of the bin directories by the system. You can manually place binaries here if you want to be able to run them in the command line from any directory.

Navigating the Directory Tree:

Now that you have an idea of what sorts of directories and files exist on your system, it is imperative that we learn how to navigate between directories. On Windows and MacOS, you might be used to clicking on folders in a file manager. This is possible on Linux as well, but we will be focusing our efforts on the command line. Firstly, we can enter the command ‘ls’ which stands for ‘list’ to list out the files and folders in our current directory. Another useful command is ‘pwd’ which stands for present working directory. Entering pwd into your shell will print out the absolute path leading to your current directory. I suppose I’ll have to explain the differences between an absolute path and a relative path while I’m at it. An absolute path means the “full path” i.e. starting from the root directory. For example, /home/user/Downloads would be an absolute path. A relative path is a path that is relative to your current location in the directory tree. For instance, if you were already in /home/user/, then Downloads/ would be a relative path.

**Note:** Sometimes you will see me enter a directory with a trailing slash at the end e.g. dir/. The trailing slash is optional and denotes that I’m referring to a directory. Failing to add the trailing slash has no functional difference.

pwd will always print the absolute path, which is usually more helpful than a relative path. With the ls command and pwd command, we can see which directories exist above and below us in the directory tree. In order to actually change directories, we use the ‘cd’ command (cd stands for change directory). cd can take either a relative or absolute path as an argument. For example, cd /etc/ would place me in the /etc/ directory. Or, assuming that my present working directory is /home/user/, I could do “cd Downloads/” to go into my Downloads folder.

cd gets a bit more confusing but bear with me. Your present working directory can be represented as a single period ‘.’. Your parent directory (i.e., the directory above you) can be represented as two periods “..”. Thus, in order to go up a directory, we can do “cd ..”. If we entered “cd .” this would actually do nothing because we’d just change directories to the directory that we were already in. Also, if you do not specify a path to change directories to, the cd utility will assume that you want to go to your home folder. In other words, “cd” by itself brings you to /home/user/. Another nuance that not many people know is that “cd –“ will change directories to the previous location that you were in. For instance, if I did cd /some/path/to/some/file/ and then I did cd by itself to return to my home directory, but then decided that I wanted to go back to /some/path/to/some/file/, instead of typing out the long cd command, I could just do “cd -“ and I’d be taken back to /some/path/to/some/file/. That’s mostly it for cd. I recommend playing with absolute paths, relative paths, and some of the other pseudo paths (., .., -) if you’re not comfortable with the material thus far.  
  
Creating and Removing Files/Directories:

Now that we’re familiar with navigating through the directory tree using cd, we can begin to create and remove files or directories. Let’s start with a file. The simplest way to create a file is to run the command “touch filename”. If you read the man page for touch (run “man touch”) you’ll see that touch is actually used to update file timestamps. If a file does not exist, touch will create it first, and update its timestamp. Linux does not use file extensions like Windows or Mac. We’ll discuss why this is later. That’s not to say you can’t use file extensions, it’s just to say that you won’t really see them as often. Anyways, in order to create a directory, we use the command mkdir (which means “make directory”). For example, “mkdir dir” will create a directory called dir. Once created, we can now cd into dir if we desired.

Removing files and directories becomes a bit more confusing, but nothing crazy. To remove a file, we simply use “rm filename”. This will delete the file, no questions asked, and no trash can. Linux assumes that you are not stupid, and are making very intentional decisions, so exert caution when moving or removing directories and files. There does happen to be an -i flag that will prompt us with a confirmation message if you’re clumsy (e.g., rm -i filename). In order to remove directories, we have to use a different command (sort of). If it’s an empty directory, we can use “rmdir dir” which is pretty self-explanatory, or we can opt to use rm -d which does the same thing. For whatever reason though, we can’t use rmdir or rm -d on directories that are not empty. Instead, we must use the rm -r which stands for “remove recursively”. This will remove the directory, as well as all its contents (including sub-folders).

To summarize, touch creates a file, mkdir creates a directory, rm by itself deletes a file, rmdir or rm -d removes an empty directory, and rm -r removes a directory recursively (if it is not empty). As with all commands, I recommend that you read the man pages for each of these so that you can see which flags are available to you that I left out.

Moving and Copying Files/Directories:

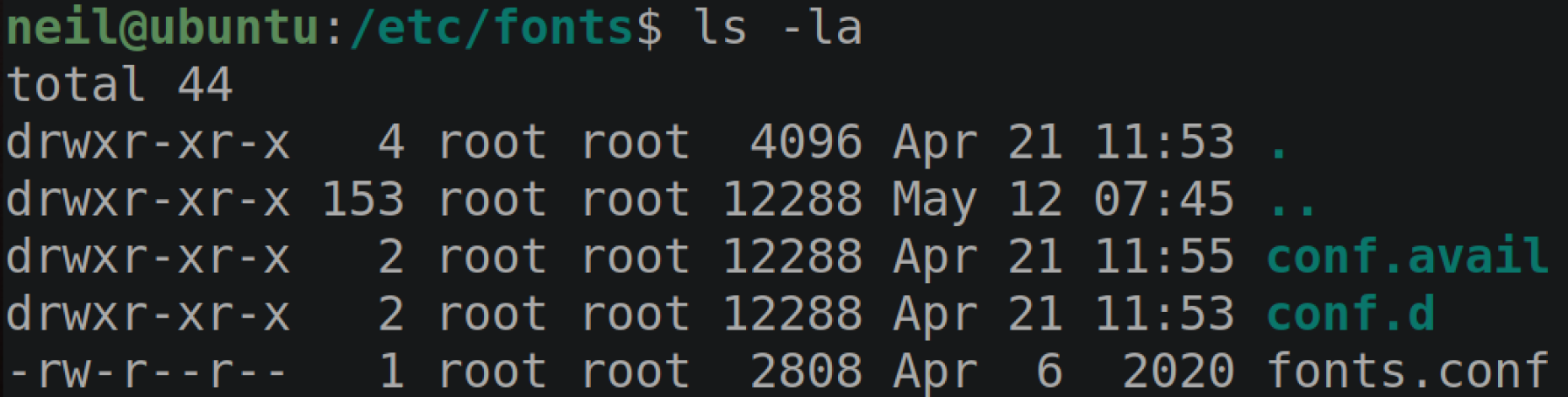
Something that we need to be able to do very often is both moving and copying files. In order to move files, we use a command called mv, and in order to copy them, we use cp. The mv command will move files or directories from a source location to a destination location e.g., “mv filename dest/”. You must be careful with mv because it will clobber existing files that share the same name. For example, if there is a file in dest/ called foo, and we move another file named foo into dest/, then the foo that we’re moving will overwrite the foo which already exists in dest/. Similar to rm, there is an -i option which will prompt you to confirm beforehand. There is also an -n option which means “no clobber”, and additionally a -b option which will create a backup file before moving it. In order to move multiple files into a directory, we can use the -t option (target) to specify the target directory, and then provide multiple files separated by whitespace. For example, in order to move foo, bar, and baz into dir/, we do “mv -t dir/ foo bar baz”.

When we use mv, under the hood, it is actually copying our file into the destination location, and then deleting the source file. We can exploit this fact to rename files using mv. In order to rename a file named foo to a file named bar, simply do “mv foo bar”. This will copy the file foo into the current directory as a file named bar, and then delete foo, which has the effect of changing the files name.

For copying, as I mentioned, we use cp. Like mv, cp will clobber files/directories with the same name. For copying directories which are not empty, we must use the -r (recursive) flag, similar to rm. cp shares many of the same flags as rm and mv, such as the -t, -b, and -i flags. Of course, you should always read the man pages to check what flags a command has, since I’ve omitted many here for the sake of brevity.

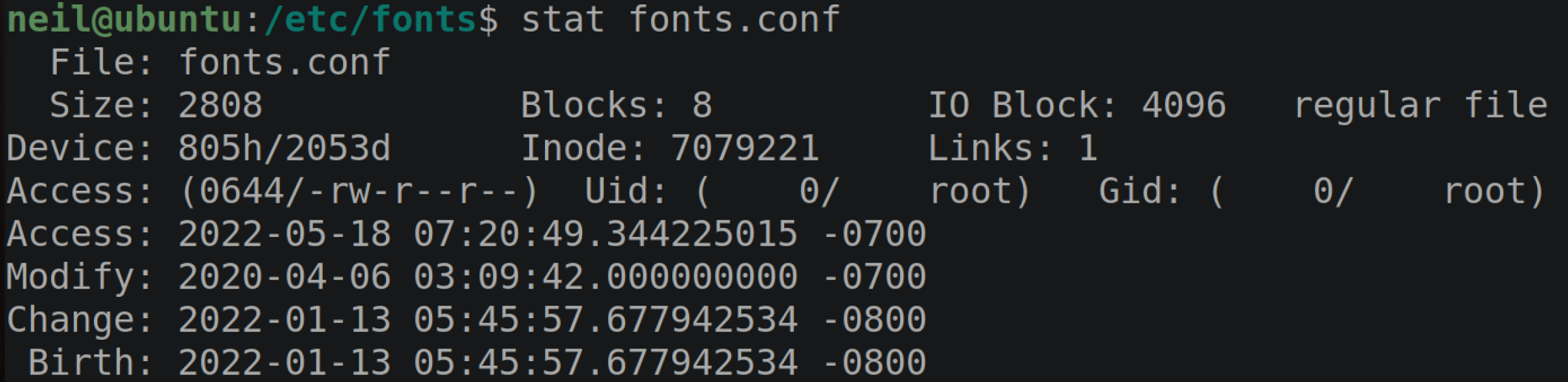
Getting File Information:

Sometimes it is nice if we can get information about a file. For example, what type of file it is, when it was last modified, its permissions, its size, etc. There are three commands that we’ll look at that can help us with this (one of which, we’ve already covered).

The first command that we’ll look at is ‘file’. Running “file filename” will display some basic information about filename. This is not really useful for text files or directories, since it will probably just tell you that they’re a text file or directory. The file command is most useful for binaries or shared object files since it will tell you what sort of binary it is, how it’s linked, what architecture it was compiled for, etc.   
  
 The second command, which we’ve already looked at is ‘ls’. The ls command has two flags that are often used in conjunction. These flags are -l and -a. The -l flag means “long list format”, and the -a flag means “all”. When it comes to using single letter flags from the command line, we can usually merge them together. Instead of “ls -l -a”, we can just do “ls -la”. Both are equivalent. So, “ls -la” will list all files (this means list hidden files and directories as well), and format them so that we can get more information about the file. Here is the output of doing ls -la in my /etc/fonts/ directory:  
  
  
  
Let’s highlight a few things. The first thing that you’ll notice is that two directories appear which are not normally shown with ls. These are the ./ and ../ directories. If you recall, these represent the present working directory and parent directory. In Linux, hidden files begin with a period. In your home directory for example, you might have a .config or .ssh folder which is not normally visible unless we use the -a option with ls to list hidden folders. Thus, the ./ directory is actually a hidden directory with no name, and ../ is a hidden directory called ‘.’. This is why we’re able to cd into them. In my shell (zsh), I have color highlighting turned on. This is usually default on shells like bash. If I didn’t have this, everything would just appear as white text. Because I have color highlighting, it is somewhat easy to piece together that the green files are directories, and the white file is a regular file. But, if we didn’t know this, we could look at the left-hand side of the output of ls -la. Notice that each row starts with the character ‘d’, except for fonts.conf. The first character of each row specifies the *type* of the file. ‘d’ stands for directory. Regular files are considered not to have a *type* which is why fonts.conf has an underscore where the ‘d’ is for the other directories. We’ll cover the different file types later. Next, we have a bunch of characters ‘r’, ‘w’, and ‘x’. These determine the file’s permissions. ‘r’ stands for read, ‘w’ stands for write, and ‘x’ stands for execute. This actually get’s quite confusing, so bear with me. Each file has 3 octal values split into 3 bits. If you’re not familiar with octal, it is sort of like hex (base 16) or binary (base 2), but the base is 8. So, counting in octal goes 001, 010, 011, 100, 101, 110, 111 and then wraps back to 0. The ‘r’, ‘w’, and ‘x’ letters represent one bit in an octal number. Therefore, rw- would be equal to 110, or r-x would be equal to 101. Since there are 3 octal numbers per-file, and 3 bits per octal number, there are 3x3 = 9 permission bits per-file. The first octal number represents the permissions of the file owner. The second octal number represents the permissions of the group owner. Finally, the third octal number represents the permissions of users who are not the file owner or belong to the group owner’s group (we’ll discuss what groups are soon).

Let’s just look at the permissions for fonts.conf as an example. We see that the permission bits are rw-r--r--. The first octal number is rw- (110). This means that the file owner is capable of reading the file, writing to the file, but not executing the file. The second octal number is r-- (100). This means that anyone who belongs to the group owner’s group can read the file, but not modify it (write to it) or execute it. The final octal number is also r-- (100), meaning that anyone who is not the file owner and does not belong to the group owner’s group can only read from the file. Hopefully that made a bit of sense. We’ll review this when we discuss changing file permissions.  
  
Let’s look at some more of the information given to us. After the permission bits, we have some numbers (4, 153, 2, 2, 1). These are not very important to us, but they are the number of symbolic links that point to the file. We won’t concern ourselves with the details of what that means for now. Next is the file owner, followed by the group owner. The screenshot that I took is a bit misleading here. The file owner is the super/root user. You might assume that the group owner is the same as the root user, but this would not technically be correct. You see, root is the name of a user on your computer, but it is also the name of a user group. You might be part of the root user group, but that would not be the same thing as saying you are the root user. In other words, root, the user, and root, the group, are different things which share commonalities, but are not the same. Regardless, the next bit of information is the file size. Unfortunately, this is in bytes, which is very hard to read. A common option for commands that list file sizes is -h which stands for “human readable”. Re-running the command as “ls -lah” will print the file sizes in terms of kilobytes or megabytes. The final bit of information (not including the file name) is the timestamp of the file. The timestamp marks the last time that the file was modified. As we discussed earlier, the timestamp can be updated with the touch command, and is used a lot for version control such as with Git or SVN.

The third command for finding file information is a utility called stat. Running “stat filename” will give us similar information to that of ls -la, but perhaps in a nicer format. Here is the output of doing a stat on fonts.conf:



First is the name of the file, then, the size in bytes. A new bit of information is the number of blocks. If you know a thing or two about solid-state drives, and memory mapping, you’ll know that on 64-bit architecture systems like to operate in blocks of 4096. This is highlighted under the IO Block entry, which tells us the optimal transfer block size. This is not actually the block size of the file that we’re operating with. fonts.conf happens to have a block size of 512, which, when multiplied with 8 gives us 4096 bytes. The reason that the file size is less than 4096 is because the actual bytes written to it are less than that of a full page (4096). In other words, Linux allocated 4096 bytes for our file, but we’ve only used 2808 of those. Anyways, moving on, we see that this is a regular file (which we already knew). Then we get to the device section. These are the major/minor device numbers of the file in hex (h) and decimal (d). This is not too important, but essentially, 8 is the major device number, and 5 is the minor device number for this file. The next section is important. This is the inode number of the file. In Linux, each file has a unique inode number assigned to it which acts as a unique identifier. These are assigned sequentially. Whenever a file is deleted, the next file that is created will take it’s inode number, assuming that it is the lowest available inode on the system. The next section shows the number of links to fonts.conf, which we already seen in the output of ls -la. In the next section, we get a bit of a cleaner output of the access/permission bits. The UID and GID stand for user ID and group ID. These are the ID’s that represent the file owner and file group owner. For the root user and root group, the user ID and group ID is always 0. Finally, we get an overview of the timestamps at which the file was last accessed, modified, changed, and when the file was created. You can get additional information from the stat command using the --printf= option in the man page.

Opening a File for Reading/Writing:

Unfortunately, opening a file is a bit harder for me to teach you because this heavily depends on what distro you’re using and what you’re comfortable with. Most distros that have a DE will come with a text editor such as gedit. Since you’re (presumably) a noob with Linux, a text editor is a quick and familiar way to edit text. A nice graphical text editor which you can install is sublime text. However, most Linux veterans who work from the command line will tell you that the first thing you ought to learn how to use is vim. Vim stands for “vi improved”. Vi was the predecessor to vim but is fairly outdated by todays standards. Vi stood for “visual editor”. So, I guess you could say that vim really stands for “visual editor improved”, but vim rolls off the tongue a little nicer. Vim does have a learning curve, and can be quite daunting, but given some practice, is extremely powerful. You can enter the command “vimtutor” into your terminal and an interactive tutorial will appear on your screen to teach you the basics of using vim. An alternative to vim is a program called nano, which is used by Visual Studio Code, but I don’t recommend nano since vim is objectively superior. There is also emacs which is the text editor created by Richard Stallman. I also don’t recommend this to beginners because it would essentially need its own document to explain. Pick one and stick with it for a while before switching to another. The tools available to you are only as powerful as how well you know them.

Linux File Types:

During the File Information section, I briefly mentioned that Linux has certain file types. There is a saying in Linux that goes “everything in Linux is a file”. And that is quite literally true, whether it be directories, programs, devices, resource managers, etc. Here, we’ll discuss each type of file in detail.  
  
**Directory:** The directory file type in Linux is marked as ‘d’ when doing an ls -la. A directory in Linux is just a normal file that contains links to other files. If you try to open a directory in a text editor such as vim, it will list out the contents of the directory, because vim can detect which files the directory is pointing to.   
  
**Character:** Character devices are files which represent devices connected via a serial bus. Since serial data is often sent in 8-bit chunks, we call these devices character devices. Character devices include anything connected via USB, since USB is a serial connection. Other character devices include your TTYs (TeleTypewriter devices), audio devices such as MIDI controllers, VGA monitors, etc. A TTY is essentially a terminal session, in case you were wondering. Character devices will appear as the letter ‘c’ in an ls -la.   
  
**Block:** Block devices are devices which are usually paged, or split into sectors such as RAM, a hard drive, solid state drive, or NVMe drive. Block devices appear as the letter ‘b’ in an ls -la. There is a command for listing out block devices, called lsblk (which stands for list block devices). This can be given the -h (human readable) option and will output all block devices and the location at which they’re mounted (if they are mounted). Usually, hard drives and solid-state drives will start with sd followed by a letter from a-z e.g. sda, sdb, sdc, etc. The letter tells you which drive it is. Letters are assigned in the order that the block devices are mounted to the file system. There will also be at least one sub-entry per-drive which represents the partitions of the drive. So for example, sda1 is the first partition on the sda drive.

**Socket:** Unix Domain Sockets (UDS) are files on Linux used for Inter-Process Communication (IPC). IPC is a useful concept in programming for communications between two processes. It works identically to network sockets, but instead of operating between two devices over a network, it operates between two applications virtually. A very basic explanation would be that one process writes to the file, then, the other process receives the information, processes, and responds through the same file. Unix Domain Sockets are represented as an ‘s’ in an ls -la.

**Pipes:** A Pipe is very similar to a UDS. The primary difference between them is that pipes do not allow for inter/intra-network communications. The most obvious way that we use pipes in Linux is with the pipe character ‘|’ which allows us to take the output of one command and feed it as an input into another command. These are displayed as the letter ‘p’ in an ls -la.

**Symbolic Links:** Probably the most important file type to understand out of the bunch are symbolic link files, or symlinks for short. There are two types of symlinks, soft, and hard. A soft symlink is basically a pointer to another file. This works very similarly to creating a shortcut on Windows. When you “open” the shortcut, it’s really opening the main application which is usually located in some other directory within the directory tree. A soft symlink is created with the command “ln -s targetfile symlinkname”. A hard symlink on the other hand is an exact duplicate of the original file. It’s not a copy, because a copy can be modified without affecting the other file. Hard links share the same inode number, meaning that modifying one file directly affects the other. These are also created with the ln command, but without the -s option e.g. “ln targetname symlinkname”.

MIME Types:

As mentioned earlier, files in Linux don’t require extensions. For certain programs, knowing the file extension can be useful. For example, vim checks the file extension for hints as to which syntax highlighting to load. Extensions are never required though. MIME stands for Multipurpose Internet Mail Extensions because they were originally created for email. Nowadays, they’ve expanded to more than just email. The file /etc/mime.types maps common MIME types to one or more file extensions. There is a command called mimetype which will list the MIME type of a given file. For instance, running mimetype foo.jpg reveals (on my system) that the associated mime type is image/jpeg. Another command called xdg-mime allows us to find out which application is associated with a given mime type. So for instance, if we wanted to know the default program that gets run when the image/jpeg mime type is encountered, we can run “xdg-mime query default image/jpeg”. On my system, it is revealed that sxiv.desktop is responsible for opening images by default (sxiv is the image viewer I use). In order to automatically open a file with an unknown mime type with its default application, we can use xdg-open. For example, running xdg-open foo.jpg opens foo.jpg with sxiv, since that is the default application for the image/jpeg mime type as we seen previously. On a final note, we can also use the xdg-mime command which we used to query to set a new default application for the associated mime type. For instance, if I wanted to change the default image viewer from sxiv to imv, we could do “xdg-mime default imv.desktop image/jpeg”.

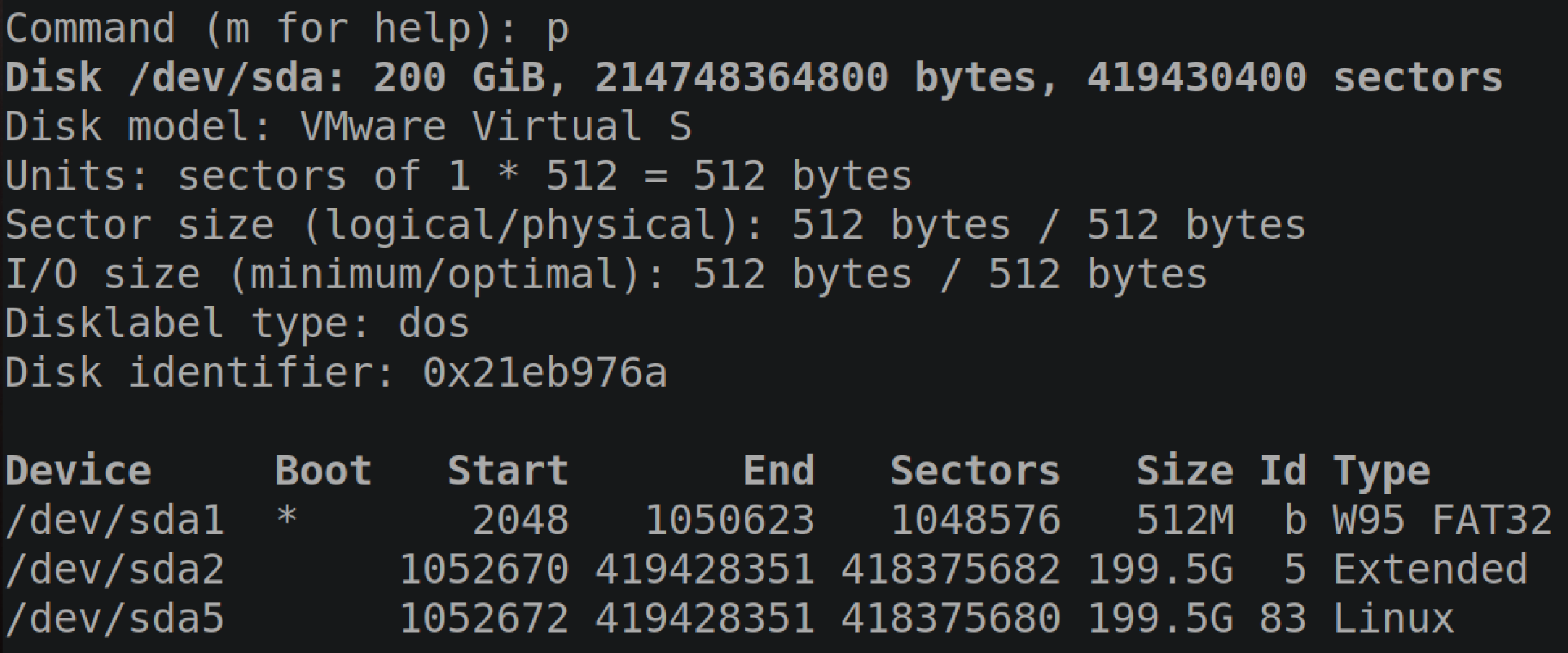
Changing File Permissions/Ownership:

The Extended File System:

If you’re familiar with Windows, you’ve probably heard of NTFS (New Technology File System), and FAT (the File Allocation Table file system). There’s nothing actually new about NTFS anymore, Windows has always branded itself as “NT”. Anyhoo, NTFS is the standard used by Windows partitions for storing data. Similarly, FAT is a more universal file system used all the way back on floppy disks with the original FAT file system. Nowadays, we have FAT16 and FAT32, exFAT, etc. These file systems can be used on Linux, but the primary file system currently used is called ext4, which stands for extended file system 4. ext4 is the successor to ext2 and ext3. ext4 is a copy-on-write file system, meaning that blocks are only allocated at the time that they are written to. The filesystem is broken in to 128MB blocks, which are further subdivided into block groups of 4096 bytes (doing the math shows that there are 32,768 block groups per-block). Data is written in little-endian format, meaning that if you do a hex dump on any file in Linux that uses an ext4 file system, you can expect that the byte order will be little endian (low order byte first).

Partitioning and Mounting a Drive:

There are two methods of partitioning a drive that I recommend. The first is a graphical option if you prefer to use a GUI. It is a program called gparted. The second option, which is a Command Line Interface (CLI) utility is fdisk (which stands for format disk). I will only be covering fdisk here, since gparted is pretty straightforward to use. fdisk will require that you run it with sudo privileges for obvious reasons. Run “sudo fdisk <drive>” where drive is the path to your block device. Block devices will always exist under the /dev/ directory. Before running fdisk, you can run lsblk to list out all block devices on your system and see which one you need to format. Once you know the name of the drive e.g. sdb, run “sudo fdisk /dev/sdb”. Now we must be careful not to play around. None of your changes will be applied until you press ‘w’ to write them, so if at any point you mess up, either press ‘q’ to quit, or fix it by redoing whichever step you messed up on. First thing we want to do is press ‘p’ to print out information about our disk. Here is the information of my /dev/sda on a Ubuntu VM:



As you can see, a good bit of information is displayed. We can see at the top, that this is indeed the correct drive (/dev/sda), and if you weren’t sure, it also shows how many GB it stores. We can also see the disk model which is another useful identifier to make sure you have the correct drive. In my case it’s a virtual hard drive created by VMWare. We can see some more information about sectors, such as how many units per sector (in this case 1), how big each sector is (sector size, logical and physical), and the minimal and optimal transfer sizes. The disklabel type depends on whether the drive uses Master Boot Record (MBR) which is why you see dos (Disk Operating System), or GUID Partitioning Table (GPT) which would show up as gpt in the disklabel type field. The identifier is given by the manufacturer. In the lower section, we can see each partition, its start offset, end location, number of sectors, total size, ID, and file system. In the case of /dev/sda1, it’s marked as being the boot sector (the bootloader e.g. GRUB will load this into memory and jump to offset 2048 to begin execution). Conventionally on MBR formatted drives, the boot sector would be the first sector of the disk and would be 512MB in size. /dev/sda5 is my actual Linux drive which contains the ext4 file system and all my files starting at root. /dev/sda2 is an extended partition which contains logical drives. This only occupies 200GB of virtual storage, not real storage. Pressing ‘l’ will list out all of the partition types, of which there are many. Instead of partitioning the drive in fdisk, we’ll do that afterwards using a separate command. For now we’re only going to format the drive (wipe it). To do that, press ‘d’ to delete. This will ask you which partition you want to delete. Just hit enter to choose the first option. Do this until there are no remaining partitions. Then, press ‘n’ to create a new partition. It will ask for information such as the partition number, whether it’s an extended or logical partition, offset, size, etc. You can hit enter for most of these, except size. The size can be entered in bytes, kilobytes with K e.g. 10K, megabytes with M e.g. 10M, or gigabytes with G e.g. 10G. Do this for as many partitions as necessary. Once you’re done, press ‘w’ to write the changes, followed by ‘q’ to quit.

Now we have our partitions, but they need a file system. To do this, we can use a command called mkfs. mkfs allows us to specify the type of file system with the -t option. For example, in order to create an ext4 filesystem, just do “mkfs -t ext4 <partition>” where partition is the absolute path to the partition device e.g. “mkfs -t ext4 /dev/sdb3”. Unfortunately, mkfs only creates ext file systems. In order to create something like a fat partition, we have to use a different command called mkfs.fat and give it the -F option along with the size e.g. “mkfs.fat -F32 /dev/sdb3”.

Finally, once your partition has a file system, we must mount it so that it is actually accessible from the directory tree. To do this, we use the mount command like so: mount <drive> <location>. Location can be any directory in the Linux partition (the file tree). We typically create a directory in /mnt and then mount the drive there. For instance, I could do “sudo mkdir /mnt/HDD” followed by mount /dev/sda1 /mnt/HDD” which would mount /dev/sda1 to /mnt/HDD. In other words, /mnt/HDD is the root directory for the sda1 partition. In order to unmount a drive (which you should do before removing it), we use the umount command like so: umount <location>. In our case, “umount /mnt/HDD” will unmount the partition currently mounted at /mnt/HDD/.

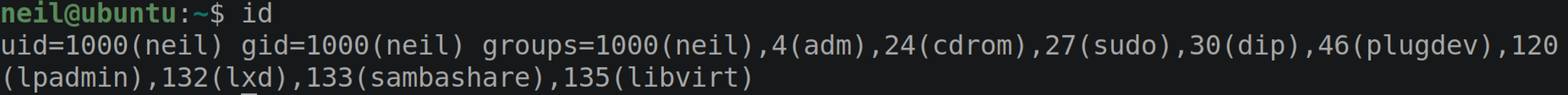
I’m aware that this all seems extremely confusing, and it certainly can be at first, but you will eventually remember the steps the more you have to partition drives and mount them to the file system. Practice makes perfect as they say.

Users and User Groups:

We can now take a break from looking at files and begin to discuss users and user groups. Users and groups actually function very similarly in Windows and MacOS as they do in Linux, though they are abstracted away much more on those operating systems, which is sad because users and groups are an important feature of an OS! If you haven’t clued in yet, I will explain clearly. Linux bases permissions off two things: Users, and user groups. You should be familiar with a user on your system, since you must login to your user account every time you boot up your PC. Entering your password is known as *authorization*. Once logged in you are *authenticated.* This means that the system has created a session which is based off your user. This determines what shell you use, where your home directory is, and most importantly, which files you’re able to access.

Not only do we have users, but we also have user groups. Groups are a way of sharing permissions between users who are interested in the same thing. For example, you may have a user group dedicated to audio management. You may have multiple users who are interested in controlling the system’s audio, thus, we can allow all members of the audio group to share equal permissions when it comes to accessing files related to audio.

As we’ve seen before, users are identified with a user ID (UID) and groups are identified with a group ID (GID). Every user that is created on the system will have a corresponding group. For example, if a user named John is created, then a John group will also be created simultaneously. By default, the user group associated with your user will be set as your “primary group”. I forgot to mention that every user has a primary group that they belong to. Other groups to which the user belongs to are secondary. In order to see your UID, the GID of your primary group, and the GIDs of all secondary groups to which you belong, we can simply enter the command ‘id’. On my Ubuntu VM, this looks like the following:



As you can see, my UID is 1000 (my username is neil), my GID is 1000 (my primary group name is also neil), and then I belong to 9 other secondary groups: adm, cdrom, sudo, dip, plugdev, lpadmin, lxd, sambashare, and libvirt. Each of the GIDs of these groups are also listed. In order to only list the groups that you are a part of, you can use the command ‘groups’, which only displays group names.

The Root/Super User:

We’ve talked a lot about the root user or super user, but I’ve never actually explained what that really is… On every Linux system, the first user to be created is the root/super user (I will refer to this user as just root user from here on out). This is why UID 0 and GID 0 will always be root on any Linux system. The root user has full permissions over everything. Obviously, becoming the root user is a massive attack vector for hackers, and there have been exploits over the years where hackers have been able to gain super user privileges. Although we are capable of logging in as the root user, it is recommended that you login as your own user, and then request temporary sudo privileges. There are a few ways that we can request super user privileges. The most common way, and one that we’ve seen a few times now, is to use the ‘sudo’ command. This command stands for “super user do”. A prerequisite for being able to use sudo is that you must belong to the sudo user group. On older systems, rather than the sudo group, you’d usually need to be part of the ‘wheel’ group. Regardless, once a part of this group, you are then allowed to use the sudo command which temporarily grants access to running commands as the root user. The Password Authentication Module (PAM) is usually in charge of handling the time interval between requesting you for your password. For instance, on you system, PAM may request that you enter your password every 5 minutes in order to keep running commands with root privileges. This can be configured, although I don’t recommend doing so unless you know really know what you’re doing since making mistakes can easily lock you out of your computer (I’m speaking from experience).

Switching Users:

We can temporarily switch users on Linux using the command ‘su’ (switch user). When we do this, we are actually creating a new user session, even though it doesn’t look like anything has changed. What this means is that we are now authenticated as the user we logged in as. In order to exit out of the user session, we just enter the command ‘exit’, and we are brought back to our original user session. Let us assume that a user named admin existed on my computer. If I wanted to login as admin, I would need to enter the command “su admin”, which would then request the password associated with admin’s account. Assuming I know admin’s password, I should be able to log in. In order to go back to my own user session, I just type exit. In order to switch to the root user, the command is a bit different. You’d do “su -". You’ll know that this worked if your nametag (called your PS1) changes from a dollar sign ($) to an octothorp (#). Generally speaking, it is convention that $ is for regular users and # is for the root user. Once again, exit should remove you from the root session.

Adding Users:

Modifying Users:

Adding Groups:

Modifying Groups:

Kernel Space, User Land, and Ring Modes:

Linux is broken into 2 primary address spaces, the smaller of which is allocated towards kernel space, and the larger of which is allocated towards user space or user land. We separate address spaces for security reasons. If a user attempts to touch an address that belongs to the kernel, we can send an IRQ and stop the write from occurring. Kernel space is where kernel code, system calls and hardware interrupts are managed. User space is where your applications live. As we discussed earlier, the init system is what kickstarts the process tree for user space.

The x86 architecture has 4 layers called rings, which determine system privileges. Kernel space runs in ring 0, whereas user land runs in ring 3. Two intermediary rings exist (1 and 2) that are intended for device drivers, but are not utilized by Linux since device drivers live within the kernel on Linux. Rings gradually lose privileges as they increase, meaning that ring 0 has the most privileges and ring 3 has the least. Arm architecture uses a similar concept, but labels its layers EL0, EL1, EL2, and EL3.

Tar Balls and Compression With tar/gzip:

Linux supports various archiving and compression algorithms/formats such as 7zip, zip, and gzip. Gzip is of course, the GNU zip format. It is actually used for compression, not archiving. It uses a compession algorithm created in the early days of computers known as Lempel-Ziv coding (LZ77). This format is a lossless algorithm, meaning that no data is lost when compressing or decompressing, unlike a lossless algorithm such as jpeg, for example. Typically files which are compressed using gzip should have the .gz extension for readability. You can specify the level of compression using options -1 to -9. The gunzip command will decompress a file compressed using gzip. There is one glaring issue with gzip, which is that it can only compress one file at a time. In order to compress multiple files and output one compressed archive, we first need to create the archive using an archiving utility such as tar. The tar (tape archive) utility can create an archive of files which typically end with the .tar extension. A tar archive is called a tar ball. Often times, you’ll see tar being used with the -czvf options. The acronym that people have invented for these options is “**T**ar **C**an **Z**ip **V**ery **F**ast”. To give an overview of what each does, -c creates the archive, -z zips it, -v is for verbosity, and -f is the files you’re adding. One thing that throws people off with that is that the name of the archive must come before the files that you want to add to the archive. To archive an entire directory, you can do: tar -czvf archive.tar directory/\*. You may also use the –exclude option to exclude certain files from the tar ball. These two commands are often used in conjunction with each other, which is why you will tend to see files ending with the .tar.gz extension on Linux. In order to open these files, you must first decompress the tar ball using gunzip, followed by tar -xvf. The only new option here is -x which means exctract.

As an aside, there exist other compression algorithms such as bzip (.bz2) which you can decompress (bunzip). Other less common algorithms include LZMA, XZ, LZ4, LZO, etc.

Listing Devices:

There are many variants of the ls command for listing information about various devices on your system. For instance the lsusb command, which will display all of your active USB devices, the USB bus that they’re connected to, and their vendor ID. The lspci command will display all PCI peripherals connected to the PCI busses. This command isn’t inherently very useful, but is good if you’re having issues with a PCI device and want to see if it’s being seen by the OS or not. The lsblk command lists block devices i.e. hard drives that are mounted to the filesystem, as well as their partitions. The lsmod command displays kernel modules which are currently loaded (more on kernel modules later).

I/O Streams:

Linux has 3 virtual devices in the /dev directory called stdin, stdout, and stderr. These virtual devices are known as I/O streams. The stdin stream A.K.A. standard input, is the stream which we send our commands to when we type them into the terminal emulator. The stdout (standard out) I/O stream is where normal output is directed to. The echo command, for example, outputs to stdout. Finally, stderr (standard error) works similar to stdout, but is specifically designed for outputting error messages. I/O streams may be implemented differently depending upon the hardware of your system or the distribution of Linux that you’re using. On most machines, these I/O streams will be symbolic links to /proc, which is what’s known as a resource manager (more on resource managers later). If you’ve developed in C, you may know stdin, stdout, and stderr as file descriptors, because that’s what they are. The stdin, stdout, and stderr reserve file descriptors 0, 1, and 2, respectively. You can duplicate these descriptors, but you cannot close them.

The Linux redirection operator is used to redirect I/O. For example, we can redirect stdout to a file like so:

echo “some text” > file.txt

Now file.txt will contain the text “some text”. Note that a single triangle bracket will overwrite existing data. If instead, you want to append the string, use a double redirection operator like so:

echo “more text” >> file.txt

Sometimes commands will output error messages to stderr. Using the redirection operator like in the previous examples to write this output to a file will not work since the previous examples redirect stdout, not stderr. In order to redirect only stderr, we can do the following:

ls fake\_file 2> file.txt

Assuming fake\_file does not exist, the error messages from the ls command will be redirected to file.txt. Alternatively, you can utilize the /dev/null virtual device to redirect all output. For example:

command > /dev/null 2&>1

The way I’ve shown you how to redirect both stdout and stderr is one of many ways, but it is POSIX compliant, meaning it’s the most portable way of doing things. Usually though, this can be abbreviated to something like:

command 2&> /dev/null

We haven’t yet looked into redirecting stdin. Similar to the stdout and stderr redirection operators, we simply reverse the direction of the operator to read from files. To read the line count of a file, we can do the following:

wc -l < file.txt

Of course, the input redirection isn’t really necessary in the previous example since the wc command accepts files anyways, but hopefully it drives the point across.

Signals:

A signal is essentially a lightweight message that can be sent to any running process. POSIX defines various signals which have unique semantic meanings. For instance, to resume a process, the SIGCONT signal can be sent. These semantic associations are merely suggestions, and are not strictly enforced. Processes are free to define the actions that they want to take when a signal is received, which includes ignoring them. The SIGKILL and SIGSTOP signals are exceptions to this rule, and cannot be so easily ignored. In Linux, shortcuts exist for these signals. Ctrl + C sends SIGKILL which force terminates a program and Ctrl + Z sends SIGSTOP which force stops a program (though it can be resumed). You may think that Ctrl + D is also a signal, but actually, this just sends an EOF character (\0).

Signals are sent using the kill command. This is a bit confusing because you’d think that a command called kill would necessarily terminate the program in question, but this is not guaranteed to be the case depending upon the actual signal that is sent. We use the -s flag to specify which signal to send followed by the PID of the program you want to send it to. You can also skip the -s flag if you know the number associated with the signal you want to send. For example, we can do kill -s SIGKILL or we can do kill -9, which is synonymous. In order to get the PID of a process, you can use the ps command, or you can use the pidof command, which is what I prefer. For example, to kill firefox, run kill -9 $(pidof firefox). Alternatively, you can use a TUI program such as htop which allows you to graphically search for a process by name and then choose the signal you want to send from a list of signals.

Shells, Subshells, and Processes on Linux:

A very important and often overlooked portion of Linux is understanding how processes work. Fundamentally, Linux is written in C, so we can expect it to behave like C. If you are familiar with C, you may know of fork(), a function used to copy a parent process into another process called the child process. This happens to be exactly how Linux operates. When the system initially boots, a program is executed, known as the process manager. The process manager is a process responsible for creating all other processes. Most modern Linux systems use a process manager called systemd. Again, if you’re familiar with C, you may know that each process has an identifier known as the process ID (PID). The process manager (systemd in out case) has PID 0. It will then call fork() to split into child processes, whose PIDs will continue to get larger the more processes are created on the system.

Another thing to consider is that many times in Linux when we run a command through a shell, the shell typically forks itself into a new process known as a subshell. The subshell executes the command behind the scenes and then may or may not return the results to the parent shell. There are actually multiple ways of creating subshells, each varying slightly from one another.

**Grouping (…):** We can create a subshell using just two closed brackets surrounding part of a command. This creates a subshell and will wait for it to terminate.

**Background &:** In order to create a subshell that runs in the background (known as daemons), we use the ampersand at the end of the command e.g. “echo ‘hello from subshell’ &”. Daemons will not wait to be terminated, meaning that they will hang until we kill the process manually or something else kills the process.  
  
**Pipe |:** We’ve covered pipes, but not in terms of subshells. Pipe is interesting because it will actually create two subshells, one for each side of the pipe, and will wait for both of them to terminate. As you know, pipe will take the output of the first command and enter it as input to the second command.

**Command Substitution $(…):** Command substitution creates a subshell with its output connected to a pipe. The parent process then collects the output of the pipe and uses it as input. This sounds confusing, but actually we’re just doing command substitution. This is most useful for when we want to set a variable to the output of a command e.g. DATE=$(date) will set DATE equal to the output of running the data command.

**Process Substitution <(…):** Process substitution creates a subshell with the output connected to a pipe. The parent process *or* any other process may open the pipe to communicate with the subshell. Using >(…) will do the same but using the subshells input rather than its output.

**Coprocess coproc …:** Using the coproc command will create a subshell and does not wait for it to terminate. The subshell’s input and output are connected to pipes with the parent connected to the other end of each pipe (basically <(…) and >(…) combined).

Text Manipulation:

ANSII Codes:

Scheduling Jobs Using chrony:

Debugging Processes:

There are a ton of useful Linux commands for debugging processes, especially for C developers. Here are a list of my favorites:

* **strings**: The strings command will print out all of the printable strings within any type of file. By default it considers any 4 consecutive printable characters as a string. This can be useful when trying to find strings that might be stored in the .data segment of an object file for example.
* **file**: The file command prints information about the type of file. For symlinks, it will display the file that is referenced by the symlink. For libraries it wiill display things like whether it’s a static or dynamic library, if the debug info has been stripped or not, whether it’s LSB or MSB, etc.
* ltrace: The ltrace command traces all of the dynamic library calls which which are called by the executed process, as well as any signals which the executed process receives. In order to use the ltrace command, you do ltrace ./prog, where prog is the program being executed. ltrace runs for the duration of the program’s execution.
* strace: The strace command is similar to ltrace but traces all of the system/kernel level calls made by the executed process. Running strace works the same way as ltrace.
* nm: The nm command lists symbols from an object file. It will also print a letter next to each symbol which specifies which segment of the ELF format that the symbol exists in. If the letter is lowercase, the symbol is local, but if it’s uppercase, the symbol is global/external (u, v, and w are special cases to this rule). You’ll need to refer to the man page for what each letter represents.
* objdump: The objdump command can dump various information about an object file. There are a ton of options which can be supplied to the command. A couple noteworthy ones are -d to disassemble into assembly code, -S to show the source code, -s to show the full contents of every section, -j to show a specific section, and -G and -W for stabs or dwarf debug info respectively (also -g will determine which is better suited between -G and -W for the current object file). I usually run the command with -dS and then use -j if I know what section I’m looking for e.g. objdump libfoo.so -dS -j .text | less.
* readelf: The readelf command is very similar to objdump. It provided a bit more abstract information, which is not always necessarily useful. Probably the most useful flag is the -h flag which displays information about the ELF header. The -s flag will print out each symbol defined within the ELF file, its type, bind (local or global), and its visibility (i.e. thread local, etc.). The -S flag displays an overview of each segment of the ELF file. I’ll just caution that the formatting is a bit strange, but it should make sense after looking at it long enough. Using the -a flag is equivallent to using all of the major flags combined.
* hexedit: The hexedit command allows you to view any file in a hex editor. The keybindings take a while to get used to, but once you get the hang of them this can be a powerful tool in your arsenal.
* valgrind: Valgrind is actually a suite of tools which are useful for instrumenting process execution. The memcheck tool is the most commonly used tool within the suite. You can run memcheck by doing valgrind --tool=memcheck ./prog. Valgrind will output the number of heap allocations and the number of blocks freed, which is how it determines if there was a leak or not.
* diff: The diff command simply displays the differences between two files. It should be noted that git also has its own diff subcommand, which is probably the preferred choice if you are using version control and want to compare the file with previous versions of itself.
* perf/netperf: The perf tool is used for performance analysis when running applications. The netperf command is similar but measures network bottlenecks and performance.
* gprof: Similar to perf, the grof command is an application profiler. The gprof command is more targetted towards C, Pascal, and Fortran and is a bit of an older command. The gprof profiler can display the execution times of each function within the call stack.
* lsof: The lsof command can display the file descriptors of file resources owned by a running process. The easiest way to achieve this is to use the -p flag which accepts the PID of the running process.
* ldd: The ldd command lists all of the libraries that an executable binary depends on to run. It will also display the location of the libraries on the user’s system and the memory address that they’re loaded into in RAM.

Agents:

A software agent is just a fancy word for a program which does some work on your behalf. A lot of helper commands in Linux refer to themselves as agents. When asked to start an agent, it’s often used as a listener which manages connections or temporarily keeps track of certain things.

Linux as a Server:

Basic Networking Tools:

Linux is king when it comes to managing networks and network tooling. Let’s look at some of the tools that make it so great! The most basic, but probably most useful is the ip command. The ip command, as you can likely guess, deals with displaying information about your IP address, but it can do a lot of other things too. The ip command on its own will just print out the usage information because you’re meant to execute one of the many subcommands. The primary subcommand that we care about is the address subcommand. The address subcommand can be aliased as just ‘addr’. Most of the ip subcommands actually have sub-sub-commands. Running ip addr or ip address will default to running ip addr show, which prints information about the interfaces found on your NIC. This will usually include your loopback (lo), ethernet ports (eth), and wireless LAN (wlan). It will list the link/MAC address, Ipv4/inet address, and ipv6/inet6 address. A lot of the other subcommands are a bit advanced, but we can do things like listen for network traffic, list the arp table, routing table, display network statistics, etc.

The next command which is great for network benchmarking is called netperf, meaning network performance. The netperf command depends upon another command called netserver to be running on localhost at a particular port. You can simply run netserver as a daemon and then run any of the predefined netperf tests using the -t flag. The predefined tests are documented in the man pages for netperf. You may also want to specify the format used for the throughput because it’s measured in bits by default. We can specify to use Kilo, Mega, or Giga bits or bytes. For example, we can test TCP upload/download speeds like so:

netperf -fM -t TCP\_STREAM

The netstat command is very useful when trying to see which ports applications are running on. Often times you’ll try to run an application on a specific port like 8080 only to find out that another app is already running on that port. In order to kill it, we use netstat to find whatever process is associated with port 8080. I pretty much always run netstat with the flags -tunlp, which I like to remember as “tunnel p”. To summarize quickly, -t is for TCP, -u is for UDP, -n is for numeric, as in numeric addresses, -l means find applications in the listening state, and -p means list the application name alongside the address, which is very helpful if you’re attempting to find the name of the process you want to kill.

OpenSSH and SSHD:

Secure Shell (SSH) is a protocol that has been around for a good while now. It uses Public Key Infrastructure (PKI) to allow for secure connections to another device. Using SSH, we can connect and login to remote devices such as servers as if they were our real PC. Due to the cryptographic nature of SSH, it can double as an authentication token. OpenSSH is an implementation of the SSH protocol that is most often used on Unix-like devices such as Linux. SSHD is the OpenSSH daemon, which must be running in order to be able to use the SSH protocol.

OpenSSH is a pretty complex beast, but let’s break it down to basics. We are able to generate any number of SSH key pairs. A key pair is a pair of cryptographic keys – one which will be public and have a .pub extension, and the other being private, which will have no extension. By default, keys are stored in /home/user/.ssh. The public key can be shared with anyone without risk of your information being compromised. This is because messages can only be decrypted if both the private and public key are present. We share our public key with the server we want to connect to, then it uses that public key to encrypt its messages back to us, which we’re able to decrypt using our locally stored private key. Even if an attacker receives our public key, without the private one it cannot decrypt the message. There are various cryptographic algorithms to choose from – the most notable being Rivest – Shamir – Adleman (RSA) and ECDSA. The longer the bit length used the harder it will be for attackers to brute force the private key. I would almost always recommend using a bit length of 4096.

Let us assume that we have a remote server such as a Virtual Private Server (VPS) that we want to remote into using SSH. We first generate a new ssh keypair using the ssh-keygen command. The -t flag can be used to specify your cryptographic algorithm, which can be one of dsa, ecdsa, ecdsa-sk, ed25519, ed25519-sk, or rsa. If you do not specify the -t flag, ssh-keygen will default to using ed25519, which is the most secure and is probably what I’d recommend. The rsa algorithm is also good and is more portable than ed25519. The dsa algorithm is deprecated and should not be used. The -b flag specifies the number of bits you want to use (likely 4096 if using something like rsa). The bit size is not applicable to ed25519 since it always uses 256 bits. The -f flag sets the base file name of the keypair. All in all, the command would like something like the following:

ssh-keygen -t rsa -b 4096 -f vps\_sshkey

When running this command, it will ask you to enter a passphrase for the ssh key. Though bad practice, I recommend just leaving this blank unless you want to have to enter a password everytime you use ssh. Anyhow, ssh-keygen should generate the files vps\_sshkey and vps\_sshkey.pub in /home/user/.ssh. In order to ssh into our vps, we use the ssh command. The -i flag specifies the identity file i.e. the private key that we want to use for the connection. We’d do something like the following:

ssh -i /home/user/.ssh/vps\_sshkey root@192.168.4.78

This would ssh into the server located at IP 192.168.4.78 as the root user, though you may specify a different user. A file ought to be created on the remote device under /home/user/.ssh called authorized\_keys (spelled exactly like that). This file shall contain the contents of the public key that was generated on your local device to avoid requiring the use of the -i flag everytime you ssh into your server. Another command which can automate the process of copying your public key to the authorized\_keys file is ssh-copy-id. This works similar to the ssh command, but instead of providing the -i flag with the private key, we give it the public key instead. For example:

ssh-copy-id -i /home/user/.ssh/vps\_sshkey.pub root@192.168.4.78

If you did decide to add a passphrase to your ssh keypair, you can use an ssh-agent to cache the password for a certain duration of time so that you don’t have to enter it over and over for the current session if making frequent connections. We run the ssh agent like so:

eval $(ssh-agent)

Then we add our private key to the cache like so:

ssh-add /home/user/.ssh/vps\_sshkey

This will prompt you to enter your passphrase and then it will temporarily cache this information for your session.

ssh sometimes gets a bit confused when you have more than one keypairs in your .ssh directory. To alleviate this issue, we can create a file in the .ssh directory called config. Within the ssh config file, we can add match clauses that ssh will use when making connections to determine which private key it ought to use. A basic entry into the config file might look something like the following:

Match host 192.168.4.78

IdentitiesOnly yes

IdentityFile ~/.ssh/vps\_sshkey

When running the ssh command to connect to our server, it will recognize the IP address or DNS hostname that we provide for host and use the identity file specified by the IdentityFile attribute. The host attribute in the match clause can match on multiple values delimited by comma.

UFW for Firewall Protection:

Nginx and Apache:

Certificates With Certbot:

Websites must have web certificates that are renewed every so often to prove their credibility. It used to be the case that a web certificate might cost around $100. This changed when the Internet Security Research Group (ISRG) created a non-profit certificate authority known as Let’s Encrypt. Let’s Encrypt is now the largest certificate authority that exists. A project called Certbot makes obtaining, deploying, and renewing Let’s Encrypt certificates an extremely easy procedure. Certbot works with Nginx, Apache, and web hosting services. The flags used will vary a bit depending upon which of these methods you’re using, but basic subcommands for certbot are the certonly and renew subcommands, which will either generate new certificates or renew existing ones respectively. Certificates are usually stored in /etc/ca-certificates.

How Mail Servers Work:

Did you know that it’s possible to host your own email? An email is nothing more than a server configured to use specific network protocols for mail transfer. Your email address is composed of a user and the hostname of the mail server. For instance, [bob@gmail.com](mailto:bob@gmail.com) refers to an actual user account that gets created in Google’s Gmail mail server. The primary protocol used for transfering and receiving electronic mail is the Simple Mail Transfer Protocol (SMTP). Two other protocols often come up when discussing mail servers, which are the Internet Message Access Protocol (IMAP) and the Post Office Protocol (POP). Both IMAP and POP allow you to access mail from your mail server from other clients. When you create directories in your mail client, they map to directories in your user’s home folder on the mail server which contain the actual email files. Normally, domain registrars allow you to setup DNS MX/email records to point towards a cname/sub domain.

Mail servers as a concept are not too difficult to grasp, however setting them up is, in reality, more complex. We require things like dovecot to manage remote mail access via IMAP or POP3, openDKIM for authenticating and validating emails using the DMARC protocol, spamassassin for blocking common spam addresses, and more. Luke Smith has a script called emailwiz which works on most Debian-based systems for automating essentially the entire process of setting up your own mail server, plus he has video tutorials on the matter.

Docker:

Advanced:

As a new user to Linux, I advise you to stop reading here. Hopefully now you have a solid foundation to begin using and learning Linux. I don’t recommend that you attempt to memorize all of the commands. You will remember the commands that are most useful the more that you continue to work from the command line. Once you are comfortable with the basics, come back to this document and read through it once more, including the upcoming sections where we will be discussing more advanced topics.

Resource Managers (procfs):

Swappiness:

You may know that modern computers are capable of allocating storage space on disk to be used as RAM if the system does not particularily have a lot of RAM. This is the opposite of RAM disks which is when you allocate space in RAM to be used as volatile (but very fast) storage space. The space that we allocate on disk to be used as RAM is called a swap partition. Here are some basic instructions on how to create a swap partition:

1. First we run sudo fallocate -l <size> <swap\_file>. fallocate is a utility used to allocate disk space for a file. The -l option stands for ‘length’ as in length of the file in bytes. The final argument is the file location/name to give the file.

2. Next we modify the permissions for the swap file e.g. sudo chmod 600 <swap\_file>

3. We must tell the kernel that the file is to be used as swap space. We do this using the mkswap command e.g. sudo mkswap <swap\_file>

4. We must enable the swap file. This is sort of like mount for regular partitions, but for swap. We use the swapon command like so: sudo swapon <swap\_file>

Running swapon --show will show us any swap storage on our system. Here is some more detailed information about how RAM is segmented on Linux:

- The lowest 16MB of RAM used to be used for Direct Memory Access (DMA).

- On 32-bit machines, there would be 32 bits used for memory addressing. 32 bits means 4 gigabytes of possible addresses, so the lower 32 bits of memory are sometimes called “below 4G”. The official name is DMA 32 as 32-bit computers could only do DMA in this zone of RAM.

- Once more on 32-bit machines, addresses above 896MB were reserved for kernel space (0xC0000000 to 0xF7FFFFFF). Anything below was considered user space (minus the space reserved for DMA). Addresses above kernel space are called high mem.

- Normal memory i.e. user memory is considered to be anything above 4G on 64-bit machines and anything between 16MB and 896MB on 32-bit machines.

Another thing to consider is that RAM (and disk partitions for that matter) are allocated in chunks called pages. The size of a page is determined by the kernel at boot time based on the architecture of the computer. Usually the page size will be 4K or 4096 bytes. You can check your page size by running getconf PAGESIZE.  
  
The memory zones that we looked at just a moment ago (DMA, DMA 32, Normal, and HighMem) are associated with one or more nodes. Each node corresponds to a CPU core. The kernel tries to allocate memory for a process running on a CPU core from the node that corresponds to that CPU core. This allows us to move the locations of these memory zones to different CPU cores if we wish to do so using something called Non-Uniform Memory Access (NUMA). Typically, however, your computer will have one node called node zero. We can check which nodes are associated with which memory zones using cat /proc/buddyinfo. The output of this command is frankly quite confusing to grasp. Essentially you might end up with something like this (looking only at the DMA32 memory zone):

Node 0, zone DMA32 6 4 6 5 5 5 9 8 ...

Each number represents the number of memory chunks with a particular size. The positions in which the numbers are arranged correspond to a multiplier of PAGESIZE calculated as 2^((2^position) \* PAGESIZE). The indices are little endian meaning that they increase as we go right. So for instance, the first number 6 means that there are 6 memory chunks in the DMA32 memory zone of size 2^(2^0 \* PAGESIZE). The second number means that there are 4 memory chunks in the DMA memory zone of size 2^(2^1 \* PAGESIZE) and so forth.

I’ve already mentioned the following concepts in my computer hardware and C notes, but it can’t hurt to repeat myself on important topics. We are able to map memory in various ways. Here is a list of each and how they work:

- File backed: These kind of memory mappings contain data that has been read from a file. It can be any type of file. Files are allocated on disk, which makes this type of backing good for swap files. If the system frees file backed memory and needs to obtain it once more, it can read from the file again. However, if the data has been altered in memory, those changes must be written to the file before the memory can be freed.  
  
- Anonymous: Anonymous memory is a mapping with no file or device backing it. These pages may contain memory requested on-the-fly, or for things such as stack and heap. Because no file is behind an anonymous mapping, a special location must be reserved for the anonymous data. That place is the swap partition/swap file. Anonymous data is written to swap before the anonymous pages are freed.  
  
- Device backed: As you know, block devices are treated as files on Linux. Data can be read or written from/to them. Device backed memory mappings have data from devices stored in them.

- Shared: Multiple page table entries can map the same page of RAM. Accessing a shared entry will display the same contents for any process accessing it, and likewise, writing to it will modify its contents for all consumers. This is typically used for Inter-Process Communication (IPC) e.g. POSIX pipes or BSD sockets.

- Copy on Write: Copy on Write (COW) a.k.a lazy backing is a technique where physical addresses only get backed to a virtual page when a process attempts to write to that location. Using this method, the system does not back the physical memory to virtual memory all at once, but rather waits until an access is attempted and only backs the pages which are requested one by one until the entire mapping has finished. The idea behind this is that having the MMU create the page table entries for a large section of RAM all at once is a compute-intensive task. This is how a function like malloc is implemented, so that if you attempt to allocate 500GB of memory all at once, the program will not crash, but instead wait until you actually attempts to access it.

For swappiness, only file and anonymous mappings are applicable. So I’ve rambled on about memory a lot up until now, but what actually is swappiness??? Well it’s simply a value between 0 and 100 which determines how much the system should rely on swap memory (where the higher the number, the more reliant the system is on swap). In a bit more detail, there are actually 2 values that swappiness controls. They go as follows:

anon\_prio = swappiness; // System reliance on anonymous mappings

file\_prio = 200 – swappiness; // System reliance on file mappings

If you look at this, you realize that as swappiness decreases, the priority of file backings increases, and as swappiness increases (which is also tied to anonymous mappings) file mapping priority decreases.

Another thing to consider is when the system actually decides to use swap... Each memory zone that we looked at earlier have a low water and high water mark. If you’re not familiar with the terms low water and high water, these are common computing terms which just mean a base threshold and max threshold (i.e. do not go below the low water mark, and do not exceed the high water mark). Another example of where I’ve seen these commonly used is with thread pools, where we set the low and high water mark for the number of threads in the pool (but I digress). In order to check the low and high water marks for each memory zone, we can run cat /proc/zoneinfo. This will list out a bunch of information including the number of free pages, the low and high water marks among other fields. In normal conditions, when free memory in a memory zone drops below the low water mark, the swap algorithm starts scanning memory pages looking for memory that it can reclaim, taking into account the relative values of anon\_prio and file\_prio. However, if Linux swappiness is set to 0, then swap occurs when the combined value of the file pages and free pages are less than the high water mark.

In order to check swappiness, run cat /proc/sys/vm/swappiness. In order to set the swappiness during runtime, we can use sysctl: sudo sysctl vm.swappiness=45. To make this a permanent change, we set the value in /etc/sysctl.conf.

Notification Daemons:

Linux has a command called notify-send which is able to send desktop notifications. These work a lot like they do on a system like Windows, though notifications are highly customizable on Linux. Many, though not all applications take advantage of notifications, however, without a notification daemon running, you will not receive any of them. A popular and lightweight notification daemon that I use is called dunst. It has a simple configuration file that allows for changing colors, borders, position, font, size, etc. dunst relies on dbus, which is a messaging bus system that provides the low level means of inter-process communication (IPC). Odds are, you already have dbus installed on your system.

Kernel Event Logging With dmesg:

Kernel events i.e. events which occur within ring 0, 1, or 2 are logged via dmesg, which is a logging daemon for level activity outside of userland. Typically you’ll se things such as usb events, the advanced configuration and power interface (ACPI), file system changes, pci events, etc. You can control the format of dmesg logs as well as the level of logs that you want to filter on. The dmesg command is not all that interesting, but it is helpful if for instance, you’re experiencing system crashes, or odd behaviour pertaining to a PCI or USB peripheral.

Loading Kernel Modules With modprobe:

Dynamic Kernel Module Support (dkms):

Modify Kernel Parameters at Runtime using sysctl:

GPG Keys and PGP Encryption:

Compiling a Custom Kernel:

The Open Group and the XDG Directory Specification:

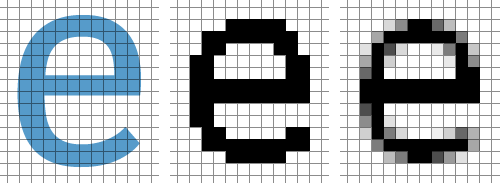
X11 vs Wayland:

Compositors:

Fonts:

I’ll bet that you never thought you would need to be an expert on fonts, but there is so much nuance when it comes to font rendering that you are almost certain to run into a font-related issue on Linux at some point. Knowing how fonts are rendered, categorized, and installed is a great skill to learn on Linux, especially on a minimalist distro where a lot of the font configuration must be handled manually. Most of this guide has been written with Ubuntu users in mind, however, I’m going to transition towards using Arch Linux because I am not familiar with advanced operations like font configuration on Ubuntu, sadly.

Let’s begin with how fonts are rendered. There are actually many things to consider when we render fonts. Things used to be easier when terminals were a set 80 characters in width and used a bit-map font with a static size. Nowadays, terminals can contain more than 80 columns, and even if they don’t support more than that, fonts can usually still be scaled. This brings us to raster vs vector fonts. As I’ve babbled about in other documents such as my computer graphics doc, raster/bit-map graphics are a 2-dimensional array or grid of pixels. The earliest methods of rendering fonts was to use hard coded bit-map fonts with only two bits-per pixel (0 and 1 representing transparent or colored respectively). As we scale these up, the jagged edges become more and more apparent. In order to circumvent this, in the 1990s, operating system began using a technique known as antialiasing, which is still used in videogames and other video graphics today! Essentially, rather than have a binary choice between foreground color or background color per pixel in the character, we could use more bits to represent a spectrum of greyscale values. Using certain mathematical techniques, we could essentially calculate the greyscale value for an individual pixel by taking the average greyscale value from amongst a group of surrounding pixels. This technique has the effect of “softening” edges by creating a smoother transition between the background and foreground color, whilst still preserving the characters shape visually.

Vector, bitmap, antialiasing

We haven’t yet discussed vector graphics, which is what you see in blue above. You see, it is very inefficient to store characters as bitmaps because you either require a scaling algorithm to have them shrink or grow, or you require a separate graphic for each character at each supported font size (which is quitecostly for disk space). Essentially all fonts these days are stored as vector graphics. Once again, I’ve covered these in my computer graphics docs, but to give the brief once more, vector graphics use mathematical functions to describe curves, splines, lines, etc. By aggregating these curves, splines, and lines together in a format (typically we use Scalable Vector Graphics, or SVG format), we can create a character. The nice thing about vector graphics is that no matter how you size them, they always remain consistent in aspect ratio and they never lose information (since we are just applying scalar values to the function outputs). Once we have a vector graphic for a character in the font set, such as the ‘e’ in blue above, we still need to render it to the screen, so it still must undergo rasterization and antialiasing, it’s just that now we are storing the font set in SVG format rather than hardcoding the bitmaps for each character.

As you are probably aware, pixels are composed of three gates which allow red, green, and blue light to filter through. These gates are placed side by side, meaning that one pixel is divided equally into 3 sectors. Programmers began to use this to their advantage in order to create a technique known as sub-pixel rendering. This technique works by turning off at least one of the gates while the other gates remain at some brightness level. This grants us even more leverage in terms of how small we can render fonts and how skinny we an make their ligatures. Thus the text becomes even more crisp than it was using antialiasing.

The next important thing to know about fonts is that they typically come in one of three formats: TrueType, Oft Fonts (based off of PostScript), or WOFF. TrueType fonts will either have a .ttf extension or an .eot extension. Oft Fonts have the .oft extension. WOFF is a format which can contain either TrueType or Oft Fonts, therefore we can’t know which it contains by looking at the file name alone. TrueType and PostScript fonts differ in the mathematics used to store the vector graphics for each glyph. The primary difference between the two is how their approach to *hinting*. Hinting describes where the position of each element that makes up the character should be placed. PostScript/Oft Fonts rely on a smart renderer since they give very abstract hinting information, whereas TrueType fonts provide very specific low-level instructions on where each component should be placed. This tends to be why TrueType fonts are preferred over PostScript/Oft Fonts.

Let’s quickly review the various character encoding schemes. We began with the ASCII standard which can store 128 characters, 93 of which are printable glyphs. Extended ASCII doubled the range of characters to 256 and added additional glyphs for Latin, Greek, various currencies, and box drawing glyphs. As the internet grew, it became clear that ASCII was not a sufficient standard considering the fact that it really only supported English. Thus Unicode was born. Unicode contains a stunning 149,186 glyphs, including characters from almost all languages and numeral systems, as well as emojis and other icons. Unicode control codes are represented as the letter U followed by the ‘+’ symbol, and then 4 digits representing the 2 hexadecimal bytes used to makeup the code. For example, the Greek letter alpha is represented by the control code U+03B1. 98% of all web pages run on the UTF-8 encoding scheme. UTF-8 is capable of encoding all 1,112,064 visible glyphs defined in the Unicode standard. But still, what is the benefit of using UTF-8 over Unicode you ask? Well basically, UTF-8 is known as a variable length character encoding, meaning that it can use 1-4 bytes per character depending on the range of glyphs that need to be accessed. This makes it backwards compatible with ASCII since ASCII characters occupy the first code unit (byte) of UTF-8, thus each ASCII character will only occupy 1 byte as opposed to 2 in Unicode.

I won’t go into too much detail about the technical aspects of how keyboards work, as that can get quite complex, but it is important to understand how your terminal determines which glyph to render when a key is pressed on the keyboard. As you are surely aware, when you press the key on your keyboard, some sort of electrical connection is closed, whether it be a spring in the case of buckling keyboards, a conductive pad in the case of membrane keyboards, or a switch in the case of a mechanical keyboard. A sequence of square waves representing a byte are sent on the wire to the I/O bus where they are recieved as an interrupt. The first byte that is sent was traditionally called the make code, which is when the key is pressed down. This would be a value between 0-127, therefore, we can only have a maximum of 128 functions on our keyboard (at least, traditionally). When a key is released, it sends another byte known as a break code, which is just the make code + 128 since we’re using the other half of the byte range. Key codes are buffered into a keyboard buffer. In the old days, you would get a blue screen of death if you held down more than 16 keys at once since you’d overflow the buffer. Anyhow, now that we know all that, we can look at what Linux does with the keycode specifically. This will depend on the display server/compositor, but on X11 for instance, it will have a specific font set loaded into memory depending on which system settings are in place, and then it will select the glyph at the offset speficied by the keycode to be rendered depending on which encoding is being used. X11 has 3 primary methods of rendering fonts:

1. **Using X11 Draw Protocol:** The “default” way that fonts are rendered by X11 is to take the vector graphic for each glyph, convert it into a bitmap, and then draw them upright. The fact that the protocol does not support drawing the bitmap in any other order other than upright means that we cannot rotate a font using the X11 draw protocol.

2. **Using libxft:** Rather than have X11 render our fonts, we can use a library called Xft (the X(11) font library) to render the characters. Using this method, X11 still loads the font set into memory, but libxft converts the vector graphics into smaller components and shapes and sends those shapes over the X server. The server draws each component and optionally provides antialiasing. We can also rotate fonts using this method. P.S. this is what I use on my machine.

3. **Using a Client-Side Library:** Client-side libraries such as libart\_lgpl or SDL\_ttf do a similar thing to the libxft method, except that rather than sending the vector components and shapes to the X server to antialias and/or rotate, all of that is done by the library *before* it is sent to the server.

Now we can finally begin discussing how fonts are categorized. A *typeface* a.k.a. “font family” is an individual font which may be composed of multiple variations of the same font. The *weight* of a font describes whether it’s bold or not, whereas the *style* determines if its italicized or not. Fonts with ligatures (the sticky-outy bits) are called serif fonts. A sans-serif font is one without ligatures. A monospace font is a font that uses consistent spacing between each character. Non-monospace fonts may use a technique known as kerning to make them flow more naturally. The kerning value determines the offset and width of each individual character, and may fluxuate per character depending on the surrounding characters. This is necessary for mimicing cursive, as we need to align each character with its neighboring characters. Naturally, monospaced fonts work the best for terminals since each character can occupy a consistent width. Serif, Sans Serif, and Mono are the most basic typefaces, and are usually used as a fallback. Arial, for example, is a specific sans-serif font, but should not be confused with the actual font “Sans Serif”.

A popular package that may or may not come with your distro are the font families provided by Google on <https://fonts.google.com/>. It was actually in Googles interest to create uniformity between browsers and other applications so they created a bunch of fonts licensed under the SIL Open Font License. Essentially, these are fonts that anyone can use for free. Notable font families that you will often see or hear mention of are the Noto fonts, Roboto fonts, Source fonts, PT fonts, etc. Speaking of free fonts, GNU provides a font family called GNU FreeFont if you’re a libre software extremist, though they’re not very nice unfortunately. The AUR even has a package for Mac fonts. Another popular “font” is Font Awesome. Font Awesome don’t actually provide a typeface, but rather a collection of icons which are open source and free to use. Font Awesome is considered to be a “font toolkit” and was originally intended to be used with Bootstrap, but is now used for basically anything and everything. One last one I have to mention is Nerd Fonts. Nerd Fonts have SVG alternatives to a lot of bitmap fonts and they bundle a bunch of popular icons from Font Awesome, Devicons, and Octicons.S

Now we can discuss how a font typeface is selected using Arch Linux as an example. On Arch, fonts are installed to /usr/share/fonts so that they are accessible to all users. Font profiles are located in /usr/share/fontconfig/. In that directory should be two sub-directories: conf.avail and conf.default. conf.default just contains soft links to the files in conf.avail. These font profiles are basically like modules that we can load into our primary font configuration file. The primary font configuration file is located by default in /etc/fonts/ and is called fonts.conf. This is an autogenerated XML file that contains syntax for the global font configuration that will apply to all users on the system. Also in /etc/fonts/ is a sub-directory called conf.d. Here we can create softlinks to the modules in /usr/share/fontconfig/conf.avail to have them be loaded into fonts.conf. You need to at least have softlinks to 50-user.conf and 51-local.conf. Another module called 10-sub-pixel-rgb.conf allows for sub-pixel rendering, and there are others that you may want to load as well. If you want a specific font configuration for your user account that will not apply globally to other user accounts, then we can create a file in ~/.config/fontconfig/ called fonts.conf which will override the /etc/fonts/fonts.conf file. There, we can also create a conf.d directory which will contain the softlinks towards the modules that we want to load in the local font configuration file. The man page for fonts.conf details the proper XML elements and attributes that are available to you so that you can customize or override specific fonts.

Finally, let’s discuss some command line utilities that are used for configuring fonts. Once you have a fonts.conf configuration file, you’ll need to use fc-cache -fv to reset your font cache. Another very powerful tool is fc-match. The fc-match command is how certain X11 applications determine what font to load based on a given input string. The fc-match command will match based on various properties defined in fonts.conf. As you can imagine, fc-match prioritizes certain properties over others, and it is also influenced by your fonts.conf file. Let’s say I want to find the font that Linux will use if an application requests a monospace font which is bold and contains characters from the english language. I can run fc-match :family=mono:weight=bold:lang=en which will spit out the best fit that fc-match can find. If you want to visually test a font you can use pango-view which comes with the pango package (you likely already have this installed). You can run pango-view --font=”font” test.txt to view the contents of test.txt rendered in the font specified. Use the --dpi flag to specify the font size rather than including it as part of the --font string.

Audio on Linux:

In my experience, audio is the most complex thing to manage on Linux. If you’re using a minimalist distro, such as myself, you may find that getting audio to work is often a huge pain in the arse, and that it tends to break frequently. The primary audio driver on Linux is the Advanced Linux Sound Architecture (Alsa) API. Alsa is builtin to the Linux kernel. Later on we would get PulseAudio from the controversial figure Lennart Poettering. Say what you might, PulseAudio became an extremely popular abstraction to Alsa, acting as an interface engine to communicate between applications and Alsa. It acts as a sound server, where a daemon runs in the background and accepts sound input from one or more sources. The PulseAudio daemon then redirects the sound sources to one or more sinks (sound cards, remote network PulseAudio servers, or other processes). Most recently introduced is PipeWire, another audio server and also video server. Of course, audio on Linux is much more messy than just Alsa, PulseAudio, and PipeWire. Another factor that adds additional complexity is the bluetooth stack, which is managed by BlueZ. BlueZ is not as confusing as the rest, but tends to break often with every new update. I hope to go into more depth on each of these so that you can have a better understanding of audio on Linux.

Alsa:

The Advanced Linux Sound Architecture (ALSA) was a flat out replacement for the Open Sound System (OSS), which was the original sound system for Linux. When I say flat out replacement, I mean that it made it’s way into the kernel in the form of a built-in kernel module, and it is still present as a part of the Linux kernel to this day. Alsa is a pretty capable low level sound system and still works well enough for some people. The alsamixer and amixer commands allow users to control audio settings from either a TUI (alsamixer) or the CLI (amixer).

PulseAudio and Jack:

PipeWire, PipeWire-Pulse, PipeWire-Alsa:

Music Player Daemon (MPD):