High Performance Computing

Muchang Bahng

July 2, 2023

Contents

In the first chapter, we introduce the basics of programming and computer hardware within the context of Windows or Mac operating systems, since that is what most people are familiar with. Then, I will explain everything within the context of Linux operating systems due to their greater flexibility and functionality, specifically Ubuntu 22.04 Jammy Jellyfish.

1 Basics

1.1 System Hardware

1.1.1 Non-Volatile Drive Storage

A **drive** is basically a computer component used to store data. It may be a static storage device (e.g. a HDD or SSD) or may use removable media (e.g. thumb, disk, CD). All drives store nonvolatile data (also called nonvolatile memory, NVM), meaning that the data is not erased when the power is turned off.

1. A **floppy disk drive** is a portable circular floppy plastic/metal disk coated with iron oxide or other magnetic material. They come in many sizes ranging from 3 8 inches in diameter, with the standard capacity being 1.44MB. When inserting the floppy disk into a computer, there is a read/write head that uses a magnet to polarize the iron particles in one of two directions, each represting a 0 or 1 in binary data. The head can also read these polarities in order to retrieve data stored on the disk in the form of polarized particles. Note that the head would read the disk "circularly" as the disk rotates. Each disk would be divided into typically 40 tracks with around 8 equal sectors.

2. A **hard disk drive (HDD)** is an electro-mechanical data storage device that stores and retrieves digital data using magnetic storage and one or more rigid (hence, the name hard) rapidly rotating platters coated with magnetic material. Data is accessed in arandom-access manner, meaning that individual blocks of data can be stored and retrieved in any order. They usually come inside a metal case enclosing the entire drive (3.5-inch for computers and 2.5-inch for laptop HDDs). Since the data on the HDD is determined by the polarities of the magnetic material on the disks, it is sensitive to external magnetic fields that may corrupt the data. Furthermore, because the drive heads must align over an area of the disk in order to read or write data, and the disk is constantly spinning, there's a delay before data can be accessed. The drive may need to read from multiple locations in order to launch a program or load a file, which means it may haveto wait for the platters to spin into the proper position multiple times before it can complete the command. If a drive is asleep or in a low-power state, it can take several seconds more for the disk to spin up to full power and begin operating. Their speeds are measured in RPM, with the normal range of desktop HDDs having 5400-7200 RPM. It is useful to know that 5400 RPM drives offer an average of 100MB/s read and 7200 RPM drives offer 120MB/s.

3. A **Solid State Drive (SSD)** is an extra step up from the HDD. From the very beginning, it was clear that hard drives couldn't possibly match the speeds at which CPUs could operate. Latency in HDDs is measured in milliseconds, compared with nanoseconds for your typical CPU. One millisecond is 1,000,000 nanoseconds, and it typically takes a hard drive 10-15 milliseconds to find data on the drive and begin reading it. The hard drive industry introduced smaller platters, on-disk memory caches, and faster spindle speeds to counteract this trend, but there's only so fast drives can spin. Western Digital's 10,000 RPM VelociRaptor family is the fastest set of drives ever built for the consumer market, while some enterprise drives spun as quickly as 15,000 RPM. The problem is, even the fastest spinning drive with the largest caches and smallest platters are still achingly slow as far as your CPU is concerned. Unlike HDDs, soid state drives do not need moving parts or spinning disks (hence their name). Instead, it uses NAND flash memory, which is a type of non-volatile storage that erases data in units called blocks and rewrites data at the Byte level. It also retains data for decades, regardless of whether the device is powered on or off. It is used in not only SSD, but also USB flash drives, SD cards, mobile phones, digital cameras, tablets, and others. The most fundamental unit of storage is the flash memory cell, which uses electron thresholds to hold certain bits of information, usually three bits (called TLC - triple level cell) or four bits (called QLC - quad level cell). If is no electron charge in the cell, the cell

Windows CMD	MacOS Terminal	Task
wmic LOGICALDISK LIST BRIEF		Lists all the drives on your computer. Note that the DeviceID is simply the drive letter, the DriveType has numerical encodings (2: Removable disk, 3: Fixed local disk, 4: Network disk), the FreeSpace and Size are in bytes, and the VolumeName is the name of the disk.
wmic diskdrive get status, model		Outputs the model name of the drive along with its status. If the status is OK, the health is good, and if it shows 'Pred Fail', your drive may crash soon.
chkdsk c:		Checks the file system and provides a summary of issues on the drive. If the bad disk sectors are not 0, get technical assistance.
dfrgui		Opens a window that tells you all types of drives on your computer.
<letter>: (d:)		Changes the drive you are working in.

represents a 111 (for TLC) and 1111 (for QLC). Since each cell can store multiple bits of information, they are arranged in a large array (consisting of millions of cells stacked on top of each other) into a block, leading to a typical storage between 256KB and 4MB.

On Windows, each type of drive on a computer are assigned a device/drive letter, a single alphabetic character A through Z. Computers containing a hard drive always have that default hard drive assigned to a C: drive letter, and external drives may be assigned different letters, such as Google Drive being assigned a G: drive letter. You may also notice that when opening the command prompt on windows, the leftmost letter represents which drive you are currently on. Note that the "wmic" is an abbreviation of Windows Management Interface Command. Some commands may require you to use an **elevated command prompt**, which can be used by opening the cmd file as an administrator.

```
C:\Users\bahng>
```

We demonstrate some of the commands here.

```
C:\Users\bahng>wmic LOGICALDISK LIST BRIEF
  DeviceID  DriveType  FreeSpace     ProviderName  Size           VolumeName
  C:        3          838628864000                1003327844352  OS
  G:        3          71506178048                 107374182400   Google Drive

C:\Users\bahng>wmic diskdrive get status, model
  Model                     Status
  PM9A1 NVMe Samsung 1024GB  OK
```

1.1.2 Volatile, Short-Term Storage

Information travels from drives and other stores to the CPU, but the physical distance that the bits must travel across the motherboard also puts an upper limit on the retrieval speed (i.e. the speed of electromagnetic waves), especially if the distance must be covered thousands or millions of times back and forth. This limit is known as **latency**. This is why computers have a hierarchy of stores reserved for information that is accessed more frequently, some closer to the CPU and others even within the CPU itself!

1. **Random Access Memory**, or **RAM**, is short-term memory that acts as a cache for the CPU that is 50-200 times faster than a regular SSD. It is volatile, meaning that all its memory is erased when

Windows CMD	MacOS Terminal	Task
wmic MEMORYCHIP (get BankLabel, DeviceLocator, MemoryType, TypeDetail, Capacity, Speed)		Outputs relevant information about the RAM.
wmic memoryship list full		Outputs a list of all specifications of each memory stick.
systeminfo — findstr /C:"Total Physical Memory"		Outputs the total RAM memory of your computer.
systeminfo —find "Available Physical Memory"		Outputs the available RAM memory of your computer.

the computer shuts down. The most recent type of RAM is DDR4, then DDR3, followed by DDR2, DDR, and SDRAM. In addition the speeds of RAM is

(a) DDR4: 2133mhz, 2400mhz, 2666mhz, 3200mhz

(b) DDR3: 1066mhz, 1300mhz, 1600mhz, 1866mhz

where the mhz value represents how many times per second the RAM can access its memory. However, know that some motherboards have technical limitations to what kind of RAM speed it can handle, so in these cases, the system will throttle your faster RAM stick to meet this need. In terms of capacity, the following gives us nice benchmarks.

(a) 4-8 GB: Laptops for web browsing and light gaming (e.g. my Macbook Air 2019)

(b) 16-32 GB: Laptops for gaming (possibly heavy) and programming

(c) 64-128 GB: Crazy stuff.

(d) 256 GB: This basically means you have a RAM that is pretty much the size of a typical SSD. This is usually for specialized research computers or clusters.

Finally, there are broad categories of RAM.

(a) **Static RAM (SRAM)** requires a constant power flow in order to function and therefore doesn't need to be refreshed to keep the data intact (hence the name static). Note that this does not mean that SRAM is nonvolatile. Therefore, the SRAM is typically used in CPU caches or video cards.

(b) **Dynamic RAM (DRAM)** requires a periodic refresh of power in order to function. The capacitors that store data in DRAM gradually discharge energy (no energy means the data becomes lost). DRAM is found in systems memory and video graphics memory.

(c) **Synchronous Dynamic RAM (SDRAM)** is a DRAM that operates in sync with the CPU clock, which means that it waits for the clock signal before responding to data input. This is advantageous since the CPU can process overlapping instructions in parallel, known as pipelining (the ability to reveive an instruction before the previous instruction has been fully resolved). This allows more instructions to be completed simultaneously. By contrast, DRAM is asynchronous, which means it responsd immediately to user input.

2. **CPU caches** have a hierarchy that is divided into (from fastest to slowest) L1, L2, L3, and sometimes even L4. The CPU will check the L1 cache first to see if there is a hit (if there is, then data is retrieved extremely fast), then the L2, and so on.

(a) L1: 8-64 KB storage typically (but there are exceptions, i.e. the Apple M1 chip has a 192 KB L1 cache)

(b) L2: 256KB-8MB storage

(c) L3: 10-64MB storage (and sometimes up to 256MB for server chips)

Windows CMD	MacOS Terminal	Task
wmic cpu list full		Outputs a list of all specifications of the CPU.
wmic cpu (get caption, deviceid, name, numberofcores, maxclockspeed, status)		Outputs relevant information about the CPU.

For my computer, the outputs are as such. It shows two memory sticks each with 8GB of memory, a memory type of 0 which is a DDR4 (type 24 means DDR3), speeds of 3200 mhz, and TypeDetail of 128 which means the RAM is synchronous (SDRAM).

```
C:\Users\bahng>wmic MEMORYCHIP get BankLabel, DeviceLocator, MemoryType, TypeDetail, Capacity, Speed
BankLabel  Capacity    DeviceLocator  MemoryType  Speed  TypeDetail
           8589934592  DIMM A         0           3200   128
           8589934592  DIMM B         0           3200   128
```

The status of your CPU can be checked with the following commands

1.2 Program Lifecycle Phases

First, we review some definitions. More on program lifecycle phases here. Programming languages are broadly classified into two types. **High-level languages** are the familiar programming languages that we work with today (that allow much more abstraction), while **low-level languages** are very close to the hardware, such as machine language and assembly language. Programmers write programs in **source code** (usually high-level languages), which are then inputted into **language processors** that translate them into **object code** (usually **machine code** consisting of binary). The duration in which the source code of the program is being edited is called the **edit time**, while the **compile time** is when the source code is translated into machine code by a language processor. There are three types of language processors.

1. A **compiler** is a language processor that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language. The source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again. (e.g. C, C++, C#, Java)

2. An **assmebler** is used to translate the program written in Assembly language (basically a low-level language with very strong correspondence between the instructions in the language and the machine code instructions) into machine code. The assembler is basically the 1st interface that is able to communicate humans with the machine. We need an assembler to fill the gap between human and machine so that they can communicate with each other. Code written in assembly language is some sort of mnemonics (instructions) like ADD, MUL, MUX, SUB, DIV, MOV and so on, and the assembler is basically able to convert these mnemonics into binary code.

3. An **interpreter** translates a single statement of the source program into machine code and executes immediately before moving on to the next line. If there is an error in the statement, the interpreter terminates its translating at that statement and displays an error message. The interpreter moves on to the next line for execution only after the removal of the error. An interpreter directly executes instructions written source code without previously converting them to an object code or machine code. (e.g. Python, Pearl, JavaScript, Ruby)

A quick compare and contrast.

The result of a successful compilation is an executable, which is a program in the form of a file containing millions of lines of very simple machine code instructions (e.g. add 2 numbers or compare 2 numbers), also called **processor instructions**. This executable can be stored somewhere in the computer drive for future use or it may be copied immediately in a faster memory state, such as the RAM. The **load time** is when

Compiler	Interpreter
Takes more time to analyze source code but execution time is faster.	Takes less time to analyze source code but execution time is slower.
Debugging is harder since the compiler generates an error message after the entire scan.	Debugging is easier since the interpreter continues translating the program until an error is met.
Requires a lot of memory for generating object codes.	Requires less memory because no object code is generated.
Generates intermediate object code.	No intermediate object code is generated.

the OS takes the program's executable from storage and puts it into an active memory (e.g. RAM) in order to begin execution.

The CPU understands only a low level machine code language (aka native code), which is contained within the executable. The language of the machine code is hardwired into the design of the CPU hardware; it is not something that can be changed at will. Each family of compatible CPUs (e.g. the popular Intel x86 family) has its own, idiosyncratic machine code which is not compatible with the machine code of other CPU families. More information here. Once the instruction bytes are copied from storage to RAM, the CPU can run through the steps/lines at the rate of about 2 billion lines/steps per second. This execution phase, when the CPU executes the instructions until normal termination or a crash, is called the **runtime**.

1.2.1 More on Executables

More specifically, an **executable** is a file that contains a list of instructions and data to cause a computer's CPU to perform indicated tasks, as opposed to the data files, which are fundamentally strings of data that must be interpreted (parsed) by a program to be meaningful. Executables usually have extension names `.exe` or `.bat`, and they can generally be run (invoked) in two ways:

1. The executable file can be run by simply double clicking on the file name, opening it, and having the user type commands in an interactive session of an interpter (like inputting commands in terminal window or a python shell).
2. Alternatively, we can start writing a program, complete writing it, and then have this program compiled into an executable to be invoked.

Some common examples of executables are:

1. python.exe - used to run python scripts that have the .py extension, located at

   `C: \Users\bahng\AppData\Local\Programs\Python\Python39`

2. pythonw.exe - used to run .pyw files for GUI programs
3. terminal.exe (on MacOS)
4. cmd.exe (on Windows OS)
5. py.exe - an executable used to run the python.exe executable like a shortcut, located at

   `C:\windows\py.exe`

1.2.2 Static vs Dynamic Languages

Type-checking is the process of checking and verifying the type of a construct (constant, variable, array, list, object) and its usage context. It helps in minimizing the possibility of type errors in the program, and type checking may occur either at compile-time (static checking) or at run-time (dynamic checking).

1. **Statically-Typed Languages**: Since we type check during compilation, every detail about the variables and all the data types must be known before we do the compiling process. Once a variable is assigned a type, it can't be assigned to some other variable of a different type, and so the data type of a declared variable is fixed. This makes sense since in Java, C, C++, etc., the programmer must specify what the data type of each variable is by writing something like `int myNum = 15`.

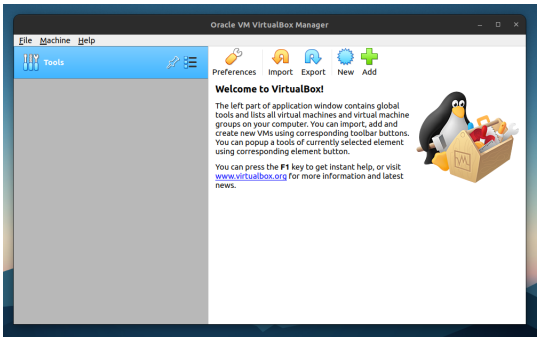
2. **Dynamically-Typed Languages**: Since we type-check during runtime, there is no need to specify the data type of each variable while writing code, which improves writing speed. These languages have the capability to identify the type of each variable during run-time, so we do not need to declare the data types of variables. In these languages, variables are bound to objects at run-time using assignment statements, and most modern languages (e.g. JavaScript, Python, PHP, etc.) are dynamically typed.

2 Linux Operating System

2.1 Setting Up Linux

2.1.1 Virtual Machines

You can always try out Ubuntu (or any other distribution) through a **virtual machine**, which is a software emulation of a physical computer system. It allows you to run multiple operating systems or instances of an operating system on a single physical machine. Each virtual machine operates independently and has its own virtual hardware, including virtual CPU, memory, storage, and network interfaces. Virtual machines are created and managed by virtualization software called **hypervisors**. The hypervisor abstracts the underlying physical hardware and allows multiple virtual machines to share the same resources while isolating them from one another. This enables efficient utilization of hardware resources and provides flexibility in deploying and managing various operating systems and software applications. VMs generally have the advantage of being completely isolated from the main computer, so if anything wrong happens in the VM, it's fine. They can be used in research environments that are beta-testing unstable packages or for white-hacking practices. One example of a hypervisor is Oracle's **VirtualBox**, which is free to download. It should look like this when you open it for the first time.



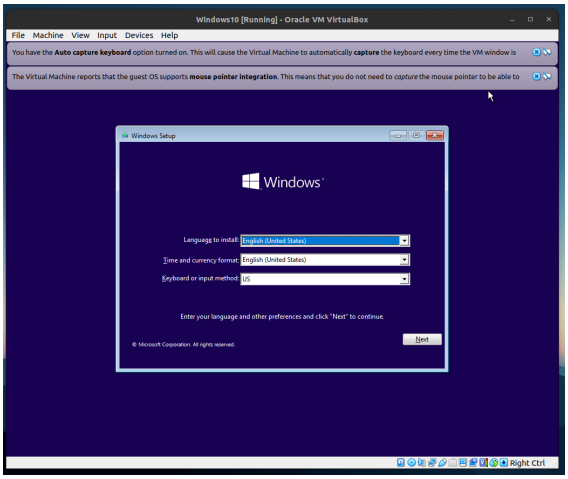
Now in order to create a VM with its own OS, you need to have the appropriate **ISO file**, which is an exact copy of an entire optical disk such as a CD, DVD, or Blu-ray archived into a single file. The essentially stores the entire software needed to operate the OS. Therefore, you should download the proper ISO file from the internet (usually a couple GBs).

1. Ubunutu ISO files
2. Windows 10 ISO files
3. Apple does not allow distribution of its ISO files, so you will need to download from unofficial sources, which may be unsafe.

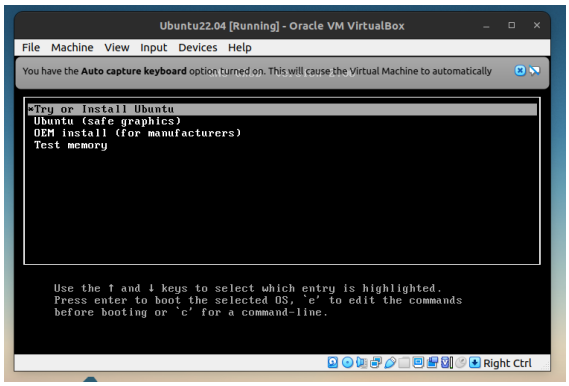
Once you have this ISO file, you can reuse it to create as many VMs as you want of that OS. Now follow these instructions: Click the new button and select where the virtual machine data will be stored, along

with its OS. You can set the RAM, but don't make it more than half of your host computer since it will hog up too much RAM. Choose "Create a virtual hard disk now". Choose "VDI (VirtualBox Disk Image)". Dynamically allocated just means that the virtual disk size will adaptively grow as your storage gets full. Set the disk size to be at least 20GB.

After you created this, go to the VM settings (this is where you can edit your CPU cores, RAM cap, etc.). To add the ISO file, click on the "Empty" tab right under the "Controller:IDE", then the CD icon to the right, and "choose a disk file". You should now choose the ISO file. Then go tweak other settings, and set the display:video memory to the max (128MB). Now you should be able to go through the installation wizard when you turn the VM on. Refer to the instructions for each OS.



(a) Windows 10 Set Up



(b) Ubuntu 22.04 Set Up

Figure 1: What you should get once you open up the VM after adding ISO files.

1. For Windows: Say I don't have a product key. Click Windows 10 Home. Accept terms. Select the custom installation. Click the drive and click new, making the parititon at least 10534MB, and click apply. Next. Wait for the system to load.
2. For Ubuntu, you should get a GRUB view. Select "Try or install Ubunutu".

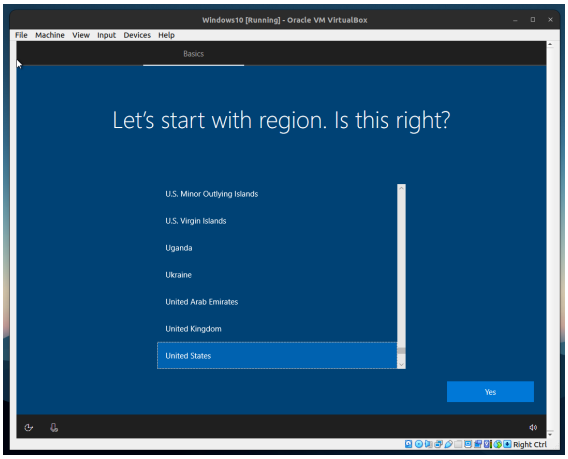
You should now see one of these two screens.

1. For Windows, select your region. Select the keyboard layout. Sign in or create a Microsoft account. Choose privacy terms. Skip whatever.
2. For Ubuntu: Select Install Ubuntu with English. Set the keyboard layout. The normal installation may take a while, so I would select minimal depending on what you need. If you are short on time, you can uncheck the download updates while installing since you can always do that after you install. Click Erase disk and install Ubunutu. Choose region and add information.

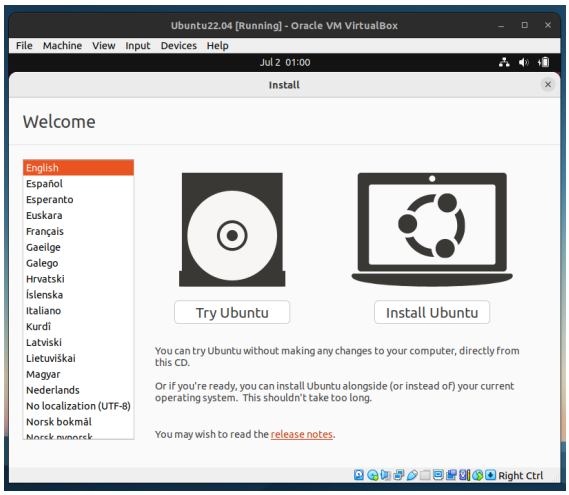
Finally, you should see your desktop.

2.1.2 Pipeline for VM

For my personal use, the packages below are ones that I end up installing every time I create a new VM to work in during research.

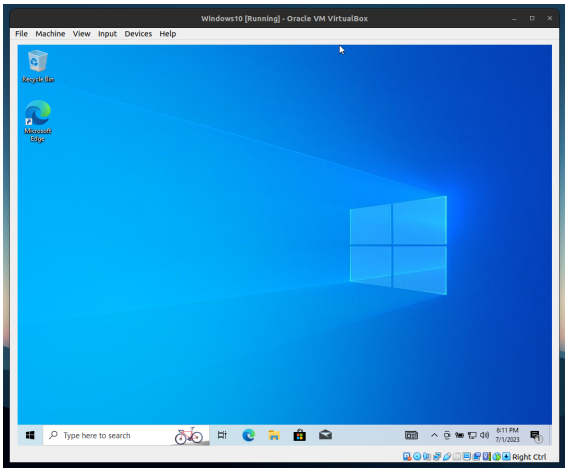


(a) Windows 10 Set Up

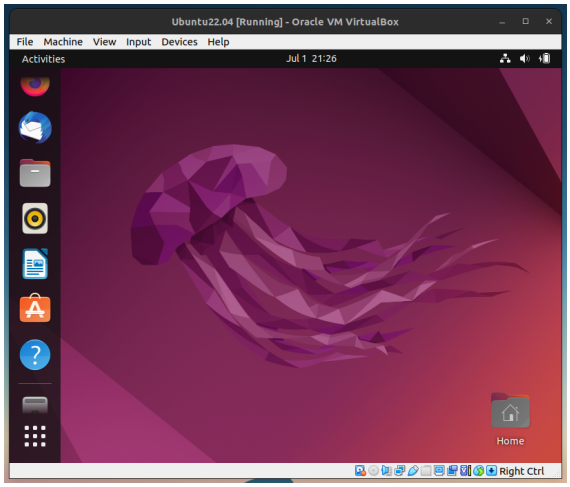


(b) Ubuntu 22.04 Set Up

Figure 2: What you should get once you open up the VM after initial configuration and log in.



(a) Windows 10 Set Up



(b) Ubuntu 22.04 Set Up

Figure 3: What you should see once everything is set up.

```
sudo apt update
sudo apt install snapd
sudo snap install --classic code
sudo snap install slack
sudo apt install git
sudo snap install spotify
sudo apt install htop


wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
sudo dpkg -i google-chrome-stable_current_amd64.deb


sudo apt install virtualbox
```

2.1.3 Dual Booting Linux with Windows

Once you are ready to use Linux consistently, it is optimal to dual boot it, which means that you have one computer that is divided into two: one for each operating system. Then you need to partition your drive and allocate it to your secondary OS. There are plenty of guides and tutorials online on how to do this.

There may be a point where you may need to resize your drive partitions as you need more or less space in one of your OS. This is when we need to do **partition resizing**. To do this, we need an empty thumb drive with at least 8GB of space in it (everything in here will be deleted). Then in your Ubuntu, install balenaEtcher and an Ubuntu (any version) ISO file. Mount the ISO file into your USB drive using balenaEtcher, following the steps in this video to eventually get into Gparted. Another popular guide uses Rufus in the Windows system, but I have found that this does not work for me.

2.2 Shell Commands

Let us first review some basic Unix commands that should be built in every new Ubuntu system.

```
ls
cd
ls --color
gedit
df -h
```

Now

```
htop
```

Additional

```
grep
```

2.3 Booting Process

The boot process on Linux is basically a series of activities that occur from the time you press the power button on your PC until the time the login screen appears. There are 4 main stages in the boot process of your operating system and they occur in the following order.

1. **BIOS** (**Basic Input/Output System**) is mainly responsible for loading the bootloader. When the computer starts, it runs a **Power on Self Test (POST)** to make sure that core hardware such as the memory and hard disk is working properly. Afterward, the BIOS will check the primary hard drives' **Master Boot Record (MBR)**, which is a section on your hard drive where the bootloader is located. **UEFI** (Unified Extensible Firmware Interface) is the successfor to BIOS, allowing for booting support for volumes over 2TB in size, but this doesn't matter.
2. The **bootloader** loads the kernel into the RAM with a set of kernel parameters.
3. The **kernel**'s primary function is to initialize devices and memory. Afterward, it loads the init process.
4. The **init** is responsible for starting and stopping essential services on your system.

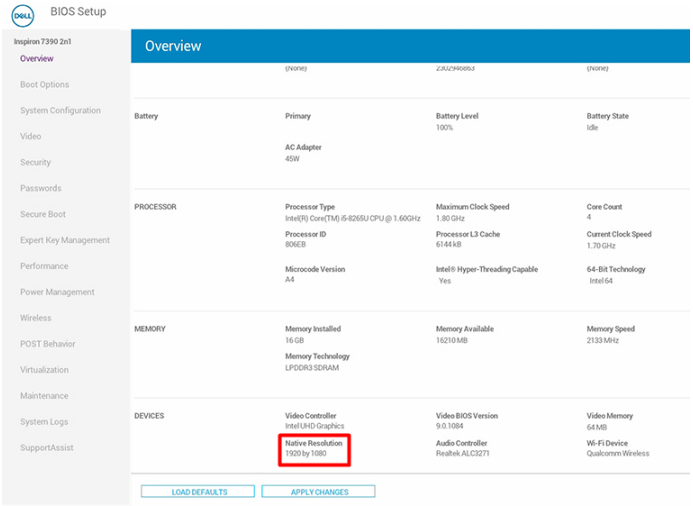
2.3.1 GRUB

Regarding the bootloader, different operating systems have different kernels, and so for instances where you have a dual boot, the **GRUB** (**Grand Unified Bootloader**) provides us with a menu so that we can select the proper operating system or environment that we wish to long into. Here is a picture of a GRUB menu.

Ubuntu does not display the GRUB menu by default. To see GRUB during boot you need to press the right-hand SHIFT key during boot. In fact, the `/boot/grub/grub.cfg` file is the main configuration file for the GRUB bootloader, but it is not advised to edit it directly. Rather, we can edit the `/etc/default/grub` file and run `sudo update-grub` to that the changes are written to `grub.cfg` automatically.

2.3.2 BIOS Setup

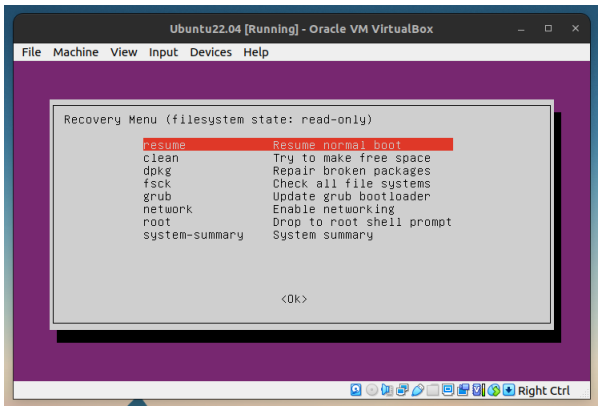
For further booting functionality, you can press a certain key when booting (F2 on my Dell XPS15 9500) to enter the **BIOS setup**, which can look very different depending on the computer but looks like this for me:



From here, we can edit different settings like boot options (priority of booting OS), certain video settings, etc.

2.3.3 Recovery Mode

Occasionally, you may run into problems with booting up the system. You can go into **recovery mode** by looking at the advanced options in the GRUB menu and selecting the option that literally says recovery mode.



This gives us a list of options that we can take to fix the system. Every setting except root is automatically done. The root command gives us root privileges (no sudo is needed). This also means we have full access to all files, and we may cause irreversible damage to our system if we made a mistake. If we had not enabled read/write access with "Enable networking" the filesystem will be mounted read only, and we are unable to edit files. In case we don't have access to a network, or this was not desired, we can remount our filesystem(s) giving write access with the following command:

```
mount -o rw,remount /
```

With editing privileges, we can hopefully better diagnose or undo our problems. Finally, from the root shell type `exit` to go back to the menu.

2.4 Directory Structure

It should be clear that the ~ stands for your user home directory, while `/` stands for the root directory.

```
(base) mbahng@xps15:~\$ pwd
/home/mbahng
(base) mbahng@xps15:~\$ cd /
(base) mbahng@xps15:/\$ pwd
/
```

Let us now take a look at the contents of the root directory:

```
(base) mbahng@xps15:~\$ ls /
bin    dev   lib    libx32      mnt   root  snap  timeshift  var
boot   etc   lib32  lost+found  opt   run   srv   tmp
cdrom  home  lib64  media       proc  sbin  sys   usr
```

You can see that the root home directory is in here, as opposed to user home directories in the `/home` folder.

1. `root`: This contains all the files for when you need to boot. You shouldn't mess with this.
2. `etc`: This is where you system wide configuration for applications is stored (unlike local configuration files for one user, which is stored in your home directory). It is often a target for backups.
3. `media`, `mnt`: Used for mounting external storage systems and even internal storage systems.
4. `opt`: A place where you can install whatever you want. Quite flexible.

2.5 Users and Permission

2.5.1 Managing Users

You should first check which users are on your system. Most people just check their home directory using

```
(base) mbahng@xps15:~\$ ls -l /home
total 8
drwxr-xr-x  3 root   root   4096 Jan 17 23:57 linuxbrew
drwxr-xr-x 44 mbahng mbahng 4096 Jul  2 13:27 mbahng
```

But this is not accurate. Rather, we should check the contents of the `/etc/passwd` file, which has a list of users in our computer (1 per line). The purpose is to contain a listing of and the options that are associated with your user accounts on your server.

```
(base) mbahng@xps15:~\$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
...
```

Let us just examine my user.

```
(base) mbahng@xps15:~\$ cat /etc/passwd | grep mbahng
mbahng:x:1000:1000:mbahng,,,:/home/mbahng:/bin/bash
```

Going from left to right, mbahng is my user, the x stands for a hashed password that cannot be shown, the 1000 is the user id (UID), the 1000 is the group id (GID), mbahng is the user information field (optional), next `/home/mbahng` is the user's home directory, and finally `/bin/bash` is the shell designated for the user. When you create a user id when first installing Ubuntu, this will almost always have uid of 1000. On most linux distributions, the user accounts that will be used by humans are given uids of 1000 and above. Note that in Ubunutu 22.04, a home directory is not created automatically (this differs based on distribution) when we create a new user. So note the following commands. To add a user called batman, we have

```
(base) mbahng@xps15:~\$ sudo useradd batman         # just add user
(base) mbahng@xps15:~\$ sudo useradd -m batman      # add user with home dir
(base) mbahng@xps15:~\$ cat /etc/passwd | grep batman
batman:x:1001:1001::/home/batman:/bin/sh
```

and it gives a new uid that is the next available one from 1000, i.e. 1001. To delete the user, just do

```
(base) mbahng@xps15:~\$ sudo userdel batman         # delete user
(base) mbahng@xps15:~\$ sudo userdel -r batman      # delete user w/ home dir
```

Now let's talk about changing passwords. If you want to change your own password, you can just type `passwd` and go through the steps. To set another user's password, you need to be in root mode and type

```
(base) mbahng@xps15:~\$ sudo passwd batman          # set password for batman
```

Note that we have a hashed version of the user's password in the `/etc/passwd` file. We can actually see the full hashed versions by going into `/etc/shadow`.

2.5.2 Changing Permission

Running `ls -l` command lists all files and directories in your current working directory, along with their permissions.

```
-rw-rw-r--  1 mbahng mbahng 4336730777 Sep 29  2022  cuda_11.8.0_520.61.05_linux.run
drwxr-xr-x  9 mbahng mbahng       4096 Jul  1 23:33  Desktop
drwxr-xr-x  8 mbahng mbahng       4096 Jul  1 15:08  Documents
drwxr-xr-x  6 mbahng mbahng      12288 Jul  1 22:36  Downloads
drwxr-xr-x  4 mbahng mbahng       4096 Jun 29 19:43  Games
drwxr-xr-x  6 mbahng mbahng       4096 Feb 22 17:27  Jts
drwxrwxr-x  5 mbahng mbahng       4096 Jun 28 19:39  KakaoTalk
drwxr-xr-x 16 mbahng mbahng       4096 Jun  2 21:13  miniconda3
drwxrwxr-x  4 mbahng mbahng       4096 Jun 22 13:12  nltk_data
```

The first columm is a string of 10 characters representing the permissions. They are divided into 4 sections:

```
d   rwx   r-x   r-x
```

The first letter can be a d, l, or -, meaning directory, link, or file, respectively. The next three groups, representing the permissions of the user (third columm), group (fourth), and everyone else, have the same format. It is rwx, which stands for read, write, execute.

1. Read: Means to read a file or read a directory.
2. Write: Means to edit a file or modify the contents of a directory.
3. Execute: Means to run the file as an executable or go `cd` into the directory.

A dash in place of any one of them means that whatever entity does not have the permissions. However, we can set the permissions using the `chmod` command. If we have a file named `testfile.txt` in our current directory, we can add or revoke permissions with

```
chmod +r testfile.txt   // assign read permissions to all users
chmod +w testfile.txt   // assign write permissions to all users
chmod +x testfile.txt   // assign execute permissions to all users

chmod g+rw testfile.txt   // assign read and write to group
chmod u-r testfile.txt   // revoke read to user
chmod o+x testfile.txt   // assign execute to other users
```

Writing all these can be tedious, so what we can do is take advantage of the numerical encodings of the permissions. Note that $r = 4, w = 2, x = 1$, and so any number between 0 and 7 can encode the three bits (through the coefficients of the binary expansion). Therefore, if we wanted every permission for all users, we can write

```
chmod 770 testfile.txt
```

where the first 7 stands for `rwx`, the next 7 stands for `rwx`, and the final 0 stands for `---`. To change the permissions for everything inside a directory (e.g. say you want to make all downloads only readable and writable by you), then you can type

```
chmod 600 ~/Downloads/*
```

2.5.3 Changing Ownership

If you have multiple users in your computer (type `ls /home`), then you may want to give ownership of a directory or folder to another user.

```
(base) mbahng@xps15: ls /home
batman mbahng
```

To change permissions of a file/directory to another user and group, we can use the `chown` command (with sudo)

```
sudo chown -R batman:batman Downloads/
```

2.6 Package Managers

2.6.1 Dpkg and Deb files

Ubuntu is a Linux distribution within the family of Debian-based systems (with Debian, Linux Mint, etc.). File of the `.deb` format is used to distribute and install software packages on these systems. A deb package contains the files for a particular software application or library, along with metadata that describes the package and instructions on how to install or remove it. The package format follows a specific structure and includes files such as control files, data files, and scripts. Therefore, many downloaded packages may come in this format, similar to how a file is zipped before we have to extract it.

Dpkg is the primary package manager for Debian based systems. It installs, builds, removes, configures, and retrieves information for Debian packages of the `.deb` format. Given that we have some file `package.deb` downloaded, the command

```
dpkg -i package.deb
```

installs the specified package from the `package.deb` file. Removing it is just (note without the suffix)

```
dpkg -r package
```

2.6.2 Apt

read more here

While dpkg is the native package manager for Debian based systems, apt is just a built-in Ubuntu tool to help install these Debian packages and manage dependencies. To run apt commands, we must have root privilege, so we should always use sudo. When these command are run, you should get a confirmation question asking whether you want to continue, with [Y/n]. The capital letter is the default, so you can either enter in 'y' or just press ENTER.

1. The update command connects to various URLs to download a list of available packages. Periodically, new packages are introduced to Debian and Ubunutu repositories all the time, so this command refreshes the index so that it knows what packages are available and at what versions. It is a good idea to run this before you use apt commands for the day.

   ```
   sudo apt update
   ```

2. The upgrade command just updates all packages and their dependencies to their latest versions. However, this does not update packages which require the installation of *additional* packages.

```
sudo apt upgrade
```

3. The dist upgrade updates packages including those that need installation of new dependencies. So it is a good idea to run upgrade first and then dist-upgrade after.

```
sudo apt dist-upgrade
```

Installing and removing packages is easy.

1. We can install from the apt repository with

```
sudo apt install htop
```

2. We can remove it with

```
sudo apt remove htop
```

If you don't know the name of the application or package you want to install, then you can search for a keyword with apt search. Say that you want to install vim but you don't know what the actual package name is called. You can just type

```
apt search vim
```

The central location where apt gets its updates from is contained in the `/etc/apt/sources.list` file. Here is a snippet of it in my system.

```
# deb cdrom:[Ubuntu 22.04.1 LTS _Jammy Jellyfish_ - Release amd64 (20220809.1)]/
jammy main restricted

# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
# newer versions of the distribution.
deb http://us.archive.ubuntu.com/ubuntu/ jammy main restricted
# deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates main restricted
# deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates main restricted

## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
## team. Also, please note that software in universe WILL NOT receive any
## review or updates from the Ubuntu security team.
deb http://us.archive.ubuntu.com/ubuntu/ jammy universe
# deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy universe
deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates universe
# deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates universe
...
```

2.6.3 Snap and Flatpak

Other package managers that you may need to use often are snap and flatpak, which can both be installed with

```
sudo apt install snap flatpak
```

2.6.4 Wget

wget is a command-line utility used to download files from the internet. It stands for "web get."

2.7 Scheduling Tasks with Cron

2.8 Desktop Customization

GNOME

2.9 Wine

Lutris

3 Bash Scripting

3.1 Vim

Vim is a text editing software that discourages the use of the mouse. It is considered very niche but can be extremely efficient when mastered. There are many different installations of Vim, but we will use `vim-nox`, which provides support for scripting with various languages.

```
sudo apt install vim-nox
apt show vim-now
```

Now we should be able to go into our shell and type `vim` to open the text editor. What shows up is a **buffer**, which is just some free memory that we can type in. This can be a new file that is not saved or an existing file that we are editing. When opening Vim, we are in **command mode**, which allows to run the following commands:

1. Quit the editor `:q` or quit without saving `:q!`
2. Go into **insert mode** at the beginning `i` or at the end SHIFT + A
3. Get out of insert mode ESC
4. Save the file `:w test.txt`, where w stands for "write" and `test.txt` is the file name
5. If you have opened an existing file, then you can save it simply by typing `:w`
6. To undo your changes, type `u`
7. Delete the line you are on `dd`

Furthermore, if you are in command mode and you type `:`, you can press the up arrow to get your previous commands.

In command mode, we can use the arrow keys to navigate around but we can just as well do that with the H, J, K, L keys as well to save the trouble of having to reach down to the arrow keys.

3.2 Bash Scripting

Learning bash scripting allows you to automate tasks that you would normally do in a terminal in a script. We just need to open a text editor and a bash script should always start with `#!/bin/bash` at the top, known as a shebang. From this, we can enter the commands that we would type as below.

```
#!/bin/bash

ls
pwd
echo "Hello World!"
```

Now we can save it as `myscript.sh` (the suffix is not really needed if we have the shebang, but for clarity). Before we can run it, we must turn it into an executable with the command:

```
chmod +x myscript.sh
```

By going into the working directory and running `ls --color`, we can color code all the files/directories, where green stands for executables and blue are directories. Now we can run it by:

```
./myscript.sh
```

3.2.1 Variables

We can set variables directly in the terminal. Note that there should not be spaces in between the equal sign and words. Furthermore, when we try to echo a variable, we must use a dollar sign to print the *contents* of it. If we do not use it, then it will literally print the name of the variable.

```
(base) mbahng@xps-15-9500:/Desktop$ myName="Muchang"
(base) mbahng@xps-15-9500:/Desktop$ echo $myName
Muchang
(base) mbahng@xps-15-9500:~/Desktop$ echo myName
myName
```

We must have the dollar sign since there are no keywords in bash. Unlike Python, where certain words like if, else, while, etc. are reserved for certain commands and therefore cannot be used as variables, we don't have this control in bash, so we need the $. Also, if we declare a variable, it is tied to the session, so closing the terminal will delete all references.

Now we can simply apply this to a bash script. Below, we want to show one more property that distinguishes single quotes and double quotes. We want to use double quotes since single quotes will output as *raw strings*.

```
#!/bin/bash

myName="Muchang"
myAge="21"

echo "Hello, my name is $myName."
echo 'Hello, my name is $myName.'
echo "I am $myAge years old."
```

Running it gives:

```
(base) mbahng@xps-15-9500:~/Desktop$ ./myscript.sh
Hello, my name is Muchang.
Hello, my name is $myName.
I am 21 years old.
```

Now to set the **variable equal to the output of a command**, like `pwd` or `ls`, we can just store it inside parentheses.

```
(base) mbahng@xps-15-9500:/Desktop$ files=$(pwd)
(base) mbahng@xps-15-9500:/Desktop$ echo $files
/home/mbahng/Desktop
```

We can use this in combination with the `date` command to create maybe a script that greets us and returns the current time:

```
#!/bin/bash

myName="Muchang"

echo "Hello, $myName."
echo "Today's date is $(date)"
```

Running it gives:

```
(base) mbahng@xps-15-9500:~/Desktop$ ./myscript.sh
Hello, Muchang.
Today's date is Thu Jun 29 03:32:13 PM EDT 2023
```

There are special types of variables, called **environment variables**, which are all in uppercase. Due to this reason, it is important to conventionally keep your own variables to not be all uppercase, as they may get confused with the environment variables. We can view all of them with the `env` command.

```
(base) mbahng@xps-15-9500:~/Desktop$ env
SHELL=/bin/bash
SESSION_MANAGER=local/xps-15-9500:@/tmp/.ICE-unix/80970,unix/xps-15-9500:/tmp/.ICE-unix/80970
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
...
```

These are always initialized and can be accessed, so we do not need to manually reference them.

```
(base) mbahng@xps-15-9500:/Desktop$ echo $HOME
/home/mbahng
(base) mbahng@xps-15-9500:/Desktop$ echo $USER
mbahng
```

3.2.2 Basic Math

To do basic math in Python, we just needed to type in `30 + 10` in the Python shell. However, in Bash, we need to use the **evaluate expression** command, abbreviated `expr`, where we can do addition, subtraction, multiplication (which needs the escape character since `*` is a wildcard), and integer division.

```
(base) mbahng@xps-15-9500:$ expr 30 + 10
40
(base) mbahng@xps-15-9500:$ expr 30 - 10
20
(base) mbahng@xps-15-9500:$ expr 30 * 10
300
(base) mbahng@xps-15-9500:$ expr 30 / 10
3
(base) mbahng@xps-15-9500:~$ expr 30 / 8
3
```

Note that the spaces between the operation and the numbers are important.

```
(base) mbahng@xps-15-9500:~$ expr 30+10
30+10
```

Again, we can store these integers in variables and operate on them.

```
(base) mbahng@xps-15-9500:$ myNum1=100
(base) mbahng@xps-15-9500:$ echo $myNum1
100
(base) mbahng@xps-15-9500:$ expr $myNum1 + 50
150
(base) mbahng@xps-15-9500:$ myNum2=88
(base) mbahng@xps-15-9500:~$ expr $myNum1 - $myNum2
12
```

3.2.3 Conditionals

The syntax for a conditional is the following, where the conditionals are within square brackets, and we must always close the if statement with a `fi`.

```
#!/bin/bash

myNum=200

if [ $myNum -eq 200 ]
then
echo "The condition is true."
else
echo "The variable does not equal 200."
fi
```

Let us clarify some more math operators:

- A `!` means not.
- Equals (`-eq`) and not equals (`-ne`).
- Greater than (`-gt`) and less than (`-lt`).

To check if there exists a file called `myfile` within a directory, we can write:

```
#!/bin/bash

if [ -f ~/myfile ]
then
echo "The file exists"
else
echo "The file does not exist"
fi
```

We can do the same for directories by simply replacing the `-f` with a `-d`.

3.2.4 Exit Codes

An **exit code** basically tells us whether our commands in the terminal ran or encountered some error. Up until now, we can make our best judgment in deciding whether our command actually ran successfully (that is, if there was no error). To properly diagnose this, we can output the contents of the `?` variable, which actually tells us whether the input command was a success or failure. The variable is initialized every time we enter a command.

```
(base) mbahng@xps-15-9500:/Desktop$ ls -l /misc
ls: cannot access '/misc': No such file or directory
(base) mbahng@xps-15-9500:/Desktop$ echo $?
2
```

Basically, an exit code of 0 means that the command was successful, and anything other than 0 means a failure. The reason we want these exit codes is that when we automate tasks, we want scripts to run based on the exit codes of our original scripts. That is, if we get an exit code of 0, then this may trigger another bash script that emails the system administrator to let them know that something was wrong. We can create an example script that installs a package using apt as such:

```
#!/bin/bash

package=htop

sudo apt install package

if [ $? -eq 0 ]
then
echo "The installation of $package was successful."
echo "The new command is available here:"
which $package
else
echo "$package failed to install."
fi
```

To manually force the exit code to be a certain number, we can do so as below. Note that since this is an *exit* code, the script will end right at the line where we call `exit`. This is not like a break statement, where it breaks out of a loop to continue on the script. It literally terminates the script right here.

```
#!/bin/bash

sudo apt install notexist
exit 0
```

3.2.5 Outputting in a File

We can also take all the outputs of whatever commands we ran and put them into a file with the `>>` symbol.

```
mbahng@xps-15-9500:~/Desktop$ apt list >> packages.txt
```

3.2.6 Loops

The syntax for a `while` loop is similar to a conditional, with square brackets. This prints out 1 through 10, with a 0.5 second timer in between.

```
#!/bin/bash

myVar=1

while [ $myVar -le 10 ]
do
echo $myVar

bash
Copy code
# Increment by 1
myVar=$(( $myVar + 1))

# Sleep for 0.5 seconds
sleep 0.5
done
```

The syntax for a `for` loop within a range of numbers is:

```
#!/bin/bash

for i in {1..10}
do
echo $i
sleep 1
done

echo "This is outside of the for loop."
```

We can also iterate over all files with a `.log` suffix in the `logfiles` directory and compress them using tar.

```
#!/bin/bash

for file in logfiles/*.log
do
tar -czvf $file.tar.gz $file
done
```

3.2.7 Storing Scripts for Convenient Execution

To conveniently execute your scripts, it is advised to put them in your `/usr/local/bin` and to remove the `.sh` suffix. There are two reasons for this:

- `/usr/local/bin` is in the PATH environment variable, so your script can be accessed directly.
- The directory is also in a convenient location where it is not in your home directory so you can allow third parties access to this script without compromising your own privacy.

3.2.8 Environment Variables

4 Secure Shell (SSH) Protocol

5 GPU Software

```
nvidia-smi
```