Linux

Muchang Bahng

January 2024

# Contents

The following set of notes describes the everyday use of a Linux operating system. I refer to it for mainly my personal desktop, but it is also useful for working in computing clusters. Some of the commands are specific to the Arch Linux distribution (since that is what I work with), but I occasionally include those from Ubuntu and Red Hat, since I run into these distributions often in servers.

I try to organize this in a way so that one who wishes to get started in Linux can go through these notes chronologically. For now, we will assume that you have a Linux distribution installed. There are many resources beyond this book that helps you do that.

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.
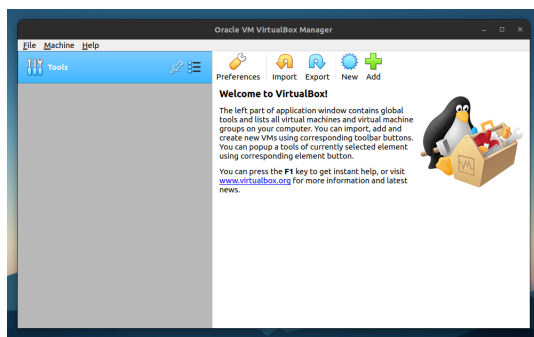

Now in order to create a VM with its own OS, you need to have the appropriate **ISO file**, which is an exact copy of an entire optical disk such as a CD, DVD, or Blu-ray archived into a single file. The essentially stores the entire software needed to operate the OS. Therefore, you should download the 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.
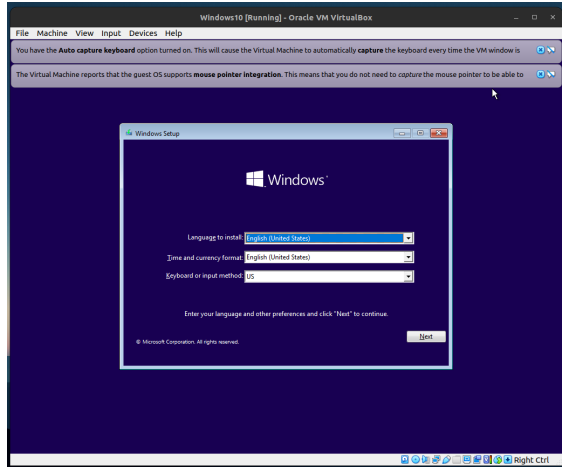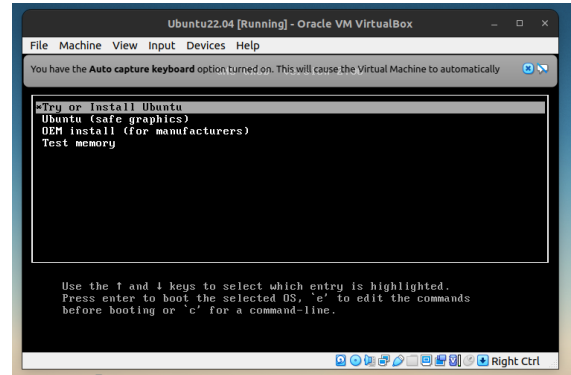
(a) Windows 10 Set Up
(b) Ubuntu 22.04 Set Up

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.
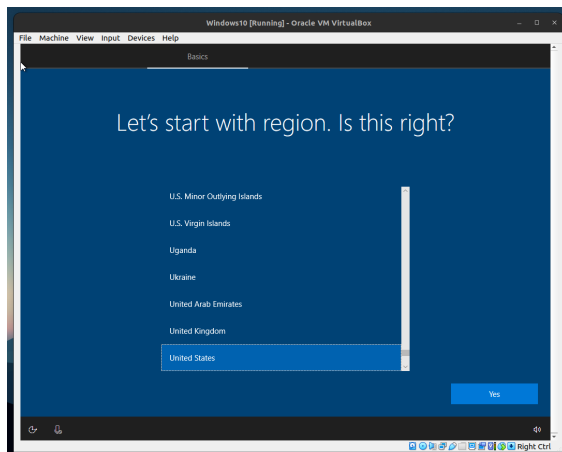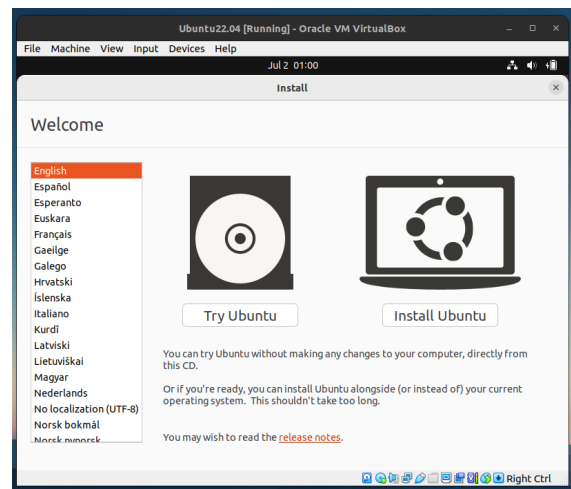
(a) Windows 10 Set Up
(b) Ubuntu 22.04 Set Up

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

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.
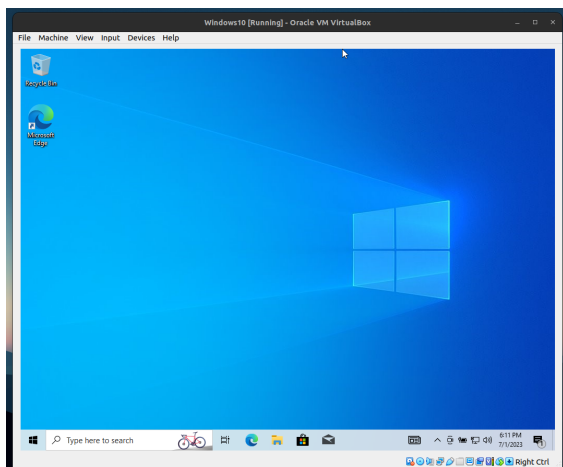
(a) Windows 10 Set Up
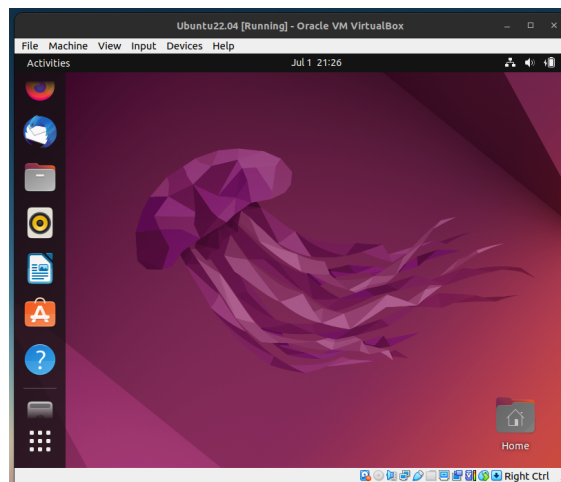
(b) Ubuntu 22.04 Set Up

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

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.

```
1   sudo apt update
2   sudo apt install snapd
3   sudo snap install --classic code
4   sudo snap install slack
5   sudo apt install git
6   sudo snap install spotify
7   sudo apt install htop
8   wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
9   sudo dpkg -i google-chrome-stable_current_amd64.deb
10  sudo apt install virtualbox
```

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.

1 Hardware

1.1 Von Neumann Architecture

It is essential to have an initial model of a computer. For this, we will first use the **von Neumann architecture**, which is the basis for most computers today. It consists of a **central processing unit** (CPU), **memory**, and an **input/output** (I/O) system.

Definition 1.1 (Memory)

The memory is where the computer stores data and instructions, which can be though of as a giant array of memory addresses, with each containing a byte. This data consists of graphical things or even instructions to manipulate other data.

Definition 1.2 (Central Processing Unit)

The CPU is responsible for taking instructions (data) from memory and executing them.
1. The CPU is composed of **registers** (different from the cache), which are small, fast storage locations. These registers can either be **general purpose** (can be used with most instructions) or **special purpose** (can be accessed through special instructions, or have special meanings/uses, or are simply faster when used in a specific way).
2. The CPU also has an **arithmetic unit** and **logic unit**, which is responsible for performing arithmetic and logical operations.
3. The CPU also has a **control unit**, which is responsible for fetching instructions from memory through the **databus**, which is literally a wire connecting the CPU and RAM, and executing them.

It executes instructions from memory one at a time and executes them, known as the **fetch-execute cycle**. It consists of 4 main operations.
1. **Fetch**: The **program counter**, which holds the memory address of the next instruction to be executed, tells the control unit to fetch the instruction from memory through the databus.
2. **Decode**: The fetched data is passed to the **instruction decoder**, which figures out what the instruction is and what it does and stores them in the registers.
3. **Execute**: The arithmetic and logic unit then carries out these operations.
4. **Store**: Then it puts the results back on the databus, and stores them back into memory.

The CPU's **clock cycle** is the time it takes for the CPU to execute one instruction. More specifically, the clock cycle refers to a single oscillation of the clock signal that synchronizes the operations of the processor and the memory (e.g. fetch, decode, execute, store), and decent computers have clock cycles of at least 2.60GHz (2.6 billion clock cycles per second).

To clarify, let us compare registers and memory. Memory is addressed by an unsigned integer while registers have names like `%rsi`. Memory is much bigger at several GB, while the total register space is much smaller at around 128 bytes (may differ depending on the architecture). The memory is much slower than registers, which is usually on a sub-nanosecond timescale. The memory is dynamic and can grow as needed while the registers are static and cannot grow.

Definition 1.3 (Input/Output Device)

The input device can read/load/write/store data from the outside world. The output device, which has **direct memory address**, can display data to the outside world.

Putting this all together, we have Figure 4.

1.2   Instruction Set Architectures

Definition 1.4 (Instruction Set Architecture)

The **ISA** or just **architecture** of a CPU is a high level description of what it can do. Some differences are listed here:
1. What instructions it can execute.
2. The instruction length and decoding, along with its complexity.
3. The performance vs power efficiency.
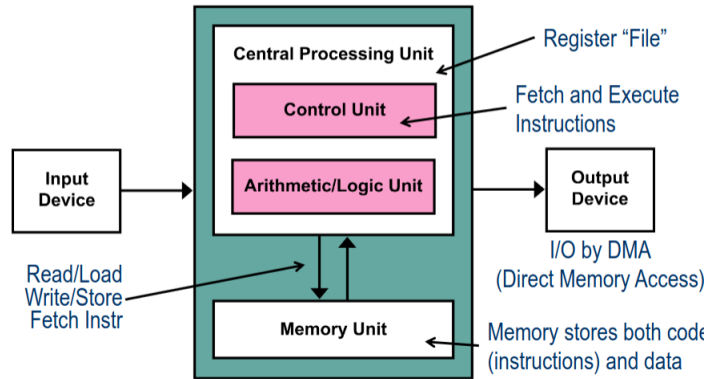
Figure 4: von Neumann Architecture

Definition 1.5 ()

ISAs can be classified into two types.
1. The **complex instruction set computer** (CISC) is characterized by a large set of complex instructions, which can execute a variety of low-level operations. This approach aims to reduce the number of instructions per program, attempting to achieve higher efficiency by performing more operations with fewer instructions.
2. The **reduced instruction set computer** (RISC) emphasizes simplicity and efficiency with a smaller number of instructions that are generally simpler and more uniform in size and format. This approach facilitates faster instruction execution and easier pipelining, with the philosophy that simpler instructions can provide greater performance when optimized.

Example 1.1 (x86 Architecture)

The x86 architecture is a CISC architecture, which is the most common architecture for personal computers. Here are important properties:
1. It is a complex instruction set computer (CISC) architecture, which means that it has a large set of complex instructions[a].
2. Byte-addressing is enabled and words are stored in little-endian format.
3. In the x86_64 architecture, registers are 8 bytes long (and 4 bytes in x86_32) and there are 16 total general purpose registers, for a total of only 128 bytes (very small compared to many GB of memory). Other special purpose registers are also documented in the wikipedia page, but it is not fully documented. The registers are listed below[b]:

```
1   %rax          # return value
2   %rbx          # callee saved
3   %rcx          # 4th argument
4   %rdx          # 3rd argument
5   %rsi          # 2nd argument
6   %rdi          # 1st argument
7   %rbp          # callee saved
8   %rsp          # stack pointer
9   %r8           # 5th argument
10  %r9           # 6th argument
11  %r10          # scratch register
12  %r11          # scratch register
13  %r12          # callee saved
14  %r13          # callee saved
```

```
15  %r14        # callee saved
16  %r15        # callee saved
```

Example 1.2 (ARM Archiecture)

Mainly in phones, tablets, laptops.

Example 1.3 (MIPS Architecture)

MIPS is a RISC architecture, which is used in embedded systems such as digital home and networking equipment.

This is a large overview of the different architectures, but Arch Linux states on their website that they have *official packages optimized for the x86-64 architecture.*[1]

Furthermore, by running **cat /proc/cpuinfo**, you can see the specs of each CPU core you have. This includes the **model name** (clock cycle), **cache size**, **flags**, and **microcode**. The flags are the most important, since they tell you what features your CPU has.[2]

1. **lm**: 64 bit architecture.

2. **vmx** (Intel) or **svm** (AMD): Hardware virtualization .

3. **aes**: Accelerate AES encryption.

4. **fpu**: Floating Point Unit, which is used for floating point operations.

5. **vme**: Virtual 8086 mode enhancements, which is used for virtualization.

6. **de**: Debugging extensions, which is used for debugging.

7. **pse**: Page Size Extensions, which is used for larger page sizes.

8. **tsc**: Time Stamp Counter, which is used for timing.

9. **msr**: Model Specific Registers, which is used for model specific operations.

10. **mce**: Machine Check Exception, which is used for error checking.

11. **pae**: Physical Address Extensions, which is used for larger memory.

12. **mce**: Machine Check Exception, which is used for error checking.

1.3 Assembly in x86_64

Definition 1.6 (Instruction)

An instruction is a single line of assembly code. It consists of some instruction followed by its (one or more) operands. The instruction is a mnemonic for a machine language operation (e.g. `mov`, `add`, `sub`, `jmp`, etc.). The **size specifier** can be appended to this instruction mnemonic to specify the size of the operands.

    1. **b** (byte) for 1 byte

---

[a]https://en.wikipedia.org/wiki/X86_instruction_listings

[b]Older x86_32 architecture has 8 general purpose registers with the `r` replaced by a `e`, e.g. `eax` instead of `rax`.

[1]https://archlinux.org/

[2]The entire list of flags and what they can do is mentioned in the Arch kernel source code, which is a good reference: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/x86/include/asm/cpufeatures.h

2. **w** (word) for 2 bytes

3. **l** (long) for 4 bytes

4. **q** (quad word) for 8 bytes

Note that due to backwards compatibility, word means 2 bytes in instruction names. Furthermore, the maximum size is 8 bytes since that is the size of each register in x86_64. An operand can be of 3 types, determined by their **mode of access**:

1. **Immediate addressing** is denoted with a `$` sign, e.g. a constant integer data `$1`.

2. **Register addressing** is denoted with a `%` sign with the following register name, e.g. `%rax`.

3. **Memory addressing** is denoted with the hexadecimal address in memory, e.g. `0x034AB`.

Like higher level programming languages, we can perform operations, do comparisons, and jump to different parts of the code. Instructions can be generally categorized into three types:

1. **Data Movement**: These instructions move data between memory and registers or between the registery and registery. Memory to memory transfer cannot be done with a single instruction.

```
%reg = Mem[address]      # load data from memory into register
Mem[address] = %reg      # store register data into memory
```

2. **Arithmetic Operation**: Perform arithmetic operation on register or memory data.

```
%reg = %reg + Mem[address]      # add memory data to register
%reg = %reg - Mem[address]      # subtract memory data from register
%reg = %reg * Mem[address]      # multiply memory data to register
%reg = %reg / Mem[address]      # divide memory data from register
```

3. **Control Flow**: What instruction to execute next.

```
jmp label       # jump to label
je label        # jump to label if equal
jne label       # jump to label if not equal
jg label        # jump to label if greater
jl label        # jump to label if less
call label      # call a function
ret             # return from a function
```

Now unlike compiled languages, which are translated into machine code by a compiler, assembly code is translated into machine code through a two-step process. First, we **assemble** the assembly code into an **object file** by an **assembler**, and then we **link** the object file into an executable by a **linker**. Some common assemblers are **NASM** (Netwide Assembler) and **GAS**/**AS** (GNU Assembler), and common linkers are **ld** (GNU Linker) and **lld** (LLVM Linker), both installable with **sudo pacman -S nasm ld**.

Definition 1.7 (mov)

Let's talk about the `mov` instruction. A good diagram to see is the following:

Parantheses indicate that we are using a pointer dereference.

Definition 1.8 (int)

The `int` instruction is used to generate a software interrupt. It is often used to invoke a system call.

Definition 1.9 (ret)

The `ret` instruction is used to return from a function. It returns the value in the `%rax` register.

Example 1.4 (Swap Function)

In **gdb**, we may have a function that swaps two integers.

```
swap:
  movq (%rdi), %rax
  movq (%rsi), %rdx
  movq %rdx, (%rdi)
  movq %rax, (%rsi)
  ret
```

which is the assembly code for the following C code.

```
void swap(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

1.4   Graphics Drivers

Note that one type of data we must store on memory is the individual pixels in a computer screen. Say that in a 1920×1080 resolution computer, there are about $1920 \times 1080 \times 3 \approx 2$ million bytes of data that we have to store. This isn't that much data (only 2MB), but we must update it quite fast since our screens are always updating. This is why all computer which have a GUI comes with a built-in graphics driver. To see the GPU hardware specifications, install **lshw**.

Definition 1.10 (Graphics Processing Unit)

The **GPU** is a specialized processing unit that is designed to handle the rendering of images and videos. It is designed to handle the rendering of images and videos, and is optimized for parallel processing. Like the CPU, it has some common metrics:
1. **Clock Speed**: The speed at which the GPU can execute instructions. This is usually measured in MHz or GHz.
2. **Memory**: The amount of memory that the GPU has. This is usually measured in GB.
3. **Memory Bandwidth**: The speed at which the GPU can read and write to its memory. This is usually measured in GB/s.
4. **Cores**: The number of cores that the GPU has. This is usually measured in thousands, which allows for parallel processing.

You can check which GPUs you have by running `lspci | grep VGA` or `neofetch`. There are generally two types of GPUs:
1. **Integrated GPU**: This type of GPU is built into the same chip as the CPU (Central Processing Unit). It shares resources with the CPU, including memory, which can lead to reduced performance for graphics-intensive tasks. However, its integrated nature makes it more power-efficient and cost-effective.
2. **Discrete GPU**: This is a separate component from the CPU and comes with its own RAM (usually called VRAM or Video RAM). It is typically installed in a dedicated slot on the motherboard. Because it operates independently of the CPU, a discrete GPU can offer significantly

better performance for graphics processing, gaming, or deep learning.

Definition 1.11 (Monitor)

Furthermore, your computer monitor, which actually displays these pixels to you, must also have metrics that match the GPU. Some properties:
1. The **resolution** is the number of pixels that the monitor can display, and is usually measured in pixels.
2. The **refresh rate** is the number of times the monitor can refresh the image on the screen per second, and is usually measured in Hz.

To see these metrics for all monitors connected to your computer, run **xrandr**, which lists all the resolutions and possible refresh rates for each resolution.

Definition 1.12 (Graphics Driver)

In order for your operating system to communicate with your GPU, you need a **graphics driver**. This is a piece of software that allows the operating system to communicate with the GPU. There are two main types of graphics drivers:
1. **Open Source Drivers**: These are drivers that are developed and maintained by the open source community. They are usually included with the Linux kernel, and are generally stable and reliable.
2. **Proprietary Drivers**: These are drivers that are developed and maintained by the GPU manufacturer. They are usually not included with the Linux kernel, and are generally more feature-rich and performant than open source drivers.

Intel drivers are open source, but Nvidia drivers are proprietary (which is why Linus Torvalds has beef with Nvidia).[a]

Some popular graphics drivers include **mesa** for Intel and **nvidia** drivers for NVIDIA.

1.4.1   Multiple GPUs

Everything is pretty straightforward when you have one graphics card, but when you have multiple graphics cards, you have to specify which one you want to use. If you want to only use one GPU, you can just disable the other one in the BIOS. However, if you have an Intel/Nvidia dual driver and want to use both, install **Nvidia Optimius** (for Ubuntu, it is supported through **nvidia-prime**).[3],[4]

Now make sure that the systemd daemon is running, and you can call `optimus-manager -switch hybrid` to enable hybrid graphics. This will log you out.

1.5   Peripheral Devices

Peripheral devices refer to other devices outside of the motherboard, including mice, keyboards for input, monitors, printers, network managers, and usb ports. Even the GPU is considered a peripheral device. These must be connected to the motherboard in some way to be managed by the operating system, and similar to the databus connecting the CPU and memory, there are buses that connect the motherboard and these peripheral devices.

---

[a]A video of Linus Torvalds saying "fuck you" to Nvidia: https://www.youtube.com/watch?v=iYWzMvlj2RQ
[3]This wiki article (https://github.com/Askannz/optimus-manager/wiki) provides a good overview of this matter.
[4]Installation instructions here: https://github.com/Askannz/optimus-manager?tab=readme-ov-file

> **Definition 1.13 (PCI Bus)**
>
> The **PCI (Peripheral Component Interconnect)** bus is a high-speed bus that connects the motherboard to peripheral devices. It is used to connect devices like network cards, sound cards, and graphics cards to the motherboard. PCI buses operated based on the PCI standard, which is a set of specifications that define the physical and electrical characteristics of the bus.

The command to use to enumerate all PCI devices is `sudo lspci` (with `-v` for verbose).

```
1   00:00.0 Host bridge: Intel Corporation 10th Gen Core Processor
2   00:01.0 PCI bridge: Intel Corporation 6th-10th Gen Core Processor
3   00:02.0 VGA compatible controller: Intel Corporation CometLake-H
4   00:04.0 Signal processing controller: Intel Corporation Xeon
5   00:08.0 System peripheral: Intel Corporation Xeon E3-1200 v5/v6
6   00:12.0 Signal processing controller: Intel Corporation Comet
7   00:13.0 Serial controller: Intel Corporation Device 06fc
8   00:14.0 USB controller: Intel Corporation Comet Lake USB 3.1
9   00:14.2 RAM memory: Intel Corporation Comet Lake PCH Shared
10  00:14.3 Network controller: Intel Corporation Comet Lake PCH
11  00:15.0 Serial bus controller: Intel Corporation Comet Lake
12  00:15.1 Serial bus controller: Intel Corporation Comet Lake
13  00:16.0 Communication controller: Intel Corporation Comet
14  00:1c.0 PCI bridge: Intel Corporation Device 06b8 (rev f0)
15  00:1c.6 PCI bridge: Intel Corporation Device 06be (rev f0)
16  00:1d.0 PCI bridge: Intel Corporation Comet Lake PCI Express
17  00:1e.0 Communication controller: Intel Corporation Comet Lake
18  00:1f.0 ISA bridge: Intel Corporation Device 068e
19  00:1f.3 Audio device: Intel Corporation Comet Lake PCH cAVS
20  00:1f.4 SMBus: Intel Corporation Comet Lake PCH SMBus Controller
21  00:1f.5 Serial bus controller: Intel Corporation Comet Lake
22  01:00.0 3D controller: NVIDIA Corporation TU117M [GeForce GTX 1650
23  02:00.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
24  03:00.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
25  03:01.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
26  03:02.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
27  03:04.0 PCI bridge: Intel Corporation JHL7540 Thunderbolt 3 Bridge
28  04:00.0 System peripheral: Intel Corporation JHL7540 Thunderbolt
29  38:00.0 USB controller: Intel Corporation JHL7540 Thunderbolt 3
30  6c:00.0 Unassigned class [ff00]: Realtek Semiconductor Co., Ltd.
31  6d:00.0 Non-Volatile memory controller: Samsung Electronics Co
```

Figure 5: This is the following output of lspci on my personal computer.

1.6 System Hardware

1.6.1 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 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>
```

```
# Lists all drives on your computer
# DeviceID = drive letter
# DriveType: 2=Removable, 3=Fixed local, 4=Network
# FreeSpace and Size in bytes
wmic LOGICALDISK LIST BRIEF # Windows cmd

```

```
7   # Outputs drive model and status
8   # Status OK = good health
9   # Status 'Pred Fail' = potential crash warning
10  wmic diskdrive get status, model # Windows cmd
11
12  # Checks file system and provides drive issues summary
13  # Non-zero bad sectors require technical help
14  chkdsk c: # Windows cmd
15
16  # Opens window showing all drive types
17  dfrgui # Windows cmd
18
19  # Changes working drive (e.g. d:)
20  <letter>: # Windows cmd
```

We demonstrate some of the commands here.

```
1   C:\Users\bahng>wmic LOGICALDISK LIST BRIEF
2     DeviceID  DriveType  FreeSpace      ProviderName  Size           VolumeName
3     C:        3          838628864000                 1003327844352  OS
4     G:        3          71506178048                  107374182400   Google Drive
5
6   C:\Users\bahng>wmic diskdrive get status, model
7     Model                    Status
8     PM9A1 NVMe Samsung 1024GB  OK
```

1.6.2 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 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)

```
1   # Outputs relevant information about the RAM.
2   wmic MEMORYCHIP (get BankLabel, DeviceLocator, MemoryType, TypeDetail, Capacity, Speed) #
        Windows cmd
3
4   # Outputs a list of all specifications of each memory stick.
5   wmic memoryship list full # Windows cmd
6
7   # Outputs the total RAM memory of your computer.
8   systeminfo | findstr /C:"Total Physical Memory" # Windows cmd
9
10  # Outputs the available RAM memory of your computer.
11  systeminfo |find "Available Physical Memory" # Windows cmd
```

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).

```
1   C:\Users\bahng>wmic MEMORYCHIP get BankLabel, DeviceLocator, MemoryType, TypeDetail, Capacity,
        Speed
2   BankLabel   Capacity      DeviceLocator  MemoryType  Speed  TypeDetail
3               8589934592  DIMM A          0           3200   128
4               8589934592  DIMM B          0           3200   128
```

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

```
1   # Outputs a list of all specifications of the CPU.
2   wmic cpu list full # Windows cmd
3
4   # Outputs relevant information about the CPU.
5   wmic cpu (get caption, deviceid, name, numberofcores, maxclockspeed, status) # Windows cmd
```

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.

1.7   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 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**.

1.7.1   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

$$\texttt{C: \textbackslash Users\textbackslash bahng\textbackslash AppData\textbackslash Local\textbackslash Programs\textbackslash Python\textbackslash Python39}$$

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

$$\texttt{C:\textbackslash windows\textbackslash py.exe}$$

1.7.2   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   Filesystems

Before we get into anything, even the loading of the firmware or the operating system kernel, we must talk about the hardware and how a computer stores data. Data, whether it is in memory or some disk, is just a bunch of sequences of bits. A **drive** is a physical device that can store data. A **partition** is a logical division of a drive, and a **filesystem** is a way to organize data on a drive. For example, if I have a 1TB SSD, I can

run it as a single partition, or I can divide it into two partitions, one for a Windows operating system and another for a Linux operating system. A filesystem is a bit more confusing, so here are some examples.

Example 2.1 (Linux Filesystems)

Listed.
1. **ext4**: The most common filesystem for Linux.
2. **XFS**: Designed for high performance and scalability, often used in enterprise environments for large-scale storage.
3. **btrfs**: A modern filesystem that offers advanced features like snapshots, dynamic inode allocation, and integrated device management for better data reliability and performance.
4. **zfs**: Originally developed by Sun Microsystems for Solaris, ZFS is known for its data integrity, support for enormous storage capacities, and features like snapshots, copy-on-write, and built-in data compression.

Example 2.2 (Windows Filesystems)

Listed.
1. **NTFS (New Technology File System)**: The standard filesystem for Windows operating systems, supporting file permissions, encryption, and large file sizes.
2. **FAT32 (File Allocation Table 32)**: An older filesystem with wide compatibility across different operating systems, including Windows, macOS, and various Linux distributions, though it has limitations on file and partition sizes.
3. **exFAT (Extended File Allocation Table)**: Designed to be a lightweight filesystem similar to FAT32 but without its limitations, exFAT is used for flash drives and external hard drives due to its support for larger files and compatibility.

Example 2.3 (MacOS Filesystems)

Listed.
1. **APFS (Apple File System)**: The default filesystem for macOS, iOS, and other Apple operating systems since 2017, designed for SSDs and featuring strong encryption, space sharing, and fast directory sizing.
2. **HFS+ (Hierarchical File System Plus)**: Also known as Mac OS Extended, it was the primary filesystem for Mac computers before APFS, supporting journaling for data integrity.

When your computer boots up, it needs to know where to find the operating system kernel. This is done by mounting the filesystems. The **mount point** is the directory where the filesystem is attached to the system. The **root filesystem** is the filesystem that contains the operating system kernel.

Depending on your hardware specs, you may have multiple drives. To list all drives and their partitions, run **lsblk**. The type determines whether it is a disk or a partitions, and the mountpoints determine where the partitions are mounted. Furthermore, the RO indicates whether this is a HDD (1) or SSD (0).

The **swap** partition is a special type of partition that is used as a temporary storage area for the operating system. It is used when the system runs out of RAM.

For a more detailed view on what the partitions consist of, you can run **fdisk -l**.

```
1  Disk /dev/nvme0n1: 953.87 GiB, 1024209543168 bytes, 2000409264 sectors
2  Disk model: PM9A1 NVMe Samsung 1024GB
3  Units: sectors of 1 * 512 = 512 bytes
4  Sector size (logical/physical): 512 bytes / 512 bytes
5  I/O size (minimum/optimal): 512 bytes / 512 bytes
6  Disklabel type: gpt
7  Disk identifier: 26D88CE9-B388-4CF1-856C-14D5EEB0C143
```

```
1    NAME         MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
2    zram0        254:0   0      4G  0 disk [SWAP]
3    nvme0n1      259:0   0 953.9G  0 disk
4      nvme0n1p1 259:1   0    240M  0 part
5      nvme0n1p2 259:2   0    128M  0 part
6      nvme0n1p3 259:3   0 309.4G  0 part
7      nvme0n1p4 259:4   0    990M  0 part
8      nvme0n1p5 259:5   0   16.7G  0 part
9      nvme0n1p6 259:6   0    1.4G  0 part
10     nvme0n1p7 259:7   0    500M  0 part /boot
11     nvme0n1p8 259:8   0    4.7G  0 part [SWAP]
12     nvme0n1p9 259:9   0  619.9G  0 part /
```

Figure 6: This is the following output on my personal computer.

```
8
9    Device              Start         End     Sectors   Size Type
10   /dev/nvme0n1p1       2048      493567      491520   240M EFI System
11   /dev/nvme0n1p2     493568      755711      262144   128M Microsoft reserved
12   /dev/nvme0n1p3     755712   649658367   648902656 309.4G Microsoft basic data
13   /dev/nvme0n1p4 1960380416 1962407935     2027520   990M Windows recovery environment
14   /dev/nvme0n1p5 1962407936 1997441023    35033088  16.7G Windows recovery environment
15   /dev/nvme0n1p6 1997443072 2000377855     2934784   1.4G Windows recovery environment
16   /dev/nvme0n1p7  649658368   650682367     1024000   500M EFI System
17   /dev/nvme0n1p8  650682368   660447231     9764864   4.7G Linux swap
18   /dev/nvme0n1p9  660447232  1960380415  1299933184 619.9G Linux filesystem
```

As you can see here, my single disk has 9 partitions.

1. The first EFI system (1) or the Microsoft reserved (2) partition contains the Windows operating system kernel.

2. The Microsoft basic data (3) partition contains the Windows files.

3. The Windows recovery environment (4, 5, 6) is a partition that contains the Windows recovery environment, which are partitions set aside by the manufacturer to hold an image of your system before it was shipped from the factory.

4. The EFI system (7) partition contains the Linux operating system kernel, which is required to load the operating system.

5. The Linux swap (8) partition is a partition that contains the Linux swap.

6. The Linux filesystem (9) is a partition that contains the actual Linux operating system itself, along with all your files.

2.1   Mounting

You can further go into the /dev directory to see the devices that are mounted, e.g. the /dev/nvme0n1p9 is the device that is mounted on the root directory, and most of these files are either device files (which are special files that provide an interface to hardware devices, allowing software and users to interact with them as if they were normal files) or symlinks.

The **mount** command is used to attach a filesystem to the system's directory tree. The **umount** command is used to detach a filesystem from the system's directory tree.

1. **Mounting a filesystem**: The general syntax is `mount -t type device dir`. For example, to mount the /dev/nvme0n1p9 to the root directory, you can run `mount -t ext4 /dev/nvme0n1p9 /mnt`.

2. **Unmounting a filesystem**: The general syntax is `umount dir`. For example, to unmount the root directory, you can run `umount /mnt`.

When the computer boots up, it must automatically mount the specific filesystems. This is configured in the **fstab** file.

> **Definition 2.1 (fstab)**
>
> The **fstab** file is a system configuration file that contains information about filesystems. It is located at `/etc/fstab`. It is used to define how disk partitions, various other block devices, or remote filesystems should be mounted into the filesystem. Each line in the file contains six fields, separated by whitespace. The fields include:
> 1. **Filesystem**: The block device or remote filesystem to be mounted. This can be the UUID (Universally Unique Identifier), the label, or the traditional device name (like /dev/sda1) that specifies which device or partition is being referred to.
> 2. **Mount Point**: The directory where the filesystem should be mounted.
> 3. **Type**: The type of the filesystem, e.g. ext4, vfat, swap, etc.
> 4. **Options**: Mount options for the filesystem, e.g. `rw` for read-write, `ro` for read-only, `noexec` to prevent execution of binaries, etc.
> 5. **Dump**: A number used by the **dump** command to determine whether the filesystem should be backed up. It is often set to 0 to disable backups.
> 6. **Pass**: A number used by the **fsck** command to determine the order in which filesystems should be checked. Root filesystems should have this set to 1, and other filesystems should either be 2 (to check after the root) or 0 (to disable checking).

```
# Static information about the filesystems.
# See fstab(5) for details.

# <file system> <dir> <type> <options> <dump> <pass>
# /dev/nvme0n1p9
UUID=abcfef03-bfae-4d1f-b463-fd6538f18a41 / ext4 rw,relatime 0 1
# /dev/nvme0n1p7
UUID=150D-7A67 /boot vfat rw,relatime,fmask=0077,dmask=0077,codepage=437,
    iocharset=ascii,shortname=mixed,utf8,errors=remount-ro  0 2
# /dev/nvme0n1p8
UUID=5c191f65-b016-475d-b04a-5b7c89bda31d none swap defaults 0 0
```

Figure 7: My personal fstab file.

2.1.1 Mounting a Remote Disk

It is actually possible to mount a folder on a server into your local machine. To do this, you use **sshfs** to mount a remote directory over SSH. The general syntax is `sshfs user@host:/remote/dir /local/dir` to mount and `fusermount -u /local/dir` to unmount.

2.2 Maintence

2.2.1 SSD

As soon as your write or delete bits from the SSD (e.g. when you're deleting a file), it degrades the speed of the read/write. To alleviate the effects, you can use TRIM, which is a command that allows the operating system to inform the SSD which blocks of data are no longer considered in use and can be wiped internally. It can be downloaded as a part of the `util-linux` package, which provides the systemd services **fstrim.timer** and **fstrim.service**. It is recommended to use weekly trims rather than continuous trims.

2.2.2 Filesystem

Occasionally, you may have a corrupt partitions, whether it is your boot or root directory. In this case, you should use the **fsck** command to check and repair a filesystem. The general steps are:

1. unmount the specific partition you want (identified with **lsblk**) using `sudo umount /dev/partition`.

2. run `sudo fsck -t type device` (or for specific filesystem types like vfat you can be a bit more specific by running `sudo fsck.vfat /dev/partition`) to check the filesystem and fix any changes.

3. mount the specific partition back using `sudo mount /dev/partition`.

2.3 Modifying Partitions

Modifying partitions require specialized software. Partitioning can be done using two main partitioning schemes **GPT** (the modern one) and **MBR** (legacy). The **parted** utility gives detailed info on your partitions. To see which scheme you have, just run `sudo parted -l`, where the output can be shown in Figure 8.

```
1   Model: PM9A1 NVMe Samsung 1024GB (nvme)
2   Disk /dev/nvme0n1: 1024GB
3   Sector size (logical/physical): 512B/512B
4   Partition Table: gpt
5   Disk Flags:
6
7   Number  Start    End      Size    File system    Name                         Flags
8    1      1049kB   253MB    252MB   fat32          EFI system partition         boot, esp
9    2      253MB    387MB    134MB                  Microsoft reserved partition msftres
10   3      387MB    333GB    332GB   ntfs           Basic data partition         msftdata
11   7      333GB    333GB    524MB   fat32                                       boot, esp
12   8      333GB    338GB    5000MB  linux-swap(v1)                              swap
13   9      338GB    1004GB   666GB   ext4
14   4      1004GB   1005GB   1038MB  ntfs                                        hidden, diag
15   5      1005GB   1023GB   17.9GB  ntfs                                        hidden, diag
16   6      1023GB   1024GB   1503MB  ntfs                                        hidden, diag
17
18
19   Model: Unknown (unknown)
20   Disk /dev/zram0: 4295MB
21   Sector size (logical/physical): 4096B/4096B
22   Partition Table: loop
23   Disk Flags:
24
25   Number  Start   End     Size    File system     Flags
26    1      0.00B   4295MB  4295MB  linux-swap(v1)
```

Figure 8: Output of `sudo parted -l` on my own machine.

It is important to know which partition scheme you should use.

1. To dual-boot with Windows (both 32-bit and 64-bit) using Legacy BIOS, the MBR scheme is required.

2. To dual-boot Windows 64-bit using UEFI mode instead of BIOS, the GPT scheme is required.

3. If you are installing on older hardware, especially on old laptops, consider choosing MBR because its BIOS might not support GPT.

4. If you are partitioning a disk that is larger than 2TB, you need to use GPT.

5. It is recommended to always use GPT for UEFI boot, as some UEFI implementations do not support booting to the MBR while in UEFI mode.

3   Firmware

Let us go through the steps of a booting (bootstrapping) process. Administrators have little direct, interactive control over most of the steps required to boot a system, but they can modify bootstrap configurations by editing config files or system startup scripts.

1. **Power On**: You power on the machine.

2. **Load firmware from NVRAM**: You want to be able to identify the specific piece of hardware to load your operating system in. The firmware is a permanent piece of software that does this.

3. **Probe for hardware**: We look for hardware that is on the computer.

4. **Select boot device (disk, network, etc.)**: We select the storage device that we want to load the operating system on.

5. **Identify EFI system partition**:

6. **Load boot loader (e.g. GRUB)**: A software that allows you to identify and load the proper OS kernel is provided.

7. **Determine which kernel to boot**: You choose which kernel you want to load.

8. **Load kernel**: The OS kernel is identified and loaded into the boot device.

9. **Instantiate kernel data structure**:

10. **Start init/systemd as PID 1**:

11. **Exectute startup scripts**:

12. **Running system**: You now have a running system!

Right above the hardware, the **system firmware**, is a piece of software that is executed whenever the computer boots up.

1. **Power Supply Activation**: Once the computer is turned on, the power supply begins to provide electricity to the system's components. One of the first signals generated is the "Power Good" signal, indicating that the power supply is stable and at the correct voltages.

2. **CPU Reset**: Upon receiving the "Power Good" signal, the CPU resets and starts its operations. The CPU is designed to start executing instructions from a predefined memory address, which is hardwired into the CPU. This address, stored in ROM, contains the starting point of the firmware.**Read Only Memory** is simply another type of computer memory that stores permanent data and instructions for the device to start up.

3. **Predefined Memory Address**: For BIOS systems, the CPU begins executing code at the firmware entry point located in the system's ROM (Read-Only Memory). In UEFI systems, the process is similar, but the UEFI firmware provides more functionalities and a more flexible pre-boot environment.

4. **POST (Power on Self Test)**: The firmware conducts a series of diagnostic tests to ensure that essential hardware components like RAM, storage devices, and input/output systems are functioning correctly. This stage is critical for verifying system integrity before loading the operating system.

To be honest, there is not a lot that the user can control here with just software. The firmware is a permanent piece of software that is executed whenever the computer boots up, which makes it relatively safe from tampering. If your computer fails to boot up, the most fundamental reason may be a firmware problem. However, we're not screwed yet.

Most firmware offers a user interface which can be accessed by pressing the F2, F11, F12, or some combination of magic keys at the instant the system first powers on. Depending on what computer model you have, you may have some control of basic functionalities.
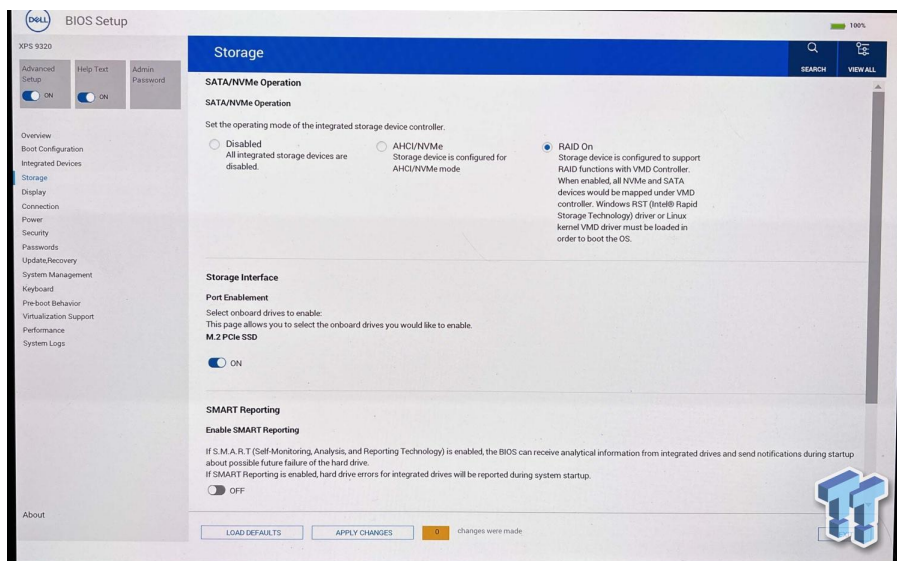

Figure 9: Firmware of Dell XPS 13 9320

Some important functionalities you can do with the firmware are:

1. Determine the boot order of the devices, usually by prioritizing a list of available options (e.g. try to boot from a DVD drive, then a USB, then the hard disk).

2.

The **BIOS**, which stands for **Basic Input/Output System**, has been used traditionally. It 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.

A more formalized and modern standard called **EFI** (**Extensible Firmware Interface**) has replaced it, and it has been revised to the **UEFI** (**Unified Extensible Firmware Interface**) standard, but we can treat EFI and UEFI as equivalent in most cases. Fortunately, most UEFI systems can fall back to a legacy BIOS implemlentation if the operating system they're booting doesn't support UEFI. Since we're likely to encounter boot firmware systems, it's worthwhile to go into both of them.

3.1   Updating Firmware

The first thing you should do when you're having trouble with firmware is use **fwupd**, which is a daemon that handles firmware updates. It is a simple daemon to allow session software to update device firmware on your local machine. Upon installation, it creates a systemd agent on `/lib/systemd/system/fwupd.service`. It does not start automatically. I have used this to update my firmware, which saved a lot of booting errors, with instructions accessed in this link.

3.2   Modifying UEFI Variables

You can directly examine and modify UEFI variables on a running system with the `efibootmgr` command. You get a following summary of the configuration:

```
1  BootCurrent: 0005
2  Timeout: 0 seconds
3  BootOrder: 0005,0001,0002,0000,0003,0004
4  Boot0000* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1    ...
5  Boot0001* ubuntu  HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd6026c1,0x800,0x780...
6  Boot0002* Windows Boot Manager   HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd60...
7  Boot0003* Linux Firmware Updater HD(1,GPT,ede98b7e-75ad-452e-ab47-...
8  Boot0004* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1 2 PciRoot(0x0)/...
9  Boot0005* Linux Boot Manager  HD(7,GPT,2d28b70f-725b-4ca3-98d4-25f5c83fc00e...
```

It shows you which disk you are currently booted into, the boot order that is currently configured, and information about each of the disks. You can use a GUI to do this as well. You can press a certain key when booting (F2 on my Dell XPS15 9500) to enter the **BIOS setup**.
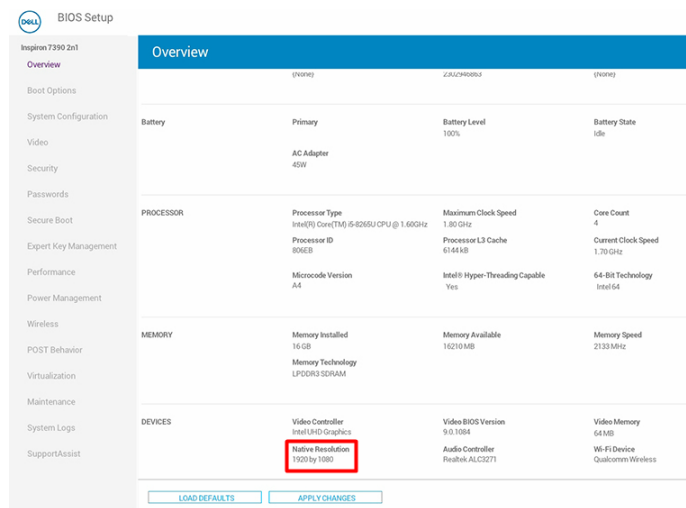
Figure 10: The BIOS setup can look very different depending on the computer but looks like this for me.

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

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.
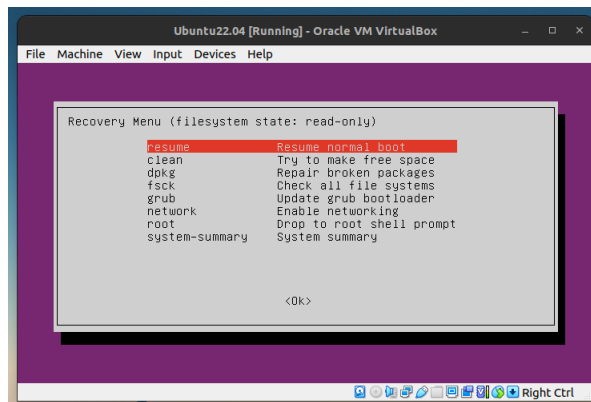
This gives us a list of options that we can take to fix the system. Every setting except root is automatically done. The root command gives us root privileges (no sudo is needed). This also means we have full access to all files, and we may cause irreversible damage to our system if we made a mistake. 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:

```
1   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.

4 Bootloaders

Once the firmware is loaded, which probes the system to find the hardware, it must load the operating system kernel. This is the job of the boot loader.

Definition 4.1 (Boot Loader, Boot Manager)

The **bootloader** is another critical piece of software that allows you to identify and load the proper operating system kernel. If it also provides an interactive menu with multiple boot choices, then it is often called a **boot manager**.

In modern systems which support UEFI (not the legacy BIOS), you must configure your partitions so that there exists an EFI partition (at `/boot`) that contains this bootloader.

EFI bootloaders usually have a `.efi` extension, and it is crucial that you know where the bootloaders are in your system in case they go missing or are corrupt. To see the configuration, you can run **efibootmgr** (with verbose), which gives you information on several things:

1. It scans the entire system for EFI bootloaders and lists them.

2. It lists the locations of the EFI bootloaders. It starts off which what partition they are in, and then lists the directory where the bootloader is located. `BootX64.efi` is the Windows bootloader and `grubx64.efi` is the GRUB bootloader. For example, you may have a bootloader at `(partition 7)/boot/efi/EFI/Boot/bootx64.efi`.

3. It lists the boot order, which is the order in which the bootloaders are loaded. In case a boot loader fails to load, the next one is loaded. Therefore, if you have an arch linux bootloader that is corrupt, and the next in line is the Windows bootloader, you will automatically boot into Windows. You can also set the boot order in the BIOS.

In case you can't boot in, you can always get an Arch ISO burned in on a thumb drive, boot into it, mount the relevant partitions containing the Arch bootloader and the root directory, and then chroot into the root directory to modify files.

4.1   GRUB

The way that these kernels can be loaded can be configured through the bootloader, and the most popular boot manager is **GRUB**, the **Grand Unified Bootloader**. GRUB, developed by the GNU project, is the default loader on most Linux distributions. There is an old version called GRUB legacy and the more modern GRUB 2. Most people refer to GRUB 2 and simply GRUB. FreeBSD, which is another complete (non-Linux) OS, have their own boot loader, but GRUB is compatible with it. Therefore, for dual-boot or triple-boot systems that have multiple kernels, GRUB is the go-to bootloader for loading any of them.


```
                    GNU GRUB   version 2.00-5ubuntu2

 ┌──────────────────────────────────────────────────────────────────────────┐
 │ Ubuntu                                                                     │
 │ Advanced options for Ubuntu                                                │
 │ Memory test (memtest86+)                                                   │
 │ Memory test (memtest86+, serial console 115200)                           │
 │                                                                            │
 │                                                                            │
 │                                                                            │
 │                                                                            │
 │                                                                            │
 │                                                                            │
 │                                                                            │
 └──────────────────────────────────────────────────────────────────────────┘

      Use the ↑ and ↓ keys to select which entry is highlighted.
      Press enter to boot the selected OS, `e' to edit the commands
      before booting or `c' for a command-line.
```

Figure 11: GRUB menu on my screen. 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.

As a critical piece of software, we would expect its configuration files to be in the NVRAM, but GRUB understands most of the filesystems in common use and can find its way into the root filesystem on its own. Therefore, we can read its configuration from a regular text file, kept in `/boot/grub/grub.cfg`. Changing the boot configuration is as simple as updating the `grub.cfg` file, 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.

5   Systemd

A **process** is really any program that is running on your computer. A **daemon** is a background process that runs continuously, performing specific tasks even when no user is logged in.

Once the kernel has been loaded and completed its initialization process, it creates a collection of *spontaneous* (as in the kernel starts them automatically) processes in user space. They're really part of the kernel implementation and don't necessarily correspond to programs in the filesystem. They're not configurable and they don't require administrative attention. These processes can be monitored with the commands `ps`, `top`, or `htop`.

The most important process is the init process, with a system PID of 1 and with special privileges. It is used to get the system running and for starting other processes.

1. Setting the name of the computer

2. Setting the time sone

3. Checking disks with `fsck`

4. Mounting filesystems

5. Removing old files from the `/tmp` directory

6. Configuring network interfaces

7. Configuring packet filter

8. Starting up other daemons and network services, along with killing zombie processes or parenting orphaned processes.

There are three flavors of system management processes in widespread use:

1. Historically, SysVinit was a series of plaintext files that ran as scripts to start processes, but due to some problems, Linux now uses systemd.

2. An init variant that derives from the BSD UNIX, used on most BSD-based systems.

3. A more recent contender called **systemd** which aims to cover the init processes and much more. This significant increase in control causes some controversy.

4. Other flavors include Apple MacOS's **launchd** before it adopted systemd. Ubuntu also used **Upstart** before migrating to systemd.

Systemd is essentially a collection of smaller programs, services, and libraries such as systemctl, journalctl, init, process management, network management, login management, logs, etc. Some processes may depend on other processes, and with hundreds of them, it's very hard to do manually, which is why systemd does it all for you. A post on the systemd blog notes that a full build of the project generates 69 different binaries (subject to change).

Definition 5.1 ()

A **unit** is anything that is managed by systemd. It can be "a service, a socket, a device, a mount point, an automount point, a swap file or partition, a startup carget, a watched filesystem path, a time controlled and supervised by systemd, a resource management slice, or a group of externally created processes." Within systemd, the behavior of each unit is defined and configured by a **unit file**. Within systemd, the behavior of each unit is defined and configured by a **unit file**.

The files are all over the place:

1. `/lib/systemd/system` contains standard systemd unit files
2. `/usr/lib/systemd/system` are from locally installed packages, e.g. if I installed a pacman package that contained unit files, then those would go here.
3. `/etc/systemd/system` is where you put your custom files. etc also has the highest priority, so it overwrites the other files.
4. `/run/systemd/system` is a scratch area for transient units.

By convention, unit files are named with a suffix that varies according to the type of unit being configured. For example, service units have a `.service` suffix and timers user `.timer`. Within the unit file, some sections e.g. (`[Unit]`) apply generically to all kinds of units, but others (e.g. `[Service]`) can appear only in the context of a particular unit type.

Example 5.1 (Service Unit File)

If we go into one of these unit files, which have the prefix `.service`, they are usually formatted as such:

```
1   # comments are just the same as in bash Scripts
```

```
2   # the headers are important!
3
4   [Unit]          #
5   Description=Description of the unit file
6   Documentation=man:something
7   After=network.target
8
9   [Service]
10  Type=forking  # tells that the process may exit and is not permanent
11  PIDFile=      #
12  ExecStartPre= # scripts to run before you start
13  ExecStart=    # scripts to run when starting
14  ExecReload=   # script to run when you try to reload the process
15  ExecStop=     # script to run to stop the process
16
17  [Install]   # Tells at what point should this be running
18  WantedBy=multi-user.target
```

5.1   systemctl: Managing systemd

systemctl is an all-purpose command for investigating the status of systemd and making changes to its configuration. Running `systemctl` without any arguments invokes the default `list-units` subcommand, which shows all loaded and activive services, sockets, targets, mounts, and devices. To show only services, use `-type=service`.

The two main commands that you will use to interact with systemd is `systemctl` and `journalctl`.

1. `systemctl status unit` checks the status, ouputting the description, whether it's enabled/disabled, and whether it's active/inactive.

2. `systemctl enable unit` enables it, which means that it will start when booting the computer. It does this by creating a symlink to the unit file. This is different from start.

3. `systemctl disable unit` disables it.

4. `systemctl start unit` starts it now and runs it immediately.

5. `systemctl stop unit` makes it inactive.

6. `systemctl reload` will run whatever is in the `ExecReload` in the unit file.

7. `systemctl restart` runs ExecStop and then ExecStart.

8. `systemctl kill unit` kills the process.

Some of the statuses that you may see are inactive (deactivated, exited), active (activating, running), failed, static (not started, frozen by systemd), bad (broken, probably due to bad unit files), masked (ignored by systemd), indirect (disabled, but another unit file references it so it could be activated).

To troubleshoot, you should run `systemctl -failed` to see if there are any failed processes, which can be a problem, and then you can use `journalctl -since=today` to view your systemd logs. This log is important for diagnosing fundamental problems with your system. To view only entries logged at the error level or above, you can set the priorities with `-p err -b`.

5.2   Targets

5.3   Systemd Logging

The **journald** daemon allows you to capture log messages produced by the kernel and services. These system messages are stored in the `/run` directory, but we can access them directly with the `journalctl` command.

> **Example 5.2 ()**

You can configure **journald** to retain messages from prior boots. To do this, edit the following file and configure the `Storage` attribute:

```
1   #/etc/systemd/journald.conf
2     [Journal]
3     Storage=persistent
```

Then, you can obtain a list of prior boots with `journalctl -list-boots` and you can access messages from a prior boot by referring to its index or by naming its long-form ID: `journalctl -b -1`.

6   Directory Structure

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

```
1   (base) mbahng@xps15:~\$ pwd
2   /home/mbahng
3   (base) mbahng@xps15:~\$ cd /
4   (base) mbahng@xps15:/\$ pwd
5   /
```

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

```
1   (base) mbahng@xps15:~\$ ls /
2   bin     dev   lib    libx32      mnt    root  snap  timeshift  var
3   boot    etc   lib32  lost+found  opt    run   srv   tmp
4   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.

6.1   Users and Permission

6.1.1   Managing Users

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

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

```
4    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.

```
1    (base) mbahng@xps15:~\$ cat /etc/passwd
2    root:x:0:0:root:/root:/bin/bash
3    daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
4    bin:x:2:2:bin:/bin:/usr/sbin/nologin
5    sys:x:3:3:sys:/dev:/usr/sbin/nologin
6    sync:x:4:65534:sync:/bin:/bin/sync
7    games:x:5:60:games:/usr/games:/usr/sbin/nologin
8    man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
9    lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
10   mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
11   news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
12   uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
13   proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
14   www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
15   backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
16   list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
17   irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
18   gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
19   nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
20   ...
```

Let us just examine my user.

```
1    (base) mbahng@xps15:~\$ cat /etc/passwd | grep mbahng
2    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

```
1    (base) mbahng@xps15:~\$ sudo useradd batman          # just add user
2    (base) mbahng@xps15:~\$ sudo useradd -m batman       # add user with home dir
3    (base) mbahng@xps15:~\$ cat /etc/passwd | grep batman
4    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

```
1    (base) mbahng@xps15:~\$ sudo userdel batman          # delete user
2    (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

```
1    (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.

6.1.2   Changing Permission

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

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

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

```
1   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 column), 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

```
1   chmod +r testfile.txt    // assign read permissions to all users
2   chmod +w testfile.txt    // assign write permissions to all users
3   chmod +x testfile.txt    // assign execute permissions to all users
4
5   chmod g+rw testfile.txt   // assign read and write to group
6   chmod u-r testfile.txt   // revoke read to user
7   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

```
1   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

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

6.1.3   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.

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

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

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

7   Display Servers

When you boot up your computer, you are greeted with a graphical user interface (GUI) that allows you to interact with your computer. This is the job of the display server, which is a program that provides graphical display capabilities for the operating system.

> **Definition 7.1 (Display Server)**
>
> A **display server** is a program that manages the communication between your computer's hardware and graphical software applications. It acts as a bridge for input and output devices; for example, it processes the input from your keyboard and mouse and outputs graphics to the monitor. The display server is responsible for the fundamental task of drawing windows and handling the low-level aspects of input and output, but it doesn't dictate how these windows look or are arranged. For almost every purpose, there are two types of display servers:
>   1. **X**: The X Window System, which is the older and more established display server.
>   2. **Wayland**: The newer and more modern display server.

> **Definition 7.2 (X Window System)**
>
> The **X Window System** is a windowing protocol for Unix/Linux OSes, similar to the way that Microsoft Windows or Apple Mac OS X can run different apps in separate windows. **X** defines the protocol for a display server what can render windows on a *display client* (your computer), inside which are running apps.[a]
>   1. **X11** refers to version 11 of the X protocol, while
>   2. **Xorg** is an open-source implementation of X.

> **Definition 7.3 (Wayland)**
>
> X, made in 1984, has developed a lot of cruft over the years, and Wayland is a modern replacement for X. It is a protocol for a compositor to talk to its clients, as well as a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel modesetting and evdev input devices, an X application, or a wayland client itself.

8   Package Management

Linux comes in many flavors of distributions. Most beginners look at screenshots of these distributions on the internet and judge them based on their aesthetics (e.g. I like how Kali Linux looks so I'll go with that

---

[a]Explanation here: https://www.reddit.com/r/linuxquestions/comments/3uh9n9/what_exactly_is_xxorgx11/

one). A common feature of all Linux distributions is that they provide the user the power to customize their system however they want, so you can essentially make every linux distribution look like any other. So what are some things you should consider when choosing a distribution?

1. First is the popularity and how well it is supported. This includes the number of people who use the distribution (e.g. the Ubuntu StackExchange is a very large community) and how good the documentation is overall (e.g. the ArchLinux wiki is very well documented).

2. Each linux distribution essentially consists of a kernel and package manager. The architecture, design, and the update scheme of the kernel may be an interest to many linux users.

3. Every distribution has its own native package manager, and the availability of certain necessary packages, the ease of installation, and the updating schemes is also something to consider.

4. The ideals of the respective communities. The community behind each distribution has a certain set of ideals that they lean more towards. For example, the Ubuntu community likes having programs that are right out of the box, with good GUI support and is more beginner-friendly while Arch has more of a minimal and extremely customizable nature to it with its software being much more CLI dependent.

Let's begin with the package managers. Every application on your system (Firefox, Spotify, pdf readers, VSCode, etc.) is a package, and manually downloading and managing each one is impossible to do. Therefore, each distribution has its own native package manager that automatically takes care of downloading, installing, removing, checking dependency requirements of each package. In order to download a package, a package manager should also know where it is downloading *from*. Essentially, a package manager itself can be downloaded with other package managers, so package managers are packages as well.

1. **apt** : The advanced packaging tool is the native manager for Ubuntu distributions.

2. **pacman** : Native package manager for Arch Linux.

3. **yay** : The package manager for software in the **Arch User Repository**.

4. **snap** :

5. **flatpak** :

6. **dpkg** : Package manager for Debian based distributions.

Chances are if you are using one distribution, you would only have to work with a small subset of these package managers. Each package manager has one or more files in the computer that specify a list of **repositories**.

8.1   Wget

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

8.2   Pacman

For example, the configuration file for pacman is located at `/etc/pacman.conf`. In the options section, I can configure stuff like text color, enabling/disabling parallel downloads, choosing specific packages to ignore upgrading, etc. Then, we can specify the servers that we should download from. In the text below, the server variable defines which server we should look at first, and then the Include variable stores the location of the file `mirrorlist` that defines a list of other servers that we should download from.

The mirrorlist file stores a list of URLs. Each URL is a **mirror**, which is a server that contains a physical replica of all the packages that are available to you via `pacman` (hence the name mirror). You can literally type in the links provided in Figure 13 (replacing `$repo` with `core` and `$arch` with `x86_64`). It contains a tarball of each package ready to be downloaded. Some repos might contain more packages than others, some might have packages that only they supply that others don't, but if you can install the piece of software via your package manager then one of your configured repos is declaring they have it available and therefore

```
 1  # The following paths are commented out with their default values listed.
 2  # If you wish to use different paths, uncomment and update the paths.
 3  #RootDir     = /
 4  #DBPath      = /var/lib/pacman/
 5  #CacheDir    = /var/cache/pacman/pkg/
 6  #LogFile     = /var/log/pacman.log
 7  #GPGDir      = /etc/pacman.d/gnupg/
 8  #HookDir     = /etc/pacman.d/hooks/
 9  HoldPkg      = pacman glibc
10  #XferCommand = /usr/bin/curl -L -C - -f -o %o %u
11  #XferCommand = /usr/bin/wget --passive-ftp -c -O %o %u
12  #CleanMethod = KeepInstalled
13  Architecture = auto
14
15  # Pacman won't upgrade packages listed in IgnorePkg and members of IgnoreGroup
16  #IgnorePkg   =
17  #IgnoreGroup =
18
19  #NoUpgrade   =
20  #NoExtract   =
21
22  # Misc options
23  #UseSyslog
24  #Color
25  #NoProgressBar
26  CheckSpace
27  #VerbosePkgLists
28  ParallelDownloads = 5
29  ILoveCandy
```

Figure 12: Subset of contents of the `/etc/pacman.conf` file

```
1  Server = https://archlinux.mailtunnel.eu/$repo/os/$arch
2  Server = https://mirror.cyberbits.eu/archlinux/$repo/os/$arch
3  Server = https://mirror.theo546.fr/archlinux/$repo/os/$arch
4  Server = https://mirror.sunred.org/archlinux/$repo/os/$arch
5  Server = https://mirror.f4st.host/archlinux/$repo/os/$arch
6  Server = https://md.mirrors.hacktegic.com/archlinux/$repo/os/$arch
7  Server = https://mirrors.neusoft.edu.cn/archlinux/$repo/os/$arch
8  Server = https://mirror.moson.org/arch/$repo/os/$arch
9  Server = https://archlinux.thaller.ws/$repo/os/$arch
```

Figure 13: Contents of the `/etc/pacman.d/mirrorlist` file

should have the file on hand to give to you if asked for it. A list of all available mirrors are available here (this only uses HTTPS, but HTTP mirrors are also available).

The mirrors that you download from should be trustworthy and fast. The speed is mainly related to how close you are to that mirror geographically, so if you are moving to another country you should probably update this mirrorlist for faster download speeds. There is a default mirrorlist file that is generated, but you can download and use the **reflector** package to update it.

Here are some common commands:

1. Install a package: `sudo pacman -S pkg1` (`-s` stands for synchronize)

2. Remove a package: `sudo pacman -R pkg`

   - remove dependencies also: `-s` (recursive)

   - also remove configuration files: `-n` (no save)

   - also removes children packages: `-c` (cascade)

3. Update all packages: `sudo pacman -Syu`

   - synchronize: `-S`

   - refresh package databases: `-y` (completely refresh: `-yy`)

   - system upgrade: `-u`

4. List installed packages: `pacman -Q`

   - List detailed info about a package: `pacman -Qi pkg`

   - List all files provided by a package: `pacman -Ql pkg`

   - List all orphaned packages: `pacman -Qdt`

   - List all packages that have updates available: `pacman -Qu`

   - List all explicitly installed packages: `pacman -Qet`

   - Display the dependency tree of a package: `pactree pkg` (from the `pacman-contrib` package)

   - List last 20 installed packages:

     ```
     1  expac --timefmt='%Y-%m-%d %T' '%l\t%n' | sort | tail -n 20
     ```

5. To check size of current packages and dependencies, download `expac` and run `expac -H M '%m t%n' | sort -h`

6. The package cache stored in `/var/cache/pacman/pkg/` keeps old or uninstalled versions of packages automatically. This is helpful since it also keeps older versions of packages in the cache, and you can manually downgrade in case some packages break.

   - We can delete all cached versions of installed and uninstalled packages, except for the most recent 3, by running `paccache -r` (provided by the `pacman-contrib` package).

   - To remove all cached packages not currently installed, run `pacman -Sc`

   - To remove all cached aggressively, run `pacman -Scc`

   - To downgrade, you go into the package cache directory and say you want to see which versions of neovim you have installed. You can `ls` the directory to see the following.

     ```
     1   neovim-0.9.5-1-x86_64.pkg.tar.zst
     2   neovim-0.9.5-1-x86_64.pkg.tar.zst.sig
     3   neovim-0.9.5-2-x86_64.pkg.tar.zst
     4   neovim-0.9.5-2-x86_64.pkg.tar.zst.sig
     ```

     We have an older version of neovim installed, and to roll it back we can use

     ```
     1   pacman -U neovim-0.9.5-1-x86_64.pkg.tar.zst
     ```

The pacman log (`/var/log/pacman.log`) is also useful since it logs all pacman outputs when you do anything with pacman. So if you are looking for the packages that have been installed in the latest `pacman -Syu`, then you can use this to individually see each package that has been upgraded.

8.3   Yay

Yay is used to install from the Arch User repository and must be updated separately. To run this, you can either run `yay -Syu` or you can just run `yay`. Since this is not officially maintained, these packages are more likely to break something. The yay logs are not stored separately can can be accessed in the pacman logs.

8.4   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

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

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

```
1   dpkg -r package
```

8.5   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.

```
1  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.

```
1  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.

```
1  sudo apt dist-upgrade
```

Installing and removing packages is easy.

1. We can install from the apt repository with

```
1  sudo apt install htop
```

2. We can remove it with

```
1  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

```
1  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.

```
1   # deb cdrom:[Ubuntu 22.04.1 LTS _Jammy Jellyfish_ - Release amd64 (20220809.1)]/
2   jammy main restricted
3
4   # See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
5   # newer versions of the distribution.
6   deb http://us.archive.ubuntu.com/ubuntu/ jammy main restricted
7   # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy main restricted
8
9   ## Major bug fix updates produced after the final release of the
10  ## distribution.
11  deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates main restricted
12  # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates main restricted
13
```

```
14   ## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
15   ## team. Also, please note that software in universe WILL NOT receive any
16   ## review or updates from the Ubuntu security team.
17   deb http://us.archive.ubuntu.com/ubuntu/ jammy universe
18   # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy universe
19   deb http://us.archive.ubuntu.com/ubuntu/ jammy-updates universe
20   # deb-src http://kr.archive.ubuntu.com/ubuntu/ jammy-updates universe
21   ...
```

8.6   Snap and Flatpak

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

```
1   sudo apt install snap flatpak
```

9   Windows Managers and Desktop Environments

These days, the terms window managers (WMs) and Desktop Environments (DEs) are used interchangeably, but they mean slightly different things. A window manager is the display software that determines how the pixels for each window overlaps with other and their movement. This is generally divided into two paradigms with the most familiar being **floating WMs** and the other being **tiling WMs**. Even before I knew about tiling WMs, I found myself manually tiling windows on floating WMs, so the move to tiling WMs was a no-brainer.

Some DEs and WMs are:

1. GNOME

2. KDE Plasma

3. Qtile

10   Shells and Terminals

Beginners may think of the shell and the terminal to be the same thing, but they are different. The **shell** is a command line interpreter, a layer that sits on top of the kernel in which the user can interact with. It is essentially the only API to the kernel where the user can input commands and processes them. The **terminal emulator** is a wrapper program that runs a shell and allows us to access the API. It may be useful to think of the shell as like a programming language and the terminal as a text editor like VSCode.

The three most common shells are the following:

1. **Bash**:

2. **Zsh**:

3. **Fish**:

Some common terminal emulators (most of which comes as a part of the desktop environment) are the following:

1. **Kitty**:

2. **Alacritty**:

3. **Gnome-Terminal**:

10.1   Crontab

To schedule jobs, you run `crontab -e`, which will give you a text file for which you can list jobs. It looks like

```
1   # Edit this file to introduce tasks to be run by cron.
2   #
3   # Each task to run has to be defined through a single line
4   # indicating with different fields when the task will be run
5   # and what command to run for the task
6   #
7   # To define the time you can provide concrete values for
8   # minute (m), hour (h), day of month (dom), month (mon),
9   # and day of week (dow) or use '*' in these fields (for 'any').
10  #
11  # Notice that tasks will be started based on the cron's system
12  # daemon's notion of time and timezones.
13  #
14  # Output of the crontab jobs (including errors) is sent through
15  # email to the user the crontab file belongs to (unless redirected).
16  #
17  # For example, you can run a backup of all your user accounts
18  # at 5 a.m every week with:
19  # 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
20  #
21  # For more information see the manual pages of crontab(5) and cron(8)
22  #
23  # m h  dom mon dow   command
```

In the bottom line, we can add the following to run `sudo apt update` every minute. The 5 columns refer to minute (0-59), hour (0-24), date of month (1-31), month (1-12), and date of week (0-7, where 0 and 7 is Sunday). The asterick means every instance of.

```
1   # Run every minute
2   * * * * * sudo apt update
3
4   # Run at 9:15am every first day of the month
5   15 9 1 * * sudo apt update
6
7   # Run for every minute of every hour for the 13th day of every month if it is Friday
8   * * 13 * 5 sudo apt update
```

You get the idea.