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.

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 1.

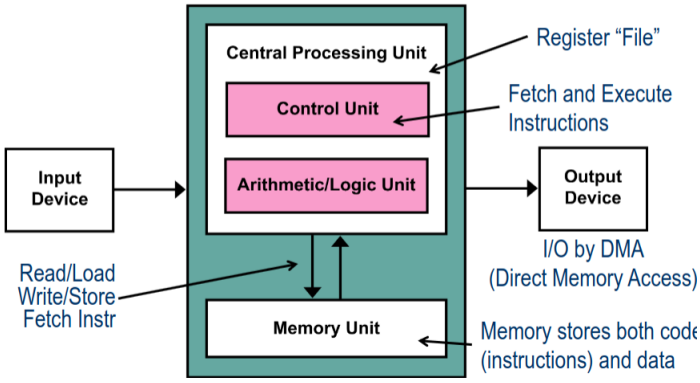


Figure 1: von Neumann Architecture

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.

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[1].
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[2]:

[1] https://en.wikipedia.org/wiki/X86_instruction_listings
[2] Older x86_32 architecture has 8 general purpose registers with the `r` replaced by a `e`, e.g. `eax` instead of `rax`.

```
%rax        # return value
%rbx        # callee saved
%rcx        # 4th argument
%rdx        # 3rd argument
%rsi        # 2nd argument
%rdi        # 1st argument
%rbp        # callee saved
%rsp        # stack pointer
%r8         # 5th argument
%r9         # 6th argument
%r10        # scratch register
%r11        # scratch register
%r12        # callee saved
%r13        # callee saved
%r14        # callee saved
%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.*[3]

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.[4]

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.

[3] https://archlinux.org/

[4] 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

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

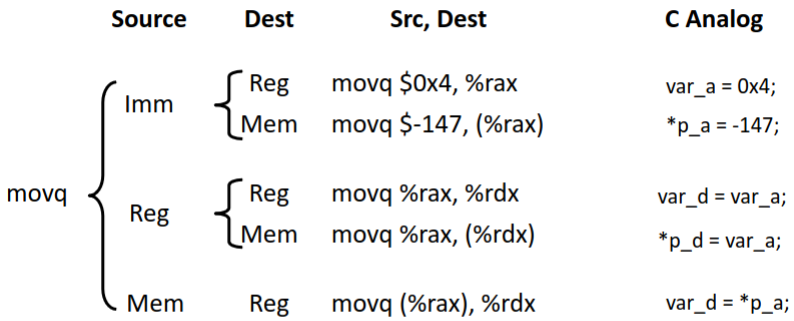
```
%reg = %reg + Mem[address]     # add memory data to register
%reg = %reg - Mem[address]     # subtract memory data from register
%reg = %reg * Mem[address]     # multiply memory data to register
%reg = %reg / Mem[address]     # divide memory data from register
```

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).[5]

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**).[6],[7]

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.

[5] A video of Linus Torvalds saying "fuck you" to Nvidia: https://www.youtube.com/watch?v=iYWzMvlj2RQ
[6] This wiki article (https://github.com/Askannz/optimus-manager/wiki) provides a good overview of this matter.
[7] 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).

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

Figure 2: This is the following output of lspci on my personal computer.

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).

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

Figure 3: This is the following output on my personal computer.

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**.

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

Device             Start        End    Sectors   Size Type
/dev/nvme0n1p1      2048     493567     491520   240M EFI System
/dev/nvme0n1p2     493568     755711     262144   128M Microsoft reserved
/dev/nvme0n1p3     755712  649658367  648902656 309.4G Microsoft basic data
/dev/nvme0n1p4 1960380416 1962407935    2027520   990M Windows recovery environment
/dev/nvme0n1p5 1962407936 1997441023   35033088  16.7G Windows recovery environment
/dev/nvme0n1p6 1997443072 2000377855    2934784   1.4G Windows recovery environment
/dev/nvme0n1p7  649658368  650682367    1024000   500M EFI System
/dev/nvme0n1p8  650682368  660447231    9764864   4.7G Linux swap
/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 4: 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 5.

```
Model: PM9A1 NVMe Samsung 1024GB (nvme)
Disk /dev/nvme0n1: 1024GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt
Disk Flags:

Number  Start   End     Size    File system     Name                            Flags
 1      1049kB  253MB   252MB   fat32           EFI system partition            boot, esp
 2      253MB   387MB   134MB                   Microsoft reserved partition    msftres
 3      387MB   333GB   332GB   ntfs            Basic data partition            msftdata
 7      333GB   333GB   524MB   fat32                                           boot, esp
 8      333GB   338GB   5000MB  linux-swap(v1)                                  swap
 9      338GB   1004GB  666GB   ext4
 4      1004GB  1005GB  1038MB  ntfs                                            hidden, diag
 5      1005GB  1023GB  17.9GB  ntfs                                            hidden, diag
 6      1023GB  1024GB  1503MB  ntfs                                            hidden, diag


Model: Unknown (unknown)
Disk /dev/zram0: 4295MB
Sector size (logical/physical): 4096B/4096B
Partition Table: loop
Disk Flags:

Number  Start  End     Size    File system     Flags
 1      0.00B  4295MB  4295MB  linux-swap(v1)
```

Figure 5: 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.

Some important functionalities you can do with the firmware are:

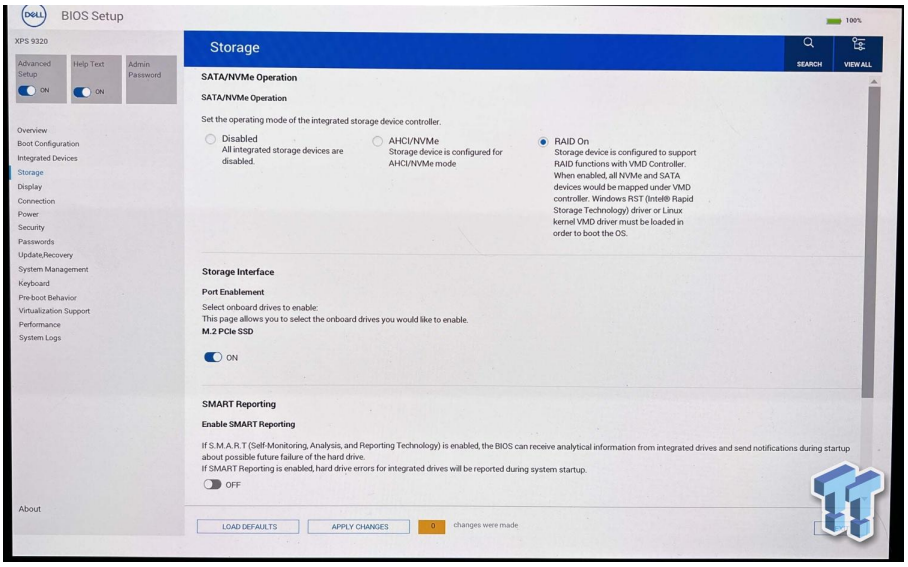


Figure 6: Firmware of Dell XPS 13 9320

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. 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 impelmentation 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:

```
BootCurrent: 0005
Timeout: 0 seconds
BootOrder: 0005,0001,0002,0000,0003,0004
Boot0000* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1        ...
Boot0001* ubuntu        HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd6026c1,0x800,0x780...
Boot0002* Windows Boot Manager        HD(1,GPT,ede98b7e-75ad-452e-ab47-3411dd60...
Boot0003* Linux Firmware Updater        HD(1,GPT,ede98b7e-75ad-452e-ab47-...
Boot0004* UEFI PM9A1 NVMe Samsung 1024GB S65VNE0R318841 1 2        PciRoot(0x0)/...
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.

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.

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.

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:

```
# comments are just the same as in bash Scripts
# the headers are important!

[Unit]        #
Description=Description of the unit file
Documentation=man:something
After=network.target

[Service]
Type=forking  # tells that the process may exit and is not permanent
PIDFile=      #
ExecStartPre= # scripts to run before you start
ExecStart=    # scripts to run when starting
ExecReload=   # script to run when you try to reload the process
ExecStop=     # script to run to stop the process

[Install]   # Tells at what point should this be running
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:

```
#/etc/systemd/journald.conf
  [Journal]
  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 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 6.1 (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 6.2 (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.[8]

1. **X11** refers to version 11 of the X protocol, while
2. **Xorg** is an open-source implementation of X.

Definition 6.3 (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.

7 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

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

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

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 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 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

```
# The following paths are commented out with their default values listed.
# If you wish to use different paths, uncomment and update the paths.
#RootDir     = /
#DBPath      = /var/lib/pacman/
#CacheDir    = /var/cache/pacman/pkg/
#LogFile     = /var/log/pacman.log
#GPGDir      = /etc/pacman.d/gnupg/
#HookDir     = /etc/pacman.d/hooks/
HoldPkg     = pacman glibc
#XferCommand = /usr/bin/curl -L -C - -f -o %o %u
#XferCommand = /usr/bin/wget --passive-ftp -c -O %o %u
#CleanMethod = KeepInstalled
Architecture = auto

# Pacman won't upgrade packages listed in IgnorePkg and members of IgnoreGroup
#IgnorePkg   =
#IgnoreGroup =

#NoUpgrade   =
#NoExtract   =

# Misc options
#UseSyslog
#Color
#NoProgressBar
CheckSpace
#VerbosePkgLists
ParallelDownloads = 5
ILoveCandy
```

Figure 7: Subset of contents of the `/etc/pacman.conf` file

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 8 (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 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`

```
Server = https://archlinux.mailtunnel.eu/$repo/os/$arch
Server = https://mirror.cyberbits.eu/archlinux/$repo/os/$arch
Server = https://mirror.theo546.fr/archlinux/$repo/os/$arch
Server = https://mirror.sunred.org/archlinux/$repo/os/$arch
Server = https://mirror.f4st.host/archlinux/$repo/os/$arch
Server = https://md.mirrors.hacktegic.com/archlinux/$repo/os/$arch
Server = https://mirrors.neusoft.edu.cn/archlinux/$repo/os/$arch
Server = https://mirror.moson.org/arch/$repo/os/$arch
Server = https://archlinux.thaller.ws/$repo/os/$arch
```

Figure 8: Contents of the `/etc/pacman.d/mirrorlist` file

- 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:

     ```
     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.

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

```
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.

8.2 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.

9 Vim and Neovim

Vim is guaranteed to be on every Linux system, so there is no need to install it. However, you may have to install Neovim (which is just a command away). Vim can be a really big pain in the ass to learn, but I got into it when I was watching some video streams from a senior software engineer at Netflix called The Primeagen. He moved around the code like I've never seen, and I was pretty much at the limit of my typing speed, so I decided to give it a try during the 2023 fall semester. My productivity plummetted during the first 2 days (which was quite scary given that I had homework due), but within a few weeks I was faster than before, so if you have the patience, I would recommend learning it. Here is a summary of reasons why I would recommend learning Vim:

1. It pushes you to know the ins and outs of your editor. As a mechanic with his tools, a programmer should know exactly how to configure their editor.
2. The plugin ecosystem is much more diverse than other editors such as VSCode. You can find plugins/extensions for everything. Here is a summmary of them here.
3. You're faster. If you're going to be coding for the next 5 years, then why no t spend a month to master something that will make you faster? You'll increase total productivity.
4. Computing clusters and servers will be much easier to navigate since they all run Linux with Vim.
5. Vim is lightweight, and you don't have to open up VSCode every time you want to edit a configuration file.

9.1 Vim vs Neovim

Experience wise, Vim and Neovim are very similar, and if you configure things rihght, you may not even be able to tell the difference. But there are 3 differences that I want to mention:

1. Neovim can be configured in Lua, which is much cleaner than Vimscript.
2. Neovim provides mouse control right out of the box, which is convenient for me at times and can be easier to transition into, while Vim does not provide any mouse support.
3. There are some plugins that are provided in Neovim that are not in Vim.

Either way, the configuration is essentially the same. At startup, the text editor will parse some predetermined configuration file and load those settings.

It may be the case that a remote server does not have neovim installed, or you may not have the permissions to install it. In this case, you can use **sshfs**, which is a file system client based on the SSH File Transfer Protocol. It allows you to mount a remote directory over SSH.

9.2 Vim Configuration File

In Vim, your configuration files are located in `/.vimrc` and plugins are located in `/.vim/`. In here, you can put in whatever options, keymaps, and plugins you want. All the configuration is written in VimScript.

```
# options
filetype plugin indent on
syntax on
set background=dark
set expandtab ts=2 sw=2 ai
set nu
set linebreak
set relativenumber

# keymaps
inoremap <C-j> <esc>dvbi
inoremap jk <esc>
nnoremap <C-h> ge
nnoremap <C-l> w
```

9.3 Neovim Configuration File

In Neovim, I organize it using Lua. It essentially looks for the `/.config/nvim/init.lua` file and loads the options from there. We also have the option to import other Lua modules for better file structure with the `require` keyword. The tree structure of this configuration file should be the following below. The extra `user` director layer is necessary for isolating configuration files on multiple user environments.

```
... ftplugin
.   ... cpp.lua
.   ... html.lua
... init.lua
... LICENSE
... lua
.   ... user
.       ... options.lua
.       ... keymaps.lua
.       ... plugins.lua
.       ... telescope.lua
.       ... toggleterm.lua
... plugin
    ... packer_compiled.lua
```

The init file is the "main file" which is parsed first. I generally don't put any explicit options in this file and reserve it only for require statements. It points to the following (group of) files:

1. **options.lua**: This is where I store all my options.
2. **keymaps.lua**: All keymaps.
3. **plugins.lua**: First contains a script to automatically install packer if it is not there, and then contains a list of plugins to download.

4. **Plugin Files**: Individual configuration files for each plugin (e.g. if I install a colorscheme plugin, I should choose which specific colorscheme I want from that plugin).
5. **Filetype Configuration Files**: Options/keymaps/plugins to load for a specific filetype. This helps increase convenience and speed since I won't need plugins like VimTex if I am working in JavaScript.

Once you have your basic options and keymaps done, you'll be spending most of your time experimenting with plugins. It is worth to mention some good ones that I use.

1. **Packer** as the essential package manager.
2. **Plenary**
3. **Telescope** for quick search and retrieval of files.
4. **Indent-blankline** for folding.
5. **Neoformat** for automatic indent format.
6. **Autopairs** and **autotag** to automatically close quotation marks and parantheses.
7. **Undotree** to generate and navigate undo history.
8. **Vimtex** for compilation of LaTeX documents.
9. **Onedark** and **Oceanic Next** for color schemes.
10. **Vim-Startify** for nice looking neovim startup.
11. **Comment** for commenting visual blocks of code.

It is also worthwhile to see how they are actually loaded in the backend. Each plugin is simply a github repo that has been cloned into `/.local/share/nvim/site/pack/packer/`, which contains two directories. The packages in `start/` are loaded up every time Neovim starts, and those in `opt/` are packages that are loaded up when a command is called in a certain file (known as lazy loading). Therefore, if you have any problems with Neovim, you should probably look into these folders (and possibly delete them and reinstall them using Packer if needed).

9.4 Troubleshooting

A good test to run is `:checkhealth`, which checks for any errors or warnings in your Neovim configuration. You should aim to have every (non-optional) warning cleared, which usually involves having to install some package, making it executable and/or adding to `$PATH`.

If you are getting plugin errors, you can also manually delete the plugin directory in 'pack/packer' and run 'PackerInstall' to re-pull the repos. This may help.

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**:

11 LaTeX and VimTex

Latex is a great way to take notes. One can go to Overleaf and have everything preconfigured, but in here I set it up on my local desktop. I will already assume you have a PDF viewer installed. I use zathura, which is lightweight and also comes with vim motions for navigation.

First install the VimTex plugin in `plugins.lua` with `use lervag/vimtex`. Then, you want to install TexLive, which is needed to compile tex files and to manage packages. The directions for TexLive installation is available [here](https://tug.org/texlive/quickinstall.html). Once I downloaded the install files, I like to run `sudo perl ./install-tl --scheme=small`. Be careful with the server location (which can be set with the `--location` parameter), as I have gotten some errors. I set `--scheme=small`, which installs about 350 packages compared to the default scheme, which installs about 5000 packages (7GB). I also did not set `--no-interaction` since I want to slightly modify the `--texuserdir` to some other path rather than just my home directory.

Once you installed everything, make sure to add the binaries to PATH, which will allow you to access the **tlmgr** package manager, which pulls from the CTAN (Comprehensive TeX Archive Network) and gives VimTex access to these executables. Unfortunately, the small scheme installation does not also install the **latexmk** compiler, which is recommended by VimTex. We can simply install this by running "‘ sudo tlmgr install latexmk "‘ Now run ‘:checkhealth‘ in Neovim and make sure that everything is OK, and install whatever else is needed.

To install other Latex packages (and even document classes), we can use tlmgr. All the binaries and packages are located in `/usr/loca/texlive/202*/` and since we're modifying this, we should run it with root privileges. The binaries can also be found here. Let's go through some basic commands:

1. List all available packages: `tlmgr list`
2. List installed packages: `tlmgr list --only-installed` (the packages with the ‘i‘ next to them are installed)
3. Install a package and dependencies: `sudo tlmgr install amsmath tikz`
4. Reinstall a package: `sudo tlmgr install amsmath --reinstall`
5. Remove a package: `sudo tlmgr remove amsmath` More commands can be found here for future reference.

After this, you can install Inkscape, which is free vector-based graphics editor (like Adobe Illustrator). It is great for drawing diagrams, and you can generate custom keymaps that automatically open Inkscape for drawing diagrams within LaTeX, allowing for an seamless note-taking experience.

12 Networking

Networking is a large field in itself, but in here I go over the most useful and practical applications of it in my everyday use. Some ways that I personally benefit from this is:

1. Connecting to WiFi and diagnosing problems.
2. Connecting to WiFi and diagnosing problems.

3. Connecting to other networks such as computing clusters or third-party blockchains.
4. Seeing how more abstract schemes such as APIs work.
5. Ethical hacking.

I introduce these concepts and how to do some basic implementation a Unix operating system.

I like to learn about networking as if I am designing it from scratch. Some big questions to ask when designing network schemes are:

1. How do we uniquely identify computers?
2. How should we establish a connection between them? Through hardware or signals?
3. What protocols should we use, like a common language, so that all computers understand what each other are saying?
4. Can we implement security measures to prevent unwanted visitors into our computer?

12.1 Computer Networks and the Internet

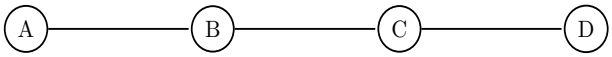
Let us first define a computer network, some of its architecture, and then move onto the Internet.

Definition 12.1 (Computer Network)**.** A **computer network** is a group of computers (i.e. computing devices) that use a set of common *communication protocols* over digital interconnections for the purpose of sharing resources located on or provided by the *network nodes*, which may include personal computers, servers, networking hardware, or other specialised or general-purpose hosts.

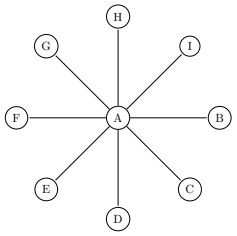
These network nodes may be able to communicate to certain neighboring nodes, and this graph architecture determines the **network topology**. A computer network can be visualized as a connected graph of nodes

Example 12.1 (Network Topologies)**.** Common layouts are:

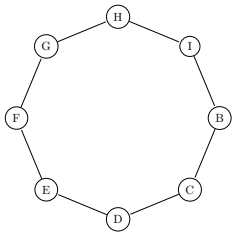
1. **Line Network**: All nodes are connected in a line.



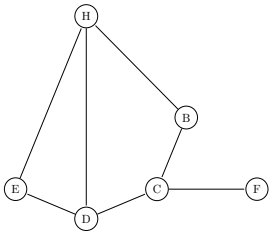
2. **Bus Network**: All nodes are connected to a common medium along this medium.
3. **Star Network**: all nodes are connected to a special central node.



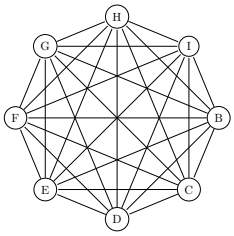
4. **Ring Network**: Each node is connected to its left and right neighbour node, such that all nodes are connected and that each node can reach each other node by traversing nodes left- or rightwards.



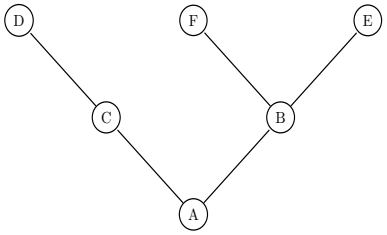
5. **Mesh Network**: each node is connected to an arbitrary number of neighbours in such a way that there is at least one traversal from any node to any other.



6. **Fully Connected Network**: each node is connected to every other node in the network.



7. **Tree Network**: nodes are arranged hierarchically.



Notice how many of these networks have **redundancy**: having multiple ways to get from one node to another. That is, when a network path is no longer available, data is still able to reach its destination through another path. Usually, we would like to avoid a **single point of failure** and construct a **fault-tolerant** system that can experience failure in its components but still continue operating properly. However, building more connections may be expensive.

Because there are multiple paths that a piece of data takes to get from point X to point Y, *routing strategies* are implemented in order to determine the most optimal path. Now in order for network nodes to communicate with each other, they should have some sort of universal method of communicating with each other.

Definition 12.2 (Communication Protocol)**.** A **communication protocol** is a system of rules that allow multiple entities of a communications to transmit information via any kind variation of a physical quantity. The protocol defines the rules, syntax, semantics and synchronization of communication and possible error recovery methods. A protocol can have many jobs, such as:

1. Determining how nodes will communicate with each other .
2. Making sure that these modes of communication is compatible with hardware .
3. Implementing security protocols such as encryption schemes.

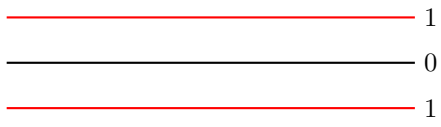
Computers can connect through **physical** (e.g. cables) or **wireless** connections.

1. The **CAT5 cable** is a *twisted pair (copper) cable* that's designed for use in computer networks. It consists of four twisted pairs of copper wires. These twisted pair cables send data through a network by transmitting pulses of electricity that represent binary data. The information transmission follow the **Ethernet** standards, which is why twisted pair cables are commonly known as Ethernet cables. Use

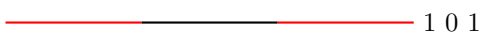
for both LANs and WANs. They can carry up to 1 Gbps across hundreds of feet, but are susceptible to interference.

2. **Fiber-optic cables** carry light instead of electricity in a fiber coated with plastic layers. The pulses of light represent binary data and also follow the Ethernet standards. They are also capable of transmitting much more data per second that copper cables, and they have the advantage of low transmission loss and immunity to electrical interference. Often used to connect networks across oceans so that data can travel quickly around the world. They can carry up to 26 Tbps acorss 50 miles (but are expensive)
3. A wireless card inside a computer turns binary data into **radio waves** and transmits them through the air. However, they do not travel very far (100 ft in office buildings or up to 1000 ft in an open field). The waves are picked up by a *wireless access point* which converts them from radio waves back into binary data. These access points would be connected to the rest of the network using physical wiring. They can carry up to 1.3 Gbps.
4. **Infrared signals** and **microwaves** are sometimes used.

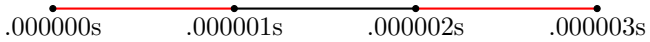
In order for the computers to send data into binary, they must convert this data into binary and send them as streams of 1s and 0s in a process called **line coding**. Furthermore, computers can raise efficiency of each wire by sending changing electric currents through a single wire. For example, rather than using three wires to encode `101` as



they send it through a single wire with intervals of $\frac{1}{3}$ seconds



or even better, at a rate of 1 megabit per second (interval of 0.000001 seconds)



As long as two computers agree on the time period in which the electricity intervals are being sent, they can communicate much more efficiently. In an electrical connection (such as Ethernet), the signal would be a voltage or current. In an optical connection (such as a fiber-optic cable), the signal would be the intensity of light.

Definition 12.3. There are many properties about line coding that are relevant, but ultimately the speed of a connection is a combination of the bandwidth and latency.

1. The **bit rate** describes the data transfer rate of a connection. It measures the number of bit states that a channel can *transmit* per unit time. It is measured in *bits per second*. We can interpret it as the amount of water flowing through a pipe.

   Bit rate is typically seen in terms of the actual data rate. But for most transmissions, the data represents part of a more complex protocol, which includes bits representing source address, destination address, error detection/correction codes, and other information. This data is called the **overhead**, while the actual data transferred is called the **payload**. At times, the overhead may be substantial (up to 20% to 50%).
2. The **throughput** is the number of bit states of usable information, that can be successfully *received* over a channel per unit time. Without any channel noise, it is really just the payload. Note that this is an *observed, dynamic parameter* with a fixed and variable loss. It is also known as **consumed bandwidth** and is measured in *bits per second*.
3. The **bandwidth** describes the *maximum* data transfer rate of a connection; that is, the maximum throughput of a communication. It is measured in *bits per second*. We can interpret it as how thick

the pipe is (i.e. how much water can flow through it at max). Note that this is different from the bandwidth used in signal processing.

Data often flows over multiple network connections, which means the connection with the smallest bandwidth (most likely your local connection) acts as a bottleneck.

4. The **latency**, or **ping-rate**, measures the round trip time between the sending of a data message to a computer and the receiving of that message, measured in *milliseconds*. We can interpret it as the speed at which the water is flowing through a pipe. We can check latency by doing

```
>>>ping www.google.com
64 bytes: icmp_seq=0 ttl=115 time=37.868 ms
```

which outputs a latency time of 37.868ms (to get to `www.google.com` and back) for sending a data packet of 64 bytes. Note that there is an intrinsic limiting factor to latency: the speed of light, which is approximately 1 foot per nanosecond. In addition to distance, another limiting factor is the congestion in the network and the type of connection.

Example 12.2. Given two computers connected by a wire that is configured to transfer 1000 bits per second, the bit rate would be 1 Kbps. However, if the channel has noise and demands retransmission of 10 bits out of every 1000 of the original transmission, then the throughput would be 990 bps.

Furthermore, the Ethernet frame can have as many as 1542 bytes. Say that there are 1500 bytes of payload and an overhead of 42 bytes. Then, the **protocol efficiency** would would be

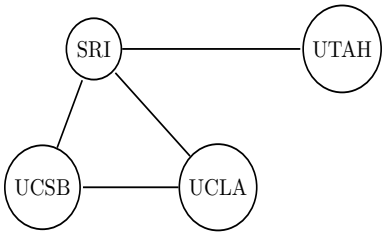
$$\frac{\text{payload}}{\text{frame size}} = \frac{1500}{1542} = 0.9727 = 97.3\%$$

Typically, the actual line rate is stepped up by a factor influenced by the overhead to achieve an actual target net data rate. In One Gigabit Ethernet, the actual line rate is 1.25 Gbits/s to achieve a net payload throughput of 1 Gbit/s. In a 10-Gbit/s Ethernet system, gross data rate equals 10.3125 Gbits/s to achieve a true data rate of 10 Gbits/s. The net data rate also is referred to as the throughput, or payload rate, of effective data rate.

12.2 History of the Internet

IETF, ICANN, IANA, ISPs.

Example 12.3 (ARPANET)**.** The ARPANET was the precursor to the Internet, the network where Internet technology was first tested out. It was started in 1969 with four computers connected to each other.



For example, even if the path between SRI and UCSB is gone, the connections between SRI and UCSB is not lost (since IP packets can travel through UCLA's router).

Now we can see an implementation of these networks in the internet.

Definition 12.4 (Internet)**.** The **Internet** is a global network of computing devices communicating with each other in some way, whether they're sending emails, downloading files, or sharing websites. The Internet is an **open network**, which means that any computing device can join as long as they follow the protocols. The internet is powered by many layers of protocols, and to create a global network of computing devices, we need:

1. **Wires & Wireless**: Physical connections between devices, plus protocols for converting electromagnetic signals into binary data.
2. **IP**: A protocol that uniquely identifies devices using IP addresses and provides a routing strategy to send data to a destination IP address.
3. **TCP/UDP**: Protocols that can transport packets of data from one device to another and check for errors along the way.
4. **TLS**: A secure protocol for sending encrypted data so that attackers can't view private information.
5. **HTTP & DNS**: The protocols powering the World Wide Web

An **ISP (Internet Service Provider)** provides internet to its region. These ISPs are managed by certain continental autonomous systems (**AS**). The **Regional Internet Registry (RIR)** is divided into their regions: AFRNIC (Africa), ARIN (American), APNIC (Asia-Pacific), LACNIC (Latin America and Carribean), and RIPE NCC (European).

The main protocol suite used by the internet is **TCP/IP**, which is a collection of protocols that the internet uses. The bulk of this chapter will describe this protocol.

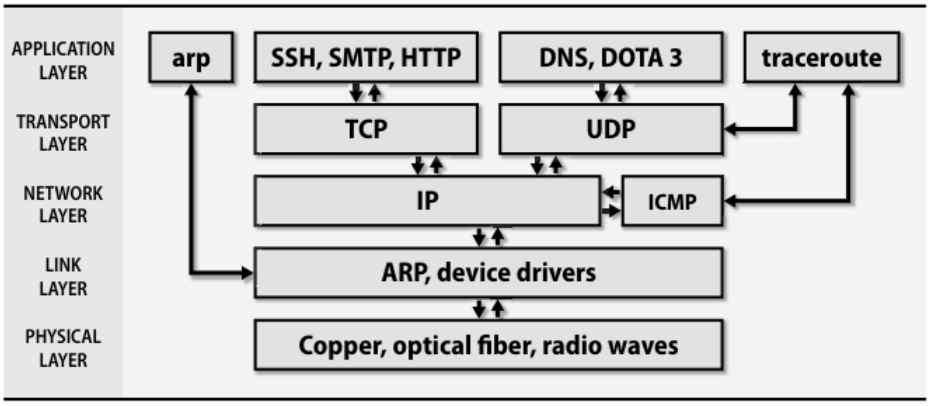


Figure 9: TCP/IP layering model.

12.3 Network Interfaces

Before we even start talking about IP addresses or protocols, we should mention that there are several interfaces from which computers can send and receive data. For example, if you are connected to both wired ethernet and WiFi, there are two paths, or interfaces, that data can travel. To see all your interfaces, use the `ip -c a` command.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
    link/ether 64:bc:58:11:c0:24 brd ff:ff:ff:ff:ff:ff
    inet 10.197.221.245/16 brd 10.197.255.255 scope global dynamic noprefixroute
       valid_lft 597085sec preferred_lft 597085sec
    inet6 fe80::b9e9:2f85:ded7:eaaf/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

The following lists out all the interfaces. We can see that we're connected to two interfaces, but there are a lot more. Usually, these interfaces also have a number following them that indexes different instances of the same type of interface.

1. **lo**: This is the loopback interface.
2. **wlan0**: For wireless connections
3. **tun**: When you are connected to VPN.
4. **en**:
5. **gif**:
6. **awd**:
7. **llw**:
8. **bridge**:
9. **utun**:

For each interface, there is a set of protocols that must be set for data to transfer.

12.4 Addresses

Every computer needs some address that determines its unique identity. The version of TCP/IP that has been in widespread use is IPv4, which uses 4-byte IP addresses. A modernized version, IPv6, expands the IP address space to 16 bytes and incorporates several additional features, making it faster and easier to implement.

Definition 12.5 (IP Address)**.** The protocol describes the use of **IP addresses** to uniquely identify Internet-connected devices (for transmission of data). That is, when a computer sends a message to another computer, it must specify the recipient's IP address and also include its own IP address so that the second computer can reply. There are two versions of the Internet Protocol in use today:

1. **IPv4**: The first version ever used on the Internet and having the form of 4 *octets* split by periods in between.

   $$[0-255].[0-255].[0-255].[0-255]$$

   Even though it presented in decimal, computers store them in binary

   $$74.125.20.113 \iff 01001011.01111101.00010100.01110001$$

   IPv4 addresses can take 2^{32} values, but IPv6 was created for more space.

2. **IPv6**: The newer standard (introduced in June 2012) is in the form

   $$\text{FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF:FFFF}$$

   with hexadecimal digits (total of 3.4×10^{39} possible IPv6 values).

Definition 12.6 (CIDR Notation)**.** Sometimes, a set of IP addresses are specified using **CIDR notation**. An address of the form

$$145.201.67.4/16$$

represents all addresses of form $145.201.*.*$.

Operating systems and network devices have supported IPv6 for a long time, and the motivation behind the deployment of IPv6 was due to the concern that devices were running out of IPv4 addresses. Asia ran out first in 2011, followed by every other continent ever since then.

But we've learned to make more efficient use of the IPv4 addresses that we have. For example, **Network Address Translation** (or **NAT**) lets entire networks of machines hide behind single IPv4 addresses. **Classless Inter-Domain Routing** (**CIDR**) subdivides networks and promotes efficient backbone routing as well. Ultimately, IPv6, with better security and engineering, is going to take over, but not for a while since it's not fundamentally different from IPv4 and the drawbacks of IPv4 haven't been bad enough to spark migration.

Definition 12.7 (Hierarchy of IP Addresses)**.** The IP addresses are formatted in an *hierarchical way*. The IPv4 address hierarchy is structured as such: The first few numbers (may or may not be divided by octets) could identify a **network** administered by an Internet Service Provider. The last numbers, which can also represent **subnetworks** (subnets), identifies a home computer on that network.

Example 12.4 (University of Michigan)**.** For example, if we represent the IP address 141.213.127.13 in binary (of 32 bits)

$$10001101.11010101.01111111.00001101$$

the first 16 bits could route to all of UMich, the next two bits could route to a specific UMich department, and the final 14 bits could route to individual computers.

1000110111010101	01	11111100001101
UMich Network	Medicine department	Lab computer

This hierarchy gives UMich the ability to differentiate between 2^2 departments and $2^{14} = 16,384$ computers within each department. In general, the ability to create hierarchical levels at any point in the IP address allows for greater flexibility in the size of each level of the hierarchy.

Example 12.5 (Duke)**.** Duke's IP addresses are of the form 153.3.___, with the DUKE-INTERCHANGE ISP provider.

Definition 12.8 (Hostname)**.** IP addresses can be quite cumbersome to memorize, which is why they are often addressed with their **hostname**. Operating systems allow one or more hostnames to be associated with an IP address so that users can type `rfc-editor.org` rather than 4.31.198.49. This mapping can be set up in multiple days, e.g. with the `/etc/hosts` file or the LDAP database system to DNS the world-wide **Domain Name System**.

12.4.1 LAN Addresses and NAT

We've talked about how entire networks of machines can hide behind a single IPv4 address. Let's elaborate on this. In fact, your computer is not connected to the internet directly. It is actually in a **private network**, or a *LAN network*, which uses a private IP address space (supported by both IPv4 and v6). Anything on the inside of your private network is not on the Internet; it is on your LAN, an entirely separate network, with its own address space. Anything on your LAN must have a unique (within the LAN) IP address to participate properly with your local network. Therefore, anyone else who has a LAN is also not part of the internet. So if you are only on your LAN network, how do you actually connect to the internet?

Definition 12.9 (Router)**.** The **router** is a device that forms a connection between your LAN network and the internet. It has both a private local address, called a **gateway address**, and a public address. It is responsible for forwarding data between the local server computers and the internet. Therefore, to the outside world, all devices identify the network internet activity by the one public IP address assigned to the router.

The gateway address can be found with `ip route` and the public address, of course, can be found with the commands previously mentioned.

Definition 12.10 (Modem)**.** A **modem**, short for **modulator/demodulator** is a device that converts a signal from your computer to some kind of signal to talk to other computers. The main difference between the router and the modem is that

1. The router crates a network between the computers in your home and routes network traffic between them (through Ethernet cables or wireless connection). Your home router has one connection to the Internet and connections to your private local network.
2. The modem serves as a bridge between your local network and the Internet.

To access our IP address, we can do the following:

1. To access local ip address, we can either run the command `hostname -i`, `ip -c a`, or `ifconfig`.
2. To access the public ip address, we can either google it or run `curl ifconfig.me`. Since this is public, any device connected to the same network/router should have the same IP address.

Definition 12.11 (NAT)**.** In order for LAN devices to connect to the Internet, their outgoing traffic has the source address changed to match that of the internet/WAN IP address of the router. The router keeps track of this, and makes sure any response traffic gets sent to the right internal machine. This is called **Network Address Translation (NAT)**. There are generally two types of NAT:

1. **Basic, one-to-one NAT**: The simplest type of NAT provides a one-to-one translation of IP addresses. In this type of NAT, only the IP addresses, IP header checksum, and any higher-level checksums that include the IP address are changed. Basic NAT can be used to interconnect two IP networks that have incompatible addressing.
2. **One-to-many NAT**: The majority of network address translators map multiple private hosts to one publicly exposed IP address. In a typical configuration, a local network uses one of the designated private IP address subnets. A router in that network has a private address of that address pace. The router it also connected to the Internet with a *public* address assigned by the ISP. As traffic passes from the local network to the Internet, the source address in each packet is translated on the fly from a private address to the public address. The router tracks basic data about each active connection (particularly the destination address and port). When a reply returns to the router, it uses the connection tracking data it stored during the outbound phase to determine the private address on the internal network to which to forward the reply.

Definition 12.12. The IP addresses that are in the private network's space are usually divided up into 3 categories. But as of now, the categories don't mean anything.

1. **Class A private range addresses**: 10.0.0.0 - 10.255.255.255 (16,777,216 IPs)
2. **Class B private range addresses**: 172.16.0.0 – 172.31.255.255 (1,048,576 IPs)
3. **Class C private range addresses**: 192.168.0.0 – 192.168.255.255 (65,536 IPs)

Since the private IPv4 address space is relatively small, many private IPv4 networks unavoidably use the same address ranges. This can create a problem when merging such networks, as some addresses may be duplicated for multiple devices. In this case, networks or hosts must be renumbered, often a time-consuming task, or a network address translator must be placed between the networks to translate or masquerade one of the address ranges.

12.4.2 Ports

IP addresses identify a machine's network interfaces, but they aren't specific enough to address individual processes or services, many of which might be actively using the network at once. TCP and UDP extend IP addresses with a concept known as a port, which is a 16-bit number that supplements an IP address to a particular communication channel. Valid ports range from 1 to $65,535$. A port, combined with an IP address, results in a **socket address** that is used to establish a connection between a client and a server.

UNIX systems restrict programs from binding to port numbers under 1024 unless they are run as root or have an appropriate Linux capability. Anyone can communicate with a server running on a low port number; the restriction only applies to the program listening on the port.

12.4.3 Hardware (MAC) Addresses

The lowest level of addressing is the network hardware. Many devices are assigned a unique 6-byte hardware address at the time of manufacture. The first 3 bytes identify the manufacturer, and the last 3 bytes are a unique serial number that the manufacturer assigns. Sysadmins can sometimes identify the brand of machine that is trashing a network by looking up the 3-byte identifier in a table of vendor IDs. In theory, ethernet hardware addresses are permanently assigned and immutable, but many network interfaces let you override the hardware address and set one of your own choosing.

12.5 TCP Packets and Encapsulation

12.6 OSI and Internet Protocols

12.7 HTTP and HTTPS

HTTP stands for hypertext transfer protocol, implemented in Layer 7, which transfers data between your computer and the server over the internet through **clear text**. This may not be the most ideal way since any interceptors can read the transferred data. This isn't a problem for regular internet browsing, but if you are inputting sensitive data, then HTTP should not be used. This is why HTTPS (which stands for secure HTTP) was invented, which is implemented in Layer 4 and encrypts the data being transferred, and every website where you input sensitive data should be using HTTPS (indicated by the `https://` prefix in the URL and a padlock symbol for modern browsers). Due to the extra security measures, HTTPS is less lightweight than HTTP, and its respective default ports are HTTP (80) and HTTPS (443).

A natural question to ask would be: which encryption scheme does HTTPS use? Both Secure Sockets Layer (SSL) and Transport Layer Security (TLS) is used in the modern web.

SSL certificate.

12.8 UDP and TCP

TCP handshake can be seen with curl.

12.9 SSH

13 Driver and Hardware Configuration

13.1 Audio Drivers

13.2 Bluetooth

Blueman.

13.3 Synaptics

13.4 Video Drivers

13.5 Monitor

13.6 Nvidia GPU Drivers

14 Development

14.1 Git

14.2 Python and Conda

Make sure to add conda path to PATH.