Navigating Linux System Commands

A guide for beginners to the Linux Shell and GNU coreutils

Sayan Ghosh

June 11, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on June 11, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

Preface **v**

Contents **vii**

1 Essentials of Linux **1**
1.1 Introduction . . . 1
1.1.1 What is Linux? . . . 1
1.1.2 Desktop Environments . . . 2
1.1.3 Window Managers . . . 3
1.1.4 Why Linux? . . . 4
1.1.5 What is Shell? . . . 4
1.1.6 Shell vs Terminal . . . 4
1.1.7 Why the Command Line? . . . 5
1.1.8 Command Prompt . . . 6
1.2 Simple Commands in GNU Core Utils . . . 7
1.2.1 File System Navigation . . . 8
1.2.2 Manuals . . . 9
1.2.3 System Information . . . 11
1.2.4 File Management . . . 14
1.2.5 Text Processing and Pagers . . . 17
1.2.6 Aliases and Types of Commands . . . 21
1.2.7 User Management . . . 24
1.2.8 Date and Time . . . 25
1.3 Navigating the File System . . . 28
1.3.1 What is a File System? . . . 28
1.3.2 In Memory File System . . . 29
1.3.3 Paths . . . 32
1.3.4 Basic Commands for Navigation . . . 33
1.4 File Permissions . . . 35
1.4.1 Read . . . 36
1.4.2 Write . . . 36
1.4.3 Execute . . . 36
1.4.4 Interesting Caveats . . . 36
1.4.5 Changing Permissions . . . 38
1.4.6 Special Permissions . . . 38
1.4.7 Octal Representation of Permissions . . . 40
1.5 Types of Files . . . 43
1.5.1 Regular Files . . . 43
1.5.2 Directories . . . 43
1.5.3 Symbolic Links . . . 43
1.5.4 Character Devices . . . 43
1.5.5 Block Devices . . . 44
1.5.6 Named Pipes . . . 44
1.5.7 Sockets . . . 45
1.5.8 Types of Regular Files . . . 45

1.6 Inodes and Links . . . 47
1.6.1 Inodes . . . 47
1.6.2 Separation of Data, Metadata, and Filename . . . 48
1.6.3 Directory Entries . . . 49
1.6.4 Hard Links . . . 49
1.6.5 Symbolic Links . . . 50
1.6.6 Symlink vs Hard Links . . . 52
1.6.7 Identifying Links . . . 52
1.6.8 What are . and ..? . . . 53

2 Command Line Editors **55**
2.1 Introduction . . . 55
2.1.1 Types of Editors . . . 55
2.1.2 Why Command Line Editors? . . . 55
2.1.3 Mouse Support . . . 55
2.1.4 Editor war . . . 56
2.1.5 Differences between Vim and Emacs . . . 56
2.1.6 Nano: The peacemaker amidst the editor war . . . 58
2.2 Vim . . . 58
2.2.1 History . . . 58
2.2.2 Ed Commands . . . 65
2.2.3 Exploring Vim . . . 72
2.3 Emacs . . . 80
2.3.1 History . . . 80
2.3.2 Exploring Emacs . . . 82
2.4 Nano . . . 84
2.4.1 History . . . 84
2.4.2 Exploring Nano . . . 85
2.4.3 Editing A Script in Nano . . . 85

List of Figures

1.1 Linux Distributions Usage . . . 2
1.2 Desktop Environment Usage . . . 3
1.3 Operating System Onion Rings . . . 4
1.4 GNU Core Utils Logo . . . 7
1.5 `ls -l` Output . . . 8
1.6 Linux Filesystem Hierarchy . . . 28
1.7 Relative Path . . . 33
1.8 File Permissions . . . 35
1.9 Octal Permissions . . . 41
1.10 System Calls . . . 48
1.11 Inode and Directory Entry . . . 49
1.12 Directed Acyclic Graph . . . 50
1.13 Abstract Representation of Symbolic Links and Hard Links . . . 52
1.14 Symbolic Links and Hard Links . . . 52

2.1 A Teletype . . . 59
2.2 Ken Thompson . . . 59
2.3 Dennis Ritchie . . . 59
2.4 Xerox Alto, one of the first VDU terminals with a GUI, released in 1973 . . . 60
2.5 A first generation Dectape (bottom right corner, white round tape) being used with a PDP-11 computer . . . 60
2.6 George Coulouris . . . 61
2.7 Bill Joy . . . 61
2.9 Stevie Editor . . . 62
2.8 The Keyboard layout of the ADM-3A terminal . . . 62
2.10 Bram Moolenaar . . . 63
2.11 The inital version of Vim, when it was called Vi IMitation . . . 64
2.12 Vim 9.0 Start screen . . . 64
2.13 Neo Vim Window editing this book . . . 64
2.14 Simplified Modes in Vim . . . 72
2.15 Detailed Modes in Vim . . . 73
2.16 Vim Cheat Sheet . . . 79
2.18 Richard Stallman - founder of GNU and FSF projects . . . 80
2.17 Emacs Logo . . . 80
2.19 Guy L. Steele Jr. combined many divergent TECO with macros to create EMACS . . . 81
2.20 James Gosling - creator of Gosling Emacs and later Java . . . 81
2.21 ADM-3A terminal . . . 81
2.22 Space Cadet Keyboard . . . 82
2.23 Nano Text Editor . . . 84

List of Tables

1.1 Basic Shortcuts in Terminal . . . 5
1.2 Basic Commands in GNU Core Utils . . . 7

1.3 Manual Page Sections . . . 10
1.4 Keys in Info Pages . . . 11
1.5 Escape Characters in `echo` . . . 18
1.6 Date Format Specifiers . . . 26
1.7 Linux Filesystem Hierarchy . . . 28
1.8 Linux Filesystem Directory Classification . . . 29
1.9 Octal Representation of Permissions . . . 41
1.10 Types of Files . . . 43
1.11 Metadata of a File . . . 48
1.12 Symlink vs Hard Link . . . 52

2.1 History of Vim . . . 58
2.2 Ed Commands . . . 65
2.3 Commands for location . . . 65
2.4 Commands for Editing . . . 65
2.5 Ex Commands in Vim . . . 73
2.6 Navigation Commands in Vim . . . 74
2.7 Moving the Screen Commands in Vim . . . 74
2.8 Replacing Text Commands in Vim . . . 75
2.9 Toggling Case Commands in Vim . . . 75
2.10 Deleting Text Commands in Vim . . . 76
2.11 Deleting Text Commands in Vim . . . 76
2.12 Address Types in Search and Replace . . . 78
2.13 Keys to enter Insert Mode . . . 78
2.14 History of Emacs . . . 80
2.15 Navigation Commands in Emacs . . . 82
2.16 Exiting Emacs Commands . . . 83
2.17 Searching Text Commands in Emacs . . . 83
2.18 Copying and Pasting Commands in Emacs . . . 83
2.19 File Handling Commands in Nano . . . 85
2.20 Editing Commands in Nano . . . 85

Essentials of Linux 1

1.1 Introduction

1.1.1 What is Linux?

Definition 1.1.1 (Linux) Linux is a **kernel** that is used in many operating systems. It is open source and free to use. Linux is not an operating system unto itself, but the core component of it.

So what is **Ubuntu**? **Ubuntu** is one of the many *distributions* that use the Linux kernel. It is a complete operating system that is free to use and open source. It is based on the **Debian** distribution of Linux. There are many other *distributions* of Linux, such as:

- **Debian** - Used primarily on servers, it is known for its stability.
  - **Ubuntu** - A commercial distribution based on Debian which is popular among new users.
  - **Linux Mint** - A distribution based on Ubuntu which is known for its ease of use. It is one of the distributions recommended to new users.
  - **Pop OS** - A distribution based on Ubuntu which is known for its focus on developers, creators, and gamers.
  - and many more
- **Red Hat Enterprise Linux (RHEL)** - A commercial distribution used primarily in enterprises. It is owned by **Red Hat** and is targeted primarily to companies with their free OS paid support model.
  - **Fedora** - A community-driven distribution sponsored by **Red Hat**. It is known for its cutting-edge features and is used by developers. It remains on the upstream of **RHEL**, receiving new features before **RHEL**.
  - **CentOS** - A discontinued distribution based on **RHEL**. It was known for its stability and was used in servers. It was downstream from **RHEL**.
  - **CentOS Stream** - It is a midstream between the upstream development in Fedora Linux and the downstream development for Red Hat Enterprise Linux.
  - **Rocky Linux** - A distribution created by the **Rocky Enterprise Software Foundation** after the announcement of discontinuation of **CentOS**. It is a downstream of **RHEL** that provides feature parity and binary compatibility with **RHEL**.
  - **Alma Linux** - A distribution created by the **CloudLinux** team after the announcement of discontinuation of **CentOS**. It is a downstream of **RHEL** that provides feature parity and binary compatibility with **RHEL**.
- **Arch Linux** - A community-driven distribution known for its simplicity and customizability. It is a *rolling release* distribution, which means that it is continuously updated. It is a bare-bones

1.1 Introduction 1
1.1.1 What is Linux? 1
1.1.2 Desktop Environments . 2
1.1.3 Window Managers 3
1.1.4 Why Linux? 4
1.1.5 What is Shell? 4
1.1.6 Shell vs Terminal 4
1.1.7 Why the Command Line? 5
1.1.8 Command Prompt 6
1.2 Simple Commands in GNU Core Utils 7
1.2.1 File System Navigation . 8
1.2.2 Manuals 9
1.2.3 System Information . . . 11
1.2.4 File Management 14
1.2.5 Text Processing and Pagers 17
1.2.6 Aliases and Types of Commands 21
1.2.7 User Management 24
1.2.8 Date and Time 25
1.3 Navigating the File System 28
1.3.1 What is a File System? . . 28
1.3.2 In Memory File System . 29
1.3.3 Paths 32
1.3.4 Basic Commands for Navigation 33
1.4 File Permissions 35
1.4.1 Read 36
1.4.2 Write 36
1.4.3 Execute 36
1.4.4 Interesting Caveats 36
1.4.5 Changing Permissions . . 38
1.4.6 Special Permissions . . . 38
1.4.7 Octal Representation of Permissions 40
1.5 Types of Files 43
1.5.1 Regular Files 43
1.5.2 Directories 43
1.5.3 Symbolic Links 43
1.5.4 Character Devices 43
1.5.5 Block Devices 44
1.5.6 Named Pipes 44
1.5.7 Sockets 45
1.5.8 Types of Regular Files . . 45
1.6 Inodes and Links 47
1.6.1 Inodes 47
1.6.2 Separation of Data, Metadata, and Filename 48
1.6.3 Directory Entries 49
1.6.4 Hard Links 49
1.6.5 Symbolic Links 50
1.6.6 Symlink vs Hard Links . 52
1.6.7 Identifying Links 52
1.6.8 What are . and ..? 53

distribution that lets the user decide which packages they want to install.

- **Manjaro** - A distribution based on Arch Linux which is known for its user-friendliness. It is a rolling release distribution that is easier to install for new users. It uses a different repository for packages with additional testing.
- **EndeavourOS** - A distribution based on Arch Linux which is known for its simplicity and minimalism. It is a rolling release distribution that is easier to install for new users. It uses the same repository for packages as **Arch Linux**.
- **Artix Linux** - It uses the **OpenRC** init system instead of **systemd**. It also offers other *init systems* like **runit**, **s6**, **dinit**.

- **openSUSE** - It is a free and open-source Linux distribution developed by the openSUSE project. It is offered in two main variations: **Tumbleweed**, an upstream rolling release distribution, and **Leap**, a stable release distribution which is sourced from SUSE Linux Enterprise.
  - **Tumbleweed** - Rolling Release upstream.
  - **Leap** - Stable Release downstream.
- **Gentoo** - A distribution known for its customizability and performance. It is a source-based distribution, which means that the user compiles the software from source code. It is known for its performance optimizations for the user's hardware.
- **Void** - It is an independent rolling-release Linux distribution that uses the X Binary Package System package manager, which was designed and implemented from scratch, and the **runit** init system. Excluding binary kernel blobs, a base install is composed entirely of free[1] software.

1: "Free software" means software that respects users' freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software. Thus, "free software" is a matter of liberty, not price. To understand the concept, you should think of "free" as in "free speech," not as in "free beer." We sometimes call it "libre software," borrowing the French or Spanish word for "free" as in freedom, to show we do not mean the software is gratis.

You may have paid money to get copies of a free program, or you may have obtained copies at no charge. But regardless of how you got your copies, you always have the freedom to copy and change the software, even to sell copies.

- GNU on Free Software

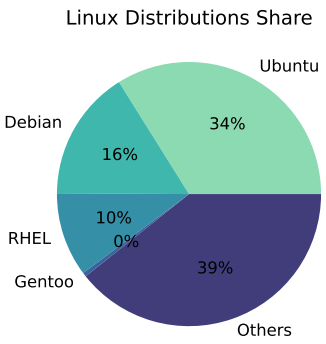

Figure 1.1: Linux Distributions Usage in 2024

1.1.2 Desktop Environments

Definition 1.1.2 (Desktop Environment) A desktop environment is a collection of software designed to give functionality and a certain look and feel to a desktop operating system. It is a combination of a window manager, a file manager, a panel, and other software that provides a graphical user interface and utilities to a regular desktop user, such as volume and brightness control, multimedia applications, settings panels, etc. This is only required by desktop (and laptop) uses and are not present on server instances.

There are many desktop environments available for Linux, but the important ones are:

- **GNOME** - One of the most popular desktop environments for Linux. It is known for its simplicity and ease of use. It is the default desktop environment for many distributions, including Ubuntu. It is based on the **GTK** Toolkit. [2] Popular distros shipping by default with GNOME are Fedora, RHEL, CentOS, Debian, Zorin, and Ubuntu. [3]
- **KDE Plasma** - A highly customizable desktop environment based on the **Qt** Tookit. [4] Many distributions like Slackware and Open-

2: GTK is a free software cross-platform widget toolkit for creating graphical user interfaces.

3: Ubuntu used to ship with Unity as the default desktop environment, but switched to GNOME in 2017.

4: Qt is cross-platform application development framework for creating graphical user interfaces.

SUSE ship with KDE Plasma as the default desktop environment, and most others have the option to install with KDE Plasma. Ubuntu's KDE Plasma variant is called **Kubuntu**.

- **Xfce** - A lightweight desktop environment known for its speed and simplicity. It is based on the **GTK** Toolkit. It is used in many distributions like Xubuntu, Manjaro, and Fedora.
- **LXQt** - A lightweight desktop environment known for its speed and simplicity. It is based on the **Qt** Toolkit. It is used in many distributions like Lubuntu.
- **Cinnamon**
- **MATE**

It is important to note that although some distributions come pre-bundled with certain Desktop Environments, it doesn't mean that you cannot use another DE with it. DE are simply packages installed on your distribution, and almost all the popular DEs can be installed on all distributions. Many distributions also come with multiple pre-bundled desktop environments due to user preferences. Most server distributions and some enthusiast distributions come with no pre-bundled desktop environment, and let the user determine which one is required, or if one is required.

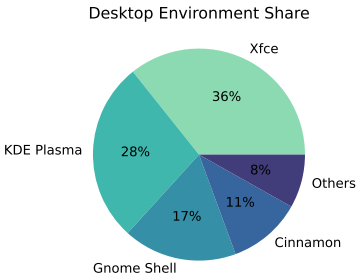

Figure 1.2: Desktop Environment Usage in 2022

1.1.3 Window Managers

Definition 1.1.3 (Window Manager) A window manager is system software that controls the placement and appearance of windows within a windowing system in a graphical user interface. It is a part of the desktop environment, but can also be used standalone. It is responsible for the appearance and placement of windows, and can also provide additional functionality like virtual desktops, window decorations, window title bars, and tiling.

Although usually bundled with a desktop environment, many window managers are also standalone and installed separately by the user if they don't want to use all the application from a single desktop environment.

Some popular window managers are:

- **Openbox** - A lightweight window manager known for its speed and simplicity. It is used in many distributions like Lubuntu.
- **i3** - It is a tiling window manager [5] which is usually one of the first window managers that users try when they want to move away from a desktop environment and to a tiling window manager.
- **awesome** - A tiling window manager that is highly configurable and extensible. It is written in Lua and is known for its beautiful configurations.
- **bspwm** - A tiling window manager. It is based on binary space partitioning.
- **dwm** - A dynamic tiling window manager that is known for its simplicity and minimalism. It is written in C and is highly configurable.

5: A tiling window manager is a window manager that automatically splits the screen into non-overlapping frames, which are used to display windows. Most desktop environments ship with a **floating** window manager instead, which users of other operating systems are more familiar with.

1.1.4 Why Linux?

You might be wondering *"Why should I use Linux?"* Most people use either **Windows** or **Mac** on their personal computers. Although these consumer operating systems get the job done, they don't let the user completely control their own *hardware* and *software*. Linux [6] is a free and open-source operating system that gives the user complete control over their system. It is highly customizable and can be tailored to the user's needs. It is also known for its stability and security. It is used in almost all servers, supercomputers, and embedded systems. It is also used in many consumer devices like Android phones, smart TVs, and smartwatches.

In this course we will be covering how to navigate the linux file system, how to manage files, how to manage the system, and how to write scripts to automate tasks. In the later part of the course we go over concepts such as pattern matching and text processing.

This course does not go into details of the linux kernel, but rather attempts to make the reader familiar with the *GNU core utils* and able to navigate around a linux server easily.

6: Although Linux is just a kernel and not an entire operating system, throughout this book I would be referring to **GNU/Linux**, the combination of **GNU core utilities** and the Linux kernel, as **Linux** in short.

1.1.5 What is Shell?

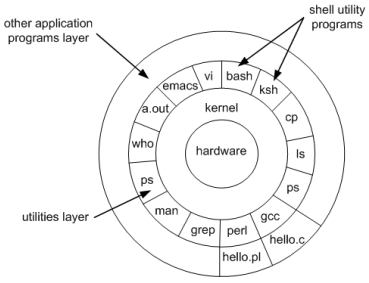

Figure 1.3: Operating System Onion Rings - The layers of an operating system

The **kernel** is the core of the operating system. It is responsible for managing the hardware and providing services to the user programs. The **shell** is the interface between the user and the kernel (Figure 1.3). Through the **shell** we can run many commands and utilities, as well as some inbuilt features of the shell.

Definition 1.1.4 (Shell) A shell is a command-line interpreter that provides a way for the user to interact with the operating system. It takes commands from the user and executes them. It is a program that provides the user with a way to interact with the operating system.

The most popular shell in Linux is the **bash** shell. It is the default shell in most distributions. It is a POSIX-compliant [7] shell. There are many other shells available, such as **zsh**, **fish** [8] , **dash**, **csh**, **ksh**, and **tcsh**. Each shell has its own features and syntax, but most of the keywords and syntax are the same. In this course we will be covering only the **bash** shell and its syntax, but most of what we learn here is also applicable on other shells as well.

7: POSIX, or Portable Operating System Interface, is a set of standards that define the interfaces and environment that operating systems use to access POSIX-compliant applications. POSIX standards are based on the Unix operating system and were released in the late 1980s.

8: Fish is a non-POSIX compliant shell that is known for its features like auto-suggestion, syntax highlighting, and tab completions. Although a useful alternative to other shells for scripting, it should not be set as the default shell.

1.1.6 Shell vs Terminal

Definition 1.1.5 (Terminal) A terminal is a program that provides a way to interact with the shell. It is a program that provides a text-based interface to the shell. It is also known as a terminal emulator.

The terminal is the window that you see when you open a terminal program. It provides a way to interact with the shell. The shell is the program that interprets the commands that you type in the terminal.

The terminal is the window that you see, and the shell is the program that runs in that window. Whereas the shell is the application that is parsing your input and running the commands and keywords, the terminal is the application that lets you see the shell graphically. There are multiple different terminal emulators, providing a lot of customization and beautification to the terminal, as well as providing useful features such as *scroll back*, copying and pasting, and so on.

Some popular terminal emulators are:

- **gnome-terminal** - The default terminal emulator for the GNOME desktop environment.
- **konsole** - The default terminal emulator for the KDE desktop environment.
- **xfce4-terminal** - The default terminal emulator for the Xfce desktop environment.
- **alacritty** - A terminal emulator known for its speed and simplicity.
- **terminator** - A terminal emulator known for its features like splitting the terminal into multiple panes.
- **tilix** - A terminal emulator known for its features like splitting the terminal into multiple panes.
- **st** - A simple terminal emulator known for its simplicity and minimalism.
- **urxvt**
- **kitty**
- **terminology**

In most terminal emulators, there are some basic shortcuts that can be used to make the terminal experience more efficient. Some of the basic shortcuts are listed in Table Table 1.1.

Table 1.1: Basic Shortcuts in Terminal

Shortcut	Description
`Ctrl + C`	Terminate the current process
`Ctrl + D`	Exit the shell
`Ctrl + L`	Clear the terminal screen
`Ctrl + A`	Move the cursor to the beginning of the line
`Ctrl + E`	Move the cursor to the end of the line
`Ctrl + U`	Delete from the cursor to the beginning of the line
`Ctrl + K`	Delete from the cursor to the end of the line
`Ctrl + W`	Delete the word before the cursor
`Ctrl + Y`	Paste the last deleted text
`Ctrl + R`	Search the command history
`Ctrl + Z`	Suspend the current process
`Ctrl + \`	Terminate the current process
`Ctrl + S`	Pause the terminal output
`Ctrl + Q`	Resume the terminal output

1.1.7 Why the Command Line?

Both the command line interface (CLI) and the graphical user interface (GUI) are simply shells over the operating system's kernel. They let you interact with the kernel, perform actions and run applications.

GUI:

The GUI requires a mouse and a keyboard, and is more intuitive and easier to use for beginners. But it is also slower and less efficient than the CLI. The GUI severely limits the user's ability to automate tasks and perform complex operations. The user can only perform those operations that the developers of the GUI have thought of and implemented.

CLI:

The CLI is faster and more efficient than the GUI as it lets the user use the keyboard to perform actions. Instead of clicking on pre-defined buttons, the CLI lets you construct your instruction to the computer using syntax and semantics. The CLI lets you combine simple commands that do one thing well to perform complex operations. The biggest advantage of the CLI is that it lets you automate tasks. It might be faster for some users to rename a file from **file1** to **file01** using the GUI, but it will always be faster to automate this using the CLI if you want to do this for thousands of files in the folder.

In this course we will be learning how to use the CLI to navigate the file system, manage files, manage the system, process text, and write scripts to automate tasks.

1.1.8 Command Prompt

The command prompt is the text that is displayed in the terminal to indicate that the shell is ready to accept commands. It usually ends with a `$` or a `#` symbol. The `$` symbol indicates that the shell is running as a normal user, and the `#` symbol indicates that the shell is running as the root user. The root user has complete control over the system and can perform any operation on the system.

An example of a command prompt is:

```
1 username@hostname:~$
```

Here, `username` is the name of the user, `hostname` is the name of the computer, and `$` indicates that the shell is running as a normal user. The `~` symbol indicates that the current working directory is the user's home directory. [9] This prompt can be changed and customized according to the user's preferences using the `PS1` variable discussed in Chapter **??**.

9: The home directory is the directory where the user's files and settings are stored. It is usually located at `/home/username`. This can be shorted to `~` in the shell.

1.2 Simple Commands in GNU Core Utils

Definition 1.2.1 (GNU Core Utils) The GNU Core Utilities are the basic file, shell, and text manipulation utilities of the GNU operating system. These are the utilities that are used to interact with the operating system and perform basic operations. [a]

[a] GNU Core Utils

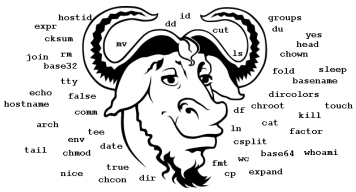

Figure 1.4: GNU Core Utils Logo

The shell lets you simply type in the name of the command and press enter to run it. You can also pass arguments to the command to modify its behavior. Although the commands are simple, they are powerful and can be combined to perform complex operations. [10]

10: The combination of commands to perform complex operations is called *piping*. This will be covered later.

Some basic commands in the core-utils are listed in Table Table 1.2.

Table 1.2: Basic Commands in GNU Core Utils

Command	Description
`ls`	List the contents of a directory
`cd`	Change the current working directory
`pwd`	Print the current working directory
`mkdir`	Create a new directory
`rmdir`	Remove a directory
`touch`	Create a new file
`rm`	Remove a file
`cp`	Copy a file
`mv`	Move a file
`echo`	Print a message
`cat`	Concatenate and display the contents of a file
`less`	Display the contents of a file one page at a time
`head`	Display the first few lines of a file
`tail`	Display the last few lines of a file
`find`	Find files and directories
`locate`	Find files and directories
`which`	Find the location of a command
`uname`	Print system information
`ps`	Display information about running processes
`kill`	Terminate a process
`chmod`	Change the permissions of a file
`chown`	Change the owner of a file
`chgrp`	Change the group of a file
`date`	Print the current date and time
`cal`, `ncal`	Print a calendar
`df`	Display disk space usage
`du`	Display disk usage
`free`	Display memory usage
`top`	Display system information
`history`	Display the command history
`sleep`	Pause the shell for a specified time
`true`	Do nothing, successfully
`false`	Do nothing, unsuccessfully
`tee`	Read from stdin and write to stdout and files
`whoami`	Print the current user
`groups`	Print the groups the user belongs to
`clear`	Clear the terminal screen
`exit`	Exit the shell

1.2.1 File System Navigation

pwd:

The `pwd` command prints the current working directory. The current working directory is the directory that the shell is currently in. The shell starts in the user's home directory when it is opened. The `pwd` command prints the full path of the current working directory.

```
$ pwd
/home/username
```

ls:

The `ls` command lists the contents of a directory. By default, it lists the contents of the current working directory. The `ls` command can take arguments to list the contents of a different directory.

```
$ ls
Desktop  Documents  Downloads  Music  Pictures  Videos
```

We can also list hidden files [11] using the `-a` flag.

```
$ ls -a
.  ..  .bashrc  Desktop  Documents  Downloads  Music  Pictures
  Videos
```

Here, the `.` and `..` directories are special directories. The `.` directory is the current directory, and the `..` directory is the parent directory. The `.bashrc` file is a configuration file for the shell which is a hidden file.

`ls` can also list the details of the files using the `-l` flag.

```
$ ls -l
total 24
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Desktop
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Documents
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Downloads
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Music
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Pictures
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Videos
```

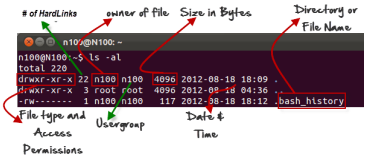

Figure 1.5: `ls -l` Output

As seen in Figure 1.5, the first column is the file type and permissions. The second column is the number of links to the file or directory. The third and fourth columns are the owner and group of the file or directory. The fifth column is the size of the file or directory. The sixth, seventh, and eighth columns are the last modified date and time of the file or directory. The ninth column is the name of the file or directory. [12]

We can also list the inode numbers [13] using the `-i` flag.

```
$ ls -i
123456 Desktop  123457 Documents  123458 Downloads  123459 Music
    123460 Pictures  123461 Videos
```

Inodes will be discussed in detail later in the course.

cd:

The `cd` command changes the current working directory. It takes the path to the directory as an argument.

11: Hidden files are files whose names start with a dot. They are hidden by default in the `ls` command.

12: More details about the file permissions and the file types will be covered later in the course.

13: An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data. This includes the file type, the file's owner, the file's group, the file's permissions, the file's size, the file's last modified date and time, and the file's location on the disk. The inode is the reference pointer to the data in the disk.

```
$ cd Documents
$ pwd
/home/username/Documents
```

The `cd` command can also take the ~ symbol as an argument to change to the user's home directory. This is the default behavior of the `cd` command when no arguments are passed.

```
$ cd
$ pwd
/home/username
```

If we want to go back to the previous directory, we can use the `-` symbol as an argument to the `cd` command. [14]

14: This internally uses the `OLDPWD` environment variable to change the directory. More about variables will be covered later in the course.

```
$ cd Documents
$ pwd
/home/username/Documents
$ cd -
$ pwd
/home/username
```

Question 1.2.1 `ls` can only show files and directories in the **cwd**[15], not subdirectories. True or False?

15: **cwd** means **Current Working Directory**

Answer 1.2.1 False. `ls` can show files and directories in the cwd, and also in subdirectories. The `-R` flag can be used to show files and directories in subdirectories, recursively.

1.2.2 Manuals

man:

How to remember so many flags and options for each of the commands? The `man` command is used to display the manual pages for a command.

Definition 1.2.2 (Manual Pages) Manual pages are a type of software documentation that provides details about a command, utility, function, or file format. They are usually written in a simple and concise manner and provide information about the command's syntax, options, and usage.

```
$ man ls
```

This will display the manual page for the `ls` command. The manual page is divided into sections, and you can navigate through the sections using the arrow keys. Press `q` to exit the manual page.

Example manual page:

```

LS(1)                    User Commands                    LS(1)

```

```
NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by
    default).  Sort entries alphabetically if none of -cftuvSUX
    nor --sort is specified.

       Mandatory arguments to long options are mandatory for short
     options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..
...
```

The manual page provides information about the command, its syntax, options, and usage. It is a good practice to refer to the manual page of a command before using it.

To exit the manual page, press `q`.

There are multiple sections in the manual page, `man` takes the section number as an argument to display the manual page from that section.

```
$ man 1 ls
```

This will display the manual page for the `ls` command from section 1. The details of the sections can be seen in Table Table 1.3.

Table 1.3: Manual Page Sections

Section	Description
1	User Commands
2	System Calls
3	Library Functions
4	Special Files usually found in `/dev`
5	File Formats and conventions
6	Games
7	Miscellaneous
8	System Administration
9	Kernel Developer's Manual

Man pages only provide information about the commands and utilities that are installed on the system. They do not provide information about the shell builtins or the shell syntax. For that, you can refer to the shell's documentation or use the **help** command.

Some commands also have a `--help` flag that displays the usage and options of the command.

Some commands have their own **info** pages, which are more detailed than the **man** pages.

To be proficient with shell commands, one needs to read the man, info, and help pages. [16]

16: An useful video by Robert Elder about the differences between **man**, **info**, and **help** can be found on YouTube.

Exercise 1.2.1 Run `man`, `info`, and `--help` on all the commands discussed in this section. Note the differences in the information provided by each of them. Read the documentations carefully and try to understand how each command works, and the pattern in which the documentations are written.

info:

The `info` command is used to display the info pages for a command. The info pages are more detailed than the **man** pages for some commands. It is navigable like a hypertext document. There are links to chapters inside the info pages that can be followed using the arrow keys and entered using the enter key. The Table Table 1.4 lists some of the keys that can be used to navigate the info pages.

Table 1.4: Keys in Info Pages

Key	Description
`h`	Display the help page
`q`	Exit the info page
`n`	Move to the next node
`p`	Move to the previous node
`u`	Move up one level
`d`	Move down one level
`l`	Move to the last node
`t`	Move to the top node
`g`	Move to a specific node
`<enter>`	Follow the link
`m`	Display the menu
`s`	Search for a string
`S`	Search for a string (case-sensitive)

help:

The `help` command is a shell builtin that displays information about the shell builtins and the shell syntax.

```
1 $ help read
```

This will list the information about the `read` builtin command.

The `help` command can also be used to display information about the shell syntax.

```
1 $ help for
```

This will list the information about the `for` loop in the shell.

Help pages are not paged, and the output is displayed in the terminal. To page the output, one can use the `less` command.

```
1 $ help read | less
```

1.2.3 System Information

uname:

The `uname` command prints system information. It can take flags to print specific information about the system. By default, it prints only the kernel name.

```
$ uname
Linux
```

The `-a` flag prints all the system information. [17]

```
$ uname -a
Linux rex 6.8.2-arch2-1 #1 SMP PREEMPT_DYNAMIC Thu, 28 Mar 2024
  17:06:35 +0000 x86_64 GNU/Linux
```

17: Here `rex` is the hostname of the system, `6.8.2-arch2-1` is the kernel version, `x86_64` is the architecture, and `GNU/Linux` is the operating system.

ps:

The `ps` command displays information about running processes. By default, it displays information about the processes run by the current user that are running from a terminal. [18]

```
$ ps
  PID TTY          TIME CMD
12345 pts/0    00:00:00 bash
12346 pts/0    00:00:00 ps
```

18: The **PID** is the process ID, the **TTY** is the terminal the process is running from, the **TIME** is the time the process has been running, and the **CMD** is the command that is running.

There are a lot of flags that can be passed to the `ps` command to display more information about the processes. These will be covered in Chapter **??**.

Remark 1.2.1 **ps** has three types of options:

- UNIX options
- BSD options
- GNU options

The UNIX options are the preceded by a hyphen (-) and may be grouped. The BSD options can be grouped, but should not be preceded by a hyphen (-). The GNU options are preceded by two hyphens (–). These are also called long options.

The same action can be performed by using different options, for example, `ps -ef` and `ps aux` are equivalent, although first is using **UNIX** options, and the latter is using **BSD** options.

Another difference in **GNU core utils** and **BSD utils** is that the **GNU** utils have long options, whereas the **BSD** utils do not.

BSD utils also usually do not support having flags after the positional arguments, whereas most **GNU** utils are fine with this.

kill:

The `kill` command is used to terminate a process. It takes the process ID as an argument.

```
$ kill 12345
```

The `kill` command can also take the **signal** number as an argument to send a signal to the process. For example, the `SIGKILL` signal can be sent to the process to terminate it. [19]

```
$ kill -9 12345
```

19: The `SIGKILL` signal is used to terminate a process immediately. It cannot be caught or ignored by the process. It is numbered as 9.

free:

The `free` command is used to display the amount of free and used memory in the system.

```
$ free
              total        used        free      shared  buff/
  cache   available
Mem:        8167840     1234560     4567890      123456
  2367890     4567890
Swap:       2097148           0     2097148
```

The `free` command can take the `-h` flag to display the memory in human-readable format.

```
$ free -h
              total        used        free      shared  buff/
  cache   available
Mem:          7.8Gi       1.2Gi       4.3Gi       120Mi
  2.3Gi       4.3Gi
Swap:         2.0Gi          0B       2.0Gi
```

df:

The `df` command is used to display the amount of disk space available on the filesystems.

```
$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1       12345678  1234567  11111111  10% /
/dev/sda2       12345678  1234567  11111111  10% /home
```

The `df` command can take the `-h` flag to display the disk space in human-readable format.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        12G  1.2G  9.9G  11% /
/dev/sda2        12G  1.2G  9.9G  11% /home
```

du:

The `du` command is used to display the disk usage of directories and files. By default, it displays the disk usage of the current directory.

```
$ du
4       ./Desktop
4       ./Documents
4       ./Downloads
4       ./Music
4       ./Pictures
4       ./Videos
28
```

The `du` command can take the `-h` flag to display the disk usage in human-readable format. The `-s` flag displays the total disk usage of the directory.

```
$ du -sh
28K     .
```

Question 1.2.2 How to print the kernel version of your system?

Answer 1.2.2 `uname -r` will print the kernel version of your system.

`uname` is a command to print system information. The `-r` flag is to print the kernel release. There are other flags to print other system information.
We can also run `uname -a` to get all fields and extract only the kernel info using commands taught in later weeks.

Question 1.2.3 How to see how long your system is running for? What about the time it was booted up?

Answer 1.2.3 `uptime` will show how long the system is running for.
`uptime -s` will show the time the system was booted up.
The `-s` flag is to show the time of last boot.

Question 1.2.4 How to see the amount of free memory? What about free hard disk space? If we are unable to understand the big numbers, how to convert them to human readable format? What is difference between MB and MiB?

Answer 1.2.4 `free` will show the amount of free memory.
`df` will show the amount of free hard disk space.
`df -h` and `free -h` will convert the numbers to human readable format.
MB is Megabyte, and MiB is Mebibyte.
1 MB = 1000 KB, 1 GB = 1000 MB, 1 TB = 1000 GB, this is SI unit.
1 MiB = 1024 KiB, 1 GiB = 1024 MiB, 1 TiB = 1024 GiB, this is 2^{10} unit.

1.2.4 File Management

file:

The `file` command is used to determine the type of a file. It can take multiple file names as arguments.

```
$ file file1
file1: ASCII text
$ file /bin/bash
/bin/bash: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV
  ), dynamically linked, interpreter /lib64/ld-linux-x86-64.so
  .2, for GNU/Linux 3.2.0, BuildID[sha1]=1234567890abcdef,
  stripped
```

mkdir:

The `mkdir` command is used to create new directories. It can take multiple directory names as arguments.

```
$ mkdir a b c
$ ls -F
a/  b/  c/
```

Exercise 1.2.2 Run `man ls` to find out what the `-F` flag does, and why we used it in the above example.

touch:

The `touch` command is used to create new files. It can take multiple file names as arguments. If a file is already present, the `touch` command updates the last modified date and time of the file, but does not modify the contents of the file.

```
$ touch file1 file2 file3
$ ls -l
-rw-r--r-- 1 username group 0 Mar  1 12:00 file1
-rw-r--r-- 1 username group 0 Mar  1 12:00 file2
-rw-r--r-- 1 username group 0 Mar  1 12:00 file3
$ sleep 60
$ touch file3
$ ls -l
-rw-r--r-- 1 username group 0 Mar  1 12:00 file1
-rw-r--r-- 1 username group 0 Mar  1 12:00 file2
-rw-r--r-- 1 username group 0 Mar  1 12:01 file3
```

Exercise 1.2.3 Notice the difference in the last modified date and time of the `file3` file from the other files. Also notice the `sleep` command used to pause the shell for 60 seconds.

rmdir:

The `rmdir` command is used to remove directories. It can take multiple directory names as arguments.

```
$ mkdir a b c d
$ rmdir a b c
$ ls -F
d/
```

Remark 1.2.2 The `rmdir` command can only remove empty directories. This is a safety feature so that users dont accidentally delete directories with files in them. To remove directories with files in them along with those files, use the `rm` command.

rm:

The `rm` command is used to remove files and directories. It can take multiple file and directory names as arguments.

```
$ touch file1 file2 file3
$ ls -F
file1  file2  file3
$ rm file1 file2
$ ls -F
file3
```

However, using `rm` to delete a directory will give an error.

```
$ mkdir a
$ rm a
```

```
rm: cannot remove 'a': Is a directory
```

This is because the `rm` command does not remove directories by default. This is a safety feature to prevent users from accidentally deleting directories with files in them.

To remove directories along with their files, use the `-r` flag.

```
$ rm -r a
```

To force the removal of files and directories without a confirmation, use the `-f` flag.

Warning 1.2.1 The `rm` command is a dangerous command. It does not move the files to the trash, but permanently deletes them. Be **extremely** careful when using the `rm` command. Only use the `-f` flag if you are absolutely sure that you want to delete the files.

To force `rm` to always ask for confirmation before deleting files, use the `-i` flag.

```
$ rm -i file3
rm: remove regular empty file 'file3'? y
```

cp:

The `cp` command is used to copy files. It takes the source file and the destination file as arguments.

```
$ touch file1
$ ls -F
file1
$ cp file1 file2
$ ls -F
file1  file2
```

The `cp` command can also take the `-r` flag to copy directories.

```
$ mkdir a
$ touch a/file1
$ cp -r a b
$ ls -R
.:
a/  b/

./a:
file1

./b:
file1
```

Exercise 1.2.4 Why did we use the `-R` flag in the above example? What does it do?

There are three ways to copy files using `cp`:

```
SYNOPSIS
       cp [OPTION]... [-T] SOURCE DEST
       cp [OPTION]... SOURCE... DIRECTORY
```

```
4       cp [OPTION]... -t DIRECTORY SOURCE...
```

Exercise 1.2.5 There are three ways of running the `cp` command to copy a file. Here we have demonstrated only one. Read the manual page of the `cp` command to find out the other two ways and try them out yourself.

mv:

The `mv` command is used to move files. The syntax is similar to the `cp` command. [20] It is used to move files from one location to another, or to rename files.

20: This means that `mv` also has three ways of running it.

```
1  $ touch file1
2  $ ls -F
3  file1
4  $ mv file1 file2
5  $ ls -F
6  file2
```

Exercise 1.2.6 Create a directory `dir1` using the `mkdir` command, then create a file `file1` inside `dir1`. Now move (rename) the `dir1` directory to `dir2` using the `mv` command. The newly created directory should be named `dir2` and should contain the `file1` file. Were you require to use the `-r` flag with the `mv` command like you would have in `cp` command?

1.2.5 Text Processing and Pagers

echo:

The `echo` command is used to print a message to the terminal. It can take multiple arguments and print them to the terminal.

```
1  $ echo Hello, World!
2  Hello, World!
```

The `echo` command can also take the `-e` flag to interpret backslash escapes.

```
1  $ echo -e "Hello, \nWorld!"
2  Hello,
3  World!
```

Some escape characters in **echo** are listed in Table Table 1.5 on the following page.

Exercise 1.2.7 Run the command `echo -e "\x41=\0101"` and try to understand the output and the escape characters used.

cat:

The `cat` command is used to concatenate and display the contents of files.

Table 1.5: Escape Characters in `echo`

Escape	Description
`\\`	backslash
`\a`	alert (BEL)
`\b`	backspace
`\c`	produce no further output
`\e`	escape
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab
`\0NNN`	byte with octal value NNN (1 to 3 digits)
`\xHH`	byte with hexadecimal value HH (1 to 2 digits)

```
$ cat file1
Hello, World! from file1
```

The `cat` command can take multiple files as arguments and display their contents one after another.

```
$ cat file1 file2
Hello, World! from file1
Hello, World! from file2
```

less:

Sometimes the contents of a file are too large to be displayed at once. Nowadays modern terminal emulators can scroll up and down to view the contents of the file, but actual **tty**s cannot do that. To view the contents of a file one page at a time, use the `less` command. **less** is a pager program that displays the contents of a file one page at a time. [21]

```
$ less file1
```

To scroll up and down, use the arrow keys, or the `j` and `k` keys. [22] Press `q` to exit the `less` command.

head:

The `head` command is used to display the first few lines of a file. By default, it displays the first 10 lines of a file.

```
$ head /etc/passwd
root:x:0:0:root:/root:/usr/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/usr/bin/nologin
dbus:x:81:81:System Message Bus:/:/usr/bin/nologin
systemd-coredump:x:984:984:systemd Core Dumper:/:/usr/bin/
  nologin
systemd-network:x:982:982:systemd Network Management:/:/usr/bin/
  nologin
```

The `head` command can take the `-n` flag to display the first *n* lines of a file. [23]

21: **more** is another pager program that displays the contents of a file one page at a time. It is older than **less** and has fewer features. **less** is an improved version of **more** and is more commonly used. Due to this, it is colloquially said that *"**less** is more"*, as it has more features.

22: Using `j` and `k` to move the cursor up and down is a common keybinding in many terminal applications. This originates from the *vi* text editor which will be covered later in the course.

23: We can also directly run `head -5 /etc/passwd` to display the first 5 lines of the file.

```
$ head -n 5 /etc/passwd
root:x:0:0:root:/root:/usr/bin/bash
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
```

Remark 1.2.3 Here we are listing the file `/etc/passwd` which contains information about the users on the system. The file will usually be present on all Unix-like systems and have a lot of system users. [24]

24: A system user is a user that is used by the system to run services and daemons. It does not belong to any human and usually is not logged into. System users have a user ID less than 1000.

tail:

The `tail` command is used to display the last few lines of a file. By default, it displays the last 10 lines of a file.

```
$ tail /etc/passwd
rtkit:x:133:133:RealtimeKit:/proc:/usr/bin/nologin
sddm:x:964:964:SDDM Greeter Account:/var/lib/sddm:/usr/bin/
  nologin
usbmux:x:140:140:usbmux user:/:/usr/bin/nologin
sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
qemu:x:962:962:QEMU user:/:/usr/bin/nologin
cups:x:209:209:cups helper user:/:/usr/bin/nologin
dhcpcd:x:959:959:dhcpcd privilege separation:/:/usr/bin/nologin
saned:x:957:957:SANE daemon user:/:/usr/bin/nologin
```

The `tail` command can take the `-n` flag to display the last n lines of a file.

```
$ tail -n 5 /etc/passwd
sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
qemu:x:962:962:QEMU user:/:/usr/bin/nologin
cups:x:209:209:cups helper user:/:/usr/bin/nologin
dhcpcd:x:959:959:dhcpcd privilege separation:/:/usr/bin/nologin
saned:x:957:957:SANE daemon user:/:/usr/bin/nologin
```

Exercise 1.2.8 Notice that the UID (3rd column) of the `sayan` user is 1000. The last column is `/bin/bash` instead of `/usr/bin/nologin` like others. This is because it is a regular user and not a system user.

wc:

The `wc` command is used to count the number of lines, words, and characters in a file. By default, it displays the number of lines, words, and characters in a file.

```
$ wc /etc/passwd
  43  103 2426 /etc/passwd
```

We can also use the `-l`, `-w`, and `-c` flags to display only the number of lines, words, and characters respectively.

```
$ wc -l /etc/passwd
43 /etc/passwd
```

Question 1.2.5 Can we print contents of multiple files using a single command?

Answer 1.2.5 `cat file1 file2 file3` will print the contents of `file1`, `file2`, and `file3` in the order given. The contents of the files will be printed one after the other.

Question 1.2.6 Can `cat` also be used to write to a file?

Answer 1.2.6 Yes, `cat > file1` will write to `file1`. The input will be taken from the terminal and written to `file1`. The input will be written to `file1` until the user presses `Ctrl+D` to indicate end of input. This is `redirection`, which we see in later weeks.

Question 1.2.7 How to list only first 10 lines of a file? How about first 5? Last 5? How about lines 105 to lines 152?

Answer 1.2.7 `head filename` will list the first 10 lines of `filename`.
`head -n 5 filename` will list the first 5 lines of `filename`.
`tail -n 5 filename` will list the last 5 lines of `filename`.
`head -n 152 filename | tail -n 48` will list lines 105 to 152 of `filename`. This uses `|` which is a pipe, which we will see in later weeks.

Question 1.2.8 Do you know how many lines a file contains? How can we count it? What about words? Characters?

Answer 1.2.8 `wc filename` will count the number of lines, words, and characters in `filename`.
`wc -l filename` will count the number of lines in `filename`.
`wc -w filename` will count the number of words in `filename`.
`wc -c filename` will count the number of characters in `filename`.

Question 1.2.9 How to delete an empty directory? What about non-empty directory?

Answer 1.2.9 `rmdir dirname` will delete an empty directory.
`rm -r dirname` will delete a non-empty directory.

Question 1.2.10 How to copy an entire folder to another name? What about moving?
Why the difference in flags?

Answer 1.2.10 `cp -r sourcefolder targetfolder` will copy an entire folder to another name.
`mv sourcefolder targetfolder` will move an entire folder to another name.
The difference in flags is because `cp` is used to copy, and `mv` is used to move or rename a file or folder. The `-r` flag is to copy recursively, and is not needed for `mv` as it is not recursive and simply changes the name of the folder (or the path).

1.2.6 Aliases and Types of Commands

alias:

The `alias` command is used to create an alias for a command. An alias is a custom name given to a command that can be used to run the command. [25]

25: Aliases are used to create shortcuts for long commands. They can also be used to create custom commands.

```
$ alias ll='ls -l'
$ ll
total 24
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Desktop
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Documents
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Downloads
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Music
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Pictures
drwxr-xr-x 2 username group 4096 Mar  1 12:00 Videos
```

The alias `ll` is created for the `ls -l` command.

Warning 1.2.2 Be careful when creating aliases. Do not create aliases for existing commands. This can lead to confusion and errors.

But this alias is temporary and will be lost when the shell is closed. To make the alias permanent, add it to the shell configuration file. For **bash**, this is the `.bashrc` file in the home directory.

```
$ echo "alias ll='ls -l'" >> ~/.bashrc
```

This will add the alias to the `.bashrc` file. To make the changes take effect, either close the terminal and open a new one, or run the `source` command.

```
$ source ~/.bashrc
```

Warning 1.2.3 Be careful when editing the shell configuration files. A small mistake can lead to the shell not working properly. Always keep a backup of the configuration files before editing them.

We can see the aliases that are currently set using the `alias` command.

```
$ alias
alias ll='ls -l'
```

We can also see what a particular alias expands to using the `alias` command with the alias name as an argument.

```
$ alias ll
alias ll='ls -l'
```

To remove an alias, use the `unalias` command.

```
$ unalias ll
```

Exercise 1.2.9 Create an alias `la` for the `ls -a` command. Make it permanent by adding it to the `.bashrc` file. Check if the alias is set using the `alias` command.

Exercise 1.2.10 Create an alias `rm` for the `rm -i` command. Make it permanent by adding it to the `.bashrc` file. Check if the alias is set using the `alias` command. Try to delete a file using the `rm` command. What happens?

which:

The `which` command is used to show the path of the command that will be executed.

```
$ which ls
/sbin/ls
```

We can also list all the paths where the command is present using the `-a` flag.

```
$ which -a ls
/sbin/ls
/bin/ls
/usr/bin/ls
/usr/sbin/ls
```

This means that if we delete the `/sbin/ls` file, the `/bin/ls` file will be executed when we run the `ls` command.

whatis:

The `whatis` command is used to show a short description of the command.

```
$ whatis ls
ls (1)               - list directory contents
ls (1p)              - list directory contents
```

Here the brackets show the section of the manual where the command is present. This short exerpt is taken from its man page itself.

whereis:

The `whereis` command is used to show the location of the command, source files, and man pages.

```
$ whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz /usr/share/man/man1p/
    ls.1p.gz
```

Here we can see that the `ls` command is present in `/usr/bin/ls`, and its man pages are present in `/usr/share/man/man1/ls.1.gz` and `/usr/share/man/man1p/ls.1p.gz`.

locate:

The `locate` command is used to find files by name. The file can be present anywhere in the system and if it is indexed by the `mlocate` database, it can be found using the `locate` command.

```
$ touch tmp/48/hellohowareyou
$ pwd
/home/sayan
$ locate hellohowareyou
/home/sayan/tmp/48/hellohowareyou
```

Note: you may have to run `updatedb` to update the database before using `locate`. This can only be run by the root user or using `sudo`.

type:

The `type` command is used to show how the shell will interpret a command. Usually some commands are both an executable and a shell built-in. The `type` command will show which one will be executed.

```
$ type ls
ls is hashed (/sbin/ls)
$ type cd
cd is a shell builtin
```

This shows that the `ls` command is an executable, and the `cd` command is a shell built-in.

We can also use the `-a` flag to show all the ways the command can be interpreted.

```
$ type -a pwd
pwd is a shell builtin
pwd is /sbin/pwd
pwd is /bin/pwd
pwd is /usr/bin/pwd
pwd is /usr/sbin/pwd
```

Here we can see that the `pwd` command is a shell built-in, and is also present in multiple locations in the system. But if we run the `pwd` command, the shell built-in will be executed.

`type` is also useful when you are not sure whether to use `man` or `help` for a command. Generally for a shell built-in, `help` is used, and for an executable the `info` and the `man` pages are used.

Types of Commands: A command can be an **alias**, a **shell built-in**, a **shell function**, **keyword**, or an **executable**.

The `type` command will show which type the command is.

- **alias**: A command that is an alias to another command defined by the user or the system.
- **builtin**: A shell built-in command is a command that is built into the shell itself. It is executed internally by the shell. This is usually faster than an external command.
- **file**: An executable file that is stored in the file system. It has to be stored somewhere in the **PATH** variable.
- **function**: A shell function is a set of commands that are executed when the function is called.

- **keyword**: A keyword is a reserved word that is part of the shell syntax. It is not a command, but a part of the shell syntax.

Exercise 1.2.11 Find the path of the `true` command using `which`. Find a short description of the `true` command using `whatis`. Is the executable you found actually the one that is executed when you run `true`? Check using `type true`

Question 1.2.11 How to create aliases? How to make them permanent? How to unset them?

Answer 1.2.11 `alias name='command'` will create an alias.
`unalias name` will unset the alias.
To make them permanent, add the alias to the `~/.bashrc` file.
The `~/.bashrc` file is a script that is executed whenever a new terminal is opened.

Question 1.2.12 How to run the normal version of a command if it is aliased?

Answer 1.2.12 `\command` will run the normal version of `command` if it is aliased.

Question 1.2.13 What is the difference between `which`, `whatis`, `whereis`, `locate`, and `type`?

Answer 1.2.13 Each of the commands serve a different purpose:

- `which` will show the path of the command that will be executed.
- `whatis` will show a short description of the command.
- `whereis` will show the location of the command, source files, and man pages.
- `locate` is used to find files by name.
- `type` will show how the command will be interpreted by the shell.

1.2.7 User Management

whoami:

The `whoami` command is used to print the username of the current user.

```
$ whoami
sayan
```

groups:

The `groups` command is used to display the groups that the current user belongs to.

```
$ groups
sys wheel rfkill autologin sayan
```

passwd

The `passwd` command is used to change the password of the current user. The root user can also change the password of other users. [26]

26: This executable is a special one, as it is a setuid program. This will be discussed in detail in Subsection 1.4.6.

who:

The `who` command is used to display the users who are currently logged in.

```
$ who
sayan    tty2         2024-05-22 13:49
sayan    pts/0        2024-05-22 15:58 (:0)
sayan    pts/1        2024-05-22 15:58 (tmux(1082933).%2)
sayan    pts/2        2024-05-22 15:58 (tmux(1082933).%1)
sayan    pts/3        2024-05-22 15:58 (tmux(1082933).%3)
sayan    pts/4        2024-05-22 15:58 (tmux(1082933).%4)
sayan    pts/5        2024-05-22 15:58 (tmux(1082933).%5)
sayan    pts/6        2024-05-22 15:58 (tmux(1082933).%6)
sayan    pts/7        2024-05-22 15:58 (tmux(1082933).%7)
sayan    pts/8        2024-05-22 15:58 (tmux(1082933).%8)
sayan    pts/9        2024-05-22 15:58 (tmux(1082933).%9)
sayan    pts/10       2024-05-22 17:58 (:0)
sayan    pts/11       2024-05-22 18:24 (tmux(1082933).%10)
sayan    pts/12       2024-05-22 18:24 (tmux(1082933).%11)
```

Exercise 1.2.12 Run the `who` command on the system commands VM. What is the output?

w:

The `w` command is used to display the users who are currently logged in and what they are doing.

```
$ w
 19:47:07 up  5:57,  1 user,  load average: 0.77, 0.80, 0.68
USER     TTY       LOGIN@   IDLE   JCPU   PCPU  WHAT
sayan    tty2      13:49    5:57m 19:10  21.82s dwm
```

This is different from the `who` command as it only considers the login shell. Here **dwm** is the window manager running on the `tty2` terminal.

1.2.8 Date and Time

date:

The `date` command is used to print formatted date and time information. Without any arguments, it prints the current date and time.

```
$ date
Mon May 20 06:23:07 PM IST 2024
```

We can specify the date and time to be printed using the `-d` flag.

```
$ date -d "2020-05-20 00:30:45"
Wed May 20 12:30:45 AM IST 2020
$ date -d "2019-02-29"
date: invalid date '2019-02-29'
$ date -d "2020-02-29"
Sat Feb 29 12:00:00 AM IST 2020
```

Exercise 1.2.13 Why did we get an error when trying to print the date `2019-02-29`?

We can also modify the format of the date and time using the `+` flag and different **format specifiers**. Some of the important format specifiers are listed in Table Table 1.6. Rest of the format specifiers can be found in the `date` manual page.

```
$ date +"%Y-%m-%d %H:%M:%S"
2024-05-20 18:23:07
$ date +"%A, %B %d, %Y"
Monday, May 20, 2024
```

Table 1.6: Date Format Specifiers

Specifier	Description
`%Y`	Year
`%m`	Month
`%d`	Day
`%H`	Hour
`%M`	Minute
`%S`	Second
`%A`	Full weekday name
`%B`	Full month name
`%a`	Abbreviated weekday name
`%b`	Abbreviated month name

We can even mention relative dates and times using the `date` command.

```
$ date -d "next year"
Tue May 19 06:23:07 PM IST 2025
$ date -d "next month"
Thu Jun 20 06:23:07 PM IST 2024
$ date -d "tomorrow"
Tue May 21 06:23:07 PM IST 2024
$ date -d "yesterday"
Sun May 19 06:23:07 PM IST 2024
```

cal:

The `cal` command is used to print a calendar. By default, it prints the calendar of the current month.

```
$ cal
   May 2024
Su Mo Tu We Th Fr Sa
           1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

We can specify the month and year to print the calendar of that month and year.

```
$ cal 2 2024
 February 2024
Su Mo Tu We Th Fr Sa
                 1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29
```

There are multiple flags that can be passed to the `cal` command to display different types of calendars and of multiple months or of entire year.

Remark 1.2.4 In Linux, there are sometimes multiple implementations of the same command. For example, there are two implementations of the `cal` command, one in the **bsdmainutils** package, which is the **BSD** implementation and also includes another binary named **ncal** for printing the calendar in vertical format. The other implementation is in the **util-linux** package, which does not contain a **ncal** binary. The flags and the output of the `cal` command can differ between the two implementations.

Question 1.2.14 How to print the current date and time in some custom format?

Answer 1.2.14 `date -d today +%Y-%m-%d` will print the current date in the format `YYYY-MM-DD`. The format can be changed by changing the format specifiers. The format specifiers are given in the `man date` page. The `-d today` can be dropped, but is mentioned to show that the date can be changed to any date. It can be strings like '2024-01-01' or '5 days ago' or 'yesterday', etc.

These are some of the basic commands that are used in the terminal. Each of these commands has many more options and flags that can be used to customize their behavior. It is left as an exercise to the reader to explore the manual pages of these commands and try out the different options and flags.

Many of the commands that we have discussed here are also explained in the form of short videos on Robert Elder's Youtube Channel.

1.3 Navigating the File System

1.3.1 What is a File System?

Unlike Windows which has different drive letters for different partitions, Linux follows a unified file structure. The filesystem hierarchy is a tree of directories and files[27]. The root[28] of the filesystem tree is the directory /. The basic filesystem hierarchy structure can be seen in Figure 1.6 and Table Table 1.7.

But what does so many directories mean? What do they do? What is the purpose of each directory?

27: A non-directory is a leaf node of a tree.

28: The root of a tree is the first node from which the tree originates. A tree can have only one root.

Table 1.7: Linux Filesystem Hierarchy

Directory Path	Description
/	Root directory
/bin	Essential command binaries
/boot	Static files of the bootloader
/dev	Device files
/etc	Host-specific system configuration
/home	User home directories
/lib	Essential shared libraries and kernel modules
/media	Mount point for removable media
/mnt	Mount point for mounting a filesystem temporarily
/opt	Add-on application software packages
/proc	Virtual filesystem providing process information
/root	Home directory for the root user
/run	Run-time variable data
/sbin	Essential system binaries
/srv	Data for services provided by the system
/sys	Kernel and system information
/tmp	Temporary files
/usr	Secondary hierarchy
/var	Variable data

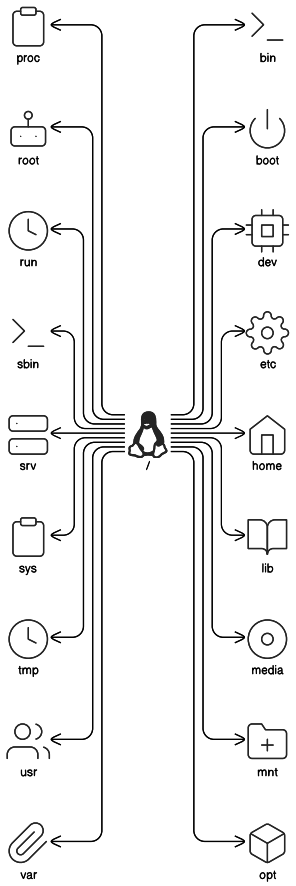

Figure 1.6: Linux Filesystem Hierarchy

Some directories do not store data on the disk, but are used to store information about the system. These directories are called *virtual* directories. For example, the **/proc** directory is a virtual directory that provides information about the running processes. The **/sys** directory is another virtual directory that provides information about the system. The **/tmp** is a *volatile* directory whose data is deleted as soon as the system is turned off. The **/run** directory is another volatile directory that stores runtime data.

Rest directories are stored on the disk. The reason for having so many directories is to categorize the type of files they store. For example, all the executable binaries of different applications and utilities installed in the system is stored in **/bin** and **/sbin** directories. All the shared libraries installed on the system are stored in **/lib** directory. Sometimes some applications are installed in **/opt** directory which are not installed directly by the package manager. [29]

We also need to store the user's documents and files. This is done in the **/home** directory. Each user has their own directory in the **/home** directory. The root user's directory is **/root**. All the application's configuration files are stored in the user's home directory in the **/home** directory itself. This

29: In Linux, you do not install applications by downloading them from the internet and running an installer like in Windows. You use a package manager to install applications. The package manager downloads the application from the internet and installs it on your system automatically. This way the package manager can also keep track of the installed applications and their dependencies and whether they should be updated. This is similar to the *Play Store* on mobile phones.

separation of application binary and per-user application settings helps people to easily change systems but keep their own **/home** directory constant and in turn, also all their application settings.

Some settings however needs to be applied system-wide and for all users. These settings are stored in the **/etc** directory. This directory contains all the system-wide configuration files.

To boot up the system, the bootloader needs some files. These files are stored in the **/boot** directory. [30] The bootloader is the first program that runs when the computer is turned on. It loads the operating system into memory and starts it.

30: Modern systems use **UEFI** instead of **BIOS** to boot up the system. The bootloader is stored in the **/boot/EFI** directory or in the **/efi** directory directly.

Although the file system is a unified tree hierarchy, this doesn't mean that we cannot have multiple partitions on Linux: au contraire, it is easier to manage partitions on Unix. We simply need to mention which empty directory in the hierarchy should be used to mount a partition. As soon as that partition is mounted, it gets populated with the data stored on that disk with all the files and subdirectories, and when the device is unmounted the directory again becomes empty. Although a partition can be mounted on any directory, there are some dedicated folders in `/` as well for this purpose. For example, the `/mnt` directory is used to mount a filesystem temporarily, and the `/media` directory is used to mount removable media like USB drives, however it is not required to strictly follow this.

Finally, the biggest revelation in Linux is that, everything is a file. Not only are all the system configurations stored as **plain text** files which can be read by humans, but the processes running on your system are also stored as files in proc. Your kernel's interfaces to the applications or users are also simple files stored in sys. Biggest of all, even your hardware devices are stored as files in dev. [31]

31: Device files are not stored as normal files on the disk, but are special files that the kernel uses to communicate with the hardware devices. These are either **block** or **character** devices. They are used to read and write data to and from the hardware devices.

The **/usr** directory is a secondary hierarchy that contains subdirectories similar to those present in **/**. This was created as the olden system had started running out of disk space for the **/bin** and **/lib** directories. Thus another directory named **usr** was made, and subdirectiores like **/usr/bin** and **/usr/lib** were made to store half of the binaries and libraries. There wasn't however any rigid definition of which binary should go where. Modern day systems have more than enough disk space to store everything on one partition, thus the **/bin** and **/lib** dont really exist any more. If they do, they are simply shortcuts [32] to the **/usr/bin** and **/usr/lib** directories which are still kept for *backwards compatibility*.

32: Shortcuts in Linux are called *symbolic links* or *symlinks*.

These can also be loosely classified into *sharable* and *non-sharable* directories and *static* and *variable* directories as shown in Table Table 1.8.

	Sharable	Non-sharable
Static	/usr, /opt	/etc, /boot
Variable	/var/spool	/tmp, /var/log

Table 1.8: Linux Filesystem Directory Classification

1.3.2 In Memory File System

Some file systems like **proc**, **sys**, **dev**, **run**, and **tmp** are not stored on the disk, but are stored in memory.

They have a special purpose and are used to store information about the system. These are called *virtual* directories.

These cannot be stored in a disk as it would be too slow to access them. Many of these files are very short lived yet are accessed very frequently. So these are stored in memory to speed up the access.

/dev and **/run** are mounted as **tmpfs** filesystems.

This can be seen by running the `mount` command or the `df` command.

```
  $ mount
/dev/sda1 on / type ext4 (rw,noatime)
devtmpfs on /dev type devtmpfs (rw,nosuid,size=4096k,nr_inodes
    =990693,mode=755,inode64)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,inode64)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,
    mode=620,ptmxmode=000)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,
    nodev,noexec,relatime)
cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,
    relatime,nsdelegate,memory_recursiveprot)
pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,noexec,
    relatime)
efivarfs on /sys/firmware/efi/efivars type efivarfs (rw,nosuid,
    nodev,noexec,relatime)
bpf on /sys/fs/bpf type bpf (rw,nosuid,nodev,noexec,relatime,mode
    =700)
configfs on /sys/kernel/config type configfs (rw,nosuid,nodev,
    noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,size=1590108k,nr_inodes
    =819200,mode=755,inode64)
systemd-1 on /proc/sys/fs/binfmt_misc type autofs (rw,relatime,fd
    =36,pgrp=1,timeout=0,minproto=5,maxproto=5,direct,pipe_ino
    =5327)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,nosuid,nodev,
    relatime,pagesize=2M)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime
    )
debugfs on /sys/kernel/debug type debugfs (rw,nosuid,nodev,noexec,
    relatime)
tracefs on /sys/kernel/tracing type tracefs (rw,nosuid,nodev,
    noexec,relatime)
fusectl on /sys/fs/fuse/connections type fusectl (rw,nosuid,nodev,
    noexec,relatime)
systemd-1 on /data type autofs (rw,relatime,fd=47,pgrp=1,timeout
    =60,minproto=5,maxproto=5,direct,pipe_ino=2930)
tmpfs on /tmp type tmpfs (rw,noatime,inode64)
/dev/sda4 on /efi type vfat (rw,relatime,fmask=0137,dmask=0027,
    codepage=437,iocharset=ascii,shortname=mixed,utf8,errors=
    remount-ro)
/dev/sda2 on /home type ext4 (rw,noatime)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,
    nosuid,nodev,noexec,relatime)
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size
    =795052k,nr_inodes=198763,mode=700,uid=1000,gid=1001,inode64)
portal on /run/user/1000/doc type fuse.portal (rw,nosuid,nodev,
    relatime,user_id=1000,group_id=1001)
```

```
/dev/sdb3 on /data type ext4 (rw,noatime,x-systemd.automount,x-
    systemd.idle-timeout=1min)
```

Here we can see that the **/dev** directory is mounted as a **devtmpfs** filesystem. The **/run** directory is mounted as a **tmpfs** filesystem. The **/proc** directory is mounted as a **proc** filesystem. The **/sys** directory is mounted as a **sysfs** filesystem.

These are all virtual filesystems that are stored in memory.

proc:

Proc is an old filesystem that is used to store information about the running processes. The **/proc** directory contains a directory for each running process. The directories are named as the process id of the process.

```
$ ls -l /proc | head
total 0
dr-xr-xr-x  9 root  root  0 May 23 13:01 1
dr-xr-xr-x  9 root  root  0 May 23 13:01 100
dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1004
dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1009
dr-xr-xr-x  9 root  root  0 May 23 13:01 102
dr-xr-xr-x  9 root  sayan 0 May 23 13:06 1029
dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1038
dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1039
dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1074
```

These folders are simply for information and do not store any data. This is why they have a size of 0. Each folder is owned by the user who started the process.

Inside each of these directories, there are files that contain information about the process.

You can enter the folder of a process that is started by you and see the information about the process.

```
$ cd /proc/301408
$ ls -l | head -n15
total 0
-r--r--r--  1 sayan sayan 0 May 23 16:55 arch_status
dr-xr-xr-x  2 sayan sayan 0 May 23 16:55 attr
-rw-r--r--  1 sayan sayan 0 May 23 16:55 autogroup
-r--------  1 sayan sayan 0 May 23 16:55 auxv
-r--r--r--  1 sayan sayan 0 May 23 16:55 cgroup
--w-------  1 sayan sayan 0 May 23 16:55 clear_refs
-r--r--r--  1 sayan sayan 0 May 23 16:55 cmdline
-rw-r--r--  1 sayan sayan 0 May 23 16:55 comm
-rw-r--r--  1 sayan sayan 0 May 23 16:55 coredump_filter
-r--r--r--  1 sayan sayan 0 May 23 16:55 cpu_resctrl_groups
-r--r--r--  1 sayan sayan 0 May 23 16:55 cpuset
lrwxrwxrwx  1 sayan sayan 0 May 23 16:55 cwd -> /home/sayan/docs/
    projects/sc-handbook
-r--------  1 sayan sayan 0 May 23 16:55 environ
lrwxrwxrwx  1 sayan sayan 0 May 23 13:41 exe -> /usr/bin/entr
```

Here you can see that the command line of the process is stored in the **cmdline** file. Here the process is of a command called **entr**.

You can also see the **current working directory** (cwd) of the process.

There are some other files in the **/proc** directory that contain information about the system.

- **cpuinfo** - stores cpu information.
- **version** - stores system information, content similar to `uname -a` command.
- **meminfo** - Diagnostic information about memory. Check `free` command.
- **partitions** - Disk partition information. Check `df`.
- **kcore** - The astronomical size (2^{47} bits) tells the maximum virtual memory (47 bits) the current Linux OS is going to handle.

sys:

Sys is a newer filesystem that is used to store information about the system. It is neatly organized and is easier to navigate than proc. This highly uses symlinks to organize the folders while maintaining redundancy as well. [33]

33: A symlink is a special type of file that points to another file or directory. It is similar to a shortcut in Windows. This will be discussed in detail in Subsection **??**.

Try running the following code snippet in a terminal if you have a caps lock key on your keyboard and are running linux directly on your bare-metal. [34]

```
$ cd /sys/class/leds
$ echo 1 | sudo tee *capslock/brightness
```

34: This will only work on your own system, not on the system commands VM, since you do not have the priviledge to modify the files there. Make sure you have the ability to run commands as root and are able to use `sudo`. It is also unlikely to work on a virtual machine. It will also not work on linux systems older than 2.6.

If you are running a linux system directly on your hardware, you will see the caps lock key light up. Most modern keyboards will quickly turn off the light again as the capslock is technically not turned on, only the led was turned on manually by you.

/sys vs /proc:

The `/proc` tree originated in System V Unix [35] , where it only gave information about each running process, using a `/proc/$PID/` format. Linux greatly extended that, adding all sorts of information about the running kernel's status. In addition to these read-only information files, Linux's `/proc` also has writable virtual files that can change the state of the running kernel. BSD [36] type OSes generally do not have `/proc` at all, so much of what you find under proc is non-portable.

35: Unix System V is one of the first commercial versions of the Unix operating system. It was originally developed by AT&T and first released in 1983.

36: BSD, or Berkeley Software Distribution, is a Unix-like operating system that was developed at the University of California, Berkeley. It was first released in 1977 and was based on the original Unix source code from AT&T. BSD is not linux, it is a totally different kernel, with similar core utils to GNU.

The intended solution for this mess in Linux's `/proc` is `/sys`. Ideally, all the non-process information that got crammed into the `/proc` tree should have moved to `/sys` by now, but historical inertia has kept a lot of stuff in `/proc`. Often there are two ways to effect a change in the running kernel: the old `/proc` way, kept for backwards compatibility, and the new `/sys` way that you're supposed to be using now.

1.3.3 Paths

Whenever we open a terminal on a Linux system, we are placed in a directory. This is called the *current working directory*. All shells and applications have a current working directory from where they are launched.

To refer to and identify the directory you are talking about, we use a **path**.

Definition 1.3.1 (Path) Path is a traversal in the filesystem tree. It is a way to refer to a file or directory.

Absolute Path:

The traversal to the directory from the root directory is called the **absolute path**. For example, if we want to refer to the directory named **alex** inside the directory **home** in the root of the file system, then it is qualified as:

```
/home/alex
```

Relative Path:

The traversal to the directory from the current working directory is called the **relative path**. For example, if we want to refer to the directory named **alex** inside the directory **home** from the **/usr/share** directory, then it will be qualified as:

```
../../home/alex
```

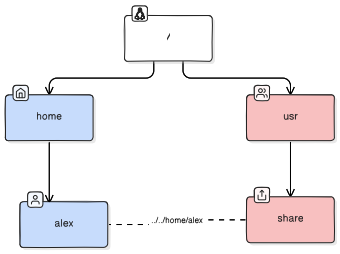

Figure 1.7: Relative Path

Remark 1.3.1 The **..** in the path refers to the parent directory. It is used in relative paths to refer to directories whose path requires travelling up the tree.

1.3.4 Basic Commands for Navigation

The file system can be navigated in the Linux command line using the following commands:

- **pwd**: Print the current working directory
- **ls**: List the contents of the current directory
- **cd**: Change the current working directory
- **mkdir**: Create a new directory
- **rmdir**: Remove a directory
- **touch**: Create a new file
- **rm**: Remove a file
- **pushd**: Push the current directory to a stack
- **popd**: Pop the current directory from a stack[37]

More details about these commands can be found in their respective man pages. For example, to find more about the **ls** command, you can type `man ls`.

Question 1.3.1 What is the command to list the contents of the current directory?

Answer 1.3.1 `ls`

37: **pushd** and **popd** are useful for quickly switching between directories in scripts.

Question 1.3.2 What is the command to list the contents of the current directory including hidden files?

Answer 1.3.2 `ls -a`

Question 1.3.3 What is the command to list the contents of the current directory in a long list format? (show permissions, owner, group, size, and time)

Answer 1.3.3 `ls -l`

Question 1.3.4 What is the command to list the contents of the current directory in a long list format and also show hidden files?

Answer 1.3.4 `ls -al` or `ls -la` or `ls -l -a` or `ls -a -l`

Question 1.3.5 The output of `ls` gives multiple files and directories in a single line. How can you make it print one file or directory per line?

Answer 1.3.5 `ls -1`
This can also be done by passing the output of `ls` to `cat` or storing the output of `ls` in a file and then using `cat` to print it. We will see these in later weeks.[38]

38: that is a one, not an L

1.4 File Permissions

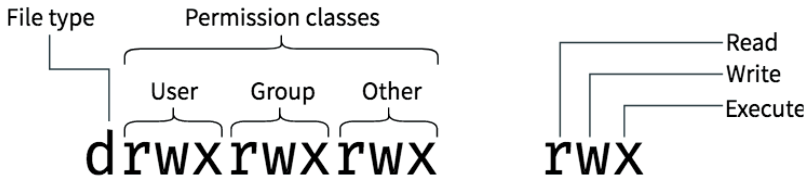

Figure 1.8: File Permissions

Definition 1.4.1 (File Permissions) File permissions define the access rights of a file or directory. There are three basic permissions: read, write, and execute. These permissions can be set for the owner of the file, the group of the file, and others.

We have already briefly seen how to see the permissions of a file using the `ls -l` command.

```
$ touch hello.txt
$ mkdir world
$ ls -l
total 4
-rw-r--r-- 1 sayan sayan    0 May 21 15:20 hello.txt
drwxr-xr-x 2 sayan sayan 4096 May 21 15:21 world
```

Here, the first column of the output of `ls -l` shows the permissions of the file or directory. As seen in Figure 1.8, the permissions are divided into four parts:

- The first character shows the type of the file. `-` for a regular file and `d` for a directory and more. [39]
- The next three characters show the permissions for the owner of the file.
- The next three characters show the permissions for the group of the file.
- The last three characters show the permissions for others.

39: There are other types of files as well, like `l` for a symbolic link, `c` for a character device, `b` for a block device, `s` for a socket, and `p` for a pipe. These will be discussed later.

Definition 1.4.2 (Owner) Although this can be changed, the owner of a file is usually the user who created it. All files in the filesystem have an owner. This is symbolically coded as **u**.

Definition 1.4.3 (Group) The group of a file is usually the group of the user who created it. But it can also be changed to any other existing group in the system. All users in the group [a] have the same permissions on the file. This is symbolically coded as **g**.

[a] except the owner of the file

Definition 1.4.4 (Others) Others are all the users who are not the owner of the file and are not in the group of the file. This is symbolically coded as **o**.

There are three actions that can be performed on a file: read, write, and execute.

- **Read**: The read permission allows the file to be read. This is symbolically coded as **r**.
- **Write**: The write permission allows the file to be modified. This is symbolically coded as **w**.
- **Execute**: The execute permission allows the file to be executed. [40] This is symbolically coded as **x**.

40: Executing a file means running the file as a program. For a directory, the execute permission allows the directory to be traversed into.

These however, have different meanings for files and directories.

1.4.1 Read

- For a file, the read permission allows the file to be read. You can use commands like `cat` or `less` to read the contents of the file if the user has **read** permissions.
- For a directory, the read permission allows the directory to be listed using `ls`.

1.4.2 Write

- For a file, the write permission allows the file to be modified. You can use commands like `echo` along with redirection [41] or a text editor like `vim` or `nano` to write to the file if the user has **write** permissions.
- For a directory, the write permission allows the directory to be modified. You can create, delete, or rename files in the directory if the user has **write** permissions.

41: Redirection is a way to send the output of a command to a file.

1.4.3 Execute

- For a file, the execute permission allows the file to be executed. This is usually only needed for special files like executables, scripts, or libraries. You can run the file as a program if the user has **execute** permissions.
- For a directory, the execute permission allows the directory to be traversed into. You can change to the directory if the user has **execute** permissions using `cd`. You can also only long-list the contents of the directory if the user has **execute** permissions on that directory.

1.4.4 Interesting Caveats

This causes some interesting edge-cases that one needs to be familiar with.

Cannot modify a file? Think again!

If you have **write** and **execute** permissions on a directory, even if you do not have **write** permission on a file inside the directory, you can **delete** the file due to your **write** permission on the directory, and then re-create

the modified version of the file with the same name. But if you try to simply modify the file directly, you will get permission error.

```
$ mkdir test
$ cd test
$ echo "hello world" > file1
$ chmod 400 file1     # 400 means read permission only
$ cat file1
hello world
$ echo "hello universe" > file1  # unable to write
-bash: file1: Permission denied
$ rm file1  # can remove as we have write permission on folder
rm: remove write-protected regular file 'file1'? y
$ echo "hello universe" > file1 # can create new file
$ cat file1
hello universe
```

However, this only works on files. You cannot remove a directory if you do not have **write** permission on the directory, even if you have **write** permission on its parent directory.

Can list names but not metadata?

If you have **read** permission on a directory but not **execute** permission, you cannot traverse into the directory, but you can still use `ls` to list the contents of the directory. However, you cannot use `ls -l` to long-list the contents of the directory. That is, you only have access to the name of the files inside, not their metadata.

```
$ mkdir test
$ touch test/1 test/2
$ chmod 600 test # removing execute permission from folder
$ ls test # we can still list the files due to read permission
1  2
$ ls -l test # but cannot long-list the files
ls: cannot access 'test/2': Permission denied
ls: cannot access 'test/1': Permission denied
total 0
-????????? ? ? ? ?            ? 1
-????????? ? ? ? ?            ? 2
```

Cannot list names but can traverse?

If you have **execute** permission on a directory but not **read** permission, you can traverse into the directory but you cannot list the contents of the directory.

```
$ mkdir test
$ touch test/1 test/2
$ chmod 300 test # removing read permission from folder
$ ls test # we cannot list the files
ls: cannot open directory 'test': Permission denied
$ cd test # but we can traverse into the folder
$ pwd
/home/sayan/test
```

Subdirectories with all permissions, still cannot access?

If you have all the permissions to a directory, but dont have **execute** permission on its parent directory, you cannot access the subdirectory, or even list its contents.

```
$ mkdir test
$ mkdir test/test2 # subdirectory
$ touch test/test2/1 # file inside subdirectory
$ chmod 700 test/test2 # all permissions to subdirectory
$ chmod 600 test # removing execute permission from parent
  directory
$ ls test
test2
$ cd test/test2 # cannot access subdirectory
-bash: cd: test/test2: Permission denied
$ ls test/test2 # cannot even list contents of subdirectory
ls: cannot access 'test/test2': Permission denied
```

1.4.5 Changing Permissions

The permissions of a file can be changed using the **chmod** command.

Synopsis:

```
chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... OCTAL-MODE FILE...
```

OCTAL-MODE is a 3 or 4 digit octal number where the first digit is for the owner, the second digit is for the group, and the third digit is for others. We will discuss how the octal representation of permissions is calculated in the next section.

The **MODE** can be in the form of **ugoa+-=rwxXst** where:

- **u** is the user who owns the file
- **g** is the group of the file
- **o** is others
- **a** is all
- **+** adds the permission
- **-** removes the permission
- **=** sets the permission
- **r** is read
- **w** is write
- **x** is execute
- **X** is execute only if its a directory or already has execute permission.
- **s** is setuid/setgid
- **t** is restricted deletion flag or sticky bit

We are already familiar with what **r**, **w**, and **x** permissions mean, but what are the other permissions?

1.4.6 Special Permissions

Definition 1.4.5 (SetUID/SetGID) The setuid and setgid bits are special permissions that can be set on executable files. When an executable file has the setuid bit set, the file will be executed with the privileges of the owner of the file. When an executable file has the setgid bit set, the file will be executed with the privileges of the group of the files.

SetUID:

This is useful for programs that need to access system resources that are only available to the owner or group of the file.

A very notable example is the `passwd` command. This command is used to set the password of an user. Although changing password of a user is a priviledged action that only the root user can do, the `passwd` command can be run by any user to change *their* password. This is possible due to the setuid bit set on the `passwd` command. When the `passwd` command is run, it is run with the privileges of the root user, and thus can change the password of that user.

You can check this out by running `ls -l /usr/bin/passwd` and seeing the `s` in the permissions.

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 80912 Apr  1 15:49 /usr/bin/passwd
```

SetGID:

The behaviour of **SetGID** is similar to **SetUID**, but the file is executed with the privileges of the group of the file.

However, **SetGID** can also be applied to a directory. When a directory has the **SetGID** bit set, all the files and directories created inside that directory will inherit the group of the directory, not the group of the user who created the file or directory. This is highly useful when you have a directory where multiple users need to work on the same files and directories, but you want to restrict the access to only a certain group of users. The primary group of each user is different from each other, but since they are also part of another group (which is the group owner of the directory) they are able to read and write the files present in the directory. However, if the user creates a file in the directory, the file will be owned by the user's primary group, not the group of the directory. So other users would not be able to access the file. This is fixed by the **SetGID** bit on the directory.

```
$ mkdir test
$ ls -ld test # initially the folder is owned by the user's
  primary group
drwxr-xr-x 2 sayan sayan 4096 May 22 16:27 test
$ chgrp wheel test # we change the group of the folder to wheel,
   which is a group that the user is part of
$ ls -ld test
drwxr-xr-x 2 sayan wheel 4096 May 22 16:27 test
$ whoami # this is the current user
sayan
$ groups # this is the users groups, first one is its primary
  group
sayan wheel
$ touch test/file1 # before setting the SetGID bit, a new file
  will have group owner as the primary group of the user
  creating it
$ ls -l test/file1 # notice the group owner is sayan
-rw-r--r-- 1 sayan sayan 0 May 22 16:29 test/file1
$ chmod g+s test # we set the SetGID bit on the directory
$ ls -ld test # now the folder has a s in the group permissions
drwxr-sr-x 2 sayan wheel 4096 May 22 16:29 test
```

```
$ touch test/file2 # now if we create another new file, it will
  have the group owner as the group of the directory
$ ls -l test/file2 # notice the group owner is wheel
-rw-r--r-- 1 sayan wheel 0 May 22 16:29 test/file2
```

Restricted Deletion Flag or Sticky Bit:

The restricted deletion flag or sticky bit is a special permission that can be set on directories.

Historically, this bit was to be applied on executable files to keep the program in memory after it has finished executing. This was done to speed up the execution of the program as the program would not have to be loaded into memory again. This was called **sticky bit** because the program would stick in memory. [42]

42: The part of memory where the program's text segment is stored is called the *swap*.

However, this is no longer how this bit is used.

When the sticky bit is set on a directory, only the owner of the file, the owner of the directory, or the root user can delete or rename files in the directory.

This is useful when you have a directory where multiple users need to write files, but you want to restrict the deletion of files to only the owner of the file or the owner of the directory.

The most common example of this is the `/tmp` directory. The `/tmp` directory is a directory where temporary files are stored. You want to let any user create files in the `/tmp` directory, but you do not want any user to delete files created by other users.

```
$ ls -ld /tmp
drwxrwxrwt 20 root root 600 May 22 16:43 /tmp
```

Exercise 1.4.1 Log into the system commands VM and cd into the `/tmp` directory. Create a file in the `/tmp` directory. Try to find if there are files created by other users in the `/tmp` directory using `ls -l` command. If there are files created by other users, try to delete them. [a]

[a] You can create a file normally, or using the **mktemp** command.

1.4.7 Octal Representation of Permissions

The permissions of a file for the file's owner, group, and others can be represented as a 3 or 4 digit octal number. [43] Each of the octal digits is the sum of the permissions for the owner, group, and others.

43: If the octal is 4 digits, the first digit is for special permissions like setuid, setgid, and sticky bit.

- **Read** permission is represented by 4
- **Write** permission is represented by 2
- **Execute** permission is represented by 1

Thus if a file has read, write, and execute permissions for the owner, read and execute permissions for the group, and only read permission for others, the octal representation of the permissions would be 751.

The octal format is usually used more than the symbolic format as it is easier to understand and remember and it is more concise.

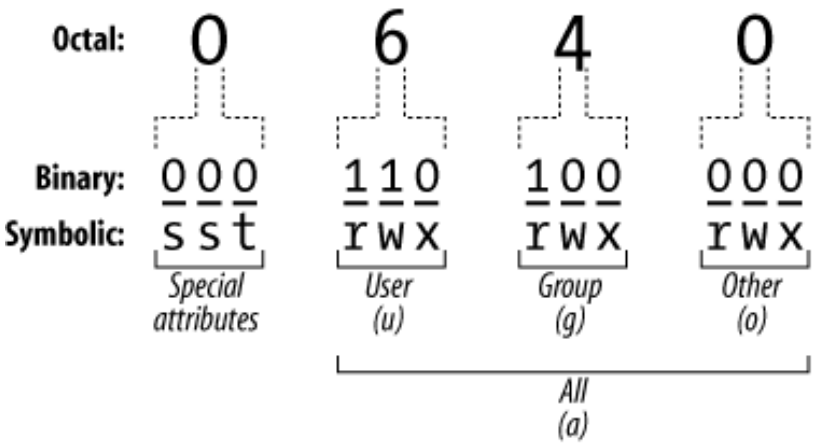

Figure 1.9: Octal Permissions

Table 1.9: Octal Representation of Permissions

Octal	Read	Write	Execute	Symbolic Representation	Description
0	0	0	0	—	No permissions
1	0	0	1	–x	Execute only
2	0	1	0	-w-	Write only
3	0	1	1	-wx	Write and execute
4	1	0	0	r–	Read only
5	1	0	1	r-x	Read and execute
6	1	1	0	rw-	Read and write
7	1	1	1	rwx	Read, write, and execute

```
$ chmod 754 myscript.sh # this sets the permissions of myscript.sh
     to rwxr-xr--
$ ./myscript.sh
Hello World!
```

However, if you want to add or remove a permission without changing the other permissions, the symbolic format is more useful.

```
$ chmod u+x myscript.sh # this adds execute permission to the
    owner of myscript.sh
$ ./myscript.sh
Hello World!
```

Question 1.4.1 How to list the permissions of a file?

Answer 1.4.1 `ls -l`
The permissions are the first 10 characters of the output.
`stat -c %A filename` will list only the permissions of a file.
There are other format specifiers of `stat` to show different statistics which can be found in `man stat`.

Question 1.4.2 How to change permissions of a file? Let's say we want to change `file1`'s permissions to `rwxr-xr-` What is the octal form of that?

Answer 1.4.2 `chmod u=rwx,g=rx,o=r file1` will change the permissions of `file1`
The octal form of `rwxr-xr-` is 754.
So we can also use `chmod 754 file1`
Providing the octal is same as using `=` to set the permissions.
We can also use `+` to add permissions and `-` to remove permissions.

1.5 Types of Files

We had briefly seen that the output of `ls -l` shows the type of the file as the first character of the permissions.

There are 7 types of files in a linux file system as shown in Table Table 1.10.

Table 1.10: Types of Files

Type	Symbol
Regular Files	-
Directories	d
Symbolic Links	l
Character Devices	c
Block Devices	b
Named Pipes	p
Sockets	s

1.5.1 Regular Files

Regular files are the most common type of file. Almost all files are regular files. Scripts and executable binaries are also regular files. All the configuration files of the system are regular files as well. The regular files are actually the only files that contain data and are stored on the disk.

1.5.2 Directories

Directories are files that contain a list of other files. Directories do not contain data, they contain references to other files. Usually the size of a directory is equal to the block size of the filesystem. Directories have some special permissions that are different from regular files as discussed in Section 1.4.

1.5.3 Symbolic Links

Symbolic links are files that point to other files. They only consume the space of the path they are pointing to. Symlinks [44] are useful to create shortcuts to files or directories. They are dependent on the original file and will stop working if the original file is deleted or moved. They are discussed in detail in Section 1.6.

44: Symlinks is short for symbolic links.

There are another type of links called **hard links**. However, hard links are not files, they are pointers to the same inode. They do not consume extra space, and are not dependent on the original file. Hard links do not have a separate type, they are just regular files.

1.5.4 Character Devices

Character devices are files that represent devices that are accessed as a stream of bytes. For example the keyboard, mouse, webcams, and most USB devices are character devices. These are not real files stored on the disk, but are files that represent devices. They can interacted with like a file using the `read` and `write` system calls to interact with the hardware directly. These files are made available by the kernel and are stored in the `/dev` directory. Any read/write operation on a character device is monitored by the kernel and the data is sent to the device.

```
$ cd /dev/input
$ ls -l
total 0
drwxr-xr-x 2 root root     220 May 22 13:49 by-id
drwxr-xr-x 2 root root     420 May 22 13:49 by-path
crw-rw---- 1 root input 13, 64 May 22 13:49 event0
crw-rw---- 1 root input 13, 65 May 22 13:49 event1
crw-rw---- 1 root input 13, 74 May 22 13:49 event10
crw-rw---- 1 root input 13, 75 May 22 13:49 event11
crw-rw---- 1 root input 13, 76 May 22 13:49 event12
```

```
crw-rw---- 1 root input 13, 77 May 22 13:49 event13
crw-rw---- 1 root input 13, 78 May 22 13:49 event14
crw-rw---- 1 root input 13, 79 May 22 13:49 event15
crw-rw---- 1 root input 13, 80 May 22 13:49 event16
crw-rw---- 1 root input 13, 81 May 22 13:49 event17
crw-rw---- 1 root input 13, 82 May 22 13:49 event18
crw-rw---- 1 root input 13, 83 May 22 13:49 event19
crw-rw---- 1 root input 13, 66 May 22 13:49 event2
crw-rw---- 1 root input 13, 84 May 22 13:49 event20
crw-rw---- 1 root input 13, 67 May 22 13:49 event3
crw-rw---- 1 root input 13, 68 May 22 13:49 event4
crw-rw---- 1 root input 13, 69 May 22 13:49 event5
crw-rw---- 1 root input 13, 70 May 22 13:49 event6
crw-rw---- 1 root input 13, 71 May 22 13:49 event7
crw-rw---- 1 root input 13, 72 May 22 13:49 event8
crw-rw---- 1 root input 13, 73 May 22 13:49 event9
crw-rw---- 1 root input 13, 63 May 22 13:49 mice
crw-rw---- 1 root input 13, 32 May 22 13:49 mouse0
crw-rw---- 1 root input 13, 33 May 22 13:49 mouse1
```

Here the `event` and `mouse` files are character devices that represent input devices like the keyboard and mouse. Note the **`c`** in the permissions, which indicates that these are character devices.

1.5.5 Block Devices

Block devices are files that represent devices that are accessed as a block of data. For example hard drives, SSDs, and USB drives are block devices. These files also do not store actual data on the disk, but represent devices. Any block file can be mounted as a filesystem. We can interact with block devices using the `read` and `write` system calls to interact with the hardware directly. For example, the `/dev/sda` file represents the first hard drive in the system.

This makes it easy to write an image to a disk directly using the `dd` command. [45]

45: The `dd` command is a powerful tool that can be used to copy and convert files. It is acronym of *data duplicator*. However, it is also known as the *disk destroyer* command, as it can be used to overwrite the entire disk if you are not careful with which disk you are writing the image to.

The following example shows how we can use the `dd` command to write an image [46] to a USB drive. It is this easy to create a bootable USB drive for linux.

46: ISO file

```
$ dd if=~/Downloads/archlinux.iso of=/dev/sdb bs=4M status=
  progress
```

Here `if` is the input file, `of` is the output file, `bs` is the block size, and `status` is to show the progress of the operation.

Warning 1.5.1 Be very careful when using the `dd` command. Make sure you are writing to the correct disk. Writing to the wrong disk can cause data loss.

1.5.6 Named Pipes

47: Also known as FIFOs

Named pipes [47] are files that are used for inter-process communication. They do not store the data that you write to them, but instead pass the

data to another process. A process can only write data to a named pipe if another process is reading from the named pipe. [48]

48: and vice versa

```
$ mkfifo pipe1
$ ls -l pipe1
prw-r--r-- -1 sayan sayan 0 May 22 18:22 pipe1
```

Here the `p` in the permissions indicates that this is a named pipe. If you now try to write to the named pipe, the command will hang until another process reads from the named pipe. Try the following in two different terminals:

Terminal 1:

```
$ echo "hello" > pipe1
```

Terminal 2:

```
$ cat pipe1
```

You will notice that whichever command you run first will hang until the other command is run.

1.5.7 Sockets

Sockets are a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.

This is similar to named pipes, but the difference is that named pipes are meant for IPC between processes in the same machine, whereas sockets can be used for communication across machines.

Try out the following in two different terminals:

Terminal 1:

```
$ nc -lU socket.sock
```

Terminal 2:

```
$ echo "hello" | nc -U socket.sock
```

Notice here, that if you run the command in terminal 2 first, it will error out with the text:

```
nc: socket.sock: Connection refused
```

Only if we run them in correct order can you see the message "hello" being printed in terminal 1. [49]

49: The `nc` command is the netcat command. It is a powerful tool for network debugging and exploration. It can be used to create sockets, listen on ports, and send and receive data over the network. This will be discussed in more detail in the networking section and in the **Modern Application Development** course.

You can press `Ctrl+C` to stop the `nc` command in both terminals.

1.5.8 Types of Regular Files

Regular files can be further classified into different types based on the data they contain. In Linux systems, the type of a file is determined by its **MIME** type. The extension of a file does not determine its type, the contents of the file do. It is thus common to have files without extensions in Linux systems, as they provide no value.

The `file` command can be used to determine the type of a file.

The bytes at the start of a file used to identify the type of file are called the **magic bytes**.
More details can be found at: https://en.wikipedia.org/wiki/List_of_file_signatures

```
$ file /etc/passwd
/etc/passwd: ASCII text
$ file /bin/bash
/bin/bash: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV)
    , dynamically linked, interpreter /lib64/ld-linux-x86-64.so
    .2, BuildID[sha1]=165d3a5ffe12a4f1a9b71c84f48d94d5e714d3db,
    for GNU/Linux 4.4.0, stripped
```

Question 1.5.1 What types of files are possible in a linux file system?

Answer 1.5.1 There are 7 types of files in a linux file system:

- Regular Files (starts with `-`)
- Directories (starts with `d`)
- Symbolic Links (starts with `l`)
- Character Devices (starts with `c`)
- Block Devices (starts with `b`)
- Named Pipes (starts with `p`)
- Sockets (starts with `s`)

Question 1.5.2 How to know what kind of file a file is? Can we determine using its extension? Can we determine using its contents? What does *MIME* mean? How to get that?

Answer 1.5.2 The `file` command can be used to determine the type of a file.
The extension of a file does not determine its type.
The contents of a file can be used to determine its type.
MIME stands for Multipurpose Internet Mail Extensions.
It is a standard that indicates the nature and format of a document.
`file -i filename` will give the MIME type of `filename`.

1.6 Inodes and Links

1.6.1 Inodes

Definition 1.6.1 (Inodes) An inode is an index node. It serves as a unique identifier for a specific piece of metadata on a given filesystem.

Whenever you run `ls -l` and see all the details of a file, you are seeing the metadata of the file. These metadata, however, are not stored in the file itself. These data about the files are stored in a special data structure called an **inode**.

Each inode is stored in a common table and each filesystem mounted to your computer has its own inodes. An inode number may be used more than once but never by the same filesystem. The filesystem id combines with the inode number to create a unique identification label.

You can check how many inodes are used in a filesystem using the `df -i` command.

```
$ df -i
Filesystem       Inodes   IUsed    IFree IUse% Mounted on
/dev/sda1       6397952  909213  5488739   15% /
/dev/sda4             0       0        0     - /efi
/dev/sda2      21569536 2129841 19439695   10% /home
/dev/sdb3      32776192    2380 32773812    1% /data
$ df
Filesystem     1K-blocks      Used Available Use% Mounted on
/dev/sda1      100063312  63760072  31174076  68% /
/dev/sda4        1021952    235760    786192  24% /efi
/dev/sda2      338553420 273477568  47805068  86% /home
/dev/sdb3      514944248 444194244  44518760  91% /data
```

You can notice the number of inodes present, number of inodes used, and number of inodes that are free. The **IUse%** column shows the percentage of inodes used. This however, does not mean how much of space is used, but how many files can be created.

Observe that although the `/data` partition has only 1% of inodes used, it has 91% of space used. This is because the files in the `/data` partition are large files, and thus the number of inodes used is less. Remember that a file will take up one inode, no matter how large it is. But the space it takes up will be the size of the file.

We can also see the inode number of a file using the `ls -i` command.

```
$ ls -i
  1234567 file1
  1234568 file2
  1234569 file3
```

Here the first column is the **inode** number of the file.

Remark 1.6.1 The inode number is unique only within the filesystem. If you copy a file from one filesystem to another, the inode number will change.

1.6.2 Separation of Data, Metadata, and Filename

In UNIX systems, the data of a file and the metadata of a file are stored separately. The inodes are stored in a inode-array or table, and contain the metadata of the file and the pointer to it in the storage block. These metadata can be retrieved using the `stat` system call. [50]

Conveniently, a userland utility to list the metadata of a file is also called `stat`.

```
$ stat /etc/profile
  File: /etc/profile
  Size: 993         Blocks: 8          IO Block: 4096   regular
    file
Device: 8,1 Inode: 2622512     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/
     root)
Access: 2024-05-21 18:30:27.000000000 +0530
Modify: 2024-04-07 23:32:30.000000000 +0530
Change: 2024-05-21 18:30:27.047718323 +0530
 Birth: 2024-05-21 18:30:27.047718323 +0530
```

We can also specify the format of the output of the `stat` command using the `-format` or `-c` flag to print only the metadata we want.

The data of the file is stored in the storage block. The inode number indexes a table of inodes on the file system. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file, thereby allowing access to the file.

On many older file systems, inodes are stored in one or more fixed-size areas that are set up at file system creation time, so the maximum number of inodes is fixed at file system creation, limiting the maximum number of files the file system can hold.

Some Unix-style file systems such as JFS, XFS, ZFS, OpenZFS, ReiserFS, btrfs, and APFS omit a fixed-size inode table, but must store equivalent data in order to provide equivalent capabilities. Common alternatives to the fixed-size table include B-trees and the derived B+ trees.

Remark 1.6.2 Although the inodes store the metadata of the file, the filename is not stored in the inode. It is stored in the directory entry. Thus the filename, file metdata, and file data are stored separately.

50: A system call is a request in a operating system made via a software interrupt by an active process for a service performed by the kernel. The diagram in Figure 1.10 shows how system calls work.

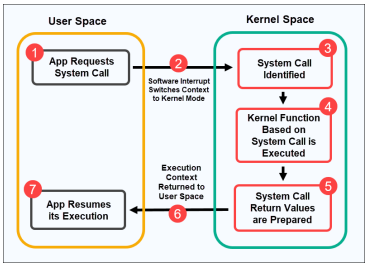

Figure 1.10: System Calls

Table 1.11: Metadata of a File

Metadata	Description
Size	Size of the file in bytes
Blocks	Number of blocks used by the file
IO Block	Block size of the file system
Device	Device ID of the file system
Inode	Inode number of the file
Links	Number of hard links to the file
Access	Access time of the file (`atime`)
Modify	Modification time of the file (`mtime`)
Change	Change time of the inode (`ctime`)
Birth	Creation time of the file

1.6.3 Directory Entries

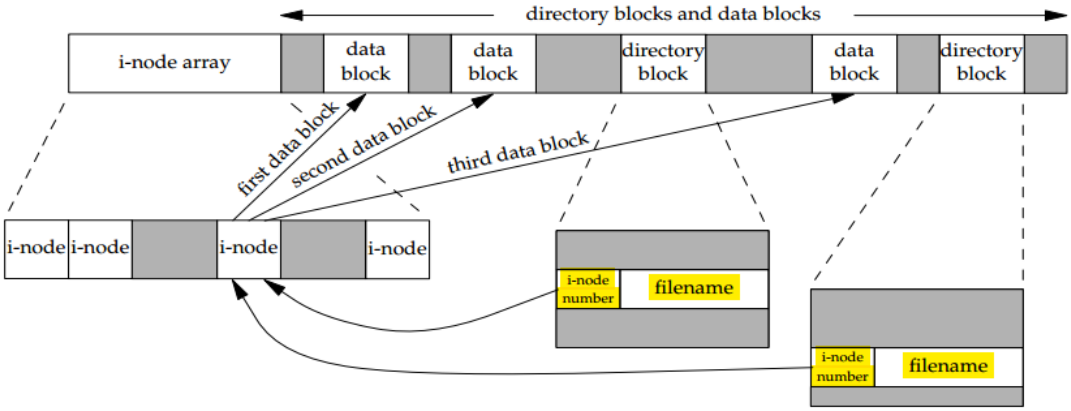

Figure 1.11: Inodes and Directory Entry

Unix directories are lists of association structures, each of which contains one filename and one inode number. The file system driver must search a directory for a particular filename and then convert the filename to the correct corresponding inode number.

Thus to read a file from a directory, first the directory's directory entry is read which stores the name of the file and its inode number. The kernel then follows the inode number to find the inode of the file. The inode stores all the metadata of the file, and the location of the data of the file. The kernel then follows the inode to find the data of the file. This is shown in Figure 1.11.

So what happens if two directory entries point to the same inode? This is called a **hard link**.

1.6.4 Hard Links

If multiple entries of the directory entry points to the same inode, they are called hard links. Hard links can have different names, but they are the same file. As they point to the same inode, they also have the same metadata.

This is useful if you want to have the same file in multiple directories without taking up more space. It is also useful if you want to keep a backup of a important file which is accessed by many people. If someone accidentally deletes the file, the other hard links will still be there and able to access the file.

Definition 1.6.2 (Hard Links) Hard Links are just pointers to the same inode. They are the same file. They are not pointers to the path of the file. They are pointers to the file itself. They are not affected by the deletion of the other file. When creating a hard link, you need to provide the path of the original file, and thus it has to be either

absolute path, or relative from the current working directory, not relative from the location of the hard link.

Hard links can be created for files only, and not directories. It can be created using the `ln` command.

```
$ ln file1 file2
```

This will create a hard link named `file2` that points to the same inode as `file1`.

Remark 1.6.3 Hard links are not dependent on the original file. They are the same file and equivalent. The first link to be created has no special status.

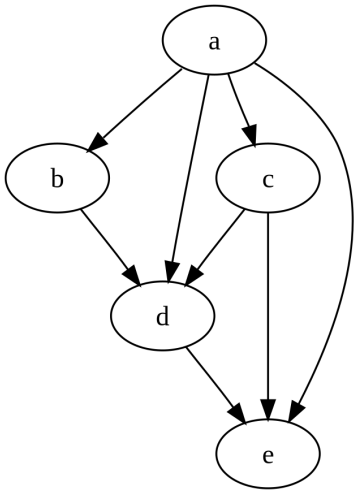

Figure 1.12: Directed Acyclic Graph

Historically directories could also have hard links, but this would cause the file tree to stop being a Directed Acyclic Graph [51] and become a Directed Cyclic Graph if a hardlink of an ancestor was put as a subdirectory. This would create confusions and infinite walks in the file system. Modern systems generally prohibit this confusing state, except that the parent of root is still defined as root. [52]

As hard links depend on the inode, they can only exist in the same filesystem as inodes are unique to a filesystem only.

If we want to create shortcuts across filesystems, or if we want to create a link to a directory, we can use **symbolic links**.

1.6.5 Symbolic Links

A symbolic link contains a text string that is automatically interpreted and followed by the operating system as a path to another file or directory. This other file or directory is called the "target". The symbolic link is a second file that exists independently of its target. If a symbolic link is deleted, its target remains unaffected. If a symbolic link points to a target, and sometime later that target is moved, renamed or deleted, the symbolic link is not automatically updated or deleted, but continues to exist and still points to the old target, now a non-existing location or file. Symbolic links pointing to moved or non-existing targets are sometimes called broken, orphaned, dead, or dangling.

Definition 1.6.3 (Soft Links) Soft Links are special kinds of files that just store the path given to them. Thus the path given while making soft links should either be an absolute path, or relative **from** the location of the soft link **to** the location of the original file. It should not be relative from current working directory.[a]

[a] This is a common mistake.

Symlinks are created using the `symlink` system call. This can be done using the `ln -s` command.

```
$ echo "hello" > file1
$ ln -s file1 file2
$ ls -l
```

51: A Directed Acyclic Graph is a graph that has no cycles as seen in Figure 1.12.

52: The most notable exception to this prohibition is found in Mac OS X (versions 10.5 and higher) which allows hard links of directories to be created by the superuser.

```
total 4
-rw-r--r-- 1 sayan sayan 6 May 23 15:27 file1
lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2 -> file1
$ cat file2
hello
```

Interesting Observation:

Usually we have seen that if we use `ls -l` with a directory as its argument, it lists the contents of the directory.

The only way to list the directory itself is to use `ls -ld`.

But if a symlink is made to a directory, then `ls -l` on that symlink will list only the symlink.

To list the contents of the symlinked directory we have to append a `/` to the symlink.

```
$ ln -s /etc /tmp/etc
$ ls -l /tmp/etc
lrwxrwxrwx 1 sayan sayan 4 May 23 15:30 /tmp/etc -> /etc
$ ls -l /tmp/etc/ | head -n5
total 1956
-rw-r--r--  1 root root      44 Mar 18 21:50 adjtime
drwxr-xr-x  3 root root    4096 Nov 17  2023 alsa
-rw-r--r--  1 root root     541 Apr  8 20:53 anacrontab
drwxr-xr-x  4 root root    4096 May 19 00:44 apparmor.d
```

Here I used `head` to limit the number of lines shown as the directory is large. [53]

53: This way of combining commands will be discussed later.

The symlink file stores only the path provided to it while creating it. This was historically stored in the data block which was pointed to by the inode. But this made it slower to access the symlink.

Modern systems store the symlink value in the inode itself if its not too large. Inodes usually have a limited space allocated for each of them, so a symlink with a small target path is stored directly in the inode. This is called a **fast symlink**.

However if the target path is too large, it is stored in the data block pointed to by the inode. This is retroactively called a **slow symlink**.

This act of storing the target path in the inode is called **inlining**.

Symlinks do not have a permission set, thus they always report `lrwxrwxrwx` as their permissions.

The size reported of a symlink file is independent of the actual file's size.

```
$ echo "hello" > file1
$ ln -s file1 file2
$ ls -l
-rw-r--r-- 1 sayan sayan 6 May 23 15:27 file1
lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2 -> file1
$ echo "a very big file" > file2
$ ls -l
-rw-r--r-- 1 sayan sayan 16 May 23 15:40 file1
lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2 -> file1
```

Rather, the size of a symlink is the length of the target path.

```
$ ln -s /a/very/long/and/non-existant/path link1
$ ln -s small link2
$ ls -l
total 0
lrwxrwxrwx 1 sayan sayan 34 May 23 15:41 link1 -> /a/very/long/and
    /non-existant/path
lrwxrwxrwx 1 sayan sayan  5 May 23 15:41 link2 -> small
```

Notice that the size of `link1` is 34, the length of the target path, and the size of `link2` is 5, the length of the target path.

1.6.6 Symlink vs Hard Links

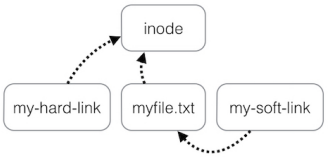

Figure 1.13: Abstract Representation of Symbolic Links and Hard Links

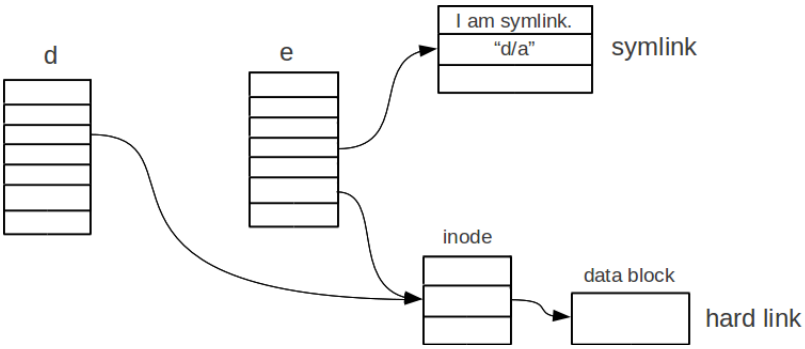

Figure 1.14: Symbolic Links and Hard Links

The difference between a symlink and a hard link is that a symlink is a pointer to the original file, while a hard link is the same file. Other differences are listed in Table 1.12.

Table 1.12: Symlink vs Hard Link

Property	Symlink	Hard Link
File Type	Special File	Regular File
Size	Length of the target path	Size of the file
Permissions	lrwxrwxrwx	Same as the original file
Inode	Different	Same
Dependency	Dependent on the original file	Independent of the original file
Creation	Can be created across filesystems	Can only be created in the same filesystem
Target	Can point to directories	Can only point to files

1.6.7 Identifying Links

Soft Links:

To identify if a file is a symlink or a hard link, you can use the `ls -l` command. If the file is a symlink, the first character of the permissions will be `l`. `ls -l` will also show the target of the symlink after a `->` symbol. However, you cannot acertain if a file has a sofy link pointing to it somewhere else or not.

Hard Links:

To identify if a file is a hard link, you can use the `ls -i` command. Hard links will have the same inode number as each other. The inode number is the first column of the output of `ls -i`.

Also the number of links to the file will be more than 1. The number of links is the second [54] column of the output of `ls -l`.

54: third if using `ls -li`

Even if a hard link is not present in current directory, you can acertain that a file has a hard link pointing to it somewhere else using the **number of hardlinks** column of `ls -l`.

```
$ touch file1
$ ln -s file1 file2
$ ln file1 file3
$ ls -li
total 0
4850335 -rw-r--r-- 2 sayan sayan 0 May 23 15:56 file1
4851092 lrwxrwxrwx 1 sayan sayan 5 May 23 15:56 file2 -> file1
4850335 -rw-r--r-- 2 sayan sayan 0 May 23 15:56 file3
```

1.6.8 What are . and ..?

`.` and `..` are special directory entries. They are hard links to the current directory and the parent directory respectively. Each directory has a `.` entry pointing to itself and a `..` entry pointing to its parent directory.

Due to this, the number of hard links to a directory is exactly equal to the number of subdirectories it has plus 2.

This is because each subdirectory has a `..` entry pointing to the parent directory, and the parent directory has a `.` entry pointing to itself.

So the directory's name in its parent directory is 1 link, the `.` in the directory is 1 link, and all the subdirectories have a `..` entry pointing to the directory, which is 1 link each.

$$\text{Number of links to a directory} = \text{Number of subdirectories} + 2$$

This formula always stands because a user cannot create additional hard links to a directory.

Question 1.6.1 How to list the inodes of a file?

Answer 1.6.1 `ls -i` will list the inodes of a file. The inodes are the first column of the output of `ls -i` This can be combined with other flags like `-l` or `-a` to show more details.

Question 1.6.2 How to create soft link of a file?

Answer 1.6.2 `ln -s sourcefile targetfile` will create a soft link of `sourcefile` named `targetfile`. The soft link is a pointer to the original file.

Question 1.6.3 How to create hard link of a file?

Answer 1.6.3 `ln sourcefile targetfile` will create a hard link of `sourcefile` named `targetfile`. The hard link is same as the original file. It does not depend on the original file anymore after creation. They are equals, both are `hardlinks` of each other. There is no parent-child relationship. The other file can be deleted and the original file will still work.

Question 1.6.4 How to get the real path of a file?
Assume three files:

- **file1** is a soft link to **file2**
- **file2** is a soft link to **file3**
- **file3** is a regular file

Real path of all these three should be the same. How to get that?

Answer 1.6.4 `realpath filename` will give the real path of `filename`. You can also use `readlink -f filename` to get the real path.

Command Line Editors 2

2.1 Introduction 55
2.1.1 Types of Editors 55
2.1.2 Why Command Line Editors? 55
2.1.3 Mouse Support 55
2.1.4 Editor war 56
2.1.5 Differences between Vim and Emacs 56
2.1.6 Nano: The peacemaker amidst the editor war . . 58
2.2 Vim 58
2.2.1 History 58
2.2.2 Ed Commands 65
2.2.3 Exploring Vim 72
2.3 Emacs 80
2.3.1 History 80
2.3.2 Exploring Emacs 82
2.4 Nano 84
2.4.1 History 84
2.4.2 Exploring Nano 85
2.4.3 Editing A Script in Nano 85

2.1 Introduction

Now that we know how to go about navigating the linux based operating systems, we might want to view and edit files. This is where command line editors come in.

Definition 2.1.1 A **command line editor** is a type of text editor that operates entirely from the command line interface. They usually do not require a graphical user interface or a mouse. [a]

[a] This means that CLI editors are the only way to edit files when you are connected to a remote server. Since remote servers do not have any graphical server like X11, you cannot use graphical editors like gedit or kate.

2.1.1 Types of Editors

- **Graphical Editors:** These are editors that require a graphical user interface. Examples include *gedit* * , *kate* † , *vs code*, etc.
- **Command Line Editors:** These are editors that operate entirely from the command line interface.

We will be only discussing command line editors in this chapter.

2.1.2 Why Command Line Editors?

Command line editors are very powerful and efficient. They let you edit files without having to leave the terminal. This is usually faster than opening a graphical editor. In many cases like sshing [1] into a remote server, command line editors are the only way to edit files. Another reason for the popularity of command line editors is that they are very lightweight.

1: SSH stands for Secure Shell. It is a cryptographic network protocol for operating network services securely over an unsecured network. This will be discussed in detail in a later chapter.

2.1.3 Mouse Support

Most command line editors do not have mouse support, and others do not encourage it. But wont it be difficult to navigate without a mouse? Not really. Once you get used to the keyboard shortcuts, you will find that you can navigate way faster than with a mouse.

Mouse editors usually require the user to click on certain buttons, or to follow multi-click procedures in nested menus to perform certain tasks.

* Gedit is the default text editor in GNOME desktop environment.
† Kate is the default text editor in KDE desktop environment.

Whereas in keyboard based editors, all the actions that can be performed are mapped to some keyboard shortcuts.

Modern CLI editors usually also allow the user to totally customize the keyboard shortcuts.

This being said, most modern CLI editors do have mouse support as well if the user is running them in a terminal emulator that supports mouse over a X11 or Wayland display server. [2]

2: X11 and Wayland are display servers that are used to render graphical applications. Although not directly covered, these can be explored in details on the internet.

2.1.4 Editor war

Although there are many command line editors available, the most popular ones are *vim* and *emacs*.

Definition 2.1.2 The editor war is the rivalry between users of the Emacs and vi (now usually Vim, or more recently Neovim) text editors. The rivalry has become an enduring part of hacker culture and the free software community. [3]

3: More on this including the history and the humor can be found on the internet.

Vim

Vim is a modal editor, meaning that it has different modes for different tasks. Most editors are modeless, this makes vim a bit difficult to learn. However, once familiar with it, it is very powerful and efficient. Vim heavily relies on alphanumeric keys for navigation and editing. Vim keybindings are so popular that many other editors and even some browsers [4] have vim-like keybindings.

4: *qutebrowser* is a browser that uses vim-like keybindings. Firefox and Chromium based browsers also have extensions that provide vim-like keybindings. These allow the user to navigate the browser using vim-like keybindings and without ever touching the mouse.

Emacs

Emacs is a modeless editor, meaning that it does not have different modes for different tasks. Emacs is also very powerful and efficient. It uses multi-key combinations for navigation and editing.

2.1.5 Differences between Vim and Emacs

Keystroke execution

Emacs commands are key combinations for which modifier keys are held down while other keys are pressed; a command gets executed once completely typed.

Vim retains each permutation of typed keys (e.g. order matters). This creates a path in the decision tree which unambiguously identifies any command.

Memory usage and customizability

Emacs executes many actions on startup, many of which may execute arbitrary user code. This makes Emacs take longer to start up (even compared to vim) and require more memory. However, it is highly customizable and includes a large number of features, as it is essentially an execution environment for a Lisp program designed for text-editing.

Vi is a smaller and faster program, but with less capacity for customization. vim has evolved from vi to provide significantly more functionality and customization than vi, making it comparable to Emacs.

User environment

Emacs, while also initially designed for use on a console, had X11 GUI support added in Emacs 18, and made the default in version 19. Current Emacs GUIs include full support for proportional spacing and font-size variation. Emacs also supports embedded images and hypertext.

Vi, like emacs, was originally exclusively used inside of a text-mode console, offering no graphical user interface (GUI). Many modern vi derivatives, e.g. MacVim and gVim, include GUIs. However, support for proportionally spaced fonts remains absent. Also lacking is support for different sized fonts in the same document.

Function/navigation interface

Emacs uses metakey chords. Keys or key chords can be defined as prefix keys, which put Emacs into a mode where it waits for additional key presses that constitute a key binding. Key bindings can be mode-specific, further customizing the interaction style. Emacs provides a command line accessed by M-x that can be configured to autocomplete in various ways. Emacs also provides a defalias macro, allowing alternate names for commands.

Vi uses distinct editing modes. Under "insert mode", keys insert characters into the document. Under "normal mode" (also known as "command mode", not to be confused with "command-line mode", which allows the user to enter commands), bare keypresses execute vi commands.

Keyboard

The expansion of one of Emacs' backronyms is Escape, Meta, Alt, Control, Shift, which neatly summarizes most of the modifier keys it uses, only leaving out Super. Emacs was developed on Space-cadet keyboards that had more key modifiers than modern layouts. There are multiple emacs packages, such as spacemacs or ergoemacs that replace these key combinations with ones easier to type, or customization can be done ad hoc by the user.

Vi does not use the Alt key and seldom uses the Ctrl key. vi's keyset is mainly restricted to the alphanumeric keys, and the escape key. This is an enduring relic of its teletype heritage, but has the effect of making most of vi's functionality accessible without frequent awkward finger reaches.

Language and script support

Emacs has full support for all Unicode-compatible writing systems and allows multiple scripts to be freely intermixed.

Vi has rudimentary support for languages other than English. Modern Vim supports Unicode if used with a terminal that supports Unicode.

2.1.6 Nano: The peacemaker amidst the editor war

Nano is a simple command line editor that is easy to use. It does not have the steep learning curve of vim or emacs. But it is not as powerful as vim or emacs as well. It is a common choice for beginners who just want to append a few lines to a file or make a few changes. It is also a non-modal editor like editor which uses modifier chording like emacs. However, it mostly uses the control key for this purpose and has only simple keybindings such as *Ctrl+O* to save and *Ctrl+X* to exit.

2.2 Vim

2.2.1 History

The history of Vim is a very long and interesting one.

Table 2.1: History of Vim

1967	QED text editor by Butler Lampson and Peter Deutsch for Berkeley Timesharing System.
1967	Ken Thompson and Dennis Ritchie's QED for MIT CTSS, Multics, and GE-TSS.
1969	Ken Thompson releases ed - The Standard Text Editor.
1976	George Coulouris and Patrick Mullaney release em - The Editor for Mortals.
1976	Bill Joy and Chuck Haley build upon em to make en, which later becomes ex.
1977	Bill Joy adds visual mode to ex.
1979	Bill Joy creates a hardlink 'vi' for ex's visual mode.
1987	Tim Thompson develops a vi clone for the Atari ST named STevie (ST editor for VI enthusiasts).
1988	Bram Moolenaar makes a stevie clone for the Amiga named Vim (Vi IMitation).

Teletypes

Definition 2.2.1 A **teletype** (TTY) or a teleprinter is a device that can send and receive typed messages from a distance.

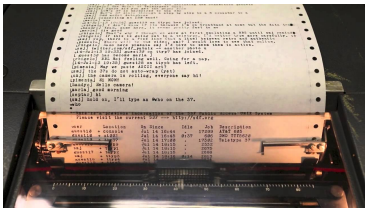

Figure 2.1: A Teletype

Very early computers used to use teletypes as the output device. These were devices that used ink and paper to actually *print* the output of the computer. These did not have an *automatic* refresh rate like modern monitors. Only when the computer sent a signal to the teletype, would the teletype print the output.

Due to these restrictions it was not economical or practical to print the entire file on the screen. Thus most editors used to print only one line at a time on the screen and did not have updating graphics.

QED

QED was a text editor developed by Butler Lampson and Peter Deutsch in 1967 for the Berkeley Timesharing System. It was a character-oriented editor that was used to create and edit text files. It used to print or edit only one character at a time on the screen. This is because the computers at that time used to use a teletype machine as the output device, and not a monitor.

Figure 2.2: Ken Thompson

Ken Thompson used this QED at Berkeley before he came to Bell Labs, and among the first things he did on arriving was to write a new version for the MIT CTSS system. Written in IBM 7090 assembly language, it differed from the Berkeley version most notably in introducing regular expressions [5] for specifying strings to seek within the document being edited, and to specify a substring for which a substitution should be made. Until that time, text editors could search for a literal string, and substitute for one, but not specify more general strings.

5: Regular expressions are a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithms for "find" or "find and replace" operations on strings. This will be discussed in detail in a later chapter.

Ken not only introduced a new idea, he found an inventive implementation: on-the-fly compiling. Ken's QED compiled machine code for each regular expression that created a NDFA (non-deterministic finite automaton) to do the search. He published this in C. ACM 11 #6, and also received a patent for the technique: US Patent #3568156.

While the Berkeley QED was character-oriented, the CTSS version was line-oriented. Ken's CTSS qed adopted from the Berkeley one the notion of multiple buffers to edit several files simultaneously and to move and copy text among them, and also the idea of executing a given buffer as editor commands, thus providing programmability.

When developing the MULTICS project, Ken Thompson wrote yet another version of QED for that system, now in BCPL [6] and now created trees for regular expressions instead of compiling to machine language.

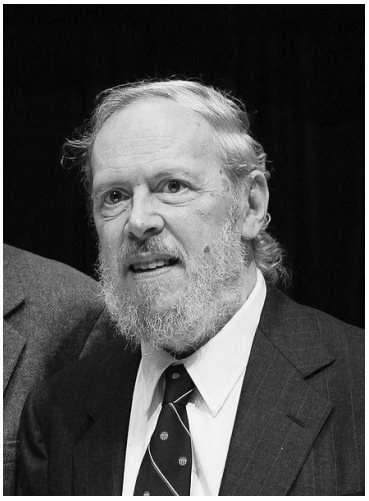

Figure 2.3: Dennis Ritchie

In 1967 when Dennis Ritchie joined the project, Bell Labs had slowly started to move away from Multics. While he was developing the initial stages of Unix, he rewrote QED yet again, this time for the GE-TSS system in Assembly language. This was well documented, and was originally intented to be published as a paper. [7]

6: BCPL ("Basic Combined Programming Language") is a procedural, imperative, and structured programming language. Originally intended for writing compilers for other languages, BCPL is no longer in common use.

ED

After their experience with multiple implementations of QED, Ken Thompson wrote **ed** in 1969.

This was now written in the newly developed B language, a predecessor to C. This implementation was much simpler than QED, and was line oriented. It stripped out much of regular expression support, and only had the support for $*$. It also got rid of multiple buffers and executing contents of buffer.

Slowly, with time, Dennis Ritchie created the C language, which is widely in use even today.

Ken Thompson re-wrote **ed** in C, and added back some of the complex features of QED, like back references in regular expressions.

Ed ended up being the **Standard Text Editor** for Unix systems.

Remark 2.2.1 Since all of Bell-Labs and AT&T's software was proprietary, the source code for ed was not available to the public. Thus, the *ed* editor accessible today in **GNU/Linux**, is another implementation of the original *ed* editor by the GNU project.

However, **ed** was not very user friendly and it was very terse. Although this was originally intented, since it would be very slow to print a lot of diagnostic messages on a teletype, slowly, as people moved to faster computers and monitors, they wanted a more user friendly editor.

VDU Terminals

Definition 2.2.2 A terminal that uses video display technology like cathode rat tubes (CRT) or liquid crystal displays (LCD) to display the terminal output is called a **VDU terminal**. (Video Display Unit)

These terminals were able to show video output, instead of just printing the output on paper. Although initially these were very expensive, and were not a household item, they were present in the research parks like Xerox PARC.

EM

George Coulouris (not the actor) was one of the people who had access to these terminals in his work at the Queen Mary College in London.

The drawbacks of **ed** were very apparent to him when using on these machines.

He found that the UNIX's **raw** mode, which was at that time totally unused, could be used to give some of the convenience and immediacy of feedback for text editing.

He claimed that although the **ed** editor was groundbreaking in its time, it was not very user friendly. He termed it as not being an editor for mortals.

He thus wrote **em** in 1976, which was an **Editor for Mortals**. [8]

Although em added a lot of features to ed, it was still a line editor, that is, you could only see one line at a time. The difference from ed was that it allowed visual editing, meaning you can see the state of the line as you are editing it.

7: At that time, systems did not have a standardized CPU architecture or a generalized low level compiler. Due to this, applications were not portable across systems. Each machine needed its own version of the application to be written from the scratch, mostly in assembly language.

The reference manual for GE-TSS QED can still be found on Dennis Ritchie's website Much of this information is taken from his blog.

Figure 2.4: Xerox Alto, one of the first VDU terminals with a GUI, released in 1973

8: George named em as Editor for Mortals because Ken Thompson visited his lab at QMC while he was developing it and said something like: "yeah, I've seen editors like that, but I don't feel a need for them, I don't want to see the state of the file when I'm editing". This made George think that Ken was not a mortal, and thus he named it Editor for Mortals.

Figure 2.5: A first generation Dectape (bottom right corner, white round tape) being used with a PDP-11 computer

Whereas most of the development of Multics and Unix was done in the United States, the development of **em** was done in the United Kingdom, in the Queen Mary College, which was the first college in the UK to have UNIX.

EN

In the summer of 1976, George Coulouris was a visiting professor at the University of California, Berkeley. With him, he had brought a copy of his **em** editor on a Dectape [9] and had installed it there on their departmental computers which were still using teletype terminals. Although em was designed for VDU terminals, it was still able to run (albeit slowly) on the teletype terminals by printing the current line every time.

9: A Dectape is a magnetic tape storage device that was used in the 1970s. It was used to store data and programs.

There he met Bill Joy, who was a PhD student at Berkeley. On showing him the editor, Bill Joy was very impressed and wanted to use it on the PDP-11 computers at Berkeley. The system support team at Berkley were using PDP-11 which used VDU Terminals, an environment where em would really shine.

He explained that 'em' was an extension of 'ed' that gave key-stroke level interaction for editing within a single line, displaying the up-to-date line on the screen (a sort of single-line screen editor). This was achieved by setting the terminal mode to 'raw' so that single characters could be read as they were typed - an eccentric thing for a program to do in 1976.

Figure 2.6: George Coulouris

Although the system support team at Berkeley were impressed by this editor, they knew that if this was made available to the general public, it would take up too much resources by going to the raw mode on every keypress. But Bill and the team took a copy of the source code just to see if they might use it.

George then took a vacation for a few weeks, but when he returned, he found that Bill had taken his ex as a starting point and had added a lot of features to it. He called it **en** initially, which later became **ex**.

EX

Bill Joy took inspiration from several other ed clones as well, and their own tweaks to ed, although the primary inspiration was **em**. Bill and Chuck Haley built upon em to make en, which later became ex.

This editor had a lot of improvements over em, such as adding the ability to add abbreviations (using the `ab` command), and adding keybindings (maps).

It also added the ability to mark some line using the `k` key followed by any letter, and then jump to that line from any arbitrary line using the `'` key followed by the letter.

Figure 2.7: Bill Joy

Slowly, with time, the modern systems were able able to handle the raw mode, and real time editing more and more. This led to the natural progression, What if we could see the entire file at once, and not just one line at a time?

VI

Bill added the **visual mode** to ex in 1977. [10] This was not a separate editor, but rather just another mode of the **ex** editor. You could open ex in visual mode using the `-v` flag to ex.

10: This visual mode is not the same as the visual mode in vim.

```
$ ex -v filename
```

This visual mode was the first time a text editor was **modal**. This means that the editor had different modes for different tasks. When you want to edit text, you would go to the insert mode, and type the text. When you want to navigate, you would go to the normal mode, and use the navigation keys and other motions defined in vi.

Slowly, as the visual mode became more and more popular, Bill added a hardlink to ex called **vi**. [11]

The modal version of vi was also inspired from another editor called **bravo**, which was developed at Xerox PARC. [12]

If you use vi/vim, you may notice that the key to exit the insert mode is `Esc`. This may seem inconveniently placed at the top left corner, but this was because the original vi was developed on a ADM-3A terminal, which had the `Esc` key to the left of the **Q** key, where modern keyboards have the `Tab` key. [13]

Also, the choice of h,j,k,l for navigation was because the ADM-3A terminal did not have arrow keys, rather, it had h,j,k,l keys for navigation.

This can be seen in Figure 2.8.

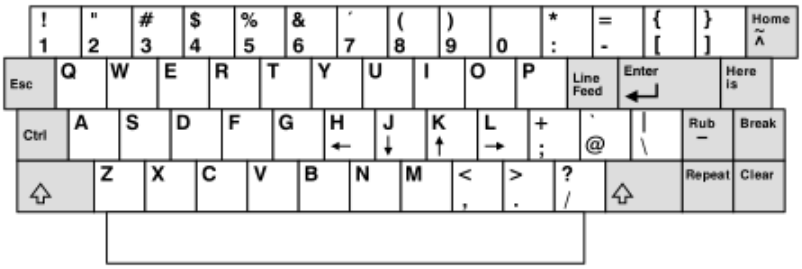

Figure 2.8: The Keyboard layout of the ADM-3A terminal

Bill Joy was also one of the people working on the **Berkeley Software Distribution (BSD) of Unix**. Thus he bundled vi with the first BSD distribution of UNIX released in 1978. The pre-installed nature of vi in the BSD Distribution made it very popular.

However, since both the source code of ed was restricted by Bell Labs - AT&T, and the source code of vi was restricted by the University of California, Berkeley, they could not be modified by the users or distributed freely.

This gave birth to a lot of clones of vi.

Vi Clones

The vi clones were written because the source code for the original version was not freely available until recently. This made it impossible to extend the functionality of vi. It also precluded porting vi to other operating systems, including Linux.

- **calvin**: a freeware "partial clone of vi" for use on MS-DOS. It has the advantages of small size (the .exe file is only 46.1KB!) and fast execution but the disadvantage that it lacks many of the ex commands, such as search and replace.

Figure 2.9: Stevie Editor

11: This means that it did not take up additional space on the disk, but was just another entry in the directory entry that pointed to the same inode which stored the ex binary path. Upon execution, **ex** would detect if it was called as **vi** and would start in visual mode by default. We have covered hardlinks in Chapter 1.

12: Xerox PARC has always been ahead of its time. The first graphical user interface was developed at Xerox PARC. The bravo editor used bitmapped graphics to display the text, and had extensive mouse support. The overdependence on the mouse in such an early time was one of the reasons that the bravo editor was not as popular as vi.

13: Since the placement of the Escape key is inconvenient in modern keyboard layouts, many people remap the Escape key to the `Caps Lock` key either in vim or in the operating system itself.

- **lemmy**: a shareware version of vi implemented for the Microsoft Windows platforms which combines the interface of vi with the look and feel of a Windows application.
- **nvi**: It is a re-implementation of the classic Berkeley vi editor, derived from the original 4.4BSD version of vi. It is the "official" Berkeley clone of vi, and it is included in FreeBSD and the other BSD variants.
- **stevie**: 'ST Editor for VI Enthusiasts' was developed by Tim Thompson for the Atari ST. It is a clone of vi that runs on the Atari ST. Tim Thompson wrote the code from scratch (not based on vi) and posted its source code as a free software to comp.sys.atari.st on June 1987. Later it was ported to UNIX, OS/2, and Amiga. Because of this independence from vi and ed's closed source license, most vi clones would base their work off of stevie to keep it free and open source.
- **elvis**: Elvis creator, Steve Kirkendall, started thinking of writing his own editor after Stevie crashed on him, causing him to lose hours of work and damaging his confidence in the editor. Stevie stored the edit buffer in RAM, which Kirkendall believed to be impractical on the MINIX operating system. One of Kirkendall's main motivation for writing his own vi clone was that his new editor stored the edit buffer in a file instead of storing it in RAM. Therefore, even if his editor crashed, the edited text could still be retrieved from that external file. Elvis was one of the first vi clones to offer support for GUI and syntax highlighting.

The clones add numerous new features which make them significantly easier to use than the original vi, especially for neophytes. A particularly useful feature in many of them is the ability to edit files in multiple windows. This facilitates working on more than one file at the same time, including cutting and pasting text among them.

Many of the clones also offer GUI versions of vi that operate under the X Windows system and can take advantage of bit-mapped (high resolution) displays and the mouse.

Vim

Bram Moolenaar, a Dutch programmer, was impressed by STeVIe, a vi clone for the Atari ST. But he was working with the Commodore Amiga at that time, and there was no vi clone for the Amiga. So Bram began working on the stevie clone for the AmigaOS in 1988.

Figure 2.10: Bram Moolenaar

He released the first public release (v 1.14) in 1991 as visible in Figure 2.11.

Since Vim was based off of Stevie, and not ed or vi so it could be freely distributed. It was licensed under a charityware license, named as Vim License. The license stated that if you liked the software, you should consider making a donation to a charity of your choice.

Moolenaar was an advocate of a NGO based in Kibaale, Uganda, which he founded to support children whose parents have died of AIDS. In 1994, he volunteered as a water and sanitation engineer for the Kibaale Children's Centre and made several return trips over the following twenty-five years.

Later Vim was re-branded as 'Vi IMproved' as seen in Figure 2.12.

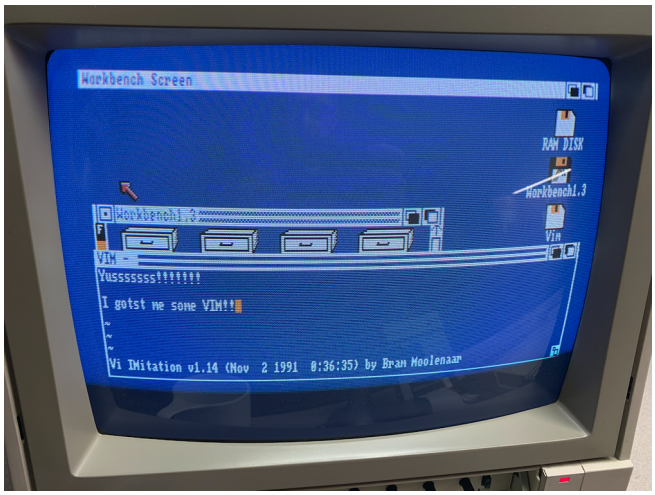

Figure 2.11: The inital version of Vim, when it was called Vi IMitation

Figure 2.12: Vim 9.0 Start screen

Vim has been in development for over 30 years now, and is still actively maintained. It has added a lot of features over the years, such as syntax highlighting, plugins, completion, PCRE support, mouse support, etc.

neovim

Recently there have been efforts to modernize the vim codebase. Since it is more than 30 years old, it has a lot of legacy code. The scripting language of vim is also not a standard programming language, but rather a custom language called vimscript.

To counter this, a new project called **neovim** has been started. It uses **lua** as the scripting language, and has a lot of modern features like out of the box support for LSP, [14] better mouse integration, etc.

In this course, we will be learning only about basic vi commands and we will be using vim as the editor.

14: LSP stands for Language Server Protocol. It is a protocol that allows the editor to communicate with a language server to provide features like autocompletion, go to definition, etc. This makes vim more like an IDE.

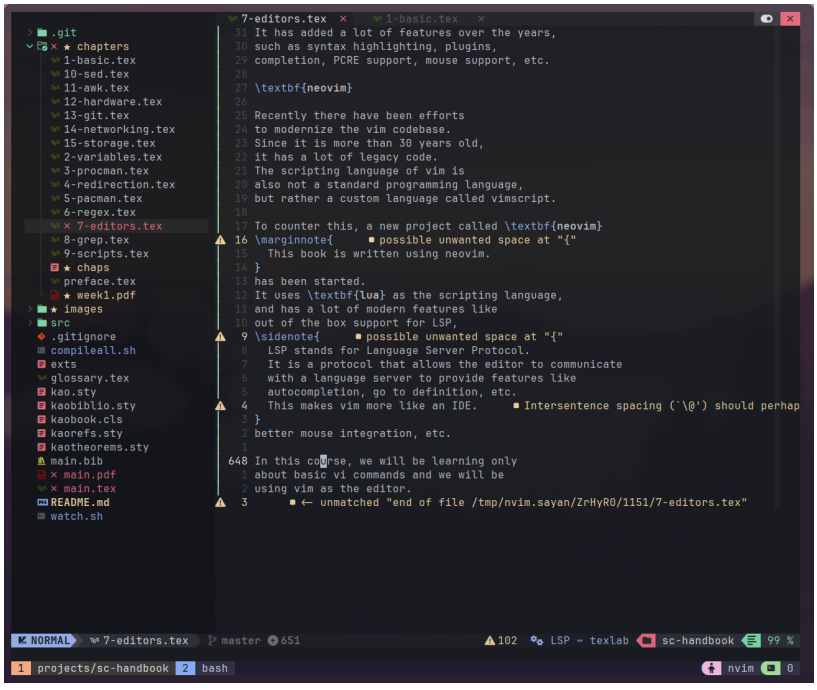

Figure 2.13: Neo Vim Window editing this book

This book is written using neovim.

2.2.2 Ed Commands

Before we move on to Vi Commands, let us first learn about the basic ed commands. These will also be useful in vim, since the ex mode of vim is based on ed/ex where we can directly use ed commands on our file in the buffer.

Description	Commands
Show the **P**rompt	`P`
Command Format	`[addr[,addr]]cmd[params]`
Commands for location	`1 . $ % + - , ; /RE/`
Commands for editing	`f p a c d i j s m u`
Execute a shell *command*	`!command`
edit a file	`e filename`
read file contents into buffer	`r filename`
read *command* output into buffer	`r !command`
write buffer to filename	`w filename`
quit	`q`

Table 2.2: Ed Commands

Commands for location

Commands	Description
a number like `2`	refers to second line of file
`.`	refers to current line
`$`	refers to last line
`%`	refers to all the lines
`+`	line after the cursor (current line)
`-`	line before the cursor (current line)
`,`	refers to buffer holding the file or last line in buffer
`;`	refers to current position to end of the file
`/RE/`	refers line matched by pattern specified by 'RE'

Table 2.3: Commands for location

Commands for Editing

Commands	Description
`f`	show name of **f**ile being edited
`p`	**p**rint the current line
`a`	**a**ppend at the current line
`c`	**c**hange the current line
`d`	**d**elete the current line
`i`	**i**nsert line at the current position
`j`	**j**oin lines
`s`	**s**earch for regex pattern
`m`	**m**ove current line to position
`u`	**u**ndo latest change

Table 2.4: Commands for Editing

Let us try out some of these commands in the ed editor.

Lets start with creating a file, which we will then open in the ed editor.

```
$ echo "line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file" > test.txt
```

This creates a file in the current working directory with the name `test.txt` and the contents as given above.

We invoke ed by using the executable `ed` and providing the filename as an argument.

```
$ ed test.txt
117
```

As soon as we run it, you will see a number, which is the number of characters in the file. The terminal may seem hung, since there is no prompt, of either the bash shell, or of the ed editor. This is because ed is a line editor, and it does not print the contents of the file on the screen.

Off the bat, we can observe the terseness of the ed editor since it does not even print a prompt. To turn it on, we can use the `P` command. The default prompt is `*`.

```
ed test.txt
117
P
*
```

Now we can see the prompt `*` is always present, whenever the ed editor expects a command from the user.

Lets go to the first line of the file using the `1` command. We can also go to the last line of the file using the `$` command.

```
*1
line-1 hello world
*$
line-4 end of file
*
```

To print out all the lines of the file, we can use the `,` or `%` with `p` command.

```
*,p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

```
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

However, if we use the `,` command without the `p` command, it will not print all the lines. Rather, it will just move the cursor to the last line and print the last line.

```
*,
line-4 end of file
```

We can also print any arbitrary line range using the line numbers separated by a comma and followed by the `p` command.

```
*2,3p
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
```

One of the pioneering features of ed was the ability to search for a pattern in the file. Let us quickly explain the syntax of the search command. [15]

15: The details of regular expressions will be covered in a later chapter.

```
*/hello/
line-1 hello world
```

We may or may not include the `p` command after the last `/` in the search command.

We can advance to the next line using the `+` command.

```
*p
line-1 hello world
*+
line-2 welcome to line editor
```

And go to the previous line using the `-` command.

```
*p
line-2 welcome to line editor
*-
line-1 hello world
```

We can also print all the lines from the current line to the end of the file using the `;p` command.

```
*.
line-2 welcome to line editor
*;p
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
```

We can also run arbitrary shell commands using the `!` command.

```
*! date
Mon Jun 10 11:36:34 PM IST 2024
!
```

The output of the command is shown to the screen, however, it is not saved in the buffer.

To read the output of a command into the buffer, we can use the `r` command.

```
*r !date
32
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
Mon Jun 10 11:37:42 PM IST 2024
```

The output after running the `r !date` command is the number of characters read into the buffer. We can then print the entire buffer using the `%p` command.

The read data is appended to the end of the file.

We can write the buffer [16] to the disk using the `w` command.

16: Remember that the buffer is the in-memory copy of the file and any changes made to the buffer are not saved to the file until we write the buffer to the file.

```
*w
149
```

The output of the `w` command is the number of characters written to the file.

To exit ed, we can use the `q` command.

```
*q
```

To delete a line, we can use the `d` command. Lets say we do not want the date output in the file. We can re-open the file in ed and remove the last line.

```
$ ed test.txt
149
P
*$
Mon Jun 10 11:38:49 PM IST 2024
*d
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
line-4 end of file
*wq
117
```

We can add lines to the file using the `a` command. This appends the line after the current line. On entering this mode, the editor will keep on taking input for as many lines as we want to add. To end the input, we can use the `.` command on a new line.

```
$ ed test.txt
117
P
*3
line-3 ed is perhaps the oldest editor out there
*a
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
.
*%p
line-1 hello world
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
line-4 end of file
*
```

We can also utilize the regular expression support in ed to perform search and replace operations. This lets us either search for a fixed string and replace with another fixed string, or search for a pattern and replace it with a fixed string.

Let us change hello world to hello universe.

```
*1
line-1 hello world
*s/world/universe/
line-1 hello universe
```

```
*%p
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QED
which was made multiple times by thompson and ritchie
before ed was made.
line-4 end of file
*
```

We can print the name of the currently opened file using the `f` command.

```
*f
test.txt
```

If we wish to join two lines, we can use the `j` command. Let us join lines 4,5, and 6.

```
*4
perhaps not, since we know it was inspired from QED
*5
which was made multiple times by thompson and ritchie
*6
before ed was made.
*4,5j
*4
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchie
*5
before ed was made.
*4,5j
*4
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
*
```

Here we can see that we do the joining in two steps, first the lines 4 and 5 are joined, and then the newly modified line 4 and 5 are joined.

We can move a line from its current position to another line using the `m` command.

Lets insert a line-0 at the end of the file and then move it to the beginning of the file.

```
*7
line-4 end of file
*a
line-0 in the beginning, there was light
.
*8
line-0 in the beginning, there was light
*m0
*1,4p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
*
```

We can also undo the last change using the `u` command. [17]

17: The undo command in ed is not as powerful as the undo command in vim. In vim, we can undo multiple changes using the `u` command. In ed, we can only undo the last change. If we run the `u` command multiple times, it will undo the last change of undoing the last change, basically redoing the last change.

```
*1
line-0 in the beginning, there was light
*s/light/darkness
line-0 in the beginning, there was darkness
*u
*.
line-0 in the beginning, there was light
*
```

If search and replace is not exactly what we want, and we want to totally change the line, we can use the `c` command. It will let us type a new line, which will replace the current line.

```
*%p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-3 ed is perhaps the oldest editor out there
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
line-4 end of file
*4
line-3 ed is perhaps the oldest editor out there
*c
line-4 ed is the standard editor for UNIX
.
*4
line-4 ed is the standard editor for UNIX
*
```

Just like the `a` command, we can also use the `i` command to insert a line at the current position. This will move the current line to the next line.

```
*6
line-4 end of file
*i
before end of file
.
*6,$p
before end of file
line-4 end of file
*
```

Finally, we can also number the lines using the `n` command.

```
*%p
line-0 in the beginning, there was light
line-1 hello universe
line-2 welcome to line editor
line-4 ed is the standard editor for UNIX
perhaps not, since we know it was inspired from QEDwhich was made
    multiple times by thompson and ritchiebefore ed was made.
before end of file
line-4 end of file
*%n
1       line-0 in the beginning, there was light
2       line-1 hello universe
3       line-2 welcome to line editor
```

```
4       line-4 ed is the standard editor for UNIX
5       perhaps not, since we know it was inspired from QEDwhich
    was made multiple times by thompson and ritchiebefore ed was
    made.
6       before end of file
7       line-4 end of file
*
```

2.2.3 Exploring Vim

There are a plethora of commands in vim. We wont be able to cover all of them in this course. Only the basic commands required to get started with using vim as your primary editor would be covered. A detailed tutorial on vim can be found by running the command `vimtutor` in your terminal.

```
$ vimtutor
```

This opens a temporary files that goes through a lot of sections of vim, explaining the commands in detail. This opens the text file in vim itself, so you can actually try out each exercise as and when you read it. Many exercises are present in this file to help you remember and master commands. Feel free to modify the file since it is a temporary file and any changes made is lost if the command is re-run.

To open a file in vim, we provide the filename as an argument to the vim executable.

```
$ vim test.txt
```

Sometimes the normal mode is called command mode or escape mode, since we can run commands in this mode and we press the `Esc` key to go to this mode. However, the ex mode is also called command mode, since we can run ex commands in this mode. To avoid confusions, we will refer to the navigational(default) mode as normal mode, since vim internally also refers to it as normal mode, and we will refer to the ex mode as ex mode.

Modal Editor

Vim is a modal editor, which means that it has different modes that it operates in. The primary modes are:

- Normal/Command Mode - The default mode where we can navigate around the file, and run vim commands.
- Insert Mode - The mode where we can type text into the file.
- Visual Mode - The mode where we can select text to copy, cut, or delete.
- Ex Mode - The mode where we can run ex commands.

Pressing the `Esc` key takes you to the normal mode from any other mode.

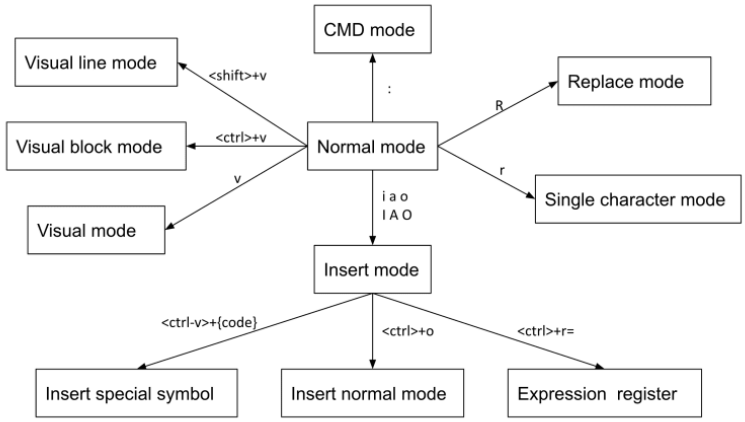

Figure 2.14: Simplified Modes in Vim

The figure Figure 2.14 demonstrates how to switch between the different modes in vim. [18]

18: This is a simplified version of the modes in vim. There are other interim modes and other keystrokes that toggle the modes. This is shown in detail in Figure 2.15.

Commands in Ex mode

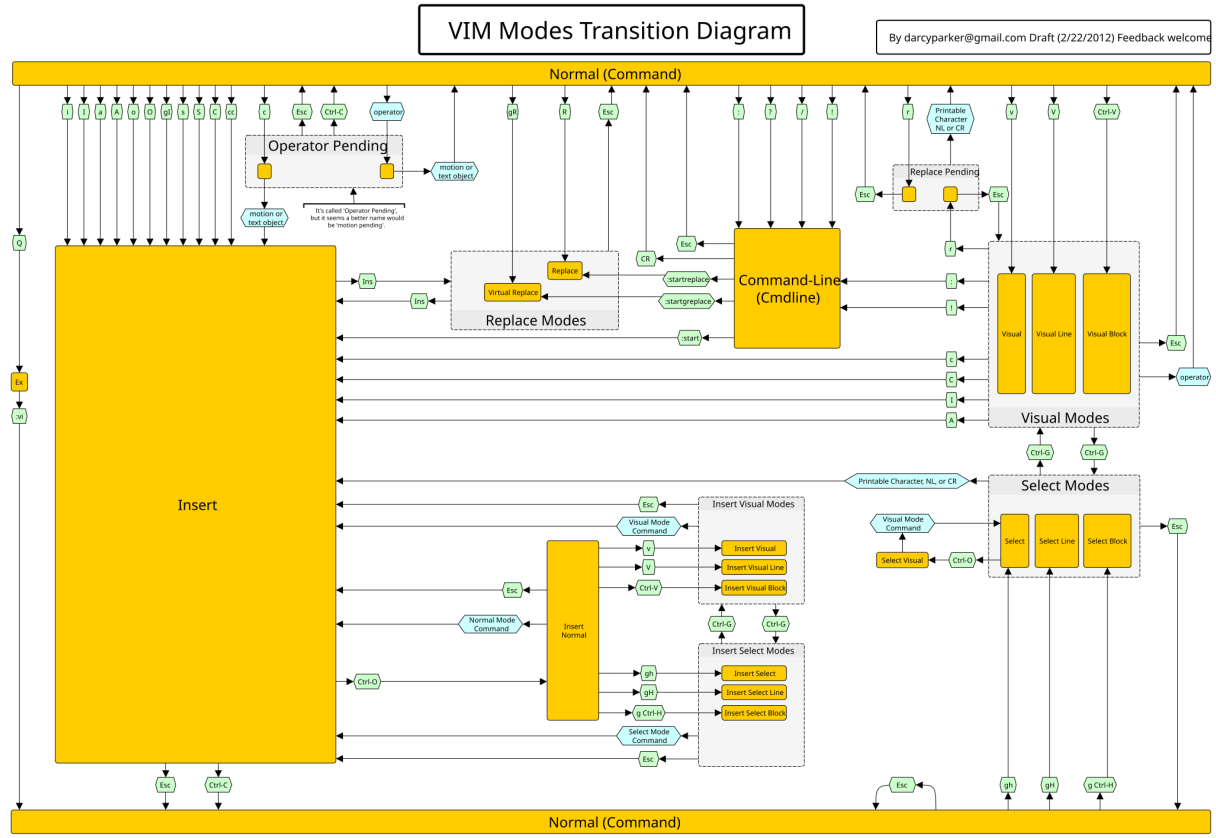

Figure 2.15: Detailed Modes in Vim

Since we are already familar with commands in **ed**, most of the commands are same/similar in **ex** mode of vim.

Table 2.5: Ex Commands in Vim

Key	Description
`:f`	show name of file
`:p`	print current line
`:a`	append at current line
`:c`	change current line
`:d`	delete current line
`:i`	insert line at current position
`:j`	join lines
`:s`	search and replace regex pattern in current line
`:m`	move current line to position
`:u`	undo latest change
`:w [filename]`	write buffer to filename
`:q`	quit if no change
`:wq`	write buffer to filename and quit
`:x`	write buffer to filename and quit
`:q!`	quit without saving
`:r filename`	read file contents into buffer
`:r !command`	read command output into buffer
`:e filename`	edit a file
`:sp [filename]`	split the screen and open another file
`:vsp [filename]`	vertical split the screen and open another file

There are many more commands in the ex mode of vim. Along with implementing the original ed commands, it also has a lot of additional

commands to make it more integrated with the vim editor, such as the ability to open split windows, new tabs, buffers, and perform normal mode commands in ex mode.

Basic Navigation

The basic keys for moving around in a text file in vim are the `h,j,k,l` keys. They move the cursor one character to the left, down, up, and right respectively. These keys are chosen because they are present on the home row of the keyboard, and do not require the user to move their hands from the home row to navigate. [19]

19: These were historically chosen because the ADM-3A terminal had these keys for navigation as seen in Figure 2.8.

Along with these, we have keys to navigate word by word, or to move to the next pattern match, or the next paragraph, spelling error, etc.

Table 2.6: Navigation Commands in Vim

Key	Description
`h`	move cursor left
`j`	move cursor down
`k`	move cursor up
`l`	move cursor right
`w`	move to the beginning of the next word
`e`	move to the end of the current word
`b`	move to the beginning of the previous word
`%`	move to the matching parenthesis, bracket, or brace
`0`	move to the beginning of the current line
`$`	move to the end of the current line
`/`	search forward for a pattern
`?`	search backward for a pattern
`n`	repeat the last search in the same direction
`N`	repeat the last search in the opposite direction
`gg`	move to the first line of the file
`G`	move to the last line of the file
`1G`	move to the first line of the file
`1gg`	move to the first line of the file
`:1`	move to the first line of the file
`{`	move to the beginning of the current paragraph
`}`	move to the end of the current paragraph
`fg`	move cursor to next occurrence of 'g' in the line
`Fg`	move cursor to previous occurrence of 'g' in the line

Wait you forgot your cursor behind!

All of the above commands move the cursor to the mentioned location. However, if you want to move the entire screen, and keep the cursor at the its current position, you can use the `z` command along with `t` to top the line, `b` to bottom the line, and `z` to center the line.

There are other commands using the `Ctrl` key that moves the screen, and not the cursor.

Table 2.7: Moving the Screen Commands in Vim

Key	Description
`Ctrl+F`	move forward one full screen
`Ctrl+B`	move backward one full screen
`Ctrl+D`	move forward half a screen
`Ctrl+U`	move backward half a screen
`Ctrl+E`	move screen up one line
`Ctrl+Y`	move screen down one line

Replacing Text

Usually in other text editors, if you have a word, phrase, or line which you want to replace with another, you would either press the backspace or delete key to remove the text, and then type the new text. However, in vim, there is a more efficient way to replace text.

Key	Description
`r`	replace the character under the cursor
`R`	replace the character from the cursor till escape is pressed
`cw`	change the word under the cursor
`c4w`	change the next 4 words
`C`	delete from cursor till end of line and enter insert mode
`cc`	delete entire line and enter insert mode
`5cc`	delete next 5 lines and enter insert mode
`S`	delete entire line and enter insert mode
`s`	delete character under cursor and enter insert mode

Table 2.8: Replacing Text Commands in Vim

Toggling Case

You can toggle the case of a character, word, line, or any arbitrary chunk of the file using the or the `g` command.

Key	Description
`~`	toggle the case of the character under the cursor
`g w`	toggle the case of the word under the cursor
`g 0`	toggle the case from cursor till beginning of line
`g $`	toggle the case from cursor till end of line
`g {`	toggle the case from cursor till previous empty line
`g {`	toggle the case from cursor till next empty line
`g %`	toggle the case from the bracket, brace, or parenthesis till its pair

Table 2.9: Toggling Case Commands in Vim

You might start to see a pattern emerging here. Many commands in vim do a particular command, and on which text it operates is determined by the character followed by it. Such as `c` to change, `d` to delete, `y` to yank, etc. The text on which the command operates is mentioned using `w` for the word, `0` for till beginning of line, etc.

This is not a coincidence, but rather a design of vim to make it more efficient to use. The first command is called the operator command, and the second command is called the motion command.

Vim follows a `operator-count-motion` pattern. For example: `d2w` deletes the next 2 words. This makes it very easy to learn and remember commands, since you are literally typing out what you want to do.

Deleting or Cutting Text

In Vim, the delete command is used to cut text from the file.

Motion - till, in, around

By now you should notice that `dw` doesn't always delete the word under the cursor. Technically `dw` means delete till the beginning of the next word. So if you press `dw` at the beginning of a word, it will delete the word under the cursor. But if your cursor is in the middle of a word and you type `dw`, it will only delete the part of the word till the beginning of the next word from the cursor position.

Table 2.10: Deleting Text Commands in Vim

Key	Description
`x`	delete the character under the cursor
`X`	delete the character before the cursor
`5x`	delete the next 5 characters
`dw`	delete the word under the cursor
`d4w`	delete the next 4 words
`D`	delete from cursor till end of line
`dd`	delete entire line
`6dd`	delete next 6 lines

To delete the entire word under the cursor, regardless of where the cursor is in the word, you can use the `diw` this means **d**elete **i**nside **w**ord.

However, now you may notice that `diw` doesn't delete the space after the word. This results in two consequtive spaces, one from the end of the word, and one from the beginning of the word being left behind. To delete the space as well, you can use `daw` which means **d**elete **a**round **w**ord.

This works not just with `w` but with any other motion such as **d**elete **i**nside **p**aragraph, which will delete the entire paragraph under the cursor, resulting in two empty lines being left behind, and **d**elete **a**round **p**aragraph, which will delete the entire paragraph under the cursor, and only one empty line being left behind.

Try out the same with deleting inside other items, such as brackets, parenthesis, braces, quotes, etc. The syntax remains the same, `di{`, `di[`, `di(`, `di"`, `di'`, etc.

Yanking and Pasting Text

Yes, copying is called yanking in vim. The command to yank is `y` and to paste is `p`. You can combine `y` with all the motions and **in** and **around** motions as earlier. You can also add the count to yank multiple lines or words.

Table 2.11: Deleting Text Commands in Vim

Key	Description
`yy`	yank the entire line
`yw`	yank the word under the cursor
⋮	⋮
`p`	paste the yanked text after the cursor
`P`	paste the yanked text before the cursor

Remark 2.2.2 Important to note that the commands

```
yy
```

and

```
0y$
```

are not the same. The first command yanks the entire line, including the newline character at the end of the line. The second one yanks the entire line, but does not include the newline character at the end of the line. Thus if you directly press `p` after the first command, it will paste the line below the current line, and if you press `p` after the

second command, it will paste the line at the end of the current line.

Undo and Redo

The undo command in vim is `u` and the redo command is `Ctrl+R`. You can undo multiple changes, unlike ed.

Remark 2.2.3 If you want to use vim as your primary editor, it is highly recommended to install the **mbbill/undotree** plugin. This plugin will show you a tree of all the changes you have made in the current buffer, and you can go to any point in the tree and undo or redo changes. This becomes very useful if you undo too many changes and by mistake make a new change, this changes your branch in undo tree, and you cannot redo the changes you undid. With the undotree plugin, you can switch branches of the undo tree and redo the changes.

Searching and Replacing

The search command in vim is `/` for forward search and `?` for backward search. You can use the `n` command to repeat the last search in the same direction, and the `N` command to repeat the last search in the opposite direction. For example, if you perform forward search then using the `n` command will search for the next occurrence of the pattern in the forward direction, and using the `N` command will search for the previous occurrence of the pattern in the backward direction. However if you perform a backward search using the `?` command, then using the `n` command will search for the previous occurrence of the pattern in the backward direction, and using the `N` command will search for the next occurrence of the pattern in the forward direction.

You can also use the $*$ command to search for the word under the cursor, and the `#` command to search for the previous occurrence of the word under the cursor.

You can perform search and replace using the `:s` command. The command takes a line address on which to perform the search and replace. Usually you can use the `%` address to search in the entire file, or the `.,$` address to search from cursor till the end of the file.

You can also use any line number to specify the address range, similar to the ed editor.

```
:[addr]s/pattern/replace/[flags]
```

The flags at the end of the search and replace command can be **g** to replace all occurrences in the line, and **c** to confirm each replacement.

The address can be a single line number, a range of line numbers, or a pattern to search for. The pattern can be a simple string, or a regular expression.

Some examples of addresses are shown in Table 2.12.

Insert Mode

You can enter insert mode from escape mode using the keys listed in Table 2.13. In insert mode, if you want to insert any non-graphical character, you can do that by pressing `Ctrl+V` followed by the key combination for

Table 2.12: Address Types in Search and Replace

Key	Description
`m,n`	from line m to line n
`m`	line m
`m,$`	from line m to end of file
`.,$`	from current line to end of file
`1,n`	from line 1 to line n
`/regex/,n`	from line containing regex to line n
`m,/regex/`	from line m to line containing regex
`.,/regex/`	from current line to line containing regex
`/regex/,.`	from line containing regex to current line
`1,/regex/`	from the first line to line containing regex
`/regex/,$`	from line containing regex to the last line
`/regex1/;/regex2/`	from line containing regex1 to line containing regex2
`%`	entire file

Table 2.13: Keys to enter Insert Mode

Key	Description
`i`	enter insert mode before the cursor
`a`	enter insert mode after the cursor
`I`	enter insert mode at the beginning of the line
`A`	enter insert mode at the end of the line
`o`	add new line below the current line and enter insert mode
`O`	add new line above the current line and enter insert mode

the character. For example, to insert a newline character, you can press `Ctrl+V` followed by `Enter`.

These are just the basic commands to get you started with vim. You can refer to vim cheat sheets present online to get more familiar with the commands.

- `https://vim.rtorr.com/` is a good text based HTML cheat sheet for vim.
- `https://vimcheatsheet.com/` is a paid graphical cheat sheet for vim. [20]

20: A free version of the graphical cheat sheet is shown in Figure 2.16.

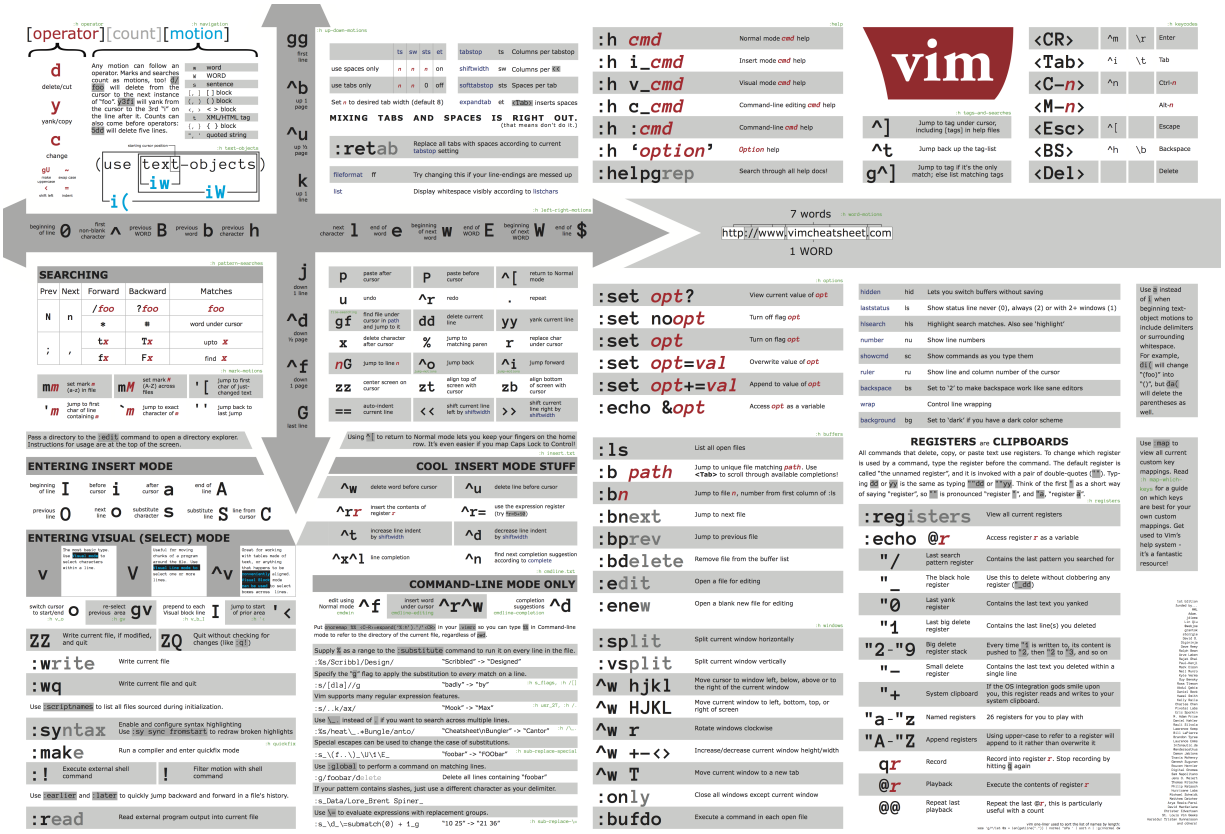

Figure 2.16: Vim Cheat Sheet

2.3 Emacs

2.3.1 History

Emacs was mostly developed by Richard Stallman and Guy Steele.

Table 2.14: History of Emacs

1962	TECO (Tape Editor and Corrector) was developed at MIT.
1976	Richard Stallman visits Stanford AI Lab and sees Fred Wright's E editor.
1978	Guy Steele accumulates a collection of TECO macros into EMACS.
1979	EMACS becomes MIT's standard text editor.
1981	James Gosling writes Gosling Emacs that runs on UNIX.
1984	Richard Stallman starts GNU Emacs - a free software alternative to Gosling Emacs.

Figure 2.17: Emacs Logo

Figure 2.18: Richard Stallman - founder of GNU and FSF projects

TECO

TECO was developed at MIT in 1962. It was a text editor used to correct the output of the PDP-1 computer. It is short for Tape Editor and Corrector. Unlike most modern text editors, TECO used separate modes in which the user would either add text, edit existing text, or display the document One could not place characters directly into a document by typing them into TECO, but would instead enter a character ('i') in the TECO command language telling it to switch to input mode, enter the required characters, during which time the edited text was not displayed on the screen, and finally enter a character (<esc>) to switch the editor back to command mode. This is very similar to how vi works.

Stallman's Visit to Stanford

In 1976, Richard Stallman visited the Stanford AI Lab where he saw Fred Wright's E editor. He was impressed by E's WYSIWYG [21] interface where you do not need to tackle multiple modes to edit a text file. This is the default behaviour of most modern editors now. He then returned to MIT where he found that Carl Mikkelsen had added to TECO a combined display/editing mode called Control-R that allowed the screen display to be updated each time the user entered a keystroke. Stallman reimplemented this mode to run efficiently and added a macro feature to the TECO display-editing mode that allowed the user to redefine any keystroke to run a TECO program.

Initially TECO was able to only edit the file sequentially, page by page. This was due to earlier memory restrictions of the PDP-1. Stallman

21: What You See Is What You Get

modified TECO to read the entire file into the buffer, and then edit the buffer in memory allowing for random access to the file.

Too Many Macros!

The new version of TECO quickly became popular at the AI Lab and soon accumulated a large collection of custom macros whose names often ended in MAC or MACS, which stood for macro. This quickly got out of hand as there were many divergent macros, and a user would be totally lost when using a co-worker's terminal.

In 1979, Guy Steele combined many of the popular macros into a single file, which he called EMACS, which stood for Editing MACroS, or E with MACroS.

To prevent thousands of forks of EMACS, Stallman declared that 'EMACS was distributed on a basis of communal sharing, which means all improvements must be given back to me to be incorporated and distributed.'

Till now, the EMACS, like TECO, ran on the PDP-10 which ran the ITS operating system and not UNIX.

Figure 2.19: Guy L. Steele Jr. combined many divergent TECO with macros to create EMACS

EINE ZWEI SINE and other clones

No, that is not German. These are some of the popular clones of EMACS made for other operating systems.

EINE [22] was a text editor developed in the late 1970s. In terms of features, its goal was to 'do what Stallman's PDP-10 (original) Emacs does'. Unlike the original TECO-based Emacs, but like Multics Emacs, EINE was written in Lisp. It used Lisp Machine Lisp.

In the 1980s, EINE was developed into ZWEI [23]. Innovations included programmability in Lisp Machine Lisp, and a new and more flexible doubly linked list method of internally representing buffers.

SINE [24] was written by Owen Theodore Anderson in 1981.

In 1978, Bernard Greenberg wrote a version of EMACS for the Multics operating system called Multics EMACS. This used Multics Lisp.

22: EINE stands for Eine Is Not EMACS

23: ZWEI stands for ZWEI Was Eine Initially

These kinds of recursive acronyms are common in the nix world. For example, GNU stands for GNU's Not Unix, WINE (A compatibility layer to run Windows applications) is short for WINE Is Not an Emulator.

24: SINE stands for SINE Is Not EINE

Gosling Emacs

In 1981, James Gosling wrote Gosling Emacs for UNIX. It was written in C and used Mocklisp, a language with lisp-like syntax, but not a lisp. It was not free software.

Figure 2.20: James Gosling - creator of Gosling Emacs and later Java

GNU Emacs

In 1983, Stallman started the GNU project to create a free software alternatives to proprietary softwares and ultimately to create a free [25] operating system.

In 1984, Stallman started GNU Emacs, a free software alternative to Gosling Emacs. It was written in C and used a true Lisp dialect, Emacs Lisp as the extension language. Emacs Lisp was also implemented in C. This is the version of Emacs that is most popular today and available on most operating systems repositories.

25: Recall from the previous chapter that free software does not mean a software provided gratis, but a software which respects the user's freedom to run, copy, distribute, and modify the software. It is like free speech, not free beer.

How the developer's keyboard influences the editors they make

Remember that ed was made while using **ADM-3A** which looked like Figure 2.21.

Figure 2.21: ADM-3A terminal

Figure 2.22: Space Cadet Keyboard

Whereas emacs was made while the **Knight keyboard** and the **Space Cadet keyboard** were in use, which can be seen in Figure 2.22.

Notice how the ADM-3A has very limited modifier keys, and does not even have arrow keys. Instead it uses `h,j,k,l` keys as arrow keys with a modifier. This is why vi uses mostly key combinations and a modal interface. Vi also uses the `Esc` key to switch between modes, which is present conveniently in place of the Caps Lock or Tab key in modern keyboard layouts.

The Space Cadet keyboard has many modifier keys, and even a key for the Meta key. This is why emacs uses many key modifier combinations, and has a lot of keybindings.

2.3.2 Exploring Emacs

This is not a complete overview of Emacs, or even its keybindings. A more detailed reference card can be found on their website.

Opening a File

We can open a file in emacs by providing its filename as an argument to the emacs executable.

```
1 $ emacs test.txt
```

Most of emacs keybindings use modifier keys such as the `Ctrl` key, and the `Meta` key. The `Meta` key is usually the `Alt` key in modern keyboards. In the reference manual and here, we will be representing the `Meta` key as `M-` and the `Ctrl` key as `C-`.

Basic Navigation

These keys are used to move around in the file. Like vim, emacs also focusses on keeping the hands free from the mouse, and on the keyboard. All the navigation can be done through the keyboard.

Table 2.15: Navigation Commands in Emacs

Key	Description
`C-p`	move up one line
`C-b`	move left one char
`C-f`	move right one char
`C-n`	move down one line
`C-a`	goto beginning of current line
`C-e`	goto end of current line
`C-v`	move forward one screen
`M-<`	move to first line of the file
`M-b`	move left to previous word
`M-f`	move right to next word
`M->`	move to last line of the file
`M-a`	move to beginning of current sentence
`M-e`	move to end of current sentence
`M-v`	move back one screen

Exiting Emacs

We can exit emacs either with or without saving the file. We can also suspend emacs and return to the shell. This is a keymapping of the shell, and not of emacs.

Key	Description
`C-x C-s`	save buffer to file
`C-z`	suspend emacs
`C-x C-c`	exit emacs and stop it

Table 2.16: Exiting Emacs Commands

Searching Text

Emacs can search for a fixed string, or a regular expression and replace it with another string.

Key	Description
`C-s`	search forward
`C-r`	search backward
`M-x`	replace string

Table 2.17: Searching Text Commands in Emacs

Copying and Pasting

Copying can done by marking the region, and then copying it.

Key	Description
`M-backspace`	cut the word before cursor
`M-d`	cut the word after cursor
`M-w`	copy the region
`C-w`	cut the region
`C-y`	paste the region
`C-k`	cut from cursor to end of line
`M-k`	cut from cursor to end of sentence

Table 2.18: Copying and Pasting Commands in Emacs

2.4 Nano

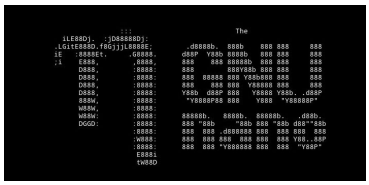

Figure 2.23: Nano Text Editor

Although vim and emacs are the most popular command line text editors, nano is also a very useful text editor for beginners. It is very simple and does not have a steep learning curve.

It is a non-modal text editor, which means that it does not have different modes for different actions. You can directly start typing text as soon as you open **nano**.

Although it uses modifier keys to invoke commands, it does not have a lot of commands as vim or emacs.

2.4.1 History

Pine [26] was a text-based email client developed at the University of Washington. It was created in 1989. The email client also had a text editor built in called Pico.

Although the license of Pine and Pico may seem open source, it was not. The license was restrictive and did not allow for modification or redistribution. [27]

Due to this, many people created clones of Pico with free software licenses. One of the most popular clones was TIP (TIP isn't Pico) which was created by Chris Allegretta in 1999. Later in 2000 the name was changed to Nano. [28] In 2001, nano became part of the GNU project.

GNU nano implements several features that Pico lacks, including syntax highlighting, line numbers, regular expression search and replace, line-by-line scrolling, multiple buffers, indenting groups of lines, rebindable key support, and the undoing and redoing of edit changes.

In most modern linux systems, the **nano** binary is present along with the **pico** binary, which is actually a symbolic link to the **nano** binary.

You can explore this by finding the path of the executable using the `which` command and long-listing the executable.

```
1 $ which pico
2 /usr/bin/pico
3 $ ls -l /usr/bin/pico
4 lrwxrwxrwx 1 root root 22 Sep  6  2023 /usr/bin/pico -> /etc/
      alternatives/pico
5 $ ls -l /etc/alternatives/pico
6 lrwxrwxrwx 1 root root 9 Sep  6  2023 /etc/alternatives/pico -> /
      bin/nano
```

Remark 2.4.1 Note that here we have a symlink to another symlink. Theoretically, you can extend to as many levels of chained symlinks as you want. Thus, to find the final sink of the symlink chain, you can use the `readlink -f` command or the `realpath` command.

```
1 $ realpath $(which pico)
2 /usr/bin/nano
```

26: It is believed that pine stands for Pine is Not Elm, Elm being another text-based email client. However, the author clarifies that it was not named with that in mind. Although if a backronym was to be made, he prefered 'Pine is Nearly Elm' or 'Pine is No-longer Elm'

27: Up to version 3.91, the Pine license was similar to BSD, and it stated that 'Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee to the University of Washington is hereby granted ...' The university registered a trademark for the Pine name with respect to 'computer programs used in communication and electronic mail applications' in March 1995. From version 3.92, the holder of the copyright, the University of Washington, changed the license so that even if the source code was still available, they did not allow modifications and changes to Pine to be distributed by anyone other than themselves. They also claimed that even the old license never allowed distribution of modified versions.

28: Mathematically, nano is 10^{-9} or one billionth. and pico is 10^{-12} or one trillionth. or put relatively, nano is 1000 times bigger than pico, although the size of nano binary is smaller than pico.

2.4.2 Exploring Nano

In nano, the Control key is represented by the ^symbol. The Meta or Alt key is represented by the `M-`.

File Handling

You can open a file in nano by providing the filename as an argument to the nano executable.

```
$ nano test.txt
```

Key	Description
`^S`	save the file
`^O`	save the file with a new name
`^X`	exit nano

Table 2.19: File Handling Commands in Nano

Editing

Nano is a simple editor, and you can do without learning any more commands than the ones listed above, but here are some more basic commands for editing text.

Key	Description
`^K`	cut current line and save in cutbuffer
`M-6`	copy current line and save in cutbuffer
`^U`	paste contents of cutbuffer
`M-T`	cut until end of buffer
`^]`	complete current word
`M-U`	undo last action
`M-E`	redo last undone action
`^J`	justify the current paragraph
`M-J`	justify the entire file
`M-:`	start/stop recording a macro
`M-;`	run the last recorded macro
`F12`	invoke the spell checker, if available

Table 2.20: Editing Commands in Nano

There are many more commands in nano, but they are omitted from here for brevity. You can find the complete list of keybindings by pressing `^G` key in nano, or by running `info nano`.

You can also find third-party cheat sheets online.

2.4.3 Editing A Script in Nano

Since learning nano is mostly to be able to edit a text file even if you are not familiar with either vim or emacs, let us try to edit a simple script file to confirm that you can use nano.

```
$ touch myscript.sh
$ chmod u+x myscript.sh
$ nano myscript.sh
```

Now try to write a simple script in the file. An example script is shown below.

```
#!/bin/bash
read -rp 'What is your name? ' name
echo "Hello $name"
date=$(date "+%H:%M on a %A")
echo "Currently it is $date"
```

If you do not understand how the script works, do not worry. It will be covered in depth in later chapters.

Now save the file by pressing `^S`

Remark 2.4.2 In some systems, the `^S` key will freeze the terminal. Any key you press after this will seem to not have any effect. This is because it is interpreted as the `XOFF` and is used to lock the scrolling of the terminal. To unfreeze the terminal, press `^Q`. In such a system, you can save the file by pressing `^O` and then typing out the name of the file if not present already, and pressing Enter. To disable this behaviour, you can add the line

```
  set -ixon
```

to your `.bashrc` file.

and exit nano by pressing `^X`.

Now you can run the script by typing

```
$ ./myscript.sh
What is your name? Sayan
Hello Sayan
Currently it is 21:50 on a Tuesday
```

Now that we are able to edit a text file using text editors, we are ready to write scripts to solve problems.