Linux Administration

Dr. Bharath Manchikodi

Chapter 1

Introduction to Linux Fundamentals

1.1 History and Philosophy of Linux

Linux is the base of a Unix-like operating system, created by Linus Torvalds, a Finnish student. The development of Linux was inspired by the Unix operating system, which was developed in the 1970s at Bell Labs. Unlike Unix, which was proprietary, Linux was developed as an open-source project, meaning its source code is freely available for anyone to view, modify, and distribute.

1.1.1 Linux as a Kernel

Linux is fundamentally a **kernel**, which is the core component of an operating system. The kernel acts as a bridge between the hardware and the software, managing system resources such as the CPU, memory, and devices. It also handles communication between hardware and software, ensuring that applications can run efficiently and securely.

The Linux kernel is **monolithic**, meaning it runs as a single large process in a single address space. However, it is also modular, allowing additional functionality to be loaded and unloaded as needed through kernel modules. This design provides both performance and flexibility.

1.1.2 The Need for the Linux Kernel

The development of the Linux kernel was driven by several factors:

- **Proprietary Unix Limitations**: In the late 1980s and early 1990s, Unix was a dominant operating system, but it was proprietary and expensive. Access to Unix source code was restricted, and licensing fees were high, making it inaccessible to many individuals and organizations.
- **Minix and Its Limitations**: Minix, a small Unix-like operating system developed by Andrew S. Tanenbaum, was used as an educational tool. However, it was not designed for practical, real-world use. Linus Torvalds, a student at the University of Helsinki, found Minix inadequate for his needs, particularly for tasks requiring more advanced features and better hardware support.

- **Desire for a Free and Open Operating System**: Linus Torvalds was inspired by the GNU Project, initiated by Richard Stallman in 1983, which aimed to create a free and open-source Unix-like operating system. While the GNU Project had developed many essential tools and utilities, it lacked a fully functional kernel. Linus saw an opportunity to fill this gap by creating a new kernel.
- **Personal Project and Learning Experience**: Linus initially developed Linux as a personal project to learn about the Intel 80386 processor's capabilities. He wanted to create a kernel that could take full advantage of the processor's features, such as protected mode and multitasking.

1.1.3 The Birth of Linux

In 1991, Linus Torvalds announced his project on the comp.os.minix newsgroup, seeking feedback and collaboration. The first version of the Linux kernel (version 0.01) was released in September 1991. It was a minimal kernel that could run basic programs like Bash (the GNU Bourne Again Shell) and GCC (the GNU Compiler Collection).

Linus released the kernel under the GNU General Public License (GPL), ensuring that it would remain free and open-source. This decision aligned with the philosophy of the Free Software Foundation (FSF) and encouraged widespread collaboration and contribution from the global developer community.

1.1.4 The Philosophy of Linux

The philosophy of Linux is deeply rooted in the principles of **open-source software** and the **Unix philosophy**. Below are the key philosophical tenets:

- **Open Source**: Linux is distributed under the GNU General Public License (GPL), which guarantees users the freedom to:
  - Run the software for any purpose.
  - Study and modify the source code.
  - Distribute the original or modified versions of the software.

  This openness fosters innovation, collaboration, and transparency.
- **Modularity**: Linux follows a modular design, where the operating system is composed of small, independent components. This modularity allows users to customize the system to their specific needs by adding or removing components.
- **Everything is a File**: In Linux, almost everything (devices, directories, processes, etc.) is treated as a file. This abstraction simplifies interactions and management, as users can use the same tools and commands to manipulate different types of resources.
- **Small, Single-Purpose Tools**: Linux adheres to the Unix philosophy of "do one thing and do it well." Instead of large, monolithic programs, Linux encourages the use of small, single-purpose tools that can be combined to perform complex tasks. This approach promotes simplicity, reusability, and efficiency.
- **Community-Driven Development**: Linux development is driven by a global community of developers, contributors, and users. This collaborative model ensures rapid innovation, peer review, and continuous improvement.

1.1.5 Impact of Linux

The development of the Linux kernel marked a turning point in the history of computing. It provided a free and open-source alternative to proprietary operating systems, empowering individuals, organizations, and governments to take control of their computing environments. Today, Linux powers a wide range of systems, from embedded devices and smartphones to servers and supercomputers.

1.1.6 Conclusion

The Linux kernel was born out of a need for a free, open, and functional operating system that could overcome the limitations of proprietary Unix and Minix. Its development was driven by Linus Torvalds' curiosity and the collaborative spirit of the open-source community. The philosophy of Linux, rooted in openness, modularity, and simplicity, has made it one of the most influential and widely used operating systems in the world.

1.2 Introduction to Open Source Software

Open Source Software (OSS) refers to software whose source code is made available to the public, allowing anyone to view, modify, and distribute the code. The open-source movement promotes collaboration, transparency, and community-driven development. Below, we explore the history of open source, the GNU Project, and the connection between OSS, GNU, and the Linux kernel.

1.2.1 History of Open Source

The concept of open-source software has its roots in the early days of computing, when software was often shared freely among researchers and developers. However, the formalization of the open-source movement began in the 1980s and 1990s, driven by the need for software freedom and collaboration.

- **Early Days of Software Sharing**: In the 1950s and 1960s, software was often distributed with source code, allowing users to modify and improve it. This collaborative culture was common in academic and research institutions.
- **Rise of Proprietary Software**: In the 1970s and 1980s, software companies began to restrict access to source code, treating software as a proprietary product. This shift led to the rise of proprietary software, which was often expensive and restrictive.
- **The Free Software Movement**: In response to the rise of proprietary software, Richard Stallman launched the **Free Software Movement** in 1983. Stallman founded the **GNU Project** and the **Free Software Foundation (FSF)** to promote the development and use of free software. The term "free" refers to freedom, not price, emphasizing the user's right to run, study, modify, and distribute software.
- **The Birth of Open Source**: In 1998, the term "open source" was coined by the Open Source Initiative (OSI) to rebrand free software in a way that emphasized its practical benefits, such as collaboration and innovation, rather than its philosophical underpinnings. The OSI also established the Open Source Definition, which outlines the criteria for software to be considered open source.

1.2.2 The GNU Project

The GNU Project, launched in 1983 by Richard Stallman, is one of the most significant initiatives in the history of open-source software. GNU stands for "GNU's Not Unix," reflecting its goal to create a free and open-source Unix-like operating system.

- **Goals of the GNU Project**: The GNU Project aimed to develop a complete operating system that would be free from proprietary restrictions. This included the development of essential tools and utilities, such as compilers, text editors, and shell programs.
- **GNU Software**: The GNU Project produced a wide range of software, including:
  - **GNU Compiler Collection (GCC)**: A collection of compilers for various programming languages.
  - **GNU Emacs**: A highly extensible text editor.
  - **GNU Bash**: A command-line shell.
  - **GNU Core Utilities**: Essential utilities for file manipulation, text processing, and system management.
- **GNU General Public License (GPL)**: The GNU Project introduced the GNU General Public License (GPL), a copyleft license that ensures software remains free and open-source. The GPL grants users the freedom to use, modify, and distribute software, provided that any derivative works are also licensed under the GPL.

1.2.3 Connection Between OSS, GNU, and the Linux Kernel

The Linux kernel, GNU software, and the open-source movement are deeply interconnected, forming the foundation of modern open-source operating systems.

- **GNU and the Missing Kernel**: By the early 1990s, the GNU Project had developed most of the components needed for a complete operating system, including compilers, libraries, and utilities. However, it lacked a fully functional kernel. The GNU Project's own kernel, called GNU Hurd, was still under development and not yet ready for production use.
- **The Linux Kernel**: In 1991, Linus Torvalds developed the Linux kernel as a free and open-source alternative to proprietary Unix kernels. Linus released the kernel under the GPL, aligning with the principles of the GNU Project and the Free Software Movement.
- **GNU/Linux Operating System**: The combination of the Linux kernel and GNU software resulted in a complete, functional operating system. This system is often referred to as **GNU/Linux** to acknowledge the contributions of both the GNU Project and the Linux kernel. Distributions like Debian, Ubuntu, and Fedora are examples of GNU/Linux systems.
- **Impact of the Collaboration**: The collaboration between the Linux kernel and GNU software demonstrated the power of open-source development. It provided a free and open alternative to proprietary operating systems, empowering users and fostering innovation. Today, GNU/Linux systems are widely used in servers, desktops, embedded systems, and supercomputers.

1.2.4 Conclusion

The history of open-source software is closely tied to the GNU Project and the development of the Linux kernel. The GNU Project laid the groundwork for free and open-source software, while the Linux kernel

provided the missing piece needed to create a complete operating system. Together, they have revolutionized the software industry, promoting collaboration, transparency, and user freedom. The success of GNU/Linux systems is a testament to the power of open-source development and the enduring impact of the Free Software Movement.

1.3 What is a Linux Distribution?

A Linux distribution (often abbreviated as **distro**) is an operating system made from a software collection that is based upon the Linux kernel and, often, a package management system. A Linux distribution is a complete operating system that includes the Linux kernel, GNU tools, additional software, and a package manager to install and update software.

1.3.1 How is it Different from an OS?

The term **Operating System (OS)** refers to the core software that manages hardware resources and provides common services for computer programs. The Linux kernel alone is not a complete OS; it requires additional software to form a functional system. A Linux distribution bundles the Linux kernel with a collection of software, utilities, and tools to create a complete, user-friendly operating system. Thus, a Linux distribution is a complete OS built around the Linux kernel.

1.3.2 What is a Desktop Environment?

A **Desktop Environment (DE)** is a graphical user interface (GUI) that provides a cohesive and user-friendly way to interact with the operating system. It includes a collection of software components such as a window manager, panels, menus, icons, widgets, and applications that allow users to manage files, run programs, and configure system settings. The desktop environment is what makes Linux visually appealing and accessible to users who prefer a graphical interface over the command line.

Components of a Desktop Environment

A typical desktop environment consists of the following components:

- **Window Manager**: Manages the placement and appearance of application windows (e.g., Metacity, KWin).
- **File Manager**: Allows users to browse and manage files and directories (e.g., Nautilus, Dolphin).
- **Panel/Taskbar**: Provides quick access to applications, system settings, and notifications (e.g., GNOME Panel, KDE Plasma Panel).
- **Application Launcher**: A menu or search tool to start applications (e.g., GNOME Activities, KDE Application Menu).
- **System Tray**: Displays icons for background processes, such as network status, volume control, and battery life.
- **Settings Manager**: A graphical tool to configure system settings like display, keyboard, and mouse.
- **Default Applications**: A suite of pre-installed applications for web browsing, text editing, multimedia playback, etc.

Popular Desktop Environments

Different Linux distributions come with different desktop environments, each offering a unique look and feel. Some of the most popular desktop environments include:

- **GNOME**: A modern and minimalist desktop environment known for its simplicity and ease of use. It is the default DE for Ubuntu and Fedora.
- **KDE Plasma**: A highly customizable and feature-rich desktop environment with a sleek design. It is the default DE for Kubuntu and openSUSE.
- **XFCE**: A lightweight and fast desktop environment suitable for older hardware. It is the default DE for Xubuntu.
- **LXQt**: A lightweight and energy-efficient desktop environment, often used in distributions like Lubuntu.
- **MATE**: A fork of GNOME 2, offering a traditional desktop experience. It is the default DE for Ubuntu MATE.
- **Cinnamon**: A modern and elegant desktop environment developed by the Linux Mint team.
- **Budgie**: A sleek and user-friendly desktop environment designed for simplicity and elegance. It is the default DE for Solus.

Why is the Desktop Environment Important?

The desktop environment plays a significant role in the user experience of a Linux distribution. It determines:

- The overall look and feel of the system.
- The ease of use for beginners and advanced users.
- The performance and resource usage (e.g., lightweight DEs like XFCE are ideal for older hardware).
- The level of customization and flexibility available to the user.

1.3.3 List of Linux Distributions

There are hundreds of Linux distributions, each tailored for specific use cases or preferences. Below is an expanded list of popular Linux distributions, categorized by their primary focus:

General-Purpose Distributions

- **Debian**: A stable and versatile distribution known for its strict adherence to free software principles.
- **Ubuntu**: A user-friendly distribution based on Debian, widely used for desktops and servers.
- **Fedora**: A cutting-edge distribution sponsored by Red Hat, often used for development and experimentation.
- **openSUSE**: A stable and user-friendly distribution suitable for both desktops and servers.
- **Linux Mint**: A beginner-friendly distribution based on Ubuntu, known for its ease of use and multimedia support.
- **Manjaro**: A user-friendly distribution based on Arch Linux, offering a balance between simplicity and customization.

- **elementary OS**: A visually appealing distribution designed for simplicity and elegance, based on Ubuntu.

Enterprise and Server Distributions

- **CentOS**: A free, community-supported distribution derived from Red Hat Enterprise Linux (RHEL), commonly used for servers.
- **Red Hat Enterprise Linux (RHEL)**: A commercial distribution designed for enterprise environments, known for its stability and support.
- **Ubuntu Server**: A server-focused version of Ubuntu, optimized for cloud and data center environments.
- **SUSE Linux Enterprise**: A commercial distribution tailored for enterprise use, offering long-term support and stability.
- **Oracle Linux**: A distribution optimized for enterprise applications, compatible with RHEL.

Lightweight and Minimalist Distributions

- **Arch Linux**: A lightweight and flexible distribution designed for advanced users who prefer a do-it-yourself approach.
- **Alpine Linux**: A security-oriented, lightweight distribution often used in containers and embedded systems.
- **Puppy Linux**: A small, fast distribution designed to run on older hardware.
- **Bodhi Linux**: A lightweight distribution based on Ubuntu, featuring the Moksha desktop environment.
- **Lubuntu**: A lightweight variant of Ubuntu using the LXQt desktop environment.

Specialized Distributions

- **Kali Linux**: A specialized distribution for penetration testing and cybersecurity.
- **Parrot OS**: A security-focused distribution for penetration testing, privacy, and development.
- **Tails**: A privacy-focused distribution designed to run from a USB stick, leaving no trace on the host system.
- **Qubes OS**: A security-oriented distribution that uses virtualization to isolate applications and data.
- **Ubuntu Studio**: A multimedia-focused distribution tailored for audio, video, and graphic production.
- **Edubuntu**: A distribution designed for educational purposes, based on Ubuntu.

Rolling Release Distributions

- **Arch Linux**: A rolling release distribution that provides the latest software updates continuously.
- **Manjaro**: A user-friendly rolling release distribution based on Arch Linux.
- **openSUSE Tumbleweed**: A rolling release version of openSUSE, offering the latest software.
- **Solus**: An independent rolling release distribution designed for desktop use.

Community-Driven Distributions

- **Gentoo**: A highly customizable distribution where software is compiled from source code.
- **Slackware**: One of the oldest Linux distributions, known for its simplicity and minimalism.
- **Void Linux**: An independent distribution featuring the `XBPS` package manager and rolling releases.
- **Devuan**: A distribution based on Debian but without the `systemd` init system.

1.3.4 How Do These Distributions Differ?

Linux distributions differ in several key aspects, including:

Package Management

Each distribution uses a specific package manager to install, update, and manage software. For example:

- Debian and Ubuntu use `APT` (Advanced Package Tool).
- Fedora and CentOS use `DNF` (Dandified YUM) or `YUM`.
- Arch Linux uses `Pacman`.
- openSUSE uses `Zypper`.

Release Cycle

Distributions have different release cycles:

- **Debian**: Stable releases are infrequent but highly tested.
- **Ubuntu**: Regular releases every six months, with Long-Term Support (LTS) versions every two years.
- **Fedora**: Frequent releases (approximately every six months) with cutting-edge features.
- **CentOS**: Focuses on stability and long-term support, with fewer updates.

Target Audience

Different distributions cater to different user groups:

- **Ubuntu** and **Linux Mint**: Aimed at beginners and general users.
- **Fedora**: Geared towards developers and enthusiasts.
- **CentOS** and **openSUSE**: Designed for enterprise and server environments.
- **Kali Linux**: Specialized for cybersecurity professionals.

Software Repositories

Distributions maintain their own repositories of software packages. The availability and freshness of software can vary:

- **Debian**: Focuses on free software, with a large repository of stable packages.
- **Arch Linux**: Provides rolling releases, meaning software is always up-to-date.
- **Ubuntu**: Offers a balance between stability and newer software.

Default Desktop Environment

A Desktop Environment (DE) is a critical component of the Linux operating system that provides the graphical interface through which users interact with their systems. It encompasses not just the visual elements like icons and taskbars but also includes system management tools, application frameworks, and user interaction layers.

Each DE offers unique features and can significantly impact the user experience. For example:

- **Unity** (used in Ubuntu) presents applications in a centralised manner with an emphasis on simplicity.
- **GNOME** offers flexibility in panel configurations and uses specific tools like GNOME Shell for customization.
- **KDE** provides more traditional Windows-like interfaces along with advanced customisation options through KDE Plasma.

These environments manage application windows, system resources, and user interactions, often integrating with underlying Linux distributions' package management systems. This integration allows DEs to present applications effectively while relying on the distribution's core functionalities for installation and management.

By comparing different DEs, users can better understand their roles and how they contribute to the overall Linux experience.

Community and Support

The size and activity of the community vary across distributions:

- **Ubuntu**: Has a large, active community and extensive documentation.
- **Arch Linux**: Relies on community-driven documentation (the Arch Wiki) and forums.
- **CentOS**: Supported by a strong enterprise community.

1.4 Linux Architecture and Anatomy

The Linux operating system is a modular and layered system, designed to be flexible, efficient, and highly customizable. It provides a wide range of choices for users, depending on their needs and preferences. The

Linux architecture can be divided into several key components, each serving a specific purpose. Below is an overview of the basic architecture and anatomy of Linux.

The architecture of the Linux operating system is modular and consists of several key components that work together to manage hardware and software resources. The primary components include:

- **Kernel**: The core component that manages system resources, including CPU, memory, and device management. It provides system calls for applications to interact with hardware.
- **Device Drivers**: These are modules that allow the kernel to communicate with hardware devices, translating OS commands into hardware-specific instructions.
- **System Libraries**: Pre-written code modules that applications use to perform common functions without needing direct access to the kernel.
- **Shell**: A command-line interface that allows users to interact with the system by executing commands and scripts.
- **Applications**: Software programs that run on top of the Linux operating system, providing various functionalities to users.

The Linux architecture is designed to be flexible, scalable, and customizable, allowing it to run on a wide range of hardware platforms.

1.4.1 Comparison with Mac and Windows Architectures

The architectures of Linux, MacOS, and Windows differ significantly in their design philosophy, components, and user interaction. Below is a comparison highlighting these differences:

Feature	**Linux**	**MacOS**	**Windows**
Kernel Type	Monolithic kernel	Hybrid kernel	Hybrid kernel
User Interface	GUI and Shell	GUI (Aqua)	GUI (Windows Shell)
Customizability	Highly customizable	Limited customization	Moderate customization
Package Management	APT, RPM, etc.	Homebrew, MacPorts	Windows Installer
Open Source	Yes	No	No
Target Audience	Developers, Servers	General users, Creatives	General users, Enterprises

Table 1.1: Comparison of Linux, MacOS, and Windows Architectures

1.4.2 Key Differences

- **Kernel Type**: Linux uses a monolithic kernel which includes all core functionalities in one large executable. MacOS employs a hybrid kernel combining features of both monolithic and microkernel architectures. Windows primarily utilizes a microkernel architecture for its core functionalities.
- **User Interface**: Linux provides a command-line interface via shells like Bash, Fish, Zsh etc. Advanced graphical interfaces such as KDE Plasma, and GNOME are also available for use on Linux. MacOS features a graphical user interface (GUI) known as Aqua. Windows also offers a GUI but has its own shell environment for command-line operations.
- **Customizability**: Linux is known for its high level of customizability due to its open-source nature. In contrast, MacOS offers limited customization options while Windows provides moderate flexibility.

- **Package Management**: Linux distributions use various package managers like APT or RPM for software installation. MacOS users can utilize Homebrew or MacPorts, while Windows typically relies on executable installers.

This comparison illustrates the distinct design philosophies and user experiences offered by each operating system.

1.4.3 Key Components of Linux Architecture

Kernel

The kernel is the core of the Linux operating system. It acts as an intermediary between the hardware and the software, managing system resources and providing essential services. Key responsibilities of the kernel include:

- **Process Management**: Creating, scheduling, and terminating processes.
- **Memory Management**: Allocating and deallocating memory for processes.
- **Device Management**: Interfacing with hardware devices via device drivers.
- **File System Management**: Handling file operations and storage.
- **Networking**: Managing network communication and protocols.

Device Drivers

Device drivers are essential modules that enable the Linux kernel to communicate with hardware devices. They act as translators, converting high-level operating system commands into hardware-specific instructions that the device can understand. Without device drivers, the kernel would be unable to interact with hardware components such as printers, keyboards, network cards, and storage devices.

Role of Device Drivers Device drivers serve as an abstraction layer between the kernel and hardware, providing the following functions:

- **Initialization**: Configuring and preparing the hardware for use.
- **Communication**: Sending and receiving data to and from the device. **Error Handling**: Detecting and managing hardware errors.
- **Power Management**: Controlling power states (e.g., sleep, wake) for energy efficiency.

Loadable Kernel Modules (LKMs) Device drivers in Linux are often implemented as Loadable Kernel Modules (LKMs). These modules can be dynamically loaded and unloaded into the kernel without requiring a system reboot. Key features of LKMs include:

- **Modularity**: Drivers can be added or removed as needed, reducing kernel bloat.
- **Flexibility**: Supports a wide range of hardware without requiring a custom kernel.
- **Ease of Development**: Developers can write and test drivers independently of the main kernel.

Types of Device Drivers Linux supports several types of device drivers, categorized based on the hardware they manage:

- **Character Device Drivers**:
  - Handle devices that process data character by character (e.g., keyboards, mice, serial ports).
  - Accessed through device files in `/dev` (e.g., `/dev/ttyS0` for a serial port).
- **Block Device Drivers**:
  - Manage devices that process data in blocks (e.g., hard drives, SSDs, USB drives).
  - Accessed through device files in `/dev` (e.g., `/dev/sda` for a hard disk).
- **Network Device Drivers**:
  - Handle network interfaces (e.g., Ethernet cards, Wi-Fi adapters).
  - Do not use device files; instead, they are managed through network interfaces (e.g., `eth0`, `wlan0`).
- **Other Drivers**:
  - Include drivers for specialized hardware (e.g., graphics cards, sound cards, printers).

Shell

The shell is a command-line interface (CLI) that allows users to interact with the kernel and execute commands. Common shells include:

- `bash` (Bourne Again Shell): The default shell on most Linux distributions.
- `sh` (Bourne Shell): A simpler, older shell.
- `zsh` (Z Shell): An extended version of `bash` with additional features.
- `ksh` (Korn Shell): A powerful scripting shell.

The shell interprets user commands, executes scripts, and manages input/output redirection.

System Libraries

System libraries provide pre-built functions that applications can use to interact with the kernel and perform common tasks. Key libraries include:

- `glibc` (GNU C Library): Provides standard C functions (e.g., memory allocation, string manipulation).
- `libc`: A smaller alternative to `glibc`.
- `libm`: Math library for mathematical functions.

These libraries abstract low-level system calls, making it easier to develop applications.

System Utilities

System utilities are programs that perform essential system management tasks. Examples include:

- `ls`: Lists directory contents.
- `cp`: Copies files and directories.
- `mv`: Moves or renames files and directories.
- `ps`: Displays information about running processes.
- `top`: Monitors system processes in real-time.

These utilities are typically part of the GNU Core Utilities package.

User Applications

User applications are programs that provide functionality to end-users. Examples include:

- Web browsers (e.g., Firefox, Chrome).
- Text editors (e.g., Vim, Nano, Gedit).
- Office suites (e.g., LibreOffice).
- Media players (e.g., VLC).

These applications rely on the underlying system libraries and utilities to function.

1.4.4 Layers of the Linux OS

The Linux operating system can be visualized as a series of layers, each building on the one below it:

1. **Hardware Layer**: The physical components of the system (e.g., CPU, memory, storage, peripherals).
2. **Kernel Layer**: Manages hardware resources and provides core services.
3. **System Libraries Layer**: Provides standardized functions for applications.
4. **Shell Layer**: Offers a user interface for interacting with the system.
5. **Application Layer**: Includes user-facing software and tools.

1.4.5 Interaction Between Components

The components of the Linux OS interact as follows:

- User applications make requests to the system libraries.
- System libraries translate these requests into system calls.
- The kernel executes the system calls and interacts with the hardware.
- The results are returned to the user application through the same chain.

1.4.6 What is a Kernel?

The kernel is the core component of an operating system, responsible for managing system resources and facilitating communication between hardware and software. There are several types of kernels, each with its own architecture and functionality. Below are the main types of kernels:

1.4.7 Types of Kernels

Monolithic Kernel

A monolithic kernel is a single large module that contains all the essential services of the operating system, including device drivers, file systems, and system calls. All components run in kernel space.

Advantages:

- Faster execution due to direct communication between components.
- Simplicity in design as all services are integrated into one module.

Disadvantages:

- A failure in one component can crash the entire system.
- Difficult to manage due to its large size.

Examples: Linux, FreeBSD.

The Linux kernel is monolithic, meaning it runs entirely in kernel space, but it supports modularity through loadable kernel modules (LKMs).

Microkernel

A microkernel includes only the most essential components necessary for system operation, such as inter-process communication (IPC) and basic scheduling. Other services run as user-mode processes.

Advantages:

- Improved stability; if one component fails, it does not affect the entire system.
- Easier to maintain and extend due to its modularity.

Disadvantages:

- Potentially slower performance due to increased context switching and IPC overhead.

Examples: QNX, MINIX.

Hybrid Kernel

A hybrid kernel combines elements of both monolithic and microkernel architectures. It includes some services in kernel space for performance while maintaining modularity.

Advantages:

- Balances performance with modularity.
- Allows for faster execution of certain services while maintaining flexibility.

Examples: Microsoft Windows NT, macOS.

Nanokernel

A nanokernel is an extremely minimalistic kernel that provides only the most basic functions needed for an operating system to run, with all other functionalities implemented as user-level processes.

Advantages:

- Very small footprint leading to high performance.
- High modularity allows for customization.

Examples: L4.

Exokernel

An exokernel separates resource protection from management, allowing applications to customize their resource usage directly. This approach enables application-specific optimizations.

Advantages:

- Greater flexibility for applications to manage resources as needed.
- Allows for high performance through application-specific customizations.

Examples: MIT Exokernel project.

Conclusion

Linux distributions provide a wide range of choices for users, depending on their needs and preferences. Whether you are a beginner, a developer, or a system administrator, there is a Linux distribution tailored for you. Understanding the differences between distributions, including their desktop environments, helps in selecting the right one for your use case.

1.5 Introduction to the Command Line Interface

The Command Line Interface (CLI) is a text-based interface used to interact with the operating system. It allows users to execute commands, manage files, and perform system administration tasks.

1.5.1 Advantages of CLI

- **Efficiency**: CLI commands can be faster and more efficient than graphical interfaces for certain tasks.
- **Automation**: CLI commands can be scripted to automate repetitive tasks.
- **Remote Access**: CLI is essential for managing remote servers via SSH.

1.6 Basic Navigation in the Linux Terminal

The terminal is the application used to access the CLI. Below are some basic commands for navigating the Linux file system:

1.6.1 Common Commands

1. `ls` - List Directory Contents

Syntax:

```
ls [options] [directory]
```

Simple Usage:

- List files and directories in the current directory:

```
    ls

```

- List files in a specific directory:

```
    ls /path/to/directory

```

Advanced Usage:

- List files in long format:

```
    ls -l

```

- List all files, including hidden ones:

```
    ls -a

```

- Sort files by modification time:

```
ls -lt
```

- Display file sizes in human-readable format:

```
ls -lh
```

- Display colors

```
ls --color=always
```

- Sort directories (folders) first

```
ls --group-directories-first
```

2. `cd` - Change Directory

Syntax:

```
cd [directory]
```

Simple Usage:

- Change to a specific directory:

```
cd /path/to/directory
```

- Go to the home directory:

```
cd
```

or

```
cd ~
```

Advanced Usage:

- Go to the previous directory:

```
cd -
```

- Move up one directory level (go to parent directory):

```
cd ..
```

- Move up two directory levels:

```
cd ../..
```

3. `pwd` - Print Working Directory

Syntax:

```
pwd
```

Simple Usage:

- Display the current directory path:

```
    pwd

```

4. `touch` - Create Empty Files

Syntax:

```
touch [filename]
```

Simple Usage:

- Create a single empty file:

```
    touch file.txt

```

Advanced Usage:

- Create multiple files at once:

```
    touch file1.txt file2.txt file3.txt

```

- Update the access and modification times of an existing file:

```
    touch existingfile.txt

```

5. `mkdir` - Make Directory

Syntax:

```
mkdir [options] [directory_name]
```

Simple Usage:

- Create a single directory:

```
    mkdir myfolder

```

Advanced Usage:

- Create nested directories:

```
    mkdir -p parent/child/grandchild

```

- Set permissions while creating a directory:

```
    mkdir -m 755 myfolder

```

6. `rmdir` - Remove Directory

Syntax:

```
rmdir [options] [directory_name]
```

Simple Usage:

- Remove an empty directory:

```
    rmdir myfolder

```

Advanced Usage:

- Remove multiple empty directories:

```
    rmdir folder1 folder2 folder3

```

- Remove nested empty directories:

```
    rmdir -p parent/child/grandchild

```

7. `rm` - Remove Files or Directories

Syntax:

```
rm [options] [file/directory]
```

Simple Usage:

- Remove a file:

```
    rm file.txt

```

Advanced Usage:

- Remove a directory and its contents recursively:

```
    rm -r myfolder

```

- Forcefully remove files without confirmation:

```
    rm -f file.txt

```

- Remove files interactively:

```
    rm -i file.txt

```

- Remove files and directories recursively and forcefully:

```
    rm -rf myfolder

```

8. `mv` - Move or Rename Files/Directories

Syntax:

```
mv [options] [source] [destination]
```

Simple Usage:

- Move a file to another directory:

```
    mv file.txt /path/to/destination

```

- Rename a file:

```
    mv oldname.txt newname.txt

```

Advanced Usage:

- Move multiple files to a directory:

```
    mv file1.txt file2.txt /path/to/destination

```

- Overwrite files without prompting:

```
    mv -f file.txt /path/to/destination

```

- Backup existing files before overwriting:

```
    mv -b file.txt /path/to/destination

```

9. cp - Copy Files/Directories

Syntax:

```
cp [options] [source] [destination]
```

Simple Usage:

- Copy a file to another location:

```
    cp file.txt /path/to/destination

```

Advanced Usage:

- Copy a directory recursively:

```
    cp -r myfolder /path/to/destination

```

- Preserve file attributes:

```
    cp -p file.txt /path/to/destination

```

- Prompt before overwriting files:

```
    cp -i file.txt /path/to/destination

```

- Copy multiple files to a directory:

```
    cp file1.txt file2.txt /path/to/destination

```

Most of the commands given here have more options than can be listed. These options can be access by passing `--help` option to the command. For example, to view the options supported by the textttmkdir command,

```
mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx - umask
  -p, --parents     no error if existing, make parent directories as needed,
                    with their file modes unaffected by any -m option
  -v, --verbose     print a message for each created directory
  -Z                   set SELinux security context of each created directory
                         to the default type
      --context[=CTX]  like -Z, or if CTX is specified then set the SELinux
                         or SMACK security context to CTX
      --help        display this help and exit
      --version     output version information and exit

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/mkdir>
or available locally via: info '(coreutils) mkdir invocation'
```

1.7 Conclusion

This unit provides a foundational understanding of Linux, its history, philosophy, and basic commands. Mastery of these concepts is essential for effective Linux administration and will serve as the basis for more advanced topics in subsequent units.

Chapter 2

Linux System Structure and Shell Basics

2.1 Logging In and Logging Out

2.1.1 Logging In from TTY

Linux systems provide multiple virtual terminals (TTYs) for text-based login. These can be accessed using keyboard shortcuts like `Ctrl+Alt+F1` to `Ctrl+Alt+F7`.

Steps to Log In

1. Switch to a TTY using `Ctrl+Alt+F1` (or any other function key up to `F7`).
2. The system displays a login prompt:

```
Ubuntu 22.04 LTS tty1
tty1 login:

```

3. Enter your username and press `Enter`.
4. Enter your password when prompted. Note that no characters (not even asterisks or dots) are displayed for security reasons.
5. If the credentials are correct, the system starts your login shell (e.g., `bash`).

Authentication

The login process involves the following system files:

- `/etc/passwd`:
  - Stores user account information in a colon-separated format:

```
username:x:1000:1000:User Name,,,:/home/username:/bin/bash

```

  - Fields: Username, Password placeholder (`x`), UID, GID, GECOS (user info), Home directory, Default shell.

- `/etc/shadow`:
  - Stores encrypted passwords and password-related settings:

```
username:$6$randomsalt$hashedpassword:19185:0:99999:7:::

```

  - Fields: Username, Encrypted password, Last password change, Minimum password age, Maximum password age, Warning period, Inactivity period, Expiration date.

- `/etc/shells`:
  - Lists valid login shells available on the system:

```
/bin/sh
/bin/bash
/usr/bin/zsh

```

Why No Echo for Passwords?

When entering a password, no characters (e.g., asterisks or dots) are displayed to prevent shoulder surfing and enhance security. This behavior is controlled by the terminal settings and the `getpass()` function in Linux.

What is a Login Shell?

A login shell is the first shell that starts when a user logs in. It differs from non-login shells in the following ways:

- Reads specific configuration files (e.g., `/etc/profile`, `~/.bash_profile`).
- Sets up the user environment (e.g., environment variables, PATH).
- Non-login shells (e.g., opening a terminal in a GUI) read different files (e.g., `~/.bashrc`).

Logging Out from TTY

To log out from a TTY:

- Type `exit` or `logout` in the shell.
- Use the keyboard shortcut `Ctrl+D`.
- To clear the TTY screen before logging out, use the `clear` command or `Ctrl+L`.

2.1.2 Logging In from GUI

Graphical login is handled by a **Display Manager** (also called a Login Manager).

Display Manager

The display manager is responsible for:

- Presenting a graphical login screen.
- Authenticating users.
- Starting the user's desktop environment (DE).

Common display managers include:

- `GDM` (GNOME Display Manager).
- `LightDM` (Lightweight Display Manager).
- `SDDM` (Simple Desktop Display Manager).

Steps to Log In

1. The display manager loads after the system boots.
2. Select your username from the list or type it manually.
3. Enter your password. Unlike TTY, some display managers show asterisks or dots for feedback.
4. Choose your desktop environment (if multiple DEs are installed).
5. The display manager starts the selected DE and loads the user session.

Other Relevant Information

- Session Management: The display manager remembers the last session and can restore it.
- Autologin: Configured in display manager settings to bypass the login screen.
- Lock Screen: Activated after inactivity or manually (e.g., `Ctrl+Alt+L`).

Logging Out from GUI

To log out from a GUI session (e.g., Ubuntu):

- Click the system menu (top-right corner).
- Select the power icon.
- Choose `Log Out` or `Sign Out`.
- The display manager will return to the login screen.

Alternatively, use keyboard shortcuts like `Ctrl+Alt+Del` (if configured) to log out.

2.2 Directory Structure and the /usr Directory

The Linux directory structure is hierarchical and starts from the root directory, denoted by `/`. This is analogous to the `C:\` drive in Windows or the "My Computer" concept, where all files and directories are organized under a single root.

2.2.1 Root Folder and Base Directories

The root folder (`/`) is the top-level directory in Linux. It contains several essential directories that form the foundation of the Linux filesystem:

- `/bin`: Contains essential command binaries (e.g., `ls`, `cp`, `mv`) that are required for system operation, even in single-user mode.
- `/boot`: Stores bootloader files, including the kernel and initial RAM disk (initrd).
- `/dev`: Contains device files that represent hardware devices (e.g., `/dev/sda` for a disk, `/dev/tty` for a terminal).
- `/etc`: Holds system-wide configuration files (e.g., `/etc/passwd`, `/etc/fstab`).
- `/home`: Contains personal directories for users (e.g., `/home/username`).
- `/lib`: Stores essential shared libraries needed by binaries in `/bin` and `/sbin`.
- `/sbin`: Contains system administration binaries (e.g., `init`, `iptables`).
- `/sys`: Provides a virtual filesystem for kernel and device information.
- `/tmp`: Used for temporary files, which are often deleted on reboot.
- `/usr`: Contains user-installed software and libraries (discussed in detail below).
- `/var`: Stores variable data files, such as logs (`/var/log`), databases, and spool files.

2.2.2 Other Important Directories

In addition to the base directories, Linux includes several other directories for specific purposes:

- `/run`: Stores runtime data for processes (e.g., PID files, sockets) that are created during system operation.
- `/snap`: Used by Ubuntu to store Snap packages, a universal package format for Linux.
- `/proc`: A virtual filesystem that provides information about running processes and kernel parameters (e.g., `/proc/cpuinfo`, `/proc/meminfo`).
- `/opt`: Reserved for optional third-party software. Each application typically installs in its own subdirectory (e.g., `/opt/google/chrome`).
- `/srv`: Contains data for services provided by the system (e.g., web server files in `/srv/www`).

2.2.3 Historical Perspective: /bin vs /usr/bin

Historically, Linux systems separated essential binaries into `/bin` and `/sbin`, while user-installed binaries were placed in `/usr/bin` and `/usr/sbin`. This separation was due to the limited storage capacity of early systems, where `/usr` was often mounted on a separate disk partition.

However, modern systems have adopted the **usr-merge** approach, where:

- `/bin` and `/usr/bin` are merged into `/usr/bin`.
- `/sbin` and `/usr/sbin` are merged into `/usr/sbin`.
- Symbolic links are created to maintain compatibility (e.g., `/bin` points to `/usr/bin`).
  Use `ls -l /`, and verify that `/bin` indeed points to `/usr/bin`.

This change simplifies the directory structure and reduces redundancy.

2.2.4 Structure of /usr

The `/usr` directory is one of the most important directories in Linux. It contains user-installed software and libraries, organized as follows:

- `/usr/bin`: Non-essential command binaries (e.g., `gcc`, `python`).
- `/usr/lib`: Libraries for user applications.
- `/usr/share`: Architecture-independent data (e.g., documentation, icons, fonts).
- `/usr/include`: Header files for C/C++ programming.
- `/usr/src`: Source code for installed software.
- `/usr/local`: Reserved for software installed locally by the system administrator (discussed below).

2.2.5 /usr/local vs /opt

Both `/usr/local` and `/opt` are used for locally installed software, but they serve different purposes:

- `/usr/local`:
  - Intended for software compiled and installed locally by the system administrator.
  - Follows the same structure as `/usr` (e.g., `/usr/local/bin`, `/usr/local/lib`).
  - Typically used for software that is not part of the distribution's package manager.
- `/opt`:
  - Used for third-party software that is self-contained and does not follow the standard directory structure.
  - Each application is installed in its own subdirectory (e.g., `/opt/google/chrome`).
  - Often used for proprietary software or large applications.

2.2.6 Virtual Filesystems

Linux includes several virtual filesystems that provide access to kernel and system information:

- `/proc`: A virtual filesystem that exposes kernel and process information as files (e.g., `/proc/cpuinfo`, `/proc/meminfo`).

- `/sys`: A virtual filesystem that provides a hierarchical view of kernel objects and device information.
- `/dev`: Contains device files that represent hardware devices (e.g., `/dev/sda`, `/dev/tty`).

These filesystems do not reside on disk but are generated dynamically by the kernel.

2.3 File Types

2.3.1 Everything is a File

In Linux, the philosophy of "everything is a file" is a fundamental concept. This means that almost everything, including hardware devices, processes, and directories, is represented as a file. This abstraction simplifies interactions with the system, as the same tools and interfaces (e.g., `cat`, `ls`, `echo`) can be used to manipulate files, devices, and other resources.

2.3.2 Directories as Files

Even directories are treated as files in Linux. A directory is essentially a special type of file that contains a list of filenames and their corresponding inode numbers. Key points about directories:

- Directories can be read like files (e.g., using functions like `readdir`).
- They have permissions (read, write, execute) just like regular files.
- The execute permission on a directory allows accessing files within it.

2.3.3 File Types

Linux supports several types of files, each serving a specific purpose. Below are the main categories:

Read-Only Files

Read-only files are typically system files that users are not allowed to modify. Examples include:

- Configuration files in `/etc` (e.g., `/etc/passwd`, `/etc/fstab`).
- Documentation files in `/usr/share/doc`.
- Kernel and boot files in `/boot`.
- Libraries in `/lib` and `/usr/lib`.

These files are critical for system operation, and modifying them can lead to instability or security issues. These files are typically modified (i.e., replaced) when one or more packages of the system in upgraded.

Read-Write Files

Read-write files are user-created files that can be modified. Examples include:

- Personal documents in `/home/username`.
- Temporary files in `/tmp`.
- Log files in `/var/log` (some logs are append-only).

These files are typically owned by the user and have read and write permissions.

Executable Files

Executable files are files that can be executed as programs. They fall into two main categories:

- **Binary Executables**:
  - Compiled programs (e.g., `/bin/ls`, `/usr/bin/gcc`).
  - Typically have no file extension.
- **Scripts**:
  - Text files containing commands (e.g., shell scripts, Python scripts).
  - Require an interpreter (e.g., `/bin/bash`, `/usr/bin/python3`).
  - Must have execute permissions to run.

User-executable files are often stored in `/usr/local/bin` or ∼`/bin`.

Symbolic and Hard Links

Links are special files that point to other files or directories. There are two types:

- **Symbolic Links (Soft Links)**:
  - Point to another file or directory by name.
  - Created using `ln -s target link`.
  - Example: `ln -s /usr/bin/python3 /usr/local/bin/python`.
  - If the target is deleted, the link becomes "dangling."
- **Hard Links**:
  - Point to the same inode as the target file.
  - Created using `ln target link`.
  - Example: `ln file1 file2`.
  - Hard links cannot span filesystems or point to directories.

Device Files

Device files in `/dev` represent hardware devices and other system resources. They are categorized as:

- **Character Devices**:
  - Accessed character by character (e.g., keyboards, terminals).
  - Example: `/dev/tty` (terminal), `/dev/null` (null device).
- **Block Devices**:
  - Accessed in blocks (e.g., hard drives, USB drives).
  - Example: `/dev/sda` (first hard disk).
- **Special Devices**:
  - `/dev/stdin`: Standard input.
  - `/dev/stdout`: Standard output.
  - `/dev/stderr`: Standard error.
  - These are symbolic links to the current terminal's input/output streams.

2.3.4 Summary

The "everything is a file" philosophy in Linux provides a unified interface for interacting with the system. Whether it's a regular file, directory, executable, link, or device, the same tools and principles apply. This abstraction is one of the key reasons for Linux's flexibility and power.

2.4 Naming Files and Directories

- **Case Sensitivity**: File and directory names are case-sensitive (e.g., `file.txt` and `File.txt` are different).
- **Special Characters**: Avoid using special characters like `/`, `*`, `?`, and spaces. Use underscores (`_`) or hyphens (`-`) instead.
- **Hidden Files**: Files and folders starting with a dot (`.`) are hidden (e.g., `.bashrc`).
- **Descriptive Names**: Use meaningful names for better organization (e.g., `project_report_2023.pdf`).

2.5 Spawning Processes

A process is an instance of a running program. This section covers how processes are created, managed, and controlled from the shell. Key concepts and commands are explained below.

2.5.1 Process Creation

- `fork()`:

  - The `fork()` system call creates a new process by duplicating the existing process. The new process is called the *child process*, and the original process is called the *parent process*.
  - After `fork()`, both the parent and child processes execute the same code but have separate memory spaces.

- `exec()` **Family of Functions**:
  - The `exec()` family of functions (e.g., `execl()`, `execvp()`, `execv()`) replaces the current process image with a new program.
  - Unlike `fork()`, `exec()` does not create a new process; it overlays the current process with the new program.
  - Example: Running `exec ls -l` in a shell replaces the shell process with the `ls` command.

2.5.2 Foreground vs. Background Processes

- **Foreground Processes**:
  - Foreground processes require user interaction and block the terminal until they complete.
  - Example: Running `ls -l` in the terminal executes it in the foreground.

- **Background Processes**:
  - Background processes run independently of the terminal and do not block user input.
  - To run a process in the background, append an ampersand (&) to the command (e.g., `sleep 100 &`).
  - The shell displays the process ID (PID) of the background process.

2.5.3 Process Management

- `ps`:
  - The `ps` command displays information about running processes.
  - Common usage: `ps aux` (shows all processes for all users).

- `top`:
  - The `top` command provides a real-time view of system processes, including CPU and memory usage.

- `kill`:
  - The `kill` command terminates a process by sending a signal to its PID.
  - Example: `kill -9 PID` forcefully terminates the process with the specified PID.

2.5.4 Controlling Processes in the Shell

- `Ctrl-Z`:
  - Pressing `Ctrl-Z` suspends the currently running foreground process and returns control to the shell.
  - The suspended process is placed in the background and stops execution.

- **Resuming a Suspended Process**:

  - Use the `fg` command to bring a suspended process back to the foreground.
  - Use the `bg` command to resume a suspended process in the background.

- `fg`:
  - The `fg` command resumes the most recently suspended process in the foreground.
  - Example: `fg %1` resumes job number 1 in the foreground.

- `bg`:
  - The `bg` command resumes a suspended process in the background.
  - Example: `bg %1` resumes job number 1 in the background.

- `disown`:
  - The `disown` command removes a background job from the shell's job table, allowing it to continue running even if the shell is closed.
  - Example: `disown %1` removes job number 1 from the shell's job table.

2.6 Shell Basics

The shell is a command-line interpreter that allows users to interact with the OS.

2.6.1 Creating User Accounts

User accounts can be created using the `useradd` command:

```
sudo useradd -m -s /bin/bash username
sudo passwd username
```

- `-m`: Create a home directory.
- `-s`: Specify the default shell.

2.6.2 Writing Shell Programs

Shell scripts automate tasks using a series of commands. Example:

```
#!/bin/bash
# This is a comment
echo "Hello, World!"
```

Make the script executable:

```
chmod +x script.sh
```

2.6.3 Working with the bash Shell

The bash shell is the default shell in most Linux distributions. Key features:

- **Command History**: Use `history` to view previous commands.
- **Tab Completion**: Automatically complete commands and filenames.
- **Environment Variables**: Store system-wide settings (e.g., `$PATH`).
- **Aliases**: Create shortcuts for commands (e.g., `alias ll='ls -la'`).

2.7 Changing the Bash Prompt

The Bash prompt is the text displayed in the terminal to indicate that the shell is ready to accept commands. By default, it often includes information such as the username, hostname, and current working directory. However, the prompt can be customized to display additional information or to suit personal preferences.

2.7.1 Understanding the Bash Prompt

The Bash prompt is controlled by the `PS1` (Prompt String 1) environment variable. Other prompt variables include:

- `PS2`: Secondary prompt (used for multi-line commands).
- `PS3`: Prompt for the `select` command.
- `PS4`: Prompt for debugging output.

The default `PS1` value is often set in the system-wide or user-specific Bash configuration files (e.g., `/etc/bash.bashrc`, `~/.bashrc`).

2.7.2 Customizing the Bash Prompt

To customize the Bash prompt, modify the `PS1` variable. The prompt can include text, special escape sequences, and command output.

Special Escape Sequences

Bash provides special escape sequences to dynamically include information in the prompt. Common sequences include:

- `\u`: Current username.
- `\h`: Hostname (up to the first dot).
- `\H`: Full hostname.
- `\w`: Current working directory (full path).

- \W: Current working directory (basename only).
- \$: Displays $ for regular users and # for the root user.
- \t: Current time in 24-hour format (HH:MM:SS).
- \d: Current date in "Weekday Month Date" format.
- \n: Newline.
- \e: Escape character (used for colors).

Examples of Custom Prompts

Here are some examples of customizing the PS1 variable:

1. **Basic Prompt**:

```
export PS1="\u@\h:\w\$ "

```

   This displays:

```
username@hostname:/current/directory$

```

2. **Prompt with Date and Time**:

```
export PS1="[\d \t] \u@\h:\w\$ "

```

   This displays:

```
[Mon Oct 23 14:35:12] username@hostname:/current/directory$

```

3. **Multi-Line Prompt**:

```
export PS1="\n\u@\h:\w\n\$ "

```

   This displays:

```
username@hostname:/current/directory
$

```

2.7.3 Adding Colors to the Prompt

Colors can be added to the prompt using ANSI escape sequences. These sequences start with \e[and end with m. Common color codes include:

- 0: Reset all attributes.
- 31: Red text.
- 32: Green text.
- 33: Yellow text.
- 34: Blue text.

- `35`: Magenta text.
- `36`: Cyan text.
- `41`: Red background.
- `42`: Green background.

Example: Colored Prompt

To create a prompt with colored text:

```
export PS1="\[\e[32m\]\u@\h\[\e[0m\]:\[\e[34m\]\w\[\e[0m\]\$ "
```

This displays:

- Username and hostname in green.
- Current working directory in blue.
- The `$` symbol in the default color.

2.7.4 Making Changes Permanent

To make changes to the Bash prompt permanent, add the `PS1` configuration to one of the following files:

- `~/.bashrc`: User-specific configuration.
- `~/.bash_profile`: Executed for login shells.
- `/etc/bash.bashrc`: System-wide configuration.

For example, add the following line to `~/.bashrc`:

```
export PS1="\u@\h:\w\$ "
```

Then, reload the configuration:

```
source ~/.bashrc
```

Advanced Customization

For advanced customization, you can include command output in the prompt. For example, to display the number of background jobs:

```
export PS1="\u@\h:\w [\j]\$ "
```

This displays:

```
username@hostname:/current/directory [0]$
```

2.8 Managing Users on Linux

Linux provides several commands to manage users, including creating, modifying, and removing user accounts. Two commonly used commands for user management are `useradd` and `adduser`. This section discusses these commands with examples.

2.8.1 Creating Users

Using `useradd`

The `useradd` command is a low-level utility for creating user accounts. It requires additional commands to set up the user's home directory and password.

```
# Create a new user
sudo useradd username

# Set a password for the user
sudo passwd username

# Create a user with a home directory
sudo useradd -m username

# Create a user with a specific user ID (UID)
sudo useradd -u 1001 username

# Create a user with a specific group ID (GID)
sudo useradd -g 1001 username

# Create a user with a custom home directory
sudo useradd -m -d /custom/home/username username
```

Using `adduser`

The `adduser` command is a higher-level utility that simplifies the process of creating users. It interactively prompts for additional information and automatically sets up the home directory.

```
# Create a new user interactively
sudo adduser username
```

During the execution, `adduser` will prompt for the user's password, full name, and other details.

2.8.2 Modifying Users

The `usermod` command is used to modify existing user accounts. Below are some examples:

```
# Change the user's login name
sudo usermod -l newname oldname

# Add the user to additional groups
sudo usermod -aG groupname username

# Change the user's home directory
sudo usermod -d /new/home/dir username

# Lock a user account
sudo usermod -L username

# Unlock a user account
sudo usermod -U username
```

2.8.3 Removing Users

The `userdel` command is used to delete user accounts. Use the `-r` option to remove the user's home directory and mail spool.

```
# Delete a user without removing the home directory
sudo userdel username

# Delete a user and remove the home directory
sudo userdel -r username
```

2.8.4 Summary

The `useradd` command provides fine-grained control over user creation but requires additional steps to set up the user environment. In contrast, `adduser` is more user-friendly and automates many of these steps. For modifying and removing users, `usermod` and `userdel` are the primary tools, respectively. Understanding these commands is essential for effective user management on Linux systems.