

Ubuntu Academy: Using Linux Terminal

Mastering the Command Line (Revised 2026 Edition)

Onno W. Purbo
onno@indo.net.id
onno@itts.ac.id
Twitter onnowpurbo

Institute of Technology Tangerang Selatan
OnnoCenter
2026

TABLE OF CONTENT

TABLE OF CONTENT.....	2
PREFACE.....	13
PART I — TERMINAL FOUNDATIONS.....	14
Chapter 1 — Understanding the Linux Terminal Environment.....	14
1. What Is a Shell? How Bash Works.....	14
2. Terminal vs TTY vs Console.....	15
Terminal.....	15
TTY (Teletypewriter).....	15
Console.....	16
3. Command Anatomy (Command, Options, Arguments).....	16
Example.....	16
Options.....	16
Arguments.....	16
4. Understanding the Prompt and Privilege Levels.....	17
Normal User vs Root.....	17
5. Exit Codes and \$?.....	18
6. Built-in vs External Commands.....	19
Identify command type.....	19
Which Command.....	19
7. Getting Help in Linux.....	20
man (Manual Pages).....	20
info.....	20
--help.....	20
whatis.....	21
apropos.....	21
/usr/share/doc.....	21
Practical Mini Lab.....	21
Conceptual Topology: How Everything Connects.....	21
Chapter Summary.....	22
PART II — NAVIGATING FILES AND FILESYSTEMS (~34%).....	23
Chapter 2 — Linux Filesystem Hierarchy (FHS).....	23
1. The Root of Everything: /.....	23
2. Key Directories in the Linux Filesystem Hierarchy.....	23
/etc — System Configuration.....	23
/var — Variable Data.....	24
/usr — User Programs and Applications.....	25
/home — User Directories.....	25
/root — The Root User's Home.....	26
/tmp — Temporary Files.....	26
3. Absolute vs Relative Paths.....	26
Absolute Paths.....	26
Relative Paths.....	27

Special Shortcuts.....	27
4. Navigating with pwd, cd, pushd, popd.....	27
pwd — Print Working Directory.....	27
cd — Change Directory.....	27
pushd and popd — Directory Stack Navigation.....	28
5. Advanced ls Usage.....	28
Common Useful Options.....	28
Understanding Long Format Output.....	28
6. stat — Detailed File Information.....	29
7. tree — Visual Directory Structure.....	29
8. Introduction to Inodes.....	30
Conceptual Illustration: Filesystem Architecture.....	31
Practical Lab.....	31
Chapter Summary.....	32
Chapter 3 — File and Directory Management.....	33
1. touch — Creating and Updating Files.....	33
Create a new file.....	33
Why this matters.....	34
2. cp — Copying Files and Directories.....	34
Basic copy.....	34
Recursive copy.....	34
Preserve attributes.....	34
Interactive copy (safe practice).....	34
Professional Tip.....	35
3. mv — Moving and Renaming Files.....	35
Rename a file.....	35
Move to another directory.....	35
Interactive mode.....	35
Important Behavior.....	35
4. rm — Removing Files (Safe Practices).....	35
Remove a file.....	35
Remove a directory recursively.....	35
Force removal.....	35
Safe Practices.....	36
5. mkdir -p — Creating Directories.....	36
6. rmdir — Removing Empty Directories.....	36
7. ln — Hard Links and Symbolic Links.....	37
Hard Links.....	37
Symbolic Links (Soft Links).....	37
Conceptual Illustration.....	38
8. file — Identifying File Types.....	38
9. namei — Path Resolution.....	39
10. umask — Default Permissions.....	39
Example.....	39

11. Link Count and Inode Behavior.....	40
Practical Lab.....	40
Conceptual Topology of File Operations.....	41
Chapter Summary.....	41
Chapter 4 — Searching and Filtering Data.....	42
1. find — The Most Powerful File Search Tool.....	42
Search by Name.....	42
Search by Type.....	42
Search by Size.....	43
Search by Time.....	43
Combine Conditions.....	43
Execute Action on Results.....	43
2. locate and updatedb.....	44
3. which and whereis.....	44
which.....	44
whereis.....	45
4. grep — Searching Inside Files.....	45
Recursive Search.....	45
Case-Insensitive Search.....	45
Invert Match.....	45
Line Numbers.....	45
5. Basic Regular Expressions.....	46
Anchors.....	46
Wildcards.....	46
6. diff vs cmp.....	47
diff — Human-Readable Differences.....	47
cmp — Byte-by-Byte Comparison.....	47
7. Introduction to Pipeline Thinking.....	47
Basic Pipeline Example.....	47
Counting Results.....	47
Advanced Example.....	48
Visualization of Pipeline Flow.....	48
Practical Lab.....	48
Conceptual Topology of Data Filtering.....	49
Chapter Summary.....	49
PART III — TEXT PROCESSING & PIPELINES.....	50
Chapter 5 — Working with Text Files.....	50
1. cat and tac — Displaying File Contents.....	50
cat — Concatenate and Display.....	50
tac — Reverse Display.....	51
2. less — Viewing Large Files Safely.....	51
3. head and tail — Viewing Portions of Files.....	51
head.....	51
tail.....	52

tail -f — Real-Time Monitoring.....	52
Real-World Example.....	52
4. wc — Counting Lines, Words, and Characters.....	52
5. sort — Sorting Text.....	52
6. uniq — Removing Duplicate Lines.....	53
7. cut — Extracting Fields.....	54
8. tr — Transforming Characters.....	54
9. column — Formatting Output.....	55
10. tee — Writing Output to File and Screen.....	55
Visualizing Text Processing Flow.....	55
Practical Lab.....	56
Conceptual Illustration.....	56
Chapter Summary.....	57
Chapter 6 — Redirection, Pipes, and Shell Operators.....	58
1. Understanding Standard Streams.....	58
2. Output Redirection: > and >>.....	59
Overwrite Output (>).....	59
Append Output (>>).....	59
Practical Example.....	59
3. Error Redirection: 2> and &>.....	59
Redirect Only Errors.....	59
Redirect Both Output and Errors.....	59
Professional Use Case.....	60
4. Input Redirection: <.....	60
5. Pipes: 	60
Basic Example.....	60
Multi-Step Pipeline.....	60
Visualization.....	61
6. xargs — Converting Input into Arguments.....	61
Example Without xargs.....	61
Safer Version.....	61
Example: Batch File Processing.....	61
7. Command Substitution: \$().....	61
Modern Syntax.....	61
Practical Example.....	62
8. Quoting Rules.....	62
Double Quotes " ".....	62
Single Quotes ' '.....	62
Backslash \.....	62
9. Globbing (*, ?, []).....	63
Asterisk *.....	63
Question Mark ?.....	63
Brackets [].....	63
10. Metacharacters and Shell Expansion.....	63

Tilde Expansion.....	64
Variable Expansion.....	64
Arithmetic Expansion.....	64
Real-World Example: Building a Data Pipeline.....	64
Practical Lab.....	65
Conceptual Model.....	65
Chapter Summary.....	66
PART IV — MANAGING SYSTEM RESOURCES (~33%).....	67
Chapter 7 — Process Management.....	67
1. Viewing Processes with ps.....	67
ps aux.....	67
ps -ef.....	68
Visualizing Process Hierarchy.....	68
2. Monitoring in Real Time: top and htop.....	68
top.....	68
htop.....	69
Understanding Load Average.....	69
3. Searching for Processes with pgrep.....	70
4. Controlling Processes: kill, pkill, killall.....	70
kill.....	70
pkill.....	70
killall.....	70
5. Signals — Communicating with Processes.....	71
Send Specific Signal.....	71
Reload Service (SIGHUP).....	71
Visual Model.....	71
6. Priority Control: nice and renice.....	71
Start Process with Lower Priority.....	71
Change Priority of Running Process.....	72
7. Background Jobs.....	72
Run in Background (&).....	72
View Jobs.....	72
Bring Job to Foreground.....	72
Send Job to Background.....	72
Visual Workflow.....	72
8. nohup — Survive Terminal Logout.....	72
Real-World Scenario.....	73
Practical Lab.....	73
Conceptual System Topology.....	74
Chapter Summary.....	74
Chapter 8 — Monitoring Resources and Logs.....	75
1. Monitoring Memory Usage — free -h.....	75
Understanding Columns.....	75
2. Checking System Load — uptime.....	76

Understanding Load Average.....	76
3. Monitoring Active Users — who and w.....	76
who.....	76
w.....	76
4. Monitoring Disk Usage — df -h.....	77
5. Measuring Directory Size — du -sh.....	77
6. Viewing Mounted Filesystems — mount.....	78
7. Log Management in Linux.....	78
/var/log.....	78
8. journalctl — systemd Log Viewer.....	79
View All Logs.....	79
Show Latest Logs.....	79
Follow Logs in Real Time.....	79
View Logs for Specific Service.....	79
Filter by Time.....	79
9. Introduction to systemd Services.....	80
View Service Status.....	80
Start Service.....	80
Stop Service.....	80
Enable Service at Boot.....	80
Disable Service.....	80
10. Understanding Target Units.....	80
System Monitoring Topology.....	81
Real-World Diagnostic Workflow.....	81
Practical Lab.....	82
Chapter Summary.....	82
PART V — SECURITY & USER MANAGEMENT (~33%).....	83
Chapter 9 — Users and Groups.....	83
1. Identifying Yourself — whoami and id.....	83
whoami.....	83
id.....	83
Why This Matters.....	84
2. Creating and Managing Users.....	84
useradd — Create a New User.....	84
Setting Password — passwd.....	84
usermod — Modify User.....	84
userdel — Remove User.....	85
3. Managing Groups.....	85
groupadd.....	85
Add User to Group.....	85
gpasswd.....	85
Example Scenario.....	86
4. Understanding /etc/passwd.....	86
5. Understanding /etc/shadow.....	87

Why /etc/shadow Exists.....	87
6. User and Group Topology.....	87
7. UID and GID Ranges.....	88
8. Best Practices for User Management.....	88
Real-World Scenario.....	89
Practical Lab.....	89
Conceptual Security Model.....	89
Chapter Summary.....	90
Chapter 10 — File Permissions and Privilege Escalation.....	91
1. Understanding Permission Basics.....	91
2. chmod — Changing Permissions.....	92
Symbolic Mode.....	92
Numeric (Octal) Mode.....	92
Visual Permission Model.....	93
3. chown — Changing Ownership.....	93
4. chgrp — Changing Group.....	94
5. Special Permissions — SUID, SGID, Sticky Bit.....	94
SUID (Set User ID).....	94
SGID (Set Group ID).....	94
Sticky Bit.....	95
6. umask — Default Permission Mask.....	95
7. Privilege Escalation — sudo.....	96
Why Avoid Root Login?.....	96
8. visudo — Safely Editing sudo Configuration.....	96
Example Entry.....	97
9. Practical Secure Configuration Example.....	97
Security Topology.....	98
Real-World Scenario.....	98
Practical Lab.....	98
Chapter Summary.....	99
PART VI — PACKAGE & SYSTEM MANAGEMENT (Ubuntu Focus).....	100
Chapter 11 — Package Management in Ubuntu.....	100
1. Understanding the APT Ecosystem.....	100
2. apt update — Refresh Package Index.....	100
3. apt upgrade — Upgrade Installed Packages.....	101
Recommended Update Workflow.....	101
4. apt install — Installing Software.....	101
Install Specific Version.....	101
5. apt remove — Removing Software.....	102
6. apt autoremove — Cleaning Dependencies.....	102
7. apt search — Finding Packages.....	102
8. Understanding dpkg Basics.....	103
Install Local .deb File.....	103
List Installed Packages.....	103

Remove Package with dpkg.....	103
9. Snap Package Management.....	103
Install Snap Package.....	103
List Installed Snaps.....	103
Remove Snap.....	103
Snap vs APT.....	104
10. DEB vs RPM — Conceptual Comparison.....	104
Key Differences.....	104
11. Repository Topology.....	105
Real-World Scenario.....	105
Security Best Practices.....	106
Practical Lab.....	106
Chapter Summary.....	106
PART VII — INTRODUCTION TO SHELL SCRIPTING.....	107
Chapter 12 — Writing Simple Bash Scripts.....	107
1. The Shebang (#!/bin/bash).....	107
2. Making Scripts Executable.....	108
Script Execution Flow.....	108
3. Variables and Expansion.....	108
Assigning Variables.....	108
Accessing Variables.....	108
Example Script.....	108
Command Substitution.....	108
4. Positional Parameters.....	109
5. Conditional Statements — if/then/else.....	109
Example: Check File Existence.....	109
Common Test Operators.....	110
6. for Loops.....	110
Basic for Loop.....	110
Example: Batch Rename.....	110
Loop Topology.....	110
7. while Loops.....	110
Example.....	110
Reading File Line-by-Line.....	111
8. case Statement.....	111
Example.....	111
9. read — Taking User Input.....	112
10. Exit Codes.....	112
11. Debugging with set -x.....	113
12. Example Complete Script.....	113
Automation Topology.....	114
Real-World Example.....	114
Practical Lab.....	115
Chapter Summary.....	115

PART VIII — PRODUCTIVITY & BEST PRACTICES.....	116
Chapter 13 — Becoming Efficient in the Terminal.....	116
1. Command History.....	116
Navigate with Arrow Keys.....	116
Limit History Size.....	116
2. Reverse Search — Ctrl + R.....	117
3. History Expansion — !! and !n.....	117
Repeat Last Command — !!.....	117
Run Specific History Command — !n.....	117
Run Most Recent Matching Command.....	118
4. Tab Completion.....	118
File Completion.....	118
Command Completion.....	118
Multiple Possibilities.....	118
5. Aliases — Creating Shortcuts.....	118
Temporary Alias.....	118
Permanent Alias.....	119
List Aliases.....	119
6. Customizing ~/.bashrc.....	119
Prompt Customization Example.....	120
7. Environment Variables.....	120
View All Variables.....	120
Important Variables.....	120
PATH.....	120
EDITOR.....	120
Persistent Environment Variables.....	121
8. Troubleshooting Common Terminal Issues.....	121
Command Not Found.....	121
Permission Denied.....	121
Terminal Freeze.....	121
Accidental Output to File Instead of Screen.....	122
Broken Terminal Display.....	122
Efficiency Workflow Model.....	122
Real-World Efficiency Example.....	122
Practical Lab.....	123
Productivity Mindset.....	123
Chapter Summary.....	124
APPENDICES.....	125
Appendix A — Command Reference by Topic.....	125
1. File Management.....	125
2. File Search.....	126
3. Text Processing.....	126
4. Compression & Archiving.....	127
5. Process Management.....	127

6. User Management.....	128
7. Ubuntu Administration.....	128
Conceptual Command Topology.....	129
How to Use This Appendix.....	129
Final Note.....	129
Appendix B — Practice Labs (Exam-Style Scenarios).....	130
Lab 1 — Freeing Disk Space.....	130
Scenario.....	130
Objective.....	130
Step 1 — Verify Disk Usage.....	130
Step 2 — Identify Large Directories.....	131
Step 3 — Clear Logs (Carefully).....	131
Step 4 — Clean Package Cache.....	131
Topology of Disk Investigation.....	131
Reflection.....	132
Lab 2 — Finding Suspicious Files.....	133
Scenario.....	133
Objective.....	133
Step 1 — Search for Executables in Home.....	133
Step 2 — Search Recently Modified Files.....	133
Step 3 — Look for SUID Files.....	133
Step 4 — Inspect Hidden Files.....	133
Step 5 — Search for Suspicious Strings.....	133
Investigation Flow.....	133
Reflection.....	134
Lab 3 — Killing Runaway Processes.....	135
Scenario.....	135
Objective.....	135
Step 1 — Check Load.....	135
Step 2 — Monitor in Real Time.....	135
Step 3 — Identify Process.....	135
Step 4 — Attempt Graceful Termination.....	135
Step 5 — Force Kill if Needed.....	135
Step 6 — Prevent Recurrence.....	135
Process Control Flow.....	135
Reflection.....	136
Lab 4 — Creating Restricted Users.....	137
Scenario.....	137
Step 1 — Create User.....	137
Step 2 — Set Password.....	137
Step 3 — Assign Limited Shell.....	137
Step 4 — Remove from sudo Group.....	137
Step 5 — Restrict Directory Access.....	137
Access Control Topology.....	137

Reflection.....	137
Lab 5 — Writing a Simple Backup Script.....	138
Scenario.....	138
Objective.....	138
Step 1 — Create Script.....	138
Step 2 — Write Script.....	138
Step 3 — Make Executable.....	138
Step 4 — Run.....	138
Step 5 — Automate with Cron (Optional Advanced).....	138
Backup Flow.....	139
Reflection.....	139
Final Mastery Challenge.....	140
Closing Advice.....	140
Appendix C — Glossary.....	141
References.....	146
AUTHOR.....	147

PREFACE

The Linux terminal is not just a text interface — it is the core tool of modern system administration, DevOps, cloud computing, and cybersecurity. While graphical interfaces simplify interaction, true control of a system comes from mastering the command line.

This book is designed to transform the terminal from something intimidating into something logical, structured, and powerful. It follows a practical competency distribution aligned with real-world Linux administration:

- **Navigating Files & Filesystems (~34%)**
- **Managing System Resources (~33%)**
- **Securing the System (~33%)**

More than teaching commands, this book promotes a professional mindset:

Think in pipelines, not clicks.

Through structured explanations, hands-on labs, and real-world scenarios, you will learn not only how commands work — but why they behave the way they do.

Master the terminal, and you master the system.

Jakarta, 2026

Onno W. Purbo

PART I — TERMINAL FOUNDATIONS

Chapter 1 — Understanding the Linux Terminal Environment

The Linux terminal is not merely a black window filled with text. It is a **direct interface to the operating system**, a place where automation, system control, debugging, and creativity converge. Mastering the terminal means thinking like the system itself.

This chapter builds your foundational understanding of how the Linux command-line environment works. By the end, you will not only know how to run commands—you will understand *why* they behave the way they do.

1. What Is a Shell? How Bash Works

At its core, a **shell** is a program that interprets user commands and passes them to the Linux kernel for execution.

Think of it as a translator:

```
You → Shell → Kernel → Hardware
```

The kernel manages CPU, memory, disk, and processes.

The shell allows humans to communicate with it.

In Ubuntu, the default shell is **Bash** (*Bourne Again Shell*). Bash is powerful because it is:

- A command interpreter
- A scripting language
- A programmable environment

When you type:

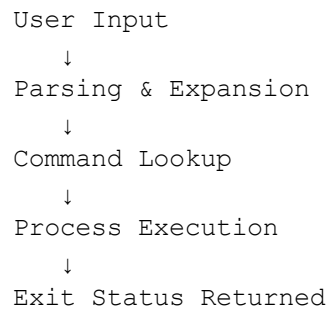
```
ls -l /home
```

Bash performs several internal steps:

1. Parses the command.
2. Expands wildcards or variables.
3. Searches for the executable in **\$PATH**.
4. Executes the program.
5. Returns an exit status.

This layered behavior explains why Bash can do more than simply “run programs.” It performs expansions, substitutions, and redirections before execution.

Illustration: The Shell Execution Flow



This understanding will later help you debug scripts and complex pipelines.

2. Terminal vs TTY vs Console

Students often use these terms interchangeably, but they have different meanings.

Terminal

A **terminal emulator** is a graphical application that allows you to interact with a shell. Examples:

- GNOME Terminal
- Konsole
- Tilix

TTY (*Teletypewriter*)

Originally physical devices, TTYs now refer to virtual text-based login sessions. You can access them using:

```
Ctrl + Alt + F3
```

You will see a login prompt like:

```
Ubuntu 24.04 LTS tty3
login:
```

Check your current terminal:

```
tty
```

Example output:

```
/dev/pts/0
```

Console

In modern Linux:

- GUI terminal = terminal emulator
- Virtual console = TTY
- System console = primary text interface

Understanding this distinction becomes important in server environments and troubleshooting boot failures. The **system console** is the primary interface connected directly to the system.

3. Command Anatomy (Command, Options, Arguments)

Linux commands follow a consistent structure:

```
command [options] [arguments]
```

Example

```
ls -l /etc
```

- `ls` → command
- `-l` → option (long listing format)
- `/etc` → argument (target directory)

Options

Options modify behavior.

```
ls -a      # show hidden files
ls -lh     # human-readable sizes
```

Short options can often be combined:

```
ls -lah
```

Arguments

Arguments specify the object of action.

```
cp file.txt backup.txt
```

The power of Linux lies in consistency. Once you understand this structure, you can quickly learn new commands.

4. Understanding the Prompt and Privilege Levels

A typical Bash prompt looks like this:

```
student@ubuntu:~$
```

Let's decode it:

- **student** → username
- **ubuntu** → hostname
- **~** → current directory (home)
- **\$** → normal user
- **#** → root user

Normal User vs Root

If you switch to root:

```
sudo -i
```

The prompt changes:

```
root@ubuntu:~#
```

The # symbol means full administrative privileges.

⚠ Always be cautious when operating as root.

A single incorrect command (like **rm -rf /**) can destroy a system.

Check current user:

```
whoami
```

Check privileges:

```
id
```

Understanding privilege separation is fundamental for Linux security.

5. Exit Codes and \$?

Every command in Linux returns an **exit code**.

- 0 → success
- Non-zero → error

Example:

```
ls /nonexistent
```

Then check:

```
echo $?
```

Output:

```
2
```

This means an error occurred.

Exit codes are critical in scripting:

```
mkdir test_dir
if [ $? -eq 0 ]; then
    echo "Directory created successfully."
fi
```

More elegant:

```
mkdir test_dir && echo "Success"
```

Or:

```
mkdir test_dir || echo "Failed"
```

This logic builds the foundation for automation.

6. Built-in vs External Commands

Not all commands are equal.

Some commands are built directly into Bash (built-ins). Others are external programs stored in directories like `/bin` or `/usr/bin`.

Identify command type

```
type cd
```

Output:

```
cd is a shell builtin
```

Another example:

```
type ls
```

Output:

```
ls is /usr/bin/ls
```

Which Command

```
which ls
```

Shows the executable location.

Understanding this distinction matters because:

- Built-ins execute faster.
- External commands create new processes.
- Overriding behavior can cause confusion.

7. Getting Help in Linux

Linux systems are self-documented.

Learning how to read documentation is a professional skill.

man (Manual Pages)

```
man ls
```

Manual pages are divided into sections:

- NAME
- SYNOPSIS
- DESCRIPTION
- OPTIONS
- EXAMPLES

Search within man:

```
/pattern
```

Quit:

```
q
```

Manual sections:

```
man 5 passwd
```

Section 5 = file formats.

info

```
info coreutils
```

Provides more detailed documentation than [man](#).

--help

Quick summary:

```
ls --help
```

Best for fast option lookup.

whatis

```
whatis ls
```

Output:

```
ls (1) - list directory contents
```

apropos

Search manuals by keyword:

```
apropos network
```

This is extremely useful when you do not know the exact command name.

/usr/share/doc

Many packages install documentation here.

Example:

```
ls /usr/share/doc/grep
```

You may find:

- README files
- Example configurations
- Licensing information

Professionals frequently consult these files.

Practical Mini Lab

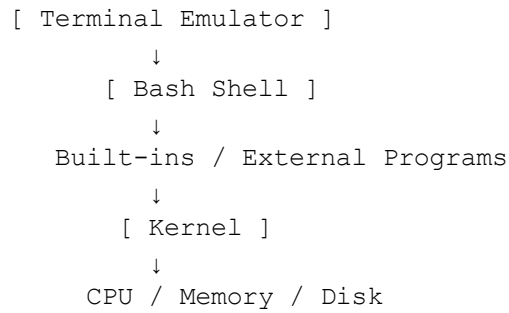
Try the following:

```
type echo
which echo
echo $?
man bash
ls -l /bin
tty
```

Reflect:

- Which commands are built-in?
- What is your current terminal?
- What exit code does a successful command return?

Conceptual Topology: How Everything Connects



Understanding this stack changes how you approach troubleshooting.

Chapter Summary

In this chapter, you learned:

- What a **shell** is and how Bash processes commands.
- The difference between **terminal, TTY, and console**.
- The structure of Linux commands.
- How prompts reflect privilege levels.
- Why **exit codes** are essential.
- The difference between built-in and external commands.
- How to efficiently use Linux documentation tools.

This knowledge transforms the terminal from a mysterious black box into a logical, predictable system.

In the next chapter, we will explore the Linux filesystem hierarchy and navigation—where you will begin interacting directly with the structure of the operating system.

PART II — NAVIGATING FILES AND FILESYSTEMS (~34%)

Chapter 2 — Linux Filesystem Hierarchy (FHS)

One of the most powerful ideas in Linux is that **everything is organized into a single hierarchical filesystem tree**.

Unlike Windows, which uses multiple drive letters (C:, D:, etc.), Linux organizes everything under one root directory:

/

From this single root, every file, device, configuration, log, and program branches out.

Understanding this structure is essential—not only for passing exams—but for becoming confident in navigating and managing real systems.

1. The Root of Everything: /

The forward slash (/) represents the **root directory**. It is the top of the filesystem hierarchy.

Visualize it as an inverted tree:

```

  /
 /|\
/ | \
etc usr home var tmp root
```

Everything in Linux lives somewhere beneath /.

To view the top-level directories:

```
ls -l /
```

This command reveals the fundamental structure of the operating system.

2. Key Directories in the Linux Filesystem Hierarchy

The *Filesystem Hierarchy Standard (FHS)* defines what each directory is intended for. Ubuntu follows this standard closely.

Understanding these directories will dramatically improve your troubleshooting and administration skills.

/etc — System Configuration

The **/etc** directory stores **configuration files**.

Think of it as the “control room” of the system.

Examples:

```
ls /etc
```

You will find:

- `/etc/passwd` → user accounts
- `/etc/shadow` → encrypted passwords
- `/etc/hosts` → hostname resolution
- `/etc/apt/` → package configuration

Important principle:

If something controls system behavior, it likely lives in `/etc`.

`/var` — Variable Data

The `/var` directory stores data that **changes frequently**.

Examples:

```
ls /var
```

Common subdirectories:

- `/var/log` → log files
- `/var/spool` → print/mail queues
- `/var/lib` → application state data
- `/var/cache` → cached files

Example:

```
ls -lh /var/log
```

Logs are crucial for debugging. If a service fails, always check `/var/log` or `journalctl`.

`/usr` — User Programs and Applications

The `/usr` directory contains **installed software and shared libraries**.

It includes:

- `/usr/bin` → user commands
- `/usr/sbin` → administrative commands
- `/usr/lib` → libraries
- `/usr/share` → documentation and shared data

Example:

```
ls /usr/bin | head
```

Most external commands you run come from `/usr/bin`.

`/home` — User Directories

Each regular user has a personal directory inside `/home`.

Example:

```
ls /home
```

You may see:

```
student
admin
```

Your home directory:

```
echo $HOME
```

Or shorthand:

```
cd ~
```

This directory stores:

- Personal files
- Downloads
- Documents
- User-specific configuration (`.bashrc`, `.profile`)

`/root` — The Root User's Home

The superuser has a separate home directory:

```
/root
```

This is not inside `/home`.

Only root should normally access it.

`/tmp` — Temporary Files

The `/tmp` directory stores temporary data.

It is:

- Writable by all users
- Often cleaned automatically
- Meant for short-lived files

Example:

```
ls -ld /tmp
```

You will notice special permissions (sticky bit), ensuring users cannot delete others' files.

3. Absolute vs Relative Paths

Understanding paths is critical for efficient navigation.

Absolute Paths

An absolute path begins with `/`.

Example:

```
cd /etc
```

It always starts from root.

Absolute paths are precise and predictable.

Relative Paths

Relative paths depend on your current location.

Example:

```
cd Documents
```

If you are in `/home/student`, this takes you to:

```
/home/student/Documents
```

Special Shortcuts

- `.` → current directory
- `..` → parent directory
- `~` → home directory

Example:

```
cd ..  
cd ~/Downloads
```

Understanding relative navigation dramatically improves speed.

4. Navigating with `pwd`, `cd`, `pushd`, `popd`

`pwd` — Print Working Directory

Shows your current location:

```
pwd
```

Output:

```
/home/student
```

`cd` — Change Directory

```
cd /var/log
```

Return home:

```
cd ~
```

Return to previous directory:

```
cd -
```

pushd and popd — Directory Stack Navigation

These commands maintain a stack of visited directories.

Example:

```
pushd /etc
pushd /var/log
dirs
```

You now have a stack.

Return:

```
popd
```

This is extremely useful in scripting and complex navigation workflows.

5. Advanced ls Usage

The `ls` command is deceptively powerful.

Common Useful Options

```
ls -l      # long format
ls -a      # include hidden files
ls -h      # human-readable sizes
ls -t      # sort by modification time
ls -S      # sort by size
ls -R      # recursive listing
```

Combine options:

```
ls -lah
```

Understanding Long Format Output

Example:

```
ls -l
```

Output:

```
-rw-r--r-- 1 student student 1200 Jan 10 10:00 notes.txt
```

Breakdown:

- `-rw-r--r--` → permissions
- `1` → link count
- `student` → owner
- `student` → group

- 1200 → size (bytes)
- Jan 10 → modification date
- notes.txt → filename

Every column tells a story.

6. stat — Detailed File Information

While `ls` provides summary information, `stat` provides detailed metadata.

Example:

```
stat notes.txt
```

Output includes:

- Access time
- Modify time
- Change time
- Inode number
- File size
- Block usage

This command is invaluable for debugging permissions and timestamp issues.

7. tree — Visual Directory Structure

If installed, `tree` shows a graphical structure.

Example:

```
tree /etc | head
```

Output:

```
/etc
├── apt
├── cron.d
├── passwd
└── shadow
```

This visual structure helps beginners understand hierarchy more intuitively.

8. Introduction to Inodes

In Linux, a file is not just its name.

Each file is represented by an **inode** (index node).

An inode stores:

- File permissions
- Owner
- Size
- Timestamps
- Disk block locations

It does **not** store the filename.

To view inode numbers:

```
ls -li
```

Example:

```
131072 notes.txt
```

Hard links share the same inode:

```
ln notes.txt copy.txt  
ls -li
```

Both files now show the same inode number.

Understanding inodes explains:

- Why deleting a file does not always free space immediately.
- How hard links work.
- Why filesystem corruption affects metadata.

Conceptual Illustration: Filesystem Architecture



The Linux filesystem is not chaotic—it is carefully structured.

Practical Lab

Try the following:

```
pwd
cd /
ls -l
cd /etc
ls -lh | head
stat /etc/passwd
ls -i /etc/passwd
pushd /var
pushd /usr
dirs
popd
```

Reflect:

- Which directory stores configuration?
- Which directory stores logs?
- What is the inode number of `/etc/passwd`?

Chapter Summary

In this chapter, you learned:

- The structure of the **Linux Filesystem Hierarchy (FHS)**.
- The purpose of `/etc`, `/var`, `/usr`, `/home`, `/root`, and `/tmp`.
- The difference between **absolute and relative paths**.
- How to navigate efficiently with `pwd`, `cd`, `pushd`, and `popd`.
- How to use advanced `ls` options.
- How `stat` reveals metadata.
- What **inodes** are and why they matter.

Mastering filesystem navigation transforms the terminal from intimidating to intuitive.

In the next chapter, we will dive deeper into file and directory management—where you will begin manipulating the filesystem confidently and safely.

Chapter 3 — File and Directory Management

In the previous chapter, you learned how Linux organizes its filesystem. Now we begin manipulating it.

File management in Linux is not about clicking icons. It is about understanding structure, metadata, permissions, and links. When you truly understand these tools, you gain control over the system.

This chapter covers the essential commands that every Linux professional must master.

1. `touch` — Creating and Updating Files

The `touch` command is deceptively simple. It creates an empty file if it does not exist. If the file already exists, it updates its timestamp.

Create a new file

```
touch notes.txt
```

Verify:

```
ls -l notes.txt
```

If the file already exists:

```
touch notes.txt
```

This updates the *modification time*.

You can verify using:

```
stat notes.txt
```

Notice the timestamps:

- Access
- Modify
- Change

Why this matters

Updating timestamps is important for:

- Build systems
- Backup tools
- Automation scripts

You can also set custom timestamps:

```
touch -t 202601010900 report.txt
```

This sets the file to January 1, 2026, at 09:00.

Key idea: `touch` manipulates metadata, not just files.

2. `cp` — Copying Files and Directories

The `cp` command duplicates files or directories.

Basic copy

```
cp notes.txt backup.txt
```

Recursive copy

To copy directories:

```
cp -r project/ project_backup/
```

The `-r` flag means *recursive*.

Preserve attributes

To preserve permissions, timestamps, and ownership:

```
cp -a project/ archive/
```

The `-a` flag means *archive mode* (recommended for backups).

Interactive copy (safe practice)

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

This prompts before overwriting.

Professional Tip

When backing up system configuration:

```
sudo cp -a /etc /backup/etc_backup
```

The `-a` option preserves critical metadata.

3. `mv` — Moving and Renaming Files

The `mv` command both moves and renames files.

Rename a file

```
mv notes.txt report.txt
```

Move to another directory

```
mv report.txt /home/student/Documents/
```

Interactive mode

```
mv -i report.txt archive/
```

Important Behavior

Unlike `cp`, `mv` does not duplicate data if the move happens within the same filesystem. It simply updates directory entries.

This makes it extremely efficient.

4. `rm` — Removing Files (Safe Practices)

The `rm` command deletes files permanently.

There is no recycle bin.

Remove a file

```
rm notes.txt
```

Remove a directory recursively

```
rm -r project/
```

Force removal

```
rm -rf project/
```

⚠ **WARNING:** `rm -rf` can destroy an entire system.

Safe Practices

- Always use `-i` when learning:

```
rm -i file.txt
```

- Use `ls` before `rm`.
- Avoid using `rm -rf /` under any circumstances.

Professional administrators often alias:

```
alias rm='rm -i'
```

This adds a safety layer.

5. `mkdir -p` — Creating Directories

Create a directory:

```
mkdir projects
```

Create nested directories:

```
mkdir -p projects/2026/january/reports
```

The `-p` flag:

- Creates parent directories if needed
- Avoids errors if they already exist

6. `rmdir` — Removing Empty Directories

```
rmdir empty_directory
```

This works only if the directory is empty.

If not empty, use:

```
rm -r directory
```

7. `ln` — Hard Links and Symbolic Links

Links are one of the most powerful concepts in Linux.

Hard Links

Create a hard link:

```
ln original.txt copy.txt
```

Both files now share the same inode.

Check:

```
ls -i
```

They show identical inode numbers.

Key properties of hard links:

- Share the same inode
- Cannot span filesystems
- Cannot link directories (usually)
- Deleting one does not delete the data until all links are removed

Symbolic Links (*Soft Links*)

Create a symbolic link:

```
ln -s original.txt shortcut.txt
```

Check:

```
ls -l
```

Output shows:

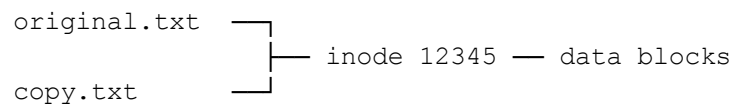
```
shortcut.txt -> original.txt
```

Symbolic links:

- Point to filenames
- Can span filesystems
- Can link directories
- Break if the target is deleted

Conceptual Illustration

Hard Link:



Symbolic Link:

shortcut.txt → original.txt → inode 12345

Understanding links is crucial for:

- Package management
- System libraries
- Version control
- Disk space optimization

8. `file` — Identifying File Types

The `file` command determines file type based on content, not extension.

Example:

```
file notes.txt
```

Output:

```
notes.txt: ASCII text
```

Example with executable:

```
file /bin/ls
```

Output:

```
ELF 64-bit LSB executable
```

This is important because Linux does not rely on extensions like `.exe`.

9. `namei` — Path Resolution

The `namei` command breaks down a path into its components.

Example:

```
namei /etc/apt/sources.list
```

It shows:

- Each directory in the path
- Permissions
- Ownership

This is extremely helpful for debugging permission problems.

10. `umask` — Default Permissions

When creating new files, permissions are not random.

They are controlled by `umask`.

Check current mask:

```
umask
```

Typical output:

```
0022
```

This means:

- Files default to 644
- Directories default to 755

Example

Set a stricter mask:

```
umask 0077
```

Now new files are accessible only to the owner.

`umask` is critical in multi-user systems and secure environments.

11. Link Count and Inode Behavior

Recall from Chapter 2 that every file has an inode.
Check link count:

```
ls -l
```

The number after permissions represents link count.
Example:

```
-rw-r--r-- 2 student student 1200 Jan 10 notes.txt
```

The **2** means:

- Two directory entries point to the same inode.

When you delete one:

```
rm copy.txt
```

The data remains until link count becomes zero.

This explains why disk space may not immediately free when files are “deleted” but still open by a process.

Advanced troubleshooting example:

```
lsof | grep deleted
```

This shows open deleted files.

Practical Lab

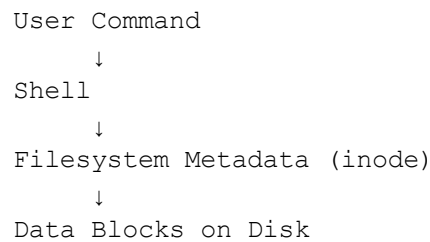
Try:

```
touch demo.txt
cp demo.txt demo_copy.txt
ln demo.txt demo_hardlink.txt
ln -s demo.txt demo_symlink.txt
ls -li
stat demo.txt
rm demo.txt
ls -li
```

Observe:

- Which files still access data?
- What happens to the symbolic link?
- How does the inode behave?

Conceptual Topology of File Operations



When you run **cp**, **mv**, or **rm**, you are manipulating metadata as much as data.

Chapter Summary

In this chapter, you mastered:

- Creating files with **touch**
- Copying with **cp -r** and **cp -a**
- Moving and renaming with **mv**
- Safe deletion with **rm**
- Creating directories using **mkdir -p**
- Removing empty directories with **rmdir**
- Understanding **hard and symbolic links**
- Identifying file types with **file**
- Resolving paths with **namei**
- Managing default permissions using **umask**
- Understanding **link count and inode behavior**

These are not just commands. They are the foundation of Linux system management.

Chapter 4 — Searching and Filtering Data

In real-world systems, data grows rapidly. Logs expand, configuration files accumulate, directories deepen, and applications generate thousands of files.

Professionals do not scroll through data manually. They filter it.

This chapter teaches you how to:

- Locate files precisely
- Search file content efficiently
- Compare files intelligently
- Combine commands strategically

These skills are fundamental for system administrators, developers, and security analysts.

1. `find` — The Most Powerful File Search Tool

The `find` command searches the filesystem based on conditions.

Basic syntax:

```
find [starting_path] [conditions]
```

Search by Name

```
find /home -name "notes.txt"
```

Case-insensitive search:

```
find /home -iname "notes.txt"
```

Wildcards are supported:

```
find /var/log -name "*.log"
```

Search by Type

- `f` → file
- `d` → directory
- `l` → symbolic link

```
find /etc -type f
```

Search only directories:

```
find /home -type d -name "project*"
```

Search by Size

```
find / -size +100M
```

This finds files larger than 100MB.

Other examples:

```
find /tmp -size -1M
```

Find files smaller than 1MB.

Search by Time

- **-mtime** → modified time (days)
- **-atime** → accessed time
- **-ctime** → changed metadata

Example:

```
find /var/log -mtime -2
```

Files modified within the last 2 days.

This is extremely useful for:

- Log analysis
- Incident investigation
- Backup validation

Combine Conditions

```
find /home -type f -name "*.txt" -size +10k
```

Find text files larger than 10 KB.

Execute Action on Results

Delete found files:

```
find /tmp -type f -name "*.tmp" -delete
```

Or safer:

```
find /tmp -type f -name "*.tmp" -exec rm {} \;
```

⚠ Always test with **-print** before using destructive options.

2. `locate` and `updatedb`

Unlike `find`, the `locate` command searches a database of indexed filenames. This makes it extremely fast.

```
locate passwd
```

The database is maintained by:

```
sudo updatedb
```

Key differences:

find	locate
Real-time search	Searches index
Slower	Very fast
Accurate	May be outdated

Use `locate` for speed.

Use `find` for precision.

3. `which` and `whereis`

These commands help identify program locations.

`which`

```
which ls
```

Shows:

```
/usr/bin/ls
```

This tells you where the executable is located in `$PATH`.

whereis

```
whereis bash
```

Output example:

```
bash: /bin/bash /usr/share/man/man1/bash.1.gz
```

It shows:

- Binary location
- Source (if available)
- Manual page

This is helpful for troubleshooting multiple installed versions.

4. **grep** — Searching Inside Files

If **find** searches filenames, **grep** searches content.

Basic usage:

```
grep "error" logfile.txt
```

Recursive Search

```
grep -r "ERROR" /var/log
```

Searches all files inside **/var/log**.

Case-Insensitive Search

```
grep -i "warning" logfile.txt
```

Invert Match

```
grep -v "INFO" logfile.txt
```

Shows lines that do NOT contain "INFO".

Line Numbers

```
grep -n "root" /etc/passwd
```

5. Basic Regular Expressions

Regular expressions (*regex*) allow pattern matching.

Anchors

- `^` → beginning of line
- `$` → end of line

Example:

```
grep "^root" /etc/passwd
```

Matches lines starting with "root".

Match lines ending with .conf:

```
grep "\.conf$" filelist.txt
```

Wildcards

- `.` matches any character:

```
grep "r..t" words.txt
```

Matches:

- root
- rust
- rent

Practical Log Filtering

```
grep -i "failed" /var/log/auth.log
```

Filter only SSH failures:

```
grep -i "failed password" /var/log/auth.log
```

Searching logs efficiently is a core system administration skill.

6. `diff` vs `cmp`

Comparing files is critical in configuration management.

`diff` — Human-Readable Differences

```
diff file1.txt file2.txt
```

Shows line-by-line differences.

Useful for:

- Comparing configuration files
- Code changes
- System snapshots

`cmp` — Byte-by-Byte Comparison

```
cmp file1.txt file2.txt
```

Reports only whether files differ and at which byte.
Use `cmp` for binary files.

7. Introduction to Pipeline Thinking

Pipeline thinking is one of the most powerful concepts in Linux.
Instead of performing one task at a time, you chain commands together.
The pipe operator:

```
|
```

Sends output of one command to another.

Basic Pipeline Example

```
ls -l | grep ".txt"
```

This lists only `.txt` files.

Counting Results

```
ls /var/log | wc -l
```

Counts log files.

Advanced Example

Find large log files and sort them:

```
find /var/log -type f -size +1M | sort
```

Or count matching errors:

```
grep -i "error" /var/log/syslog | wc -l
```

Visualization of Pipeline Flow

Command A → Output → Command B → Output → Command C

Example:

```
cat logfile.txt | grep "error" | sort | uniq
```

Data flows like water through pipes.

This is how Linux professionals think.

Practical Lab

Try:

```
find /etc -type f -name "*.conf"
grep -r "^root" /etc
locate bash
which python3
whereis ssh
diff /etc/passwd /etc/passwd
```

Reflect:

- Which command is faster: **locate** or **find**?
- What happens when you combine **find** with **grep**?
- Can you count failed login attempts in **/var/log/auth.log**?

Conceptual Topology of Data Filtering

```
Filesystem
  ↓
find
  ↓
grep
  ↓
sort
  ↓
wc
```

You are not just running commands.
You are constructing data-processing flows.

Chapter Summary

In this chapter, you learned:

- How to search files using **find**
- How to use **locate** and update its database
- How to identify executable locations with **which** and **whereis**
- How to search inside files using **grep**
- How to use anchors (^, \$) in regular expressions
- The difference between **diff** and **cmp**
- How to think in pipelines using |

Searching and filtering is where the Linux command line becomes transformative.

In the next chapter, we will explore text processing and advanced data manipulation — turning raw data into meaningful information.

PART III — TEXT PROCESSING & PIPELINES

Chapter 5 — Working with Text Files

In Linux, text is everywhere. System configuration files are text. Log files are text. Even device information is often presented as text.

This chapter introduces the essential tools used to:

- View text
- Monitor text in real time
- Count and summarize text
- Sort and filter text
- Extract specific fields
- Transform characters
- Format output
- Save processed results

These commands are simple individually. Together, they become extremely powerful.

1. `cat` and `tac` — Displaying File Contents

`cat` — Concatenate and Display

The `cat` command displays file content.

```
cat notes.txt
```

It prints the file to standard output.

Combine files:

```
cat file1.txt file2.txt
```

Number lines:

```
cat -n notes.txt
```

Show end-of-line markers:

```
cat -E notes.txt
```

⚠ Important: Avoid using `cat` for very large files. It dumps everything at once.

`tac` — Reverse Display

The `tac` command displays lines in reverse order.

```
tac logfile.txt
```

This is useful when the newest log entries appear at the bottom. Think of it as “cat” backwards.

2. `less` — Viewing Large Files Safely

Unlike `cat`, `less` opens files interactively.

```
less logfile.txt
```

Navigation inside `less`:

- `/pattern` → search forward
- `n` → next match
- `G` → end of file
- `g` → beginning
- `q` → quit

Professional administrators rely heavily on `less`.

Example:

```
sudo less /var/log/auth.log
```

This allows you to analyze logs without overwhelming your terminal.

3. `head` and `tail` — Viewing Portions of Files

`head`

Shows first lines:

```
head logfile.txt
```

Specify number of lines:

```
head -n 20 logfile.txt
```

tail

Shows last lines:

```
tail logfile.txt
```

tail -f — Real-Time Monitoring

This is one of the most powerful options.

```
tail -f /var/log/syslog
```

It continuously updates as new lines are added.

Use case: Monitor live server activity.

Press **Ctrl + C** to stop.

Real-World Example

Monitor failed login attempts:

```
sudo tail -f /var/log/auth.log
```

You now see authentication attempts in real time.

This is invaluable for debugging and security monitoring.

4. wc — Counting Lines, Words, and Characters

The **wc** command summarizes file content.

```
wc notes.txt
```

Output format:

```
lines words characters filename
```

Common options:

```
wc -l notes.txt    # count lines
wc -w notes.txt    # count words
wc -c notes.txt    # count bytes
```

Example: Count log entries.

```
grep "ERROR" logfile.txt | wc -l
```

This counts how many errors occurred.

5. sort — Sorting Text

Sorting is essential for organizing data.

Basic usage:

```
sort names.txt
```

Sort in reverse:

```
sort -r names.txt
```

Numeric sort:

```
sort -n numbers.txt
```

Sort by specific column:

```
sort -k2 data.txt
```

This is extremely useful for analyzing structured output.

6. **uniq** — Removing Duplicate Lines

The **uniq** command removes adjacent duplicates.

Important: The file must be sorted first.

Example:

```
sort names.txt | uniq
```

Count duplicates:

```
sort names.txt | uniq -c
```

Find only repeated lines:

```
sort names.txt | uniq -d
```

Practical example:

```
cut -d: -f1 /etc/passwd | sort | uniq
```

This lists unique usernames.

7. `cut` — Extracting Fields

The `cut` command extracts specific columns from text.

Example file:

```
student:1001:1001:/home/student:/bin/bash
```

Extract username:

```
cut -d: -f1 /etc/passwd
```

- `-d:` → delimiter is colon
- `-f1` → field 1

Extract fields 1 and 7:

```
cut -d: -f1,7 /etc/passwd
```

This is useful for parsing configuration files.

8. `tr` — Transforming Characters

The `tr` command translates or deletes characters.

Convert lowercase to uppercase:

```
echo "linux" | tr 'a-z' 'A-Z'
```

Remove digits:

```
echo "abc123" | tr -d '0-9'
```

Replace spaces with underscores:

```
echo "file name.txt" | tr ' ' '_'
```

`tr` is simple but extremely powerful when combined with pipelines.

9. `column` — Formatting Output

The `column` command formats text into aligned columns.

Example:

```
cat /etc/passwd | column -t -s :
```

This makes the file easier to read.

Output becomes neatly aligned.

Useful for:

- Reports
- Structured text
- Data review

10. `tee` — Writing Output to File and Screen

The `tee` command writes output to both standard output and a file.

Example:

```
ls -l | tee listing.txt
```

You see output on screen AND it is saved to listing.txt.

Append mode:

```
ls -l | tee -a listing.txt
```

This is useful for:

- Logging command output
- Capturing troubleshooting sessions
- Creating reports

Visualizing Text Processing Flow

Imagine this pipeline:

```
cat logfile.txt | grep "error" | sort | uniq -c | sort -nr | head
```

Flow:

```
File → Filter → Sort → Count → Rank → Display Top Results
```

This is command-line data analytics.

Practical Lab

Try:

```
cat /etc/passwd
less /etc/passwd
head -n 5 /etc/passwd
tail -n 5 /etc/passwd
wc -l /etc/passwd
cut -d: -f1 /etc/passwd
sort /etc/passwd | uniq
echo "linux terminal" | tr 'a-z' 'A-Z'
ls -l | tee output.txt
```

Reflect:

- How many users exist?
- Which command is best for large logs?
- How can you monitor logs live?

Conceptual Illustration

```
Raw Text
↓
Filter (grep)
↓
Transform (tr, cut)
↓
Sort (sort)
↓
Summarize (wc, uniq)
↓
Save (tee)
```

This is the mindset of a Linux professional.

Chapter Summary

In this chapter, you mastered:

- Viewing files with **cat** and **tac**
- Interactive viewing with **less**
- Monitoring files with **tail -f**
- Counting data using **wc**
- Sorting data with **sort**
- Removing duplicates with **uniq**
- Extracting fields with **cut**
- Transforming text with **tr**
- Formatting output using **column**
- Saving output using **tee**

You now have the tools to turn raw system data into structured insight.

In the next chapter, we will combine these commands using redirection and pipelines to build powerful workflows.

Chapter 6 — Redirection, Pipes, and Shell Operators

So far, you have learned how to:

- Navigate files
- Search for data
- Manipulate text

Now you will learn how to **connect commands together**.

Linux does not expect you to run isolated commands. It expects you to build chains of operations.

Understanding redirection, pipes, and shell expansion is what separates a beginner from a professional.

1. Understanding Standard Streams

Before learning redirection, you must understand that every Linux process has three standard data streams:

Stream	Number	Purpose
Standard Input	0	Input to program
Standard Output	1	Normal output
Standard Error	2	Error messages

Visualize it:

```
Command
  ↑
stdin (0)
  ↓
stdout (1)
  ↓
stderr (2)
```

Redirection allows you to control where these streams go.

2. Output Redirection: > and >>

Overwrite Output (>)

Redirect standard output to a file:

```
ls -l > listing.txt
```

Instead of printing to the screen, the output is saved in `listing.txt`.

⚠ If the file exists, it will be overwritten.

Append Output (>>)

Append output to an existing file:

```
date >> log.txt
```

This preserves existing content and adds new data at the bottom.

Practical Example

Create a simple system report:

```
echo "System Report" > report.txt
uptime >> report.txt
df -h >> report.txt
free -h >> report.txt
```

You have just built a multi-command report using redirection.

3. Error Redirection: 2> and &>

Not all output is the same. Errors are sent to *stderr* (stream 2).

Redirect Only Errors

```
ls /nonexistent 2> error.log
```

The error message is saved in `error.log`.

Redirect Both Output and Errors

```
command &> output.txt
```

Or more explicitly:

```
command > output.txt 2>&1
```

This sends:

- Standard output → file
- Standard error → same file

Professional Use Case

Log everything during troubleshooting:

```
sudo apt update &> update_debug.log
```

This captures all messages for later review.

4. Input Redirection: <

Input redirection feeds a file into a command.

Example:

```
wc -l < notes.txt
```

This counts lines using file input instead of specifying filename as argument. While often optional, input redirection becomes useful in scripting and automation.

5. Pipes: |

The pipe operator sends the output of one command into another. Instead of saving to a file, data flows directly between commands.

Basic Example

```
ls -l | grep ".txt"
```

Flow:

```
ls -l → grep → output
```

Multi-Step Pipeline

```
cat logfile.txt | grep "error" | sort | uniq -c
```

This:

1. Reads file
2. Filters lines containing "error"
3. Sorts results
4. Counts duplicates

Visualization

Command A → Command B → Command C → Command D

Data flows through each stage like water through pipes.
This concept is central to Linux philosophy.

6. `xargs` — Converting Input into Arguments

Some commands expect arguments rather than input from a pipe.
`xargs` bridges that gap.

Example Without `xargs`

```
find . -name "*.tmp"
```

To delete results:

```
find . -name "*.tmp" | xargs rm
```

`xargs` converts each line into arguments for `rm`.

Safer Version

```
find . -name "*.tmp" -print0 | xargs -0 rm
```

This safely handles filenames with spaces.

Example: Batch File Processing

```
find /var/log -name "*.log" | xargs gzip
```

Compress all log files.

7. Command Substitution: `$( )`

Command substitution inserts output of one command into another.

Modern Syntax

```
echo "Today is $(date)"
```

The shell executes `date` first, then substitutes its output.

Practical Example

Create a dated backup:

```
cp report.txt report_$(date +%F).txt
```

If today is 2026-03-15:

```
report_2026-03-15.txt
```

This is extremely useful in scripting.

8. Quoting Rules

The shell interprets special characters unless quoted.

There are three main quoting styles:

Double Quotes " "

Allow variable expansion:

```
name="Linux"
echo "Hello $name"
```

Output:

```
Hello Linux
```

Single Quotes ' '

Prevent expansion:

```
echo 'Hello $name'
```

Output:

```
Hello $name
```

Backslash \

Escape a single character:

```
echo \$HOME
```

Understanding quoting prevents unexpected behavior.

9. Globbing (*, ?, [])

Globbing expands wildcard patterns before command execution.

Asterisk *

Matches any number of characters:

```
ls *.txt
```

Question Mark ?

Matches one character:

```
ls file?.txt
```

Matches:

- file1.txt
- fileA.txt

Brackets []

Match character ranges:

```
ls file[1-5].txt
```

Matches:

- file1.txt
- file2.txt
- file3.txt
- file4.txt
- file5.txt

Globbing happens before the command runs.

The shell expands patterns first.

10. Metacharacters and Shell Expansion

The shell processes special characters in stages:

1. Brace expansion
2. Tilde expansion
3. Parameter expansion
4. Command substitution
5. Arithmetic expansion
6. Globbing

Example:

```
echo {A,B,C}.txt
```

Expands to:

```
A.txt B.txt C.txt
```

Tilde Expansion

```
cd ~
```

Expands to your home directory.

Variable Expansion

```
echo $HOME
```

Arithmetic Expansion

```
echo $((5 + 3))
```

Output:

```
8
```

Understanding shell expansion explains why certain commands behave unexpectedly.

Real-World Example: Building a Data Pipeline

Find largest log files and count errors:

```
find /var/log -type f -size +1M | tee large_logs.txt | xargs  
grep -i "error" | wc -l
```

Flow:

```
Find files → Save list → Search content → Count results
```

You have built a multi-stage diagnostic workflow.

Practical Lab

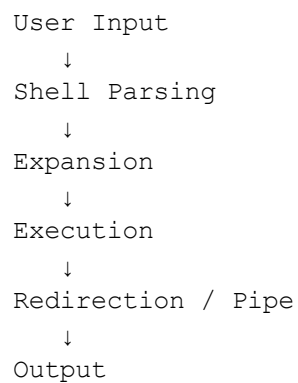
Try:

```
ls > files.txt
ls /nonexistent 2> error.txt
ls -l | grep ".conf"
echo "Backup created at $(date)"
find /etc -name "*.conf" | xargs ls -l
echo *.txt
echo file{1..5}.log
```

Reflect:

- What expands before execution?
- What is the difference between `>` and `>>`?
- How does quoting change behavior?

Conceptual Model



This is the internal workflow of Bash.
Understanding this model gives you control.

Chapter Summary

In this chapter, you mastered:

- Output redirection (`>`, `>>`)
- Error redirection (`2>`, `&>`)
- Input redirection (`<`)
- Pipes (`|`)
- Argument processing with `xargs`
- Command substitution `$( )`
- Quoting rules
- Globbing patterns (`*`, `?`, `[ ]`)
- Shell expansion and metacharacters

These tools allow you to build complex command chains and automate real-world tasks.

In the next chapter, we will explore process management and system resource control — where you begin managing running programs directly.

PART IV — MANAGING SYSTEM RESOURCES

(~33%)

Chapter 7 — Process Management

Every program you run becomes a **process**. The shell itself is a process. Your terminal emulator is a process. Even the graphical desktop is composed of processes.

A Linux system is not a collection of files — it is a collection of running processes.

This chapter teaches you how to:

- View running processes
- Monitor resource usage
- Search for specific processes
- Send signals to control them
- Adjust scheduling priority
- Manage background jobs

These skills are essential for system administrators and developers alike.

1. Viewing Processes with `ps`

The `ps` command shows process information.

```
ps aux
```

```
ps aux
```

Breakdown:

- `a` → show processes for all users
- `u` → display user-oriented format
- `x` → include processes without terminal

Output columns include:

- USER
- PID (Process ID)
- %CPU
- %MEM
- VSZ (virtual memory size)
- RSS (resident memory)
- TTY
- STAT (process state)
- START
- TIME
- COMMAND

Example:

```
student  2345  0.1  1.2  23456  3456 pts/0  S   09:00  0:01
bash
```

`ps -ef`

```
ps -ef
```

This displays:

- UID
- PID
- PPID (Parent PID)
- C (CPU usage factor)
- STIME
- TTY
- TIME
- CMD

Example:

```
student  2345  1234  0  09:00 pts/0  00:00:01 bash
```

The **PPID** column shows process hierarchy.

Visualizing Process Hierarchy

```
systemd (PID 1)
├─ sshd
│   └─ bash
│       └─ ps
└─ cron
```

Understanding parent-child relationships helps diagnose runaway processes.

2. Monitoring in Real Time: **top** and **htop**

`top`

```
top
```

top displays:

- CPU usage
- Memory usage
- Running processes
- Load averages

Important keys inside `top`:

- `P` → sort by CPU
- `M` → sort by memory
- `k` → kill process
- `q` → quit

htop

If installed:

```
htop
```

`htop` provides:

- Colorful interface
- Scrollable process list
- Mouse support
- Easier filtering

Install if needed:

```
sudo apt install htop
```

Understanding Load Average

Displayed at top of `top`:

```
load average: 0.30, 0.45, 0.60
```

These represent system load over:

- 1 minute
- 5 minutes
- 15 minutes

On a 4-core system:

- Load below 4 → healthy
- Load consistently above 4 → overloaded

3. Searching for Processes with `pgrep`

Instead of manually scanning `ps`, use `pgrep`.

```
pgrep ssh
```

This returns process IDs.

Show full command:

```
pgrep -a ssh
```

Combine with `ps`:

```
ps -fp $(pgrep ssh)
```

This is more precise and script-friendly.

4. Controlling Processes: `kill`, `pkill`, `killall`

`kill`

Send a signal using PID:

```
kill 2345
```

Default signal is SIGTERM (15).

`pkill`

Kill by name:

```
pkill firefox
```

`killall`

Kills all processes with a given name:

```
killall firefox
```

5. Signals — Communicating with Processes

Signals are how you communicate with running processes.

Common signals:

Signal	Number	Purpose
SIGTERM	15	Graceful termination
SIGKILL	9	Force kill
SIGHUP	1	Reload configuration

Send Specific Signal

```
kill -15 2345  
kill -9 2345
```

⚠ **SIGKILL cannot be ignored.** It immediately terminates the process.

Reload Service (SIGHUP)

```
kill -HUP 2345
```

Many daemons reload configuration without stopping.

Visual Model

```
User → kill → Signal → Process → Kernel
```

Signals are instructions, not commands.

6. Priority Control: **nice** and **renice**

Linux uses a scheduler to allocate CPU time.

Each process has a priority (niceness value).

Range:

- -20 (highest priority)
- 19 (lowest priority)

Start Process with Lower Priority

```
nice -n 10 backup.sh
```

This makes the process “nicer” to others.

Change Priority of Running Process

```
renice 5 -p 2345
```

Only root can assign negative values.

This is essential when:

- Running heavy backups
- Preventing system slowdown
- Managing multi-user servers

7. Background Jobs

The shell allows job control.

Run in Background (&)

```
sleep 60 &
```

Output:

```
[1] 3456
```

View Jobs

```
jobs
```

Bring Job to Foreground

```
fg %1
```

Send Job to Background

```
bg %1
```

Visual Workflow

```
Foreground → Ctrl+Z → Background → fg
```

8. `nohup` — Survive Terminal Logout

Normally, processes terminate when the terminal closes.

`nohup` prevents that.

```
nohup long_script.sh &
```

Output is written to:

```
nohup.out
```

This is critical for:

- Remote SSH sessions
- Long-running tasks
- Server maintenance

Real-World Scenario

Imagine you start a large data export over SSH.
Instead of:

```
./export.sh
```

Use:

```
nohup ./export.sh > export.log 2>&1 &
```

Now:

- It runs in background
- Survives disconnect
- Logs output
- Captures errors

This is a professional-grade workflow.

Practical Lab

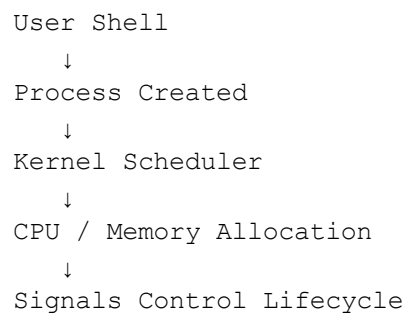
Try:

```
ps aux | grep bash
top
sleep 100 &
jobs
kill %1
nice -n 10 sleep 200 &
pgrep sleep
killall sleep
```

Reflect:

- What is the PID?
- What happens when you kill a process?
- How does priority affect scheduling?

Conceptual System Topology



The Linux system is constantly managing thousands of processes. Understanding this dynamic is fundamental to system mastery.

Chapter Summary

In this chapter, you learned:

- How to inspect processes with **ps aux** and **ps -ef**
- Monitor activity using **top** and **htop**
- Search processes with **pgrep**
- Terminate processes using **kill**, **pkill**, and **killall**
- Understand signals like **SIGTERM**, **SIGKILL**, and **SIGHUP**
- Adjust process priority using **nice** and **renice**
- Manage background jobs using **&**, **bg**, **fg**, and **jobs**
- Keep processes running with **nohup**

You now understand how Linux runs — not just what it stores.

In the next chapter, we will explore monitoring system resources and logs, where process behavior meets system performance analysis.

Chapter 8 — Monitoring Resources and Logs

Linux systems constantly consume and release resources:

- Memory
- CPU
- Disk space
- Network bandwidth
- File descriptors

At the same time, they continuously generate logs describing what is happening internally.

This chapter teaches you how to:

- Monitor memory and load
- Check disk usage
- Track active users
- Inspect mounted filesystems
- Analyze logs
- Understand how system services are managed

These are core operational skills for administrators and DevOps professionals.

1. Monitoring Memory Usage — `free -h`

The `free` command shows memory usage.

```
free -h
```

The `-h` option displays sizes in *human-readable* format.

Example output:

	total	used	free	shared	buff/cache	available
Mem:	7.7G	2.1G	3.5G	200M	2.1G	5.2G
Swap:	2.0	0.0	2.0G			

Understanding Columns

- **total** → Total physical memory
- **used** → Memory currently in use
- **free** → Completely unused memory
- **buff/cache** → Memory used for caching
- **available** → Realistic free memory available for applications

Important concept:

Linux aggressively uses RAM for caching. High “used” memory is not necessarily bad.

2. Checking System Load — **uptime**

`uptime`

Example:

```
10:12:45 up 2 days, 3:14, 2 users, load average: 0.45, 0.60, 0.75
```

This shows:

- Current time
- System uptime
- Number of logged-in users
- Load averages (1, 5, 15 minutes)

Understanding Load Average

Load represents the number of processes waiting for CPU.

On a 4-core system:

- Load < 4 → healthy
- Load ≈ 4 → fully utilized
- Load > 4 → overloaded

Monitoring load trends helps identify performance bottlenecks.

3. Monitoring Active Users — **who** and **w**

`who`

`who`

Shows logged-in users.

Example:

```
student pts/0 2026-03-15 09:00 (192.168.1.10)
```

`w`

`w`

Displays:

- Logged-in users
- What they are running
- System load
- Idle time

Example:

USER	TTY	FROM	LOGIN@	IDLE	JCPU	PCPU	WHAT
student	pts/0	192.168.1.10	09:00	5:00	0.01s	0.00s	bash

This command is extremely useful on multi-user servers.

4. Monitoring Disk Usage — `df -h`

`df` shows filesystem disk space usage.

```
df -h
```

Example:

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda1	50G	30G	18G	63%	/

Important columns:

- Size
- Used
- Available
- Use%
- Mount point

When `Use%` approaches 100%, system problems begin.

5. Measuring Directory Size — `du -sh`

While `df` shows overall disk usage, `du` shows usage by directory.

```
du -sh /var/log
```

Example:

```
450M    /var/log
```

Break down subdirectories:

```
du -h /var/log | sort -h
```

Find largest directories:

```
du -sh * | sort -h
```

This is essential when cleaning disk space.

6. Viewing Mounted Filesystems — `mount`

The `mount` command displays mounted filesystems.

```
mount
```

Example output:

```
/dev/sda1 on / type ext4 (rw,relatime)
```

Important components:

- Device
- Mount point
- Filesystem type
- Mount options

Check specific mount:

```
mount | grep /home
```

To manually mount:

```
sudo mount /dev/sdb1 /mnt
```

Mounted filesystems are listed in:

```
/etc/fstab
```

This file defines persistent mount configuration.

7. Log Management in Linux

Logs tell the story of your system.

`/var/log`

Most traditional log files reside here.

Examples:

- `/var/log/syslog`
- `/var/log/auth.log`
- `/var/log/kern.log`
- `/var/log/dpkg.log`

View a log:

```
sudo less /var/log/syslog
```

Search for errors:

```
grep -i error /var/log/syslog
```

Logs are crucial for:

- Troubleshooting
- Security auditing
- Performance analysis

8. `journalctl` — systemd Log Viewer

Modern Ubuntu systems use *systemd*.
`journalctl` provides access to systemd logs.

View All Logs

```
journalctl
```

Show Latest Logs

```
journalctl -n 20
```

Follow Logs in Real Time

```
journalctl -f
```

Equivalent to `tail -f`.

View Logs for Specific Service

```
journalctl -u ssh
```

This shows logs related to the SSH service.

Filter by Time

```
journalctl --since "1 hour ago"
```

This allows targeted troubleshooting.

9. Introduction to systemd Services

Modern Ubuntu systems use *systemd* as the init system.
The first process in Linux:

```
PID 1 → systemd
```

All services are managed by *systemd*.

View Service Status

```
systemctl status ssh
```

Start Service

```
sudo systemctl start nginx
```

Stop Service

```
sudo systemctl stop nginx
```

Enable Service at Boot

```
sudo systemctl enable nginx
```

Disable Service

```
sudo systemctl disable nginx
```

10. Understanding Target Units

Systemd uses target *units* instead of traditional runlevels.
Common targets:

- **multi-user.target** → text mode
- **graphical.target** → GUI mode

Check current target:

```
systemctl get-default
```

Change target:

```
sudo systemctl set-default multi-user.target
```

This controls system boot behavior.

System Monitoring Topology

Visualizing how monitoring tools relate:

```
Applications
  ↓
Processes
  ↓
CPU / Memory / Disk
  ↓
Logs (/var/log, journalctl)
  ↓
systemd Service Management
```

Monitoring is about understanding this flow.

Real-World Diagnostic Workflow

Imagine a user reports slow system performance.

Steps:

Check load:

```
uptime
```

Check memory:

```
free -h
```

Check disk usage:

```
df -h
```

Identify large directories:

```
du -sh /var/*
```

Inspect logs:

```
journalctl -xe
```

This structured approach is what professionals use.

Practical Lab

Try:

```
free -h
uptime
who
df -h
du -sh /var/log
mount | head
journalctl -n 10
systemctl status ssh
```

Reflect:

- How much memory is available?
- Which filesystem is nearly full?
- Which services are active?
- What do recent logs indicate?

Chapter Summary

In this chapter, you learned how to:

- Monitor memory with **free -h**
- Check load with **uptime**
- View users using **who** and **w**
- Inspect disk space with **df -h**
- Measure directory size with **du -sh**
- View mounted filesystems with **mount**
- Analyze logs in **/var/log**
- Use **journalctl** for systemd logging
- Manage services with **systemctl**
- Understand systemd target units

Monitoring transforms reactive troubleshooting into proactive system management.

In the next chapter, we will explore user management and file permissions — where security and operational control intersect.

PART V — SECURITY & USER MANAGEMENT

(~33%)

Chapter 9 — Users and Groups

In Linux, identity matters. Every action performed on the system is associated with:

- A **User ID (UID)**
- A **Group ID (GID)**

Every file has:

- An owner
- A group
- Permissions

If you understand users and groups, you understand how Linux enforces security.

1. Identifying Yourself — `whoami` and `id`

Before managing accounts, you must understand your current identity.

`whoami`

```
whoami
```

Output:

```
student
```

This shows the username of the current effective user.

`id`

```
id
```

Example output:

```
uid=1000(student) gid=1000(student)  
groups=1000(student),27(sudo)
```

Breakdown:

- **uid** → User ID
- **gid** → Primary Group ID
- **groups** → Supplementary groups

This command tells you what privileges you have.

Why This Matters

If you cannot modify a file, check:

```
id
ls -l filename
```

Permissions are always linked to identity.

2. Creating and Managing Users

Linux provides administrative tools for account management. These commands require root privileges.

`useradd` — Create a New User

```
sudo useradd newuser
```

By default, this creates:

- A new user
- A UID
- A primary group

To create a home directory:

```
sudo useradd -m newuser
```

Specify shell:

```
sudo useradd -m -s /bin/bash newuser
```

Setting Password — `passwd`

```
sudo passwd newuser
```

You will be prompted to enter and confirm a password.

`usermod` — Modify User

Add user to a group:

```
sudo usermod -aG sudo newuser
```

Important:

- `-a` → append
- `-G` → specify supplementary group

Without `-a`, existing group memberships may be removed.

Change login shell:

```
sudo usermod -s /bin/sh newuser
```

userdel — Remove User

```
sudo userdel newuser
```

Remove user and home directory:

```
sudo userdel -r newuser
```

Be careful: this deletes user data.

3. Managing Groups

Groups simplify permission management.

Instead of granting access per user, you grant access per group.

groupadd

Create a new group:

```
sudo groupadd developers
```

Add User to Group

```
sudo usermod -aG developers student
```

gpasswd

Manage group members:

Add user:

```
sudo gpasswd -a student developers
```

Remove user:

```
sudo gpasswd -d student developers
```

Example Scenario

You want all developers to access `/project`.

Steps:

```
sudo groupadd developers
sudo chown :developers /project
sudo chmod 770 /project
sudo usermod -aG developers alice
sudo usermod -aG developers bob
```

Now:

- Owner → root
- Group → developers
- Permissions → rwx for owner and group

This is secure multi-user collaboration.

4. Understanding `/etc/passwd`

The `/etc/passwd` file stores user account information.

View it:

```
cat /etc/passwd
```

Example entry:

```
student:x:1000:1000:Student User:/home/student:/bin/bash
```

Fields (colon-separated):

1. Username
2. Password placeholder (x)
3. UID
4. GID
5. Comment (GECOS)
6. Home directory
7. Login shell

Important:

The `x` means the password is stored in `/etc/shadow`.

This file must be readable by all users because system processes depend on it.

5. Understanding `/etc/shadow`

The `/etc/shadow` file stores encrypted passwords.

View (root only):

```
sudo cat /etc/shadow
```

Example:

```
student:$6$abcdefg...:19000:0:99999:7:::
```

Fields include:

- Username
- Encrypted password hash
- Last password change
- Password expiration policy

This file is readable only by root.

Why `/etc/shadow` Exists

Separating password hashes from `/etc/passwd` prevents non-root users from reading password hashes.

This enhances security.

6. User and Group Topology

Understanding relationships visually:

```
User (UID)
  ↓
Primary Group (GID)
  ↓
Supplementary Groups
  ↓
File Ownership
  ↓
Permission Enforcement
```

Example:

```
ls -l file.txt
```

Output:

```
-rw-r----- 1 alice developers 1024 Mar 15 file.txt
```

Meaning:

- Owner → alice
- Group → developers
- Others → no access

7. UID and GID Ranges

Typical ranges:

- 0 → root
- 1–999 → system accounts
- 1000+ → regular users

Check:

```
awk -F: '{print $1, $3}' /etc/passwd
```

This shows username and UID.

System accounts are used by services (e.g., [www-data](#), [mysql](#)).

8. Best Practices for User Management

Professional guidelines:

- Use groups for shared access
- Avoid logging in directly as root
- Use [sudo](#) instead
- Remove unused accounts
- Enforce password policies

Check password aging:

```
sudo chage -l student
```

This shows:

- Password expiration
- Account expiration

Real-World Scenario

Imagine you deploy a web server.
You want:

- Web service to run as **www-data**
- Developers to edit website files
- No public access

Steps:

```
sudo groupadd webdev
sudo usermod -aG webdev alice
sudo usermod -aG webdev bob
sudo chown -R www-data:webdev /var/www/html
sudo chmod -R 750 /var/www/html
```

This creates controlled access.

Practical Lab

Try:

```
whoami
id
sudo useradd -m testuser
sudo passwd testuser
sudo usermod -aG sudo testuser
grep testuser /etc/passwd
sudo userdel -r testuser
```

Reflect:

- What is your UID?
- What groups are you part of?
- How does adding to **sudo** group change privileges?

Conceptual Security Model

```
Authentication → Identity (UID)
↓
Authorization → Group Membership
↓
Permission Check → File/System Access
```

Linux security is identity-based.
Understanding users and groups gives you control.

Chapter Summary

In this chapter, you learned:

- How to identify yourself using **whoami** and **id**
- How to create users with **useradd**
- How to modify users with **usermod**
- How to set passwords with **passwd**
- How to remove users with **userdel**
- How to manage groups using **groupadd** and **gpasswd**
- How `/etc/passwd` stores account information
- How `/etc/shadow` protects password hashes

User management is the backbone of Linux security.

In the next chapter, we will explore file permissions and privilege escalation — where identity meets access control.

Chapter 10 — File Permissions and Privilege Escalation

Linux security is built on three pillars:

1. **Identity (UID/GID)**
2. **Permissions**
3. **Privilege escalation control**

Every file, directory, and executable in Linux follows a structured permission model. Understanding this model gives you full control over system security.

1. Understanding Permission Basics

Run:

```
ls -l file.txt
```

Example output:

```
-rw-r--r-- 1 alice developers 1024 Mar 15 file.txt
```

Breakdown:

```
-rw-r--r-
```

This is the permission string.
It consists of:

```
[Type] [Owner] [Group] [Others]
```

- **-** → regular file
- **d** → directory
- **l** → symbolic link

Permissions:

- **r** → read
- **w** → write
- **x** → execute

Structure:

Owner	Group	Others
rwX	r-X	r--

2. `chmod` — Changing Permissions

There are two methods: **symbolic** and **numeric (octal)**.

Symbolic Mode

Add execute permission:

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

Remove write permission:

```
chmod u-w file.txt
```

Grant group read permission:

```
chmod g+r file.txt
```

Where:

- **u** → user (owner)
- **g** → group
- **o** → others
- **a** → all

Example:

```
chmod u=rwx,g=rx,o= file.txt
```

Numeric (Octal) Mode

Each permission has numeric value:

Permission	Value
Read	4
Write	2
Execute	1

Example:

```
rwX = 4+2+1 = 7  
rw- = 4+2+0 = 6  
r-- = 4+0+0 = 4
```

Set permission:

```
chmod 755 script.sh
```

Meaning:

- Owner → 7 (rwx)
- Group → 5 (r-x)
- Others → 5 (r-x)

Common permission patterns:

Mode	Meaning
644	Standard file
755	Executable
600	Private file
700	Private script

Visual Permission Model

```
chmod 750 file
Owner: rwx
Group: r-x
Others: ---
```

This model is foundational to Linux security.

3. **chown** — Changing Ownership

Change file owner:

```
sudo chown alice file.txt
```

Change owner and group:

```
sudo chown alice:developers file.txt
```

Recursive ownership change:

```
sudo chown -R alice:developers project/
```

Ownership determines primary control over a file.

4. `chgrp` — Changing Group

```
sudo chgrp developers file.txt
```

This modifies only the group ownership.
Group ownership enables controlled collaboration.

5. Special Permissions — SUID, SGID, Sticky Bit

Beyond basic permissions, Linux supports special bits.

SUID (Set User ID)

When set on an executable, the program runs with the owner's privileges.
Example:

```
ls -l /usr/bin/passwd
```

You may see:

```
-rwsr-xr-x
```

The **s** indicates SUID.
Set manually:

```
sudo chmod 4755 program
```

The leading **4** sets SUID.
⚠ Use carefully. SUID can create security risks.

SGID (Set Group ID)

When set on a directory:

- Files created inside inherit the directory's group.

Set SGID:

```
sudo chmod 2755 shared_dir
```

Leading **2** sets SGID.
This is useful for team directories.

Sticky Bit

Used primarily on shared directories like `/tmp`.

Check:

```
ls -ld /tmp
```

Example:

```
drwxrwxrwt
```

The `t` indicates a sticky bit.

Meaning:

- Users can delete only their own files.

Set manually:

```
sudo chmod 1777 shared_dir
```

Leading `1` sets a sticky bit.

6. `umask` — Default Permission Mask

When creating files, Linux applies default permissions:

- Files → 666
- Directories → 777

Then subtracts the `umask`.

Check current mask:

```
umask
```

Example:

```
0022
```

Calculation:

```
666 - 022 = 644  
777 - 022 = 755
```

Set stricter mask:

```
umask 0077
```

Now:

- Files → 600
- Directories → 700

This is recommended for secure environments.

7. Privilege Escalation — `sudo`

Linux separates normal users from administrators.
Instead of logging in as root, use:

```
sudo command
```

Example:

```
sudo apt update
```

`sudo` temporarily grants administrative privileges.
Check current user:

```
whoami
```

Check effective user:

```
sudo whoami
```

You should see:

```
root
```

Why Avoid Root Login?

Logging in directly as root increases risk.

Best practice:

- Use `sudo`
- Log actions
- Limit privilege duration

8. `visudo` — Safely Editing `sudo` Configuration

The file controlling `sudo` access is:

```
/etc/sudoers
```

Never edit it directly with `nano` or `vim`.
Instead use:

```
sudo visudo
```

visudo:

- Validates syntax
- Prevents configuration errors
- Locks file during editing

Example Entry

```
alice ALL=(ALL) ALL
```

Meaning:

- User **alice**
- Can run any command
- On any host
- As any user

Grant group access:

```
%developers ALL=(ALL) ALL
```

The % indicates the group.

9. Practical Secure Configuration Example

Scenario:

You want a deployment user to restart the web server but not access everything.

Steps:

Edit sudoers:

```
sudo visudo
```

Add:

```
deploy ALL=(ALL) /bin/systemctl restart nginx
```

Now:

```
sudo systemctl restart nginx
```

Works for deploy.

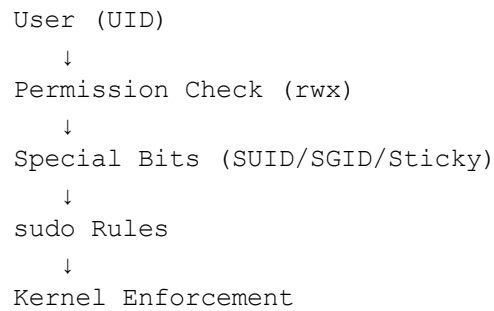
But:

```
sudo rm -rf /
```

Is denied.

This is controlled privilege escalation.

Security Topology



Linux security is layered and structured.

Real-World Scenario

Imagine a shared project directory:

```
sudo mkdir /project
sudo chown root:developers /project
sudo chmod 2775 /project
```

Now:

- Group inheritance is enforced (SGID)
- Only developers can access
- Others are denied

This is collaborative but secure.

Practical Lab

Try:

```
touch demo.txt
ls -l demo.txt
chmod 600 demo.txt
chmod 755 demo.txt
sudo chown root demo.txt
ls -l demo.txt
umask
sudo visudo
```

Reflect:

- What changes when you alter permissions?
- How does **umask** influence new files?
- What risks exist with SUID?

Chapter Summary

In this chapter, you learned:

- How to change permissions with **chmod**
- How to modify ownership with **chown**
- How to manage groups with **chgrp**
- How SUID, SGID, and Sticky Bit affect execution
- How **umask** defines default permissions
- How **sudo** enables controlled privilege escalation
- How **/etc/sudoers** defines administrative policy
- Why **visudo** must be used safely

Permissions and privilege control are at the heart of Linux security.

In the next chapter, we will explore package management in Ubuntu — where system configuration meets software control.

PART VI — PACKAGE & SYSTEM MANAGEMENT (Ubuntu Focus)

Chapter 11 — Package Management in Ubuntu

Ubuntu uses the **APT** (*Advanced Package Tool*) system for managing software distributed in **.deb** format. In addition, Ubuntu supports **Snap packages**, which provide sandboxed applications. Understanding both systems allows you to manage software confidently and professionally.

1. Understanding the APT Ecosystem

APT works on top of:

- **dpkg** → low-level package manager
- A repository system → centralized software sources
- Dependency resolution engine

Software in Ubuntu is stored in repositories defined in:

```
/etc/apt/sources.list  
/etc/apt/sources.list.d/
```

APT retrieves package metadata and downloads packages securely using GPG verification.

2. **apt update** — Refresh Package Index

Before installing or upgrading software, always refresh the package index:

```
sudo apt update
```

This command:

- Contacts configured repositories
- Downloads latest package lists
- Updates local metadata

It does **not** upgrade packages.

Think of it as refreshing the catalog before shopping.

3. `apt upgrade` — Upgrade Installed Packages

After updating the package index:

```
sudo apt upgrade
```

This upgrades installed packages to the latest available versions.

APT will:

- Resolve dependencies
- Download updated packages
- Replace older versions

For full system upgrade (including dependency changes):

```
sudo apt full-upgrade
```

Recommended Update Workflow

```
sudo apt update  
sudo apt upgrade
```

This ensures:

- Security patches are applied
- System remains stable
- Dependencies remain consistent

4. `apt install` — Installing Software

Install a package:

```
sudo apt install nginx
```

APT will:

- Identify dependencies
- Download required packages
- Configure software

You can install multiple packages:

```
sudo apt install git curl htop
```

Install Specific Version

```
sudo apt install nginx=1.22.0-1ubuntu1
```

This is useful for compatibility requirements.

5. `apt remove` — Removing Software

Remove package but keep configuration files:

```
sudo apt remove nginx
```

Completely remove including configuration:

```
sudo apt purge nginx
```

6. `apt autoremove` — Cleaning Dependencies

When removing software, some dependencies may remain unused.

Clean them:

```
sudo apt autoremove
```

This helps:

- Free disk space
- Keep system clean
- Avoid dependency clutter

7. `apt search` — Finding Packages

Search for packages:

```
apt search docker
```

Example output:

```
docker.io - Linux container runtime
docker-compose - Multi-container Docker applications
```

To view detailed package information:

```
apt show nginx
```

This displays:

- Version
- Dependencies
- Description
- Maintainer

8. Understanding **dpkg** Basics

dpkg is the low-level Debian package tool.
APT uses **dpkg** internally.

Install Local **.deb** File

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

If dependency errors occur:

```
sudo apt -f install
```

List Installed Packages

```
dpkg -l
```

Filter:

```
dpkg -l | grep nginx
```

Remove Package with **dpkg**

```
sudo dpkg -r nginx
```

Note:

- **dpkg** does not automatically resolve dependencies.
- APT is generally preferred for daily use.

9. Snap Package Management

Ubuntu supports Snap packages.

Snap advantages:

- Sandboxed applications
- Automatic updates
- Bundled dependencies
- Cross-distribution compatibility

Install Snap Package

```
sudo snap install code --classic
```

List Installed Snaps

```
snap list
```

Remove Snap

```
sudo snap remove code
```

Snap vs APT

Feature	APT (.deb)	Snap
Traditional system integration	Yes	Limited
Sandboxing	No	Yes
Automatic updates	Manual	Automatic
Startup speed	Faster	Slightly slower

Use Snap for:

- Desktop applications
- Isolated services
- Cross-platform tools

Use APT for:

- Core system software
- Libraries
- Server packages

10. DEB vs RPM — Conceptual Comparison

Linux distributions use different package systems.

Ubuntu and Debian use:

```
.deb (Debian package format)
```

Red Hat, Fedora, and CentOS use:

```
.rpm (Red Hat Package Manager format)
```

Key Differences

Feature	DEB	RPM
Default in Ubuntu	Yes	No
Managed by	APT/dpkg	DNF/YUM
File format	.deb	.rpm
Dependency resolution	APT	DNF/YUM

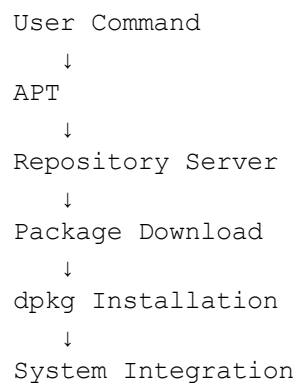
Conceptually:

- Both manage software
- Both track dependencies
- Both integrate with repositories
- Only format and management tools differ

Understanding both helps when working across Linux distributions.

11. Repository Topology

Conceptual model:



APT acts as the intelligent manager.
dpkg performs the actual installation.

Real-World Scenario

Deploy a web server:

```
sudo apt update
sudo apt install nginx
sudo systemctl enable nginx
sudo systemctl start nginx
```

Check installation:

```
systemctl status nginx
```

Remove later:

```
sudo apt remove nginx
sudo apt autoremove
```

This is lifecycle management.

Security Best Practices

- Always run **apt update** before upgrade
- Use official repositories
- Avoid random **.deb** downloads
- Monitor automatic updates
- Review installed packages periodically

List manually installed packages:

```
apt-mark showmanual
```

Practical Lab

Try:

```
sudo apt update
apt search curl
sudo apt install curl
dpkg -l | grep curl
sudo apt remove curl
sudo apt autoremove
snap list
```

Reflect:

- What dependencies were installed?
- What changed after autoremove?
- How do Snap packages differ?

Chapter Summary

In this chapter, you learned:

- How to refresh repositories with **apt update**
- Upgrade packages with **apt upgrade**
- Install software with **apt install**
- Remove packages with **apt remove**
- Clean unused dependencies with **apt autoremove**
- Search packages using **apt search**
- Understand **dpkg** basics
- Manage Snap packages
- Conceptual differences between **DEB** and **RPM**

Package management gives you control over the software lifecycle.

In the next chapter, we will explore networking tools in the Linux terminal — where systems connect to the world.

PART VII — INTRODUCTION TO SHELL SCRIPTING

Chapter 12 — Writing Simple Bash Scripts

A script is simply a text file containing commands that the shell executes sequentially. Instead of typing commands repeatedly, you write them once — and reuse them.

This chapter introduces:

- Script structure
- Variables
- Control flow
- Input handling
- Exit codes
- Debugging

You will learn not just syntax — but *thinking in automation*.

1. The Shebang (`#!/bin/bash`)

Every Bash script should begin with a **shebang**:

```
#!/bin/bash
```

This line tells the system:

```
"Use the Bash interpreter to execute this file."
```

Without it, the script may execute under a different shell.

Example script:

```
#!/bin/bash
echo "Hello, Ubuntu Academy!"
```

Save as:

```
hello.sh
```

2. Making Scripts Executable

After creating a script, make it executable:

```
chmod +x hello.sh
```

Then run it:

```
./hello.sh
```

The `./` indicates execution from the current directory.

Script Execution Flow

User → Shell → Interpreter (/bin/bash) → Commands → Output

This is how Linux executes scripts internally.

3. Variables and Expansion

Variables store data.

Assigning Variables

```
name="Linux"
```

No spaces around `=`.

Accessing Variables

```
echo $name
```

Or:

```
echo "${name}"
```

Using braces prevents ambiguity.

Example Script

```
#!/bin/bash
user="student"
echo "Welcome $user"
```

Output:

```
Welcome student
```

Command Substitution

```
today=$(date)
echo "Today is $today"
```

The shell executes `date` and stores the output.

4. Positional Parameters

Scripts can accept arguments.

Example:

```
#!/bin/bash
echo "First argument: $1"
echo "Second argument: $2"
```

Run:

```
./script.sh apple banana
```

Output:

```
First argument: apple
Second argument: banana
```

Important variables:

- `$0` → script name
- `$1`, `$2`, ... → arguments
- `$#` → number of arguments
- `$@` → all arguments

5. Conditional Statements — `if/then/else`

Decision-making is essential in scripting.

Basic syntax:

```
if [ condition ]; then
    command
else
    command
fi
```

Example: Check File Existence

```
#!/bin/bash

if [ -f "$1" ]; then
    echo "File exists"
else
    echo "File not found"
fi
```

Test:

```
./check.sh notes.txt
```

Common Test Operators

Option	Meaning
<code>-f</code>	File exists
<code>-d</code>	Directory exists
<code>-r</code>	Readable
<code>-w</code>	Writable
<code>-x</code>	Executable

6. `for` Loops

Loops allow repetition.

Basic `for` Loop

```
for file in *.txt; do
    echo "$file"
done
```

This loops through all `.txt` files.

Example: Batch Rename

```
for file in *.log; do
    mv "$file" "$file.bak"
done
```

This renames all `.log` files.

Loop Topology

List → Iterate → Execute → Next → Complete

7. `while` Loops

Used for conditional repetition.

Example

```
count=1

while [ $count -le 5 ]; do
    echo "Count: $count"
    ((count++))
done
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Reading File Line-by-Line

```
while read line; do
    echo "$line"
done < file.txt
```

This is useful for processing structured data.

8. case Statement

Used for multiple conditions.

Syntax:

```
case $variable in
    pattern1)
        command
        ;;
    pattern2)
        command
        ;;
    *)
        default
        ;;
esac
```

Example

```
#!/bin/bash

read -p "Enter option (start/stop): " action

case $action in
    start)
        echo "Starting service..."
        ;;
    stop)
        echo "Stopping service..."
        ;;
    *)
        echo "Invalid option"
        ;;
esac
```

case improves readability compared to multiple **if** statements.

9. read — Taking User Input

Example:

```
read -p "Enter your name: " username
echo "Hello, $username"
```

The `-p` flag displays a prompt.
This enables interactive scripts.

10. Exit Codes

Every command returns an exit status.

- 0 → success
- Non-zero → error

Check exit code:

```
echo $?
```

Example:

```
ls file.txt
echo $?
```

Use in script:

```
if [ $? -eq 0 ]; then
    echo "Command successful"
fi
```

Better approach:

```
if ls file.txt; then
    echo "Exists"
else
    echo "Missing"
fi
```

11. Debugging with `set -x`

Debugging helps track script execution.
Add at top of script:

```
set -x
```

Or run:

```
bash -x script.sh
```

This prints commands before execution.
Example:

```
+ echo Hello  
Hello
```

Remove debugging with:

```
set +x
```

12. Example Complete Script

System Information Script:

```
#!/bin/bash  
  
echo "System Report"  
echo "-----"  
echo "Hostname: $(hostname) "  
echo "Date: $(date) "  
echo "Uptime: $(uptime -p) "  
echo "Memory Usage:"  
free -h
```

Make executable:

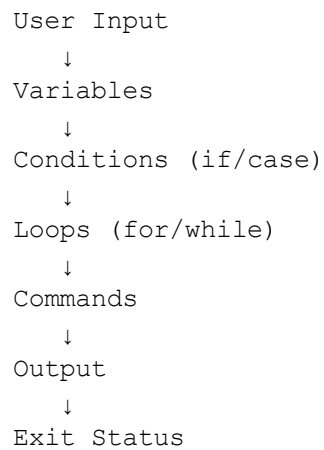
```
chmod +x system_report.sh
```

Run:

```
./system_report.sh
```

You have created a reusable diagnostic tool.

Automation Topology



This is structured automation logic.

Real-World Example

Backup script:

```
#!/bin/bash

backup_dir="/backup"
source_dir="/home/student"

mkdir -p "$backup_dir"
tar -czf "$backup_dir/backup_$(date +%F).tar.gz" "$source_dir"

if [ $? -eq 0 ]; then
    echo "Backup successful"
else
    echo "Backup failed"
fi
```

This demonstrates:

- Variables
- Command substitution
- Conditional logic
- Exit code handling

Practical Lab

Try:

```
nano test.sh
chmod +x test.sh
./test.sh
```

Add:

- A loop
- A condition
- User input
- Debugging

Experiment and modify.

Chapter Summary

In this chapter, you learned:

- How to structure scripts with **shebang**
- Make scripts executable using **chmod +x**
- Use variables and expansion
- Accept positional parameters
- Implement **if/then/else**
- Use **for** and **while** loops
- Apply **case** statements
- Read user input
- Understand exit codes
- Debug with **set -x**

You have now crossed the line from command user to automation creator.

In the next chapter, we will explore advanced Bash features and script best practices.

PART VIII — PRODUCTIVITY & BEST PRACTICES

Chapter 13 — Becoming Efficient in the Terminal

The Linux terminal is designed for power users. Once you understand its shortcuts and customization mechanisms, your productivity increases dramatically.

Efficiency comes from three principles:

1. **Reuse what you typed**
2. **Let the shell complete for you**
3. **Customize your environment**

1. Command History

Every command you execute is stored in history.

View history:

```
history
```

Example output:

```
1001  ls -l
1002  cd /var/log
1003  sudo apt update
```

The history file is stored in:

```
~/.bash_history
```

Navigate with Arrow Keys

- ↑ → previous command
- ↓ → next command

This alone saves time.

Limit History Size

Check:

```
echo $HISTSIZE
```

Increase history capacity by editing:

```
nano ~/.bashrc
```

Add:

```
HISTSIZE=5000
HISTFILESIZE=10000
```

Apply changes:

```
source ~/.bashrc
```

2. Reverse Search — **Ctrl + R**

One of the most powerful features.

Press:

```
Ctrl + R
```

Start typing part of a previous command.

Example:

```
(reverse-i-search)`apt': sudo apt update
```

Press **Ctrl + R** repeatedly to cycle backward.

This is dramatically faster than scrolling through history.

3. History Expansion — **!!** and **!n**

Bash supports history expansion shortcuts.

Repeat Last Command — **!!**

```
!!
```

Example:

```
apt update
sudo !!
```

This becomes:

```
sudo apt update
```

Very useful when you forget **sudo**.

Run Specific History Command — **!n**

```
!1003
```

Executes command number 1003 from history.

Run Most Recent Matching Command

```
!apt
```

Executes the last command starting with "apt".

⚠ Use carefully — always verify before execution.

4. Tab Completion

Tab completion saves time and prevents typos.

File Completion

```
cd Doc<Tab>
```

Expands to:

```
Documents/
```

Command Completion

```
apt ins<Tab>
```

Expands to:

```
apt install
```

Multiple Possibilities

Press Tab twice to display options:

```
ls /et<Tab><Tab>
```

Shows:

```
/etc/
```

Tab completion works because of Bash's programmable completion.

5. Aliases — Creating Shortcuts

Aliases create custom shortcuts.

Temporary Alias

```
alias ll='ls -lah'
```

Now:

```
ll
```

Executes:

```
ls -lah
```

Permanent Alias

Edit:

```
nano ~/.bashrc
```

Add:

```
alias update='sudo apt update && sudo apt upgrade'
```

Reload:

```
source ~/.bashrc
```

Now typing:

```
update
```

Performs a full update process.

List Aliases

```
alias
```

6. Customizing `~/.bashrc`

The file:

```
~/.bashrc
```

Contains:

- Aliases
- Environment variables
- Shell behavior settings

Example configuration:

```
# Custom prompt
PS1='\u@\h:\w\$ '

# Add custom bin directory
export PATH="$HOME/bin:$PATH"

# Useful aliases
alias gs='git status'
alias cls='clear'
```

After modification:

```
source ~/.bashrc
```

Prompt Customization Example

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

This creates a colored prompt.

7. Environment Variables

Environment variables define system behavior.

View All Variables

```
printenv
```

Or:

```
env
```

Important Variables

PATH

```
echo $PATH
```

Example:

```
/usr/local/bin:/usr/bin:/bin
```

The shell searches these directories for commands.
Add custom directory:

```
export PATH="$HOME/scripts:$PATH"
```

EDITOR

```
echo $EDITOR
```

Set default editor:

```
export EDITOR=nano
```

This affects programs that rely on a default editor.

Persistent Environment Variables

Add to:

```
~/.bashrc
```

Or system-wide:

```
/etc/environment
```

8. Troubleshooting Common Terminal Issues

Even professionals encounter terminal issues.

Command Not Found

Error:

```
command not found
```

Possible causes:

- Not installed
- Not in PATH

Check:

```
which command  
echo $PATH
```

Permission Denied

```
Permission denied
```

Possible reasons:

- Not executable
- Insufficient privileges

Fix:

```
chmod +x script.sh  
sudo command
```

Terminal Freeze

Press:

```
Ctrl + C
```

To terminate the current process.

Or:

```
Ctrl + Z
```

To suspend.

Accidental Output to File Instead of Screen

If you run:

```
command > file
```

You may not see output.

Check file contents:

```
cat file
```

Broken Terminal Display

If characters look corrupted:

```
reset
```

This restores terminal settings.

Efficiency Workflow Model

```
Command
  ↓
History
  ↓
Shortcut (!! , !n)
  ↓
Tab Completion
  ↓
Alias
  ↓
Customization (~/.bashrc)
  ↓
Optimized Workflow
```

Efficiency builds layer by layer.

Real-World Efficiency Example

Instead of typing:

```
sudo apt update && sudo apt upgrade
```

Create alias:

```
alias upgradeall='sudo apt update && sudo apt upgrade'
```

Now:

```
upgradeall
```

Time saved daily becomes hours saved yearly.

Practical Lab

Try:

```
history
Ctrl + R
alias ll='ls -lah'
echo $PATH
nano ~/.bashrc
source ~/.bashrc
```

Experiment with:

- Creating at least 3 aliases
- Modifying your prompt
- Adding a custom directory to PATH

Productivity Mindset

- Beginners type commands.
- Intermediate users reuse commands.
- Advanced users customize the shell.
- Experts automate everything.

Chapter Summary

In this chapter, you learned:

- How to use command history
- Reverse search with **Ctrl + R**
- History expansion (**!!**, **!n**)
- Tab completion
- Creating aliases
- Editing `~/ .bashrc`
- Understanding environment variables (`PATH`, `EDITOR`)
- Troubleshooting common terminal issues

You now understand not only how Linux works — but how to work efficiently inside it. This concludes the core terminal mastery section.

From here, you are ready to move into advanced automation, DevOps workflows, or system administration specialization.

APPENDICES

Appendix A — Command Reference by Topic

1. File Management

Command	Purpose	Basic Syntax	Common Options	Example
ls	List directory contents	ls [options]	-l, -a, -h	ls -lah
pwd	Print working directory	pwd	—	pwd
cd	Change directory	cd path	.., ~	cd /etc
mkdir	Create directory	mkdir dir	-p	mkdir -p project/src
rmdir	Remove empty directory	rmdir dir	—	rmdir olddir
cp	Copy files	cp src dest	-r, -p	cp -rp folder backup/
mv	Move/rename files	mv src dest	—	mv file.txt new.txt
rm	Remove files	rm file	-r, -f	rm -rf temp/
ln	Create links	ln target link	-s	ln -s file link
stat	File metadata	stat file	—	stat file.txt
file	Identify file type	file filename	—	file image.jpg

Important Concept: File ownership and permissions determine access control.

2. File Search

Command	Purpose	Syntax	Common Options	Example
<code>find</code>	Search filesystem	<code>find path criteria</code>	<code>-name, -type, -size</code>	<code>find / -name "*.conf"</code>
<code>locate</code>	Search via database	<code>locate pattern</code>	—	<code>locate passwd</code>
<code>which</code>	Locate executable	<code>which command</code>	—	<code>which python3</code>
<code>whereis</code>	Locate binary/man	<code>whereis command</code>	—	<code>whereis bash</code>
<code>grep</code>	Search inside files	<code>grep pattern file</code>	<code>-r, -i, -v</code>	<code>grep -r error /var/log</code>
<code>diff</code>	Compare text files	<code>diff file1 file2</code>	—	<code>diff old new</code>
<code>cmp</code>	Compare binary files	<code>cmp file1 file2</code>	—	<code>cmp img1 img2</code>

Tip: Combine `find` and `grep` for powerful filtering.

3. Text Processing

Command	Purpose	Syntax	Options	Example
<code>cat</code>	Display file	<code>cat file</code>	<code>-n</code>	<code>cat -n file.txt</code>
<code>less</code>	View interactively	<code>less file</code>	—	<code>less logfile.txt</code>
<code>head</code>	First lines	<code>head file</code>	<code>-n</code>	<code>head -n 20 file</code>
<code>tail</code>	Last lines	<code>tail file</code>	<code>-f</code>	<code>tail -f log</code>
<code>wc</code>	Count lines/words	<code>wc file</code>	<code>-l, -w</code>	<code>wc -l file</code>
<code>sort</code>	Sort lines	<code>sort file</code>	<code>-n, -r</code>	<code>sort -n data</code>
<code>uniq</code>	Remove duplicates	<code>uniq file</code>	<code>-c, -d</code>	<code>`sort file</code>
<code>cut</code>	Extract fields	<code>cut -d: -f1 file</code>	—	<code>cut -d: -f1 /etc/passwd</code>
<code>tr</code>	Translate chars	<code>tr set1 set2</code>	<code>-d</code>	<code>`echo hi</code>
<code>column</code>	Format columns	<code>column -t</code>	<code>-s</code>	<code>column -t -s : file</code>
<code>tee</code>	Output to file + screen	<code>tee file</code>	<code>-a</code>	<code>`ls</code>

Core Idea: Text is the universal data format in Linux.

4. Compression & Archiving

Command	Purpose	Syntax	Example
tar	Archive files	tar -cvf archive.tar dir	tar -czvf backup.tar.gz folder
gzip	Compress file	gzip file	gzip data.log
gunzip	Decompress	gunzip file.gz	gunzip data.log.gz
zip	Zip archive	zip file.zip file	zip docs.zip *.txt
unzip	Extract zip	unzip file.zip	unzip docs.zip

Common flags:

- -c → create
- -x → extract
- -Z → gzip compression
- -v → verbose

5. Process Management

Command	Purpose	Example
ps aux	Show processes	ps aux
top	Live monitoring	top
htop	Advanced monitor	htop
pgrep	Find process	pgrep ssh
kill	Send signal	kill PID
pkill	Kill by name	pkill nginx
nice	Set priority	nice -n 10 script.sh
renice	Change priority	renice 5 -p PID
jobs	List background jobs	jobs
nohup	Run after logout	nohup script.sh &

Signal Types:

- **SIGTERM (15)** → Graceful stop
- **SIGKILL (9)** → Force stop

6. User Management

Command	Purpose	Example
<code>whoami</code>	Current user	<code>whoami</code>
<code>id</code>	User identity	<code>id</code>
<code>useradd</code>	Create user	<code>sudo useradd -m user</code>
<code>usermod</code>	Modify user	<code>sudo usermod -aG sudo user</code>
<code>passwd</code>	Set password	<code>sudo passwd user</code>
<code>userdel</code>	Delete user	<code>sudo userdel -r user</code>
<code>groupadd</code>	Create group	<code>sudo groupadd dev</code>
<code>gpasswd</code>	Manage group	<code>sudo gpasswd -a user dev</code>

System files:

- `/etc/passwd`
- `/etc/shadow`
- `/etc/group`

7. Ubuntu Administration

Command	Purpose	Example
<code>apt update</code>	Refresh package list	<code>sudo apt update</code>
<code>apt upgrade</code>	Upgrade packages	<code>sudo apt upgrade</code>
<code>apt install</code>	Install package	<code>sudo apt install nginx</code>
<code>apt remove</code>	Remove package	<code>sudo apt remove nginx</code>
<code>apt autoremove</code>	Clean dependencies	<code>sudo apt autoremove</code>
<code>dpkg -l</code>	List packages	<code>dpkg -l</code>
<code>snap install</code>	Install Snap	<code>sudo snap install code</code>
<code>systemctl</code>	Manage services	<code>sudo systemctl status ssh</code>
<code>journalctl</code>	View logs	<code>journalctl -u ssh</code>
<code>df -h</code>	Disk usage	<code>df -h</code>
<code>free -h</code>	Memory usage	<code>free -h</code>

Conceptual Command Topology

File System → File Commands

Text → Text Processing Tools

Processes → Process Management

Users → Identity & Permissions

Packages → System Administration

Logs → Monitoring & Debugging

Linux commands are modular tools.
Each tool does one job well.

How to Use This Appendix

When troubleshooting:

1. Identify the category (file, process, user, package).
2. Locate relevant commands.
3. Combine commands using pipelines.

Example workflow:

```
find /var/log -name "*.log" | grep error
```

This combines *file search* and *text processing*.

Final Note

This appendix is designed to be:

- **A daily reference**
- A quick troubleshooting guide
- A memory reinforcement tool

Mastery of Linux comes not from memorizing commands — but from understanding how they connect.

Use this reference often.

Practice combining commands.

Build your own workflows.

You are now equipped to operate Linux with confidence.

Appendix B — Practice Labs (Exam-Style Scenarios)

This section is designed to simulate **exam-style practical challenges**. Each scenario requires you to combine commands from multiple chapters.

These labs are:

- Realistic
- Troubleshooting-oriented
- Performance-focused
- Security-aware

Do not simply read them — execute them.

Each scenario includes:

- Situation
- Objective
- Tools to use
- Step-by-step solution
- Reflection questions

Think like a system administrator.

Lab 1 — Freeing Disk Space

Scenario

A server reports:

```
Filesystem is 100% full.
```

Users cannot log in, and applications are failing.

Objective

Identify what is consuming space and safely free disk capacity.

Step 1 — Verify Disk Usage

```
df -h
```

Look for filesystems near 100%.

Step 2 — Identify Large Directories

Navigate to root:

```
cd /
```

Check usage:

```
sudo du -sh * | sort -h
```

Drill down:

```
cd /var
sudo du -sh * | sort -h
```

Common large locations:

- `/var/log`
- `/var/cache`
- `/home`

Step 3 — Clear Logs (Carefully)

List large logs:

```
sudo du -sh /var/log/*
```

Truncate a large log:

```
sudo truncate -s 0 /var/log/syslog
```

Or remove rotated logs:

```
sudo rm /var/log/*.gz
```

Step 4 — Clean Package Cache

```
sudo apt clean
sudo apt autoremove
```

Topology of Disk Investigation

```
Disk Full
↓
df -h
↓
du -sh
↓
Identify Directory
↓
Remove / Clean
```

Reflection

- Which directory was largest?
- Could this issue have been prevented?
- Should log rotation be configured?

Lab 2 — Finding Suspicious Files

Scenario

You suspect a user may have uploaded unauthorized scripts.

Objective

Locate potentially dangerous files.

Step 1 — Search for Executables in Home

```
find /home -type f -executable
```

Step 2 — Search Recently Modified Files

```
find /home -type f -mtime -2
```

Find files modified within last 2 days.

Step 3 — Look for SUID Files

```
find / -perm -4000 -type f 2>/dev/null
```

SUID files may indicate privilege escalation risks.

Step 4 — Inspect Hidden Files

```
ls -la /home/username
```

Hidden files begin with `.`

Step 5 — Search for Suspicious Strings

```
grep -r "nc -e" /home
```

Search for reverse shell patterns.

Investigation Flow

```
Suspicion
  ↓
Search by Time
  ↓
Search by Permission
  ↓
Search by Content
  ↓
Review Ownership
```

Reflection

- Which files were executable?
- Were any owned by root unexpectedly?
- Did any contain suspicious network commands?

Lab 3 — Killing Runaway Processes

Scenario

CPU usage is at 100%. The system is slow.

Objective

Identify and stop the runaway process.

Step 1 — Check Load

```
uptime
```

Step 2 — Monitor in Real Time

```
top
```

Sort by CPU (P).

Step 3 — Identify Process

Note the PID.

Example:

```
PID 3456 using 95% CPU
```

Step 4 — Attempt Graceful Termination

```
kill 3456
```

Step 5 — Force Kill if Needed

```
kill -9 3456
```

Step 6 — Prevent Recurrence

Check parent process:

```
ps -ef | grep 3456
```

Check if started by cron or service.

Process Control Flow

```
High Load
  ↓
top
  ↓
Identify PID
  ↓
kill
  ↓
Verify
```

Reflection

- Was the process user-initiated?
- Did killing it impact other services?
- Should limits (nice) be applied?

Lab 4 — Creating Restricted Users

Scenario

You must create a user who:

- Can log in
- Cannot use sudo
- Has restricted shell access

Step 1 — Create User

```
sudo useradd -m restricted
```

Step 2 — Set Password

```
sudo passwd restricted
```

Step 3 — Assign Limited Shell

Change shell:

```
sudo usermod -s /usr/sbin/nologin restricted
```

Or restrict via:

```
sudo usermod -s /bin/rbash restricted
```

Step 4 — Remove from sudo Group

```
sudo gpasswd -d restricted sudo
```

Step 5 — Restrict Directory Access

```
sudo chmod 700 /home/restricted
```

Access Control Topology

```
User Created
  ↓
Group Assignment
  ↓
Shell Restriction
  ↓
Permission Enforcement
```

Reflection

- Can restricted user access system files?
- What happens if shell is set to nologin?
- How can access be audited?

Lab 5 — Writing a Simple Backup Script

Scenario

You must create a daily backup of `/home`.

Objective

Automate backup with logging.

Step 1 — Create Script

```
nano backup.sh
```

Step 2 — Write Script

```
#!/bin/bash

SOURCE="/home"
DEST="/backup"
DATE=$(date +%F)

mkdir -p "$DEST"

tar -czf "$DEST/home_$(date +%F).tar.gz" "$SOURCE" &>
"$DEST/backup_$(date +%F).log"

if [ $? -eq 0 ]; then
    echo "Backup successful"
else
    echo "Backup failed"
fi
```

Step 3 — Make Executable

```
chmod +x backup.sh
```

Step 4 — Run

```
./backup.sh
```

Step 5 — Automate with Cron (Optional Advanced)

Edit crontab:

```
crontab -e
```

Add:

```
0 2 * * * /path/to/backup.sh
```

This runs backup daily at 2 AM.

Backup Flow

```
Define Variables
↓
Create Directory
↓
Archive with tar
↓
Log Output
↓
Check Exit Code
```

Reflection

- Where are logs stored?
- What happens if the disk is full?
- How can backups be verified?

Final Mastery Challenge

Combine multiple skills:

1. Create a new user
2. Restrict permissions
3. Install required packages
4. Monitor resource usage
5. Write and schedule a backup
6. Verify logs

This simulates real certification exams.

Closing Advice

These labs are not just exercises.

They represent:

- Real server incidents
- Real security concerns
- Real performance problems

Mastery of Linux comes from solving real problems under pressure.

Practice repeatedly.

Combine commands creatively.

Challenge yourself.

You are now ready to operate Linux with confidence — not as a user, but as a system professional.

Appendix C — Glossary

Term	Definition	Technical Explanation	Example Command	Why It Matters
APT	Advanced Package Tool.	Ubuntu package manager handling repositories and dependency resolution.	<code>sudo apt update</code>	Central to Ubuntu software management.
Bash	GNU Bourne Again Shell.	Default Ubuntu shell supporting scripting, job control, and expansion.	<code>bash --version</code>	Foundation for automation and scripting.
command substitution	Insert command output into another command.	Uses <code>\$(command)</code> syntax to embed command output.	<code>echo \$(date)</code>	Enables dynamic scripting and automation.
daemon	Background service process.	Runs independently of user sessions, usually managed by <i>systemd</i> .	<code>systemctl status ssh</code>	Essential for understanding server services.
dpkg	Debian package manager.	Low-level <code>.deb</code> installer without automatic dependency resolution.	<code>dpkg -l</code>	Underlying engine of Ubuntu package management.
environment variable	Dynamic variable affecting process behavior.	Key-value pairs inherited by child processes	<code>echo \$PATH</code>	Controls execution environment and command lookup.

		(e.g., <code>PATH</code> , <code>HOME</code>).		
FHS (Filesystem Hierarchy Standard)	Standard Linux directory structure.	Defines the purpose of top-level directories such as <code>/etc</code> , <code>/var</code> , <code>/usr</code> .	<code>ls /etc</code>	Ensures consistent system organization across Linux systems.
GID (Group ID)	Numeric group identifier.	Defines group membership and shared permissions.	<code>id</code>	Enables collaborative permission management.
globbing	Wildcard pattern expansion.	Shell expands <code>*</code> , <code>?</code> , <code>[]</code> before command execution.	<code>ls *.txt</code>	Automates file matching and selection.
inode	Filesystem metadata structure.	Stores file permissions, ownership, timestamps, size, and block pointers — but not filename.	<code>ls -li file.txt</code>	Explains hard links and file deletion behavior.
journalctl	systemd log viewer.	Displays logs stored in the system journal.	<code>journalctl -u ssh</code>	Essential for troubleshooting services.
load average	System workload measurement.	Average number of processes waiting for CPU over time intervals.	<code>uptime</code>	Indicates system performance health.
PID (Process ID)	Unique process identifier.	Assigned by kernel to each running process.	<code>ps aux</code>	Required for process monitoring and control.

pipeline	Command output chaining mechanism.	The <code>`</code>	<code>`</code> operator connects output of one command to input of another.	<code>`grep error log</code>
PPID (Parent Process ID)	Parent process identifier.	Shows process hierarchy relationship.	<code>ps -ef</code>	Helps trace process origins.
process state	Current execution status of a process.	Common states: R (Running), S (Sleeping), T (Stopped), Z (Zombie).	<code>ps aux</code>	Helps diagnose performance issues.
repository	Software package source server.	Remote server containing package metadata and binaries.	<code>/etc/apt/sources.list</code>	Determines where software updates originate.
SGID (Set Group ID)	Special group permission bit.	Files created in directory inherit group ownership.	<code>chmod 2755 directory</code>	Useful for shared team directories.
shebang	Script interpreter declaration.	First line of script (e.g., <code>#!/bin/bash</code>) specifying interpreter.	<code>#!/bin/bash</code>	Ensures consistent script execution.
shell	Command interpreter interface.	Program (e.g., <i>Bash</i>) that reads and executes commands.	<code>echo \$SHELL</code>	Core interface between user and system.
signal	Software interrupt sent to a process.	Allows kernel or users to control processes (e.g., <i>SIGTERM</i> , <i>SIGKILL</i>).	<code>kill -15 PID</code>	Enables controlled process management.

Snap	Universal Linux package format.	Sandboxed, self-contained application packaging system.	<code>snap install code</code>	Enables cross-distribution deployment.
SUID (Set User ID)	Special executable permission bit.	Program runs with file owner's privileges.	<code>chmod 4755 program</code>	Can introduce privilege escalation risks.
stdin (Standard Input)	Input data stream (file descriptor 0).	Receives input passed to commands.	<code>wc -l < file.txt</code>	Enables automation and input redirection.
stderr (Standard Error)	Error output stream (file descriptor 2).	Separates error messages from normal output.	<code>ls /noexist 2> error.txt</code>	Critical for debugging and logging.
stdout (Standard Output)	Normal output stream (file descriptor 1).	Displays standard command output.	<code>ls > output.txt</code>	Allows output redirection and pipelines.
Sticky Bit	Directory protection mechanism.	Prevents users from deleting files owned by others in shared directories.	<code>chmod 1777 directory</code>	Secures shared spaces like <code>/tmp</code> .
systemd	Modern Linux init system.	PID 1; manages services and targets.	<code>systemctl status</code>	Controls service lifecycle and boot behavior.
UID (User ID)	Numeric user identifier.	Unique number assigned to each user account.	<code>id username</code>	Determines file ownership and permissions.

umask

Default
permission
mask.

Subtracts
permissions
from newly
created files
and
directories.

umask 022

Defines
baseline
security
behavior.

References

Basler, M. (2025). *The Linux terminal for beginners: The command line made easy* (1st ed.). Rowena Bongcaron.

Blum, R., & Bresnahan, C. (2026). *Linux® command line and shell scripting bible* (5th ed.). John Wiley & Sons, Inc.

Kofler, M. (2025). *Linux command reference: Shell commands from A to Z*. Rheinwerk Computing.

Negus, C. (2026). *Linux bible* (11th ed.). John Wiley & Sons, Inc.

Shotts, W. (2026). *The Linux command line: A complete introduction* (3rd ed.). No Starch Press.

AUTHOR



Onno W. Purbo currently serves as Rector at Institut Teknologi Tangerang Selatan (ITTS). He earned his Ph.D. in Electrical Engineering from the University of Waterloo, Canada. He is a copyleft advocate, educator, and ICT evangelist. He has published more than 50 books, including free ICT e-books for schools in 2008. Some of his latest books include *"Internet-TCP/IP: Konsep Dan Implementasi"* (2018); *"Sistem Operasi, Konsep Dan Membuat Linux OpenWRT Dan ROM Android"* (2019); *"IPv6 Untuk Mendukung Operasi Jaringan Dan Domain Name System"* (2019); *"Kubernetes untuk Pemula"* (2024); and *"Membuat Operator Seluler 5G Sendiri"* (2024).

He led the first Internet connection at Institut Teknologi Bandung from 1993–2000 and used it to establish Indonesia's first educational Internet network. He played a key role in liberating WiFi frequencies, introducing RT/RW-net, the Wajanbolic antenna, as well as OpenBTS-based cellular networks and private 5G networks. He also led the first telephone network over the Internet, VoIP Merdeka, which later became known as VoIP Rakyat based on SIP and using the area codes +62520 and +62521.

He is currently active in promoting e-Learning and operates an e-Learning server at <http://lms.onnocenter.or.id/moodle/> serving more than 66,000 students.