LINUX ADMINISTRATION

Muhammad Ayman

Linux Administration

Table of Contents

What is Linux?

- **Linux kernel:** The core of the operating system, first released in 1991 by Linus Torvalds while he was a student at the University of Helsinki.
- **Open-source model:** Licensed under the GNU General Public License (GPL), meaning anyone can use, modify, and distribute it freely.
- **Unix-like foundation:** Inspired by MINIX and Unix, designed for stability, multitasking, and multi-user environments.
- **Distributions (distros):** Variants like Ubuntu, Fedora, Debian, and Arch Linux combine the kernel with software packages to form complete operating systems.
- **Cross-platform support:** Runs on PCs, servers, mobile devices (Android is Linux-based), embedded systems, and even supercomputers.

What is a Linux Distribution?

- **Complete operating system:** Combines the Linux kernel with system libraries, utilities, and often a graphical interface.
- **Package manager:** Each distro uses its own system for installing and updating software (e.g., APT, DNF, Pacman).
- **Default environment:** Distros ship with different desktop environments (GNOME, KDE, XFCE) or server-focused setups.
- **Target audience:** Some are beginner-friendly, others are designed for advanced users or specific industries.
- **Philosophy and goals:** Distros vary in their approach—some prioritize stability, others cutting-edge features, or minimalism.

Key Differences Between Popular Distros

Distro	Package Manager	Strengths	Target Users
Ubuntu	[APT (Debian-based)	Beginner-friendly, large community	New users, desktop computing
Debian	APT	Stability, reliability	Servers, conservative users
Fedora	DNF (RPM-based)	Cutting-edge features, Red Hat upstream	Developers, testers
Arch Linux	Pacman	Rolling release, full customization	Advanced users, tinkerers
openSUSE	Zypper	Enterprise-ready, YaST configuration tool	System admins, enterprises

CentOS / RHEL	YUM/DNF	Enterprise stability, long-term support	Businesses, production servers
Gentoo	Portage	Extreme customization, source-based	Power users, performance enthusiasts

What Exactly is the Terminal?

- **Program that connects to the shell**: The terminal itself is not the shell—it's a window or application that lets you type commands. The shell (like Bash or Zsh) interprets those commands.
- **Direct system access:** Unlike graphical user interfaces (GUIs), the terminal gives you precise control over files, processes, and configurations.
- **Efficiency and automation:** You can chain commands, use scripts, and automate repetitive tasks far faster than clicking through menus.
- **Universal tool:** Available on Linux, macOS, and even Windows (via PowerShell or WSL), making it a cross-platform skill.

Linux Commands

If you want to know all command's options, write `--help` after the command.
If you want to know the manual docs for any command, use `man` before the command

File and Directory Management

Command	Options	Purpose	Example
`pwd`	(none)	Show current directory	`pwd`
`ls`	-l (details), -a (hidden), -h (human-readable)	List files and directories	`ls -lah`
`cd`	(none)	Change directory	`cd /home/user/docs`
`mkdir`	-p (nested), -v (verbose)	Create directory	`mkdir -pv projects/code`
`rmdir`	(none)	Remove empty directory	`rmdir testdir`
`touch`	(none)	Create empty file	`touch notes.txt`

`rm`	-r (recursive), -f (force), -i (interactive)	Remove files/directories	`rm -rf folder`
`cp`	-r (recursive), -i (prompt), -v (verbose)	Copy files/directories	`cp -rv src/ backup/`
`mv`	-i (prompt), -v (verbose)	Move/rename files	`mv -iv old.txt new.txt`
`tree`	-L (depth), -d (directories only)	Show directory structure	`tree -L 2`

File Viewing and Editing

Command	Options	Purpose	Example
`cat`	-n (number lines), -b (number non-empty lines)	View file contents	`cat -n notes.txt`
`less`	-N (show line numbers)	Scroll through file	`less -N bigfile.txt`
`head`	-n (number of lines)	Show first lines	`head -n 20 file.txt`
`tail`	-n (lines), -f (follow live)	Show last lines	`tail -f logfile.log`
`nano`	(none)	Simple text editor	`nano notes.txt`
`vim`	-R (read-only)	Advanced text editor	`vim -R script.sh`
`wc`	-l (count lines only), -w (count words only), -c (count bytes), -m (count character)	Count text elements	`wc -w article.txt`

Process and System Management

Command	Options	Purpose	Example
`ps`	aux (detailed), -e (all processes)	Show running processes	`ps aux`
`top`	-u (filter by user)	Real-time process monitor	`top -u root`

htop	(interactive)	Interactive process viewer	`htop`
kill	-9 (force kill)	Terminate process	`kill -9 1234`
jobs	(none)	Show background jobs	`jobs`
bg	(none)	Resume job in background	`bg %1`
fg	(none)	Resume job in foreground	`fg %1`

Networking

Command	Options	Purpose	Example
ping	-c (count), -i (interval)	Test connectivity	`ping -c 4 google.com`
curl	-O (save file), -I (headers)	Transfer data from URLs	`curl -I https://example.com`
wget	-c (resume), -q (quiet)	Download files	`wget https://example.com/file.zip`
ssh	-p (port), -i (identity file)	Remote login	`ssh -p 2222 user@server`
scp	-r (recursive)	Copy files over SSH	`scp -r folder user@server:/path/`
netstat	-tuln (list ports)	Show network connections	`netstat -tuln`
ip	addr, route	Show/manage interfaces	`ip addr show`

Permissions and Ownership

Command	Options	Purpose	Example
chmod	+x (add execute), -r (remove read), numeric modes	Change permissions	`chmod 755 script.sh`
chown	user:group	Change owner	`chown root:root file.txt`
chgrp	(none)	Change group ownership	`chgrp staff file.txt`
umask	(none)	Set default permissions	`umask 022`

Package Management

Command	Options	Purpose	Example
apt	install, remove, update, upgrade	Debian/Ubuntu package manager	`apt install nginx`
dnf	install, remove, update	Fedora/Red Hat package manager	`dnf update`

`pacman`	-S (install), -R (remove), -U (upgrade)	Arch Linux package manager	`pacman -S firefox`
`snap`	install, remove, list	Universal package manager	`snap install vlc`

System Monitoring and Info

Command	Options	Purpose	Example
`df`	-h (human-readable)	Disk usage	`df -h`
`du`	-s (summary), -h (human-readable)	Directory/file size	`du -sh folder/`
`free`	-h (human-readable)	Memory usage	`free -h`
`uptime`	*(none)*	System running time	`uptime`

Using the sudo Command in Linux

sudo (short for "**superuser do**") is a command-line utility in Linux that allows a permitted user to execute commands with elevated privileges, typically as the root user. It is essential for performing administrative tasks securely without logging in as root.

Key Features:

- Grants temporary administrative access.
- Requires the user's password for authentication.
- Logs all usage for auditing.
- Only works for users listed in the sudoers file.

Common Use Cases:

- `sudo apt update` – Update system packages.
- `sudo reboot` – Restart the system.
- `sudo nano /etc/hosts` – Edit system configuration files.
- `sudo useradd muhammed` – Add a new user.
- `sudo chmod 755 script.sh` – Change file permissions.

Tips:

- Use `sudo !!` to re-run the last command with sudo.
- Use `sudo -u username command` to run as a different user.
- Use `sudo -k` to reset the sudo timer and require re-authentication.

Text Editor

vi Text Editor

vi (short for *visual*) is a classic text editor available by default on most Unix-like operating systems, including Linux. It was developed in the late 1970s and remains widely used due to its speed, reliability, and availability even on minimal systems.

Key Features:

- **Modal editing**: vi operates in multiple modes:
  - **Normal mode:** for navigation and commands.
  - **Insert mode:** for typing and editing text.
  - **Command-line mode:** for saving, quitting, and searching.
- **Lightweight and fast**: Runs in terminal environments with minimal resources.
- **Always available**: Found on nearly every Linux distribution by default.

Basic Usage:

- **Open a file:** `vi filename.txt`
- **Enter insert mode:** Press i
- **Exit insert mode:** Press Esc
- **Save and quit:** Type `:wq` then press Enter
- **Quit without saving:** Type `:q!` then press Enter

Common Commands:

Mode	Command	Action
Normal	`i`	Enter insert mode before cursor
Normal	`a`	Enter insert mode after cursor
Normal	`x`	Delete character under cursor
Normal	`dd`	Delete current line
Normal	`yy`	Copy (yank) current line
Normal	`p`	Paste after cursor
Command-line	`:w`	Save file
Command-line	`:q`	Quit
Command-line	`:wq`	Save and quit
Command-line	`:q!`	Quit without saving

nano Text Editor

nano is a beginner-friendly, command-line text editor available on most Linux systems. It's simple, intuitive, and ideal for quick edits to configuration files, scripts, or notes—especially for users who find vi or vim too complex.

Key Features:

- **Straightforward interface**: Commands are displayed at the bottom of the screen.
- **No modes**: Unlike vi, you can start typing immediately—no need to switch between modes.
- **Keyboard shortcuts**: Uses Ctrl key combinations for saving, exiting, searching, and more.
- **Safe editing**: Prompts before overwriting or exiting unsaved changes.

Basic Usage:

- **Open a file:** `nano filename.txt`
- **Save changes:** `Ctrl + O` (then press Enter)

- **Exit editor:** `Ctrl + X`
- **Cut a line:** `Ctrl + K`
- **Paste a line:** `Ctrl + U`
- **Search text:** `Ctrl + W`
- **Go to line:** `Ctrl + _` (then enter line number)

Common Options

Option	Purpose	Example
`-w`	Disable line wrapping	`nano -w config.txt`
`-m`	Enable mouse support	`nano -m notes.txt`
`-l`	Enable line numbers	`nano -l script.sh`
`-v`	View-only mode	`nano -v readonly.txt`
`-B`	Create backup before editing	`nano -B settings.conf`

File Globbing

File globbing is a shell feature in Linux that allows users to specify patterns to match filenames and directories. Instead of typing each file name manually, globbing uses special characters—called *wildcards*—to expand patterns into matching paths.

Common Wildcards Used in Globbing:

Wildcard	Meaning	Example	Matches
`*`	Matches any number of characters	`*.txt`	All `.txt` files
`?`	Matches exactly one character	`file?.txt`	`file1.txt`, `fileA.txt`
`[]`	Matches any one character inside brackets	`file[12].txt`	`file1.txt`, `file2.txt`
`[!x]` or `[^x]`	Matches any character *except* x	`file[!3].txt`	All except `file3.txt`
`{}`	Matches comma-separated patterns	`{file1,file2}.txt`	`file1.txt`, `file2.txt`

Examples of File Globbing in Action:

- `ls *.jpg` – Lists all `.jpg` files in the current directory.
- `rm file?.txt` – Deletes files like `file1.txt`, `fileA.txt`, but not `file10.txt`.
- `cp data[1-3].csv backup/` – Copies `data1.csv`, `data2.csv`, and `data3.csv` to the backup folder.
- `mv {report,summary}.docx archive/` – Moves both `report.docx` and `summary.docx` to the archive.

Variables

What Are Shell Variables?

Shell variables are named memory locations used to store data such as strings, numbers, or command outputs. They allow scripts to be dynamic and reusable.

Types of Shell Variables:

Type	Description	Examples
User-defined variables	Created by users to store custom values	`name="Muhammed"`
Environment variables	System-wide variables used by shell and programs	`PATH, HOME, USER`
Positional parameters	Automatically set when a script is run with arguments	`$1, $2, $@`
Special variables	Provide metadata about the shell or script	`$?, $$, $#, $0`

Declaring and Using Variables:

- **Assignment**: `greeting="Hello"` (no spaces around =)
- **Access**: `echo $greeting`
- **Quotes**: Use double quotes to preserve spaces: `echo "$greeting"`
- **Command substitution**: `today=$(date)` stores command output
- **Arithmetic**: `sum=$((a + b))`
- **Exporting**: `export VAR=value` makes it available to child processes
- **Unsetting**: `unset VAR` removes the variable

Positional Parameters in Scripts:

Variable	Meaning
`$0`	Script name
`$1, $2, …`	First, second, etc. arguments
`$@`	All arguments as separate words
`$*`	All arguments as a single word
`$#`	Number of arguments
`$$`	Process ID of the script
`$?`	Exit status of last command

Environment Variables:

Environment variables are predefined and affect system behavior.

Variable	Purpose
`PATH`	Directories searched for commands
`HOME`	User's home directory
`USER`	Current username
`SHELL`	Default shell
`LANG`	Language and locale settings

- To view all: `printenv`, `env`, or `set`

- To set temporarily: `export LANG=en_US.UTF-8`

Advanced Variable Features

- **Arrays** (Bash only):
```
fruits=("apple" "banana" "cherry")
echo ${fruits[1]}  # banana
```
- **Indirect references**:
```
varname="greeting"
greeting="Hello"
echo ${!varname}  # Hello
```
- **Default values**:
```
echo ${name:-Guest}  # prints "Guest" if $name is unset
```

Arithmetics

Arithmetic in Linux shell scripting allows you to perform calculations directly in the terminal or within scripts. It supports basic operations like addition, subtraction, multiplication, division, and modulus, and is essential for loops, counters, and logic.

Arithmetic Syntax in Bash:

Syntax	Purpose	Example
`$((expression))`	Standard arithmetic expansion	`sum=$((3 + 5))`
`expr`	Legacy command for arithmetic	`expr 7 \* 2`
`let`	Assign and evaluate expressions	`let "x = 10 * 2"`
`bc`	Precision math and floating point	`echo "scale=2; 5 / 3" | bc`

Supported Operators:

Operator	Function
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division (integer only)
`%`	Modulus (remainder)
`**`	Exponentiation (Bash 4+)
`++ / --`	Increment / Decrement

Examples:

```
# Basic arithmetic
a=5
b=3
sum=$((a + b))       # 8
diff=$((a - b))      # 2
prod=$((a * b))      # 15
```

```
quot=$((a / b))      # 1
mod=$((a % b))       # 2
# Increment
((a++))              # a becomes 6
((b--))              # b becomes 2
# Using let
let "total = a + b"
# Floating point with bc
echo "scale=2; 5 / 3" | bc   # Output: 1.66
```

Arithmetic in Loops:

```
for i in {1..5}; do
  echo "Square of $i is $((i * i))"
done
```

grep Command

What Is grep?

- grep stands for **Global Regular Expression Print**.
- It searches files or input streams for lines that match a given pattern and prints those lines.
- It supports **plain text searches** and **regular expressions** for advanced pattern matching.

Basic Syntax

```
grep [options] pattern [file...]
```

- **pattern:** The text or regular expression to search for.
- **file:** One or more files to search in.
- **options:** Flags to modify behavior.

Common Options

Option	Purpose	Example
`-i`	Case-insensitive search	`grep -i "error" logfile.txt`
`-n`	Show line numbers	`grep -n "main" program.c`
`-r`	Recursive search in directories	`grep -r "TODO" ./project`
`-v`	Invert match (show non-matching lines)	`grep -v "success" logfile.txt`
`-c`	Count matching lines	`grep -c "warning" logfile.txt`
`-l`	Show only filenames with matches	`grep -l "config" *.conf`

`--color=auto`	Highlight matches	`grep --color=auto "root" /etc/passwd`

Examples

```
# Search for "python" in notes.txt
grep "python" notes.txt
# Search case-insensitive
grep -i "linux" article.txt
# Search recursively in a directory
grep -r "password" /etc/
# Count occurrences of a word
grep -c "failed" auth.log
# Show line numbers with matches
grep -n "main" script.sh
```

Redirection

Redirection in Linux allows you to control where input comes from and where output goes. Instead of displaying output on the screen or reading input from the keyboard, you can redirect it to files, devices, or other commands.

What Is Redirection?

Redirection is a shell feature that lets you change the default input/output behavior of commands. By default:

- **Standard input (stdin)** comes from the keyboard.
- **Standard output (stdout)** goes to the screen.
- **Standard error (stderr)** also goes to the screen.

Redirection lets you reroute these streams.

Types of Redirection:

Symbol	Stream	Purpose	Example
`>`	stdout	Redirect output (overwrite)	`ls > files.txt`
`>>`	stdout	Redirect output (append)	`echo "new" >> log.txt`
`<`	stdin	Redirect input from file	`sort < names.txt`
`2>`	stderr	Redirect error output	`grep "x" file.txt 2> errors.log`
`2>>`	stderr	Append error output	`command 2>> errors.log`
`&>`	stdout + stderr	Redirect both to one file	`command &> output.log`
`<<`	stdin	Here-document (multi-line input)	`cat << EOF`
`<<<`	stdin	Here-string (single-line input)	`grep "word" <<< "some text"`

Examples:

```
# Save command output to a file
ls > list.txt
# Append output to a file
echo "Log entry" >> log.txt
# Redirect input from a file
sort < names.txt
# Redirect errors to a file
grep "pattern" missingfile.txt 2> errors.txt
# Redirect both output and errors
./script.sh &> full_output.log
```

Pipes

What Are Pipes?

- A **pipe** connects two or more commands.
- The **stdout (standard output)** of the first command becomes the **stdin (standard input)** of the next command.
- This avoids temporary files and makes workflows efficient.

Syntax

```
command1 | command2 | command3
```

Examples of Pipes

- **Filter output with grep:**
  `ls -l | grep ".txt"`
  Shows only .txt files from the directory listing.
- **Count lines in a file:**
  `cat logfile.log | wc -l`
  Counts the number of lines in logfile.log.

Users

What Is a User?

- A **user** is an account created to access and operate the system.
- Each user has a **username** and a **User ID (UID)**.
- Users are associated with **groups** that define collective permissions.
- The system uses users to enforce **security, isolation, and accountability**.

Types of Users

Type	Description	Example
Root user	Superuser with full control over the system	`root`
Regular user	Standard account with limited privileges	`muhammed, student`

System user	Created by the system for services and processes	`www-data, mysql`
Guest user	Temporary account with minimal access	`guest`

User Information

Stored in `/etc/passwd` file, which contains:

- **Username**
- **UID (User ID)**
- **GID (Group ID)**
- **Home directory**
- **Default shell**

Example entry:

`muhammed:x:1001:1001:Muhammed:/home/muhammed:/bin/bash`

Common User Commands

Command	Purpose	Example
`whoami`	Show current user	`whoami`
`id`	Show UID, GID, and groups	`id muhammed`
`adduser / useradd`	Create new user	`sudo adduser student`
`passwd`	Change user password	`passwd muhammed`
`su`	Switch user	`su root`
`deluser / userdel`	Delete user	`sudo userdel student`

Passwords

What Are Passwords?

- A **password** is a secret string linked to a user account.
- It ensures that only the rightful owner can log in or perform privileged actions.
- Passwords are stored in a **hashed** form, not plain text, for security.

Where Passwords Are Stored?

- Historically: `/etc/passwd` contained user info and passwords (in plain text).
- Modern systems: `/etc/shadow` stores **hashed passwords** with restricted access.
- Hashing algorithms: SHA-512, bcrypt, or other secure methods.

Example entry:

`muhammed:$6$abc123$xyzHashedPasswordData:19000:0:99999:7:::`

Managing Passwords

Command	Purpose	Example
`passwd`	Change your password	`passwd muhammed`
`sudo passwd user`	Change another user's password (admin)	`sudo passwd student`

`chage`	Manage password expiry	`chage -l muhammed`
`passwd -l user`	Lock a user's password	`sudo passwd -l guest`
`passwd -u user`	Unlock a user's password	`sudo passwd -u guest`

Groups

What Is a Group?

- A **group** is an entity that contains one or more users.
- Each user belongs to at least one group (their **primary group**) and may belong to additional groups (**secondary groups**).
- Groups are used to control access to files, directories, and system resources.

Group Information

- Stored in `/etc/group` file.
- Example entry:
  `developers:x:1002:muhammed,ahmed,sara`
  - group_name → Name of the group
  - x → Placeholder for password (rarely used today)
  - GID → Group ID
  - user_list → Members of the group

Types of Groups

Type	Description	Example
Primary group	Assigned when a user is created; owns files created by the user	`muhammed → GID 1001`
Secondary group	Additional groups a user can join for shared permissions	`developers, admins`
System groups	Created for services or processes	`www-data, mysql`

Common Group Commands

Command	Purpose	Example
groups user	Show groups a user belongs to	`groups muhammed`
groupadd	Create a new group	`sudo groupadd developers`
groupdel	Delete a group	`sudo groupdel testers`
usermod -aG	Add user to a group	`sudo usermod -aG developers muhammed`
gpasswd -d	Remove user from group	`sudo gpasswd -d muhammed developers`

File Management

What Is File Management?

- It is the process of **creating, organizing, editing, moving, and deleting files and directories**.
- Linux treats **everything as a file**: documents, directories, devices, processes, and even hardware.
- The **file system** provides a hierarchical structure starting from the root `/`.

File System Hierarchy

- `/` → Root directory (top of the hierarchy).
- `/home` → User home directories.
- `/etc` → Configuration files.
- `/bin` → Essential binaries (commands).
- `/var` → Variable data (logs, caches).
- `/dev` → Device files.
- `/tmp` → Temporary files.

For more details → https://refspecs.linuxfoundation.org/FHS_3.0/index.html

Common File Management Commands

Command	Purpose	Example
`ls`	List files and directories	`ls -l`
`pwd`	Show current directory	`pwd`
`cd`	Change directory	`cd /home/muhammed`
`touch`	Create empty file	`touch notes.txt`
`cp`	Copy files/directories	`cp file.txt backup/`
`mv`	Move/rename files	`mv file.txt report.txt`
`rm`	Delete files	`rm old.txt`
`mkdir`	Create directory	`mkdir projects`
`rmdir`	Remove empty directory	`rmdir testdir`
`find`	Search for files	`find /home -name "*.txt"`
`stat`	Show file statistics	`stat file.txt`

File Permissions and Ownership

- Each file has:
  - **Owner** (user who created it).
  - **Group** (users with shared access).
  - **Permissions** (read `r`, write `w`, execute `x`).
    Example:
    ```
    -rwxr-xr--  1 muhammed developers  1024 Nov 28  report.sh
    ```
    - Owner (muhammed) → full rights.

- Group (developers) → read + execute.
- Others → read only.

Commands:

- `chmod` → Change permissions.
- `chown` → Change ownership.
- `chgrp` → Change group ownership.

Advanced File Management

- **Links**:
  - Hard links (`ln file1 file2`) → multiple names for the same file.
  - Symbolic (Soft) links (`ln -s file1 link1`) → shortcuts to files.
- **Archiving & Compression**:
  - `tar`, `gzip`, `bzip2`, `zip` for backups.
- **File redirection & pipelines**:
  - Redirect output (>), append (>>), or chain commands (|).

Hard Links vs Soft Links

Hard Link

- A **hard link** is another name for the same file on disk.
- It points directly to the file's **`inode`** (the data structure storing file metadata and location).
- Multiple hard links share the same `inode`, meaning they are indistinguishable from the original file.

Characteristics:

- File remains accessible as long as at least one hard link exists.
- Deleting one hard link does not delete the actual data if other links exist.
- Cannot span across different file systems or partitions.
- Cannot be created for directories (to avoid loops).

Example:

```
ln file1.txt file2.txt
```

- Now `file1.txt` and `file2.txt` are hard links pointing to the same `inode`.

Soft Link (Symbolic Link)

- A **soft link** (or symlink) is a shortcut or reference to another file.
- It stores the **path** to the target file, not the inode.
- Works like a pointer: if the target file is deleted, the symlink becomes broken.

Characteristics:

- Can span across different file systems.
- Can link to directories as well as files.
- If the original file is deleted, the symlink points to nothing (broken link).

Example:

```
ln -s /home/muhammed/file1.txt shortcut.txt
```

- Here `shortcut.txt` is a soft link pointing to `/home/muhammed/file1.txt`.

Comparison Table

Feature	Hard Link	Soft Link
Points to	Inode (file data)	File path
File system	Same filesystem only	Can cross filesystems
Directories	Not allowed	Allowed
If original deleted	Data still accessible	Link breaks
Command	`ln file1 file2`	`ln -s file1 linkname`

Hard links are duplicate names for the same file data, while soft links are pointers (shortcuts) to files or directories.

xargs Command

What Is xargs?

- Some commands (like `rm`, `echo`, `mkdir`) expect arguments, not piped input.
- `xargs` bridges this gap by **reading input from stdin** (standard input) and passing it as arguments to another command.
- It's often used with commands like `find`, `grep`, or `cat` to process lists of files or strings.

Basic Syntax

```
xargs [options] [command]
```

Common Options

Option	Purpose	Example
`-0`	Handle input separated by null characters (safe for filenames with spaces)	`find . -print0 | xargs -0 rm`
`-n`	Use a fixed number of arguments per command	`echo "a b c d" | xargs -n 2`
`-I {}`	Replace {} with input items	`echo file.txt | xargs -I {} cp {} /backup/`
`-L`	Read a fixed number of lines per command	`cat list.txt | xargs -L 1 echo`
`-p`	Prompt before executing each command	`echo *.txt | xargs -p rm`

Examples

1. **Delete files found by `find`**
   ```
   find . -name "*.tmp" | xargs rm
   ```
   - Deletes all `.tmp` files.
2. **Copy files to a directory**
   ```
   echo "file1.txt file2.txt" | xargs cp -t /backup/
   ```
3. **Count lines in multiple files**
   ```
   ls *.log | xargs wc -l
   ```
4. **Safe deletion with spaces in filenames**
   ```
   find . -name "*.bak" -print0 | xargs -0 rm
   ```

Archiving

What Is Archiving?

- Archiving is the process of **bundling files together** into one archive file.
- Commonly used for **backups, distribution, and organization**.
- Archives preserve file structure, permissions, and metadata.

Archiving vs Compression

- **Archiving**: Groups files into one container (e.g., `.tar`).
- **Compression**: Reduces file size using algorithms (e.g., `.gz`, `.zip`).
- Often combined: `.tar.gz` or `.tar.bz2` → archive + compression.

Common Archiving Tools in Linux

Tool	Purpose	Example
`tar`	Create/extract archives	`tar -cvf backup.tar /home/muhammed`
`gzip`	Compress files	`gzip report.txt → report.txt.gz`
`bzip2`	Better compression than gzip	`bzip2 data.csv`
`zip`	Archive + compress (cross-platform)	`zip archive.zip file1 file2`
`unzip`	Extract zip archives	`unzip archive.zip`

Examples

```
# Create an archive
tar -cvf project.tar project/
# Extract an archive
tar -xvf project.tar
# Create compressed archive
tar -czvf project.tar.gz project/
# Extract compressed archive
tar -xzvf project.tar.gz
```

Ownership

What Is Ownership?

- Each file/directory has two levels of ownership:
  - User (Owner) → The person who created the file.
  - Group → A collection of users who share access rights.
- Ownership works hand-in-hand with permissions (r, w, x) to enforce security and access control.

Example

From `ls -l` output:

```
-rwxr-xr--  1 muhammed developers  1024 Nov 28  report.sh
```

- Owner: `muhammed` → has `rwx` (read, write, execute).
- Group: `developers` → has `r-x` (read, execute).
- Others: everyone else → has `r--` (read only).

Commands for Managing Ownership

Command	Purpose	Example
`chown`	Change file owner	`sudo chown ahmed file.txt`
`chgrp`	Change group ownership	`sudo chgrp developers file.txt`
`ls -l`	View ownership and permissions	`ls -l file.txt`

Permissions

What Are Permissions?

Every file and directory has three types of permissions:

- Read (r) → View file contents / list directory contents.
- Write (w) → Modify file contents / add or remove files in a directory.
- Execute (x) → Run a file as a program / enter a directory.

Permissions apply to three categories of users:

- Owner (user) → The person who created/owns the file.
- Group → Users who belong to the file's group.
- Others → Everyone else on the system.

Example

From ls -l output:

```
-rwxr-xr--
```

- `-` → Regular file.
- `rwx` → Owner can read, write, execute.
- `r-x` → Group can read, execute.

- `r--` → Others can only read.

Changing Permissions

- Symbolic method (using letters):
```
chmod u+x file.sh      # Add execute for owner
chmod g-w file.txt     # Remove write for group
chmod o+r file.txt     # Add read for others
```
- Numeric method (using octal values):
  `r = 4, w = 2, x = 1`
  Example:
```
chmod 755 script.sh
```
  - Owner: `rwx` (7), Group: `r-x` (5), Others: `r-x` (5).

Process

What Is a Process?

- A process is a program in execution.
- It includes the program code, current activity, allocated resources (CPU, memory), and metadata.
- Each process is identified by a unique Process ID (PID).

Types of Processes

Type	Description	Example
Foreground process	Runs interactively, tied to the terminal	Running `vim file.txt`
Background process	Runs independently, not tied to terminal	`./script.sh &`
Daemon process	Background service, starts at boot	`sshd, cron`
Zombie process	Finished execution but still has an entry in process table	Defunct processes
Orphan process	Parent terminated, but child still runs	Child adopted by `init/systemd`

Process Attributes

- PID → Unique identifier.
- PPID → Parent process ID.
- UID → User ID of the owner.
- State → Running, sleeping, stopped, zombie.
- Priority/Nice value → Determines scheduling preference.

Common Process Commands

Command	Purpose	Example
`ps`	Show running processes	`ps aux`
`top`	Real-time process monitoring	`top`
`htop`	Interactive process viewer	`htop`
`kill`	Terminate a process by PID	`kill 1234`
`jobs`	List background jobs	`jobs`
`fg`	Bring background job to foreground	`fg %1`
`bg`	Resume job in background	`bg %1`

Services

What Is a Service?

- A service is a program that runs in the background, not tied to a user's terminal.
- It provides system functionality (like SSH for remote login, Apache for web hosting, or Cron for scheduling tasks).
- Services are managed by the `init` system (`SysVinit`, Upstart, or modern `systemd`).

Examples of Services

- sshd → Secure Shell service for remote login.
- httpd / apache2 → Web server service.
- cron → Task scheduler.
- mysql → Database service.
- cups → Printing service.

Managing Services (systemd)

Most modern Linux distributions use **`systemd`** to manage services.

Command	Purpose	Example
`systemctl start service`	Start a service	`sudo systemctl start ssh`
`systemctl stop service`	Stop a service	`sudo systemctl stop apache2`
`systemctl restart service`	Restart a service	`sudo systemctl restart mysql`
`systemctl status service`	Check service status	`systemctl status cron`
`systemctl enable service`	Enable service at boot	`sudo systemctl enable ssh`

`systemctl disable service`	Disable service at boot	`sudo systemctl disable apache2`

Service vs Process

- **Process** → Any running program (foreground or background).
- **Service** → A special type of background process that provides ongoing functionality.

Software Package Manager

What Is a Package?

- A **package** is a compressed file containing software binaries, configuration files, and metadata.
- Metadata includes version, dependencies, and installation instructions.
- Example: `vim-8.2.3456.rpm` or `firefox_120.0.deb`.

What Is a Package Manager?

A package manager is a system tool that:

- Downloads software from repositories (official or third-party).
- Installs it with correct dependencies.
- Updates software when new versions are released.
- Removes software cleanly without leaving broken files.

Common Linux Package Managers

Distribution	Package Manager	Example Command
Debian/Ubuntu	apt, dpkg	`sudo apt install nginx`
Fedora/Red Hat	dnf, yum, rpm	`sudo dnf install httpd`
Arch Linux	pacman	`sudo pacman -S firefox`
openSUSE	zypper	`sudo zypper install git`
Alpine Linux	apk	`sudo apk add curl`

Universal package managers

Tools designed to work across all distributions, providing a consistent way to install, update, and run software regardless of whether the system uses .deb, .rpm, or another native format.

Main Universal Package Managers

Tool	How It Works	Strengths	Weaknesses
Snap	Packages software with dependencies in a sandbox. Managed by snapd.	Works on most distros, auto-updates, good for desktop apps.	Larger package size, slower startup, centralized store (Canonical).

Flatpak	Uses runtimes + sandboxing for apps. Managed by flatpak.	Strong sandboxing, decentralized repos (Flathub), good for GUI apps.	Not ideal for CLI/system tools, can be heavy.
AppImage	Portable executable file containing everything needed.	No installation required, just download and run.	No auto-updates, less integration with system.
UPT	A wrapper tool that unifies commands across native managers.	One command interface for any OS, adapts to local package manager.	Still experimental, relies on underlying package managers.

SSH

What SSH Does?

- **Remote login** → Lets you securely connect to another computer/server.
- **Command execution** → Run commands on a remote machine as if you were sitting at its terminal.
- **File transfer** → Securely copy files using `scp` or `sftp`.
- **Tunneling/forwarding** → Encrypt traffic for other applications (e.g., forwarding web traffic through SSH).

How SSH Works?

- Uses **TCP port 22** by default.
- Requires an **SSH client** (on your local machine) and an **SSH server** (on the remote machine).
- Authentication can be done via:
  - **Password login**
  - **SSH keys** (public/private key pair, more secure).

Basic Syntax

```
ssh [username]@[hostname or IP]
```

Example

```
ssh muhammed@192.168.1.10
```

- This connects to the server at `192.168.1.10` as user `muhammed`.

Common SSH Commands

Command	Purpose	Example
`ssh user@host`	Connect to remote server	`ssh root@server.com`

`scp file user@host:/path`	Copy file securely	`scp report.txt muhammed@192.168.1.10:/home/muhammed/`
`sftp user@host`	Secure FTP session	`sftp muhammed@192.168.1.10`
`ssh-keygen`	Generate SSH key pair	`ssh-keygen -t rsa -b 4096`
`ssh-copy-id user@host`	Copy public key to server	`ssh-copy-id muhammed@192.168.1.10`

Shell & Bash Scripting in Linux

What Is a Shell?

- A shell is a command-line interface (CLI) that allows users to interact with the operating system.
- It interprets commands typed by the user and passes them to the kernel for execution.
- Common shells:
  - Bash (Bourne Again Shell) → Default on most Linux distros.
  - Zsh, Fish, Ksh, Tcsh → Alternative shells with extra features.

What Is a Bash Script?

- A Bash script is a text file containing a series of commands written for the Bash shell.
- Instead of typing commands one by one, you can automate tasks by writing them in a script.
- Scripts usually start with a shebang (`#!`) line to specify the interpreter.

Example:

```
1 #!/bin/bash
2 echo "Hello, Muhammed!"
3 date
4 ls -l /home/muhammed
```

- `#!/bin/bash` → tells the system to use Bash to run the script.
- Commands run sequentially when the script is executed.

Why Use Bash Scripts?

- Automation → Repetitive tasks (backups, monitoring, deployments).
- Efficiency → Run multiple commands with one script.
- System administration → Manage users, processes, services.
- Customization → Create personal tools and shortcuts.

Running a Bash Script

1. Create the script file:
   `nano myscript.sh`
2. Add commands inside.
3. Make it executable:
   `chmod +x myscript.sh`
4. Run it:
   `./myscript.sh`

Recommended Playlist

https://youtu.be/gojeTqXdBH0?si=TpVsI43CWB-_goQ4

Facebook: https://www.facebook.com/M0ham6d/
LinkedIn: https://www.linkedin.com/in/muhammad-ayman-63a02524a/
GitHub: https://github.com/M0oham6d
Mail: eng.muhammed.ayman@gmail.com