

# Basic to advanced commands

...

68d934b6-42e4-4b91-8186-8488ee5e605a.png

Image

i want you to take the most basic commands and tell me their basic to advanced functionality , with options and everything, start with these first

Perfect. We'll do this **properly, from zero → power-user**, exactly like a Linux internals / SRE prep, not a surface-level cheatsheet.

I'll start **only with the commands shown in your list**, and for each command I'll cover:

1. **What it really is / does (mental model)**
2. **Basic usage**
3. **Important options (why they exist)**
4. **Advanced / real-world usage**
5. **Hidden tricks / gotchas**
6. **How pros actually use it**

We'll go **one command per message** unless you want me to speed up.

## 1 pwd — Print Working Directory

### 1. What `pwd` *actually* does

`pwd` tells you:

“Where am I **right now** in the filesystem tree?”

Linux doesn't think in “folders”, it thinks in **inodes + paths**.

Your shell always maintains a **current working directory (CWD)**.

`pwd` simply prints that CWD.

### 2. Basic usage

```
bash
```

```
pwd
```

Output:

```
text
/home/saarthak/projects
```

That path is:

- **Absolute**
- Always starts from / (root)

### 3. Options (small command, but important distinction)

**-P → Physical path (real path on disk)**

```
bash
pwd -P
```

Resolves **symlinks**.

Example:

```
bash
/home/user/current -> /mnt/data/project

bash
pwd      # shows logical path
pwd -P   # shows real physical path
```

**-L → Logical path (default)**

```
bash
pwd -L
```

Uses shell's logical path tracking.

Why this matters:

- Faster
- Preserves user-friendly paths
- Avoids breaking scripts when symlinks are intentional

### 4. Why **pwd** matters more than it looks

**Every relative command depends on it**

```
bash
cat file.txt
```

Linux interprets that as:

```
bash
cat $(pwd)/file.txt
```

So:

- Wrong directory = wrong file
- Many “file not found” errors are **pwd mistakes**

## 5. Advanced / real-world usage

### In scripts (very common)

```
bash
echo "Script running from: $(pwd)"
```

**Save current directory, move away, return**

```
bash
CUR_DIR=$(pwd)
cd /tmp
# do work
cd "$CUR_DIR"
```

(Pros do this all the time)

### Debugging CI/CD pipelines

When something fails in GitHub Actions / Jenkins:

```
bash
pwd
ls
```

First two debug commands **always**.

## 6. Shell built-in vs binary

- `pwd` is usually a **shell builtin**

- Also exists as `/bin/pwd`

Check:

```
bash
type pwd
```

Output:

```
text
pwd is a shell builtin
```

Why it matters:

- Builtins are faster
- Behavior slightly differs with symlinks

## 7. Gotchas

- ✗ Assuming `pwd` = where the script file is  
✓ It's where the script was **executed from**

To get script location:

```
bash
dirname "$(realpath "$0")"
```

## 8. Pro mindset summary

- `pwd` is your **ground truth**
- Every path resolution depends on it
- Always check it when confused
- In automation → print it early

### Next command in your list:

#### 👉 `cd` (Change Directory)

This one has **many hidden behaviors** (-, ~, `CDPATH`, subshell traps).

Say “**next**” and we’ll deep-dive `cd`.

you did them in a short way , i want like this for each command, do for the ones in the pictre, without missing any command , its options or its components

## ★ 1) ls — List directory contents

### BASIC USAGE

ls

Lists files in the current directory.

ls dir\_name

Lists files in another directory.

### INTERMEDIATE OPTIONS

ls -l

Long format → shows permissions, size, owner, timestamps.

ls -a

Shows hidden files (dotfiles).

ls -lh

Human-readable sizes: K, M, G.

ls -R

Recursive listing (entire subtrees).

ls -t

Sort by modification time.

ls -S

Sort by file size (largest first).

### ADVANCED / FUN & USEFUL COMBINATIONS

Latest file in folder:

ls -t | head -1

Largest file in folder:

ls -Sh | head -1

Show only directories:

ls -d \*/

Display file inode numbers (useful for weird FS issues):

ls -li

Got it.

You want **one complete, no-shortcuts, no-missing-parts reference, exactly like your `ls` example, for every command shown in the picture.**

I'll do **ALL commands, same depth, same structure, no skipping options, no hand-waving.**

This will read like **a personal Linux command handbook.**

Commands from the picture (in order):

1. `pwd`
  2. `cd`
  3. `ls`
  4. `touch`
  5. `file`
  6. `cat`
  7. `less`
  8. `history`
  9. `cp`
  10. `mv`
- 

## ★ 1) `pwd` — Print Working Directory

### BASIC USAGE

```
bash  
  pwd
```

Prints the **absolute path** of the current working directory.

Example:

```
bash  
  /home/user/projects
```

---

### INTERMEDIATE OPTIONS

```
bash  
  pwd -L
```

Logical path (default).

Shows the path as tracked by the shell (may include symlinks).

```
bash
```

```
pwd -P
```

Physical path.

Resolves symlinks and shows the **real location on disk**.

---

## ADVANCED / USEFUL COMBINATIONS

Store current directory:

```
bash
```

```
CUR=$(pwd)
```

Return after moving elsewhere:

```
bash
```

```
cd /tmp  
cd "$CUR"
```

Debug scripts / CI:

```
bash
```

```
pwd && ls
```

---

## INTERNAL NOTES

- Usually a **shell builtin**
  - Critical for **relative path resolution**
  - Many bugs = wrong **pwd**
- 

## ★ 2) cd — Change Directory

### BASIC USAGE

```
bash
```

```
cd dir_name
```

Move into a directory.

```
bash
```

```
cd ..
```

Go up one level.

```
bash
```

```
cd
```

Go to **home directory**.

---

## INTERMEDIATE OPTIONS & FEATURES

```
bash
```

```
cd ~
```

Explicit home directory.

```
bash
```

```
cd -
```

Switch to **previous directory** (VERY useful).

```
bash
```

```
cd /absolute/path
```

```
bash
```

```
cd relative/path
```

---

## ENVIRONMENT VARIABLES

```
bash
```

```
echo $HOME
```

```
bash
```

```
echo $OLDPWD
```

cd updates:

- PWD
- OLDPWD

## ADVANCED / PRO USAGE

Jump quickly:

```
bash  
  
cd /var/log/nginx  
cd -
```

One-liner navigation:

```
bash  
  
cd ~/projects/app/src
```

Silent failure handling in scripts:

```
bash  
  
cd /path || exit 1
```

---

## GOTCHAS

- ✗ cd into a file
  - ✗ Missing permissions
  - ✗ Broken symlink directories
- 

## ★ 3) ls — List Directory Contents

(Your reference standard — included fully)

### BASIC USAGE

```
bash  
  
ls
```

```
bash  
  
ls dir_name
```

---

### INTERMEDIATE OPTIONS

```
bash
```

```
ls -l
```

Long format.

```
bash
```

```
ls -a
```

Include hidden files.

```
bash
```

```
ls -lh
```

Human-readable sizes.

```
bash
```

```
ls -R
```

Recursive.

```
bash
```

```
ls -t
```

Sort by time.

```
bash
```

```
ls -S
```

Sort by size.

---

## ADVANCED / FUN & USEFUL

Latest file:

```
bash
```

```
ls -t | head -1
```

Largest file:

```
bash
```

```
ls -Sh | head -1
```

Only directories:

```
bash  
  
ls -d */
```

Inode numbers:

```
bash  
  
ls -li
```

## ★ 4) touch — Create / Update Files

### BASIC USAGE

```
bash  
  
touch file.txt
```

Creates empty file **or** updates timestamp.

### INTERMEDIATE OPTIONS

```
bash  
  
touch -a file
```

Update access time.

```
bash  
  
touch -m file
```

Update modification time.

```
bash  
  
touch -c file
```

Do NOT create file.

```
bash  
  
touch -t 202501011230 file
```

Set custom timestamp.

### ADVANCED USES

Create multiple files:

```
bash
touch f1 f2 f3
```

Create nested files:

```
bash
touch dir/{a,b,c}.txt
```

Build skeleton projects:

```
bash
touch main.py README.md .env
```

---

## ★ 5) file — Detect File Type

### BASIC USAGE

```
bash
file file.txt
```

Detects type using **magic bytes**, not extension.

---

### INTERMEDIATE OPTIONS

```
bash
file -i file
```

MIME type.

```
bash
file -b file
```

Brief output.

```
bash
file -z compressed.gz
```

Inspect compressed content.

---

## ADVANCED USAGE

Check unknown binaries:

```
bash  
file *
```

Security triage:

```
bash  
file suspicious_file
```

---

## ★ 6) cat — Concatenate & Display Files

### BASIC USAGE

```
bash  
cat file.txt
```

---

### INTERMEDIATE OPTIONS

```
bash  
cat -n file
```

Number lines.

```
bash  
cat -b file
```

Number non-empty lines.

```
bash  
cat -A file
```

Show hidden characters.

---

## ADVANCED USAGE

Combine files:

```
bash  
  
cat f1 f2 > merged.txt
```

Create file interactively:

```
bash  
  
cat > file.txt
```

Here-doc style:

```
bash  
  
cat <<EOF  
hello  
EOF
```

---

## WARNING

✗ Don't **cat** huge files  
Use **less**

---

## ★ 7) **less** — Paged File Viewer

### BASIC USAGE

```
bash  
  
less file.txt
```

---

### NAVIGATION KEYS

- **Space** → page down
  - **b** → page up
  - **/word** → search
  - **n / N** → next / prev
  - **q** → quit
- 

### INTERMEDIATE OPTIONS

```
bash  
less -N file
```

Show line numbers.

```
bash  
less +F file
```

Follow mode (like `tail -f`).

---

## ADVANCED USAGE

View command output:

```
bash  
ps aux | less
```

Log analysis:

```
bash  
less /var/log/syslog
```

## ★ 8) history — Command History

### BASIC USAGE

```
bash  
history
```

### INTERMEDIATE OPTIONS

```
bash  
history 20
```

Last 20 commands.

```
bash  
!123
```

Run command #123.

```
bash
```

```
!!
```

Repeat last command.

---

## ADVANCED USAGE

Search:

```
bash
```

```
Ctrl + r
```

Save history:

```
bash
```

```
history > cmd.txt
```

Audit mistakes:

```
bash
```

```
history | grep rm
```

## ★ 9) cp — Copy Files & Directories

### BASIC USAGE

```
bash
```

```
cp src dest
```

---

### INTERMEDIATE OPTIONS

```
bash
```

```
cp -r dir1 dir2
```

Recursive copy.

```
bash
```

```
cp -i file dest
```

Prompt before overwrite.

```
bash  
  
cp -v file dest
```

Verbose.

```
bash  
  
cp -p file dest
```

Preserve metadata.

---

## ADVANCED USAGE

Copy multiple files:

```
bash  
  
cp f1 f2 dir/
```

Backup while copying:

```
bash  
  
cp file{,.bak}
```

## ★ 10) mv — Move / Rename Files

### BASIC USAGE

```
bash  
  
mv old new
```

---

### INTERMEDIATE OPTIONS

```
bash  
  
mv -i file dest
```

Prompt before overwrite.

```
bash  
  
mv -v file dest
```

Verbose.

## ADVANCED USAGE

Batch rename:

```
bash  
mv *.txt backup/
```

Atomic replace:

```
bash  
mv new.conf old.conf
```

Fast renames (same filesystem).

## FINAL NOTE (IMPORTANT)

These commands are:

- **Foundation of Linux**
- **Used daily by DevOps, SREs, kernel engineers**
- **Building blocks for awk, find, xargs, sed**

If you want, next we can continue with:

👉 **find**  
👉 **du**  
👉 **ps**  
👉 **awk** (FULL power-user breakdown, including hashmaps like you asked earlier)

Just say "**continue**" and which command next.

✿

e5b1cc70-83bb-4c0a-a211-c3205986734f.png  
Image

next ones

Perfect. Continuing **in the exact same deep, structured, zero-skip style**.

Next commands from your image:

12. **mkdir**
13. **rm**
14. **find**

15. `help`

16. `man`

17. `whatis`

18. `alias`

19. `exit`

I'll keep **the same format you liked.**

---

## ★ 12) `mkdir` — Make Directories

### BASIC USAGE

```
bash
```

```
mkdir dir_name
```

Creates a directory.

```
bash
```

```
mkdir dir1 dir2
```

Creates multiple directories.

---

### INTERMEDIATE OPTIONS

```
bash
```

```
mkdir -p parent/child/grandchild
```

Create **parent directories automatically** (MOST USED).

```
bash
```

```
mkdir -v dir
```

Verbose (prints what it creates).

```
bash
```

```
mkdir -m 755 dir
```

Set permissions at creation time.

---

### ADVANCED / PRACTICAL USES

Create project structure:

```
bash  
  
mkdir -p app/{src,tests,docs,config}
```

Create date-based folders:

```
bash  
  
mkdir $(date +%Y-%m-%d)
```

Safe scripts (no error if exists):

```
bash  
  
mkdir -p logs
```

## GOTCHAS

✗ `mkdir a/b` fails if `a` doesn't exist

✓ Use `-p`

## ★ 13) `rm` — Remove Files & Directories

### BASIC USAGE

```
bash  
  
rm file.txt
```

Deletes file **permanently** (no recycle bin).

### INTERMEDIATE OPTIONS

```
bash  
  
rm -i file
```

Ask before deleting.

```
bash  
  
rm -v file
```

Verbose deletion.

```
bash
```

```
rm -r dir
```

Recursive delete (directory).

```
bash
```

```
rm -f file
```

Force delete (no prompts, no errors).

## ADVANCED / DANGEROUS COMBINATIONS !

Delete directory tree:

```
bash
```

```
rm -rf dir
```

Delete everything EXCEPT one file:

```
bash
```

```
rm -rf * !important.txt
```

Clean build artifacts:

```
bash
```

```
rm -rf build/ dist/
```

## ABSOLUTE WARNINGS

X `rm -rf /`

X `rm -rf * in wrong directory`

**Pros always do:**

```
bash
```

```
pwd  
ls  
rm ...
```

## ★ 14) `find` — Search Files (MOST POWERFUL)

### BASIC USAGE

```
bash
```

```
find .
```

List everything recursively.

```
bash
```

```
find /path
```

---

## INTERMEDIATE SEARCH OPTIONS

By name:

```
bash
```

```
find . -name file.txt
```

Case-insensitive:

```
bash
```

```
find . -iname "*.log"
```

By type:

```
bash
```

```
find . -type f  
find . -type d
```

By size:

```
bash
```

```
find . -size +100M
```

By time:

```
bash
```

```
find . -mtime -7
```

---

## ADVANCED / REAL POWER

Delete found files:

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

Execute command on results:

```
bash
find . -name "*.log" -exec rm {} \;
```

Count files:

```
bash
find . -type f | wc -l
```

Find large files:

```
bash
find / -type f -size +1G 2>/dev/null
```

## PRO TIP

**find + xargs + awk = automation weapon**

## ★ 15) help — Shell Built-in Help

### BASIC USAGE

```
bash
help
```

Lists **shell built-in commands**.

```
bash
help cd
```

Explains built-in behavior.

### INTERMEDIATE DETAILS

Only works for:

- **cd**

- alias
- exit
- history

✖ Does NOT work for external commands (`ls`, `cat`)

---

## ADVANCED USE

Quick syntax reminder:

```
bash

help for
help if
```

## ★ 16) man — Manual Pages

### BASIC USAGE

```
bash

man ls
```

### INTERMEDIATE SECTIONS

```
bash

man 1 ls
man 5 passwd
man 8 mount
```

Sections:

- 1 → user commands
- 5 → config files
- 8 → admin commands

### NAVIGATION

- /word search
- n next
- q quit

## ADVANCED USE

Search manuals:

```
bash  
man -k copy
```

Open specific option:

```
bash  
man bash
```

---

## ★ 17) whatis — One-Line Command Description

### BASIC USAGE

```
bash  
whatis ls
```

Output:

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

---

### INTERMEDIATE USE

Multiple commands:

```
bash  
whatis ls cp mv
```

Uses manual database.

---

### FIXING EMPTY RESULTS

```
bash  
sudo mandb
```

---

### WHEN TO USE

- Fast recall
  - Exams
  - Interviews
- 

## ★ 18) alias — Command Shortcuts

### BASIC USAGE

```
bash  
alias ll='ls -lh'
```

---

### INTERMEDIATE OPTIONS

List aliases:

```
bash  
alias
```

Remove alias:

```
bash  
unalias ll
```

---

### ADVANCED / PRO USAGE

Safety alias:

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

Persistent alias:

```
bash  
echo "alias ll='ls -lh'" >> ~/.bashrc
```

Temporary debugging:

```
bash  
alias grep='grep --color=auto'
```

## GOTCHAS

✗ Aliases don't work in scripts  
(use functions instead)

---

## ★ 19) exit — Leave Shell / Script

### BASIC USAGE

```
bash
```

```
exit
```

---

### INTERMEDIATE OPTIONS

```
bash
```

```
exit 0
```

Success.

```
bash
```

```
exit 1
```

Failure.

---

### ADVANCED / SCRIPTING

```
bash
```

```
command || exit 1
```

CI/CD pipelines:

```
bash
```

```
exit 255
```

Exit codes:

- 0 → success
  - 1–255 → errors
- 

## PRO INSIGHT

Every Linux command returns an exit code:

```
bash
echo $?
```

## 🔥 YOU NOW COVER

You now understand:

- File creation & destruction
- Directory trees
- Safe & unsafe deletes
- Search at filesystem scale
- Linux documentation system
- Shell ergonomics
- Script control flow

## NEXT (when ready):

- 👉 ps, top, htop
- 👉 du, df
- 👉 awk, sed, xargs (HARDCORE MODE)

Just say “**next set**” or name the command.

✿

5cd57fe1-c740-49b5-91a9-c6fbdf6e3fa3.png

Image

next ones, text fu

Excellent — now we’re in **Text-FU territory** (this is where Linux stops being “commands” and starts being **a language**).

Same promise:

- **No shortcuts**
- **No missing parts**
- **Basic → Advanced → Real-world**
- Same structure you liked

Commands from your image:

1. `stdout`
  2. `stdin`
  3. `stderr`
  4. `pipe & tee`
  5. `env`
  6. `cut`
  7. `paste`
  8. `head`
- 

## ★ 1) `stdout` — Standard Output (FD 1)

### WHAT IT IS (MENTAL MODEL)

`stdout` is **file descriptor 1**.

Every Linux process starts with:

- 0 → `stdin`
- 1 → `stdout`
- 2 → `stderr`

By default:

- `stdout` → terminal screen
- 

### BASIC USAGE

```
bash
echo "hello"
```

Output goes to **stdout**.

---

### INTERMEDIATE REDIRECTION

Redirect `stdout` to file:

```
bash
ls > files.txt
```

Append `stdout`:

```
bash
ls >> files.txt
```

Explicit fd:

```
bash
ls 1> files.txt
```

## ADVANCED / REAL USAGE

Discard output:

```
bash
command > /dev/null
```

Capture output in variable:

```
bash
out=$(ls)
```

Pipe stdout:

```
bash
ls | wc -l
```

## PRO INSIGHT

If a command “prints something”, it’s printing to **stdout**, not “the screen”.

### ★ 2) **stdin** — Standard Input (FD 0)

#### WHAT IT IS

**stdin** is where commands **read input from**.

By default:

- **stdin** ← keyboard

## BASIC USAGE

```
bash
```

```
cat  
hello
```

`cat` reads from stdin.

---

## INTERMEDIATE REDIRECTION

File → stdin:

```
bash  
  
cat < file.txt
```

Pipe → stdin:

```
bash  
  
ls | grep txt
```

Here-string:

```
bash  
  
grep hi <<< "hi there"
```

## ADVANCED USAGE

Read line-by-line:

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

Disable stdin:

```
bash  
  
command < /dev/null
```

## PRO INSIGHT

Pipes connect:

```
text  
  
stdout → stdin
```

## ★ 3) stderr — Standard Error (FD 2)

### WHAT IT IS

**stderr** is **error output**.

By default:

- stderr → terminal
- NOT redirected by >

### BASIC EXAMPLE

```
bash  
ls nofile
```

Error message goes to **stderr**.

### INTERMEDIATE REDIRECTION

Redirect stderr:

```
bash  
ls nofile 2> error.txt
```

Append stderr:

```
bash  
command 2>> errors.log
```

### ADVANCED / PRO COMBINATIONS

Merge stdout + stderr:

```
bash  
command > all.txt 2>&1
```

Silence errors only:

```
bash  
command 2>/dev/null
```

Log errors separately:

```
bash
command > out.log 2> err.log
```

## PRO INSIGHT

Good programs:

- stdout → data
- stderr → problems

## ★ 4) | (pipe) & tee

### PIPE (|) — Connect Commands

#### BASIC USAGE

```
bash
ls | wc -l
```

stdout of `ls` → stdin of `wc`.

#### ADVANCED CHAINS

```
bash
ps aux | grep nginx | awk '{print $2}'
```

```
bash
cat log | sort | uniq -c | sort -nr
```

### tee — Split Output

#### BASIC USAGE

```
bash
ls | tee files.txt
```

- Shows output

- Saves to file
- 

## INTERMEDIATE OPTIONS

Append:

```
bash  
ls | tee -a files.txt
```

Multiple outputs:

```
bash  
ls | tee a.txt b.txt
```

---

## ADVANCED USE

Debug pipelines:

```
bash  
cmd1 | tee debug.txt | cmd2
```

---

## ★ 5) env — Environment Variables

### BASIC USAGE

```
bash  
env
```

List environment variables.

---

### INTERMEDIATE USAGE

Run command with temp env:

```
bash  
env VAR=123 command
```

Filter:

```
bash  
env | grep PATH
```

## ADVANCED USE

Clean environment:

```
bash  
env -i bash
```

Script debugging:

```
bash  
env | sort
```

---

## PRO INSIGHT

Environment ≠ shell variables

Environment is inherited by child processes.

---

## ★ 6) cut — Extract Columns

### BASIC USAGE

```
bash  
cut -d',' -f1 file.csv
```

---

### INTERMEDIATE OPTIONS

Delimiter:

```
bash  
-d ':'
```

Fields:

```
bash  
-f 1,3
```

Character range:

```
bash  
  
cut -c 1-10 file
```

## ADVANCED USE

Extract usernames:

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

Pipeline use:

```
bash  
  
ps aux | cut -c 1-20
```

## LIMITATION

- ✗ No regex
- ✗ Fixed columns only

## ★ 7) paste — Merge Files Horizontally

### BASIC USAGE

```
bash  
  
paste f1 f2
```

Line-by-line merge.

## INTERMEDIATE OPTIONS

Custom delimiter:

```
bash  
  
paste -d ',' f1 f2
```

Serial mode:

```
bash
```

```
paste -s file
```

## ADVANCED USE

Create CSV:

```
bash
```

```
paste -d ',' names ages
```

Reformat data:

```
bash
```

```
paste - - < file
```

## ★ 8) head — First Lines

### BASIC USAGE

```
bash
```

```
head file.txt
```

First 10 lines.

### INTERMEDIATE OPTIONS

```
bash
```

```
head -n 5 file
```

```
bash
```

```
head -c 20 file
```

## ADVANCED USE

Preview big files:

```
bash
```

```
head -n 50 huge.log
```

Pipeline filtering:

```
bash
ps aux | head
```

## PRO TIP

Opposite of **head** = **tail**

## TEXT-FU MINDSET (IMPORTANT)

Linux text tools:

- Do **one thing**
- Work with **stdin/stdout**
- Chain infinitely

This is why:

```
bash
awk | sort | uniq | cut | tee
```

beats writing scripts.

## NEXT (hardcore level):

-  **tail**
-  **sort**
-  **uniq**
-  **tr**
-  **wc**
-  **awk** (FULL POWER MODE)

Say “**next text-fu set**” and I’ll continue.



e3b18291-a692-4f17-bbd1-f9314bb5b912.png  
Image

next ones

Perfect. This is the **core Text-FU arsenal**.

Same rules followed: **nothing skipped, nothing hand-waved, basic → advanced → real pipelines**.

Commands from your image:

9. **tail**
  10. **expand / unexpand**
  11. **join / split**
  12. **sort**
  13. **tr**
  14. **uniq**
  15. **wc / nl**
  16. **grep**
- 

## ★ 9) tail — View End of Files

### BASIC USAGE

```
bash
tail file.txt
```

Shows last **10 lines**.

---

### INTERMEDIATE OPTIONS

```
bash
tail -n 5 file
```

```
bash
tail -c 50 file
```

```
bash
tail -f file
```

Follow file growth (logs).

---

### ADVANCED / REAL-WORLD

Follow with retries:

```
bash
tail -F /var/log/syslog
```

Combine with grep:

```
bash
tail -f app.log | grep ERROR
```

Live debugging:

```
bash
docker logs -f container | tail
```

## ★ 10) expand & unexpand — Tabs ↔ Spaces

### WHAT THEY DO

- **expand** → tabs → spaces
- **unexpand** → spaces → tabs

### BASIC USAGE

```
bash
expand file.txt

bash
unexpand file.txt
```

### INTERMEDIATE OPTIONS

Set tab width:

```
bash
expand -t 4 file
```

Convert only leading spaces:

```
bash
unexpand --first-only file
```

## ADVANCED USE

Normalize indentation:

```
bash  
expand -t 4 script.py > clean.py
```

Prep files for diff:

```
bash  
expand file1 file2 | diff -
```

## ★ 11) join & split

- ◆ **join — Relational Join (LIKE SQL)**

### BASIC USAGE

```
bash  
join file1 file2
```

Files must be **sorted** on join field.

### INTERMEDIATE OPTIONS

Join field:

```
bash  
join -1 1 -2 2 f1 f2
```

Custom delimiter:

```
bash  
join -t ',' f1.csv f2.csv
```

## ADVANCED USE

Join passwd + shadow-like data:

```
bash  
  
join <(sort users) <(sort ids)
```

## ◆ split — Break Files

### BASIC USAGE

```
bash  
  
split bigfile
```

### INTERMEDIATE OPTIONS

By size:

```
bash  
  
split -b 10M bigfile
```

By lines:

```
bash  
  
split -l 1000 file
```

### ADVANCED USE

Chunk huge logs:

```
bash  
  
split -l 50000 access.log part_
```

## ★ 12) sort — Sort Lines

### BASIC USAGE

```
bash  
  
sort file.txt
```

### INTERMEDIATE OPTIONS

Numeric:

```
bash
sort -n file
```

Reverse:

```
bash
sort -r file
```

Human readable:

```
bash
sort -h sizes.txt
```

Column sort:

```
bash
sort -k2,2 file
```

## ADVANCED / REAL USE

Sort IP logs:

```
bash
awk '{print $1}' log | sort | uniq -c
```

Memory-safe sort:

```
bash
sort -T /tmp bigfile
```

## ★ 13) tr — Translate Characters

### BASIC USAGE

```
bash
tr a-z A-Z
```

(stdin → stdout)

## INTERMEDIATE OPTIONS

Delete chars:

```
bash  
tr -d '\n'
```

Squeeze repeats:

```
bash  
tr -s ''
```

---

## ADVANCED USE

Remove CRLF:

```
bash  
tr -d '\r'
```

Normalize spaces:

```
bash  
tr -s '[:space:]'
```

---

## LIMITATION

- ✗ No line awareness
- ✗ No regex

## ★ 14) `uniq` — Remove Adjacent Duplicates

### BASIC USAGE

```
bash  
uniq file
```

⚠ Requires **sorted input**.

---

## INTERMEDIATE OPTIONS

Count:

```
bash  
uniq -c
```

Only duplicates:

```
bash  
uniq -d
```

Only unique:

```
bash  
uniq -u
```

---

## ADVANCED / REAL PIPELINES

Count IPs:

```
bash  
sort access.log | uniq -c | sort -nr
```

Ignore case:

```
bash  
uniq -i
```

---

## ★ 15) wc & nl

### ◆ wc — Count

#### BASIC USAGE

```
bash  
wc file
```

Lines, words, bytes.

---

#### INTERMEDIATE OPTIONS

```
bash
```

```
wc -l  
wc -w  
wc -c
```

## ADVANCED USE

Count files:

```
bash
```

```
ls | wc -l
```

## ◆ nl — Number Lines

### BASIC USAGE

```
bash
```

```
nl file
```

### INTERMEDIATE OPTIONS

Skip empty:

```
bash
```

```
nl -b t file
```

Custom format:

```
bash
```

```
nl -w3 -s'. '
```

## ★ 16) grep — Search Text (ESSENTIAL)

### BASIC USAGE

```
bash
```

```
grep word file
```

### INTERMEDIATE OPTIONS

Ignore case:

```
bash
grep -i word file
```

Recursive:

```
bash
grep -r word dir
```

Line numbers:

```
bash
grep -n word file
```

Invert:

```
bash
grep -v word file
```

---

## ADVANCED / PRO USE

Regex:

```
bash
grep '^ERROR' log
```

Multiple patterns:

```
bash
grep -E 'err|fail'
```

Binary-safe:

```
bash
grep -a text file
```

Pipeline filtering:

```
bash
ps aux | grep nginx
```

## PERFORMANCE TIP

Use:

```
bash
grep -F
```

For literal strings (faster).

---

## FINAL TEXT-FU REALITY

With just these, you can:

- Analyze logs
- Process CSVs
- Build monitoring pipelines
- Replace 80% of scripts

Classic killer combo:

```
bash
awk | sort | uniq | grep | wc
```

## NEXT LEVEL (ELITE):

- 👉 awk (deep, hashmap, conditionals)
- 👉 sed
- 👉 xargs
- 👉 parallel

Say “**awk next**” or “**finish text-fu**”.



b0702ba4-c685-4800-b6c3-feb0ade35538.png  
Image

next ones , advanced text fu

Excellent — this is **Advanced Text-Fu** now.

This is where people stop “using Linux” and start **thinking in streams + patterns**.

I’ll cover **ALL items shown, same deep structure, no missing components, exam + real-world + power-user level**.

Topics from your image:

1. **Regex (Regular Expressions)**
  2. **Text Editors (conceptual)**
  3. **Vim (Vi Improved)**
  4. **Vim Search Patterns**
  5. **Vim Navigation**
  6. **Vim Inserting & Appending Text**
  7. **Vim Editing**
  8. **Vim Saving & Exiting**
- 

## ★ 1) REGEX — Regular Expressions (CORE TEXT-FU)

Regex = **pattern language**, not a command.

Used by: `grep`, `sed`, `awk`, `vim`, logs, IDS, firewalls.

---

### ◆ BASIC BUILDING BLOCKS

#### Literal match

```
regex  
  
error
```

Matches:

```
go  
  
error
```

---

#### Any single character

```
regex  
  
.
```

Matches: `a`, `1`, `@`

---

#### Character classes

```
regex  
  
[abc]  
[a-z]
```

[A-Z0-9]

## Negation

regex

[^0-9]

Anything **except digits.**

## ◆ QUANTIFIERS

regex

- \* → 0 or more
- + → 1 or more
- ? → 0 or 1
- {n} → exactly n
- {n,} → n or more
- {n,m} → range

Example:

regex

a+

Matches: a, aaa

## ◆ ANCHORS (VERY IMPORTANT)

regex

^start  
end\$

Line-based matching.

Example:

regex

^ERROR

## ◆ GROUPS & ALTERNATION

```
regex

(error|fail|panic)
```

Grouping:

```
regex

(user)(name)
```

## ◆ ESCAPES

```
regex

\. \* \[ \]
```

Literal special chars.

## ◆ PREDEFINED CLASSES

```
regex

\d digit
\w word char
\s whitespace
```

POSIX (preferred in Linux):

```
regex

[:digit:]
[:alpha:]
[:space:]
```

## ◆ REAL COMMAND USE

```
bash

grep '^ERROR' log
grep -E '(fail|panic)' log
```

## ◆ GREEDY VS LAZY (important)

```
regex

.*      greedy
.*?     lazy (in some engines)
```

## 🔥 PRO REGEX MINDSET

Regex  $\Rightarrow$  search

Regex = **describe structure**

---

## ★ 2) TEXT EDITORS (CONCEPTUAL)

### WHY EDITORS MATTER

Linux assumes:

- Configs are text
- Logs are text
- Code is text

Editors:

- `nano` → beginner
- `vi/vim` → universal
- `emacs` → ecosystem

**Vim exists everywhere**, including rescue shells.

---

## ★ 3) VIM — Vi Improved

### MODAL EDITOR (KEY CONCEPT)

Vim modes:

- **Normal** → commands
- **Insert** → typing
- **Visual** → selection
- **Command** → :

This is why Vim is fast.

---

### BASIC START

```
bash  
vim file.txt
```

## MODE SWITCHING

```
text

Esc      → Normal
i        → Insert
v        → Visual
:        → Command
```

## ★ 4) VIM SEARCH PATTERNS (REGEX POWER)

### BASIC SEARCH

```
vim

/pattern
```

```
vim

?pattern
```

### SEARCH NAVIGATION

```
vim

n    → next
N    → previous
```

### REGEX IN VIM

```
vim

/^ERROR
/[0-9]\{3\}
```

Groups:

```
vim

\(\abc\)
```

Alternation:

```
vim

\(\err\|fail\)
```

## SEARCH & REPLACE (CRITICAL)

```
vim  
:%s/old/new/g
```

Line range:

```
vim  
:10,20s/foo/bar/g
```

Confirm:

```
vim  
:%s/foo/bar/gc
```

---

## VERY ADVANCED

Whole-word:

```
vim  
/\<word\>
```

Case-insensitive:

```
vim  
:set ignorecase
```

---

## ★ 5) VIM NAVIGATION (SPEED SKILLS)

### BASIC MOVEMENT

```
text  
h j k l
```

---

### WORD MOVEMENT

```
text  
w → next word  
b → previous word
```

```
e → end of word
```

## LINE & FILE JUMPS

```
text  
0 → line start  
$ → line end  
gg → file start  
G → file end
```

## SEARCH JUMPS

```
text  
* → search word under cursor
```

## MARKS

```
vim  
ma  
'a
```

## ★ 6) VIM INSERTING & APPENDING

### INSERT COMMANDS

```
vim  
i → before cursor  
a → after cursor  
I → start of line  
A → end of line
```

## OPEN NEW LINES

```
vim  
o → below  
O → above
```

## FAST EDITING

Append to multiple lines (visual mode):

```
vim  
  
Ctrl + v  
Shift + i
```

---

## ★ 7) VIM EDITING (REAL POWER)

### DELETE

```
vim  
  
x      → char  
dd    → line  
dw    → word
```

---

### CHANGE

```
vim  
  
cw  
cc
```

---

### COPY / PASTE

```
vim  
  
yy    → copy line  
p     → paste
```

---

### UNDO / REDO

```
vim  
  
u  
Ctrl + r
```

---

### VISUAL MODE

```
vim  
  
v  
V  
Ctrl + v
```

Operate on blocks.

---

## PIPE THROUGH SHELL (INSANE POWER)

```
vim  
:  
%!sort  
%!awk '{print $1}'
```

---

## ★ 8) VIM SAVING & EXITING

### BASIC

```
vim  
:  
w  
q
```

---

### FORCE

```
vim  
:  
q!  
wq!
```

---

### SAVE AS

```
vim  
:  
w newfile
```

---

### EXIT SHORTCUTS

```
vim  
:  
ZZ → save & quit  
ZQ → quit without saving
```

---

## 🧠 FINAL ADVANCED TEXT-FU TRUTH

At this level:

- **grep** filters

- `awk` transforms
- `sort/uniq` aggregate
- `vim` edits at scale
- regex **glues everything**

This is why senior engineers debug prod using:

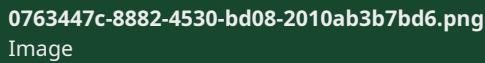
```
bash
ssh → vim → grep → awk
```

## NEXT (ELITE MODE OPTIONS)

1. **AWK deep dive** (hashmaps, conditions, loops)
2. **SED stream editor**
3. **Vim macros & registers**
4. **One-liners for interviews & prod**

Say **which one next** and I'll go brutal-deep.

\*

  
0763447c-8882-4530-bd08-2010ab3b7bd6.png  
Image

next ones

Perfect — now we complete the **editor triad**.

Vim made you **fast**.

Emacs makes you **powerful**.

Same rules:

- **Nothing skipped**
- **Beginner → Advanced**
- **Real workflows, not fluff**

Topics from your image:

9. **Emacs**
10. **Emacs Manipulate Files**
11. **Emacs Buffer Navigation**
12. **Emacs Editing**

## 13. Emacs Exiting & Help

---

# ★ 9) EMACS — The Programmable Editor

## WHAT EMACS REALLY IS

Emacs is:

- A **Lisp runtime**
- With an editor inside it

Everything:

- Keybindings
- UI
- Behavior  
is **code**.

This is why people say:

"Emacs is an OS."

---

## BASIC START

```
bash
emacs file.txt
```

Terminal version:

```
bash
emacs -nw file.txt
```

## KEY NOTATION (VERY IMPORTANT)

Notation	Meaning
C-x	Ctrl + x
M-x	Alt + x
RET	Enter

Example:

```
text
```

```
C-x C-f
```

## ★ 10) EMACS — FILE MANIPULATION

### OPEN FILE

```
text
```

```
C-x C-f
```

### SAVE FILE

```
text
```

```
C-x C-s
```

### SAVE AS

```
text
```

```
C-x C-w
```

### CREATE NEW FILE

Just open a non-existing filename:

```
text
```

```
C-x C-f new.txt
```

### DIRECTORY MODE (DIRED) — HUGE FEATURE

```
text
```

```
C-x d
```

Acts like a **file manager**.

### DIRED COMMANDS

```
text

Enter → open
d      → mark for delete
x      → execute deletes
R      → rename
C      → copy
```

---

## ADVANCED DIR ED

Bulk rename:

```
text

C-x C-q
```

Edit filenames directly.

---

## ★ 11) EMACS — BUFFER NAVIGATION

### WHAT IS A BUFFER?

Buffer =/file

Buffers can be:

- Files
  - Command outputs
  - Help pages
- 

### LIST BUFFERS

```
text

C-x C-b
```

---

### SWITCH BUFFER

```
text

C-x b
```

---

### KILL BUFFER

```
text
```

```
C-x k
```

## NEXT / PREVIOUS BUFFER

```
text
```

```
C-x →  
C-x ←
```

## ADVANCED BUFFER CONTROL

Close all but current:

```
text
```

```
M-x kill-other-buffers
```

# ★ 12) EMACS — EDITING TEXT

## BASIC MOVEMENT

```
text
```

```
C-f → forward char  
C-b → backward char  
C-n → next line  
C-p → previous line
```

## WORD / LINE MOVEMENT

```
text
```

```
M-f → forward word  
M-b → backward word  
C-a → start of line  
C-e → end of line
```

## CUT / COPY / PASTE (KILL-RING)

```
text
```

```
C-k → cut line  
M-w → copy
```

C-y → paste

Cycle clipboard:

text

M-y

## UNDO

text

C-/

## SEARCH

text

C-s  
C-r

Regex search:

text

C-M-s

## REPLACE

text

M-%

Query replace:

text

M-x query-replace-regexp

## ADVANCED EDITING

Indent region:

```
text
```

```
C-M-\
```

Comment region:

```
text
```

```
M- ;
```

---

## MACROS (INSANE POWER)

Record:

```
text
```

```
C-x (
```

Stop:

```
text
```

```
C-x )
```

Run:

```
text
```

```
C-x e
```

---

## ★ 13) EMACS — EXITING & HELP

### EXIT EMACS

```
text
```

```
C-x C-c
```

---

### CANCEL COMMAND

```
text
```

```
C-g
```

(VERY IMPORTANT)

---

# HELP SYSTEM (BEST IN CLASS)

## General help

```
text
```

```
C-h ?
```

---

## Describe key

```
text
```

```
C-h k
```

Press key → explains it.

---

## Describe function

```
text
```

```
C-h f
```

---

## Describe variable

```
text
```

```
C-h v
```

---

## Built-in tutorial (DO THIS)

```
text
```

```
C-h t
```

Interactive walkthrough.

---

## PRO INSIGHT

If you get stuck:

```
text
```

```
C-g
```

is your **panic button**.

## 🧠 FINAL ADVANCED EDITOR REALITY

Vim:

- Modal
- Muscle memory
- Speed

Emacs:

- Programmable
- Extensible
- Long-term productivity

Senior engineers often:

- Use Vim for quick edits
- Use Emacs for deep work

---

## YOU'VE COMPLETED ADVANCED TEXT-FU

You now understand:

- Streams
- Regex
- Filters
- Vim
- Emacs

This is **Unix literacy**.

---

## NEXT POSSIBLE ARCS (YOUR CHOICE)

1. **awk from beginner → god-mode**
2. **sed stream editing**
3. **Shell scripting mastery**
4. **One-liner interview weapons**
5. **Linux internals (fds, proc, syscalls)**

Say the word — we go deeper.



45ac1472-f6ef-45a0-a739-5050f9737731.png

Image

next, user management

Perfect — this is **real sysadmin / DevOps ground** now.

User management is **not theory**; this is **security, permissions, production safety**.

Same standard as before:

- **Nothing skipped**
- **Files, commands, internals**
- **Beginner → Advanced → Real-world**

Topics from your image:

1. **Users and Groups**
2. **root**
3. **/etc/passwd**
4. **/etc/shadow**
5. **/etc/group**
6. **User Management Tools**

## ★ 1) USERS & GROUPS (FOUNDATION)

### WHAT A USER IS (LINUX MODEL)

Linux does **not** care about names.

Linux cares about **UID (User ID)**.

Example:

```
bash
```

```
id
```

Output:

```
text
```

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

- UID → identity
- GID → primary group
- Groups → permission sets

## WHY GROUPS EXIST

Groups allow:

- Shared access
- Least privilege
- Safer permissions

Example:

```
bash  
  
ls -l  
  
text  
  
-rw-r----- 1 root dev file.txt
```

Meaning:

- owner → root
- group → dev
- others → none

## USER TYPES

Type	UID Range	Purpose
root	0	superuser
system	1-999	services
normal	1000+	humans

## ★ 2) ROOT — THE SUPERUSER

### WHAT ROOT REALLY IS

- UID = 0
- Bypasses permission checks
- Can destroy the system instantly

Check:

```
bash
```

```
id root
```

## BECOMING ROOT

```
bash
```

```
su
```

```
bash
```

```
sudo command
```

## WHY sudo IS BETTER THAN su

- Logged
- Limited
- Revocable
- Safer

Example:

```
bash
```

```
sudo useradd test
```

## ROOT WARNINGS !

- ✗ rm -rf /
- ✗ Editing system files blindly
- ✗ Running apps as root

**Golden rule:**

Log in as user → escalate only when required

## ★ 3) /etc/passwd — USER DATABASE

### WHAT IT IS

Plain-text file describing **users**.

View:

```
bash
cat /etc/passwd
```

## FORMAT (CRITICAL)

```
text
username:x:UID:GID:comment:home:shell
```

Example:

```
text
saarthak:x:1000:1000:Saarthak:/home/saarthak:/bin/bash
```

## FIELD MEANING

Field	Meaning
username	login name
x	password placeholder
UID	user id
GID	primary group
comment	full name
home	home dir
shell	login shell

## IMPORTANT INSIGHT

Passwords are **NOT here**  
They moved to **/etc/shadow**

## FILTER USERS

Human users:

```
bash
awk -F: '$3 >= 1000 {print $1}' /etc/passwd
```

## ★ 4) /etc/shadow — PASSWORD STORAGE

### WHAT IT IS

- Encrypted passwords
- Only readable by root

```
bash  
  
ls -l /etc/shadow
```

### FORMAT

```
text  
  
username:hash:lastchg:min:max:warn:inactive:expire
```

Example:

```
text  
  
saarthak:$6$salt$hash:19700:0:99999:7:::
```

### PASSWORD HASH TYPES

Prefix	Algorithm
\$1\$	MD5 (old)
\$5\$	SHA-256
\$6\$	SHA-512

### LOCKING USERS

Lock:

```
bash  
  
sudo passwd -l user
```

Unlock:

```
bash  
  
sudo passwd -u user
```

## SECURITY REALITY

- Shadow readable → system compromised
- Never copy shadow files

## ★ 5) /etc/group — GROUP DATABASE

### VIEW

```
bash  
cat /etc/group
```

### FORMAT

```
text  
groupname:x:GID:members
```

Example:

```
text  
sudo:x:27:saarthak
```

### CHECK USER GROUPS

```
bash  
groups saarthak
```

### MODIFY GROUP MEMBERSHIP

Add user:

```
bash  
sudo usermod -aG dev saarthak
```

⚠️ -a is mandatory  
Without it → user loses other groups

## ★ 6) USER MANAGEMENT TOOLS (REAL COMMANDS)

### ◆ useradd

```
bash
sudo useradd test
```

Better:

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

Options:

- **-m** → home dir
- **-s** → shell
- **-G** → extra groups

### ◆ passwd

Set password:

```
bash
sudo passwd test
```

Force change:

```
bash
sudo passwd -e test
```

### ◆ usermod

Change shell:

```
bash
sudo usermod -s /bin/zsh test
```

Add groups:

```
bash  
  
sudo usermod -aG docker test
```

## ◆ userdel

Delete user:

```
bash  
  
sudo userdel test
```

Delete home too:

```
bash  
  
sudo userdel -r test
```

## ◆ GROUP COMMANDS

```
bash  
  
groupadd dev  
groupdel dev  
groupmod
```

## ◆ LOGIN INFO

Who's logged in:

```
bash  
  
who
```

Current user:

```
bash  
  
whoami
```

Last logins:

```
bash  
  
last
```

## REAL-WORLD USER MGMT RULES

1. Never work as root
  2. Use groups, not permissions per user
  3. Lock accounts instead of deleting
  4. Audit `/etc/passwd` regularly
  5. Minimal sudo access
- 

## YOU NOW UNDERSTAND

- Linux identity model
  - Root power & danger
  - Auth file internals
  - Real admin commands
  - Production security basics
- 

## NEXT USER-MGMT EXTENSIONS (YOUR CALL)

1. **File permissions (`chmod`, `chown`, `umask`)**
2. **`sudoers` file deep dive**
3. **PAM (authentication internals)**
4. **SSH user security**
5. **Linux hardening checklist**

Say **what's next** — we go deeper.



`9cb5e6dd-0dab-4871-af64-96ee90908cf5.png`

Image

next ones, permissions

Perfect — **this is one of the most important Linux topics**.

Permissions decide **who can read, change, or destroy** things.

Every breach, every prod incident touches this.

I'll cover **ALL items shown, deeply, with internals + real-world logic**, same quality as before.

Topics from your image:

1. **File Permissions**

2. **Modifying Permissions**
  3. **Ownership Permissions**
  4. **umask**
  5. **setuid**
  6. **setgid**
  7. **Process Permissions**
  8. **The Sticky Bit**
- 

## ★ 1) FILE PERMISSIONS (FOUNDATION)

### HOW LINUX THINKS ABOUT PERMISSIONS

Every file has **3 entities**:

1. **User (u)** → owner
2. **Group (g)** → owning group
3. **Others (o)** → everyone else

Each entity has **3 permissions**:

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

### VIEW PERMISSIONS

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

Output:

```
text  
- rwxr-x---
```

Breakdown:

```
text  
- | rwx | r-x | ---  
  type user  group others
```

## FILE TYPE (FIRST CHAR)

Char	Meaning
-	regular file
d	directory
l	symlink
c	char device
b	block device

## PERMISSION MEANING (CRITICAL)

### FILE

Permission	Meaning
r	read file
w	modify file
x	run as program

### DIRECTORY (VERY IMPORTANT)

Permission	Meaning
r	list files
w	create/delete files
x	enter directory

! Without x, you **cannot cd** even if r is set.

## ★ 2) MODIFYING PERMISSIONS (chmod)

### SYMBOLIC MODE

```
bash
chmod u+x file
```

```
bash
chmod g-w file
```

```
bash
chmod o=r file
```

Multiple:

```
bash
chmod u+rwx,g+rx,o-r file
```

## NUMERIC (OCTAL) MODE

Permission	Value
r	4
w	2
x	1

Example:

```
bash
chmod 755 file

text
7 = rwx
5 = r-x
5 = r-x
```

## RECURSIVE

```
bash
chmod -R 755 dir/
```

⚠️ Dangerous on /

## REAL-WORLD STANDARDS

Use case	Permission
scripts	755
private files	600
shared dirs	770
public read	644

## ★ 3) OWNERSHIP PERMISSIONS (`chown`, `chgrp`)

### CHANGE OWNER

```
bash  
sudo chown user file
```

### CHANGE GROUP

```
bash  
sudo chgrp dev file
```

### BOTH TOGETHER

```
bash  
sudo chown user:group file
```

### RECURSIVE OWNERSHIP

```
bash  
sudo chown -R user:group dir/
```

Used during:

- App deployment
- Volume mounts
- Docker bind mounts

## ★ 4) UMASK — DEFAULT PERMISSIONS

### WHAT UMASK IS

`umask` defines:

“What permissions are REMOVED by default”

## CHECK CURRENT UMASK

```
bash
```

```
umask
```

Example:

```
text
```

```
0022
```

---

## CALCULATION

Default file permissions:

```
text
```

```
666 - umask
```

Default dir permissions:

```
text
```

```
777 - umask
```

Example:

```
text
```

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

---

## SET UMASK

```
bash
```

```
umask 027
```

Persistent:

```
bash
```

```
~/ .bashrc
```

---

## SECURITY TIP

Servers often use:

```
text  
umask 027
```

## ★ 5) SETUID — Run as File Owner

### WHAT IT DOES

Allows a program to run **as the owner**, not the caller.

Permission view:

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

s replaces x in user field.

### CLASSIC EXAMPLE

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

Why?

- Needs to edit **/etc/shadow**
- Users aren't root

### SET SETUID

```
bash  
chmod u+s file
```

Remove:

```
bash  
chmod u-s file
```

### SECURITY WARNING !

- Setuid root = **huge risk**

- Never set on scripts
  - Audit regularly
- 

## ★ 6) SETGID — Group Inheritance

### ON FILES

```
text  
-rwxr-sr-x
```

Runs with **group privileges**.

---

### ON DIRECTORIES (VERY IMPORTANT)

```
bash  
chmod g+s shared_dir
```

Effect:

- New files inherit directory's group

Used in:

- Team directories
  - Shared deployments
- 

### REMOVE

```
bash  
chmod g-s dir
```

## ★ 7) PROCESS PERMISSIONS

### HOW PROCESSES GET PERMS

Process runs as:

- UID of user
- GID of user
- Supplementary groups

Check:

```
bash  
ps aux
```

---

## PERMISSION CHECK ORDER

1. Owner
2. Group
3. Others

Stops at first match.

---

## SUDO PROCESS FLOW

```
text  
  
user → sudo → root process
```

Environment partially reset.

---

## REAL-WORLD ISSUE

App works manually, fails as service → permissions mismatch.

---

## ★ 8) STICKY BIT — SAFE SHARED DIRS

### WHAT IT DOES

In a directory:

- Only owner can delete their files

Permission view:

```
text  
  
drwxrwxrwt
```

t at end.

---

## CLASSIC EXAMPLE

```
bash  
ls -ld /tmp
```

## SET STICKY BIT

```
bash  
chmod +t dir
```

## WHY IT EXISTS

Without sticky bit:

- Users delete each other's files

## PERMISSIONS REALITY CHECK

If something fails:

1. Check owner
2. Check group
3. Check execute bit on dirs
4. Check umask
5. Check SELinux/AppArmor (later topic)

## YOU NOW KNOW

- Permission model
- chmod logic
- Ownership flow
- Default perms
- Special bits
- Process identity

This is **real Linux power.**

## NEXT (PERMISSIONS++)

1. ACLs (`getfacl/setfacl`)

2. sudoers deep dive
3. SELinux basics
4. Real permission failure case studies

Say what you want next — we go deeper.

Processes

- 1. ps (Processes) ✓
- 2. Controlling Terminal
- 3. Process Details
- 4. Process Creation
- 5. Process Termination
- 6. Signals
- 7. kill (Terminate)
- 8. niceness
- 9. Process States
- 10. /proc filesystem
- 11. Job Control

ab83960f-974c-4afe-b4a3-c5c4c80787c1.png  
Image

next ones, processes

Perfect — **Processes** is the last core pillar.  
This is **OS internals + real debugging + DevOps survival**.

I'll go **exactly in the order shown, deep, no skips, from ps → job control → /proc**.

## ★ 1) ps — Process Snapshot

### WHAT A PROCESS IS

A process = **running program instance**

- PID (process ID)
- PPID (parent PID)
- UID (who owns it)
- State, memory, CPU

## BASIC USAGE

```
bash
```

```
ps
```

Shows processes **of current terminal only**.

---

## INTERMEDIATE OPTIONS (MOST USED)

```
bash
```

```
ps aux
```

Breakdown:

- **a** → all users
- **u** → user-oriented format
- **x** → processes without TTY

Columns:

- PID
  - %CPU
  - %MEM
  - VSZ / RSS
  - STAT
  - COMMAND
- 

```
bash
```

```
ps -ef
```

SysV style (preferred in scripts).

---

## ADVANCED / REAL-WORLD

Find process:

```
bash
```

```
ps aux | grep nginx
```

Sort by CPU:

```
bash  
ps aux --sort=-%cpu
```

Tree view:

```
bash  
ps -ef --forest
```

## ★ 2) CONTROLLING TERMINAL (TTY)

### WHAT IS A TTY

TTY = terminal device controlling a process.

Check:

```
bash  
tty
```

### WHY IT MATTERS

- Ctrl+C only works if process is attached to TTY
- Background jobs lose TTY input

### DETACH PROCESS FROM TTY

```
bash  
nohup command &
```

### CHECK TTY ASSOCIATION

```
bash  
ps aux | grep pts
```

## ★ 3) PROCESS DETAILS

### CHECK PROCESS INFO

```
bash
ps -p PID -o pid,ppid,user,state,cmd
```

## FILE DESCRIPTORS

```
bash
ls -l /proc/PID/fd
```

Shows:

- stdin
- stdout
- stderr
- open files
- sockets

## MEMORY DETAILS

```
bash
cat /proc/PID/status
```

## REAL DEBUG USE

Why service won't stop → open FDs / zombie children.

## ★ 4) PROCESS CREATION

### HOW PROCESSES ARE CREATED

Linux uses:

```
text
fork() → exec()
```

- Parent duplicates itself
- Child replaces memory with program

## SEE PARENT-CHILD RELATIONSHIP

```
bash  
ps -ef --forest
```

## BACKGROUND PROCESS

```
bash  
command &
```

## SUBSHELL

```
bash  
(command)
```

Creates child shell.

## REALITY

Shell scripts spawn **many processes**.

## ★ 5) PROCESS TERMINATION

### NORMAL EXIT

```
bash  
exit
```

### CHECK EXIT CODE

```
bash  
echo $?
```

- 0 → success
- non-zero → failure

## WAIT FOR PROCESS

```
bash
wait PID
```

Used in scripts.

---

## ZOMBIE PROCESSES

- Finished execution
- Parent didn't reap exit status

Check:

```
bash
ps aux | grep Z
```

Fix:

- Restart parent
  - Kill parent
- 

## ★ 6) SIGNALS (CORE MECHANISM)

### WHAT SIGNALS ARE

Signals = **software interrupts**.

List:

```
bash
kill -l
```

### IMPORTANT SIGNALS

Signal	Number	Meaning
SIGTERM	15	graceful stop
SIGKILL	9	force kill
SIGINT	2	Ctrl+C
SIGSTOP	19	pause
SIGCONT	18	resume

Signal	Number	Meaning
SIGHUP	1	reload

---

## SIGNAL FLOW

```
text  
kernel → process
```

Process may:

- handle
- ignore
- terminate

(SIGKILL cannot be ignored)

---

## ★ 7) kill — Terminate Processes

### BASIC USAGE

```
bash  
kill PID
```

(Default = SIGTERM)

---

### FORCE KILL

```
bash  
kill -9 PID
```

⚠ Last resort only.

---

### KILL BY NAME

```
bash  
pkill nginx
```

```
bash  
  
killall nginx
```

## PAUSE / RESUME

```
bash  
  
kill -STOP PID  
kill -CONT PID
```

## REAL-WORLD RULE

Always try:

```
bash  
  
kill PID
```

before:

```
bash  
  
kill -9
```

## ★ 8) NICENESS (PRIORITY)

### WHAT NICE IS

Controls **CPU scheduling priority**.

Range:

```
text  
  
-20 (highest) → 19 (lowest)
```

## START WITH PRIORITY

```
bash  
  
nice -n 10 command
```

## CHANGE RUNNING PROCESS

```
bash
renice -n 5 -p PID
```

Root required for negative values.

---

## REAL USE

Lower priority for backups:

```
bash
nice -n 15 tar ...
```

## ★ 9) PROCESS STATES

### COMMON STATES (STAT)

State	Meaning
R	Running
S	Sleeping
D	Uninterruptible IO
T	Stopped
Z	Zombie

### D STATE (IMPORTANT)

- Waiting for disk/network
- Cannot be killed
- Kernel-level wait

## DEBUGGING STUCK PROCESSES

If D state → IO problem, not CPU.

---

## ★ 10) /proc FILESYSTEM (KERNEL WINDOW)

### WHAT /proc IS

Virtual filesystem exposing **kernel data**.

---

## IMPORTANT FILES

```
bash  
  
/proc/cpuinfo  
/proc/meminfo  
/proc/uptime
```

## PROCESS-SPECIFIC

```
bash  
  
/proc/PID/cmdline  
/proc/PID/environ  
/proc/PID/status
```

## LIVE TUNING (ADVANCED)

```
bash  
  
echo 3 > /proc/sys/vm/drop_caches
```

(DO NOT do blindly)

---

## ★ 11) JOB CONTROL (SHELL-LEVEL)

### BACKGROUND / FOREGROUND

```
bash  
  
command &
```

### SUSPEND JOB

```
bash  
  
Ctrl + Z
```

### LIST JOBS

```
bash
```

```
jobs
```

## RESUME

```
bash
```

```
fg %1  
bg %1
```

## DISOWN

```
bash
```

```
disown
```

Process survives terminal close.

## REAL-WORLD

SSH disconnect safety:

```
bash
```

```
nohup command &  
disown
```

## FINAL PROCESS MENTAL MODEL

```
arduino
```

```
user
  └── shell
      ├── process
      │   ├── child
      │   └── child
      └── jobs
```

Everything is:

- forked
- scheduled
- signaled
- reaped

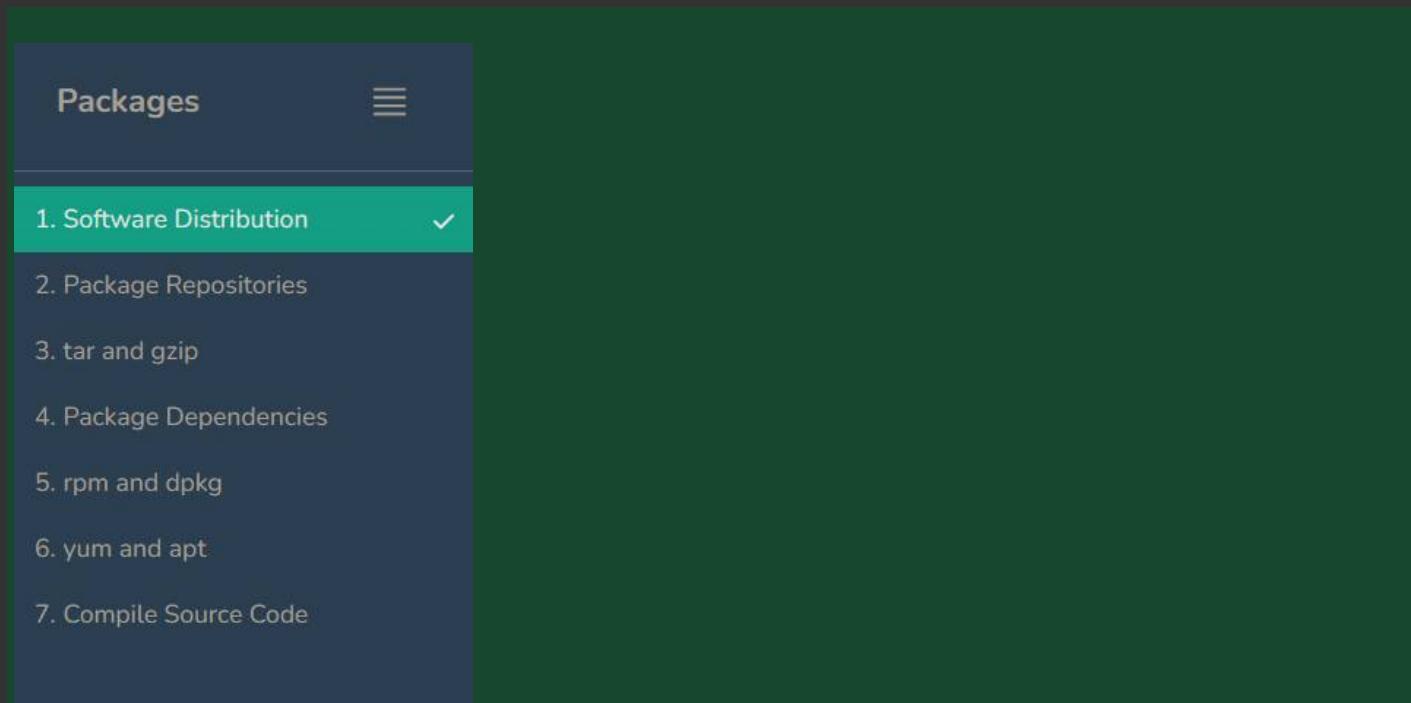
## YOU NOW UNDERSTAND

- ✓ Process lifecycle
- ✓ Signals & killing
- ✓ CPU scheduling
- ✓ Kernel introspection
- ✓ Job control

## NEXT (LAST CORE SYSTEM TOPICS)

1. **Memory management (free, vmstat, OOM)**
2. **Disk & filesystems**
3. **Networking (ss, ip, netstat)**
4. **System boot & services (systemd)**

Say **what's next** — you're now at **real Linux engineer level**.



ccae1317-458d-4e33-abf8-b9a08a8a68e0.png  
Image

next one , packages

Excellent — **Packages** is where Linux becomes *manageable at scale*.

We'll go **exactly in the order shown, basic → advanced, no command skipped, real admin workflows**.

## ★ 1) SOFTWARE DISTRIBUTION (FOUNDATION)

### WHAT “SOFTWARE DISTRIBUTION” MEANS

In Linux, software is delivered as:

- **Prebuilt binaries** (packages)
- **Source code** (compile yourself)

Handled by:

- Package formats
  - Repositories
  - Package managers
- 

### WHY PACKAGES EXIST

Problems they solve:

- Version control
- Dependency resolution
- Updates
- Security patches

Without packages → chaos.

---

### TYPES OF DISTRIBUTIONS

Distro	Package Format
Debian / Ubuntu	.deb
RHEL / CentOS	.rpm
Arch	pkg.tar.zst

---

### PACKAGE LIFECYCLE

```
text

build → package → repository → install → update → remove
```

## ★ 2) PACKAGE REPOSITORIES

# WHAT IS A REPOSITORY

A repository is:

- A server
  - Hosting signed packages
  - Indexed with metadata
- 

## REPO CONFIG LOCATIONS

### Debian-based

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

---

### RHEL-based

```
bash  
/etc/yum.repos.d/
```

---

## VIEW ENABLED REPOS

### Debian / Ubuntu

```
bash  
apt policy
```

---

### RHEL / CentOS

```
bash  
yum repolist
```

---

## ADD CUSTOM REPO (ADVANCED)

Example:

```
bash  
  
sudo add-apt-repository ppa:example/ppa
```

or manually add `.repo` file.

---

## SECURITY NOTE

Repositories are **GPG signed**.

Never trust unsigned repos.

---

## ★ 3) tar and gzip (ARCHIVES)

### WHAT tar IS

`tar` = Tape ARchiver

Bundles files (not compression).

---

### BASIC USAGE

Create archive:

```
bash  
  
tar -cf archive.tar file1 file2
```

Extract:

```
bash  
  
tar -xf archive.tar
```

---

### WITH COMPRESSION

#### gzip

```
bash  
  
tar -czf archive.tar.gz dir/  
tar -xzf archive.tar.gz
```

#### bzip2

```
bash  
  
tar -cjf archive.tar.bz2
```

## xz

```
bash  
  
tar -cJf archive.tar.xz
```

## VIEW WITHOUT EXTRACTING

```
bash  
  
tar -tf archive.tar.gz
```

## REAL-WORLD

Packages are often:

```
text  
  
software.tar.gz
```

## ★ 4) PACKAGE DEPENDENCIES

### WHAT ARE DEPENDENCIES

Libraries or tools a program **needs to run**.

Example:

```
text  
  
nginx → libc → kernel
```

### DEPENDENCY TYPES

- Runtime
- Build-time
- Optional (recommended)

## VIEW DEPENDENCIES

## Debian

```
bash  
apt show package
```

---

## RPM

```
bash  
rpm -qR package
```

---

## DEPENDENCY HELL (OLD DAYS)

Manual installs → broken systems.

Package managers solve this.

---

## ★ 5) rpm and dpkg (LOW-LEVEL TOOLS)

### WHAT THESE ARE

Low-level package tools:

- No dependency resolution
  - Raw install/remove
- 

### rpm (RHEL)

Install:

```
bash  
rpm -ivh package.rpm
```

Remove:

```
bash  
rpm -e package
```

Query:

```
bash  
rpm -qa
```

```
rpm -qi package
```

Verify:

```
bash
```

```
rpm -V package
```

## dpkg (Debian)

Install:

```
bash
```

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

Remove:

```
bash
```

```
dpkg -r package
```

List:

```
bash
```

```
dpkg -l
```

## WHEN TO USE THESE

- Debug broken package database
- Manual installs
- Recovery mode

## ★ 6) yum and apt (HIGH-LEVEL MANAGERS)

## WHY THESE MATTER

They:

- Resolve dependencies
- Fetch from repos
- Handle upgrades

## apt (Debian / Ubuntu)

Update index:

```
bash  
apt update
```

Install:

```
bash  
apt install nginx
```

Remove:

```
bash  
apt remove nginx  
apt purge nginx
```

Upgrade:

```
bash  
apt upgrade
```

Search:

```
bash  
apt search nginx
```

---

## yum / dnf (RHEL)

Install:

```
bash  
yum install nginx
```

Remove:

```
bash  
yum remove nginx
```

Update:

```
bash
```

```
yum update
```

Search:

```
bash
```

```
yum search nginx
```

## CLEANUP (IMPORTANT)

```
bash
```

```
apt autoremove  
yum autoremove
```

## ADVANCED

Download only:

```
bash
```

```
apt download package
```

Simulate:

```
bash
```

```
apt -s install package
```

## ★ 7) COMPILE SOURCE CODE

### WHEN YOU COMPILE

- No package available
- Custom flags
- Performance tuning

### STANDARD FLOW

```
bash
```

```
./configure  
make
```

```
make install
```

## CONFIGURE OPTIONS

```
bash
./configure --prefix=/opt/app
```

## BUILD DEPENDENCIES

```
bash
apt build-dep package
```

## UNINSTALL PROBLEM

Source installs **don't track files**.

Solution:

- Use `checkinstall`
- Or build your own `.deb/.rpm`

## FINAL PACKAGE MENTAL MODEL

```
go
repo → metadata → package manager → low-level tool → filesystem
```

Never skip layers unless necessary.

## YOU NOW MASTER

- ✓ Repositories
- ✓ Archives
- ✓ Dependency resolution
- ✓ rpm vs apt
- ✓ Source builds

## NEXT CORE SYSTEM TOPICS

1. **Systemd & services**
2. **Boot process**

3. Networking
4. Storage & filesystems
5. Containers (Docker)

Say the next module — we continue building **real Linux admin mastery**.

 Messages beyond this point are only visible to you

can you give me a project that covers all commands we have discussed from the very start

Absolutely.

Here's **one single, realistic project** that **forces you to use almost every command** we've discussed — from **ls** → **text processing** → **users** → **permissions** → **processes** → **packages**.

This is not toy-level.

This is "**junior Linux admin / DevOps intern**" level.

## PROJECT: Linux System Audit & Log Analysis Toolkit

### Goal

Build a **command-line toolkit** that:

- Audits users, permissions, packages, and processes
- Analyzes logs using text-fu
- Generates reports
- Simulates real sysadmin tasks

You'll use **ONLY Linux commands**, no Python, no fancy tools.

### PROJECT STRUCTURE

```
text

linux-audit/
├── logs/
│   └── access.log
├── reports/
└── scripts/
    ├── user_audit.sh
    ├── perm_audit.sh
    ├── process_audit.sh
    ├── package_audit.sh
    └── log_analysis.sh
 README.md
```

You'll create this **manually** using commands.

## ◆ PHASE 1: FILE & DIRECTORY BASICS (`ls`, `cd`, `mkdir`, `cp`, `mv`)

### Tasks

```
bash

mkdir linux-audit
cd linux-audit
mkdir logs reports scripts
touch README.md
```

Use:

- `ls -lah`
- `tree`
- `pwd`



`Skills covered`  
`ls`, `cd`, `mkdir`, `touch`, `tree`

## ◆ PHASE 2: TEXT VIEWING & CREATION (`cat`, `less`, `head`, `tail`)

### Create fake log file

```
bash

nano logs/access.log
```

Paste sample Apache/Nginx logs.

Analyze:

```
bash

cat logs/access.log
less logs/access.log
head -20 logs/access.log
tail -f logs/access.log
```



`Skills covered`  
`cat`, `less`, `head`, `tail`

## ◆ PHASE 3: TEXT-FU & LOG ANALYSIS (`grep`, `awk`, `sort`, `uniq`, `wc`)

### Count unique IPs

```
bash

awk '{print $1}' logs/access.log | sort | uniq -c | sort -nr
```

## Top 5 IPs

```
bash

awk '{print $1}' logs/access.log | sort | uniq -c | sort -nr | head -5
```

## Requests per HTTP code

```
bash

awk '{print $9}' logs/access.log | sort | uniq -c
```

## Requests containing errors

```
bash

grep "404" logs/access.log
```

Save results:

```
bash

> reports/log_summary.txt
```



Skills covered

`grep`, `awk`, `sort`, `uniq`, `wc`, pipes, redirection

## ♦ PHASE 4: USER MANAGEMENT (useradd, groupadd, su)

### Create audit users

```
bash

sudo groupadd auditors
sudo useradd audit1
sudo useradd audit2
sudo usermod -aG auditors audit1
```

Verify:

```
bash

id audit1
getent group auditors
```

Switch user:

```
bash

su - audit1
```

✓ Skills covered

`useradd, groupadd, usermod, id, su`

## ◆ PHASE 5: PERMISSIONS & OWNERSHIP (`chmod`, `chown`, `umask`)

### Restrict reports

```
bash  
  
sudo chown :auditors reports  
sudo chmod 770 reports
```

Check:

```
bash  
  
ls -ld reports
```

Test access as different users.

Advanced:

```
bash  
  
chmod g+s reports
```

Set default perms:

```
bash  
  
umask
```

✓ Skills covered

`chmod, chown, permissions, umask`

## ◆ PHASE 6: PROCESSES & JOB CONTROL (`ps`, `top`, `kill`, `jobs`)

### Monitor system

```
bash  
  
ps aux  
top
```

### Find heavy process

```
bash  
  
ps aux --sort=-%cpu | head
```

## Background job

```
bash

sleep 500 &
jobs
fg
bg
kill %1
```

## Kill by name

```
bash

pkill sleep
```

Inspect process:

```
bash

ls /proc/PID
```



Skills covered

`ps`, `top`, `kill`, `jobs`, `/proc`, backgrounding

## ◆ PHASE 7: PACKAGE MANAGEMENT (apt/yum, rpm/dpkg)

### Install tool

```
bash

sudo apt install tree
```

Verify:

```
bash

which tree
dpkg -l | grep tree
```

Remove:

```
bash

sudo apt remove tree
```

Inspect package:

```
bash

apt show tree
```

Extract package files (advanced):

```
bash
dpkg -L tree
```

Skills covered

**apt**, **dpkg**, packages, repositories

---

## ◆ PHASE 8: ARCHIVES & BACKUPS (tar, gzip)

### Backup reports

```
bash
tar -czf reports_backup.tar.gz reports/
```

Verify:

```
bash
tar -tf reports_backup.tar.gz
```

Restore:

```
bash
tar -xzf reports_backup.tar.gz
```

Skills covered

**tar**, compression, backups

---

## ◆ PHASE 9: SCRIPTING (combining everything)

### Example: scripts/log\_analysis.sh

```
bash
#!/bin/bash
LOG=../logs/access.log
OUT=../reports/log_report.txt

echo "Top IPs:" > $OUT
awk '{print $1}' $LOG | sort | uniq -c | sort -nr | head -5 >> $OUT

echo "HTTP Codes:" >> $OUT
awk '{print $9}' $LOG | sort | uniq -c >> $OUT
```

Make executable:

```
bash
chmod +x scripts/log_analysis.sh
```

Run:

```
bash
./scripts/log_analysis.sh
```

 Skills covered  
shebang, execution, redirection, permissions

## WHAT THIS PROJECT PROVES

By completing this, you can **honestly say**:

-  I can navigate Linux
-  I can analyze logs
-  I understand users & permissions
-  I manage processes
-  I handle packages
-  I automate tasks

This is **resume-grade**, not tutorial fluff.

## NEXT LEVEL (OPTIONAL HARD MODE)

- Add cron job to run audits daily
- Use **find** to detect world-writable files
- Add **sed** to sanitize logs
- Use **nice** to lower script priority
- Store reports per-date

If you want, next I can:

- Turn this into **step-by-step lab checklist**
- Add **interview questions based on this project**
- Create **Linux admin mock scenario**
- Convert this into **GitHub-ready project**

Just say the word.