Orientation — Shell & Getting Help (Core)

Linux Commands Course · Section 0

IDSchool

Terminal vs Shell

A **terminal** is the window where you type. A **shell** is the program that reads what you type and runs it (e.g., bash, zsh, fish).

You talk to the OS through the shell. This course uses a Bourne-style shell (bash/zsh).

Which shell am I using?

Print the value of the SHELL environment variable:

```
echo $SHELL
```

Typical outputs: `/bin/bash`, `/bin/zsh`.

---

bash and zsh — at a glance

- bash: ubiquitous default on many distros; great for scripts.
- zsh: interactive niceties (completion, prompts) while staying Bourne-compatible for most everyday commands.

You can learn one and be productive in both.

Prompt anatomy

A common prompt looks like this:

```
user@host:~$
```

- user – your account name
- host – machine name
- ~ – your home directory
- $ – normal user (# means root)

Command anatomy

Pattern you’ll see everywhere:

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

Example:

```
echo Hello
```

`echo` is the command; `Hello` is an argument printed to the screen.

---

echo — printing text

Print simple text:

```
echo Hello
```

Preserve spaces by quoting:

```
echo "Multiple words stay together"
```

Show special characters literally by single-quoting:

```
echo 'Use $ and * literally'
```

---

type vs which — what will run?

Discover how the shell resolves a name.

type (shell builtin) tells if something is a builtin, alias, function, or an external program:

```
type echo
```

which searches your PATH and shows the path to an external program:

```
which echo
```

If type says “builtin”, which may print nothing for that name.

Getting help — quick options

Many programs support a short help message:

```
echo --help
```

Bash builtins have builtin help:

```
help echo
```

Use these when you just need a brief synopsis and flags.

Manual pages (man)

Read full documentation for a command:

```
man echo
```

Navigation keys inside man:

- Space / Page Down – next page
- b / Page Up – previous page
- /pattern – search forward
- n / N – next / previous match
- q – quit

---

man sections (concept)

Manuals are grouped into sections (1: user cmds, 5: file formats, 8: admin, etc.).

Open a specific section if names clash:

```
man 1 printf
man 3 printf
```

(Only use if you encounter multiple entries.)

whatis and apropos

Show a one-line description for a command name:

```
whatis echo
```

Search across man page descriptions by keyword:

```
apropos print
```

Use apropos when you know the task but not the command name.

---

info pages

Some tools use the GNU Info system for their primary docs:

```
info coreutils
```

Navigation: Space → next, Backspace → previous, q → quit.

Session hygiene — history

List your recent commands with line numbers:

```
history
```

Press ↑ or ↓ to scroll through previous commands at the prompt. You can re-edit and re-run them quickly.

Session hygiene — clear & reset

Clear the visible screen contents:

```
clear
```

If your terminal display gets garbled (binary noise, weird characters), re-initialize it:

```
reset
```

`reset` is safe; it just redraws and resets modes.

Exit the shell

End the current shell session:

```
exit
```

Keyboard shortcut: **Ctrl+D** (sends End-Of-File to the shell).

Keyboard shortcuts (must-know)

Shortcut	What it does
Ctrl+C	Stop current running command
Ctrl+D	Exit shell or end input line
Ctrl+L	Clear screen (like `clear`)
↑ / ↓	Browse command history
Tab	Auto-complete names

Summary

- Shell: the interpreter you talk to (bash, zsh)
- Identify commands with type / which
- Learn quickly via --help, help, man, whatis, apropos, info
- Keep sessions tidy with history, clear, reset
- Exit cleanly with exit or **Ctrl+D**

Processes & Jobs (Core)

Linux Commands Course · Section 10

IDSchool

What Is a Process?

A **process** is a running instance of a program.

Every process has:

- A **PID** (Process ID)
- A **parent process**
- A **state** (running, sleeping, stopped, zombie)
- An **owner** and resource usage (CPU, memory)

All running processes form a hierarchy — you can view it at any time.

Viewing Processes – ps

ps lists running processes.

```
ps aux
```

Columns:

- USER – process owner
- PID – process ID
- %CPU, %MEM – resource usage
- STAT – process state
- COMMAND – what’s running

Example:

```
USER     PID %CPU %MEM COMMAND
root       1  0.0  0.1 systemd
student  213  0.1  0.5 bash
student  230  2.5  1.2 python3 script.py
```

Interactive Process Viewer – top

Displays a live updating view of running processes.

```
top
```

Common controls inside top:

- **P** – sort by CPU
- **M** – sort by memory
- **K** – kill a process by PID
- **Q** – quit

More colorful alternative (if installed): htop

```
htop
```

Use F6 to sort columns, F9 to kill, F10 to quit.

Finding Processes — pgrep and pkill

Search processes by name:

```
pgrep bash
```

This prints PIDs for all matching processes.

Kill by name (no need for PID):

```
pkill firefox
```

Force kill with signal 9 (SIGKILL):

```
pkill -9 firefox
```

Killing by PID — kill

Stop a process using its PID.

Example:

```
ps aux | grep sleep
kill 1234
```

Graceful termination (default SIGTERM 15):

```
kill 1234
```

Force kill if unresponsive:

```
kill -9 1234
```

Kill Multiple at Once — killall

Ends all processes with the same name.

```
killall python3
```

Useful for stopping multiple instances of a command quickly.

Adjusting Priority — nice

Each process has a **niceness** value (priority).
Lower value → higher priority.

Start a command with custom priority:

```
nice -n 10 script.sh
```

Default nice value = 0.
Range = **-20 (highest)** to **19 (lowest)**.

Changing Priority of Running Process — renice

Change niceness for an existing process:

```
renice +5 -p 2134
```

Example: lower CPU priority of a background task.

Only root can increase (raise) priority (negative nice values).

Foreground and Background Jobs

When you run a command normally, it runs in the **foreground**.

To run it in the **background**, append &:

```
sleep 60 &
```

Output example:

```
[1] 1234
```

[1] is the job number, 1234 is the PID.

Listing Jobs — jobs

Show jobs started from the current terminal:

```
jobs
```

Example output:

```
[1]+  Running  sleep 60 &
```

Jobs are tied to your shell session.

Bringing Jobs to Foreground or Background

Bring job to foreground:

```
fg %1
```

Send a suspended job to background again:

```
bg %1
```

Stop a running job temporarily with **Ctrl+Z**.

Disowning Jobs — disown

Detach a job from the current shell so it keeps running after you close the terminal.

```
disown %1
```

This removes it from the job table.

Example workflow:

```
sleep 300 &
disown %1
exit
```

The job keeps running even after logout.

Persistent Background Processes — nohup

nohup runs a command immune to hangups or logouts.

```
nohup long_task.sh &
```

Output is redirected to nohup.out by default.

Perfect for running long scripts or background services safely.

Signals Overview

Processes can receive signals (software interrupts).

Common ones:

Signal	Number	Meaning
SIGTERM	15	Graceful termination
SIGKILL	9	Force kill, cannot be trapped
SIGSTOP	19	Pause process
SIGCONT	18	Resume process

Send custom signals:

```
kill -STOP 2134   # pause
kill -CONT 2134   # resume
```

Combining Process Tools

Practical usage example:

```
ps aux | grep nginx
sudo systemctl restart nginx
pgrep nginx
pkill -HUP nginx
```

- View → restart → verify → reload gracefully

Monitoring Processes Efficiently

Show top memory users:

```
ps -eo pid,comm,%mem --sort=-%mem | head
```

Show tree of process relationships:

```
pstree -p | less
```

These are great for debugging and audits.

Recap

- View: `ps aux`, `top`, `htop`
- Find / kill: `pgrep`, `pkill`, `kill`, `killall`
- Adjust priority: `nice`, `renice`
- Manage jobs: `&`, `jobs`, `fg`, `bg`, `disown`, `nohup`
- Know signals: graceful vs forceful termination

Mastering these makes you fluent in process control and multitasking.

Services, Boot & Logs (Core)

Linux Commands Course · Section 11

IDSchool

What Is systemd?

systemd is the default **init system** on most modern Linux distributions.

It manages:

- Service startup and shutdown
- Boot targets (runlevels)
- System logging (via journalctl)
- Time and clock synchronization

All of this is unified under the systemctl command.

Managing Services — systemctl

Check service status:

```
systemctl status nginx
```

Start, stop, or restart a service:

```
sudo systemctl start nginx
sudo systemctl stop nginx
sudo systemctl restart nginx
```

Enable service to start at boot:

```
sudo systemctl enable nginx
```

Disable service at boot:

```
sudo systemctl disable nginx
```

List all active services:

```
systemctl list-units --type=service
```

Inspecting All Units

A “unit” can be a service, device, socket, or timer.

List failed units:

```
systemctl --failed
```

List all (loaded) units:

```
systemctl list-units
```

Viewing Boot Targets

A **target** defines which services and environment are active — like traditional runlevels.

Show the current target:

```
systemctl get-default
```

Common targets:

- `graphical.target` — GUI mode
- `multi-user.target` — multi-user text mode
- `rescue.target` — maintenance mode

Switch (temporarily) to another target:

```
sudo systemctl isolate multi-user.target
```

Set the default boot target permanently:

```
sudo systemctl set-default graphical.target
```

System Time Management — timedatectl

Display current date, time, and time zone:

```
timedatectl
```

Set the system time zone:

```
sudo timedatectl set-timezone Europe/Baku
```

Enable NTP (Network Time Protocol) synchronization:

```
sudo timedatectl set-ntp true
```

This ensures automatic time syncing with internet servers.

Service Logs — `journalctl`

`journalctl` reads logs from the `systemd` journal — a binary log database maintained by `systemd-journald`.

Show all logs:

```
journalctl
```

Show logs for a specific service:

```
journalctl -u nginx
```

View logs since the last boot:

```
journalctl -b
```

Filter by time:

```
journalctl --since "2025-10-21 12:00" --until "2025-10-21 14:00"
```

Follow live logs (like `tail -f`):

```
journalctl -f
```

Filtering by Priority

Show only errors:

```
journalctl -p err
```

Show warnings and higher:

```
journalctl -p warning
```

Priority levels range from 0 (emerg) to 7 (debug).

Classic Log Files – /var/log

Older and non-systemd logs still live under /var/log.

Common log files:

File	Description
/var/log/syslog	General system activity (Debian/Ubuntu)
/var/log/messages	General system log (RHEL/Fedora)
/var/log/auth.log	Authentication and sudo logs
/var/log/dmesg	Kernel messages during boot
/var/log/nginx/	Web server logs
/var/log/secure	Security messages (RHEL-based)

Inspect with standard tools:

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

Boot Diagnostics

View boot performance and failures:

```
systemd-analyze
systemd-analyze blame
```

See which services delayed boot and how long startup took.

Reboot logs only:

```
journalctl -b -1
```

(-b -1 means previous boot.)

Combining Tools

Practical example — check a web server status, restart it, and read its logs:

```
sudo systemctl status nginx
sudo systemctl restart nginx
journalctl -u nginx --since today
```

You’ll often use systemctl and journalctl together when troubleshooting.

Recap

- **Services:** manage with `systemctl start/stop/restart/status`
- **Boot control:** `systemctl get-default`, `isolate`, `set-default`
- **Time management:** `timedatectl`
- **Logs:** use `journalctl` and `/var/log/` for full visibility

Together, these tools give total control over system services and events.

Networking (Core)

Linux Commands Course · Section 12

IDSchool

Network Interfaces — ip a

Show all network interfaces and their IP addresses:

```
ip a
```

Example output:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet 192.168.1.10/24 brd 192.168.1.255 scope global eth0
```

- eth0, wlan0 — interface names
- inet — IPv4 address
- inet6 — IPv6 address
- state — interface status (UP/DOWN)

Bring an interface up or down (root required):

```
sudo ip link set eth0 up
sudo ip link set eth0 down
```

Routing Table — ip r

View the system routing table:

```
ip r
```

Example output:

```
default via 192.168.1.1 dev eth0
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.10
```

- `default via` → default gateway
- `dev eth0` → which interface is used
- `src` → local source IP

Add or delete temporary routes:

```
sudo ip route add 10.10.0.0/16 via 192.168.1.1
sudo ip route del 10.10.0.0/16
```

Active Connections — ss (modern tool)

ss (socket statistics) shows open ports and connections.

```
ss -tulpn
```

- t → TCP
- u → UDP
- l → listening sockets
- p → show process using port
- n → show numeric addresses

Example:

```
Netid State  Recv-Q Send-Q Local Address:Port  Peer Address:Port  Process
tcp   LISTEN 0      128    0.0.0.0:22        0.0.0.0:*          users:(("sshd",pid=745,fd=3))
```

Legacy tool (if available):

```
netstat -tulpn
```

Connectivity — ping

Test reachability of a host.

```
ping 8.8.8.8
```

Send a limited number of packets:

```
ping -c 4 example.com
```

Interrupt anytime with **Ctrl+C**.

Tracing Network Path — traceroute / tracepath

Show each hop between you and a destination.

```
traceroute example.com
```

If not installed, try:

```
tracepath example.com
```

Output shows latency at each hop — useful for debugging routing or latency issues.

DNS Lookups — dig and host

▒▒▒▒ Query DNS records with dig

```
dig example.com
```

Show only the IP address:

```
dig +short example.com
```

Query specific record types:

```
dig example.com MX
dig example.com NS
```

▒▒▒▒ Simple lookup with host

```
host example.com
```

Reverse lookup (IP → hostname):

```
host 8.8.8.8
```

HTTP & File Transfers — curl and wget

▒ curl

Fetch a URL or API data:

```
curl https://example.com
```

Save output to a file:

```
curl -o page.html https://example.com
```

Show headers only:

```
curl -I https://example.com
```

Send JSON data to an API:

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"test"}' https://api.example.com/data
```

▒ wget

Download files from the web:

```
wget https://example.com/file.iso
```

Resume interrupted download:

```
wget -c https://example.com/file.iso
```

Remote Access — ssh

Securely log into another machine:

```
ssh user@192.168.1.50
```

Use a key file instead of a password:

```
ssh -i ~/.ssh/id_rsa user@host
```

Exit remote session with `exit` or **Ctrl+D**.

Copy files securely using SSH:

```
scp report.txt user@192.168.1.50:/home/user/
```

Copy entire directories recursively:

```
scp -r project/ user@host:/backup/
```

Legacy Tool — telnet

Used for basic connectivity testing (not secure).

```
telnet example.com 80
```

If it connects, the port is open.

Use only for debugging — not for remote login.

NetworkManager CLI — nmcli

nmcli manages network connections on systems using **NetworkManager**.

List all connections:

```
nmcli connection show
```

Show active interfaces:

```
nmcli device status
```

Bring a connection up or down:

```
sudo nmcli connection up "Wired connection 1"
sudo nmcli connection down "Wired connection 1"
```

View details for a specific interface:

```
nmcli device show eth0
```

Set static IP (example):

```
sudo nmcli connection modify "Wired connection 1" ipv4.addresses 192.168.1.20/24 ipv4.gateway 192.168.1.1 ipv4.method manual
sudo nmcli connection up "Wired connection 1"
```

Recap

- **ip a**, **ip r** – view interfaces and routes
- **ss -tulpn** – active sockets and ports
- **ping**, **traceroute**, **tracepath** – connectivity testing
- **dig**, **host** – DNS queries
- **curl**, **wget** – HTTP and file transfers
- **ssh**, **scp**, **telnet** – remote access and copy
- **nmcli** – manage connections via NetworkManager

These tools form the backbone of network troubleshooting and configuration.

Packages & Software Management (Core)

Linux Commands Course · Section 13

IDSchool

What Are Packages?

A **package** is a compressed bundle that contains:

- Program files (binaries, libraries, icons)
- Configuration files
- Metadata (version, dependencies, maintainer info)

Instead of manually copying files, the **package manager** handles installation, updates, and removal automatically.

Where Packages Come From — Repositories

Linux distributions host packages on remote **repositories (repos)** — organized servers containing signed software.

Each system has a list of repositories stored in config files, such as:

- `/etc/apt/sources.list` (Debian/Ubuntu)
- `/etc/yum.repos.d/` (RHEL/Fedora)
- `/etc/pacman.conf` (Arch)
- `/etc/zypp/repos.d/` (openSUSE)

The package manager connects to these servers to download and verify packages.

APT (Advanced Package Tool) — Debian/Ubuntu

APT is the package manager used by **Debian**, **Ubuntu**, and their derivatives (Mint, Kali, etc.).

Updating Package Information

Before installing anything, update your local list of available software:

```
sudo apt update
```

This syncs your system with the repository metadata — names, versions, and dependencies.

Then upgrade installed software:

```
sudo apt upgrade
```

- `apt update` → refreshes the list
- `apt upgrade` → installs newer versions of already-installed packages

To upgrade all packages and remove obsolete ones:

```
sudo apt full-upgrade
```

Installing Packages

Install one or multiple packages:

```
sudo apt install curl vim git
```

APT automatically downloads dependencies and installs them.

Install a specific version:

```
sudo apt install nginx=1.18.0-0ubuntu1
```

Removing Packages

Remove a package but keep its config files:

```
sudo apt remove nginx
```

Remove a package **and** its configs:

```
sudo apt purge nginx
```

Clean up unnecessary packages and cache:

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

Inspecting Packages

Check if a package is installed:

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

Show detailed info:

```
apt show nginx
```

List files installed by a package:

```
dpkg -L nginx
```

Find which package a file belongs to:

```
dpkg -S /usr/bin/ls
```

Installing Local .deb Files — dpkg

Install a .deb file manually (downloaded from a website):

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

If dependencies are missing, fix them with:

```
sudo apt -f install
```

This tells APT to install the required packages automatically.

RHEL/Fedora — `dnf` and `rpm`

`dnf` (successor to `yum`) is used on **RHEL**, **Fedora**, and **CentOS**.

Install software:

```
sudo dnf install nginx
```

Remove software:

```
sudo dnf remove nginx
```

Update all packages:

```
sudo dnf update
```

Query package info:

```
dnf info nginx
```

Manual package management via RPM:

```
sudo rpm -qi nginx    # info
sudo rpm -ql nginx    # list files
sudo rpm -ivh file.rpm  # install
sudo rpm -e package    # remove
```

Arch Linux — pacman

The **pacman** package manager uses `.pkg.tar.zst` packages from Arch repositories.

Update repository and system in one command:

```
sudo pacman -Syu
```

Install a package:

```
sudo pacman -S firefox
```

Remove a package:

```
sudo pacman -R firefox
```

Search for packages:

```
pacman -Ss python
```

View info:

```
pacman -Qi firefox
```

openSUSE — zypper

zypper is the package tool for **openSUSE** and **SLE** systems.

Refresh repositories:

```
sudo zypper refresh
```

Install packages:

```
sudo zypper in vim
```

Remove packages:

```
sudo zypper rm vim
```

Update system:

```
sudo zypper up
```

Show package info:

```
zypper info vim
```

Universal Package Systems

Some distributions support **universal formats** — portable across distros.

Snap

Developed by Canonical, runs sandboxed applications.

List installed snaps:

```
snap list
```

Install a snap package:

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

Remove a snap:

```
sudo snap remove code
```

Flatpak

Another universal, sandboxed format.

Install a Flatpak application:

```
flatpak install flathub org.gimp.GIMP
```

Run a Flatpak app:

```
flatpak run org.gimp.GIMP
```

List installed Flatpaks:

```
flatpak list
```

Comparing Package Managers

Distro	Tool	Install Example	Notes
Debian/Ubuntu	`apt`	`apt install nginx`	Most common; uses `.deb`
RHEL/Fedora	`dnf`	`dnf install nginx`	Uses `.rpm`
Arch	`pacman`	`pacman -S nginx`	Very fast, rolling updates
openSUSE	`zypper`	`zypper in nginx`	Enterprise-grade
Universal	`snap`, `flatpak`	Cross-platform apps	Great for desktop software

Recap

- **APT** – update, install, remove, purge, inspect packages
- **Repositories** – centralized sources of verified software
- **dpkg** – for manual .deb installs
- **dnf / rpm**, **pacman**, **zypper** – alternatives for other distros
- **snap / flatpak** – universal sandboxed packages

Mastering package management makes system maintenance fast, secure, and reliable.

Disks, Partitions & Filesystems (Core)

Linux Commands Course · Section 14

IDSchool

Storage Concepts

- **Disk** → physical device (e.g., `/dev/sda`, `/dev/nvme0n1`)
- **Partition** → logical segment of a disk (e.g., `/dev/sda1`)
- **Filesystem** → structure that defines how data is stored (e.g., ext4, xfs, btrfs)

Linux uses a single unified directory tree — all disks and partitions get *mounted* somewhere under `/`.

Listing Storage Devices — lsblk

List block devices (disks, partitions, LVM volumes).

```
lsblk -f
```

Example output:

```
NAME   FSTYPE LABEL  UUID                                 MOUNTPOINT
sda
├─sda1 ext4   root   21f0-4c3f                            /
├─sda2 swap   swap   a1b2c3d4-e5f6-7890-1122-334455667788 [SWAP]
```

- NAME → device name
- FSTYPE → filesystem type
- MOUNTPOINT → where it’s mounted

Display UUIDs and Filesystem Info — blkid

Show filesystem type and unique IDs.

```
sudo blkid
```

Example:

```
/dev/sda1: UUID="21f0-4c3f" TYPE="ext4" PARTLABEL="root"
/dev/sda2: UUID="a1b2c3d4" TYPE="swap"
```

UUIDs are used in /etc/fstab for stable device mounting.

Check Disk Usage — df -h

View mounted filesystems and their space usage.

```
df -h
```

Example:

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        50G   20G   28G  42% /
tmpfs           2.0G  2.0M  2.0G   1% /run
```

-h makes sizes human-readable.

Directory Usage — du

Show how much space files and directories take.

```
du -sh *
```

- -s → summarize totals
- -h → human-readable sizes

Example output:

```
4.0K  Documents
1.2G  Downloads
400M  Pictures
```

Great for finding large folders.

Mounting a Filesystem — mount

Mount a device (attach it to a directory).

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

Verify with:

```
df -h | grep sdb1
```

Unmount when done:

```
sudo umount /mnt
```

Mount read-only:

```
sudo mount -o ro /dev/sdb1 /mnt
```

Persistent Mounts — /etc/fstab

/etc/fstab defines filesystems that auto-mount at boot.

View file:

```
cat /etc/fstab
```

Example entry:

```
UUID=21f0-4c3f  /data  ext4  defaults  0 2
```

Fields:

1. Device or UUID
2. Mount point
3. Filesystem type
4. Options (defaults, ro, noatime, etc.)
5. Dump (backup flag)
6. fsck order (1=root, 2=others)

Partitioning (Demo Only!) — fdisk and parted

Use **only with care** — modifying partitions can erase data!

List disks and partitions:

```
sudo fdisk -l
```

Interactive mode (dangerous!):

```
sudo fdisk /dev/sdb
```

For modern disks (>2TB) use `parted`:

```
sudo parted /dev/sdb
```

You can view, create, and delete partitions. Always unmount before modifying.

Creating a Filesystem — mkfs

Format a partition with a filesystem.

Example (ext4):

```
sudo mkfs.ext4 /dev/sdb1
```

Other examples:

```
sudo mkfs.xfs /dev/sdb1
sudo mkfs.vfat /dev/sdb1
```

Check before formatting to avoid destroying data!

Filesystem Check — fsck

Scans and repairs filesystem errors.

```
sudo fsck /dev/sdb1
```

Run only on **unmounted** filesystems.

You can auto-confirm fixes with `-y`:

```
sudo fsck -y /dev/sdb1
```

Filesystem Tuning — tune2fs

View or modify filesystem parameters (ext filesystems).

```
sudo tune2fs -l /dev/sda1
```

Example adjustments:

```
sudo tune2fs -m 1 /dev/sda1   # reserve 1% space for root
sudo tune2fs -c 0 /dev/sda1   # disable auto-check by mount count
```

Resizing Filesystems — resize2fs

Resize an **ext** filesystem after adjusting partition size.

Shrink (offline only):

```
sudo resize2fs /dev/sdb1 20G
```

Expand to fill available space:

```
sudo resize2fs /dev/sdb1
```

Run after resizing partition with `fdisk` or `parted`.

Swap Space — Virtual Memory

Linux uses **swap** as overflow for RAM.

Enable swap area:

```
sudo swapon /dev/sda2
```

Disable it:

```
sudo swapoff /dev/sda2
```

Show current swap usage:

```
swapon --show
```

Or view via `free -h`.

Creating a Swap File (Alternative)

If no swap partition exists, create one as a file.

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

Make it permanent in /etc/fstab:

```
/swapfile none swap sw 0 0
```

Recap

- Inventory: `lsblk`, `blkid`, `df -h`, `du -sh *`
- Mounting: `mount`, `umount`, `/etc/fstab`
- Partitioning (demo): `fdisk`, `parted`
- Filesystems: `mkfs`, `fsck`, `tune2fs`, `resize2fs`
- Swap: `swapon`, `swapoff`, `/swapfile`

These commands form the foundation of disk and storage management in Linux.

Scheduling (Core)

Linux Commands Course · Section 15

IDSchool

What Is Job Scheduling?

Linux can run commands automatically at specific times or intervals.

Two main tools handle this:

- **cron** → recurring jobs (daily, hourly, weekly, etc.)
- **at** → one-time jobs

The scheduler runs in the background and executes tasks even if you’re not logged in.

Recurring Jobs — cron

cron reads scheduled jobs from special files called **crontabs**.

List current user’s scheduled jobs:

```
crontab -l
```

Edit your crontab:

```
crontab -e
```

Each line defines one job using this format:

```
* * * * * command_to_run
│ │ │ │ │
│ │ │ │ └── Day of week (0–7)  (Sunday = 0 or 7)
│ │ │ └──── Month (1–12)
│ │ └────── Day of month (1–31)
│ └──────── Hour (0–23)
└────────── Minute (0–59)
```

Example: run a script every day at 2:30 AM

```
30 2 * * * /home/student/backup.sh
```

Special Cron Keywords

You can use shortcuts instead of the 5-field format:

Keyword	Meaning
@reboot	once at startup
@daily	once a day
@hourly	every hour
@weekly	once a week
@monthly	once a month

Example:

```
@reboot /usr/local/bin/monitor.sh
@daily  /usr/local/bin/cleanup.sh
```

System-Wide Cron Directories

In addition to user crontabs, system-wide jobs live in these directories:

Location	Purpose
/etc/crontab	main system cron file
/etc/cron.hourly/	scripts run every hour
/etc/cron.daily/	scripts run daily
/etc/cron.weekly/	scripts run weekly
/etc/cron.monthly/	scripts run monthly

System crontab includes an extra field for the **user** to run as:

```
# m h dom mon dow user command
17 * * * * root run-parts /etc/cron.hourly
```

Controlling Cron Jobs

List cron service status (systemd-based systems):

```
systemctl status cron
```

Restart it if needed:

```
sudo systemctl restart cron
```

You can temporarily disable user cron jobs by commenting them out in `crontab -e`.

Viewing Cron Logs

Cron logs are usually stored under /var/log.

```
sudo grep CRON /var/log/syslog
```

Or for Red Hat-based systems:

```
sudo grep CROND /var/log/cron
```

You can also redirect cron job output manually in your job definition:

```
0 1 * * * /usr/local/bin/backup.sh >> /var/log/backup.log 2>&1
```

One-Shot Jobs — at

Use at for tasks you want to run **once in the future**.

Make sure the atd service is running:

```
sudo systemctl enable --now atd
```

Schedule a job:

```
at 14:00
```

Then type your command(s):

```
echo "System check complete" >> /tmp/check.log
Ctrl+D
```

View scheduled jobs:

```
atq
```

Remove a scheduled job:

```
atrm <job_number>
```

Flexible Time Syntax with at

Examples of valid scheduling times:

```
at now + 1 hour
at midnight
at 8pm tomorrow
at 10:30am next Monday
```

at is perfect for one-off delayed commands or testing automation tasks.

Examples — Real Use Cases

Daily backup with cron:

```
0 2 * * * /usr/local/bin/backup.sh
```

Run maintenance 5 minutes from now with at:

```
echo "apt update && apt upgrade -y" | at now + 5 minutes
```

Weekly report via email:

```
0 9 * * 1 /usr/local/bin/report.sh | mail -s "Weekly Report" admin@example.com
```

Recap

- **cron** – recurring tasks (`crontab -e`, `/etc/cron.*`)
- **at** – one-time jobs (`at`, `atq`, `atrm`)
- **systemctl status cron / atd** – ensure schedulers are active
- Use log redirection for auditing outputs

Automation keeps your system consistent, efficient, and hands-free.

Bash Scripting (Core → Plus)

Linux Commands Course · Section 16

IDSchool

Your First Script — Shebang

A **shebang** tells the system which interpreter to run.

```
#!/usr/bin/env bash
echo "Hello from a script"
```

Save as `hello.sh`, make executable, then run:

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

`/usr/bin/env bash` finds `bash` via your PATH.

Safer Defaults for Scripts

Use defensive options to catch errors early.

```
#!/usr/bin/env bash
set -Eeu -o pipefail
IFS=$'
    '
```

- `-e` → exit on error
- `-u` → treat unset variables as errors
- `-o pipefail` → fail a pipeline if any command fails
- `IFS` set to safe defaults for word-splitting

Variables and Expansion

Assign and use variables:

```
name="Alice"
echo "Hello, $name"
```

Braced expansion avoids ambiguity:

```
echo "File: ${name}.txt"
```

Length of a variable:

```
echo "Name length: ${#name}"
```

Default values and replacement:

```
echo "${user:-anonymous}"       # default if unset/empty
echo "${path//:/ }"             # replace all ':' with space
```

Command substitution:

```
today="$(date +%F)"
echo "Today is $today"
```

Quoting Rules

Single quotes **prevent** expansion; double quotes **allow** it.

```
echo '$HOME literally'
echo "HOME is $HOME"
```

Use quotes to keep whitespace intact and avoid globbing surprises.

Arithmetic

Use (( )) for integer arithmetic.

```
a=5; b=7
(( sum = a + b ))
echo "$sum"
```

Exit status of arithmetic context:

```
if (( sum > 10 )); then echo "big"; fi
```

If / Elif / Else

```
read -r score
if (( score >= 90 )); then
  echo "A"
elif (( score >= 80 )); then
  echo "B"
else
  echo "Keep going"
fi
```

Test files/strings with `[[ ]]`:

```
file="report.txt"
if [[ -f "$file" && -s "$file" ]]; then
  echo "File exists and is non-empty"
fi
```

Case Statements

```
read -r ans
case "$ans" in
  yes|y|Y) echo "proceed" ;;
  no|n|N)  echo "abort" ;;
  *)       echo "unknown" ;;
esac
```

Great for parsing modes and flags.

Loops — for / while / until

```
for i in 1 2 3; do echo "$i"; done
```

Read a file line-by-line safely:

```
while IFS= read -r line; do
  printf '%s
' "$line"
done < input.txt
```

Repeat until condition becomes true:

```
count=0
until (( count >= 3 )); do
  echo "$count"
  ((count++))
done
```

Functions, Scope, and Exit Codes

Define functions and return statuses.

```
greet() { printf 'Hi, %s
' "$1"; }

greet "Alice"
```

Check exit codes via $? or with || and &&:

```
cp source.txt dest.txt && echo "ok" || echo "copy failed"
```

Local variables:

```
sum() { local a="$1" b="$2"; echo $((a+b)); }
```

Positional Parameters and $@

```
# script.sh
echo "Script: $0"
echo "Arg count: $#"
for arg in "$@"; do
  echo "-> $arg"
done
```

Quote "$@" to preserve args with spaces.

Reading Input — read and getopts

Read a line from stdin:

```
read -r name
echo "Hello, $name"
```

Parse flags with getopts:

```
#!/usr/bin/env bash
while getopts ":f:n:" opt; do
  case "$opt" in
    f) file="$OPTARG" ;;
    n) name="$OPTARG" ;;
    *) echo "Usage: $0 -f FILE -n NAME" >&2; exit 2 ;;
  esac
done
echo "file=$file name=$name"
```

Arrays and Associative Arrays

Indexed arrays:

```
nums=(10 20 30)
echo "${nums[1]}"
echo "${#nums[@]}"      # length
echo "${nums[@]}"       # iterate values
```

Associative arrays (Bash 4+):

```
declare -A user=( [name]="Alice" [role]="admin" )
echo "${user[name]}"
for k in "${!user[@]}"; do echo "$k=${user[$k]}"; done
```

printf — Safer Output

Prefer printf over echo for predictable formatting.

```
printf 'User: %s  Score: %03d
' "Alice" 7
```

Traps and Cleanup

Run code on exit or on specific signals.

```
tmp="$(mktemp -d)"
cleanup(){ rm -rf "$tmp"; }
trap cleanup EXIT INT TERM

echo "Working in $tmp"
sleep 1
```

This ensures resources are cleaned up even if interrupted.

Here-Docs and Process Substitution

Here-doc to create files inline:

```
cat <<'EOF' > script.sh
#!/usr/bin/env bash
echo "inline"
EOF
```

Process substitution to compare streams without temp files:

```
diff <(sort a.txt) <(sort b.txt)
```

Subshells vs Current Shell

Subshell inherits environment but not variable changes back to parent.

```
pwd; (cd /tmp; pwd); pwd
```

Use `{ }` for grouping without subshell:

```
{ echo one; echo two; } > out.txt
```

Error Handling Patterns

Immediate exit on failure (already enabled via set -e).
For guarded steps, use || with messages:

```
mkdir -p /data || { echo "cannot create /data" >&2; exit 1; }
```

Time a block:

```
start=$(date +%s); sleep 1; end=$(date +%s); echo "took $((end-start))s"
```

Reusable Script Template

```bash
#!/usr/bin/env bash
set -Eeuo pipefail
IFS=$'
    '

usage(){ echo "Usage: $0 -i INPUT -o OUTPUT"; }

input="" output=""
while getopts ":i:o:" opt; do
  case "$opt" in
    i) input="$OPTARG" ;;
    o) output="$OPTARG" ;;
    *) usage; exit 2 ;;
  esac
done

[[ -f "$input" && -n "$output" ]] || { usage; exit 2; }

tmp="$(mktemp)"; trap 'rm -f "$tmp"' EXIT
cp "$input" "$tmp"
printf 'Processed %s -> %s
' "$input" "$output"
mv "$tmp" "$output"
```

Testing and Formatting

Lint scripts with **shellcheck**:

```
shellcheck script.sh
```

Auto-format with **shfmt**:

```
shfmt -w script.sh
```

Install via your package manager or from upstream releases.

Recap

- Shebang and safe defaults make scripts portable and robust
- Quoting, expansion, and command substitution are fundamentals
- Use control flow, functions, and arrays for structure
- Parse input with `getopts`; clean up with `trap`
- Process substitution and here-docs enable elegant workflows
- Use `shellcheck` and `shfmt` to maintain quality

Environment & Customization (Plus)

Linux Commands Course · Section 17

IDSchool

What Is the Shell Environment?

When you start a shell session, it loads **configuration files** that define your environment.

They control:

- Which variables are set (e.g., PATH)
- Which aliases and functions are available
- How your prompt looks
- Which scripts run at login or for new terminals

Profile and RC Files Overview

File	Loaded When	Purpose
~/.bash_profile	login shells	Personal startup settings
~/.bashrc	interactive shells	Aliases, functions, variables
~/.profile	login shells (if no .bash_profile)	Environment setup
/etc/profile	system-wide login setup	Default for all users
/etc/profile.d/*.sh	system-wide scripts	Extend /etc/profile

Login vs Non-Login Shells

- **Login shell** → first shell after login (e.g., via console or SSH).
  - Reads /etc/profile → ~/.bash_profile → optionally ~/.bashrc.
- **Non-login shell** → when opening a new terminal window or running a script.
  - Reads ~/.bashrc only.

You can make .bash_profile load .bashrc manually:

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

Editing .bashrc

Add custom commands, aliases, and variables.

Example entries:

```
# Custom aliases
alias ll='ls -lh --color=auto'
alias grep='grep --color=auto'

# Custom PATH
export PATH="$PATH:$HOME/scripts"

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

After editing, reload it:

```
source ~/.bashrc
```

Environment Variables — export

List environment variables:

```
printenv
```

Set a variable for current session:

```
MYVAR="hello"
echo $MYVAR
```

Make it available to child processes:

```
export MYVAR="hello"
```

Unset a variable:

```
unset MYVAR
```

Permanent exports belong in `.bashrc` or `.profile`.

The PATH Variable

PATH defines where the shell looks for executables.

View it:

```
echo $PATH
```

Add a new directory to PATH (for current session):

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

To make it permanent, add the export line to your .bashrc.

Check where a command is found:

```
which python
```

Aliases — Shortcuts for Commands

Create simple command shortcuts.

```
alias cls='clear'
alias update='sudo apt update && sudo apt upgrade -y'
```

View all aliases:

```
alias
```

Remove an alias:

```
unalias cls
```

For persistence, define them in `~/.bashrc`.

Command Completion — bash-completion

bash-completion provides smart tab-completion for many commands.

Check if it’s installed:

```
type _init_completion
```

Install if missing (Debian/Ubuntu):

```
sudo apt install bash-completion
```

Then source it in your .bashrc (if not already):

```
[[ $PS1 && -f /usr/share/bash-completion/bash_completion ]] &&     . /usr/share/bash-completion/bash_completion
```

Now commands like git, docker, and ssh autocomplete intelligently.

History Behavior – Environment Variables

Customize how Bash records your command history.

▒ HISTCONTROL

Defines how duplicates and leading spaces are handled.

```
export HISTCONTROL=ignoredups:ignorespace
```

Options:

- ignoredups – skip duplicate commands
- ignorespace – don’t save commands starting with a space

▒ HISTTIMEFORMAT

Adds timestamps to history entries.

```
export HISTTIMEFORMAT="%F %T "
```

View history with time:

```
history
```

Other Useful History Variables

Variable	Description
HISTSIZE	number of commands kept in memory
HISTFILESIZE	number of lines kept in history file
HISTFILE	path to history file (usually ~/.bash_history)
HISTIGNORE	pattern list to skip saving certain commands

Example:

```
export HISTSIZE=5000
export HISTIGNORE="ls:cd:exit"
```

Making Persistent Customizations

When satisfied with your customizations:

```
source ~/.bashrc
```

To apply system-wide for all users, use `/etc/profile.d/custom.sh`:

```
sudo nano /etc/profile.d/custom.sh
```

Example:

```
export PATH="$PATH:/opt/tools"
alias ll='ls -lh --color=auto'
```

This will auto-load for all users.

Recap

- **Startup files:** `.bashrc`, `.bash_profile`, `.profile`, `/etc/profile`
- **Variables:** use `export` to persist and `PATH` to find commands
- **Aliases:** make common tasks faster
- **bash-completion:** improves workflow
- **History tuning:** timestamped and filtered history improves traceability

Customizing your shell makes Linux truly *yours*.

System Information & Troubleshooting (Plus)

Linux Commands Course · Section 18

IDSchool

System Facts — uname, hostnamectl, lsb_release

▒▒▒▒ Kernel and architecture

```
uname -a
```

Example output:

```
Linux workstation 6.8.0-45-generic #1 SMP x86_64 GNU/Linux
```

Displays kernel name, version, architecture, and OS type.

▒▒▒▒ Host identity

```
hostnamectl
```

Output example:

```
   Static hostname: workstation
         Icon name: computer-laptop
           Chassis: laptop
        Machine ID: 3c68f3e8d9284f2d8b22a
           Boot ID: 86b9e1c9bbd64c83bb7e2
  Operating System: Ubuntu 24.04 LTS
            Kernel: Linux 6.8.0-45-generic
      Architecture: x86-64
```

▒▒▒▒ Distribution info

```
lsb_release -a
```

or

```
cat /etc/os-release
```

Shows distribution name, release, and codename.

Quick Health Snapshot

Uptime and load

```
uptime
```

Output:

```
10:25:41 up 3 days,  2:41,  3 users,  load average: 0.20, 0.25, 0.18
```

Shows system uptime and average CPU load over 1, 5, and 15 minutes.

Memory usage

```
free -h
```

Example:

```
              total        used        free      shared  buff/cache   available
Mem:           15Gi       3.5Gi       9.4Gi       256Mi       2.2Gi        11Gi
Swap:          2.0Gi       0.0Gi       2.0Gi
```

System performance overview

```
vmstat 2 5
```

- Displays CPU, memory, swap, and I/O stats every 2 seconds (5 times).

Disk I/O stats

```
iostat -x 2 3
```

- Requires `sysstat` package (`sudo apt install sysstat`).
- Shows read/write rates and utilization per device.

Kernel Messages — dmesg

Displays boot and kernel messages.

```
dmesg | less
```

Filter for hardware errors:

```
dmesg | grep -i error
```

Or view only recent messages:

```
dmesg --ctime | tail -n 20
```

Helpful for diagnosing device issues or driver problems.

Hardware Overview

CPU Information

```
lscpu
```

Example output:

```
Architecture:            x86_64
CPU(s):                  8
Model name:              Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Thread(s) per core:      2
Core(s) per socket:      6
```

Memory Layout

```
lsmem
```

Shows detected memory blocks and sizes.

PCI Devices — lspci

Lists hardware on the PCI bus (network cards, GPUs, etc.).

```
lspci | less
```

Example snippet:

```
00:02.0 VGA compatible controller: Intel Corporation UHD Graphics
01:00.0 3D controller: NVIDIA Corporation RTX 3060
```

Add -v or -vv for verbose details.

USB Devices — lsusb

Show all connected USB devices.

```
lsusb
```

Example output:

```
Bus 001 Device 004: ID 046d:c52b Logitech USB Receiver
Bus 002 Device 002: ID 0781:5567 SanDisk Cruzer Blade
```

Use `lsusb -t` for a tree view by USB port.

System BIOS and Hardware Metadata — dmidecode

dmidecode reads the **DMI/SMBIOS table** for low-level system details.

```
sudo dmidecode | less
```

Examples of sections:

- BIOS version and vendor
- Baseboard (motherboard) info
- Chassis and serial numbers
- Memory slot info

To target a specific type:

```
sudo dmidecode -t bios
sudo dmidecode -t memory
sudo dmidecode -t system
```

Read-only — safe to inspect, not modify.

Example — Quick System Summary

Combine tools for a complete picture:

```
echo "==== SYSTEM ===="
hostnamectl
echo "==== CPU ===="
lscpu | grep 'Model name'
echo "==== MEMORY ===="
free -h
echo "==== DISKS ===="
lsblk -f
echo "==== NETWORK ===="
ip a | grep inet
```

This gives an at-a-glance report of your machine.

Recap

- **System facts:** `uname`, `hostnamectl`, `lsb_release`, `/etc/os-release`
- **Health:** `uptime`, `free -h`, `vmstat`, `iostat`, `dmesg`
- **Hardware:** `lscpu`, `lsmem`, `lspci`, `lsusb`, `dmidecode`

These commands together let you audit, benchmark, and troubleshoot your Linux system effectively.

Security & Firewall (Plus)

Linux Commands Course · Section 19

IDSchool

Linux Security Layers

Linux security operates on multiple levels:

1. **Discretionary Access Control (DAC):** standard file permissions and ownership.
2. **Capabilities:** fine-grained privileges for executables.
3. **Mandatory Access Control (MAC):** enforced security frameworks (SELinux, AppArmor).
4. **Network Firewall:** traffic filtering with ufw, firewalld, or nftables.

File Capabilities — getcap, setcap

Traditionally, privileged actions required root (UID 0).
Capabilities allow splitting root powers into smaller permissions.

List file capabilities:

```
sudo getcap /bin/ping
```

Output example:

```
/bin/ping = cap_net_raw+ep
```

This means ping can use raw network sockets without being setuid root.

Assign a capability:

```
sudo setcap cap_net_bind_service=+ep /usr/bin/nginx
```

Remove it:

```
sudo setcap -r /usr/bin/nginx
```

View all files with capabilities:

```
sudo getcap -r / 2>/dev/null
```

Mandatory Access Control (MAC)

Beyond standard ownership and permissions, Linux can enforce additional security through **SELinux** or **AppArmor**.

SELinux (Security-Enhanced Linux)

Developed by the NSA, SELinux enforces strict policy rules for processes and files.

Check mode:

```
getenforce
```

Possible modes:

- Enforcing – policy actively blocks violations
- Permissive – logs violations but allows actions
- Disabled – inactive

Temporarily change mode (root only):

```
sudo setenforce 0   # switch to Permissive
sudo setenforce 1   # back to Enforcing
```

View logs:

```
sudo cat /var/log/audit/audit.log | grep denied
```

Permanent configuration is in /etc/selinux/config.

AppArmor (Ubuntu and Debian)

AppArmor provides per-application confinement via security profiles.

Check AppArmor status:

```
sudo aa-status
```

Output example:

```
apparmor module is loaded.
26 profiles are loaded.
22 profiles are in enforce mode.
```

List profiles and modes:

```
sudo aa-status | grep enforce
```

Enable or disable specific profiles:

```
sudo aa-enforce /etc/apparmor.d/usr.bin.firefox
sudo aa-disable /etc/apparmor.d/usr.bin.firefox
```

AppArmor is easier to manage than SELinux but provides similar isolation.

Host Firewalls — Overview

Linux firewalls filter traffic using the **netfilter** framework.
There are several user-friendly frontends built on top of it.

UFW (Uncomplicated Firewall)

Simplified interface (Ubuntu and derivatives).

Check status:

```
sudo ufw status
```

Enable the firewall:

```
sudo ufw enable
```

Allow or deny rules:

```
sudo ufw allow 22/tcp
sudo ufw deny 23/tcp
```

Delete a rule:

```
sudo ufw delete allow 22/tcp
```

View detailed numbered list:

```
sudo ufw status numbered
```

Disable firewall:

```
sudo ufw disable
```

Reset to default:

```
sudo ufw reset
```

firewalld and firewall-cmd

Used by RHEL, Fedora, and openSUSE.

Start and enable service:

```
sudo systemctl enable --now firewalld
```

Check active zones:

```
sudo firewall-cmd --get-active-zones
```

Allow a service in the default zone:

```
sudo firewall-cmd --add-service=http --permanent
sudo firewall-cmd --reload
```

Add a custom port:

```
sudo firewall-cmd --add-port=8080/tcp --permanent
sudo firewall-cmd --reload
```

List all active rules:

```
sudo firewall-cmd --list-all
```

nftables and iptables (Conceptual Overview)

nftables is the **modern packet filter** replacing iptables.

- iptables — legacy interface (still widely used)
- nftables — unified replacement for IPv4/IPv6

Check active rules:

```
sudo nft list ruleset
```

Example nftables rule snippet:

```
table inet filter {
  chain input {
    type filter hook input priority 0;
    policy drop;
    iif "lo" accept
    ct state established,related accept
    tcp dport {22,80,443} accept
  }
}
```

iptables equivalent (legacy):

```
sudo iptables -L -n -v
```

When to Use Which

Tool	Recommended for	Notes
ufw	Simple desktop/server setups	Easy syntax
firewalld	Enterprise systems (RHEL/Fedora)	Zone-based rules
nftables	Advanced configurations	Modern standard
iptables	Legacy compatibility	Being replaced

Recap

- **File capabilities:** getcap, setcap (fine-grained privileges)
- **MAC systems:** SELinux (getenforce, setenforce), AppArmor (aa-status)
- **Firewalls:** ufw, firewall-cmd, nftables, iptables
- Defense layers work together — never rely on just one.

Navigation & Filesystem Concepts (Core)

Linux Commands Course · Section 1

IDSchool

What is a File?

In Linux, **everything is treated as a file** — ordinary files, directories, devices, sockets, even processes.

A file is a named collection of data stored on disk.

Examples:

- Text files — contain readable text.
- Binary files — contain executable or machine data.
- Directories — special files that list other files.

Long Format (ls -l) Explained

The long format shows multiple properties of each file:

```
ls -l
```

Example output:

```
-rw-r--r-- 1 student users  4096 Oct 22 10:30 notes.txt
```

Parts of this line:

1. **Type & permissions** – file type and access rights
2. **Links** – number of hard links
3. **Owner** – user who owns the file
4. **Group** – group owning the file
5. **Size** – file size in bytes
6. **Date & time** – last modification
7. **Name** – file name

This view helps you identify files and their properties at a glance.

Home Directory Meaning

Every user has a personal **home directory** — their private workspace.

It’s where:

- Your personal files and folders are stored.
- Configuration files (dotfiles) live.
- You usually start when logging in.

Path example:

```
/home/student
```

Shortcut:

```
~
```

`~` always expands to your current user’s home directory.

Where am I?

Show your current working directory:

```
pwd
```

Example output:

```
/home/student
```

This tells you your exact location in the filesystem hierarchy.

Listing files — `ls`

`ls` lists directory contents.

```
ls
```

Show hidden files (those starting with `.`):

```
ls -a
```

Detailed view with permissions, sizes, owners, and dates:

```
ls -l
```

Combine both:

```
ls -la
```

Human-friendly details

Add `-h` for human-readable sizes (KB, MB, GB):

```
ls -lh
```

Sort by modification time (newest last):

```
ls -lt
```

Reverse order (newest first):

```
ls -ltr
```

List one entry per line:

```
ls -1
```

You can combine flags in one command — for example, `ls -lah`.

Colorized output & file types

Many distros colorize `ls` output automatically (directories in blue, executables in green).

Each leading character in `ls -l` shows type:

Symbol	Type
`-`	regular file
`d`	directory
`l`	symbolic link
`c`	character device
`b`	block device

Changing directories — cd

Move to another location:

```
cd /etc
```

Return to your home directory:

```
cd
```

Go up one level (parent directory):

```
cd ..
```

Return to the previous working directory:

```
cd -
```

Home directory shortcut

~ always represents your home directory.

```
cd ~
```

To access subfolders in home, append paths:

```
cd ~/Documents
```

~user accesses another user’s home (if permitted).

Tree view (optional tool)

If installed, tree shows a visual directory structure.

```
tree -L 2
```

-L 2 limits depth to 2 levels.

Install it if missing:

```
sudo apt install tree
```

or

```
sudo dnf install tree
```

Absolute vs Relative Paths

Absolute paths start from / (root).
Relative paths start from your current directory.

Examples:

Type	Example	Meaning
Absolute	/home/student/docs	Always points to the same location
Relative	../docs	Moves relative to where you are

Tip: Use pwd before running a command to confirm your location.

Globbing — Wildcards

The shell expands wildcard patterns automatically before running the command.

Pattern	Matches
*	any number of any characters
?	any single character
[abc]	any one of a, b, or c
[0-9]	any digit
[!x]	anything *except* x

Example:

```
ls *.txt
```

lists all files ending in .txt in the current directory.

Brace Expansion {}

Create multiple arguments or names in one command.

```
echo file_{a,b,c}.txt
```

→ expands to file_a.txt file_b.txt file_c.txt

Make multiple directories:

```
mkdir project/{src,bin,docs}
```

{} saves typing repetitive parts.

Environment Variables

Special variables store information about your shell environment.

Show your home directory path:

```
echo $HOME
```

Show your PATH (where executables are searched):

```
echo $PATH
```

PATH is a colon-separated list of directories.

PATH in action

When you type a command name, the shell searches each directory in $PATH from left to right.

You can inspect the order by printing it:

```
echo $PATH
```

To see where a command is found:

```
which ls
```

If multiple versions exist, the first one found in $PATH runs.

Recap

- `pwd` – print current directory
- `ls`, `ls -lah` – list files with detail
- `cd`, `cd ..`, `cd -` – move around
- `tree` – visual hierarchy
- Absolute vs relative paths – know your context
- Wildcards `* ? [ ] { }` – powerful pattern matching
- `$HOME`, `$PATH` – key environment variables

Quality-of-Life & Modern CLI Helpers (Optional)

Linux Commands Course · Section 20

IDSchool

Quick References — tldr

tldr provides concise, example-driven help pages for common commands.

Install it:

```
sudo apt install tldr
# or
pip install tldr
```

Usage:

```
tldr tar
tldr grep
```

Example output for tldr tar:

```
tar — Archiving utility
Examples:
  - Create a gzip-compressed archive:
      tar -czvf archive.tar.gz directory/
  - Extract an archive:
      tar -xzvf archive.tar.gz
```

Fast, clean, and easy to read — great for quick reminders.

Faster File Searching — fd

fd is a simple, fast, user-friendly alternative to find.

Install (Debian/Ubuntu):

```
sudo apt install fd-find
```

Basic usage:

```
fd main
fd -e txt -e md
```

- Ignores .gitignore-excluded files by default
- Colorful output
- Much faster than find for interactive use

You might need to symlink for convenience:

```
ln -s $(which fdfind) ~/.local/bin/fd
```

Faster Grep — ripgrep (rg)

ripgrep (rg) is a blazing-fast alternative to grep that respects .gitignore and highlights matches.

Install:

```
sudo apt install ripgrep
```

Search recursively for text:

```
rg "TODO"
```

Limit by file type:

```
rg "error" --type py
```

Count matches per file:

```
rg -c "pattern"
```

Faster than both grep and ack.

Better Cat – bat

bat enhances cat with syntax highlighting, line numbers, and paging.

Install:

```
sudo apt install bat
```

Usage:

```
bat script.sh
bat /etc/passwd
```

You can also use it like cat in pipelines:

```
bat --plain file.txt | grep keyword
```

Alias it for convenience:

```
alias cat='batcat'
```

Monitoring Tools — btop, nload, iftop

▒ btop

A colorful and interactive resource monitor.

```
sudo apt install btop
btop
```

Features:

- CPU, memory, disk, and network usage graphs
- Process viewer with search and sorting
- Mouse support and theme customization

▒ nload

Real-time network bandwidth monitor (per interface).

```
sudo apt install nload
nload
```

Displays incoming and outgoing traffic separately.

▒ iftop

Interactive bandwidth usage by connection.

```
sudo apt install iftop
sudo iftop
```

Shows which hosts are consuming bandwidth — great for network debugging.

Comparisons and Recommendations

Classic Tool	Modern Alternative	Benefit
man	tldr	Quick, example-focused help
find	fd	Faster, simpler syntax
grep	ripgrep (rg)	Fast recursive search with colors
cat	bat	Pretty printing and highlighting
top	btop	Graphical monitoring
iftop	(modern already)	Network inspection
nload	(modern already)	Lightweight bandwidth view

Bonus Tips

You can create an **alias group** for modern replacements:

```
alias cat='batcat'
alias grep='rg'
alias find='fd'
alias top='btop'
```

Store them in `~/.bashrc` to persist.

Recap

- **tldr** — short and friendly command reference
- **fd** — faster file search
- **ripgrep (rg)** — next-gen grep
- **bat** — colorful file viewer
- **btop**, **nload**, **iftop** — intuitive performance monitors

These tools make everyday terminal work faster, clearer, and more enjoyable.

Congratulations!

You’ve completed the **Linux Commands Course** – from fundamentals to modern workflow tools.

Next: combine your skills to create your own shell utilities and automation scripts!

Text Editors (Core) — Nano & Vi

Linux Commands Course · Section 21

Nano — The Beginner-Friendly Editor

Nano is simple and intuitive, ideal for quick file edits.

Open or create a file:

```
nano filename.txt
```

You’ll see the file contents and a list of commands at the bottom.

Commands use the **Ctrl (^)** and **Meta (M or Alt)** keys.

Common Nano Shortcuts

Command	Action
Ctrl + O	Write (save) file
Ctrl + X	Exit (prompts to save if needed)
Ctrl + G	Help
Ctrl + K	Cut selected text or line
Ctrl + U	Paste (after cut)
Ctrl + W	Search text
Alt + W	Repeat search
Ctrl + \	Replace text
Ctrl + A	Move to start of line
Ctrl + E	Move to end of line
Ctrl + Y	Page up
Ctrl + V	Page down
Alt + A	Start selecting text
Ctrl + C	Show cursor position

Nano Configuration Tips

Enable line numbers and syntax highlighting by default:

Edit or create your `~/.nanorc` file:

```
set linenumbers
set tabsize 4
set autoindent
set mouse
```

Exit and restart Nano to apply changes.

Vi / Vim — The Power User’s Editor

vi (or its enhanced version vim) is a **modal editor** — it operates in different modes.

Modes:

- **Normal mode** — navigation and commands
- **Insert mode** — text editing
- **Command-line mode** — executing editor commands

Open or create a file:

```
vi filename.txt
```

Basic Vi Workflow

When Vi starts, it opens in **Normal mode**.

1. Press `i` → enter **Insert mode** to type text
2. Write or edit your content
3. Press `Esc` → return to **Normal mode**
4. Save and quit (see below)

Essential Vi Commands

Mode	Command	Description
Normal	i	Insert mode before cursor
Normal	a	Insert mode after cursor
Normal	o	New line below and insert
Normal	O	New line above and insert
Normal	x	Delete character under cursor
Normal	dd	Delete entire line
Normal	yy	Copy (yank) current line
Normal	p	Paste below
Normal	P	Paste above
Normal	u	Undo
Normal	Ctrl + r	Redo
Normal	/text	Search for text
Normal	n	Next search result
Normal	N	Previous search result
Normal	:%s/old/new/g	Replace all occurrences

Saving and Quitting in Vi

Enter **Command mode** (press `:` while in Normal mode).

Command	Action
`:w`	Save (write) file
`:q`	Quit (fails if unsaved changes)
`:wq` or `:x`	Save and quit
`:q!`	Quit without saving
`:w newfile.txt`	Save as a new file

Navigation in Vi

Key	Action
h	Move left
l	Move right
j	Move down
k	Move up
0	Beginning of line
$	End of line
gg	Beginning of file
G	End of file
Ctrl + f	Page down
Ctrl + b	Page up
:set number	Show line numbers
:set nonumber	Hide line numbers

Visual Selection (Highlighting Text)

Enter **Visual mode** from Normal mode:

- v – character-wise selection
- V – line-wise selection
- Ctrl + v – block (column) selection

After selecting, you can:

- y – yank (copy)
- d – delete
- p – paste

Vi Configuration File

Customize Vim behavior via ~/.vimrc (create if missing).

Example settings:

```
set number
set autoindent
set tabstop=4
set shiftwidth=4
set expandtab
set hlsearch
set incsearch
syntax on
```

Reload Vim to apply these settings.

Quick Comparison — Nano vs Vi

Feature	Nano	Vi / Vim
Ease of use	Very easy	Steeper learning curve
Modes	None	Modal (Normal, Insert, Command)
Highlighting	Yes (with config)	Yes (default in Vim)
Mouse support	Yes	Optional
Best for	Quick edits	Power editing, scripting
Exit confusion	Rare	Frequent :)

Recap

- **Nano** – simple editor with intuitive shortcuts.
- **Vi/Vim** – modal editor with powerful commands.
- Both are preinstalled on nearly all Linux systems.
- Knowing both ensures you can edit files on any Linux machine.

Files & Directories (Core)

Linux Commands Course · Section 2

IDSchool

Everything is a File

In Linux, almost everything is treated as a **file** — whether it’s a document, folder, device, or socket.

- Regular files → data you create (`.txt`, `.py`, `.jpg`)
- Directories → special files that store file lists
- Devices → `/dev/sda`, `/dev/null`
- Processes → `/proc/<pid>`
- Links → alternate names or shortcuts to files

Creating Files — touch

touch creates an empty file if it doesn’t exist.

```
touch notes.txt
```

If the file exists, touch updates its *modification timestamp*.

You can create multiple files at once:

```
touch a.txt b.txt c.txt
```

Reading Files — cat, less, nl

cat: print the whole file to the screen.

```
cat notes.txt
```

less: scroll interactively (recommended for long files).

```
less /etc/passwd
```

Controls inside less:

- Space → next page
- b → previous page
- /pattern → search
- q → quit

nl: display with line numbers.

```
nl notes.txt
```

Previewing Files — head and tail

See the beginning of a file:

```
head notes.txt
```

Show only first 10 lines by default, or specify a count:

```
head -n 5 notes.txt
```

See the last lines of a file:

```
tail notes.txt
```

Monitor a file as it grows (useful for logs):

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

Stop following with **Ctrl+C**.

Renaming & Moving — mv

mv moves or renames files and directories.

Rename a file:

```
mv oldname.txt newname.txt
```

Move a file into another directory:

```
mv report.txt /tmp/
```

Move multiple files:

```
mv *.txt ~/Documents/
```

Tip: Always use tab completion to avoid typos!

Copying Files — cp

Copy a single file:

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

Copy multiple files into a directory:

```
cp file1.txt file2.txt ~/Documents/
```

Copy directories recursively:

```
cp -r project backup_project
```

Add -i to prompt before overwrite, and -v for verbose output:

```
cp -ivr project backup_project
```

Deleting Files & Folders — rm, rmdir

Delete a file:

```
rm file.txt
```

Delete multiple files:

```
rm *.log
```

Remove a directory *recursively* (careful!):

```
rm -r old_project
```

Ask before each deletion:

```
rm -ri old_project
```

Remove empty directories safely:

```
rmdir emptydir
```

Creating Directories — mkdir

Create one directory:

```
mkdir projects
```

Create nested directories in one go:

```
mkdir -p projects/python/scripts
```

-p ensures parent folders are created if missing.

Inspecting File Metadata — stat

stat displays detailed information about a file.

```
stat notes.txt
```

Example output:

```
File: notes.txt
Size: 4096       Blocks: 8    IO Block: 4096 regular file
Device: 802h/2050d   Inode: 1234567  Links: 1
Access: (0644/-rw-r--r--)  Uid: (1000/student)  Gid: (1000/student)
Access, Modify, Change times...
```

Shows size, type, permissions, timestamps, and inode (unique identifier).

Detecting File Type — file

Check what kind of data a file contains.

```
file /bin/bash
file photo.jpg
file script.sh
```

Output examples:

- ELF 64-bit executable (for programs)
- JPEG image data
- ASCII text

It’s a quick way to understand what a file *really is*, regardless of its extension.

Links — Hard vs Symbolic

Links are alternative names for files.

Hard link: another name pointing to the same data.

```
ln notes.txt hardlink_to_notes
```

Symbolic (soft) link: a shortcut that points by path.

```
ln -s /etc/hosts hosts_link
```

Check with:

```
ls -l
```

Symbolic links show an arrow (→) pointing to their target.

Differences Between Link Types

Feature	Hard Link	Symbolic Link
Points to	file’s inode (real data)	file path (name)
Works across filesystems	❌	✅
Affected if original deleted	stays (until inode reused)	breaks (dangling link)
Shown in `ls -l`	same inode number	with → target path

Use symbolic links for convenience and hard links for redundancy.

Safety Tips

- Use `-i` (interactive) with `cp`, `mv`, and `rm` while learning.
- Always double-check paths before using `rm -r`.
- Use `less` instead of `cat` for large files.
- For log monitoring, combine `tail -f` with `grep`.

Example:

```
tail -f /var/log/syslog | grep "error"
```

Recap

- **Create** files → `touch`
- **Read** → `cat`, `less`, `nl`, `head`, `tail -f`
- **Modify / Move** → `cp`, `mv`, `rm`
- **Directories** → `mkdir -p`, `rmdir`
- **Inspect** → `stat`, `file`
- **Links** → `ln`, `ln -s`

These are your daily drivers for file management in Linux.

Permissions & Ownership (Core)

Linux Commands Course · Section 3

IDSchool

Why Permissions Matter

Permissions protect your system and data from unauthorized changes.

Each file and directory has:

- **Owner** — the user who owns it.
- **Group** — users sharing the same project/team.
- **Others** — everyone else.

Each of these can have **read**, **write**, or **execute** rights.

Viewing Permissions — ls -l

List files with detailed information:

```
ls -l
```

Example output:

```
-rwxr-xr-- 1 student staff  1024 Oct 22 10:30 script.sh
```

Parts of the first field (-rwxr-xr--):

Symbol	Meaning
-	file type (-=file, d=directory, l=symlink)
r	read permission
w	write permission
x	execute permission
-	permission missing

Grouped as: **user / group / others** → rwx r-x r--.

Basic Permission Categories

Entity	Description	Example
User (u)	The file owner	rwx
Group (g)	Members of the file’s group	r-x
Others (o)	Everyone else	r--

Example string breakdown:

```
-rwxr-xr--
 ↑   ↑   ↑
 u   g   o
```

Each group has three bits — total 9 permission bits.

Changing Permissions — chmod (symbolic)

Change permissions using symbolic notation.

Format:

```
chmod [who][operator][permission] file
```

Examples:

```
chmod u+x script.sh     # give user execute permission
chmod g-w file.txt      # remove group write permission
chmod o+r notes.txt     # allow others to read
chmod a-rwx test.log    # remove all access for everyone
```

Who: u (user), g (group), o (others), a (all)
Operators: + (add), - (remove), = (set exactly)

Changing Permissions — chmod (octal)

Each permission is represented by a **number**:

Permission	Binary	Value
read (r)	100	4
write (w)	010	2
execute (x)	001	1

Sum them per group → **rwx = 7**, **r-x = 5**, **r-- = 4**.

Example:

```
chmod 755 script.sh
```

Breakdown:

User	Group	Others	Mode
rwx	r-x	r-x	755

Common Octal Modes

Mode	Meaning	Use case
644	rw-r--r--	normal text files
600	rw-------	private files
755	rwx-r-x-r-x	executable scripts, public dirs
700	rwx------	private scripts
777	rwx-rwx-rwx	full access (dangerous)

Avoid 777 unless in temporary environments.

Changing Ownership — chown

Set the file’s owner (and optionally group).

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

Change both owner and group:

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

Recursively change everything inside a directory:

```
sudo chown -R alice:developers /var/www/
```

Only root or file owners can change ownership.

Changing Group Ownership — chgrp

Change only the group part of ownership.

```
sudo chgrp staff report.txt
```

Useful when multiple users share access via a group.

Default Permissions — umask

umask defines what permissions **new files** start with.

Display your current mask:

```
umask
```

Example output: 0022 → means remove write for *group* and *others*.

Base defaults:

- Files start as 666 (rw-rw-rw-)
- Directories start as 777 (rwxrwxrwx)

So, 666 - 022 = 644 → normal default for new files.

Special Permission Bits

In addition to basic read/write/execute, three **special bits** exist:

Bit	Applies to	Symbol	Purpose
setuid	Executable files	s in user field	Run as file’s owner
setgid	Executables / directories	s in group field	Run as file’s group; new files inherit group
sticky bit	Directories	t in others field	Only owner can delete own files

---

setuid Example

If a binary has setuid bit, it runs as its owner (often root).

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

You’ll see:

```
-rwsr-xr-x 1 root root ...
```

The `s` in the user part means it runs with owner privileges.

setgid Example

For executables:

- setgid means they run with the group of the file.

For directories:

- Files created inside inherit the directory’s group.

```
chmod g+s shared_dir
```

This helps in group collaboration environments.

Sticky Bit Example

Used on shared directories like /tmp to prevent deletion by others.

```
ls -ld /tmp
```

Output:

```
drwxrwxrwt ...
```

The final t means sticky bit is set — only file owners can delete their own files.

Checking Special Bits (Numeric Form)

Special bits occupy a **fourth digit** before the usual 3-digit mode.

Bit	Octal	Combined example
setuid	4	4755
setgid	2	2755
sticky	1	1755

Example:

```
chmod 1777 /shared/tmp
```

makes it world-writable but protected by sticky bit.

Recap

- View: `ls -l`
- Change: `chmod` (symbolic or octal)
- Ownership: `chown`, `chgrp`
- Defaults: `umask`
- Special bits: `setuid`, `setgid`, `sticky`

Permissions define *who* can read, write, or execute — the backbone of Linux security.

Finding Things (Core)

Linux Commands Course · Section 4

IDSchool

Finding Commands — which, whereis, type

▒ which

Shows the full path of a command as found in $PATH.

```
which ls
```

Output example:

```
/bin/ls
```

If nothing prints, the command isn’t in your PATH.

▒ whereis

Locates executables, source code, and man pages for a command.

```
whereis ls
```

Example output:

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

Useful when you want both program and documentation locations.

▒ type

Displays how a command name will be interpreted by the shell.

```
type echo
```

Possible results:

- **builtin** — internal to the shell
- **alias** — shortcut defined by the user
- **file** — external program in the PATH

`type` is the most complete tool for identifying command nature.

Searching Files — find

find scans directories recursively and matches patterns or conditions.

Basic syntax:

```
find [path] [tests] [actions]
```

Example — find files by name:

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

The dot (.) means “start from current directory”.

Search by Type, Size, and Time

By file type:

```
find /etc -type d
```

→ shows only directories.

By size:

```
find /var/log -size +10M
```

→ finds files larger than 10 MB.

By modification time (in days):

```
find /home -mtime -2
```

→ modified in the last 2 days.

Combining Conditions

You can combine filters with logical operators.

Example – find .log files modified recently:

```
find /var/log -type f -name "*.log" -mtime -1
```

You can also negate tests:

```
find /etc -type f ! -name "*.conf"
```

→ every file that is *not* a .conf file.

Running Actions — -exec

Execute a command on each found file.

Example — list detailed info:

```
find . -type f -name "*.sh" -exec ls -lh {} \;
```

Each `{}` represents the current file; `\;` ends the `-exec` clause.

Or remove safely (after verifying!):

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

Avoid Unwanted Paths — -prune

Exclude directories from search with -prune.

Example — skip .git folders:

```
find . -path "./.git" -prune -o -type f -name "*.py" -print
```

How it works:

- -prune skips matched directories.
- The -o means “OR” — only the right side runs when left fails.

Using find with xargs

xargs efficiently passes found files to another command.

Example – count lines in all .c files:

```
find . -name "*.c" | xargs wc -l
```

Faster than repeated -exec calls.

For safety with spaces in filenames, use -print0 + xargs -0:

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

Locate — database-based search

locate searches a prebuilt database of filenames — much faster than find.

```
locate passwd
```

The database is usually updated daily.

If results seem outdated, refresh manually:

```
sudo updatedb
```

locate searches **by name only**, not by content or modification time.

Comparing find vs locate

Feature	find	locate
Searches live filesystem	✅	❌ (uses index)
Needs database update	❌	✅
Can filter by time/size/type	✅	❌
Speed	Slower	Instant
Accuracy	Always current	May be outdated

Use locate for quick lookups, and find for precise, real-time results.

Recap

- **Command locations:** `which`, `whereis`, `type`
- **File system search:** `find` (name, size, time, exec, prune)
- **Indexed search:** `locate`, `updatedb`
- Combine with `xargs` for high performance.

These are your search toolkit for any Linux environment.

Text Viewing & Pipelines (Core)

Linux Commands Course · Section 5

IDSchool

Viewing Text — less

less is the most convenient pager for reading long text output or files.

```
less /etc/passwd
```

Inside less:

- Space / Page Down → next page
- b / Page Up → previous page
- /pattern → search
- n / N → next/previous match
- q → quit

You can pipe output into it too:

```
ls -l /etc | less
```

Counting Text — wc

wc = **word count**, but it can count lines, words, and characters.

```
wc file.txt
```

Example output:

```
120  560  4200  file.txt
```

Columns = lines, words, bytes.

Count only lines:

```
wc -l file.txt
```

You can pipe text into wc:

```
cat /etc/passwd | wc -l
```

Redirection Basics

Every command has three data streams:

- **stdin (0)** – input
- **stdout (1)** – normal output
- **stderr (2)** – error output

You can redirect these streams to files.

Output Redirection

Write command output to a file:

```
echo Hello > message.txt
```

Append instead of overwrite:

```
echo World >> message.txt
```

Redirect both stdout and stderr to a single file:

```
command &> output.log
```

Send only errors:

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

Discard unwanted output:

```
command > /dev/null 2>&1
```

Input Redirection

Feed a file as input to a command:

```
sort < unsorted.txt
```

This reads from `unsorted.txt` instead of keyboard input.

Pipelines — `|`

The **pipe operator** (`|`) connects one command’s output to another’s input.

```
cat /etc/passwd | grep bash | wc -l
```

This example:

1. Reads `/etc/passwd`
2. Filters lines containing “bash”
3. Counts them

Pipelines chain tools to form complex processing flows.

Here-Documents (<<)

A here-document feeds a block of text directly into a command.

```
cat <<EOF > welcome.txt
Welcome to Linux!
This file was generated from a here-doc.
EOF
```

Everything until EOF is sent as input to cat and saved into the file.

You can use any marker instead of EOF.

Here-Strings (<<<)

Feed a **single line of text** as input:

```
cat <<< "Hello from here-string"
```

Equivalent to:

```
echo "Hello from here-string" | cat
```

Splitting and Merging Streams — tee

tee writes output to **both the terminal and a file** simultaneously.

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

Append instead of overwrite:

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

Useful for saving logs while still viewing output live.

Combine Multiple Sources — paste

paste merges lines from multiple files side by side.

Example:

```
paste names.txt ages.txt
```

Output:

```
Alice   25
Bob     30
Charlie 28
```

You can specify a custom delimiter:

```
paste -d ":" file1 file2
```

Compare Common Lines — comm

comm compares two sorted files line by line.

```
comm fileA fileB
```

Columns:

1. Lines unique to fileA
2. Lines unique to fileB
3. Lines common to both

Hide specific columns:

```
comm -12 fileA fileB    # show only common lines
```

Files must be **sorted** beforehand.

Join Files by Common Field — join

join merges two files based on a shared column (like a database join).

Example:

```
join users.txt departments.txt
```

Use -1 and -2 to choose which fields to join on:

```
join -1 1 -2 2 file1 file2
```

Sort both files first for reliable results.

Combining Tools in Pipelines

Real power comes from chaining commands.

Example — count unique shell types:

```
cat /etc/passwd | cut -d: -f7 | sort | uniq -c
```

Another example — save and view results:

```
ps aux | grep ssh | tee ssh_processes.txt | wc -l
```

Recap

- **less** – scroll through text interactively
- **wc** – count lines, words, or characters
- **>**, >>, 2>, &> – redirect output and errors
- **|** – pipe between commands
- **<<**, <<< – here-docs and here-strings
- **tee**, **paste**, **join**, **comm** – split, merge, and compare streams

Together, these form the foundation of Linux text processing.

Text Processing (Core → Plus)

Linux Commands Course · Section 6

IDSchool

Filtering Lines — grep

grep searches for patterns inside text files or input streams.

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

Shows all lines containing “root”.

Case-insensitive search:

```
grep -i "bash" /etc/passwd
```

Show line numbers:

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

Recursive search through directories:

```
grep -R "error" /var/log
```

Regular Expressions (regex)

grep -E enables extended regex for more expressive matching.

Examples:

```
grep -E "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+$" emails.txt
```

→ matches email-like lines.

Regex basics:

Symbol	Meaning
.	any single character
^	start of line
$	end of line
[]	character class
*, +, ?	repetition quantifiers
`	`

Use -v to invert (show non-matching lines).

Extracting Columns — cut

Split text into fields and extract specific columns.

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

→ prints username and shell columns.

Here, `-d:` sets delimiter to `:` and `-f` specifies which fields to output.

Extract fixed-width positions:

```
cut -c1-10 filename.txt
```

Transforming Characters — tr

tr replaces, deletes, or squeezes characters.

Uppercase to lowercase:

```
cat names.txt | tr '[:upper:]' '[:lower:]'
```

Remove digits:

```
cat file.txt | tr -d '0-9'
```

Replace spaces with tabs:

```
cat file.txt | tr ' ' '	'
```

Sorting and Uniqueness — sort, uniq

Sort alphabetically:

```
sort names.txt
```

Sort numerically and by human sizes:

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

Eliminate duplicates (must be sorted first):

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

Count repeated lines:

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

Shows how many times each entry occurs.

Editing Streams – sed

sed edits text as it flows through a pipeline.

Substitute “foo” with “bar”:

```
sed 's/foo/bar/' file.txt
```

Replace globally on each line:

```
sed 's/foo/bar/g' file.txt
```

In-place modification:

```
sed -i 's/error/ERROR/g' logfile.txt
```

Delete specific lines (e.g., 2–4):

```
sed '2,4d' file.txt
```

Print only certain lines:

```
sed -n '1,5p' file.txt
```

Reporting Language — awk

awk is a text-based data extraction and reporting DSL.

Print the first field of each line:

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

Use multiple fields and text:

```
awk -F: '{print "User:", $1, "Shell:", $7}' /etc/passwd
```

Conditionals:

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

Perform arithmetic and aggregation:

```
awk '{sum += $2} END {print "Total:", sum}' data.txt
```

Power Combinations – xargs

xargs converts input lines into command arguments.

Example – delete found files:

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

Count lines of all .txt files:

```
ls *.txt | xargs wc -l
```

Safer with spaces:

```
find . -name "*.txt" -print0 | xargs -0 wc -l
```

Process Substitution <()

Run two commands in parallel and compare results without temporary files.

```
diff <(sort a.txt) <(sort b.txt)
```

Also useful with join, comm, or paste to feed preprocessed data.

Encoding Tools — iconv, dos2unix

Convert between character encodings with iconv:

```
iconv -f ISO-8859-1 -t UTF-8 old.txt -o new.txt
```

Fix Windows line endings (CRLF) in text files:

```
dos2unix script.sh
```

Makes scripts compatible on Linux systems.

JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of name–value pairs and arrays. It is a commonly used data format with diverse uses in electronic data interchange, including that of web applications with servers.

```json
{
    "name": "Elnur",
    "job": [
        "Teacher",
        "Cyber Security Engineer"
    ],
    "age": 22
}
```

JSON Processing — jq

jq is a lightweight command-line JSON processor.

Format JSON neatly:

```
jq . data.json
```

Extract specific fields:

```
jq '.users[].name' data.json
```

Filter with conditions:

```
jq '.users[] | select(.age > 25)' data.json
```

Combine with other commands:

```
curl -s https://api.github.com/users/torvalds | jq '.name, .public_repos'
```

Recap

- **grep** — match/filter text using regex
- **cut**, **tr**, **sort**, **uniq** — extract and transform columns
- **sed** — substitute or delete text patterns
- **awk** — structured reporting and logic
- **xargs**, **<()** — advanced composition
- **iconv**, **dos2unix**, **jq** — encoding and JSON utilities

Together, these make Linux text processing infinitely flexible.

Archiving & Compression (Core)

Linux Commands Course · Section 7

IDSchool

What Is Archiving?

Archiving combines multiple files or folders into one container file.
Compression makes that container smaller.

Common reasons to archive:

- Backup and transfer data
- Package projects or logs
- Preserve directory structures

Linux standard tools: `tar`, `gzip`, `bzip2`, `xz`, `zstd`, `zip`.

Creating Tar Archives

tar (tape archive) is the most common archiving utility.

Create an archive from a folder:

```
tar -cvf backup.tar project/
```

Options:

- c – create
- v – verbose (show files)
- f – file name

Extract archive:

```
tar -xvf backup.tar
```

List contents without extracting:

```
tar -tvf backup.tar
```

Compressed Tarballs

tar can compress directly using gzip, bzip2, xz, or zstd.

▒▒ Create compressed archive (gzip)

```
tar -czvf project.tar.gz project/
```

▒▒ Extract it

```
tar -xzvf project.tar.gz
```

You can also use different extensions to choose the compression algorithm automatically.

---

bzip2 and xz Examples

▒▒▒▒ Create bzip2 tarball

```
tar -cjvf data.tar.bz2 data/
```

Extract it:

```
tar -xjvf data.tar.bz2
```

▒▒▒▒ Create xz tarball

```
tar -cJvf data.tar.xz data/
```

Extract it:

```
tar -xJvf data.tar.xz
```

Flag	Algorithm	Extension
z	gzip	.gz
j	bzip2	.bz2
J	xz	.xz

Modern Compression — zstd

zstd (Zstandard) is a fast modern compressor with excellent ratios.

```
tar -I zstd -cvf project.tar.zst project/
```

Extract:

```
tar -I zstd -xvf project.tar.zst
```

You can also use the standalone tools:

```
zstd file.txt        # creates file.txt.zst
unzstd file.txt.zst  # decompresses it
```

---

gzip and gunzip (Classic Pair)

▒ Compress

```
gzip notes.txt
```

This replaces notes.txt with notes.txt.gz.

▒ Decompress

```
gunzip notes.txt.gz
```

Or with gzip -d notes.txt.gz.

You can test compression ratio with:

```
gzip -l notes.txt.gz
```

---

bzip2 and bunzip2

Better compression, slower speed.

▒▒▒▒ Compress

```
bzip2 report.txt
```

Creates report.txt.bz2 and removes original.

▒▒▒▒ Decompress

```
bunzip2 report.txt.bz2
```

▒▒▒▒ Keep original file

```
bzip2 -k report.txt
```

---

xz and unxz

High compression ratio; often used for distributing software packages.

▒ Compress

```
xz archive.tar
```

Produces `archive.tar.xz`.

▒ Decompress

```
unxz archive.tar.xz
```

To view progress while compressing:

```
xz -v archive.tar
```

Cross-Platform Archives — zip and unzip

ZIP is widely supported across operating systems.

▒ Create zip archive

```
zip -r project.zip project/
```

▒ Extract zip file

```
unzip project.zip
```

▒ Extract to specific folder

```
unzip project.zip -d /tmp/project
```

List contents:

```
unzip -l project.zip
```

Choosing the Right Tool

Tool	Format	Speed	Compression	Portability	Use case
gzip	.gz	Fast	Medium	High	Everyday backups
bzip2	.bz2	Medium	Higher	Medium	Logs, archives
xz	.xz	Slow	Very High	Medium	Software packaging
zstd	.zst	Very Fast	High	Medium	Modern systems
zip	.zip	Fast	Medium	Very High	Cross-platform

Inspecting Archive Contents

List files in an archive without extracting:

```
tar -tvf archive.tar
unzip -l project.zip
```

Test integrity (for `.zip`):

```
unzip -t project.zip
```

Combine with Pipelines

Create and compress on the fly:

```
tar -czf - project/ | ssh backup@server "cat > /backups/project.tgz"
```

Or decompress remotely:

```
ssh backup@server "cat /backups/project.tgz" | tar -xz
```

This allows archiving without intermediate files.

Recap

- **tar** – archive multiple files (-cvf, -xvf)
- **gzip / bzip2 / xz / zstd** – compression algorithms
- **zip / unzip** – cross-platform archives
- Choose based on speed, ratio, and compatibility needs.

Essential Linux Directories & Files (Core)

Linux Commands Course · Section 8

IDSchool

Linux Filesystem Philosophy

Everything in Linux is organized under a **single root directory** `/`.

- No drive letters (like C: or D:)
- All devices, users, and applications live under `/`
- Each directory has a specific purpose

Think of it as a **tree**, with `/` as the root and everything branching from it.

Core Directory Layout Overview

Directory	Purpose
/bin	Essential user binaries (commands needed for all users)
/sbin	System binaries (commands for system administration)
/usr/bin	Most user commands and applications
/usr/sbin	System admin tools not required for booting
/etc	System configuration files
/var	Variable data (logs, mail, cache)
/home	User home directories
/tmp	Temporary files (cleared on reboot)
/dev	Device files (represent hardware or virtual devices)
/proc	Virtual filesystem exposing process and kernel info
/sys	System and hardware information (kernel interface)
/opt	Optional or third-party software packages

/bin and /sbin

Contain the most **fundamental executables**.

Examples:

```
ls /bin
ls /sbin
```

Typical contents:

- `/bin/ls`, `/bin/cp`, `/bin/mv`, `/bin/cat`
- `/sbin/reboot`, `/sbin/ifconfig`, `/sbin/fsck`

Used during system boot and single-user recovery mode.

/usr/bin and /usr/sbin

/usr = “Unix System Resources.”

Holds the **main body of user utilities** and optional system tools.

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

Applications like vim, python, gcc, systemctl, etc., live here.

This is where most installed software resides.

/etc — Configuration Files

Contains **system-wide configuration** for all programs and services.

Examples:

```
ls /etc
```

Subdirectories like `/etc/network`, `/etc/ssh`, `/etc/systemd` hold their respective configs.

These files are usually **plain text**, editable with a text editor.

/var — Variable Data

“Variable” because contents change frequently.

Common uses:

- /var/log/ → log files (syslog, auth.log)
- /var/spool/ → print/mail queues
- /var/cache/ → cached data
- /var/lib/ → application state (databases, package info)

Example:

```
ls /var/log
```

Logs are vital for troubleshooting.

/home — User Directories

Each user has a personal workspace under /home.

Example structure:

```
/home/alice
/home/bob
```

Contains personal files, downloads, and shell settings.

```
ls ~
```

The ~ symbol always refers to your current user's home directory.

/tmp — Temporary Storage

Used for short-lived files and data exchange between programs.

```
cd /tmp
ls
```

Cleared on reboot or after a set time.

Accessible to everyone but protected by the **sticky bit** so users can’t delete others’ files.

/dev – Devices as Files

Represents hardware and virtual devices as files.

```
ls /dev | head
```

Examples:

- /dev/sda – first hard drive
- /dev/null – data sink (discards anything written)
- /dev/tty – current terminal
- /dev/random – random data source

Device files enable programs to interact with hardware using normal file operations.

/proc — Process and Kernel Info

A **virtual filesystem** reflecting live system state.

```
ls /proc | head
```

Contains pseudo-files for each running process (`/proc/<PID>/`).

Key files:

- `/proc/cpuinfo` — CPU details
- `/proc/meminfo` — memory usage
- `/proc/uptime` — uptime information

Read-only for observation; data is generated dynamically.

/sys — Kernel and Device Management

/sys is similar to /proc but more structured.

Contains live info about devices, drivers, and kernel modules.

Example:

```
ls /sys/class
```

Used by udev and other system components to manage devices dynamically.

/opt — Optional Software

Holds **add-on applications** installed outside the package manager.

Example paths:

- `/opt/google/chrome/`
- `/opt/lampp/`

You can place self-contained programs or third-party tools here.

Must-Know System Files

File	Description
/etc/passwd	User account information (username, UID, home, shell)
/etc/shadow	Encrypted passwords and aging info (root-only)
/etc/group	Group membership definitions
/etc/fstab	Filesystems to mount at boot
/etc/hosts	Local hostname-to-IP mapping
/etc/resolv.conf	DNS resolver configuration
/etc/sudoers	sudo access control rules
~/.bashrc	User-specific shell customization
~/.profile	Environment setup on login

/etc/passwd Example

```
cat /etc/passwd | head -3
```

Example line:

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

Fields (colon-separated):

1. Username
2. Placeholder (historically password, now stored in `/etc/shadow`)
3. UID (User ID)
4. GID (Group ID)
5. Comment / full name
6. Home directory
7. Login shell

/etc/shadow Example

Only readable by root.

```
sudo head /etc/shadow
```

Stores encrypted passwords and password aging information.

Never edit this file manually – use `passwd` command instead.

/etc/group Example

Defines group memberships.

```
cat /etc/group | head
```

Each line: group name, password placeholder, GID, and members.

/etc/fstab — Filesystem Table

Defines which partitions and devices to mount automatically at boot.

Example:

```
UUID=xxxx-xxxx  /  ext4  defaults  0 1
UUID=yyyy-yyyy  /home  ext4  defaults  0 2
```

View safely:

```
cat /etc/fstab
```

/etc/hosts and /etc/resolv.conf

/etc/hosts — manual hostname resolution.

Example:

```
127.0.0.1   localhost
192.168.1.10 server.local
```

/etc/resolv.conf — nameserver (DNS) configuration.

Example:

```
nameserver 1.1.1.1
nameserver 8.8.8.8
```

/etc/sudoers

Controls who can run commands as other users (usually root).

Always edit with **visudo** to prevent syntax errors:

```
sudo visudo
```

Example rule:

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

Meaning: Alice can run any command as any user.

User Configuration Files

~/.bashrc – executed for interactive non-login shells.
Custom aliases, colors, and shell variables live here.

~/.profile – executed for login shells; sets environment variables like PATH and locale.

```
cat ~/.bashrc | head
```

Keep personal shell tweaks in .bashrc, system-wide ones in /etc/bash.bashrc.

Recap

- Linux filesystem is a single tree rooted at `/`.
- Know what each main directory stores.
- Learn critical system files under `/etc` and your home.
- Configuration lives in plain text.
- Reading these files (not editing them blindly) is key to system literacy.

Users, Groups & Sudo (Core)

Linux Commands Course · Section 9

IDSchool

Users and Groups in Linux

Every user on Linux has:

- A **username** (like `student` or `root`)
- A **UID** (user ID number)
- A **primary group**
- Optional secondary groups
- A **home directory** and **default shell**

Groups organize users for shared permissions and access control.

Inspecting User Information — id

Show your user and group identity:

```
id
```

Example output:

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

- uid → your user ID
- gid → your main group
- groups → all groups you belong to

Who Am I? — whoami

Prints your current effective username:

```
whoami
```

Useful in scripts or when using sudo to confirm who you are.

Listing Group Memberships — groups

Show which groups you belong to:

```
groups
```

Example:

```
student : student sudo docker
```

Active Users — who and w

who shows users currently logged in:

```
who
```

w gives more detail — what each user is doing:

```
w
```

Example:

```
student  pts/0  2025-10-22 10:31  bash
```

Login History – last

Displays recent logins and reboots.

```
last
```

Output example:

```
student pts/0  192.168.1.15  Wed Oct 22 10:00   still logged in
reboot  system boot  Wed Oct 22 09:55
```

This information is stored in /var/log/wtmp.

Understanding sudo

sudo lets authorized users run commands as another user — typically **root**.

Example:

```
sudo apt update
```

You’ll be prompted for your **own password**, not root’s.

Why use sudo instead of logging in as root?

- Safer (tracks every action)
- Temporarily elevates privileges
- Logs activity to /var/log/auth.log

How sudo Works

sudo checks its configuration file /etc/sudoers to see who can run what.

You can view effective privileges with:

```
sudo -l
```

If allowed, your command runs as if root executed it.

Example:

```
sudo whoami
# Output: root
```

Editing sudo Rules — visudo

You **must** use visudo to safely edit sudo privileges.

```
sudo visudo
```

Why?

- visudo checks syntax before saving, preventing broken access.
- Editing /etc/sudoers manually can lock out admin access!

Example rule in the file:

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

Meaning:

- **alice** → username
- **ALL** → any host
- **(ALL:ALL)** → can act as any user and group
- **ALL** → may run any command

You can restrict to specific commands:

```
bob ALL=(ALL) /usr/bin/systemctl restart nginx
```

Now Bob can only restart nginx, not anything else.

Granting Group Access via `sudo`

Instead of editing user-by-user, use groups.

Example line in `/etc/sudoers`:

```
%sudo   ALL=(ALL:ALL) ALL
```

Meaning: anyone in the **sudo** group has full admin rights.

Add user to that group:

```
sudo usermod -aG sudo alice
```

On RHEL/Fedora, the equivalent group is **wheel**.

Security Tips for Sudo

- Never edit `/etc/sudoers` directly — always use `visudo`.
- Limit commands users can run if full access isn’t needed.
- Use `sudo -l` to verify your privileges.
- Avoid `sudo su` (defeats auditing and accountability).

Creating a New User — useradd

Add a new user to the system (requires root).

```
sudo useradd -m alice
```

Options:

- -m → create home directory
- -s /bin/bash → set shell
- -G → add to extra groups

Example:

```
sudo useradd -m -s /bin/bash -G sudo alice
```

Set password:

```
sudo passwd alice
```

Modifying a User — usermod

Change user settings such as shell, group, or name.

Add a user to a group:

```
sudo usermod -aG docker alice
```

Change shell:

```
sudo usermod -s /bin/zsh alice
```

Rename user:

```
sudo usermod -l newname oldname
```

Deleting a User — userdel

Remove a user account.

```
sudo userdel alice
```

Delete the user’s home directory as well:

```
sudo userdel -r alice
```

Always ensure data is backed up before deletion.

Changing Passwords — passwd

Change your password:

```
passwd
```

Change another user’s password (admin only):

```
sudo passwd bob
```

Force password change on next login:

```
sudo passwd -e alice
```

Password Aging and Expiry — chage

View password aging info:

```
sudo chage -l alice
```

Set password expiration (e.g., 90 days):

```
sudo chage -M 90 alice
```

Set account expiry date:

```
sudo chage -E 2025-12-31 alice
```

This helps enforce password rotation and account control.

Managing Groups — groupadd

Create a new group:

```
sudo groupadd developers
```

Add an existing user to it:

```
sudo usermod -aG developers alice
```

Group Passwords and Administration — gpasswd

gpasswd manages group membership and optional passwords.

Add or remove users from a group:

```
sudo gpasswd -a bob developers
sudo gpasswd -d bob developers
```

Set a group administrator (can add/remove users):

```
sudo gpasswd -A alice developers
```

Set a group password (rarely used today):

```
sudo gpasswd developers
```

Recap

- Inspect users: `id`, `whoami`, `groups`, `who`, `w`, `last`
- Manage users: `useradd`, `usermod`, `userdel`, `passwd`, `chage`
- Manage groups: `groupadd`, `gpasswd`
- Privilege control: `sudo`, `visudo`

sudo is the bridge between normal users and root privileges — use it carefully.
