Linux

Efficient process management is one of the core skills when working with Linux. The operating system provides several commands—`nohup`, `nice`, `bg`, `fg`, and `jobs`—to control how processes run, whether in the foreground, background, or with specific priorities. Here's a breakdown of each and how to use them.

🔹 1. `jobs` Command

Theory

- The `jobs` command shows the status of jobs started in the current shell session.
- Jobs can be **Running**, **Stopped**, or **Terminated**.
- It only tracks jobs launched **from that shell** (not system-wide like `ps`).
- Lists all jobs started in the current shell session.
- Jobs are assigned IDs like `[1]`, `[2]`.

Syntax

```
jobs [options]
```

Example

```
ping google.com &
nano file.txt
jobs
```

Explanation

1. `ping google.com &` → Runs `ping` in background (`&` means background).
2. `nano file.txt` → Opens editor in foreground (blocks the shell).
3. `jobs` → Lists all jobs in that shell.

Output:

```
[1]+  Running   ping google.com &
[2]-  Stopped   nano file.txt
```

- `[1]` → Job number
- `+` → Current job (default target for `fg`/`bg`)
- `Running` / `Stopped` → Status
- `ping google.com` → Command

Use Case

- To monitor and control multiple processes launched from the same shell.

🔹 2. `bg` Command

Theory

- Used to **resume a stopped job in the background**.
- Jobs paused with `CTRL+Z` can be restarted with `bg`.
- Resumes a stopped job in the **background**.
- Usage:
  bg %1
- This resumes job ID 1 in the background.

Syntax

```
bg %job_id
```

Example

```
sleep 100
CTRL+Z
bg %1
```

Explanation

1. `sleep 100` → Command runs for 100 seconds.
2. `CTRL+Z` → Suspends it (status becomes *Stopped*).
3. `bg %1` → `%1` means job number 1. The job resumes in background.

Use Case

- If you accidentally stopped a long-running task but don't want to keep terminal blocked.

🔹 3. `fg` Command

Theory

- Used to bring a job running in background into **foreground**.
- Useful when you want to interact with it (like editors).

Syntax

```
fg %job_id
```

Example

```
ping google.com &
jobs
fg %1
```

Explanation

1. `ping google.com &` → Starts in background.
2. `jobs` → Confirms it's running (`%1`).
3. `fg %1` → Brings it to foreground, so output floods your screen.

Use Case

- Switching between background/foreground for multitasking.

🔹 4. `nice` Command

Theory

- Controls **process scheduling priority**.
- Lower nice value = higher priority.
- Sets the scheduling priority of a process.
- Nice values range from **-20** (highest priority) to **19** (lowest priority). Default = 0.
- Run a command with a specific nice value:

- Change the nice value of a running process:
  renice -n -5 -p <pid>

Syntax

```
nice -n <value> command
```

Example

```
nice -n 10 gzip largefile.iso
```

Explanation

1. `nice` → Run command with a specified nice value.
2. `-n 10` → Nice value of 10 (lower priority).
3. `gzip largefile.iso` → Command being run.

So Linux gives CPU time to more important tasks before compressing the file.

Use Case

- Run heavy background jobs (compression, backups) with lower priority so they don't slow down other processes.

🔹 5. `renice` Command

Theory

- Changes priority of **already running process**.

Syntax

```
renice -n <new_value> -p <pid>
```

Example

```
ps -ef | grep myscript.sh
renice -n -5 -p 12345
```

Explanation

1. `ps -ef | grep myscript.sh` → Find process ID (say 12345).
2. `renice -n -5 -p 12345` → Increase priority (`-5` is higher priority).

Use Case

- Increase priority of critical processes or reduce priority of non-urgent tasks.

🔹 6. nohup Command

Theory

- Stands for **No Hang Up**.
- Prevents a process from being killed when the terminal is closed.
- Default output is written to `nohup.out` unless redirected.
- Runs a process immune to terminal hangups, meaning it will continue running even if you log out or close the terminal.
- Basic usage:
  nohup ./script.sh &
- Redirect output to a file:
  nohup ./script.sh > output.log 2>&1 &
- Useful for long-running processes like backups, servers, or scripts when you don't want them to terminate after logging out.
- Protects the process from receiving the **SIGHUP** signal.

Syntax

```
nohup command [arguments] > output.log 2>&1 &
```

Example

```
nohup python3 myscript.py > myscript.log 2>&1 &
```

Explanation

1. `nohup` → Ignore hangup signal (keep process alive).
2. `python3 myscript.py` → Script being executed.
3. `>` → Redirect standard output.

4. `myscript.log` → File to store logs.
5. `2>&1` → Redirect error output (stderr) to the same file.
6. `&` → Run in background.

So even if you **logout**, the script keeps running.

Use Case

- Running servers, cron jobs, or long scripts remotely over SSH.

🔹 7. Foreground vs Background

Foreground process → Runs directly in terminal, blocks shell until it finishes.
`tar -czf backup.tar.gz /home/user`

-
- **Background process** → Add `&` to run without blocking shell.
  `tar -czf backup.tar.gz /home/user &`

Viewing and Resuming Jobs
See active jobs:
jobs

Resume in background:
bg %job_id

Resume in foreground:
fg %job_id

jobs → List active jobs
bg → Resume stopped jobs in background
fg → Bring jobs to foreground
nice/renice → Manage process priority
nohup → Run processes immune to logout/terminal close

What is a Cron Job?

- **Cron** is a time-based job scheduler in Unix/Linux.

- A **cron job** is a scheduled task that runs automatically at a specified time/date/interval.
- Great for repetitive tasks: backups, cleanup, sending reports, restarting services, etc.

Cron Syntax

A cron job is defined in the **crontab** (cron table). Each line follows this format:

```
* * * * * command-to-be-executed
- - - - -
| | | | |
| | | | +---- Day of week (0 - 7) (Sunday=0 or 7)
| | | +------ Month (1 - 12)
| | +-------- Day of month (1 - 31)
| +---------- Hour (0 - 23)
+------------ Minute (0 - 59)
```

So there are **5 time fields** followed by the command.

🔹 How Cron Works

1. Cron Daemon

- Cron jobs are managed by a background service called the **cron daemon** (`crond`).
- The daemon **runs continuously** and wakes up every minute to check scheduled jobs.
- On most Linux systems, you can check if cron is running:
- A **Cron Job** is a **scheduled task** in Linux that runs automatically at a specific time/date/interval.
- Managed by the **cron daemon (`crond`)**.
- Common use cases: backups, log rotation, sending reports, running scripts at intervals.

systemctl status cron # Debian/Ubuntu

systemctl status crond # RedHat/CentOS/Fedora

“A cron job is a scheduled task that runs automatically at defined times using the cron daemon.”
“The cron syntax has 5 fields: minute, hour, day of month, month, day of week.”
“I use `crontab -e` to add jobs, and common use cases are backups, monitoring scripts, log rotation, and scheduled reports.”

Add your job, e.g.:

0 3 * * * /home/user/backup.sh
(Runs every day at 3 AM)

Command	Description
`crontab -e`	Edit crontab file (create/modify cron jobs).
`crontab -l`	List user's cron jobs.
`crontab -r`	Remove all cron jobs for the user.
`crontab -u <user> -l`	List jobs for another user (root only).

Cron Expression Examples

Expression	Meaning
`* * * * *`	Run every minute.
`0 * * * *`	Run at start of every hour.
`30 5 * * *`	Run daily at 5:30 AM.
`0 0 1 * *`	Run on 1st day of every month at midnight.
`0 9 * * 1-5`	Run at 9 AM on weekdays (Mon–Fri).
`*/10 * * * *`	Run every 10 minutes.
`0 2 * * 0`	Run every Sunday at 2 AM.

Instead of writing long expressions, cron supports special strings:

Special String	Meaning

`@reboot`	Run once after reboot.
`@yearly` or `@annually`	Run once a year (Jan 1 at 00:00).
`@monthly`	Run once a month (1st day, 00:00).
`@weekly`	Run once a week (Sunday, 00:00).
`@daily` or `@midnight`	Run once a day (00:00).
`@hourly`	Run once an hour.

- Cron is a **time-based scheduler** running as a daemon.
- It checks **user/system crontabs** every minute.
- Executes commands whose schedule **matches the current time**.
- Runs commands in a **minimal environment**, handles output via email or log files

Crontab Commands

- Open crontab for current user: crontab -e
- List cron jobs: crontab -l
- Remove all cron jobs: crontab -r

1. Run a script every day at 5:30 AM

Eg. 30 5 * * * /home/user/backup.sh

- **`30` = minute (30th)**
- **`5` = hour (5 AM)**
- **`* * *` = every day, month, week**

Use case: Daily database backup

2. Run a script every 10 minutes

Eg. */10 * * * * /home/user/cleanup.sh

- **`*/10` = every 10 minutes**
- **Other fields `*` = every hour/day/month/week**

✅ **Use case: Cleanup temp files regularly.**

3. Run at midnight on the first day of every month

Eg. 0 0 1 * * /home/user/report.sh
✅ Use case: Generate monthly repo

4. Run script only on Sundays at 2 AM

Eg. 0 2 * * 0 /home/user/sunday_task.sh

`0` = Sunday (can also use `7`)

✅ Use case: Weekly log rotation.

5. Run every weekday (Mon–Fri) at 9 AM

Eg. 0 9 * * 1-5 /home/user/start.sh

`1-5` = Monday to Friday

✅ Use case: Start workday services.

6. Run script at reboot

Eg. @reboot /home/user/startup.sh

✅ Use case: Start a service automatically when system reboots.

`at` command in Linux, which is used to schedule tasks to run once at a specific time.

Linux `at` Command – Schedule Task to Run Once

1. What is the `at` command?

- **The `at` command schedules a command or script to run once at a specific future time.**
- **Unlike `cron`, it does not repeat the task.**
- **Requires the `atd` daemon to be running (similar to cron daemon).**

Check if `atd` is running:

systemctl status atd # RedHat/CentOS/Fedora

systemctl status at # Debian/Ubuntu

2. Install `at` (if not installed)

sudo apt install at # Debian/Ubuntu

sudo yum install at # RedHat/CentOS

3. Syntax

at [TIME] [OPTIONS]

`[TIME]` **can be in formats like:**

- `HH:MM` → **24-hour format today**
- `HH:MM YYYY-MM-DD` → **exact date and time**
- `now + 5 minutes` → **relative time**
- `tomorrow`, `noon`, `midnight`, `teatime`

4. Common Options

Option	Description
`-l` or `q`	List pending `at` jobs.
`-d` or `-r`	Remove/cancel a scheduled `at` job.
`-f <file>`	Schedule commands from a file.

5. Practical Examples

Example 1: Run a script at a specific time

at 23:30

at> /home/user/backup.sh

at> <EOT> # Press Ctrl+D to end input

explanation:

`at 23:30` → **Schedule job for 11:30 PM today**

`/home/user/backup.sh` → **Command to run**

`<EOT>` → **End of input (Ctrl+D)**

Example 2: Run a command in 5 minutes

at now + 5 minutes

at> echo "Backup started" >> /home/user/backup.log

at> <EOT>

Runs the command exactly 5 minutes from now

Example 3: Run commands from a file

Create a file `tasks.txt`:

```
#!/bin/bash

echo "Task started" >> /home/user/task.log

/home/user/script.sh
```

Schedule it:

```
at -f tasks.txt 22:00
```

- **Runs the commands in `tasks.txt` at 10 PM.**

6. Check Scheduled Jobs

```
atq
```

Lists all pending jobs for the current user.

7. Remove a Scheduled Job

```
atrm <job_id>
```

`job_id` comes from `atq`.

9. Difference Between `cron` and `at`

Feature	`cron`	`at`
Runs	Repeatedly	Once
Scheduling	Time intervals, recurring	Specific time/date
Daemon	crond	atd
Use case	Daily backups, recurring tasks	One-time scripts or commands

> **"I use `at` when I need a command to run only once at a specified future time, and `cron` for repeating tasks. `at`schedules jobs via the atd daemon, and pending jobs can be listed with `atq` and removed with `atrm`."**

Linux Anacron – Schedule Tasks on Your Terms

1. What is Anacron?

- Anacron is used to schedule periodic tasks, just like cron.
- Key difference: Anacron does not assume the system is running continuously.
- If a scheduled job was missed (e.g., system was powered off), Anacron runs the job as soon as the system is back online.
- Perfect for laptops, desktops, or servers that are not 24/7.
- Anacron is a Linux utility that allows you to schedule commands periodically, similar to cron, but with one big difference:
  It does not require the system to be running continuously.
- If your system is shut down when a scheduled job should have run, Anacron ensures that the job runs as soon as the system is back online.
- This makes it crucial for laptops, desktops, and systems with downtime, unlike cron, which is best suited for servers that are always on.

💡 Example:
If you set a daily backup with cron at 2 AM but your laptop is off, it will never run. With Anacron, the backup will execute the next time you boot up.

💡 In simple words:

- Cron → assumes system is always running (missed jobs are lost).
- Anacron → ensures jobs run at least once, even if delayed.

---

2. Where are Anacron Jobs Defined?

- Anacron jobs are typically defined in:
  - `/etc/anacrontab` (main configuration file)
  - `/etc/cron.daily/`, `/etc/cron.weekly/`, `/etc/cron.monthly/` (called by Anacron automatically)

---

3. Syntax of `/etc/anacrontab`

Each line follows this format:

```
period   delay   job-identifier   command
```

- period → how often the job runs (in days).
  - `1` = daily
  - `7` = weekly
  - `30` = monthly
- delay → number of minutes to wait after Anacron starts before running the job.
- job-identifier → unique name for the job (for logging).

- **command → actual command/script to execute.**

---

4. Example `/etc/anacrontab`

```
# period delay identifier command

1  5  cron.daily   run-parts /etc/cron.daily

7 10  cron.weekly  run-parts /etc/cron.weekly

30 15 cron.monthly run-parts /etc/cron.monthly
```

Explanation:

- `1 5 cron.daily` → **Run all scripts in** `/etc/cron.daily` **once per day, wait 5 minutes after startup.**
- `7 10 cron.weekly` → **Run weekly jobs 10 minutes after startup.**
- `30 15 cron.monthly` → **Run monthly jobs 15 minutes after startup.**

---

5. Creating a Custom Anacron Job

Suppose you want to run a backup script daily:

```
1 7 backupjob /home/user/backup.sh
```

- **Runs** `/home/user/backup.sh` **once a day.**
- **Starts 7 minutes after system boot.**

---

6. Running Anacron Manually

```
anacron -n   # Run jobs immediately, ignore delays

anacron -d   # Run in foreground (debug mode)

anacron -s   # Run jobs sequentially
```

---

7. Use Cases of Anacron

- **Laptop users (system not always on).**

- Desktops (powered off at night).
- Servers with scheduled downtime.

> Anacron is like cron but designed for machines that are not always on. It ensures jobs run at least once within a given period (daily, weekly, monthly), even if they were missed during downtime. Jobs are defined in `/etc/anacrontab`, with fields for frequency, delay, identifier, and command. For example, `1 5 myjob /home/user/script.sh` means run the script once daily, 5 minutes after system startup."

Configuring Anacron

1. Anacrontab File

- Location: `/etc/anacrontab`

Format:

```
period   delay   job-identifier   command
```

-
  - period: How often to run (in days: 1 = daily, 7 = weekly, 30 = monthly).
  - delay: Delay in minutes after startup before job runs.
  - job-identifier: Unique name for the job (for logs).
  - command: Script or command to execute.

2. Example `/etc/anacrontab`

```
# period delay job-id    command

1   5     cron.daily   run-parts /etc/cron.daily

7   10    cron.weekly  run-parts /etc/cron.weekly

30  15    cron.monthly run-parts /etc/cron.monthly
```

3. Adding a Custom Job

Example: Run a backup script once a day, 7 minutes after boot:

```
1 7 backupjob /home/user/backup.sh
```

✅ Backups – Schedule daily/weekly backups on laptops.
✅ System Updates – Ensure updates run even if PC wasn't on at scheduled time.
✅ Log Rotation – Maintain logs regularly without missing cycles.
✅ Database Maintenance – Run DB cleanup or reindex jobs periodically.
✅ File Synchronization – Daily syncing of files to cloud storage.

💡 Real-world Example:

```
7 15 weeklycleanup /home/user/scripts/cleanup.sh
```

Runs the cleanup script once a week, 15 minutes after boot.

1. What is `ping`?

- `ping` is a simple network diagnostic tool.
- It sends ICMP Echo Request packets to a target host and waits for ICMP Echo Reply.
- Purpose:
  - Check if a host (server, website, or device) is reachable.
  - Measure latency (round-trip time) between your machine and the target.
  - Detect packet loss and network reliability.

💡 In simple words: `ping` asks, *"Are you alive and how fast can I reach you?"*

. Syntax

```
ping [OPTIONS] destination
```

- `destination` = hostname or IP (e.g., `google.com` or `8.8.8.8`).

Option	Meaning
`-c <count>`	Send a fixed number of packets (e.g., `-c 4`).
`-i <interval>`	Time gap between packets (default = 1 second).
`-W <timeout>`	Timeout in seconds to wait for a reply.
`-s <size>`	Packet size in bytes (test MTU issues).
`-q`	Quiet output (summary only).
`-f`	Flood ping (stress test, root only).

. Practical Examples

✅ Example 1: Basic Connectivity Test

```
ping google.com
```

- Sends continuous packets to `google.com`.
- Output shows reply times, packet loss, TTL (time-to-live).

✅ Example 2: Limit Number of Pings

```
ping -c 4 8.8.8.8
```

- **Sends 4 packets to Google's DNS server (`8.8.8.8`).**
- **Commonly used for quick checks.**

✅ Example 3: Test Latency Between Hosts

```
ping -c 5 server.example.com
```

- **Measures round-trip time.**
- **Useful to compare latency between different servers.**

✅ Example 4: Detect Packet Loss

```
ping -c 10 google.com
```

- **Output includes `% packet loss`.**
- **High packet loss = network congestion or faulty link.**

✅ Example 5: Set Packet Size

ping -s 1000 google.com

Sends packets with 1000-byte payload.

Helps test for fragmentation or MTU issues.

✅ Example 6: Timeout for Response

ping -W 2 google.com

Waits 2 seconds max for a reply before declaring timeout.

"`ping` is a Linux network diagnostic tool that uses ICMP packets to check if a host is reachable, measure latency, and detect packet loss. For example, `ping -c 4 google.com` sends 4 echo requests and shows round-trip times. It's often used to troubleshoot connectivity, DNS resolution, or network reliability issues."

1. What is Netstat?

The `netstat` command in Linux is used to **display network connections, routing tables, interface statistics, masquerade connections, and multicast memberships**.
It's an old but still widely used tool for diagnosing network issues and checking which processes are using which ports.

- `netstat` (**network statistics**) is a Linux/Unix command-line tool to monitor:
    - Network connections (incoming & outgoing)
    - Routing tables
    - Interface statistics
    - Open ports and services

It tells you what ports are open, which connections are active, and how traffic flows.

Command	Description
`netstat -a`	Shows **all connections** (listening + established).
`netstat -t`	Shows **TCP** connections only.
`netstat -u`	Shows **UDP** connections only.
`netstat -l`	Shows only **listening** ports.
`netstat -p`	Shows **process ID (PID)** and program name using each socket.
`netstat -n`	Shows addresses and ports as **numbers** (no DNS lookup).
`netstat -r`	Displays the **routing table**.
`netstat -i`	Shows **network interface** statistics.
`netstat -s`	Displays **per-protocol statistics** (TCP, UDP, ICMP, etc.).

2. Basic Syntax

```
netstat [options]
```

Linux Netstat Commands — Cheatsheet

---

1. Show all active connections

Syntax:
```
netstat -a
```
Example:
```
netstat -a
```

- **Explanation**: Shows every socket → listening + established + closing.
- **Description**: Full view of network activity.
- **Use Case**: Quick check of all open/active connections.

---

2. Show only TCP connections

Syntax:
`netstat -t`
Example:
`netstat -t`

- **Explanation**: Displays all TCP connections (web, SSH, DB).
- **Description**: Filters for TCP only.
- **Use Case**: Monitor TCP-heavy apps like HTTP/HTTPS, MySQL, SSH.

---

3. Show only UDP connections

Syntax:
`netstat -u`
Example:
`netstat -u`

- **Explanation**: Displays UDP connections (no handshake, faster).
- **Description**: Filters for UDP only.
- **Use Case**: Check DNS, DHCP, VoIP, streaming.

---

4. Show only listening ports

Syntax:
`netstat -l`
Example:
`netstat -l`

- **Explanation**: Lists services actively **waiting for new connections**.
- **Description**: Focus on listening sockets.
- **Use Case**: Verify if server services are running (e.g., SSH on port 22).

---

5. Show numeric IP/port (skip DNS lookups)

Syntax:
`netstat -n`
Example:
`netstat -an`

- **Explanation**: Displays raw IPs and port numbers instead of hostnames.
- **Description**: Faster output (no name resolution).
- **Use Case**: Script-friendly and avoids DNS delays.

---

6. Show process ID (PID) & program

Syntax:
`sudo netstat -p`
Example:
`sudo netstat -tulnp | grep 8080`

- **Explanation**: Shows which process/program is using port 8080.
- **Description**: Maps open ports to processes.
- **Use Case**: Resolve “port already in use” issues.

---

7. Show all listening ports with process info

Syntax:
`sudo netstat -tulnp`
Example:
`sudo netstat -tulnp`

- **Explanation**: Lists all listening TCP/UDP ports + process IDs.
- **Description**: Most powerful & common combo.
- **Use Case**: Audit server services, detect unauthorized apps.

---

8. Show routing table

Syntax:
`netstat -r`
Example:
`netstat -rn`

- **Explanation**: Displays kernel routing table (like `route -n`).
- **Description**: Shows gateways, destinations, masks.
- **Use Case**: Debug routing issues, ensure correct gateway.

---

9. Show network interface stats

Syntax:
`netstat -i`
Example:
`netstat -i`

- **Explanation**: Displays packet counts, errors, drops per interface.
- **Description**: NIC-level statistics.
- **Use Case**: Check for faulty NIC/drivers or packet loss.

---

10. Show statistics by protocol

Syntax:
`netstat -s`
Example:
`netstat -s`

- **Explanation**: Shows protocol stats (TCP retransmissions, UDP errors).
- **Description**: Protocol-level breakdown.
- **Use Case**: Debug protocol issues (e.g., TCP packet drops).

---

11. Count number of connections on a port

Syntax:
`netstat -an | grep :<port> | wc -l`

-

Example:
`netstat -an | grep :80 | wc -l`

- **Explanation**: Counts total connections to port 80 (HTTP).
- **Description**: Filter by port + count.
- **Use Case**: Detect traffic spikes, check load or DDoS.

---

12. Show established connections only

Syntax:
`netstat -an | grep ESTABLISHED`
Example:
`netstat -an | grep ESTABLISHED`

- **Explanation**: Shows only active connections.
- **Description**: Filters for ESTABLISHED state.
- **Use Case**: Monitor real-time connected clients.

Each port = door of the hotel.

- **`netstat` = CCTV camera → shows:**
  - **Which doors (ports) are open.**
  - **Which guests (clients/IPs) came in.**
  - **How many guests are sitting inside (connections).**
  - **Which staff member (process) opened which door.**

Real-World Use Cases (Interview Friendly)

- **Open port issue:**
  App won't start → use `netstat -tulnp | grep <port>` to find what's blocking it.
- **Network troubleshooting:**
  Too many dropped packets on `netstat -i` → check NIC/driver issues.
- **Security check:**
  See what services are exposed → `netstat -tulpn`.
- **Performance monitoring:**
  Count number of connections per service → `netstat -tn | grep :443 | wc -l`.

What is Traceroute?

- `traceroute` is a network diagnostic tool in Linux/Unix.
- It shows the path packets take from your machine to a destination (like google.com).
- Helps in identifying where the delay or failure occurs in the route.

🔹 How Does Traceroute Work?

1. `traceroute` sends packets (UDP, ICMP, or TCP) with gradually increasing TTL (Time To Live) values.
2. Each router in the path decrements the TTL by 1.
3. When TTL = 0, the router returns an ICMP Time Exceeded message.
4. By collecting these messages, `traceroute` maps the path and response times of each hop.
5. `traceroute` is a network diagnostic tool used to trace the path packets take from your computer to a destination (IP/hostname).
6. Helps detect:
    a. Where latency occurs
    b. Which routers/firewalls may be dropping packets
    c. Route taken by network traffic

Example explanation for interview:
"Traceroute works like a map of the internet path between your machine and the destination, showing each hop and its response time."

How Traceroute Works

1. Traceroute sends packets (UDP by default, ICMP/TCP optionally) with Time-To-Live (TTL) starting at 1.
2. Each router decreases TTL by 1.
3. When TTL = 0, the router responds with ICMP Time Exceeded.
4. Traceroute increases TTL step by step, mapping each hop.

Visual analogy:

- Your packet is a postcard; each router reads it and says, “I got it in X ms,” until it reaches the final destination

🔹 Syntax

```
traceroute [options] <destination> [packet_size]
```

- `<destination>` → hostname or IP (e.g., google.com, 8.8.8.8)
- `[packet_size]` → optional, size of packets to send

Change Packet Length (Size)

- Use the last argument for packet size (default = 60 bytes)

```
traceroute -n -q 3 -w 2 google.com 100
```

- Here `100` → packet size in bytes
- Options explanation:

- `-n` → numeric IPs (no DNS)
- `-q 3` → 3 queries per hop
- `-w 2` → wait 2 seconds for a reply

Use case: Test larger packet sizes to detect MTU issues in the network.

---

5. Change Number of Probes per Hop

- **Default is 3 probes per hop**
- **Option:** `-q <number>`

```
traceroute -q 5 google.com
```

- **Sends 5 probes per hop instead of 3**
- **Use case: More probes → better measurement of latency variation per hop.**

---

6. Specify Destination Port

- **By default, traceroute uses UDP ports 33434+**
- **Change port with** `-p`

```
traceroute -T -p 80 google.com
```

- **Explanation:**
  - `-T` → use TCP instead of UDP
  - `-p 80` → send TCP SYN to port 80 (HTTP)
- **Use case: Some networks block UDP; TCP/port-specific traceroute bypasses that.**

---

7. Use IPv4 or IPv6

- **IPv4 (default):**

```
traceroute -4 google.com
```

- **IPv6:**

```
traceroute -6 ipv6.google.com
```

- **Use case: Test routes in dual-stack networks; verify if IPv6 connectivity works.**

---

8. Route Through a Specific Gateway

- Use `-g <gateway>` to force traceroute through a specific gateway (source routing).

```
traceroute -g 192.168.1.1 google.com
```

- Explanation: Packet will go via `192.168.1.1` first.
- Use case: Test specific ISP path, VPN routing, or troubleshoot multi-homed networks.

---

9. Quick Examples

1. Basic traceroute:

```
traceroute google.com
```

2. Numeric IP + larger packet:

```
traceroute -n google.com 120
```

3. TCP port 443 (HTTPS):

```
traceroute -T -p 443 google.com
```

4. IPv6 traceroute:

```
traceroute -6 ipv6.google.com
```

5. 5 probes per hop:

```
traceroute -q 5 google.com
```

Core Difference

- `su` ("substitute user")
    - Switches your shell to another user account.
    - Requires the target user's password (e.g., root's password).
    - Gives you a full session as that user until you exit.
- `sudo` ("superuser do")
    - Runs a single command as another user (default = root).
    - Requires your own password, provided you are authorized in `/etc/sudoers`.
    - After the command finishes, you're back as yourself.

1. Difference Between `su` and `sudo`

Command	Meaning	Example	Use Case
`su`	Switch User → Opens a shell as another user (default = root) after entering that user's password.	`su -` → switch to root user	When you want to become another user for a long session (root or service account).
`sudo`	SuperUser Do → Runs a single command with elevated privileges after entering your own password.	`sudo apt update`	When you want to run just one or few commands as root/admin without fully switching.

`su` = become another user.

`sudo` = run a command as another user (default = root).

Aspect	`su`	`sudo`
Authentication	Needs the password of the target user (e.g., root).	Needs your own password if listed in `/etc/sudoers`.
Scope	Opens a new shell session as that user. You stay root (or other user) until `exit`.	Runs only one command with elevated privileges. Then returns to your session.
Default target	Root (`su -` = root login shell).	Root (`sudo command` = run as root).
Security model	Everyone who needs root access must know the root password.	Users can be given granular access (some commands only, not full root).
Auditing	Harder to track, because once switched, all actions are by root.	Every sudo command is logged (`/var/log/auth.log`), showing who ran what.
Usage style	Long administrative sessions as root.	Quick administrative actions without fully switching.

1. What is SSH?

SSH (Secure Shell) is a protocol used to securely connect to remote systems (like Linux servers) over a network.

SSH stands for Secure Shell — it's a protocol used in Linux (and other systems) to securely connect to and control remote computers over a network.

It encrypts all communication between your machine (the client) and the remote server (the host), protecting passwords, commands, and data from eavesdropping.

- **It encrypts communication (unlike old `telnet` or `rlogin` which send passwords in plain text).**
- **Provides confidentiality (encryption), integrity (no tampering), and authentication (verify who you are).**
- **Commonly used for:**
  - **Remote login to servers**
  - **Executing commands remotely**
  - **File transfers (via `scp` or `sftp`)**
  - **Tunneling/port forwarding**

2. Why it is called SSH?

- **SSH = Secure Shell**
- **Called so because:**
  - **It provides a shell (command-line interface) to interact with a remote machine.**
  - **It's secure because all traffic (passwords, commands, outputs) is encrypted.**

🔑 3. Basics of SSH

- **Default port: 22**
- **Authentication methods:**
  - **Password-based → enter your password to log in.**
  - **Key-based → use a pair of SSH keys (`id_rsa` private, `id_rsa.pub` public).**
- **Config files:**
  - **`/etc/ssh/sshd_config` → SSH server configuration**
  - **`~/.ssh/known_hosts` → stores fingerprints of servers you connected to**
  - **`~/.ssh/id_rsa` & `~/.ssh/id_rsa.pub` → your private/public keys**

Example of SSH

```
# Connect to server at 192.168.1.100 as user 'swapnil'

ssh swapnil@192.168.1.100



# Connect using custom port (say 2222)

ssh -p 2222 swapnil@192.168.1.100



# Connect using private key

ssh -i ~/.ssh/mykey.pem swapnil@192.168.1.100
```

6. Access Linux Server using PuTTY (Windows)

PuTTY is a popular SSH client for Windows.
Steps:

1. **Download & install PuTTY.**

   Open PuTTY → In Host Name, enter:
   `username@server_ip`

2. **or just `server_ip` and specify username after connection.**
3. **Default port is 22.**
4. **Click Open.**
5. **Enter username & password (or load private key `.ppk` if using key-based login).**
6. **SSH = Secure Shell protocol for safe remote login.**
7. **Called SSH because it provides a secure shell.**
8. **Basics: port 22, password or key auth, encrypted communication.**
9. **Usage: `ssh user@server_ip`.**
10. **On Windows: use PuTTY.**
11. **From another Linux server: same `ssh` command works.**

What is a Firewall in Linux?

- **A firewall is a network security layer that controls traffic (inbound/outbound) based on rules.**
- **It protects servers from unauthorized access, malware, and DDoS attacks.**
- **Works at packet level (network layer).**

---

2. Why do we need Firewall?

- **To allow trusted traffic (e.g., SSH, web ports).**
- **To block/deny untrusted traffic (e.g., unknown IPs, unused ports).**
- **To enforce least privilege principle (only necessary ports open).**
- **To improve security posture of servers.**

SUID/SGID

“In Linux, apart from normal read, write, and execute permissions, we also have special permissions called SUID and SGID.

In Linux, every file has permissions for user, group, and others. Normally, when a program runs, it runs with the permissions of the user who executed it.

But sometimes, we need a program to run with the permissions of the file owner (not the user). That's where SUID and SGID come in.

SUID, or Set User ID, means when a file is executed, it runs with the privileges of the file owner instead of the user who runs it. A common example is the `/usr/bin/passwd` command. Even though a normal user runs it, it needs root privileges to update `/etc/shadow`, so it has the SUID bit set.

What is SUID?

- **SUID (Set User ID) is a special permission in Linux.**
- **When applied to an executable file:**
  - **The program runs with the file owner's privileges, not the privileges of the user who runs it.**
- **Commonly used when normal users need temporary elevated permissions to run a program.**

Example of SUID

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

You'll see something like:

```
-rwsr-xr-x 1 root root 54256 Aug  1  /usr/bin/passwd
```

Notice the `s` in the user (owner) permissions → that's SUID.

- **`passwd` command needs to modify `/etc/shadow` (owned by root).**
- **Even when a normal user runs `passwd`, it executes with root privileges because of SUID.**

What is SGID?

- **SGID (Set Group ID) works similarly but for groups.**
- **When applied to:**
  1. **Executable file → the program runs with the group privileges of the file.**
  2. **Directory → any file created inside inherits the group ownership of the directory, not of the user's default group.**

🔹 Example of SGID (on directory)

```
sudo mkdir /shared

sudo chgrp developers /shared

sudo chmod 2775 /shared
```

- **The 2 in `2775` sets SGID on the directory.**
- **Now, any file created in `/shared` will automatically belong to the `developers` group.**
- **This ensures collaboration without changing user's default group.**

SGID, or Set Group ID, works in two ways:

- **On an executable file, it runs with the file's group privileges.**
- **On a directory, it ensures that all new files created inside automatically inherit the group ownership of that directory. This is commonly used in shared directories, for example, a `developers` group working in `/shared`.**

So in short, SUID gives temporary owner's power to the user, and SGID ensures group-based collaboration or execution with group rights."

- **SUID: Runs a file with owner's permissions (example: `/usr/bin/passwd`).**
- **SGID: Runs a file with group's permissions; on directories, new files inherit the directory's group (useful for shared project folders).**

🔹 What is Sticky Bit?

- **The Sticky Bit is another special permission in Linux.**
- **When set on a directory, it means:**
  - **Users can create files inside the directory (if they have write permission).**
  - **But they can only delete or rename their own files, not files owned by others.**

👉 This prevents users from deleting each other's files in shared directories.

The Sticky Bit is a special permission in Linux, mainly used on directories. It allows all users to create files inside the directory, but only the file's owner or root can delete or rename them. A common example is `/tmp`, where every user can create temporary files but cannot delete each other's files. You can identify it by the 't' at the end of permissions, and set it using `chmod +t`."

What is UMASK?

- **UMASK (User File Creation Mask) is a setting in Linux that defines the default permission bits for newly created files and directories.**
- **Instead of directly specifying permissions, it acts like a filter (mask) that subtracts permissions from the system's default.**

"UMASK in Linux defines the default permissions for newly created files and directories. Files normally start with 666 and directories with 777, and UMASK subtracts permissions from this. For example, with UMASK 022, new files get 644 (`rw-r--r--`) and directories get 755 (`rwxr-xr-x`). You can check it with `umask`, change it temporarily using `umask value`, and make it permanent by adding the setting in shell profile files."

Linux `chown` and `chgrp` Commands

🔹 1. Ownership in Linux

Every file in Linux has two types of ownership:

- **User (owner) → usually the person who created the file.**

- **Group → a set of users who share access to the file.**

🔹 2. `chown` (Change Owner)

- **Used to change the user ownership (and optionally group ownership) of a file or directory.**

Syntax:

```
chown [options] new_owner filename
```

Examples:

```
# Change owner to 'swapnil'

sudo chown swapnil file.txt


# Change both owner and group

sudo chown swapnil:developers project.log


# Recursively change owner/group for a directory

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

🔹 3. `chgrp` (Change Group)

- **Used only to change the group ownership of a file or directory.**

Syntax:

```
chgrp [options] new_group filename
```

Examples:

```
# Change group to 'developers'

sudo chgrp developers file.txt
```

```
# Recursively change group

sudo chgrp -R developers /project
```

🔹 4. Difference between `chown` and `chgrp`

- `chown` → can change both owner and group.
- `chgrp` → changes only the group.
- In fact, `chgrp` is basically a shortcut; you can achieve the same with `chown :group file`.

🔹 5. Interview-Ready Summary

"In Linux, every file has a user and group owner. The `chown` command changes the file's user ownership, and optionally its group ownership, while the `chgrp` command changes only the group. For example, `chown swapnil:developers file.txt` changes both owner and group, while `chgrp developers file.txt` only changes the group. Both support `-R` to apply changes recursively."

Log files are automatically generated computer files that contain records of events, operations, and activities within a system, application, or device, such as timestamps, errors, and user actions. They serve as a historical record to track performance, diagnose problems, and investigate security incidents. Log files are crucial for understanding system behavior and are generated by operating systems, servers, applications, and other devices.

in Linux, log files are system-generated text files that record activities, events, and messages from the operating system, services, and applications. They are extremely important for troubleshooting, auditing, and monitoring system behavior.

🔹 What are Log Files?

- A log file is like a diary of the system — it keeps track of what's happening.
- Logs may include:
  - User login attempts
  - Errors
  - Security events
  - Application behavior
  - Kernel/system messages
- By reading logs, an administrator can find out what went wrong and when.

What is Log Monitoring in Linux?

- Log monitoring means continuously checking system/application log files to detect issues, security breaches, or abnormal behavior.
- Since logs contain critical information, monitoring them helps proactive troubleshooting instead of waiting for failures.

🔹 Why is Log Monitoring Important?

1. Detect security issues → Failed login attempts, unauthorized access.
2. Identify system errors → Hardware failures, kernel panics.
3. Monitor application performance → Web servers, databases.
4. Ensure compliance/auditing → Legal or regulatory requirements.

---

1. Basic Commands

- `tail -f /var/log/syslog` → Real-time monitoring of system log.
- `less /var/log/auth.log` → Search inside authentication logs.
- `grep "Failed" /var/log/auth.log` → Find failed login attempts.
- `journalctl -xe` → View logs managed by systemd journal.

Log monitoring in Linux means keeping track of log files in `/var/log/` to detect errors, security issues, and performance problems. I usually use `tail -f`, `journalctl`, and `grep` for real-time analysis. For automation, I configure `logrotate` to manage log sizes and sometimes use `logwatch` for daily reports. In enterprise environments, centralized solutions like ELK stack or Splunk are preferred for large-scale monitoring."

What is `dmesg`?

- `dmesg` stands for “diagnostic message”.
- It prints the kernel ring buffer, which contains system messages related to hardware, drivers, and boot processes.
- Useful for monitoring hardware events, detecting device errors, and troubleshooting boot issues.

🔹 Why Use `dmesg`?

- Detect hardware issues: disks, USB devices, network cards.
- Check driver/module loading: kernel modules during boot.
- Monitor boot messages: errors, warnings, or kernel panics.
- Troubleshoot system crashes or freezes.

“The `dmesg` command in Linux reads the kernel ring buffer, which contains messages about hardware, drivers, and system events. It’s commonly used to monitor system hardware, check driver load status, or troubleshoot boot and device issues. For example, `dmesg | grep -i usb` shows messages related to USB devices, and `dmesg -T`converts timestamps to readable format. For live monitoring, `dmesg --follow` can be used.”

What is Logrotate?

- Logrotate is a Linux utility that manages log files automatically.
- It prevents logs from consuming too much disk space by:
  - Rotating (archiving old logs)
  - Compressing logs
  - Deleting old logs
  - Mailing logs to admins

- **Runs automatically via cron job (usually daily).**

🔹 Why Logrotate is Needed?

- **Log files in `/var/log/` keep growing → can fill up disk space.**
- **Old logs are rarely needed daily, so archiving them is better.**
- **Keeps system logs organized, small, and easy to manage.**

🔹 Default Location

- **Global config: `/etc/logrotate.conf`**
- **Service-specific configs: `/etc/logrotate.d/`**

Running Logrotate Manually

```
# Test config

logrotate -d /etc/logrotate.conf

# Force rotation

logrotate -f /etc/logrotate.conf
```

Logrotate is a Linux utility that automatically manages log files by rotating, compressing, and deleting old logs. It prevents logs from consuming disk space and keeps them organized. For example, I can configure `/etc/logrotate.d/httpd` to rotate Apache logs daily, keep 7 backups, and compress old ones. It runs automatically via cron, but I can also trigger it manually using `logrotate -f`."

What is `rsync`?

- rsync = Remote Synchronization.
- A command-line tool for copying and synchronizing files/directories between:
  - Local → Local
  - Local → Remote
  - Remote → Local
- It copies only the differences (deltas) instead of full files → makes it fast & efficient.

🔹 Why Use `rsync`?

- Faster than `scp` or `cp` (copies only changed parts).
- Preserves file permissions, ownership, and timestamps.
- Supports compression during transfer.
- Can delete files on destination if deleted on source (for exact sync).
- Can run over SSH → secure transfers.

Basic Syntax

```
rsync [options] source destination
```

🔹 Common Options

- `-a` → Archive mode (preserves permissions, symlinks, etc.).
- `-v` → Verbose (detailed output).
- `-z` → Compress data during transfer.
- `-h` → Human-readable sizes.
- `--delete` → Remove files from destination if not in source.
- `-e ssh` → Use SSH for secure transfer.

Examples

1. Copy a file locally

```
rsync -avh /home/user/file.txt /backup/
```

👉 Copies `file.txt` to `/backup/` with attributes preserved.

2. Copy a directory to another server

```
rsync -avz /home/user/ user@192.168.1.10:/backup/
```

👉 Syncs `/home/user/` to `/backup/` on remote server securely via SSH.

3. Sync from remote server to local

```
rsync -avz user@192.168.1.10:/var/log/ /home/user/logs/
```

Downloads logs from remote server.

Real-World Use Cases

1. Server Backups → Copy `/etc/` or `/var/log/` to backup server.
2. Website Deployment → Sync website code to production.
3. Log Collection → Pull logs from multiple servers.
4. Incremental Backups → Faster because only changes are transferred.

Interview-Ready Explanation

👉 *"Rsync is a Linux utility for synchronizing files and directories locally or remotely. Unlike `scp`, it copies only changes, which makes it efficient. For example, I can run `rsync -avz /home/user/ user@server:/backup/` to sync my local home directory to a server over SSH. It preserves file permissions and supports options like `--delete` to mirror directories and `--dry-run` to test before running. It's commonly used for backups and deployments."*

What is Linux Hardening?

- *Linux Hardening = applying security measures to reduce vulnerabilities and protect the system from attacks.*
- *Goal: Make the system more secure, less exploitable, and compliant with policies.*

Linux awk Command

awk is a powerful text-processing command. It works line by line and divides each line into fields (default separator is whitespace).

Syntax

```
awk 'pattern { action }' filename
```

- **pattern → what to search for**
- **action → what to do when the pattern matches**

Examples

1. **Print entire file:**

```
awk '{print}' file.txt
```

2. **Print only the first column:**

```
awk '{print $1}' file.txt
```

3. **Print first and third column:**

```
awk '{print $1, $3}' file.txt
```

4. **Filter lines where column 2 equals "swapnil":**

```
awk '$2 == "swapnil" {print $0}' file.txt
```

5. **Use custom delimiter (e.g., `:` in `/etc/passwd`):**

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

Linux `cut` Command

`cut` extracts specific columns or fields from text.

Syntax

```
cut OPTION... [FILE]
```

Options

- `-c` → select by character position
- `-f` → select by field number
- `-d` → specify delimiter

Examples

1. Extract first 5 characters from each line:

```
cut -c1-5 file.txt
```

2. Extract 2nd and 4th fields (delimiter `:`):

```
cut -d: -f2,4 /etc/passwd
```

3. Show only first field (like username from `/etc/passwd`):

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

4. Extract characters from 3rd to 7th position:

```
cut -c3-7 file.txt
```

AWK vs CUT

Feature	`awk`	`cut`
Power	More powerful (supports conditions, loops, formatting)	Simple extraction
Delimiter	Default = whitespace (can change with `-F`)	Must specify with `-d` (default TAB)
Use cases	Filtering, reporting, text formatting	Quick field/column extraction
Example	`awk -F: '{print $1,$7}' /etc/passwd`	`cut -d: -f1,7 /etc/passwd`