Devops Shack

THE ULTIMATE 100 LINUX QUESTIONS FOR INTERVIEW SUCCESS

BY DEVOPS SHACK

DevOps Shack.com

DevOps Shack

The Ultimate 100 Linux Questions for Interview Success

📜Table of Contents

10. Explain sticky bit, SUID, and SGID with examples.

☑ Section 3: Linux Shell and Scripting

11. What is a shell? List different types of shells in Linux.

12. What are environment variables? How do you manage them?

13. What is the difference between `cron`, `at`, and `systemd timers`?

14. Explain how to write and debug a shell script.

15. What is the significance of `#!/bin/` in a shell script?

☑ Section 4: Linux User and Permission Management

16. How do you create, delete, and manage users in Linux?

17. Explain how groups work in Linux. Difference between primary and secondary groups?

18. How to set password aging policies and enforce them?

19. What is `/etc/passwd`, `/etc/shadow`, and `/etc/group`?

20. How do you manage sudo access for a user?

☑ Section 5: Process Management and Job Scheduling

21. Explain `ps`, `top`, `htop`, `nice`, `renice`, and `kill` commands.

22. What is the difference between foreground and background processes?

23. How do you trace zombie and orphan processes?

24. What is `strace` and how do you debug using it?

25. How does the `cron` scheduler work? How to schedule jobs with examples?

☑ Section 6: Disk and Storage Management

26. How to create and manage partitions using `fdisk` or `parted`?

27. Explain LVM (Logical Volume Manager) in detail.

28. How to mount and unmount file systems?

29. How do you monitor disk space usage with `df`, `du`, and `ncdu`?

30. What is the difference between ext3, ext4, xfs, and btrfs?

☑ Section 7: Networking in Linux

31. Explain how IP addressing and subnetting works in Linux.

32. How do you check and configure network interfaces?

33. Explain `netstat`, `ss`, `ip`, and `ifconfig` commands.

34. How do you troubleshoot network issues using `ping`, `traceroute`, `nmap`, and `tcpdump`?

35. How does Linux firewall (`iptables` and `firewalld`) work?

☑ Section 8: System Monitoring and Performance Tuning

36. How to monitor system performance with `top`, `vmstat`, `iostat`, `sar`, and `dstat`?

37. What is load average? How to interpret it?

38. How to monitor memory usage in real-time?

39. How to analyze logs and system events?

40. What are cgroups and how do they help in resource management?

☑ Section 9: Package Management

41. How does package management work in Debian vs RedHat-based systems?

42. What are the differences between `apt`, `yum`, `dnf`, and `zypper`?

43. How to build and install packages from source?

44. What is a `.deb` and `.rpm` file? How are they created?

45. How do you handle dependency issues in Linux?

☑ Section 10: Systemd and Boot Management

46. What is `systemd` and how is it different from `init`?

47. How do you manage services using `systemctl`?

48. How to create a custom service in systemd?

49. How do you troubleshoot boot failures in Linux?

50. Explain `journalctl` and log management in systemd.

☑ Section 11: Linux Security

51. What are the best practices to secure a Linux system?
52. How does SELinux or AppArmor work?
53. How to set up firewall rules using `iptables` or `ufw`?
54. How do you audit logs and file integrity?
55. What is SSH hardening? How do you secure remote access?

☑ Section 12: Linux Commands – In-depth

56. Advanced usage of `grep`, `sed`, `awk`, `cut`, and `tr`.
57. What is the difference between `find` and `locate`?
58. How to use `xargs` effectively?
59. How to search for patterns recursively using `grep` and `find`?
60. Explain piping (`|`) and redirection (`>`, `>>`, `2>`) with examples.

☑ Section 13: Kernel, Modules, and Updates

61. What is the role of the Linux kernel?
62. How do you view and manage kernel modules?
63. What is `dmesg` and how do you interpret its output?

64. How do you update the kernel safely?
65. How do you compile a custom kernel?

☑ Section 14: Logs and Troubleshooting

66. Where are the major log files stored in Linux?
67. How to troubleshoot system crashes?
68. How to use `logrotate` for log management?
69. How to enable and analyze audit logs?
70. How to recover from a broken or corrupted filesystem?

☑ Section 15: Linux for DevOps and Cloud

71. How to use Linux in cloud environments like AWS/GCP?
72. What is `cloud-init` and how does it work?
73. How to write scripts for CI/CD pipelines?
74. How to use Linux with Docker?
75. How to use Linux with Kubernetes (e.g. kubelet, containerd)?

☑ Section 16: Advanced Troubleshooting and Production Scenarios

76. How to troubleshoot high CPU or memory usage in production?
77. How to debug application crashes and service failures?
78. How do you perform a root cause analysis of a production issue?

79. How to trace open ports and detect malware?
80. How to track file access and audit trails?

☑ Section 17: Backup and Recovery

81. How to use `rsync`, `tar`, `dd`, and `scp` for backups?
82. How to automate backup tasks in production?
83. How to restore deleted files from snapshots?
84. How to configure and restore a bootable system image?
85. How do you manage offsite backups securely?

☑ Section 18: Virtualization and Containers

86. How do containers differ from VMs in Linux?
87. How to use `chroot`, `lxc`, or `Docker` on Linux?
88. What is KVM? How to set up a virtual machine using KVM?
89. How does container networking work?
90. How to troubleshoot container issues on a Linux host?

☑ Section 19: Automation and Configuration Management

91. What is Ansible and how does it manage Linux servers?
92. How do you use scripting for configuration automation?

93. How to automate Linux patching and updates?

94. What is Infrastructure as Code (IaC) in Linux environments?

95. How do you use cron jobs for automated health checks?

☑ Section 20: Scenario-Based and Behavioral Questions

96. Describe a time when you resolved a critical system failure.

97. How do you prioritize multiple system alerts?

98. Describe a production issue you handled end-to-end.

99. How do you manage Linux systems at scale?

100. What are your go-to tools for Linux troubleshooting and why?

🆕Introduction

Linux is the backbone of modern computing—from cloud servers and supercomputers to embedded systems and smartphones. Whether you're applying for a system administrator role, DevOps engineer, cloud architect, or security analyst, a strong grasp of Linux is essential.

This guide covers 100 of the most commonly asked and **technically detailed Linux interview questions**, ranging from beginner concepts to complex real-world production scenarios. Each question is followed by a **deep-dive explanation**, usage examples, relevant command syntax, and **real-life use cases** so that you're not just memorizing facts—you're gaining insight.

This document is tailored for:

- System administrators
- DevOps & Cloud engineers
- Linux enthusiasts
- Interview preparation
- Job transitions from QA/manual testing to infrastructure roles

By the end, you'll not only be **interview-ready** but also **production-ready**.

1. What is Linux and how is it different from Unix?

Answer:

Linux is a free, open-source operating system based on Unix. It was originally developed by Linus Torvalds in 1991 and has grown to power everything from personal computers to supercomputers, mobile devices, servers, and embedded systems.

Key differences between Linux and Unix:

- **Source code:** Linux is open source; Unix is mostly proprietary (e.g., AIX, HP-UX).
- **Cost:** Linux is free to use and modify; Unix often requires a commercial license.
- **Hardware compatibility:** Linux supports a wide range of hardware, while Unix is typically hardware-specific.
- **User base:** Linux is community-driven with a huge ecosystem; Unix is mostly used in enterprise environments.
- **Variants:** Linux has multiple distributions like Ubuntu, CentOS, Debian; Unix has versions like Solaris, AIX, HP-UX.

In production environments, Linux is often favored due to its flexibility, cost-effectiveness, and massive community support.

2. Explain the Linux boot process step-by-step.

Answer:

The Linux boot process has 6 major stages:

1. **BIOS (Basic Input/Output System)**
   This is the firmware on the motherboard. It checks hardware (Power-On Self Test - POST) and looks for a boot device (like HDD/SSD).

2. **MBR (Master Boot Record) or UEFI**
   If BIOS finds a valid bootloader in the MBR (first 512 bytes of the disk), it passes control to it. Modern systems use UEFI instead of MBR.

3. **GRUB (Grand Unified Bootloader)**
   GRUB is the default bootloader for most Linux systems. It lets you choose which OS/kernel to load and loads the selected kernel into memory.

4. **Kernel Loading**
   The Linux kernel is loaded into memory and initialized. It detects hardware, mounts the root filesystem, and starts the first process (usually `init` or `systemd`).

5. **Init/Systemd**
   The init system takes over and begins starting services based on the runlevel or target. On most modern systems, it's `systemd`.

6. **Login Prompt (TTY or GUI)**
   Once all services are started, the user gets a login prompt (CLI or graphical login screen).

3. What are runlevels? How are they configured in modern distros?

Answer:

Runlevels are states of the machine that define what services are running. In traditional `SysVinit` systems, they were used to control system behavior:

- 0 – Halt
- 1 – Single user mode
- 2 – Multi-user (without networking)
- 3 – Multi-user (with networking)
- 4 – Unused/custom
- 5 – Multi-user with GUI
- 6 – Reboot

In modern systems using systemd, runlevels are replaced by **targets**:

- `runlevel3.target` → `multi-user.target`
- `runlevel5.target` → `graphical.target`

Command to check current target:

```
systemctl get-default
```

To change the default target (like changing runlevel):

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

To switch immediately:

```
systemctl isolate graphical.target
```

Note: If you're working on older distros (RHEL 6, CentOS 6), `runlevel` and `/etc/inittab` are still used. In systemd, the equivalent logic is managed by symbolic links inside `/etc/systemd/system/default.target`.

4. What are the major components of the Linux operating system?

Answer:

Linux OS has 4 major components:

1. **Kernel**
   Core part of the OS. It manages hardware, memory, processes, I/O, and system calls. It abstracts hardware so apps don't need to deal with it directly.

2. **System Libraries**
   These are special functions or programs that apps use to interact with the kernel (e.g., `glibc` – GNU C library).

3. **System Utilities**
   Tools and programs that perform individual, user-level tasks. Examples include `ls`, `cp`, `mkdir`, etc. These come from coreutils and other packages.

4. **User Interface**
   This can be CLI (Command Line Interface) or GUI (Graphical User Interface). CLI uses shells like , zsh. GUI uses X Window System, GNOME, KDE, etc.

Analogy: Think of the kernel as the engine of a car, libraries as wiring, utilities as controls, and user interfaces as the dashboard.

5. What are the differences between a process and a thread?

Answer:

Both processes and threads are used to execute code but they differ in how they operate and share system resources.

Process:

- Independent execution unit.
- Has its own memory space, file descriptors, and address space.
- More overhead for context switching.
- Created using `fork()` in Linux.

Thread:

- Lightweight execution unit inside a process.
- Shares memory and resources with other threads in the same process.
- Faster to create and manage.
- Created using `pthread_create()` or similar threading libraries.

Example:
A browser can be a process. Each tab can be a thread. They share

memory (like bookmarks, cache) but perform tasks independently (load different web pages).

Commands:

- `ps -ef` or `top` shows processes.
- `ps -L -p <PID>` shows threads inside a process.

Use case difference:
Use threads when tasks need shared memory (e.g., real-time applications), use processes when tasks are isolated or fault-tolerant (e.g., microservices).

6. What is the Linux Filesystem Hierarchy Standard (FHS)?

Answer:

The **Filesystem Hierarchy Standard (FHS)** defines the directory structure and directory contents in Linux distributions. It ensures consistency so that users, developers, and scripts know where to find or place files.

Here’s a breakdown of the most important directories:

- `/` – Root of the filesystem. Everything starts here.
- `/bin` – Essential binary commands (e.g., `ls`, `cp`, `mv`, `cat`). Available in single-user mode.
- `/sbin` – System binaries (e.g., `fsck`, `reboot`, `iptables`). Used by root/admin.
- `/etc` – Configuration files (e.g., `/etc/passwd`, `/etc/ssh/sshd_config`).

- `/dev` – Device files (e.g., `/dev/sda`, `/dev/null`).
- `/proc` – Virtual filesystem with kernel and process info (e.g., `/proc/cpuinfo`, `/proc/meminfo`).
- `/var` – Variable data (e.g., logs, mail, spool files).
- `/tmp` – Temporary files. Cleaned on reboot.
- `/usr` – Secondary hierarchy for read-only user applications and libraries.
- `/home` – User home directories.
- `/boot` – Files needed for booting (e.g., kernel, GRUB).
- `/lib`, `/lib64` – Shared libraries needed to boot the system.

Pro Tip: During troubleshooting or system recovery, knowing where config, logs, and binaries live is critical.

7. What is an inode in Linux? How does the filesystem use it?

Answer:

An **inode (index node)** is a data structure that stores metadata about a file, excluding the file name.

What does an inode contain?

- File type (regular, directory, block device, etc.)
- File permissions and ownership (UID/GID)

- Size
- Timestamps (accessed, modified, changed)
- Number of hard links
- Pointers to disk blocks (where actual data is stored)

File name is not stored in inode, it's stored in the **directory entry**, which maps the name to an inode number.

How to view inode information:

```
ls -li
```

View inode usage:

```
df -i
```

Why it matters in real life:
Even if your disk has space, if you run out of inodes, you can't create new files. This is common in directories with millions of small files (e.g., mail queues, cache dirs).

8. What are hard links and soft (symbolic) links? When should you use each?

Answer:

Hard Link:

- A hard link is an additional name for an existing file.
- Both the original file and the hard link point to the **same inode**.

- Deleting the original file does **not** delete the content as long as a hard link exists.
- Cannot link across different filesystems.

Create hard link:

```
ln original.txt hardlink.txt
```

Soft Link (Symbolic Link):

- A soft link is like a shortcut. It stores the **path** to another file.
- Points to the filename, not the inode.
- If the target is deleted, the symlink becomes broken (dangling).
- Can link across filesystems and to directories.

Create soft link:

```
ln -s /path/to/original symlink.txt
```

Use hard links when:

- You want multiple names for the same data on the same filesystem.

Use soft links when:

- You need flexibility (link across mounts, point to directories, etc.).

Check link types:

```
ls -l
```

```
# Hard links: same inode

# Symlinks: show -> in the output
```

9. How does Linux handle file permissions? Explain chmod, chown, and umask.

Answer:

File permissions in Linux control access at the user, group, and others level.

Each file has:

- **User (u)** – owner
- **Group (g)** – group associated
- **Others (o)** – everyone else

Permissions:

- `r` – read
- `w` – write
- `x` – execute

Example:

```
-rwxr-xr-- 1 user group 1234 file.sh
```

- User: read, write, execute
- Group: read, execute

- Others: read

chmod (change mode): Used to modify file permissions.

Symbolic:

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

Numeric:

```
chmod 755 script.sh
# 7 = rwx, 5 = r-x, 5 = r-x
```

chown (change ownership): Used to change file owner or group.

```
chown root:admin file.txt
```

umask (user mask): Defines default permissions for new files and directories.

```
umask
# Common value: 0022 → files: 644, dirs: 755
```

It subtracts permissions from the base:

- Base for files: 666
- Base for dirs: 777

So:

```
666 - 022 = 644 (rw-r--r--)
777 - 022 = 755 (rwxr-xr-x)
```

Interview Tip: Be ready to calculate umask effect and explain the difference between chmod +x and numeric equivalents.

10. Explain sticky bit, SUID, and SGID with examples.

Answer:

These are **special permission bits** in Linux:

Sticky Bit (`t`)

- Used on directories.
- Only the owner (or root) can delete files within the directory, even if others have write access.
- Common on `/tmp`.

Set it:

```
chmod +t

/shared ls -ld

/shared #

drwxrwxrwt
```

SUID (Set User ID):

- Executes a file with the **file owner's permissions**, not the user's.

Example: `/usr/bin/passwd` – owned by root, but users can run it to change their own password.

```
chmod u+s /path/to/program
ls -l
# -rwsr-xr-x
```

SGID (Set Group ID):

- On files: runs with group privileges.
- On directories: new files inherit the directory's group.

Example:

```
chmod g+s
/project_dir ls -ld
/project_dir #
drwxrwsr-x
```

All files created inside will belong to the same group.

Real-world use cases:

- Sticky bit for shared temp folders.
- SUID for critical system binaries that require elevated permissions.
- SGID for group collaboration on shared folders.

11. Explain how to write and debug a basic shell script in Linux.

Answer:

A **shell script** is a file containing a series of shell (usually) commands that are executed in sequence.

Basic structure:

```
#!/bin/

echo "Hello, $USER!"
```

- The first line `#!/bin/` is called a **shebang**, telling the OS which interpreter to use.
- Save the file as `script.sh`, then make it executable:

```
chmod +x script.sh

./script.sh
```

Debugging techniques:

1. **Run script in debug mode:**

```
-x script.sh
```

This shows each command before it is executed (great for catching logic errors).

2. **Set internal debug flag:**

```
#!/bin/

set -x  # Turn on debugging

echo "This will show step-by-step"

set +x  # Turn off debugging
```

3. **Syntax checking without running:**

```
-n script.sh
```

4. **Log outputs for review:**

```
./script.sh >output.log 2>&1
```

5. **Use `echo` or `printf` to inspect variable values.**

Real-World Tip: Always check for `-n`, and make sure your scripts work with `set -e` to abort on any command failure—especially in CI/CD pipelines.

12. What are environment variables? How do you manage them in Linux?

Answer:

Environment variables are **key-value pairs** that define the working environment for processes. They affect how processes behave.

Common ones:

- `PATH`: Locations to search for executable files.
- `HOME`: User's home directory.
- `USER`: Current username.
- `SHELL`: Default shell.

View all environment variables:

```
Printenv
```

`Env`

Set environment variables (temporarily):

```
export VAR_NAME=value
```

```
echo $VAR_NAME
```

This exists only in the current session.

Persist variables:

- For current user:
  - Add to `~/.rc`, `~/._profile`, or `~/.profile`
- For all users:
  - Add to `/etc/environment` or `/etc/profile`

Example:

```
export JAVA_HOME=/usr/lib/jvm/java-11
```

```
export PATH=$PATH:$JAVA_HOME/bin
```

Unset a variable:

```
unset VAR_NAME
```

Pro Tip: Always quote variables if they may contain spaces:

```
echo "$HOME"
```

13. What is the difference between `cron`, `at`, and `systemd timers`?

Answer:

All three are used for **task scheduling**, but they serve different use cases.

`cron`: Recurring jobs

- Runs tasks at specified intervals (daily, weekly, hourly).
- Syntax:

```
* * * * * /path/to/script.sh
```

(Min Hour Day Month Weekday Command)

- Manage using:

```
crontab -e    # Edit user's cron jobs
crontab -l    # List them
```

- System-wide jobs: `/etc/crontab`, `/etc/cron.d/`

`at`: One-time tasks

- Run a command once at a specific time.

```
at 10:30 AM
> echo "Hello" >> /tmp/test.txt
> <Ctrl+D>
```

- View jobs:

```
atq
```

- Remove jobs:

```
atrm <job_number>
```

`systemd timers`: Modern and powerful

- Replacement for cron, especially in systemd-enabled distros.
- Allows better control, logging via `journalctl`, and integration with services.

Example:

- Create a timer `.timer` and a service

`.service`.

```
# mytask.service
[Service]
ExecStart=/path/to/script.sh
# mytask.timer
[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true
[Install]
WantedBy=timers.target
```

Enable the timer:

```
systemctl enable --now mytask.timer
```

Choose:

- Use `cron` for recurring tasks.

- Use `at` for one-offs.
- Use `systemd timers` for production-grade scheduling with logs and system integration.

14. What is the significance of `#!/bin/` in a shell script?

Answer:

This line is known as a **shebang** or hashbang.

- It tells the **kernel** which interpreter to use to execute the script.
- Without it, the script will be run using the current shell, which may not behave consistently (e.g., `sh`, `dash`, `zsh`, etc.).

Example:

```
#!/bin/

echo "Running with "
```

Other valid shebangs:

- `#!/bin/sh` → POSIX-compliant shell (often symbolic link to `dash` or)
- `#!/usr/bin/env python3` → Portable way to invoke Python
- `#!/usr/bin/env` → Preferred in portable scripts

If you omit the shebang and run:

```
./script.sh
```

It may fail if the default shell is not compatible.

If you run it as:

`script.sh`

Then will be used regardless of the shebang.

Interview Tip: Mention portability and how omitting the shebang can cause subtle bugs across systems.

15. How do you create, delete, and manage users in Linux?

Answer:

Managing users is a core Linux sysadmin task. Here's how it's done:

Create a user:

```
useradd john
passwd john
```

Or:

```
adduser john    # More interactive, distro-dependent
```

Creates:

- `/home/john` (unless disabled)
- Entry in `/etc/passwd`, `/etc/shadow`
- Default shell and UID/GID

Set user shell:

```
usermod -s /bin/ john
```

Add to group:

```
usermod -aG sudo john   # Add to sudo group
```

Delete a user:

```
userdel john             # Deletes user
userdel -r john          # Deletes user and home directory
```

Lock/unlock user:

```
passwd -l john    # Lock
passwd -u john    # Unlock
```

Check user details:

```
id john
getent passwd john
```

Files involved:

- `/etc/passwd`: Basic user info
- `/etc/shadow`: Password hashes
- `/etc/group`: Group info
- `/etc/login.defs`: Default user configs

Real-World Tip: In production, you'd automate user creation with scripts, Ansible, or LDAP-based provisioning.

16. How do you create, delete, and manage users in Linux?

Answer:

We touched on the basics earlier, but let's go deeper into managing Linux users, especially from a **production and troubleshooting perspective**.

Create a user with custom options:

```
useradd -m -s /bin/ -G developers -c "John Dev" john
```

- `-m`: create a home directory (`/home/john`)
- `-s`: assign shell
- `-G`: add to supplementary group(s)
- `-c`: comment or full name (used by `finger` or GUI tools)

Set password:

```
passwd john
```

Force password reset at next login:

```
chage -d 0 john
```

View user info:

```
id john
```

```
getent passwd john
```

Delete a user and their home directory:

```
userdel -r john
```

Pro Tip:
In environments with LDAP or Active Directory integration, users may not exist in `/etc/passwd` but can still be seen via `getent passwd username`.

17. Explain how groups work in Linux. What's the difference between primary and secondary groups?

Answer:

In Linux, **groups** are used to manage permissions collectively.

Primary Group:

- Defined in `/etc/passwd`.
- When a user creates a file, it belongs to their **primary group** by default.
- Only one primary group per user.

Secondary Groups:

- Additional groups a user can belong to.
- Stored in `/etc/group`.
- A user can be part of multiple secondary groups for access to different resources.

Check groups:

```
id john
```

```
groups john
```

Add user to secondary group:

```
usermod -aG devops john
```

Note: Use `-a` (append) or it will overwrite existing group membership.

Change primary group:

```
usermod -g developers john
```

Real-world use case:
You create a group `docker`, add users to it, and then only users in that group can run Docker commands without sudo.

18. How to set password aging policies and enforce them in Linux?

Answer:

Password aging ensures users rotate their passwords periodically, which is a **security best practice**.

Use `chage` to set aging rules:

```
chage -M 90 -m 7 -W 10 john
```

- `-M 90`: maximum days before password must be changed
- `-m 7`: minimum days before it can be changed again
- `-W 10`: warn user 10 days before expiration

View password policy:

`chage -l john`

Set global defaults:

In `/etc/login.defs`:

nginx

`PASS_MAX_DAYS   90`

`PASS_MIN_DAYS   7`

`PASS_WARN_AGE   10`

Force password expiration immediately:

`chage -d 0 john`

This forces the user to reset the password at next login.

Audit tip:
Check for accounts with `never expire` status — those are potential vulnerabilities.

19. What is `/etc/passwd`, `/etc/shadow`, and `/etc/group`?

Answer:

These files are essential to user and group management.

`/etc/passwd`:

- Stores basic user info.
- Format:

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

- Example:

```
john:x:1001:1001:John Dev:/home/john:/bin/
```

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

`/etc/shadow`:

- Stores password hashes and aging policies.
- Only readable by root.
- Format:

```
username:$6$hashedPassword:lastChange:min:max:warn:inactive:expire
```

`/etc/group`:

- Defines groups and their members.
- Format:

makefile

```
groupname:x:GID:member1,member2
```

Important:
If `/etc/passwd` is misconfigured, you may not be able to login or escalate privileges. Always back it up before editing.

20. How do you manage sudo access for a user in Linux?

Answer:

Sudo allows users to run commands as root or another user, without giving away the root password.

Grant sudo access:

1. Add the user to the `sudo` group (Debian/Ubuntu):

```
usermod -aG sudo john
```

For RHEL/CentOS, the group is often `wheel`:

```
usermod -aG wheel john
```

2. **Verify access:**

```
sudo -l -U john
```

3. **Edit sudoers safely:**

```
visudo
```

This prevents syntax errors which can lock you out.

Add a custom rule:

pgsql

```
john ALL=(ALL) NOPASSWD: /bin/systemctl restart nginx
```

This allows John to restart nginx without a password.

Use sudo:

```
sudo command
```

Real-world tips:

- Use **NOPASSWD** only for automation scripts, not humans.

- Restrict access to only required commands for least privilege.
- Use `/etc/sudoers.d/` to maintain per-user or per-role rules.

21. Explain `ps`, `top`, `htop`, `nice`, `renice`, and `kill` commands.

Answer:

These are core utilities for process monitoring and management.

`ps` (Process Status)

Used to list running processes.

```
ps aux
```

- `a`: all users
- `u`: user-oriented format
- `x`: include processes without a terminal

You can also filter by PID:

```
ps -p 1234 -o pid,ppid,cmd,%mem,%cpu
```

Real-time system monitoring.

Shows CPU/memory usage, uptime, load average, process details. Key shortcuts:

- `k`: kill process
- `r`: renice process
- `P`: sort by CPU

- `M`: sort by memory

An enhanced, interactive version of `top`.

- Scrollable, user-friendly interface
- Tree view of processes
- Easily renice/kill via function keys

Install via:

```
sudo apt install htop      # Debian/Ubuntu
sudo yum install htop    # RHEL/CentOS
```

`nice` and `renice`

Control the scheduling priority of a process.

- `nice` starts a process with a given priority (default is 0, range is -20 to 19).

```
nice -n 10 ./my_script.sh
```

- `renice` changes priority of an existing process.

```
renice -n -5 -p 1234
```

- Negative value = higher priority
- Only root can set negative priorities

`kill`

Sends signals to processes.

```
kill -9 1234       # Forcefully terminate
kill -15 1234      # Gracefully stop
```

Other related commands:

- `pkill nginx`: kill by name
- `killall firefox`: kill all processes with exact name

Interview Tip: Always try `SIGTERM (-15)` before `SIGKILL (-9)` to allow cleanup.

22. What is the difference between foreground and background processes?

Answer:

In Linux, a **foreground process** runs attached to the terminal, whereas a **background process** runs independently.

Foreground process:

Runs interactively.

```
ping google.com
```

It locks your terminal until stopped.

Stop it with:

- `Ctrl+C`: terminate
- `Ctrl+Z`: pause/suspend

Background process:

Runs without blocking the terminal.

Start directly:

```
./long_script.sh &
```

Or resume a suspended process:

```
bg
```

Check background jobs:

```
jobs
```

Bring to foreground:

```
fg %1
```

Kill a background process:

```
kill %1
```

Use case:
Run long-running scripts in the background so you can continue working in the terminal.

23. How do you trace zombie and orphan processes in Linux?

Answer:

These are special process states you must be aware of in production environments.

Zombie Process:

- The process has completed but still has an entry in the process table.
- It's waiting for the parent to read its exit status.
- Consumes a PID but no resources.

Identify them:

`ps aux | grep 'Z'`

Or:

`ps -el | grep Z`

If persistent:

- Identify the parent PID (`PPID`)
- Restart the parent process if it's not collecting exit status.

Orphan Process:

- A process whose parent has died.
- It's adopted by `init` (PID 1) or `systemd`.

They usually don't cause harm, but can indicate poor application behavior.

View:

`ps -eo pid,ppid,cmd | grep -w 1`

Look for unusual processes with PPID = 1.

Prevention:
Use `wait()` or `waitpid()` in scripts to collect child exit statuses and avoid zombies.

24. What is `strace` and how do you debug using it?

Answer:

`strace` traces **system calls and signals** of a running process.

This is one of the most powerful tools for **debugging mysterious behaviors**, especially when a binary crashes or hangs.

Common usage:

- Trace a command:

```
strace -o output.txt ./my_app
```

- Attach to a running PID:

```
strace -p 1234
```

- Trace specific syscalls:

```
strace -e openat,read,write ./my_script.sh
```

- Trace network activity:

```
strace -e trace=network curl google.com
```

Real-world examples:

- Find missing config files:

```
strace ./app 2>&1 | grep ENOENT
```

- See where a process is hanging (e.g., waiting on a file, stuck on `futex`)

Caution: `strace` is resource-intensive; don't run it lightly in production on critical processes.

25. How does the `cron` scheduler work? How to schedule jobs with examples?

Answer:

`cron` is a daemon used to schedule recurring tasks at specific times/dates.

Crontab format:

sql

Edit current user's cron jobs:

List them:

Sample entries:

- Run every day at 1 AM:

```
* * * /home/user/backup.sh
*   *   *   *   * command_to_execute

|   |   |   |   |

|   |   |   |   └── Day of week (0-7, Sun=0 or 7)

|   |   |   └──── Month (1-12)

|   |   └────── Day of month (1-31)

|   └──────── Hour (0-23)

└────────── Minute (0-59)

crontab -e

crontab -l

0 1
```

- Every 15 minutes:

```
*/15 * * * * /home/user/check_status.sh
```

- Every Monday at 10 AM:

```
0 10 * * 1 /home/user/report.sh
```

Best practices:

- Redirect output to logs:

```
* * * * * /script.sh >> /var/log/script.log 2>&1
```

- Always use **absolute paths** for scripts and executables.
- Set up environment variables if the script relies on them:

```
SHELL=/bin/

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

26. How to create and manage partitions using `fdisk` or `parted` in Linux?

Answer:

Partitioning divides a physical disk into logical sections. Two of the most common tools are:

Using `fdisk` (for MBR disks):

```
sudo fdisk /dev/sdX
```

Common steps inside `fdisk`:

- `m` – help
- `p` – print partition table

- `n` – create new partition
- `d` – delete partition
- `t` – change partition type
- `w` – write changes
- `q` – quit without saving

After creating a new partition (`/dev/sdX1`), format it:

```
sudo mkfs.ext4 /dev/sdX1
```

Mount it:

```
sudo mount /dev/sdX1 /mnt/mydata
```

Make it persistent in `/etc/fstab`:

```
/dev/sdX1  /mnt/mydata  ext4  defaults  0  2
```

Using `parted`:

```
sudo parted /dev/sdX
```

Commands:

- `mklabel`
- `mkpart primary ext4 0% 100%` – create partition
- `print` – show partitions
- `quit` – exit

Pro Tip: After any partitioning changes, always run: `sudo partprobe`

To inform the kernel of partition table changes.

27. Explain LVM (Logical Volume Manager) in detail.

Answer:

LVM provides flexibility and scalability in managing disk storage. It allows dynamic resizing and combining multiple physical volumes.

LVM hierarchy:

- **PV (Physical Volume):** actual disk or partition
- **VG (Volume Group):** pool of storage made of PVs
- **LV (Logical Volume):** like a partition, carved from a VG

Setup Example:

1. **Create PVs:**

```
pvcreate /dev/sdb1 /dev/sdc1
```

2. **Create VG:**

```
vgcreate myvg /dev/sdb1 /dev/sdc1
```

3. **Create LV:**

```
lvcreate -L 10G -n mydata myvg
```

4. **Format and mount:**

```
mkfs.ext4 /dev/myvg/mydata
```

```
mount /dev/myvg/mydata /mnt/lvmdata
```

Resize LV:

- To extend:

```
lvextend -L +5G /dev/myvg/mydata
```

```
resize2fs /dev/myvg/mydata
```

- To reduce:

```
umount /mnt/lvmdata
```

```
e2fsck -f /dev/myvg/mydata
```

```
resize2fs /dev/myvg/mydata 10G
```

```
lvreduce -L 10G /dev/myvg/mydata
```

```
mount /dev/myvg/mydata /mnt/lvmdata
```

Warning: Reducing an LV is risky. Always back up data.

Advantages:

- Resize volumes live
- Snapshot support
- Use multiple disks as a single storage unit

28. How to mount and unmount filesystems in Linux?

Answer:

Mounting makes a filesystem accessible in the Linux directory tree.

Mount a disk:

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

You can verify with:

```
df -h
```

```
mount | grep /mnt/mydata
```

Unmount:

```
sudo umount /mnt/mydata
```

Important: Unmount before formatting or removing devices.

Persistent Mounting via `/etc/fstab`:

Add:

ini

```
UUID=abcd-1234  /mnt/mydata  ext4  defaults  0  2
```

Find UUID:

```
blkid /dev/sdb1
```

Use mount options like `noatime`, `rw`, `ro`, `errors=remount-ro` to optimize behavior.

29. How do you monitor disk space usage with `df`, `du`, and `ncdu`?

Answer:

Monitoring disk usage is key to system reliability.

`df` – Disk Free

Shows filesystem-level usage.

```
df -h
```

- `-h`: human-readable (MB/GB)
- `-i`: show inode usage

`du` – Disk Usage

Shows space used by files/directories.

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

- `-s`: summary
- `-h`: human-readable
- `--max-depth=1`: useful for large folders

`ncdu` – Interactive disk usage viewer

More user-friendly, excellent for finding large files.

```
sudo ncdu /
```

Install via:

```
sudo apt install ncdu
```

```
sudo yum install ncdu
```

Real-world troubleshooting example:

- If `/` fills up, use:

```
du -shx /* 2>/dev/null | sort -h
```

To locate the biggest directories.

- Check inodes:

```
df -i
```

If inodes are full, clean up many small files (e.g., mail queues, cache).

30. What is the difference between ext3, ext4, xfs, and btrfs?

Answer:

These are different Linux **filesystems**, each with trade-offs in performance, features, and reliability.

- Journaled version of ext2
- Stable but outdated
- Max file size: 2TB
- Lacks modern features like extents and delayed allocation
- Successor to ext3
- Faster performance, extents support
- Delayed allocation, reduced fragmentation

- Max file size: 16TB
- Default on most distros
- High-performance journaling FS
- Great for large files, high concurrency
- Fast at streaming data, logs
- No shrinking support
- Common in RHEL/CentOS
- Advanced features: snapshots, checksums, RAID
- Copy-on-write (CoW) system
- Still maturing in terms of stability (especially under RAID)
- Great for backups, container workloads

Use cases:

- `ext4`: general-purpose, default, well-tested
- `xfs`: high-performance, large data, logs
- `btrfs`: snapshotting, backups, container-heavy environments

31. Explain how IP addressing and subnetting work in Linux.

Answer:

IP addressing in Linux follows the same principles as general networking but is managed through specific commands and config files.

IP Addressing Basics:

- **IPv4**: 32-bit, e.g., `192.168.1.10`
- **Subnet mask**: defines the network portion. Example: `/24` = `255.255.255.0`
- **CIDR notation**: combines IP and mask: `192.168.1.10/24`

Check current IP:

```
ip addr show
```

or

```
ip a
```

Assign IP temporarily:

```
sudo ip addr add 192.168.1.100/24 dev eth0
sudo ip link set dev eth0 up
```

Assign IP permanently:

- On Debian/Ubuntu: Edit `/etc/netplan/*.yaml` or `/etc/network/interfaces`
- On RHEL/CentOS: Edit `/etc/sysconfig/network-scripts/ifcfg-eth0`

Subnetting Example:

If you have a `/29` subnet:

- IPs: `192.168.1.0` to `192.168.1.7`
- Network: `192.168.1.0`

- Broadcast: `192.168.1.7`
- Usable IPs: `192.168.1.1 - 192.168.1.6`

Command-line subnet calculator:

```
ipcalc 192.168.1.10/29
```

32. How do you check and configure network interfaces in Linux?

Answer:

Modern Linux systems use the `ip` command (preferred over `ifconfig`).

View interfaces:

```
ip link
ip addr
```

Bring interface up/down:

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

Assign static IP:

```
sudo ip addr add 192.168.0.100/24 dev eth0
sudo ip route add default via 192.168.0.1
```

On older systems:

```
ifconfig eth0 192.168.0.100 netmask 255.255.255.0 up
route add default gw 192.168.0.1
```

Persistent configuration:

Debian/Ubuntu (Netplan):

yaml

```yaml
network:
  version: 2
  ethernets:
    eth0:
      dhcp4: no
      addresses: [192.168.0.100/24]
      gateway4: 192.168.0.1
      nameservers:
        addresses: [8.8.8.8,8.8.4.4]
```

RHEL/CentOS (ifcfg):

ini

```ini
BOOTPROTO=static
IPADDR=192.168.0.100
NETMASK=255.255.255.0
GATEWAY=192.168.0.1
DNS1=8.8.8.8
```

Restart network service:

```
sudo systemctl restart NetworkManager
```

33. Explain `netstat`, `ss`, `ip`, and `ifconfig` commands.

Answer:

These tools provide network interface, socket, and connection details.

`netstat` ***(older, still used)*:**

```
netstat -tulnp
```

- `-t`: TCP
- `-u`: UDP
- `-l`: listening
- `-n`: numeric IPs
- `-p`: process info

List open ports:

```
netstat -plnt
```

`ss` ***(replacement for netstat, faster)*:**

```
ss -tulnp
```

Gives faster and more detailed socket statistics.

`ip` ***(modern replacement for ifconfig & route)*:**

```
ip addr
ip route
ip link
```

`ifconfig` ***(deprecated, but still present on many systems)***:

```
ifconfig -a
```

Tip: `ip` and `ss` are faster, script-friendly, and preferred for scripting and automation.

34. How do you troubleshoot network issues using `ping`, `traceroute`, `nmap`, and `tcpdump`?

Answer:

These tools help isolate and diagnose connectivity, routing, port issues, and firewall blocks.

`ping`:

Tests reachability.

```
ping 8.8.8.8
ping -c 4 google.com
```

If DNS fails but IP pings work, it's a **DNS issue**.

`traceroute`:

Tracks the path of packets to destination.

```
traceroute google.com
```

Helps detect where the connection breaks (e.g., ISP routing issue).

`nmap`:

Scans open ports and services.

```
nmap -sS -p 1-1000 192.168.0.10
```

- `-sS`: stealth scan
- `-p`: port range

Useful for firewall audits and verifying if a service is accessible.

`tcpdump`:

Packet sniffer for deep network debugging.

```
sudo tcpdump -i eth0 port 80
```

Capture packets to file:

```
sudo tcpdump -i eth0 -w capture.pcap
```

Analyze in Wireshark.

Real-world flow:

1. `ping` – check reachability
2. `traceroute` – check path
3. `nmap` – check port openness
4. `tcpdump` – capture traffic if packets are being dropped or misrouted

35. How does Linux firewall (`iptables` and `firewalld`) work?

Answer:

Linux uses **iptables** (and its frontend `firewalld`) to control incoming/outgoing traffic.

`iptables` basics:

It's a rule-based system to filter packets. Chains:

- `INPUT`: packets coming into the server
- `OUTPUT`: packets going out
- `FORWARD`: packets routed through the system

List rules:

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

Allow SSH:

```
sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

Block an IP:

```
sudo iptables -A INPUT -s 192.168.1.100 -j DROP
```

Flush all rules:

```
sudo iptables -F
```

Save rules:

```
sudo iptables-save > /etc/iptables.rules
```

`firewalld` (modern systemd-based interface):

Zone-based abstraction layer over iptables.

Start and enable:

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

Check status:

```
sudo firewall-cmd --state
```

Allow a service:

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

sudo firewall-cmd --reload
```

Add port:

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

Key Differences:

- `iptables`: rule-based, detailed, manual
- `firewalld`: zone-based, easier to use, supports runtime/permanent configs

36. How to monitor system performance with `top`, `vmstat`, `iostat`, `sar`, and `dstat`?

Answer:

Each of these tools provides a different angle on system performance: CPU, memory, disk I/O, processes, and more.

`top`

Real-time process and system resource monitor.

```
top
```

- `%CPU` – CPU usage
- `%MEM` – memory usage
- `load average` – system load over 1, 5, 15 minutes
- Use `P`, `M`, `T` to sort by CPU, memory, time

`vmstat` (Virtual Memory Statistics)

```
vmstat 2 5
```

- Reports memory, CPU, swap, I/O
- Key columns:
  - `r`: run queue (waiting for CPU)
  - `si/so`: swap in/out
  - `wa`: IO wait time

High `wa` value? Likely a disk bottleneck.

`iostat` (I/O statistics)

```
iostat -xz 2 5
```

- Shows per-device I/O stats
- Important metrics:
  - `%util`: device utilization (if near 100%, it's a bottleneck)
  - `await`: average time for I/O requests
  - `tps`: transfers per second

Install via:

```
sudo apt install sysstat
```

`sar` (System Activity Reporter)

```
sar -u 1 5          # CPU usage
sar -r 1 5          # Memory
sar -n DEV 1 5      # Network
```

View historical data:

```
sar -u -f /var/log/sysstat/sa10
```

`dstat` (All-in-one performance viewer)

```
dstat -cdngy
```

- Shows CPU, disk, net, paging in one view
- Helpful for real-time multi-resource monitoring

Install:

```
sudo apt install dstat
```

37. What is load average? How to interpret it?

Answer:

Load average represents the average number of processes waiting for CPU over time.

Shown in tools like `top`, `uptime`, and `w`. Example:

```
load average: 1.20, 0.80, 0.50
```

- 1.20 = average over last 1 minute
- 0.80 = over last 5 mins
- 0.50 = over last 15 mins

How to interpret it:

- If your system has **1 core**: a load of 1.00 means full utilization.
- For a **4-core CPU**, load average of 4.00 = 100% usage.
- **More than number of cores?** Processes are queuing — system is overloaded.

Key Tip: Load average includes processes in the **runnable and uninterruptible** states (e.g., waiting for CPU or disk I/O).

38. How to monitor memory usage in real-time?

Answer:

Memory monitoring involves checking **RAM**, **swap**, and **cache** usage.

`free` command:

```
free -h
```

Columns:

- `total`, `used`, `free`
- `buff/cache`: memory used by buffers and file cache
- `available`: truly usable memory

Important: `used` may appear high, but Linux aggressively caches. Always check `available` for real usage.

`top` memory view:

Press `M` to sort by memory usage.

`vmstat`:

```
vmstat 1 5
```

Fields:

- `free`: actual free RAM
- `si/so`: swap in/out

`smem`:

Displays per-process memory usage including shared memory. Install:

```
sudo apt install smem
smem -r
```

Check swap usage:

```
swapon --show
```

If swap is constantly used and memory is free, system may be misconfigured or have a memory leak.

39. How to analyze logs and system events in Linux?

Answer:

Linux stores logs primarily in `/var/log`, and `journalctl` is the modern logging tool for systemd-based distros.

Traditional logs:

```
/var/log/messages         # System logs (RHEL/CentOS)
/var/log/syslog           # General logs (Debian/Ubuntu)
/var/log/auth.log         # Authentication events
/var/log/kern.log         # Kernel messages
/var/log/dmesg            # Boot & hardware logs
/var/log/secure           # Security events (RHEL)
```

Use `tail` to watch logs live:

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

Search logs:

```
grep "error" /var/log/syslog
```

`journalctl` **(for systemd systems):**

```
journalctl                  # Show all logs
journalctl -u nginx         # Logs for a service
journalctl -xe              # Show last critical logs
journalctl --since "2 hours ago"
```

Logs are stored in binary under `/var/log/journal`.

40. What are cgroups and how do they help in resource management?

Answer:

cgroups (control groups) are a Linux kernel feature that **limits, accounts for, and isolates** resource usage (CPU, memory, I/O, etc.) of a set of processes.

Key use cases:

- Prevent one process from hogging all memory or CPU
- Enforce memory limits per container (used by Docker/Kubernetes)
- Isolate workloads in multi-tenant environments

Common cgroup controllers:

- `cpu` – limit CPU usage
- `memory` – set RAM limits
- `blkio` – limit disk I/O

- `net_cls` – classify network traffic

Check current cgroup assignment:

```
cat /proc/self/cgroup
```

To manually create a cgroup (v1 example):

```
sudo cgcreate -g memory,cpu:/mygroup

sudo cgset -r memory.limit_in_bytes=500M mygroup sudo

cgset -r cpu.shares=512 mygroup

sudo cgexec -g memory,cpu:mygroup ./heavy_script.sh
```

In production:

- Kubernetes uses cgroups under the hood to enforce **resource limits**.
- Docker also uses cgroups to restrict container-level resource usage.

Pro Tip: Understanding cgroups is crucial when debugging containers that crash due to **OOMKilled** (Out Of Memory).

41. How does package management work in Debian vs RedHat-based systems?

In Linux, package managers automate installing, updating, configuring, and removing software. The major difference comes down to the type of package format and package manager each system uses.

Debian-based systems (like Ubuntu) use `.deb` packages and the `dpkg` tool under the hood. For ease, they use `apt` or `apt-get`.

Example:

```
sudo apt update
```

```
sudo apt install nginx
```

`apt` handles dependencies automatically. It pulls packages from `/etc/apt/sources.list` or files under `/etc/apt/sources.list.d/`.

RedHat-based systems (like CentOS, RHEL, Fedora) use `.rpm` packages and `rpm` as the backend. The high-level tool is `yum` or `dnf` (on newer systems).

Example:

```
sudo yum install nginx
```

or:

```
sudo dnf install nginx
```

The core difference:

- Debian: `.deb` + `dpkg` + `apt`
- RedHat: `.rpm` + `rpm` + `yum` or `dnf`

Dependency handling is better with apt/yum/dnf than `dpkg` or `rpm` alone.

42. What are the differences between `apt`, `yum`, `dnf`, and `zypper`?

All are package managers, but they belong to different families.

`apt` is for Debian/Ubuntu. It's fast, straightforward, and widely used in cloud environments. You can do updates, upgrades, and install/remove packages with it.

yum is the older tool for RHEL-based systems. It handles repositories and dependencies well but has been replaced by `dnf`.

dnf (Dandified Yum) is the next-gen replacement for `yum`. It's faster, supports parallel downloads, better dependency resolution, and has a cleaner architecture.

zypper is the package manager for SUSE/openSUSE systems. It's powerful and similar in functionality to `yum/dnf` with good repo management and transactional updates.

In practice:

- Use `apt` on Ubuntu
- Use `dnf` on RHEL/CentOS 8 and above
- Use `yum` on older CentOS
- Use `zypper` on SUSE

43. How to build and install packages from source?

Sometimes, the latest version of a tool isn't available through a package manager, or you need to compile it with custom options. That's when building from source is necessary.

Here's the typical flow:

1. **Install build tools:**

```
sudo apt install build-essential     # Debian

sudo yum groupinstall "Development Tools"    # RHEL
```

2. **Download the source code:**

```
wget https://example.com/tool.tar.gz
tar -xvzf tool.tar.gz
cd tool
```

3. **Configure the build:**

```
./configure
```

This checks for dependencies and prepares the Makefile.

4. **Compile:**

```
make
```

5. **Install:**

```
sudo make install
```

This installs binaries under `/usr/local/bin` or `/usr/bin` by default.

You can also use:

```
checkinstall
```

To create a `.deb` or `.rpm` file for easy uninstallation later.

This method is especially common when dealing with bleeding-edge tools like Nginx modules, Redis, or Git from source.

44. What is a `.deb` and `.rpm` file? How are they created?

Both `.deb` and `.rpm` files are precompiled software packages used by their respective systems.

- A `.deb` **file** is for Debian/Ubuntu
- A `.rpm` **file** is for RHEL/CentOS/Fedora

They contain:

- Executable binaries
- Configuration files
- Metadata (version, dependencies, scripts)

To create a `.deb` package:

1. Create a directory structure:

```
myapp/
├── DEBIAN/
│   └── control
├── usr/
│   └── local/
│       └── bin/
│           └── myapp
```

2. Add a `control` file under `DEBIAN/` with metadata:

```
Package: myapp Version:
  1.0 Architecture:
  amd64 Maintainer:
  Your Name
  Description: My custom app
```

3. Build the package:

```
dpkg-deb --build myapp
```

To create an `.rpm` file, use the `rpmbuild` tool and an `.spec` file with build and install scripts.

These files are great for enterprise deployments, CI pipelines, and distributing custom apps.

45. How do you handle dependency issues in Linux?

Dependency issues usually occur when required libraries or packages are missing, conflicting, or incompatible. They often show up as "dependency hell."

For Debian/Ubuntu: `apt` handles dependencies pretty well, but if you use `dpkg` directly and it complains: `sudo`

```
dpkg -i package.deb sudo
apt-get install -f
```

The second command fixes missing dependencies.

For RHEL/CentOS: If using `rpm`:

```
sudo rpm -ivh package.rpm
```

If it fails:

```
sudo yum deplist package-name
```

Or use:

```
yum install ./package.rpm
```

This resolves dependencies automatically.

Other tools:

- `ldd binary` – shows which shared libraries are needed and if any are missing.
- `strace` – can help find dynamic loading failures.
- `alien` – converts between `.rpm` and `.deb` if needed, but use with caution.

Best practice: Always prefer package managers over manually downloading binaries or using `dpkg`/`rpm` directly unless you know what you're doing.

46. What is `systemd` and how is it different from `init`?

`systemd` is the modern system and service manager used in most major

Linux distributions today. It's the replacement for the traditional `SysVinit` and `Upstart` systems.

The core difference is that `init` works sequentially—starting one service at a time based on scripts in `/etc/init.d/`, while `systemd` uses parallel execution and unit files for better performance and dependency handling.

With `init`, services are started with numbered scripts (`S01service`, `S02next`) that run in order. It's slow and not great at tracking dependencies.

`systemd`, on the other hand:

- Starts services in parallel when possible
- Uses unit files (`*.service`, `*.target`, etc.)
- Tracks dependencies between services
- Has built-in logging (`journalctl`)
- Handles sockets, timers, devices, and more—not just daemons

You can check if your system uses systemd by running:

```
ps -p 1 -o comm=
```

If it returns `systemd`, then you're using it.

47. How do you manage services using `systemctl`?

`systemctl` is the CLI tool used to control systemd

services. Here's what you can do with it:

Start a service:

```
sudo systemctl start nginx
```

Stop a service:

```
sudo systemctl stop nginx
```

Restart it:

```
sudo systemctl restart nginx
```

Enable on boot:

```
sudo systemctl enable nginx
```

Disable autostart:

```
sudo systemctl disable nginx
```

Check status:

```
sudo systemctl status nginx
```

View all active services:

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

Check if a service failed:

```
systemctl --failed
```

This command is especially useful after a reboot or service crash.

48. How to create a custom service in systemd?

Let’s say you have a custom script or application you want to run as a managed service.

Here's how you create a custom service:

1. Create a systemd unit file:

```
sudo nano /etc/systemd/system/myapp.service
```

2. Add the following content:

ini

```
[Unit]

Description=My Custom App

After=network.target

[Service]

ExecStart=/usr/local/bin/myapp.sh

Restart=on-failure

User=myuser

[Install]

WantedBy=multi-user.target
```

3. Reload systemd to recognize it:

```
sudo systemctl daemon-reexec

sudo systemctl daemon-reload
```

4. Start and enable your service:

```
sudo systemctl start myapp
sudo systemctl enable myapp
```

5. Check status:

```
sudo systemctl status myapp
```

This is how you “daemonize” your own scripts and run them reliably like a native Linux service.

49. How do you troubleshoot boot failures in Linux?

Boot failures are critical, and knowing how to troubleshoot them is a key sysadmin skill.

Step-by-step approach:

1. **Watch for kernel panic or GRUB errors** during boot. If you see something like “grub rescue>”, then GRUB is corrupted.

You may need to boot from a Live CD/USB and repair with:

```
grub-install /dev/sda
update-grub
```

2. **Enter rescue mode** or **emergency mode** if systemd can’t continue.
    - Rescue mode: system boots to single-user shell with networking.
    - Emergency mode: bare minimum shell, no services.

- Use `systemctl default` to return to normal if fixed.

Use `journalctl` to view logs:

```
journalctl -xb
```

3. This shows boot logs and error traces.

Check for failing services:

```
systemctl list-units --failed
```

4. **Look for filesystem issues:** If `/etc/fstab` has a wrong entry or a device is missing, it can stall the boot. You may need to comment out broken lines and reboot.

5. **Use `dracut`, `fsck`, or `chroot`** depending on the failure depth.

This process often involves booting into recovery mode and applying fixes from there.

50. Explain `journalctl` and how it helps in log management with `systemd`.

`journalctl` is the logging command for systemd-based systems. It replaces traditional text log files with a binary logging system.

Here's how you use it:

View all logs:

```
journalctl
```

View recent logs:

```
journalctl -xe
```

Logs for a specific service:

```
journalctl -u nginx.service
```

Logs since a specific time:

```
journalctl --since "2024-12-01 12:00"
```

Follow logs in real-time (like `tail -f`):

```
journalctl -f
```

To view previous boots:

```
journalctl --list-boots

journalctl -b -1
```

Log rotation and size can be controlled via `/etc/systemd/journald.conf`. Logs are stored in binary under `/var/log/journal/`.

You can also export logs:

```
journalctl > logs.txt
```

Key benefit: Unlike traditional logs, systemd journals include rich metadata like process ID, user ID, session ID, boot ID, and can be filtered in powerful ways.

51. What are the best practices to secure a Linux system?

Securing a Linux system is about reducing the attack surface, enforcing least privilege, and ensuring all activity is monitored.

Start with basic but critical hardening steps:

- Always **disable root SSH access**. Instead, use `sudo` with individual accounts.
- Enforce **SSH key-based authentication** instead of passwords.
- Set up a **firewall** using `iptables`, `ufw`, or `firewalld` to allow only necessary ports.
- Keep the system updated regularly using `apt update && apt upgrade` or `yum update`.
- Remove **unused services** and daemons that might expose ports or vulnerabilities.
- Install **fail2ban** to protect against brute force login attempts.
- Use **auditd** to monitor file access and user activity.
- Set up **automatic logout** for inactive sessions using shell timeout (`TMOUT`).
- Set up strong **password policies** and enforce expiration with tools like `chage`.
- Limit use of `sudo` and audit `sudo` logs.

Security is not about one tool—it's about creating layers. Even if one layer fails (like a user getting access), the attacker shouldn't be able to do much.

52. How does SELinux or AppArmor work?

Both **SELinux** and **AppArmor** are **Mandatory Access Control (MAC)** frameworks that go beyond traditional user/group permissions. They enforce security policies at a much more granular level.

SELinux (Security-Enhanced Linux) is based on labeling:

- Every file, process, and port gets a label.
- The system checks these labels against security policies to determine access.
- Modes:
  - `Enforcing`: policy is enforced
  - `Permissive`: policy violations are only logged
  - `Disabled`: no policy enforcement

Check mode:

```
getenforce
```

Switch mode:

```
setenforce 0     # Permissive
setenforce 1     # Enforcing
```

View file context:

```
ls -Z
```

AppArmor, on the other hand, works with profiles:

- Profiles are path-based rather than label-based.
- Easier to understand for beginners.
- Used in Ubuntu and SUSE systems by default.

Check AppArmor status:

```
sudo aa-status
```

Set a profile to enforce or complain mode:

```
sudo aa-enforce /etc/apparmor.d/usr.sbin.nginx

sudo aa-complain /etc/apparmor.d/usr.sbin.nginx
```

In real-world use, **SELinux is powerful but complex**. If you're on RHEL or CentOS, you're using SELinux by default and should know how to manage its logs (`/var/log/audit/audit.log`) to troubleshoot access denials.

53. How to set up firewall rules using `iptables` or `ufw`?

Firewalls are the first line of defense in a Linux server. You should only allow what's needed and block everything else by default.

With `iptables`, you can do things like:

Allow SSH:

```
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
```

Allow HTTP and HTTPS:

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT

iptables -A INPUT -p tcp --dport 443 -j ACCEPT
```

Drop all other traffic:

```
iptables -P INPUT DROP

iptables -P FORWARD DROP

iptables -P OUTPUT ACCEPT
```

Save your config so it persists:

```
iptables-save > /etc/iptables.rules
```

But for simplicity, many use **ufw** (Uncomplicated Firewall), especially on Ubuntu:

Enable it:

```
sudo ufw enable
```

Allow services:

```
sudo ufw allow ssh

sudo ufw allow 80/tcp

sudo ufw allow 443
```

Deny something:

```
sudo ufw deny 23
```

Check status:

```
sudo ufw status verbose
```

In production environments, consider using `firewalld` or managing rules via `iptables-persistent` or Ansible for better scalability.

54. How do you audit logs and file integrity?

Auditing logs is crucial for tracking unauthorized activity, data tampering, or suspicious behavior.

Start with the **auditd** service. It records events like file access, permission changes, and system calls.

Install and start auditd:

```
sudo apt install auditd

sudo systemctl enable --now auditd
```

To monitor a file:

```
auditctl -w /etc/passwd -p wa -k passwd_watch
```

- `-w`: watch file
- `-p`: permissions to monitor (write, attribute)
- `-k`: keyword to identify log entries

Check logs:

```
ausearch -k passwd_watch
```

To monitor file integrity, use **AIDE (Advanced Intrusion Detection Environment)**.

Install AIDE:

```
sudo apt install aide
```

Initialize database:

```
sudo aideinit
```

Run integrity checks:

```
sudo aide --check
```

This helps you know if any critical file was altered. It's especially useful for detecting rootkits or tampering after a breach.

55. What is SSH hardening? How do you secure remote access?

SSH is the most common attack vector on Linux servers, and hardening it is essential.

Here's how to do it:

1. **Disable root login**: Edit `/etc/ssh/sshd_config` and set:

nginx

```
PermitRootLogin no
```

2. **Use only key-based authentication**: Disable passwords by setting:

nginx

```
PasswordAuthentication no
```

3. **Use a non-standard port**: Change `Port 22` to something else to reduce brute-force attempts.
4. **Limit users who can SSH**: Add to `sshd_config`:

nginx

```
AllowUsers alice bob deploy
```

5. **Use Fail2Ban**: Installs filters to block IPs after failed login

attempts. Install it:

```
sudo apt install fail2ban
```

6. **Enable 2FA**: Use `google-authenticator` PAM module to enable time-based tokens.

7. **Use firewall rules to restrict SSH**: Allow only known IPs to access your SSH port.

8. **Audit login attempts**: Review logs:

```
journalctl -u ssh
```

Or check `/var/log/auth.log`

These measures make it much harder for attackers to get in, even if they know your IP or scan for open ports.

56. Advanced usage of `grep`, `sed`, `awk`, `cut`, and `tr`

These tools form the foundation of text processing in Linux.

grep is for searching patterns:

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

Case-insensitive search for "error".

Use regex:

```
grep -E "fail|error" /var/log/syslog
```

Search recursively:

```
grep -r "nginx" /etc/
```

Show lines before/after a match:

```
grep -A 3 -B 2 "failure" /var/log/app.log
```

sed is for stream editing: Replace a string:

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

Remove blank lines:

```
sed '/^$/d' file.txt
```

Delete lines matching a pattern:

```
sed '/^#/d' config.conf
```

Edit in place:

```
sed -i 's/http/https/g' urls.txt
```

awk is for field-based processing: Print second column:

```
awk '{print $2}' data.txt
```

Sum a column:

```
awk '{sum+=$3} END {print sum}' data.csv
```

Use field separator:

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

cut is simpler than awk, perfect for column slicing:

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

Get usernames.

tr is for translating or deleting characters:

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

Remove all digits:

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

These tools together let you transform data in files or pipelines without ever opening a GUI. In scripts, they're indispensable.

57. What is the difference between `find` and `locate`?

Both are used to search for files, but they work very differently.

find searches the filesystem **in real time**. It's accurate and powerful but slower.

Example:

```
find /etc -name "nginx.conf"
```

Search by type:

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

Find files modified in the last 24 hours:

```
find /var/log -mtime -1
```

Find large files:

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

Combine with exec:

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

locate uses a prebuilt database (`updatedb`) to search

quickly. Example:

```
locate nginx.conf
```

It's lightning fast, but may show outdated results if the file was created or deleted after the last index update.

Update the index manually:

```
sudo updatedb
```

Use `find` for precision and advanced logic. Use `locate` for speed when accuracy is less critical.

58. How to use `xargs` effectively?

`xargs` is used to take input from one command and pass it as arguments to another. It's especially useful when you need to process output line by line and feed it to commands like `rm`, `mv`, `cp`, or even curl and ssh.

Here's a classic use case with `find`:

```
find . -name "*.log" | xargs rm -f
```

This deletes all `.log` files. You could do the same without `xargs`, but `xargs` improves performance by batching the

input. If the filenames have spaces:

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

Another example with `xargs` and `mkdir`:

```
echo "dir1 dir2 dir3" | xargs -n 1 mkdir
```

This will create each directory listed in the string.

You can use it with `curl` to fetch URLs from a list:

```
cat urls.txt | xargs -n 1 curl -O
```

`xargs` is a force multiplier—it helps scale up one-liner scripts and pipelines quickly.

59. How to search for patterns recursively using `grep` and `find`?

You can search within directories and subdirectories with `grep -r` or a combination of `find` and `grep`.

Using recursive grep:

```
grep -r "password" /etc/
```

That searches all files in `/etc/` and below.

Want line numbers and filenames?

```
grep -rn "failed login" /var/log
```

Ignore case and binary files:

```
grep -rni --exclude="*.log" "error" .
```

With `find` and `grep` together:

```
find . -type f -name "*.conf" -exec grep "ssl" {} +
```

This approach gives more control—filter files with `find`, then search their content with `grep`.

Want to search all `.sh` files for a function name?

```
find . -type f -name "*.sh" | xargs grep "my_function"
```

When searching across codebases or config directories, recursive grep is your best friend.

60. Explain piping (|) and redirection (>, >>, 2>) with examples

Piping and redirection are how Linux glues together commands and handles outputs and errors.

The **pipe (|)** connects the output of one command to the input of another:

```
ps aux | grep nginx
```

You can build long chains:

```
cat access.log | grep "404" | sort | uniq -c | sort -nr
```

Redirection:

`>` writes output to a file (overwrite):

```
echo "hello" > hello.txt
```

`>>` appends to a file:

```
echo "world" >> hello.txt
```

`2>` redirects stderr (errors):

```
ls /no/such/dir 2> error.txt
```

`&>` redirects both stdout and stderr:

```
./build.sh &> build.log
```

You can also redirect stderr to stdout:

```
command 2>&1
```

A useful pattern:

```
some_command > output.log 2>&1
```

This logs everything—output and errors—to the same file.

Mastering redirection and pipes means you can script anything, debug anything, and write efficient command-line workflows.

61. What is the role of the Linux kernel?

The Linux kernel is the core component of the operating system. It acts as a bridge between the hardware and user applications. Everything that touches your CPU, memory, disk, or network goes through the kernel.

Here's what the kernel handles:

– **Process management**: Scheduling, creating, terminating, and context switching between processes
– **Memory management**: Allocating and freeing memory, managing virtual memory and paging
– **Device drivers**: Providing a layer for interacting with hardware like disks, keyboards, or network cards
– **System calls**: Interface for userspace programs to request kernel-level operations
– **Networking stack**: Manages communication over protocols like TCP/IP, UDP, ARP, and so on
– **Security**: Manages permissions, access control, namespaces, and cgroups

Without the kernel, your Linux system is just a bunch of files. It's what boots your machine, handles hardware, and enforces boundaries between user applications.

You can check your kernel version with:

```
uname -r
```

62. How do you view and manage kernel modules?

Kernel modules are pieces of code that can be loaded into the kernel as needed. They extend the kernel’s functionality without needing to reboot or recompile.

You can think of them like plug-ins—for example, a filesystem driver or a USB controller driver.

To list currently loaded modules:

```
lsmod
```

To load a module:

```
sudo modprobe <module_name>
```

To unload a module:

```
sudo modprobe -r <module_name>
```

To get information about a module:

```
modinfo <module_name>
```

To check what modules are loaded for a specific device:

```
lspci -k
```

You can also blacklist modules you don't want the system to load automatically. To do this, add a line like this in `/etc/modprobe.d/blacklist.conf`:

nginx

```
blacklist bluetooth
```

Kernel modules are essential for customizing behavior without bloating the base kernel. You only load what you need.

63. What is `dmesg` and how do you interpret its output?

`dmesg` stands for "diagnostic message." It displays messages from the kernel ring buffer—basically everything the kernel says during boot and while running.

It's especially useful for hardware-related debugging. When you plug in a USB, or when a device fails, the kernel logs it here.

Run it like this:

```
dmesg | less
```

If you just plugged in a USB and want to see its logs:

```
dmesg | tail
```

You can also filter:

```
dmesg | grep eth0
```

Or for disk detection:

```
dmesg | grep sd
```

During boot, `dmesg` shows messages like:

- Kernel version
- CPU initialization
- Memory detection
- Filesystem mounting
- Driver loading
- Network interfaces being initialized

It’s a vital tool for any hardware issue—disks not mounting, network cards not appearing, or USBs not being detected.

On newer systems, `journalctl -k` is an alternative, which gives kernel logs with timestamps.

64. How do you update the kernel safely?

Kernel updates are essential for patching vulnerabilities and adding hardware support, but they must be handled carefully to avoid breaking your boot process.

On Debian/Ubuntu:

```
sudo apt update

sudo apt install --install-recommends linux-generic
```

This pulls the latest stable kernel package.

On RHEL/CentOS:

```
sudo yum update kernel
```

To list available kernel versions:

```
rpm -q kernel
```

After updating the kernel, reboot the system:

```
sudo reboot
```

To confirm the system is using the new kernel:

```
uname -r
```

If the system fails to boot with the new kernel, you can select an older kernel from the GRUB menu during boot.

To keep multiple kernels:

- On Ubuntu: you can have multiple kernel images installed
- On RHEL: old kernels are usually retained unless you explicitly remove them

Always test kernel updates in staging or with snapshotting (if using LVM or Btrfs) before applying in production.

65. How do you compile a custom kernel?

Compiling your own kernel gives you control over features, modules, and performance optimizations. This is usually done in embedded systems, custom hardware setups, or when you want bleeding-edge features.

Here’s the typical workflow:

1. **Install dependencies**:

```
sudo apt install build-essential libncurses-dev bison
flex libssl-dev libelf-dev
```

2. **Download source code**:

```
wget
https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.x.
y.tar.xz

tar -xf linux-6.x.y.tar.xz

cd linux-6.x.y
```

3. **Configure the kernel**:

```
make menuconfig
```

You'll see a menu where you can include/exclude kernel features.

4. **Compile the kernel**:

```
make -j$(nproc)
```

5. **Install modules and kernel**:

```
sudo make modules_install

sudo make install
```

This places the kernel under `/boot`, updates GRUB, and prepares the initramfs.

6. **Update bootloader**: Usually done automatically, but you can force it:

```
sudo update-grub
```

7. **Reboot into the new kernel**:

```
sudo reboot
```

Compiling a kernel can take time and CPU. Always back up and know how to revert in case something breaks. This is not common in day-to-day admin work but highly relevant in kernel development, embedded systems, and advanced performance tuning.

66. Where are the major log files stored in Linux?

In Linux, most logs live under `/var/log`. This directory is where the system, kernel, services, and applications write their activity logs.
Understanding what lives where helps you pinpoint issues fast. Here

are the most important ones:

- `/var/log/syslog` or `/var/log/messages`: These are the general system logs. On Debian-based systems, it's `syslog`, and on RHEL-based systems, it's `messages`.
- `/var/log/auth.log` or `/var/log/secure`: Tracks authentication events like logins, sudo attempts, and SSH access.
- `/var/log/kern.log`: Contains kernel-level messages.
- `/var/log/dmesg`: Hardware-related boot logs.
- `/var/log/boot.log`: Summary of services during boot.
- `/var/log/faillog`: Records failed login attempts.

- `/var/log/wtmp`, `/var/log/btmp`, and `/var/log/lastlog`: Binary logs that store login history.

To inspect logs:

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

Or follow logs in real time:

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

Also, journal-based systems store logs using systemd's journal, which can be queried with `journalctl`. For example:

```
journalctl -xe
```

Understanding where logs go helps a ton when diagnosing failed services, crashes, or unauthorized access attempts.

67. How to troubleshoot system crashes?

Troubleshooting a crash starts with isolating where the failure happened—was it a kernel panic, an out-of-memory kill, a hardware issue, or a misbehaving service?

Start with:

```
dmesg | less
```

This will show kernel-level messages. Look for lines like "Kernel panic" or OOM kill events.

Check systemd logs for the timeframe of the crash:

```
journalctl --since "10 minutes ago"
```

If a particular service failed:

```
systemctl status nginx
```

```
journalctl -u nginx
```

If you're debugging a complete freeze or kernel panic, you may want to enable **kdump** or check for crash dumps under `/var/crash` (if enabled).

Other things to check:

- Disk space:

```
df -h
```

- RAM/swap:

```
free -m
```

- Inodes:

```
df -i
```

If your system goes unresponsive randomly, it could also be a hardware issue—like overheating, bad memory, or disk failures. In those cases, tools like `smartctl` can help diagnose disk health.

The key is to correlate the exact time of the crash with relevant logs across `dmesg`, `journalctl`, and `/var/log`.

68. How to use `logrotate` for log management?

Linux systems generate logs constantly. Without management, logs can fill up your disk and create performance issues. `logrotate` is the tool that handles rotating, compressing, and removing old log files automatically.

Configuration lives in two places:

- `/etc/logrotate.conf`: global settings
- `/etc/logrotate.d/`: individual app configs

Each log can be configured with parameters like:

- `daily`, `weekly`, `monthly`: how often to rotate
- `rotate`: how many versions to keep
- `compress`: whether to gzip old logs
- `missingok`: don't error if the log file doesn't exist
- `postrotate`: script or command to run after rotation

Example config for nginx:

```
/var/log/nginx/*.log {
    daily
  missingok
    rotate 7
    compress
    Delaycompress
```

```
    notifempty
    create 0640 www-data adm
    postrotate
        systemctl reload nginx > /dev/null 2>/dev/null
|| true
    endscript
}
```

You can test it with:

```
logrotate -d /etc/logrotate.conf
```

And force it:

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

This keeps your `/var/log` clean and ensures services don't fail due to full disks.

69. How to enable and analyze audit logs?

Audit logs are crucial for tracking security-related events—file modifications, permission changes, logins, or attempts to access restricted areas. The `auditd` daemon is responsible for this.

Start by installing auditd if it's not already present:

```
sudo apt install auditd
sudo systemctl enable --now auditd
```

To watch a specific file:

```
auditctl -w /etc/passwd -p war -k passwd-watch
```

This sets up a watch on `/etc/passwd` for write (`w`), attribute (`a`), and read (`r`) operations with the tag `passwd-watch`.

To search the audit log:

```
ausearch -k passwd-watch
```

You can filter logs by user, PID, event type, or time

range. All audit logs are stored in:

lua

```
/var/log/audit/audit.log
```

This is especially useful for compliance (PCI, HIPAA, etc.), forensic analysis after a breach, or just tracking admin activity.

To persist rules, add them to:

swift

```
/etc/audit/rules.d/audit.rules
```

Be aware that auditd can generate a lot of data, so set up `logrotate` to manage it efficiently.

70. How to recover from a broken or corrupted filesystem?

When a filesystem becomes corrupted—due to an improper shutdown, power failure, or hardware issues—you'll often get boot errors or mounting failures.

If you suspect corruption, the tool to fix it is `fsck`.

First, unmount the affected filesystem:

```
umount /dev/sdb1
```

Then run:

```
fsck /dev/sdb1
```

If it's your root filesystem and you can't boot, you'll need to:

1. Boot from a Live CD/USB
2. Mount the root disk manually
3. Run `fsck` on the affected partition

Example from live environment:

```
sudo fsck -y /dev/sda1
```

The `-y` flag answers "yes" to all fix prompts, which is useful in automation or recovery scripts.

In case of journaled filesystems like ext3/ext4, this can recover from most soft errors. If the disk is physically damaged, consider cloning it with `dd` first to avoid data loss.

And always—after recovery—review logs, run `smartctl` to assess drive health, and consider replacing the hardware if issues persist.

71. How to use Linux in cloud environments like AWS or GCP?

Linux is the standard OS for virtual machines in cloud platforms like AWS, Azure, and GCP. Whether you're spinning up EC2 instances on AWS or

Compute Engine VMs on GCP, you're interacting with cloud-optimized Linux distributions.

Key points to know:

– Most cloud providers offer prebuilt images: Amazon Linux, Ubuntu, CentOS, Debian, or even container-optimized OSes
– SSH key-based login is the default; password authentication is usually disabled
– `cloud-init` is used heavily to automate instance configuration on first boot
– You manage instances using CLI tools like `aws ec2`, `gcloud compute`, or via Terraform

Example: launching an EC2 with AWS CLI:

```
aws ec2 run-instances \

  --image-id ami-xyz \

  --count 1 \

  --instance-type t2.micro \

  --key-name mykey \

  --security-groups my-sg
```

In practice, once your Linux VM is up, you'll: – Use `ssh` to connect – Install packages via `apt` or `yum` – Configure web servers, databases, monitoring agents, etc. – Set up firewall rules with cloud-specific tools or iptables

Linux is also the foundation for container hosts, autoscaling groups, and most cloud-native services. As a DevOps engineer, working in the terminal on cloud Linux VMs becomes second nature.

72. What is `cloud-init` and how does it work?

`cloud-init` is a tool that runs during the first boot of a cloud instance. It's used to initialize settings like hostname, users, SSH keys, and even run commands or install software.

It pulls metadata from the cloud provider (like AWS or GCP) and executes based on what's defined in the user-data script.

Here's an example of a `cloud-init` user-data file:

yaml

```
#cloud-config hostname:
  devserver users:
    - name: devops
      groups: sudo
      shell: /bin/
    ssh-authorized-keys:
        - ssh-rsa AAAA...
  packages:
    - nginx
    - git
  runcmd:
    - systemctl start nginx
```

When you attach this to a VM during provisioning, `cloud-init` will: – Set the hostname
– Create the user
– Install packages
– Start services

This is incredibly useful for automated provisioning with Terraform, EC2 launch templates, or autoscaling workflows.

To debug `cloud-init` behavior:

```
cat /var/log/cloud-init.log
```

You can also re-run it manually for testing:

```
sudo cloud-init clean

sudo cloud-init init

sudo cloud-init modules --mode=config
```

Understanding `cloud-init` is crucial when you're building scalable, reproducible infrastructure in the cloud.

73. How to write scripts for CI/CD pipelines?

scripting is the glue in most CI/CD pipelines—especially for tasks like setting environment variables, building code, running tests, or deploying applications.

Here's a breakdown of what a typical CI/CD script might do:

– Checkout code from Git
– Export environment variables
– Run test or build commands

– Lint code or check quality gates
– Build Docker images
– Push to artifact registry or container registry
– Deploy to a Kubernetes cluster or remote server

Example of a -based deploy script:

```
#!/bin/

set -e

echo "Building the app..."

npm install

npm run build

echo "Running tests..." npm

test

echo "Building Docker image..."

docker build -t myapp:latest .

echo "Pushing to registry..."

docker tag myapp:latest myrepo/myapp:latest

docker push myrepo/myapp:latest

echo "Deploying to cluster..." kubectl

apply -f k8s/deployment.yaml
```

Best practices for in CI/CD: – Always use `set -e` to fail on errors

– Log all output for debugging
– Use variables for configs like image names, branch names, etc.
– Use `trap` to clean up on exit if needed

Even if your main pipeline is written in YAML (GitHub Actions, GitLab, Jenkinsfile), usually powers the core execution blocks.

74. How to use Linux with Docker?

Docker is built on top of Linux, using cgroups, namespaces, and other kernel features. When you use Docker, you're running processes in isolated environments using Linux capabilities.

To install Docker on Linux:

```
curl -fsSL https://get.docker.com | sudo
```

Basic commands: – Start the daemon (if not using systemd):

```
sudo dockerd
```

– Run a container:

```
docker run -d -p 80:80 nginx
```

– Check running containers:

```
docker ps
```

– View container logs:

```
docker logs <container_id>
```

– Execute inside a running container:

```
docker exec -it <container_id>
```

– Stop and remove containers:

```
docker stop <id>
```

```
docker rm <id>
```

Linux lets you dig deeper into Docker internals: – Check container processes via `ps aux` on the host
– Inspect container network bridges using `ip a`
– View container mount points using `mount`, `df -h`, or `lsns`

If you're running Docker in production, you'll also want to manage: – Resource limits with `--memory` and `--cpus`
– Volumes for persistent storage
– Custom networks for microservices
– Docker Compose or Swarm for orchestration

Docker is Linux-native. Knowing Linux internals makes you 10x more effective with containers.

75. How to use Linux with Kubernetes (e.g., kubelet, containerd)?

Kubernetes is heavily dependent on Linux to manage containers and system resources. Every node in a K8s cluster is typically a Linux VM running several components:

– `kubelet`: the agent that communicates with the control plane and runs pods
– `containerd` or `CRI-O`: the container runtime
– `kube-proxy`: handles networking rules
– `cni0`, `flannel.1`, or other virtual interfaces for pod networking
– systemd or Docker to manage services like `kubelet`

If you SSH into a Linux worker node, you can inspect pod processes with:

```
ps aux | grep kube
```

Check kubelet logs:

```
journalctl -u kubelet
```

Check the container runtime:

```
crictl ps
```

```
crictl info
```

View pod containers on the node:

```
sudo ctr containers list
```

To debug a stuck pod, you might need to: – Inspect logs in `/var/log/pods/` or `/var/log/containers/`
– Look at iptables rules added by kube-proxy
– Check if the node is under memory or CPU pressure with `top`, `htop`, or `vmstat`
– Look for OOMKills in `dmesg` or `journalctl -k`

Having Linux mastery makes you more effective at running and troubleshooting Kubernetes. Most advanced Kubernetes issues boil down to Linux-level problems.

76. How to troubleshoot high CPU or memory usage in production?

When your system is slow or laggy, the first signs usually show up as spikes in CPU or memory. To figure out what's happening, you start by identifying which process is responsible.

For CPU:

```
top
```

Sort by `%CPU` to find the culprit. Press `P` in `top` to sort by

CPU. Use `htop` if installed—it's more visual:

```
htop
```

If one process is hogging the CPU, you'll see it at the top. If it's a runaway script or service, you can inspect it further using:

```
strace -p <pid>
```

This helps you see what syscalls it's making. If it's stuck in a loop or waiting on I/O, you'll know.

For memory issues:

```
free -m
```

Shows how much RAM is being used. If swap is high and RAM is full, you're under memory pressure.

To check which process is using the most memory:

```
ps aux --sort=-%mem | head
```

Also, watch out for zombie processes (`Z` status in `ps`). If memory leak is suspected, tools like `valgrind`, `smem`, or application-specific profilers come in handy.

If the issue is regular and not just a one-time thing, consider setting up alerts via Prometheus, CloudWatch, or another monitoring tool.

77. How to debug application crashes and service failures?

When an application crashes, you want to figure out: – What caused it
– When it happened

– If it can be reproduced

Start by checking the service logs. If it's managed by systemd:

```
journalctl -u myapp.service
```

Check the exit status:

```
systemctl status myapp
```

If it failed with a segmentation fault or core dump, make sure core dumps are enabled:

```
ulimit -c unlimited
```

Then inspect the core file with `gdb`:

```
gdb /path/to/app core
```

If it's a Python or Node.js app, logs might be printed to a file or standard error. Make sure logs aren't being redirected to `/dev/null`.

Also check for file descriptor leaks, full disks, permission issues, or bad configuration files.

Use `strace` to run the app and catch syscalls:

```
strace -o trace.log ./myapp
```

This shows where it failed—missing files, permission denied errors, failed system calls.

If it's a service that starts and then immediately stops, it might be missing environment variables, dependencies, or files it expects at runtime.

Always correlate with logs and context: Was there a deployment? A config change? A spike in traffic?

78. How do you perform a root cause analysis of a production issue?

Root Cause Analysis (RCA) is the post-mortem where you identify what broke, why it broke, and how to prevent it next time.

The approach usually looks like this:

1. **Timeline analysis** – When did the issue start? Correlate logs, metrics, and alerts.
2. **Symptom collection** – What exactly went wrong? Was it downtime, slowness, data loss?
3. **Scope** – Was it one host, one service, or the entire system?
4. **Immediate cause** – Did a process crash? Did a config change go out?
5. **Underlying cause** – Why was that change made? Why did the safeguard fail?

Example: – Issue: App was unreachable
– Immediate cause: Nginx failed to start
– Deeper cause: Config had a bad port
– Root cause: Deployment script didn't validate the config before reload
– Fix: Add a `nginx -t` check before deploying

Tools that help during RCA: – `journalctl` or log aggregation tools (ELK, Loki)
– Monitoring dashboards (Prometheus, Grafana, Datadog)
– Deployment logs from CI/CD
– Command histories (`history | grep kube`, etc.)

Once you've found the cause, document the fix and add a check or alert to prevent recurrence. That's how you mature systems.

79. How to trace open ports and detect malware?

First, to see what ports are open and which processes are using them:

```
ss -tulnp
```

This shows TCP/UDP listeners and which PID owns each.

Or with `netstat`:

```
netstat -tulnp
```

Now, if you suspect malware, look for strange ports, especially high ones (like 1337, 6667, etc.), or odd services listening.

Then check for unknown processes:

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

Or unknown binaries in `/tmp`, `/var/tmp`, or `/dev/shm`:

```
find /tmp /dev/shm -type f -executable
```

To dig deeper, use `lsof` to list open files and sockets:

```
lsof -i
```

Scan for rootkits with:

```
chkrootkit
```

```
rkhunter
```

You can also use `auditd` to monitor what processes are being launched, or run `tripwire` or `AIDE` to look for modified system files.

If something is really shady—like a hidden process—you can use `unhide` to detect kernel-level rootkits.

80. How to track file access and audit trails?

If you want to know who accessed or modified a file, you need either auditing tools or proper logging.

The most robust tool is `auditd`. To watch a file:

```
auditctl -w /etc/passwd -p war -k passwd_watch
```

This tracks writes, attribute changes, and reads.

To view the audit logs:

```
ausearch -k passwd_watch
```

You can filter by PID, UID, or time range to see exactly who did what.

If you want to watch directories:

```
auditctl -w /etc/ssh/ -p wa -k ssh_dir_watch
```

Beyond `auditd`, you can use: – `inotifywait` for real-time file change tracking
– `ls -ltu` to check last access times
– `find /etc -amin -10` to find files accessed in the last 10 minutes

In mission-critical environments, file access tracking is paired with alerts, so you know when someone even opens a sensitive file—let alone edits it.

81. How to use `rsync`, `tar`, `dd`, and `scp` for backups?

These are four of the most versatile tools for backing up and moving data in Linux.

Start with **`rsync`**, the most efficient for syncing files across directories or remote systems. It copies only the differences between source and destination.

Basic usage:

```
rsync -av /source/ /backup/
```

The `-a` flag preserves permissions, timestamps, etc. and `-v` gives verbose output.

To backup over SSH:

```
rsync -av -e ssh /etc/ user@remote:/backup/etc/
```

`tar` is used to compress and archive files into a single `.tar` or `.tar.gz` file.

Create a backup:

```
tar -czvf backup.tar.gz /etc /var/log
```

Extract:

```
tar -xzvf backup.tar.gz
```

It’s perfect for packaging configs or entire directories in one go.

`dd` works at the block level. It’s often used to make complete disk or partition images.

Create a disk image:

```
dd if=/dev/sda of=/mnt/backup/sda.img bs=4M
```

Restore it:

```
dd if=/mnt/backup/sda.img of=/dev/sda bs=4M
```

It’s very powerful but risky—mistakes can overwrite critical data.

`scp` is simple and secure for copying files between systems over SSH.

Basic use:

```
scp backup.tar.gz user@remote:/backups/
```

To copy a directory:

```
scp -r /etc user@remote:/backups/etc
```

These tools together give you the flexibility to automate backups, sync large data sets, create snapshots, or move critical files between hosts.

82. How to automate backup tasks in production?

Automation is key to ensuring consistency and reliability in backup routines.

Start with a simple script:

```
#!/bin/ DATE=$

(date +%F)

BACKUP_DIR="/backups"

TARGET="/etc"

tar -czf $BACKUP_DIR/etc-backup-$DATE.tar.gz $TARGET
```

Make it executable:

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

Now schedule it with `cron`:

```
crontab -e
```

Add this line for a daily backup at 2AM:

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

Log the output:

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

In production, you'll often use: – `rsync` for incremental backups
– `cron` or `systemd timers` for scheduling
– `logrotate` for rotating old backup logs
– Cloud CLIs (like `aws s3 cp`) to push to remote storage
– Tools like BorgBackup, Restic, or Velero (for Kubernetes) for advanced backup handling

Always test your backups by restoring them on staging to confirm they're valid.

83. How to restore deleted files from snapshots?

If you've lost or deleted a file and you're using snapshot-based backups like LVM, ZFS, Btrfs, or cloud-managed snapshots, you can often restore it without much hassle.

With **LVM**, create a snapshot before making risky changes:

```
lvcreate --size 1G --snapshot --name mysnap
/dev/vg0/root
```

Mount the snapshot:

```
mount /dev/vg0/mysnap /mnt/snap
```

Then you can copy back deleted or corrupted files:

```
cp /mnt/snap/etc/important.conf /etc/
```

With **ZFS or Btrfs**, snapshots are even easier:

```
zfs snapshot pool/data@snapshot1

zfs rollback pool/data@snapshot1
```

In the **cloud**, tools like AWS Backup or EBS snapshots can restore full volumes: – Create a snapshot
– Restore it to a new volume
– Mount the new volume on another instance
– Copy the file back using `scp`, `rsync`, or just plain `cp`

Snapshot-based recovery is fast and doesn't need you to unpack large archive files. It's ideal for systems that need quick rollbacks with minimal downtime.

84. How to configure and restore a bootable system image?

If you're preparing for full system disaster recovery, a complete disk image is your best bet. You can make the system bootable again using tools like `dd`, `Clonezilla`, or creating your own recovery ISO.

With **`dd`**, create an image of the entire disk:

```
dd if=/dev/sda of=/mnt/backup/sda.img bs=4M
status=progress
```

To restore:

```
dd if=/mnt/backup/sda.img of=/dev/sda bs=4M
```

This will restore everything—partitions, MBR, filesystem, bootloader.

To make a system bootable after restoring: – Ensure GRUB is reinstalled:

```
grub-install /dev/sda
```

```
update-grub
```

– Check `/etc/fstab` for correct UUIDs if restoring to different hardware

You can also use tools like **Clonezilla** to create full disk images and restore them interactively. It supports compression, encryption, and partition cloning.

On cloud platforms, take machine images (like AMIs in AWS) to recreate full bootable systems in one step.

The key is to have both system files and boot records backed up—without that, restoring won't bring the system back online.

85. How do you manage offsite backups securely?

Offsite backups are critical for disaster recovery. Fire, hardware failure, or ransomware can take out your primary storage, but if you've got offsite backups, you're protected.

For secure transfers, use encrypted protocols: – `rsync -e ssh`
– `scp`
– `sftp`

To sync a backup folder to a remote server:

```
rsync -avz /backups/ user@remote:/safe-storage/
```

If you're using the cloud, tools like:

```
aws s3 cp /backups/ s3://mybucket/ --recursive --sse
```

Use `--sse` or bucket policies to ensure server-side encryption.

For even more security: – Encrypt files using `gpg` or `openssl` before uploading
– Use `restic` or `duplicity` with encryption enabled
– Store backups in a different region or

provider Add checksums to verify integrity after

upload:

```
md5sum backup.tar.gz > backup.md5
```

Always test restore procedures from offsite locations. A backup isn't complete unless it can be restored.

86. What is the difference between containers and virtual machines?

Both containers and virtual machines are used to isolate workloads, but the way they achieve that isolation is fundamentally different.

Virtual Machines (VMs) virtualize the hardware. They run full operating systems on top of a hypervisor like KVM, VMware, or VirtualBox. Each VM has its own kernel, filesystem, and virtual hardware. This makes them heavy in terms of resource usage and slower to boot.

Containers, on the other hand, use the host's kernel. They isolate applications using Linux kernel features like namespaces and cgroups. Because containers don't need to boot an entire OS, they start in milliseconds and use far fewer resources.

Think of it like this: – A VM is like a full house with plumbing, power, and walls.
– A container is like a room inside a shared apartment with private access but shared infrastructure.

Containers are ideal for microservices and cloud-native apps. VMs are better when you need full OS-level separation, e.g., for legacy apps, custom kernels, or multiple OS types on one host.

87. How does `chroot` work and how is it different from containers?

`chroot` is an older Linux technique to change the apparent root directory for a process. It restricts the process's view of the filesystem—it can't access anything outside the new root path.

You can use it to build minimal environments:

```
mkdir /mychroot

debootstrap stable /mychroot
http://deb.debian.org/debian

chroot /mychroot /bin/
```

Once inside, the process thinks `/mychroot` is `/`.

But here's the thing: `chroot` only restricts the filesystem. It doesn't isolate network, user IDs, or processes. That's why it's not secure by itself.

Containers (like Docker) use multiple layers of isolation: – Namespaces: PID, network, mount, IPC, UTS, user
– cgroups: resource limits
– Union filesystems for layered image management

So, while `chroot` is useful for sandboxing or building recovery environments, containers are the proper way to isolate full workloads today.

88. What are namespaces and cgroups in Linux?

These are the two core features that make containers possible in Linux.

Namespaces isolate what a process can see: – **PID namespace**: each container has its own process tree
– **NET namespace**: isolated network stack (interfaces, routes, ports)

– **MNT namespace**: separate mount points
– **IPC namespace**: isolates inter-process communication
– **UTS namespace**: allows hostname isolation
– **USER namespace**: maps user IDs differently inside vs. outside container

Each container has its own set of these, which means it can't see or interact with the host or other containers.

Cgroups (control groups) limit what a process can use: – Memory
– CPU
– Disk I/O
– Network bandwidth
– Number of processes

Example: Docker uses cgroups to make sure one container can't consume all system memory and bring the node down.

To see cgroups in action:

```
cat /proc/self/cgroup
```

Or use:

```
systemd-cgls
```

Namespaces provide isolation, cgroups provide limitation. Together, they power the security and performance guarantees of containers.

89. How to manage containers using `podman`, `docker`, or `containerd`?

While Docker is the most popular container tool, alternatives like Podman and containerd have become important in production, especially in Kubernetes environments.

Docker is a full-stack container platform. It includes: – Container runtime
– Image builder

– Registry client
– CLI and API

You use it like:

```
docker run -d nginx
```

Podman is Docker-compatible but daemonless and rootless. It doesn't require a background service (`dockerd`). It supports the same CLI commands:

```
podman run -d nginx
```

Podman can be run as a normal user, which improves security, and works well in rootless containers and CI/CD pipelines.

containerd is a lower-level runtime used by Kubernetes. It doesn't have a built-in CLI, but you can interact with it using:

```
ctr
```

Or better yet, with:

```
crictl
```

Which talks to the Kubernetes CRI.

Docker uses containerd under the hood. So in Kubernetes, if you're using containerd directly, you're skipping Docker's extra layers.

In production: – Use Docker for development and testing
– Use Podman for security-sensitive environments
– Use containerd in Kubernetes clusters

90. How to inspect running containers and their internals?

Whether you're troubleshooting or just curious, inspecting what's happening inside a container is a regular part of container operations.

To list running containers:

```
docker ps
```

To see what's inside:

```
docker exec -it <container_id> /bin/
```

Or inspect metadata:

```
docker inspect <container_id>
```

You'll see IP address, mount points, image info, restart policy, and more.

To view logs:

```
docker logs <container_id>
```

To see the container's processes:

```
docker top <container_id>
```

To look at the filesystem from outside:

```
docker diff <container_id>
```

Shows what files were added or changed since the container started.

For resource usage:

```
docker stats
```

Live view of CPU, memory, and I/O usage per container.

With `crictl` on Kubernetes:

```
crictl ps
```

```
crictl inspect <container_id>
```

Knowing how to peek into containers helps you diagnose: – App-level issues
– File permission problems
– Networking failures
– Resource bottlenecks

91. How to write an interactive shell script?

An interactive shell script prompts the user for input, processes it, and responds accordingly. These scripts are great for tools that require user confirmation or setup, like installers, CLI utilities, or interactive DevOps scripts.

Here's a simple pattern:

```
#!/bin/

echo "Welcome to the project setup."

read -p "Enter your project name: " project

read -p "Do you want to continue? (yes/no): " confirm

if [ "$confirm" == "yes" ]; then

    mkdir "$project"

    echo "Project '$project' created."

else

    echo "Operation canceled."

fi
```

You can take it further with menus using `select`:

```
select option in start stop status quit; do
  case $option in
    start) echo "Starting app...";;
    stop) echo "Stopping app...";;
    status) echo "Checking status...";;
    quit) break;;
    *) echo "Invalid option";;
  esac
done
```

Interactive scripts are useful for internal tooling or when building helper utilities for your team.

92. How to schedule and monitor scripts with cron?

`cron` is the go-to scheduler for recurring tasks in Linux. To set it up:

First, create a script:

```
#!/bin/
echo "Backup started at $(date)" >>
/var/log/mybackup.log
tar -czf /backup/home-$(date +%F).tar.gz /home
```

Make it executable:

```
chmod +x /usr/local/bin/mybackup.sh
```

Now edit your crontab:

```
crontab -e
```

And schedule it:

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

This runs daily at 2 AM.

To log errors and outputs:

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

To monitor whether it ran: – Check `cron` logs: `/var/log/syslog` (Debian) or `/var/log/cron` (RHEL)
– Or use `grep` with timestamps
– Add custom logging in the script itself

For more advanced monitoring: – Integrate with `monit`, `cronwrap`, or a healthcheck endpoint
– Set up alerts when the script fails or doesn't run

93. How to pass arguments to a shell script?

Shell scripts accept command-line arguments just like a CLI tool.

Inside the script:

- `$1`, `$2`, `$3`, etc. represent positional parameters
- `$#` is the number of arguments
- `$@` is the list of all arguments

Example:

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

Run it:

```
./script.sh dev production
```

To make it safer:

```
if [ "$#" -lt 2 ]; then
  echo "Usage: $0 <source> <destination>" exit
    1
fi
```

You can also loop through all arguments:

```
for arg in "$@"; do
  echo "Processing
  $arg"
done
```

This lets you create flexible, reusable scripts like:

```
./deploy.sh staging us-east-1
```

In real-world automation, this is how scripts handle multiple environments, regions, services, or versions.

94. How to make scripts modular with functions?

Functions break your script into logical blocks. This makes your scripts more maintainable, reusable, and testable.

Here's a basic structure:

```
#!/bin/

log() {

  echo "$(date +%F_%T) - $1"

}

backup() {

  log "Starting backup..."

  tar -czf /backup/home.tar.gz /home

  log "Backup completed."

}

cleanup() {

  log "Cleaning old backups..."

  find /backup -name "*.gz" -mtime +7 -delete

  log "Cleanup complete."

}

# Main execution

Backup
```

```
cleanup
```

You can also pass arguments to functions:

```
deploy_app() {
  env=$1
  echo "Deploying to $env environment"
}
```

Use this structure to build libraries of functions (`utils.sh`, `deploy.sh`, etc.) and `source` them in multiple scripts.

It's a solid pattern for growing from quick scripts to real automation tools.

95. How to handle errors and logging in scripts?

Handling errors cleanly is what separates good scripts from fragile ones. You don't want your script to fail silently or continue on an error.

Start with:

```
set -e
```

This makes the script exit immediately on any command failure.

Also add:

```
set -o pipefail
```

To catch errors in pipelines.

Use traps to catch exits:

```
trap 'echo "Something went wrong. Exiting..."' ERR
```

For logging, build a simple logger function:

```
log() {
  echo "$(date +%F_%T) $1" >> /var/log/myscript.log
}
```

You can add levels:

```
log_info() { echo "$(date) [INFO] $1"; }
log_error() { echo "$(date) [ERROR] $1" >&2; }
```

Then wrap your commands:

```
log_info "Starting backup"
tar -czf backup.tar.gz /home || log_error "Backup failed"
```

And if you want to redirect all script output to a log:

```
exec > >(tee -a /var/log/myscript.log) 2>&1
```

Solid error handling and logging makes your scripts production-grade and ready to be scheduled, monitored, and trusted.

96. How would you approach troubleshooting a slow Linux system?

Troubleshooting performance requires a clear, layered strategy — start broad, then narrow down.

First step: ask *"Is this system-wide or just one process?"*

Start with:

```
uptime
```

This gives you system load. Then:

```
top
```

Shows high CPU/memory processes.

Look for: – One core pegged at 100%
– A process with increasing memory
– High `%wa` (I/O wait), which means disk is the

bottleneck If it's memory pressure:

```
free -m
```

If it's disk:

```
iostat -xz 1
```

Or:

```
iotop
```

If it's network-related:

```
ss -s
```

```
iftop
```

```
nload
```

Next, look at logs:

```
journalctl -xe
```

```
dmesg | tail
```

And lastly, check recent changes: – New deployments?
– Package updates?
– Cron jobs?

The key is to isolate symptoms and rule out components step-by-step: CPU, memory, disk, network, or app layer. Don't jump straight to guessing — start at the system level and drill down.

97. What's your strategy for managing hundreds of Linux servers?

You can't SSH into every box. Managing at scale means treating infrastructure like code, automating everything, and having centralized control.

Start with configuration management: – Use **Ansible**, **Puppet**, or **Chef** to define server state
– Push config changes in bulk
– Automate user creation, package installation, cron setup, and more

Next, use **SSH key management** via tools like: – HashiCorp Vault
– AWS Systems Manager (SSM)
– LDAP or centralized auth (FreeIPA)

For monitoring, deploy agents: – **Prometheus** + **Grafana** for metrics
– **ELK** or **Loki** for logs
– **Node Exporter** for resource metrics

Use orchestration and grouping: – Host tagging (`web`, `db`, `staging`, etc.)
– Inventory management via dynamic sources (EC2, GCP, etc.)

Also important: – Backup strategies per tier
– Patch management workflows
– Immutable infrastructure (via Packer, Terraform, etc.)

The real trick is visibility and automation — one mistake across 100 boxes is a disaster if you don't have checks and balances in place.

98. What would you include in a Linux system hardening checklist?

System hardening is about reducing the attack surface. A hardened system resists intrusion, even if an attacker gets through the first layer.

Here’s a high-level checklist:

– Disable root SSH login (`PermitRootLogin no`)
– Disable password login (`PasswordAuthentication no`)
– Use key-based authentication only
– Install and configure **fail2ban**
– Configure **ufw**, **iptables**, or **firewalld**
– Remove unused services (`systemctl list-units --type=service`)
– Run security updates regularly (`apt update && apt upgrade`)
– Set file permissions correctly
– Monitor `/etc/shadow`, `/etc/passwd`, `/etc/sudoers`
– Enable auditing (`auditd`)
– Harden `/etc/sysctl.conf` with things like `net.ipv4.conf.all.rp_filter = 1`
– Configure strong password policies with `pam_pwquality`
– Limit user access via `/etc/security/access.conf`

And don’t forget: – Backup regularly
– Test recovery procedures
– Monitor login activity (`last`, `w`, `who`, `journalctl -u ssh`)
– Use a host-based intrusion detection system like **AIDE** or **Tripwire**

Security is not a one-time thing. Set up automatic alerts and reviews to continuously harden over time.

99. How do you keep your Linux skills sharp and production-ready?

Linux is too vast to know everything at once, but staying sharp is about consistent exposure and real projects.

Here's how to stay sharp:

– Practice **real-world scripting problems**
– Contribute to open-source infrastructure tools
– Spin up cloud servers and build things from scratch
– Watch logs, tune system performance, break things on purpose
– Solve challenges on platforms like **OverTheWire**, **TryHackMe**, or **HackTheBox**
– Read man pages for tools you use daily (`man grep`, `man`)
– Teach others — when you explain something, you learn it deeper
– Maintain dotfiles, aliases, or helper scripts
– Follow kernel changelogs and security feeds (like CVEs)
– Automate repetitive tasks — that's where mastery comes from

If you're always trying to *understand, not just use*, you'll evolve naturally from user to engineer to architect.

100. What advice would you give someone new to Linux aiming for DevOps roles?

Start with the **why**, not just the commands. Linux powers cloud, automation, containers, and CI/CD. Every DevOps pipeline sits on a Linux box somewhere.

Here's what I'd tell them:

– Learn the command line first: navigation, file management, process handling
– Then move to scripting: , conditionals, loops, arguments, error handling
– Learn version control (Git), then package management
– Understand systemd, networking, logs, firewalls

– Pick one config management tool (Ansible is beginner-friendly)
– Learn Docker and Kubernetes — containers are key in DevOps
– Practice on real cloud infrastructure — AWS, GCP, or Azure
– Don't skip security. Learn the basics of SSH, sudo, file permissions, and auditing
– Build your own projects: deploy apps, automate tasks, create CI/CD pipelines
– Be curious. When you see a command in a tutorial, look it up. Reverse engineer things.

Most importantly, **don't just learn Linux — live in it.** Use it daily. Break it. Fix it. That's how you become production-ready.