Linux Command Line Course

Contents

Linux Command Line Course

From First Principles to Networking Commands

Chapter 1: How Commands Actually Work

Before memorizing commands, understand the machine underneath.

The shell is a program (bash, zsh, sh) that reads text you type, interprets it, and asks the kernel to do something. It is not the OS — it's an interpreter sitting on top of it.

A command is one of four things:

1. **A shell built-in** — code that lives inside the shell itself (`cd`, `echo`, `export`, `alias`). No separate process is created.
2. **An external program** — a binary file on disk (`/usr/bin/ls`, `/usr/bin/find`). The shell searches `$PATH`, finds the file, and asks the kernel to execute it as a new process.
3. **A shell function** — a named block of shell code you or a script defined.
4. **An alias** — a shorthand the shell expands before running (e.g., `alias ll='ls -la'`).

How the shell finds a command:

```
type ls
which ls
```

Both tell you where a command resolves from. The shell checks, in order: aliases → functions → built-ins → each directory in `$PATH`, left to right. Run `echo $PATH` to see the search list. This is

why two systems can run “the same command” and get different behavior — different binary, different version, different path.

Anatomy of a command:

```
command  -flags  --long-options  arguments
```

Example:

```
grep -i --color "error" server.log
```

- `grep` — the program
- `-i` — a short flag (case-insensitive)
- `--color` — a long option
- `"error"` — the pattern (argument)
- `server.log` — the file (argument)

Exit codes. Every command returns a number when it finishes: `0` means success, anything else means some kind of failure. Check it with:

```
echo $?
```

This is how scripts make decisions — && runs the next command only if the previous one succeeded; || runs it only if the previous one failed.

```
mkdir backup && cp file.txt backup/
ping -c1 google.com || echo "no internet"
```

Processes. When you run an external command, the kernel creates a process with a PID (process ID), gives it memory, and schedules it on the CPU. `ps`, `top`, and `kill` all operate on that PID, not on the command name.

Chapter 2: Navigating and Reading

Command	What it does
`pwd`	print current directory
`ls`, `ls -la`	list files (all, long format)
`cd path`, `cd ..`, `cd ~`	change directory
`cat file`	print a file’s contents
`less file`	scroll through a file (q to quit)
`head -n 20 file`	first 20 lines
`tail -n 20 file`	last 20 lines
`tail -f logfile`	follow a file live (logs)
`file name`	identify file type
`tree`	show directory structure

~ is your home directory. `.` is the current directory. `..` is the parent. `-` (after `cd`) takes you to the previous directory.

Chapter 3: Files and Directories

Command	What it does
`touch file`	create empty file / update timestamp
`mkdir dir`, `mkdir -p a/b/c`	create directory (`-p` makes parents too)
`cp src dst`, `cp -r dir1 dir2`	copy (recursive for directories)
`mv src dst`	move or rename
`rm file`, `rm -r dir`, `rm -rf dir`	delete (recursive, force — dangerous)
`ln -s target link`	create a symbolic link

`rm -rf` deletes without confirmation and without a trash bin. Treat it like a loaded weapon — always double-check the path before pressing enter, especially with wildcards.

Chapter 4: Finding Things

This is where people usually get lost, because `find`, `grep`, and `locate` solve different problems.

`find` — search by filename, type, size, date. It walks the directory tree.

```
find /home -name "*.log"
find . -type f -mtime -7        # files modified in last 7 days
find . -type d -empty           # empty directories
find . -size +100M              # files over 100MB
find . -name "*.tmp" -delete    # find and delete
```

`grep` — search by file content. It reads inside files for a pattern.

```
grep "error" app.log
grep -r "TODO" ./src            # recursive
grep -i "warning" *.log         # case-insensitive
grep -c "fail" access.log       # count matches
grep -n "port" config.yml       # show line numbers
```

`locate` — search a prebuilt filename index (fast, but can be stale).

```
locate nginx.conf
sudo updatedb   # refresh the index
```

`which` / `whereis` — locate an executable, not a general file.

Chapter 5: Permissions and Ownership

Linux permissions are three sets of three: owner, group, others — each with read (r), write (w), execute (x).

```
ls -l file.txt
-rw-r--r--  1 alice staff  1024 Jul 24 09:00 file.txt
```

Reading the string: type, owner perms, group perms, other perms.

```
chmod 755 script.sh      # rwx for owner, r-x for group and others
chmod +x script.sh       # add execute permission
chown alice:staff file   # change owner and group
sudo command             # run one command as root
```

Numeric permissions: r=4, w=2, x=1. Add them per group: `7 = rwx`, `5 = r-x`, `4 = r--`.

Chapter 6: Processes and Resource Use

Command	What it does
`ps aux`	list all running processes
`top` / `htop`	live process/resource monitor
`kill PID`	terminate a process (SIGTERM)
`kill -9 PID`	force-kill (SIGKILL, last resort)
`killall name`	kill by process name
`jobs`, `bg`, `fg`	manage jobs in the current shell
`command &`	run in background
`nohup command &`	keep running after you log out
`df -h`	disk space by filesystem
`du -sh dir`	size of a directory
`free -h`	memory usage
`uptime`	load average and uptime

Chapter 7: Text Processing and Pipes

The Unix philosophy: small programs, each doing one thing, chained together with | (pipe), where one command's output becomes the next command's input.

```
cat access.log | grep "404" | wc -l
```

Reads the log, filters lines containing "404", counts them.

Command	What it does
`wc -l`	count lines
`sort`	sort lines
`uniq -c`	collapse duplicate adjacent lines with a count
`cut -d',' -f2`	extract a column by delimiter
`awk '{print $1}'`	extract fields (more powerful)
`sed 's/foo/bar/g'`	find and replace
`xargs`	turn piped input into command arguments
`tr 'a-z' 'A-Z'`	translate characters

Common combo — top 5 most frequent IPs hitting a server:

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

Redirection:

```
command > file      # overwrite file with output
command >> file     # append output to file
command < file       # feed file in as input
command 2> err.log  # redirect errors only
command > out.log 2>&1   # redirect both output and errors
```

Chapter 8: Networking Commands

This is the section most people are actually here for. Networking commands answer three questions: what's my address, can I reach the other side, and what's actually happening on the wire.

Find your own IP address

```
ip addr show         # modern, replaces ifconfig — shows all interfaces and IPs
ip -4 addr             # IPv4 only
hostname -I            # quick list of local IPs
curl ifconfig.me        # your public IP (as seen from the internet)
curl ipinfo.io          # public IP + geolocation info
```

`ifconfig` still exists on many systems but is deprecated in favor of `ip`, part of the `iproute2` package.

Find DNS information

```
nslookup example.com          # basic DNS lookup
dig example.com                # detailed DNS lookup
dig example.com +short         # just the IP
dig -x 8.8.8.8                  # reverse lookup (IP → hostname)
host example.com                # quick alternative to nslookup
cat /etc/resolv.conf           # see which DNS servers you're configured to use
```

`dig` is the professional tool — it shows the full DNS response, TTL, and which record type (A, AAAA, MX, TXT, NS) you asked for:

```
dig example.com MX      # mail servers
dig example.com NS      # nameservers
```

Test connectivity

```
ping example.com             # send ICMP packets, check reachability + latency
ping -c 4 example.com          # send only 4 packets
traceroute example.com         # show the hop-by-hop path to a destination
mtr example.com                 # traceroute + ping combined, live-updating
```

Check ports and connections

```
ss -tulwn           # list listening TCP/UDP ports (modern replacement for netstat)
netstat -tulwn            # older equivalent, still common
nc -zv example.com 443    # test whether a specific port is open
telnet example.com 80     # manually connect to a port (legacy but instructive)
```

Transfer data / talk to servers

```
curl https://example.com              # fetch a URL, print response
curl -I https://example.com            # headers only
curl -O https://example.com/file.zip   # download and save
wget https://example.com/file.zip      # download-focused alternative to curl
```

Remote access

```
ssh user@host                 # remote shell
ssh -i key.pem user@host       # use a specific private key
scp file.txt user@host:/path/  # copy a file to a remote host
scp user@host:/path/file .     # copy a file from a remote host
rsync -avz src/ user@host:dst/  # efficient sync, only transfers changes
```

Routing and interfaces

```
ip route                # show the routing table
ip route get 8.8.8.8    # which interface/route would be used to reach an address
arp -a                    # show the ARP table (IP-to-MAC mappings on the LAN)
```

Quick reference — "how do I find X"

You want to know	Command
My local IP	`ip addr show`
My public IP	`curl ifconfig.me`
An IP's hostname / vice versa	`dig`, `nslookup`, `host`
If a host is reachable	`ping`
The path packets take	`traceroute` / `mtr`
What's listening on my machine	`ss -tulwn`
If a specific port is open	`nc -zv host port`
My DNS servers	`cat /etc/resolv.conf`
My routing table	`ip route`

Chapter 9: Package Management

Varies by distro family:

```
# Debian/Ubuntu
sudo apt update && sudo apt install package
sudo apt remove package

# RHEL/Fedora/CentOS
sudo dnf install package
sudo dnf remove package

# Arch
sudo pacman -S package
sudo pacman -R package
```

Chapter 10: Putting It Together — Shell Scripting Basics

A script is just a sequence of commands in a file, run by an interpreter.

```bash
#!/bin/bash
# check_site.sh — checks if a site is up and logs the result

SITE="example.com"

if ping -c 1 "$SITE" > /dev/null 2>&1; then
  echo "$(date): $SITE is up" >> uptime.log
else
  echo "$(date): $SITE is DOWN" >> uptime.log
fi
```

Run it:

```
chmod +x check_site.sh
./check_site.sh
```

`#!/bin/bash` (the shebang) tells the kernel which interpreter to run the file with. `$VAR` reads a variable. `$(command)` substitutes a command's output into a string. `if/then/else/fi` is the conditional block; the condition is really just "did the command before `then` exit with code 0."

Appendix: One-Page Cheat Sheet

```
NAVIGATION        pwd, ls -la, cd, tree
FILES             touch, mkdir -p, cp -r, mv, rm -rf, ln -s
FIND              find . -name "*.log", grep -r "text" ., locate name
PERMISSIONS       chmod 755, chown user:group, sudo
PROCESSES         ps aux, top, kill -9 PID, jobs, bg/fg
DISK/MEM          df -h, du -sh, free -h
TEXT              cat, less, head, tail -f, sort, uniq -c, awk, sed, wc -l
PIPES/REDIRECT    cmd1 | cmd2, > file, >> file, 2>&1
IP                ip addr show, curl ifconfig.me
DNS               dig example.com, nslookup example.com, dig -x IP
CONNECTIVITY      ping -c4 host, traceroute host, mtr host
PORTS             ss -tulwn, nc -zv host port
TRANSFER          curl -O url, wget url, scp file user@host:/path
REMOTE            ssh user@host, rsync -avz src/ dst/
ROUTING           ip route, arp -a
EXIT STATUS       echo $?, cmd1 && cmd2, cmd1 || cmd2
```