Linux Performance Tuning

A complete 8-chapter guide to diagnosing and resolving CPU, memory, disk, network, and system bottlenecks

TABLE OF CONTENTS

CHAPTER 1 OF 8

Chapter 1 — Performance Overview & Essential Tools

Before you can fix a performance problem, you need to know what kind of problem you have. A server that "feels slow" could be running out of CPU, exhausting its RAM, waiting on a disk, throttled by the network, or it might just be a caching layer warming up. This chapter gives you a framework for thinking about performance and a toolkit for finding out which one it actually is.

> **What this chapter covers:** Load average — what it really means. The USE method — a structured diagnostic framework. The core toolkit (uptime, top, htop, vmstat, free, df, dmesg). How to read htop's columns. A practical scenario: a user reports the server is slow — where do you start?

Load Average — What It Actually Means

The first number you look at on a sluggish server is load average. Run `uptime` or look at the top line of `top` or `htop`:

```
$ uptime
 14:32:08 up 42 days, 3:21,  2 users,  load average: 0.85, 1.42, 2.10
```

The three numbers are the average number of processes that were either **running or waiting to run** (runnable) or **waiting on I/O** (uninterruptible) over the last 1, 5, and 15 minutes. They are *not* percentages.

1 MINUTE	5 MINUTES	15 MINUTES
0.85	1.42	2.10
Most recent snapshot. Spikes here but not in the 15-minute figure suggest a short burst, not a sustained problem.	The most useful number for spotting a developing trend. Rising from 5-min to 1-min means load is increasing.	The long-term baseline. If this is high you have a sustained problem, not a blip. Compare against CPU count.

Interpreting load average against CPU count

A load average of 4.0 means very different things on a single-core machine versus a 16-core machine. The meaningful figure is **load per core**: divide load average by the number of logical CPUs.

```
$ nproc
8
# On this 8-core machine, a load of 8.0 means 100% utilisation — every core is busy
# A load of 4.0 means 50% — half the cores are occupied on average
# A load of 12.0 means 150% — 4 processes are actively queuing for CPU time
```

> **Load average includes I/O wait, not just CPU.** A process blocked waiting for a slow disk is still counted in load average even though it's not using any CPU. This is why a machine with a dying hard drive can show a load of 20 with CPU usage at just 5%. High load + low CPU = look at disk or network first.

Reading the trend — the numbers tell a story

1-MIN	15-MIN	WHAT IT MEANS	ACTION
0.5	0.5	Quiet and stable. System is idle.	No action needed.
3.0	0.5	Recent spike, settling back down.	Check what ran recently: `dmesg -T | tail`, check cron logs.
0.5	3.0	Something busy before, calmed now.	Check logs for earlier in the window. May be resolved.
8.0	8.0	Sustained high load (on an 8-core machine: 100%).	Investigate CPU and I/O immediately.
20.0	2.0	Sudden severe spike right now.	Open `htop` immediately — something just went wrong.

The USE Method — A Diagnostic Framework

Created by Brendan Gregg (the foremost authority on Linux performance), the USE method gives you a structured way to check every system resource rather than guessing. For each resource — CPU, memory, disk, network — ask three questions:

U — Utilisation

What percentage of time is this resource busy? High utilisation isn't always a problem, but at 100% there's no headroom left.

CPU: `top` → %CPU
Disk: `iostat -x` → %util
Network: `iftop` → bandwidth %

S — Saturation

Is the resource overloaded — are things queuing up waiting for it? Saturation means the resource can't keep up with demand, even if utilisation looks manageable.

CPU: load average above core count
Disk: `iostat` avgqu-sz > 1
Memory: swap in use and growing

E — Errors

Are there any error conditions being reported? Errors often explain performance problems that utilisation numbers miss entirely.

`dmesg -T | tail -20`
`journalctl -p err -n 50`
NIC: `ip -s link` → errors/dropped

> **Use the USE method as your opening move.** When a server is slow, check each major resource (CPU, memory, disk, network) against all three questions before diving into specifics. It prevents the trap of spending an hour optimising the wrong thing.

The Core Diagnostic Toolkit

These tools ship with or are available on almost every Linux distribution. Between them they cover the entire USE method across all major resources.

ESSENTIAL

`uptime`

Prints system uptime and the three load average figures. The fastest possible first check — single line, immediate output.

No flags needed. Also appears on the first line of `top` and `htop`. Alternative: `cat /proc/loadavg` for scripting.

ESSENTIAL

`htop`

Interactive process viewer with colour-coded bars, tree view, mouse support, and easy sorting/filtering. The best all-round first look at a system under load.

`htop` — open interactive view
`htop -u username` — filter by user
`htop -p PID` — watch a specific process
Keys: `F6` sort · `F4` filter · `F5` tree · `t` tree toggle · `k` send signal

ESSENTIAL

`top`

The classic real-time process viewer. Always available, no install needed. Less friendly than htop but works everywhere including minimal servers.

`top` — default interactive view
`top -b -n 1` — single snapshot (good for scripts)
Keys: `P` sort CPU · `M` sort memory · `1` show per-core · `k` kill · `q` quit

ESSENTIAL

`vmstat`

Reports on processes, memory, swap, I/O, interrupts, and CPU — all in one compact line updated at an interval. Excellent for spotting the relationship between resources.

`vmstat 1` — update every second
`vmstat 1 10` — 10 updates then stop
Key columns: `r` run queue · `b` blocked · `si/so` swap in/out · `wa` I/O wait

ESSENTIAL

`free`

Shows memory and swap usage. The `-h` flag makes it human-readable. Simple but often misread — see the memory chapter for why "available" is the number to watch, not "free".

`free -h` — human-readable (MB/GB)
`free -h -s 2` — refresh every 2 seconds
Watch: `available` column, not `free`. Watch `swap used` growing over time.

ESSENTIAL

`df`

Shows disk space usage per mounted filesystem. The first check for "disk full" problems. Don't forget to check inodes too.

`df -h` — human-readable sizes
`df -i` — inode usage (not space)
`df -hT` — include filesystem type
Red flag: any filesystem at 100% Use%.

USEFUL

`dmesg`

USEFUL

`iostat`

Prints the kernel ring buffer — messages from the kernel itself about hardware events, errors, and OOM kills. Invaluable for finding hardware problems and driver errors.

`dmesg -T` — include human-readable timestamps
`dmesg -T | tail -30` — recent kernel messages
`dmesg -T | grep -i error` — filter for errors
`dmesg -T | grep -i oom` — OOM killer events

Reports CPU statistics and I/O statistics for block devices. The go-to tool for identifying disk bottlenecks. Usually needs the `sysstat` package.

`iostat -x 1` — extended stats, every second
`iostat -xh 1` — human-readable sizes
Key columns: `%util` (disk busy %) · `await` (avg wait ms) · `r/s` and `w/s` (ops per sec)

USEFUL

sar

The historical performance recorder. Logs CPU, memory, disk and network metrics over time. Essential for answering "was it slow yesterday too?" Needs `sysstat` service running.

`sar -u 1 5` — CPU: 5 readings, 1 sec apart
`sar -r 1 5` — memory stats
`sar -b 1 5` — I/O stats
`sar -u -f /var/log/sysstat/sa14` — historical (day 14)

USEFUL

w / who

Shows who is logged in and what they're doing, plus the load average. Quick check for whether a human user is running something unexpected on the server.

`w` — logged-in users, their processes, load average
`who` — simpler: just login info
`last` — login history (who logged in when)
Useful when you see unexpectedly high CPU and want to know if a developer is running a build job.

Reading htop — What Every Column Means

htop is the most information-dense single screen you'll look at when diagnosing a performance issue. Here's what you're seeing:

htop — Annotated Layout

```
┌──────────────────────────────────────────────────────────────────────────┐
│  CPU[|||||||||||||||||||||||||            45.2%]   Tasks: 142, 312 thr; 2 running   │
│  CPU[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||| 98.0%]   Load average: 3.82 2.10 1.45
│  CPU[||||||                                        12.1%]   Uptime: 42 days, 03:21:07     │
│  CPU[||||                                           9.4%]                                 │
│  Mem[|||||||||||||||||||||||||||||||||||||         11.2G/16.0G]                           │
│  Swp[                                               0K/2.00G]                            │
├──────────────────────────────────────────────────────────────────────────┤
│  PID   USER     PRI  NI   VIRT    RES    SHR  S  CPU%  MEM%   TIME+   Command           │
│  4821  apache    20   0   520M    85M    18M  R  98.0   0.5  0:03.12  python3 worker.py  │
│  1234  mysql     20   0   2.5G   800M   200M  S   5.0   5.0 45:23.12  mysqld             │
│   891  root      20   0   185M    12M     9M  S   0.5   0.1  0:12.05  sshd               │
└──────────────────────────────────────────────────────────────────────────┘
 ▲ CPU bars: blue=normal · green=low priority · red=kernel · orange=IRQ
```

COLUMN	MEANING	WHAT TO WATCH FOR
`PID`	Process ID — unique number assigned by the kernel	Use this to target commands: `kill PID`, `strace -p PID`
`USER`	The user the process is running as	Unexpected users (root running web processes, or www-data running shells) are a red flag
`PRI / NI`	Priority and Nice value. NI ranges −20 (highest priority) to +19 (lowest)	NI = 0 is normal. Negative = high priority. Use `renice` to adjust
`VIRT`	Total virtual memory the process has mapped (includes shared libs, mmap'd files). Usually large, usually misleading.	Rarely the number to worry about
`RES`	Resident Set Size — RAM actually in physical memory right now. The real memory figure.	This is the one to watch. A process with 8GB RES is using 8GB of RAM
`SHR`	Shared memory — portion of RES that's shared with other processes (shared libs)	Real exclusive memory ≈ RES − SHR
`S`	Process state: **R** running, **S** sleeping (interruptible), **D** sleeping (uninterruptible/disk wait), **Z** zombie, **T** stopped	Many **D** state processes = disk or NFS problem. **Z** zombies = a parent isn't reaping children
`CPU%`	CPU usage over the last sampling interval, across all cores. Can exceed 100% on multi-core systems (e.g. 800% = 8 full cores)	A single process at 100% is using one full core. Look for processes unexpectedly high
`MEM%`	Percentage of total physical RAM used by this process (based on RES)	Quick relative view. Absolute RES figure is more useful
`TIME+`	Total CPU time consumed since the process started (not wall clock time)	A process that's been running 3 minutes but shows 2h45m of CPU time is CPU-intensive
`Command`	The command that launched the process	Press `F5` in htop to switch to tree view — shows parent/child relationships

> **htop colour legend (CPU bars):** Blue = normal user processes. Green = low-priority (niced) processes. Red = kernel threads. Orange = hardware interrupts (IRQ). A mostly-red CPU bar means the kernel itself is very busy — often a driver issue, not an application problem.

vmstat — Seeing Everything at Once

`vmstat 1` gives a continuous stream of system-wide stats that lets you see how resources are moving together. The first line after the header is an average since boot — ignore it. Watch the subsequent lines.

```
$ vmstat 1
#procs  --------memory(KB)-------- --swap-- ---io--- -system-- ------cpu-----
 r  b    swpd   free    buff   cache   si   so    bi    bo   in    cs us sy id wa st
 1  0       0 4823400 201480 8432100    0    0     0    12  312  1821 12  3 84  1  0
 6  2       0 4012300 201480 8432100    0    0     0  8420 2100  4200 45 12 40  3  0
 2  8    2048 312000  201480 8432100 1200 800  9800  8420 3100  5200 20  8 12 60  0
```

COLUMN	MEANING	ALERT WHEN…
r	Processes runnable (waiting for CPU time)	Consistently above number of CPUs = CPU saturation
b	Processes in uninterruptible sleep (blocked on I/O)	Greater than 2–3 sustained = disk or NFS problem
swpd	Swap space in use (KB)	Any value above 0 when memory should be adequate
si / so	Swap in / Swap out (KB/sec)	Any non-zero sustained value = active swap thrashing
bi / bo	Block in / Block out (blocks/sec) — disk reads/writes	Very high sustained bi or bo = disk-heavy workload
wa	% of time CPU spent waiting for I/O	Above 10–15% sustained = I/O is slowing things down
us / sy	User CPU% / System (kernel) CPU%	High sy (kernel%) = driver issue, syscall-heavy app, NFS
id	CPU idle %	If this is high but the server feels slow, the problem is elsewhere (disk, network, lock contention)

Scenario — "The Server Feels Slow"

A user contacts you: the web application is responding slowly. Nobody has deployed anything recently. You have SSH access. Here's the systematic first-response approach.

SCENARIO · CHAPTER 1

Unknown slowness — applying the USE method from scratch

1. **Get the overview in 5 seconds.** Run `uptime` the moment you log in. Note the load average and how it compares to the CPU count (`nproc`).

   Load 0.9 on 8 cores → CPU is idle, look elsewhere. Load 6.2 on 8 cores → moderate, dig into CPU. Load 24 on 8 cores → severe saturation, act quickly.

2. **Check for kernel errors immediately.** `dmesg -T | tail -20` . Look for OOM kills, disk errors (I/O errors, sector failures), or driver warnings. If the kernel is reporting errors, that's often the root cause before you look anywhere else.

3. **Open htop and sort by CPU%.** Press `F6` → CPU%. Is one process consuming an unusual amount? Look at the state column — lots of `D` (uninterruptible) processes means disk or NFS. Lots of `R` means CPU contention. Check if the top processes make sense (your app, database) or are unexpected (a cron job, a compiler, a rogue script).

4. **Sort htop by MEM% (press F6 → MEM%).** Is any process consuming a disproportionate amount of RAM? Check whether the system is using swap: the `Swp` bar at the top of htop shows swap usage. Any swap in use on a server that should have enough RAM is a signal.

5. **Run `vmstat 1` for 10 seconds.** Watch the `wa` (I/O wait) column. If it's consistently above 10%, the bottleneck is disk — move to the disk chapter's tools (`iostat -x 1` , `iotop`). If `si` / `so` (swap in/out) are non-zero, the bottleneck is memory.

6. **Check disk space.** `df -h` — any filesystem at 100% will cause write failures and very strange application behaviour that looks like a performance problem but is actually a hard error. Also run `df -i` to check inodes separately.

7. **Check who else is on the server.** `w` shows logged-in users and what they're running. A developer running a `find / -name "*.log"` or a database dump at the wrong time can saturate I/O for everyone else. Friendly conversation often resolves this faster than tuning.

8. **Identify the bottleneck type and move to the right chapter.** By this point you should know roughly what you're dealing with:

   High CPU% → Chapter 2 (CPU) · High MEM / swap activity → Chapter 3 (Memory) · High wa / disk errors → Chapter 4 (Disk) · All normal here but app slow → Chapter 5 (Network)

> **The diagnostic flow as a one-liner sequence:** `uptime` → `dmesg -T | tail -20` → `htop` (CPU sort, then MEM sort) → `vmstat 1` (10 seconds) → `df -h && df -i` → `w`. Run these in order and you'll have a confident hypothesis within 3 minutes of logging in.

Quick Reference — Chapter 1 Commands

COMMAND	PURPOSE	KEY FLAGS
`uptime`	Load average + uptime at a glance	None needed
`nproc`	Number of logical CPU cores (context for load average)	None needed
`htop`	Interactive process viewer — best all-round first look	`-u user` · `-p PID` · F5 tree · F6 sort · F4 filter
`top`	Classic process viewer — always available	`-b -n 1` (snapshot) · `P` sort CPU · `M` sort mem · `1` per-core
`vmstat 1`	System-wide stats: CPU, memory, swap, I/O, all in one	`vmstat 1 10` — 10 readings then stop
`free -h`	Memory and swap usage in human-readable form	`-s 2` refresh every 2s · watch the "available" column
`df -h`	Disk space per mounted filesystem	`-i` inode usage · `-T` include FS type
`dmesg -T`	Kernel messages — OOM kills, hardware errors, driver issues	`| tail -30` · `| grep -i oom` · `| grep -i error`
`iostat -x 1`	Disk I/O stats per device (needs sysstat package)	`-h` human-readable · watch `%util` and `await`
`sar -u 1 5`	CPU history — 5 readings, 1 second apart (needs sysstat)	`-r` memory · `-b` I/O · `-f /var/log/.../saNN` historical
`w`	Logged-in users + what they're running + load average	No flags. `last` for login history

CHAPTER 2 OF 8

Chapter 2 — CPU Bottlenecks

High CPU load is one of the most visible performance problems — the fans spin up, response times climb, and htop fills with red bars. But not all CPU consumption is a problem. A machine compiling code or transcoding video at 100% is doing exactly what it should. This chapter is about telling the two apart, and acting safely when intervention is needed.

> **What this chapter covers:** CPU utilisation vs load average — why they differ. Per-core breakdown with mpstat. Finding CPU-hungry processes. Scenario 1: dozens of the same process — how to count, identify, and safely resolve. Scenario 2: one process at 100% — runaway vs legitimate. Zombie processes. Adjusting priority with nice/renice. CPU affinity with taskset. Context switching and CPU steal.

CPU Utilisation vs Load Average — The Key Difference

Chapter 1 covered load average. CPU utilisation is a different measurement — and the relationship between them tells you a great deal about what's actually happening.

📊

CPU Utilisation (%)

What percentage of CPU time is being used right now. Shown in `top` and `htop` as `%Cpu(s): us sy id wa`. A value of 100% means a core is fully occupied — it cannot give more.

⚖️

Load Average

Average number of processes wanting to run (or blocked on I/O) over 1, 5, and 15 minutes. Includes I/O-waiting processes that use *no CPU at all*. Load can be high while CPU is idle.

🔍

The telling combination

High load + high CPU% → CPU bottleneck.
High load + low CPU% + high wa → disk/I/O bottleneck, not CPU.
Low load + slow app → look at network or locking.

The CPU utilisation breakdown in top

When you press `1` in `top` or look at the per-core bars in htop, you see multiple CPU components. Understanding these tells you where CPU time is going:

```
$ top  # then press 1 to show per-core
%Cpu0 : 45.2 us,  12.1 sy,  0.0 ni, 40.2 id,  2.1 wa,  0.0 hi,  0.4 si,  0.0 st
%Cpu1 :  2.1 us,   1.0 sy,  0.0 ni, 96.0 id,  0.5 wa,  0.0 hi,  0.4 si,  0.0 st
%Cpu2 : 98.0 us,  1.5 sy,  0.0 ni,  0.0 id,  0.5 wa,  0.0 hi,  0.0 si,  0.0 st
# Cpu2 at 98% user — one process is saturating a single core
# Cpu0 shows 12.1% kernel (sy) — unusually high, often a driver or syscall issue
```

FIELD	NAME	WHAT IT MEANS	ALERT WHEN…
us	User	CPU time in user-space processes (your applications)	High is normal for CPU-bound apps; watch for unexpected processes
sy	System	CPU time in kernel-space (system calls, drivers)	Above 20% sustained — driver issue, NFS, or syscall-heavy app
ni	Nice	CPU time used by niced (low-priority) processes	Rarely a concern on its own
id	Idle	CPU time doing nothing	If high but system is slow, the bottleneck is elsewhere (disk, network)
wa	I/O wait	CPU idle while waiting for disk I/O to complete	Above 10–15% sustained — disk bottleneck, not CPU
hi	Hardware IRQ	Servicing hardware interrupts (NIC packets, disk)	Above 5% — very high network or disk interrupt rate
si	Software IRQ	Kernel software interrupt processing	High on a busy network server — normal; high on a quiet box — investigate
st	Steal	CPU time stolen by the hypervisor (VMs only)	Above 5% on a VM — your host is overloaded; contact your cloud provider

Process States — The S Column in htop

Every process is always in one of a small number of states. The **S** column in htop and `ps` tells you the current state — and the pattern of states across all your processes reveals what the system is waiting on.

R

Running / Runnable

Process is either on a CPU right now or queued waiting for one. Many R-state processes = CPU saturation.

S

Sleeping (interruptible)

Process is waiting for an event (keyboard input, network data, a timer). Will wake immediately when the event arrives. Normal and healthy.

D

Disk Wait (uninterruptible)

Blocked on I/O — disk read, NFS response, or similar. Cannot be interrupted or killed until the I/O completes. Many D-state processes = I/O bottleneck.

Z

T

Zombie

Process has exited but its parent hasn't called wait() to collect the exit code. Uses no CPU or memory. Kill -9 does nothing — the fix is in the parent.

Stopped

Suspended by SIGSTOP or Ctrl+Z. Will not run until it receives SIGCONT. Sometimes left behind by debug sessions or background jobs.

> **The D-state trap:** A process in D state cannot be killed. It holds its PID, any file locks it owns, and any resources it had open — and it will stay that way until the I/O it's waiting on completes (or times out). If you have many D-state processes, the disk or NFS mount is the problem, not the processes. Fix the I/O, and they'll wake up.

Per-Core Breakdown with mpstat

htop shows per-core bars visually. `mpstat` gives you the numbers — useful when you want to identify which specific core is being saturated, or when you're working over SSH without a graphical display.

```
$ mpstat -P ALL 1 3
# -P ALL = show all CPUs, 1 = every second, 3 = three readings

14:35:01  CPU   %usr  %nice  %sys %iowait  %irq  %soft  %steal  %idle
14:35:02  all  35.2    0.0   3.1     8.2   0.0    0.3     0.0   53.2
14:35:02    0  12.0    0.0   2.1    18.0   0.0    0.2     0.0   67.7
14:35:02    1  98.0    0.0   1.5     0.0   0.0    0.0     0.0    0.5
14:35:02    2   8.0    0.0   3.2    12.0   0.0    0.4     0.0   76.4
14:35:02    3   2.8    0.0   6.1     2.4   0.0    0.6     0.0   88.1
# CPU 1 is at 98% user — one single-threaded process is saturating it entirely
# The 'all' line averages to ~35% — masking the real problem if you only look there
```

> **Always look at per-core stats, not just the average.** A single-threaded runaway process on a 16-core machine will show as 6% average CPU load — completely invisible in the top-line average — while completely saturating one core. `mpstat -P ALL 1` or pressing `1` in top/htop reveals it instantly.

Finding CPU-Hungry Processes

Once you know the system is CPU-bound, the next step is identifying which process is responsible. Several tools approach this from different angles:

```
# Sort ps output by CPU descending — snapshot, not live
$ ps aux --sort=-%cpu | head -10
USER       PID %CPU %MEM    VSZ   RSS  STAT START   TIME COMMAND
www-data  4821 97.8  0.5  520000 85000 R    14:12   3:42 python3 /opt/app/worker.py
mysql     1234  4.2  5.1 2500000 840000 S   09:00  45:23 mysqld

# How long has the top process been running at this CPU%?
$ ps -p 4821 -o pid,etime,pcpu,pmem,comm
  PID ELAPSED %CPU %MEM COMMAND
 4821   00:03  97.8  0.5 python3
# Running for 3 minutes. TIME column in htop shows 3:42 of CPU time consumed.
# This matches — it's been using CPU the entire time it's been running.

# Alternative: watch a specific process live
$ top -p 4821
```

```
# pidstat — per-process CPU history (from sysstat package)
$ pidstat -u 1 5
14:35:01  UID  PID   %usr %system  %CPU CPU  Command
14:35:02 1000 4821  97.0    0.5  97.5   1  python3
14:35:03 1000 4821  98.0    0.3  98.3   1  python3
# CPU column shows which core — confirms CPU 1 as seen in mpstat
# %usr vs %system breakdown: 97% user means it's the app code, not the kernel
```

Scenario 1 — Dozens of the Same Process

SCENARIO · CHAPTER 2 · SCENARIO 1

htop shows 40+ "python3" processes — are they normal workers or a runaway?

1. **Count exactly how many there are.** The number matters — 8 Python workers for a web app is normal; 400 is not.

```
$ pgrep -c python3
47
# -c = count. 47 python3 processes running right now.

$ pgrep -c -u www-data python3
47
# All 47 belong to www-data — likely your web app user, not root
```

2. **Find out what they're actually running** — are they all the same script, or different things?

```
$ ps aux | grep python3 | grep -v grep | awk '{print $NF}' | sort | uniq -c | sort -rn
     47 /opt/app/worker.py
# All 47 are the same script — this is a worker pool pattern
# uniq -c counts duplicates; sort -rn puts highest count first

# Or see the full command line with arguments:
$ ps aux | grep python3 | grep -v grep | head -5
www-data  4821 12.1  0.5  520M  85M S 14:12 0:03 python3 /opt/app/worker.py --queue celery
www-data  4822  9.8  0.5  518M  83M S 14:12 0:02 python3 /opt/app/worker.py --queue celery
```

3. **Find the parent process** — who spawned all these workers? The parent's PID is the key to controlling the pool.

```
$ ps aux --forest | grep -A 50 celery | head -20
www-data  4800  0.5  0.3  200M  45M S 14:11 0:01 celery worker -A myapp --concurrency=50
www-data  4821 12.1  0.5  520M  85M S 14:12 0:03  \_ python3 /opt/app/worker.py
www-data  4822  9.8  0.5  518M  83M S 14:12 0:02  \_ python3 /opt/app/worker.py
# Parent is PID 4800 (celery) with --concurrency=50 — that explains 47 workers
# --forest shows the tree. This is normal behaviour for this config.

# Alternative: find the parent PID of any child process
$ ps -p 4821 -o ppid=
4800
```

4. **Decide: is this a normal worker pool or a runaway?**

Normal signs: all workers belong to one parent, the parent is a known service (celery, gunicorn, uwsgi, apache), the count matches the configured concurrency level, workers are in S state (sleeping, waiting for work).

Runaway signs: workers are all in R state simultaneously, the count is growing over time (`watch -n 1 'pgrep -c python3'`), there's no obvious parent managing them, or the parent is a plain shell script.

5. **If it's a runaway — stop the parent first, not the children.** Killing children while the parent is alive just causes it to respawn them.

```
# SIGTERM the parent — lets it shut down gracefully and clean up children
$ kill 4800
# Wait 10-15 seconds. Verify it's gone:
$ pgrep -c python3
0

# If SIGTERM is ignored after 15 seconds, escalate to SIGKILL:
$ kill -9 4800
# Then clean up orphaned children if any remain:
$ pkill -9 -u www-data python3
```

6 **If it's a legitimate pool that's overloaded** — don't kill it. Reduce the concurrency setting in the service configuration (e.g., celery's `--concurrency`, gunicorn's `--workers`) and reload the service. Killing workers under load will drop in-flight requests and may corrupt queued tasks.

If you're unsure whether the process count is normal, check the service's documentation or configuration file for its concurrency setting. A well-configured service should document how many workers it's expected to spawn.

Scenario 2 — One Process at 100% CPU

SCENARIO · CHAPTER 2 · SCENARIO 2

A single process is pegged at 100% — runaway loop or legitimate work?

1. **Find the process and note its PID, user, and command.**

```
$ ps aux --sort=-%cpu | head -3
USER       PID  %CPU %MEM    VSZ   RSS  STAT START   TIME COMMAND
deploy    9142  99.5  0.8 180000 32000 R    15:04   8:32 /usr/bin/python3 -c "while True: pass"
# TIME shows 8:32 of accumulated CPU time. ELAPSED would confirm wall-clock duration.
```

2. **Check elapsed time vs CPU time** — they should roughly match for a legitimate job.

```
$ ps -p 9142 -o pid,etime,cputime,pcpu,comm
  PID     ELAPSED   TIME   %CPU COMMAND
 9142   00:08:41 00:08:32  99.5 python3
# Elapsed: 8m41s. CPU time: 8m32s. Near-identical — this process has been using
# 100% CPU for almost its entire lifetime. Very likely an infinite loop.

# For a legitimate job (e.g. video encoding running 1 hour):
 9000   01:02:15 01:00:44  97.3 ffmpeg
# Also near 100%, but context (ffmpeg, known job) makes this expected.
```

3. **Attach strace to see what the process is doing right now** — is it making system calls (doing real work) or spinning with no system calls (tight loop)?

```
$ strace -p 9142 -c -e trace=all
# Run for 5 seconds then Ctrl+C to see the summary
strace: Process 9142 attached
^C
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- --------
  0.00    0.000000           0         2           read
  0.00    0.000000           0         1           write
# Only 3 syscalls in 5 seconds — the process is running pure user-space code
# in a tight loop. Nothing is being read, written, or computed productively.
# This is a runaway.

# A legitimate CPU-bound job shows many syscalls:
  99.8    8.432100        1234      6832           read   ← reading input data
   0.1    0.012000          80       150           write  ← writing output
```

4. **If it's a legitimate but disruptive job — reduce its priority with renice** instead of killing it. This lets the job complete while giving other processes more CPU time.

```
# Renice to +10 (lower priority — other processes get CPU preference)
$ renice +10 -p 9142
9142 (process ID) old priority 0, new priority 10
# The process still runs, but the scheduler deprioritises it when others need CPU.
# Verify in htop — it will show NI column as 10.
```

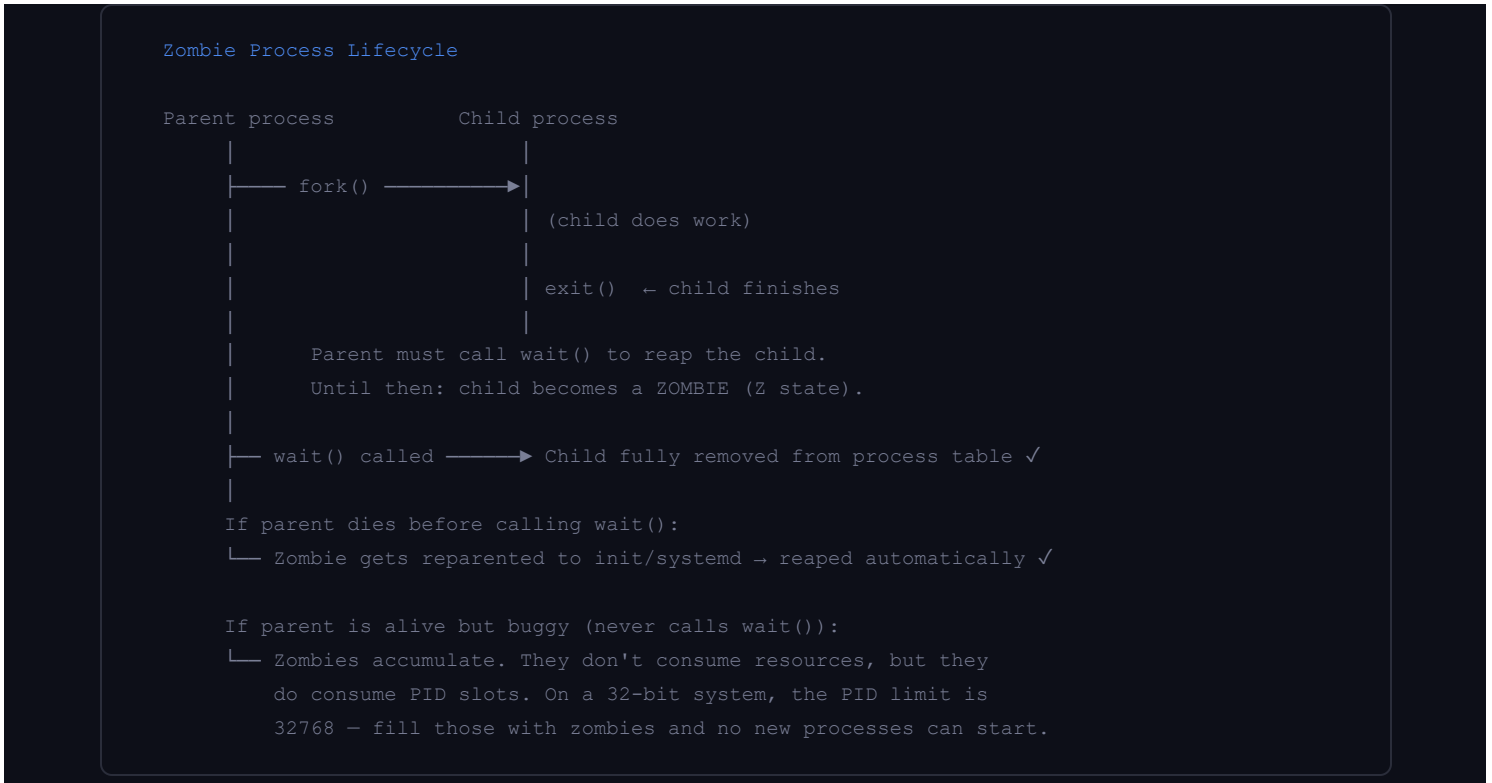
5 **If it's genuinely a runaway — send SIGTERM first, then SIGKILL.** Always try SIGTERM first — it lets the process clean up (close files, flush buffers, release locks). Only use SIGKILL if SIGTERM is ignored.

```
$ kill 9142              # SIGTERM — please shut down gracefully
$ sleep 5 && ps -p 9142  # Wait 5 seconds and check if it's gone
  PID TTY      STAT   TIME COMMAND
 9142 pts/0    R      8:41 python3  ← still running after 5s
$ kill -9 9142           # SIGKILL — forced termination, no cleanup
$ ps -p 9142
  PID TTY      STAT   TIME COMMAND
                              ← empty, process is gone
```

Before killing any unfamiliar process, check what it is: `ls -la /proc/9142/exe` shows the real binary path, and `cat /proc/9142/cmdline | tr '\0' ' '` shows the full command including all arguments.

Zombie Processes

A zombie (state Z) is a process that has finished executing but hasn't been fully removed from the process table because its parent hasn't called `wait()` to collect its exit status. The zombie itself is harmless — it uses no CPU, no memory, and no file descriptors. It occupies only a PID slot.

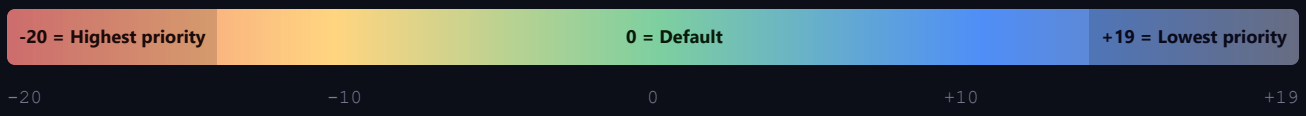



```
# Find zombie processes
$ ps aux | awk '$8=="Z" {print $0}'
www-data 12041  0.0  0.0      0     0 Z    15:32   0:00 [defunct]
www-data 12042  0.0  0.0      0     0 Z    15:32   0:00 [defunct]

# How many zombies total?
$ ps aux | awk '$8=="Z"' | wc -l
2

# Find the parent of a zombie (PPID column)
$ ps -p 12041 -o ppid=
12000

# 1-2 zombies: normal, ignore them.
# 50+ zombies and growing: bug in the parent process (PID 12000).
# Fix: restart the parent — that clears its zombie children.

# CANNOT kill a zombie with kill -9. It's already dead.
# Kill -9 the PARENT to clear zombies (zombies reparent to init which reaps them)
$ kill 12000   # SIGTERM parent first
```

> **1–5 zombies on a busy server is normal** — they typically clear within a few seconds as the parent gets around to calling `wait()`. Only investigate if the count is large and stable (the parent is stuck), or if it's growing steadily (a bug causing the parent to never reap its children).

Adjusting Priority — nice and renice

The **nice value** tells the Linux scheduler how much to favour or deprioritise a process when competing for CPU time. It does not limit CPU usage — on an otherwise idle machine, a niced process still runs at full speed. The effect is only felt when there's competition.




```
# Launch a new process at low priority (nice +10)
$ nice -n 10 python3 heavy_script.py

# Launch at very low priority — use for batch jobs, backups, builds
$ nice -n 19 make -j8

# Adjust priority of an already-running process (renice)
$ renice +10 -p 4821        # lower priority of PID 4821
$ renice +10 -u www-data    # lower priority of ALL processes by www-data
$ renice -5  -p 4821        # raise priority (requires root)

# Verify the change in ps:
$ ps -p 4821 -o pid,ni,comm
  PID  NI COMMAND
 4821  10 python3
```

> **Only root can set negative nice values** (raising priority above default). Any user can lower their own processes' priority (raise the nice value toward +19). You cannot renice another user's processes without root.

CPU Affinity — taskset

By default the kernel scheduler can run any process on any available CPU core. **CPU affinity** lets you pin a process to specific cores — useful when you want to isolate a CPU-intensive job so it doesn't interfere with latency-sensitive processes on other cores.

```
# Launch a process pinned to CPU core 0 only
$ taskset -c 0 python3 heavy_script.py

# Pin to cores 0 and 1 only (comma-separated, or range with dash)
$ taskset -c 0,1 python3 heavy_script.py
$ taskset -c 0-3 make -j4    # use first 4 cores for a build

# Change affinity of a running process (PID 4821)
$ taskset -cp 2,3 4821
pid 4821's current affinity list: 0-7
pid 4821's new affinity list: 2,3

# Check current affinity of a running process
$ taskset -cp 4821
pid 4821's current affinity list: 2,3
```

> **Practical use case:** You have a production web server on cores 0–3 and want to run a CPU-intensive batch job without it competing for those cores. Pin the batch job to cores 4–7 with `taskset -c 4-7 batch_job`. The web server stays responsive on its own cores regardless of what the batch job does.

Context Switching

Every time the kernel switches from running one process to another, it performs a **context switch** — saving the outgoing process's state and loading the incoming one's. Context switches are cheap individually but add up: a system with thousands of context switches per second per core is spending meaningful time on overhead rather than useful work.

```
# vmstat cs column shows context switches per second
$ vmstat 1 5
 r  b  swpd  free   buff  cache  si  so  bi  bo  in   cs  us sy id wa
 2  0     0 4.1G  201M   8.2G   0   0   0  12 412 1821  12  3 84  1
 8  0     0 4.0G  201M   8.2G   0   0   0   8 890 9200  45  8 46  1
12  0     0 3.9G  201M   8.2G   0   0   0  14 1100 24000 60 15 24  1
# cs (context switches) jumping to 24,000/sec with r=12 waiting processes
# This is CPU saturation — too many threads fighting for too few cores

# Per-process context switching with pidstat
$ pidstat -w -p 4821 1 5
14:42:01   PID   cswch/s  nvcswch/s  Command
14:42:02 4821    240.0     8120.0  python3
# cswch/s  = voluntary context switches (process yielded the CPU, e.g. waiting for I/O — healthy)
# nvcswch/s = non-voluntary (kernel forced it off because its time slice expired — high = contenti
```

Kill vs Throttle — Choosing the Right Response

THROTTLE FIRST (RENICE / TASKSET)

- → The process is doing legitimate work (backup, build, encoding)
- → Killing it would mean losing progress or corrupting output
- → The process belongs to a production service that must keep running
- → The problem is timing — the job ran at peak hours when it shouldn't
- → The impact is "annoying but manageable" — not an outage
- → You can talk to the person who started it and co-ordinate

KILL (AFTER SIGTERM → SIGKILL)

- → The process is confirmed to be an infinite loop / runaway
- → It is actively causing an outage for other users or services
- → strace shows no productive work — pure CPU spin
- → The process belongs to a test/dev environment, not production
- → It's already consuming all available CPU and growing
- → The process is owned by a user who isn't reachable for co-ordination

Always SIGTERM before SIGKILL. SIGTERM gives the process a chance to close files, flush buffers, release database connections, and delete temp files. SIGKILL bypasses all of that — it's the equivalent of pulling the power cable. For a database, SIGKILL can mean corruption. For a web service, it means dropped connections. Reserve it for when SIGTERM has demonstrably failed.

Quick Reference — Chapter 2 Commands

COMMAND	PURPOSE	KEY FLAGS / NOTES
`mpstat -P ALL 1`	Per-core CPU breakdown every second	Press `1` in top/htop for the same view interactively
`ps aux --sort=-%cpu`	Snapshot of all processes sorted by CPU usage (highest first)	`| head -10` to limit output
`ps aux --forest`	Process tree — shows parent/child relationships	Combine with `| grep -A 20 processname`
`ps -p PID -o etime,cputime,pcpu,comm`	Elapsed time vs accumulated CPU time for one process	Match these two to spot runaway processes
`pgrep -c name`	Count processes matching name	`-u user` filter by user · `-l` list PIDs
`pgrep -a name`	List PIDs and full command lines matching name	Better than `ps | grep` — no grep process in results
`strace -p PID -c`	Attach to running process, summarise system calls	Run for 5–10 seconds then Ctrl+C for the summary
`pidstat -u 1`	Per-process CPU stats updated every second	`-w` for context switches · `-p PID` for one process
`renice +10 -p PID`	Reduce a running process's CPU priority	Range +1 to +19 lowers priority; negative values need root
`nice -n 10 cmd`	Launch a command at reduced priority	`-n 19` for lowest priority batch jobs
`taskset -c 0,1 cmd`	Launch command pinned to specific CPU cores	`taskset -cp 2,3 PID` to change affinity of running process
`kill PID`	Send SIGTERM — graceful shutdown request	Always try this first. Wait 10–15 seconds before escalating.
`kill -9 PID`	Send SIGKILL — forced, immediate termination	Last resort. No cleanup. Can't be caught or ignored.
`pkill -P PID`	Send SIGTERM to all children of a parent process	`-9` for SIGKILL · `-u user name` by user and name
`ps aux | awk '$8=="Z"'`	List zombie processes	Find parent with `ps -p ZOMBIEPID -o ppid=` then restart parent

CHAPTER 3 OF 8

Chapter 3 — Memory Bottlenecks

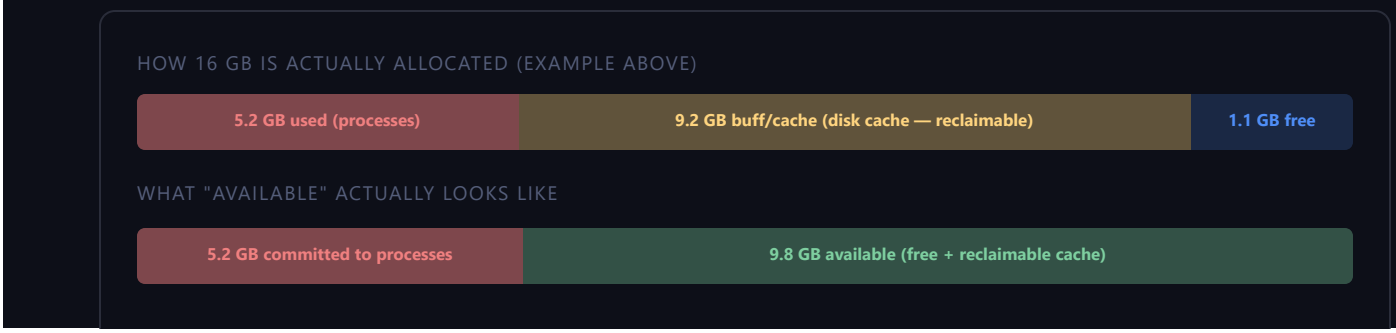
Memory is the most misread resource on Linux. The default output of `free` routinely alarms people who see only 200 MB "free" on a 16 GB server — when in reality that server has 14 GB readily available and is performing perfectly. This chapter starts by correcting that misreading, then covers the cases where memory really is a problem: swap thrashing, OOM kills, and memory leaks.

> **What this chapter covers:** Why "free" memory isn't what it looks like. The available vs free distinction. Buffers and page cache explained. Reading /proc/meminfo. Swap and vm.swappiness. Scenario 1: swap climbing overnight. Scenario 2: the OOM killer fired — reading dmesg. Scenario 3: a service's memory keeps growing — identifying a leak. The OOM score system. When (and when not) to clear the page cache.

The Fundamental Misreading — "Free" Is Not "Available"

Linux uses all available RAM productively. Memory that isn't being used by processes is used as a disk cache — so pages read from disk recently are kept in RAM and served from there on subsequent reads, making the system faster. This cached memory is immediately reclaimed when a process needs it. It shows up as "used" in many tools, but it's not really in use in a way that matters.

```
$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi        5.2Gi       1.1Gi       342Mi       9.2Gi       9.8Gi
Swap:          2.0Gi          0B       2.0Gi
# "used" looks high at 5.2 GB — but that includes process memory AND some overhead
# "free" is only 1.1 GB — looks alarming
# "buff/cache" is 9.2 GB of disk cache — reclaimed whenever processes need it
# "available" is 9.8 GB — what a new process can actually use without going to swap
# The server has 9.8 GB readily available. It is not in trouble.
```



> **The number to watch is "available", not "free".** When available drops toward zero (especially below 200–300 MB on a production server), then you have a memory pressure problem. Until then, the kernel is just being efficient with your RAM.

What are buffers and cache?

Page Cache

Contents of files that have been read from disk, kept in RAM for fast re-access. If your app reads the same log file 1,000 times, after the first read it comes from RAM at memory speed. The kernel evicts oldest pages when processes need more RAM. **This is the majority of "buff/cache" on a typical server.**

Buffer Cache

Filesystem metadata: directory entries (dentries), inode information, and raw block device buffers. Smaller than the page cache on most systems, but important for workloads that do many small file operations (millions of tiny files, email servers).

Anonymous Memory

Process heap, stack, and privately allocated memory that isn't backed by a file. This is the memory processes actually "own" and that **cannot** be reclaimed without going to swap. It's what RSS measures. `free` calls this the "used" column (roughly).

/proc/meminfo — The Full Picture

`free` is a summary. When you need more detail, `/proc/meminfo` is the authoritative source — the kernel's own memory accounting. The fields that matter most for performance diagnosis:

```
$ cat /proc/meminfo
MemTotal:       16384000 kB   # Total physical RAM
MemFree:         1126400 kB   # Truly unused pages
MemAvailable:   10035200 kB   # ← The real "can I fit more?" number
Buffers:          206080 kB   # Block device buffer cache
Cached:          9214976 kB   # Page cache (file contents)
SwapCached:            0 kB   # Pages in swap that are also still in RAM
SwapTotal:       2097152 kB
SwapFree:        2097152 kB   # ← Swap used = SwapTotal - SwapFree. Here: 0 used.
Active:          6291456 kB   # Recently used — less likely to be reclaimed
Inactive:        3670016 kB   # Less recently used — first candidates for reclaim
Dirty:             49152 kB   # Written to but not yet flushed to disk
Writeback:             0 kB   # Currently being written to disk
AnonPages:       5324800 kB   # Anonymous process memory (heap/stack) — cannot be reclaimed
Mapped:          1638400 kB   # Files mapped into process address spaces (mmap)
Shmem:            350208 kB   # Shared memory (tmpfs, IPC)
Slab:             524288 kB   # Kernel data structures (inodes, dentries, etc.)
SReclaimable:     393216 kB   # Slab memory that can be reclaimed
CommitLimit:    10289152 kB   # Maximum memory the kernel will commit to (overcommit limit)
Committed_AS:    8192000 kB   # How much has been promised. If near CommitLimit, risk of OOM.
```

FIELD	WHAT IT MEANS	ALERT WHEN…
`MemAvailable`	Memory a new process can use without swap. The most useful single number.	Drops below ~200 MB on a production server
`SwapFree`	Unused swap. Swap used = SwapTotal − SwapFree.	Swap in use at all — investigate why
`Dirty`	Data written by processes but not yet flushed to disk. Normal in bursts.	Persistently high (GB range) — disk can't keep up with writes
`AnonPages`	Process heap/stack memory. Cannot be reclaimed without swap.	Growing steadily with no new processes starting — possible leak
`Committed_AS`	Total memory promised to all processes (including not-yet-used allocations).	Approaching CommitLimit — system is over-committed, OOM risk
`Slab`	Kernel object caches. Can grow large on servers with many files/sockets.	Several GB with nothing to explain it — possible kernel memory leak

Swap — What It Is and When It Becomes a Problem

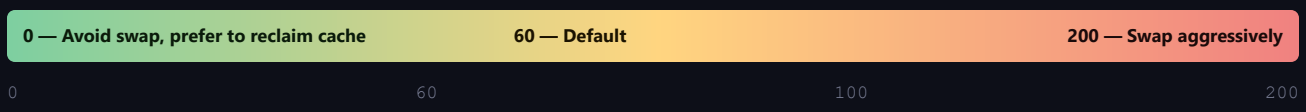
Swap is disk space used as overflow when physical RAM is exhausted. When the kernel needs to free RAM for a new allocation and can't reclaim enough page cache, it evicts anonymous process memory (heap/stack pages) to the swap device. If that memory is needed again, it's read back from disk — this is called **swapping in**.

The problem: modern NVMe SSDs deliver ~7 GB/s. RAM delivers ~50 GB/s. Even on the fastest storage, swap I/O is 7× slower than RAM at best — and swap on a spinning HDD is catastrophic. A system actively swapping

(non-zero si/so in vmstat) will feel sluggish even if CPU load is low.

```
# Quick swap check
$ free -h | grep Swap
Swap:          2.0Gi       1.4Gi       614Mi
# 1.4 GB in use. Now check if it's actively swapping (worse than just being in use):

$ vmstat 1 5 | awk 'NR==1{print} NR>2{print}'
 r  b  swpd    free   buff  cache   si   so  bi  bo  in   cs  us sy id wa
 3  1  1433M  614M  201M  2.1G  840  320  9800 8420 3100 5200 20  8 12 60
# si=840 KB/s (swapping in) + so=320 KB/s (swapping out) = active thrashing
# wa=60% — CPU spending 60% of time waiting for disk I/O from swap
```

vm.swappiness — controlling swap eagerness




```
# Check current swappiness
$ cat /proc/sys/vm/swappiness
60

# Reduce it temporarily (survives until next reboot)
$ sysctl -w vm.swappiness=10

# Make it permanent across reboots:
$ echo "vm.swappiness=10" | sudo tee /etc/sysctl.d/99-swappiness.conf
$ sysctl -p /etc/sysctl.d/99-swappiness.conf
```

> **vm.swappiness=0 does not disable swap.** It tells the kernel to avoid swapping anonymous memory and prefer to reclaim page cache instead — but the kernel will still swap if there's genuinely no other choice. Setting it to 0 can actually cause OOM kills on workloads where swapping out cold pages would have been better. A value of 10 is a reasonable middle ground for servers that have enough RAM.

Scenario 1 — Swap Is Climbing Overnight

SCENARIO · CHAPTER 3 · SCENARIO 1

Morning check shows swap at 1.8 GB used. Yesterday it was 0. What's consuming RAM?

1 **Confirm swap is in use and check whether it's actively swapping.** Used swap alone isn't urgent. Active swapping (si/so in vmstat) is.

```
$ free -h
Swap:   2.0Gi   1.8Gi   204Mi   # 1.8 GB used — significant

$ vmstat 1 3 | awk 'NR==1{print} NR>2{print}'
 r  b  swpd  free  buff  cache  si   so   bi  bo  in   cs
 1  0  1843M  204M  180M  420M   0    0   12   8 210 840
# si=0, so=0 — not actively swapping right now. Memory was pushed to swap
# overnight but the system is stable. Now find what consumed the RAM.
```

2 **Find the memory hogs with ps, sorted by RSS (resident memory).**

```
$ ps aux --sort=-%mem | head -10
USER       PID   %CPU %MEM    VSZ     RSS  STAT  START    TIME  COMMAND
postgres  2341   0.5  18.4  850000  3014656  S    Mon09   8:42  postgres: worker
app       8821   0.2  12.1  650000  1982464  S    Mon09   3:14  node /opt/app/server.js
app       8834   0.1  11.8  640000  1933312  S    Mon09   3:05  node /opt/app/server.js
# RSS is in KB. 3,014,656 KB = ~2.9 GB for postgres worker
# Two node processes each using ~1.9 GB
# Total: ~6.7 GB — this server only has 16 GB, so that explains the swap pressure

# Human-readable RSS:
$ ps aux --sort=-%mem | awk 'NR==1 || NR<=11 {printf "%-12s %6s %6.1f MB  %s\n", $1, $2, $6/1024, $11}'
```

3 **Use smem for a more accurate per-process view** — it shows Unique Set Size (USS), which excludes memory shared with other processes and gives the real exclusive memory cost.

```
# smem may need: apt install smem / yum install smem
$ smem -r -k | head -15
  PID User     Command                         Swap      USS      PSS      RSS
 2341 postgres postgres: worker                1.4G     2.8G     2.9G     3.0G
 8821 app      node /opt/app/server.js          180M     1.7G     1.8G     1.9G
 8834 app      node /opt/app/server.js          160M     1.6G     1.7G     1.9G
# Swap column shows how much each process has in swap right now
# USS = Unique Set Size (truly private memory — the real cost of running this process)
# PSS = Proportional Set Size (USS + fair share of shared libs)
```

4 **Determine if this is a growth problem or a sizing problem.** If processes are at a stable high-water mark and swap is stable, the server may simply need more RAM. If memory is growing, investigate a leak (Scenario 3). Check what ran overnight:

```
# Did a cron job run overnight that caused this?
$ grep "$(date -d yesterday '+%b %e')" /var/log/syslog | grep -i cron
$ journalctl --since yesterday --until "6 hours ago" -u cron
```

5 **Resolution options:**

- **Short term:** If the system is stable (si/so = 0), leave it. Pages in swap that aren't needed won't be read back, and the system is coping.
- **If actively swapping and sluggish:** Identify the largest non-essential process (step 2) and restart it during a maintenance window — this frees its RSS and its swap pages. Do not kill a database or web server without preparation.
- **Medium term:** Lower `vm.swappiness` to make the kernel prefer reclaiming page cache over swapping, giving processes more RAM before they're pushed to swap.
- **Long term:** Add RAM, or reduce the memory footprint of the services (connection pool sizes, worker counts, JVM heap limits).

Never run production without swap entirely. If a workload occasionally spikes, swap is the safety net that prevents an OOM kill. The goal is a system that rarely uses swap, not one that has none.

Scenario 2 — The OOM Killer Fired Overnight

SCENARIO · CHAPTER 3 · SCENARIO 2

A service is down this morning. No one deployed anything. What happened?

1 **Check dmesg for OOM kill events — this is the first thing to look at.**

```
$ dmesg -T | grep -i "oom\|killed process\|out of memory"
[Mon Jun 16 03:47:22 2025] Out of memory: Kill process 8821 (node) score 482 or sacrifice child
[Mon Jun 16 03:47:22 2025] Killed process 8821 (node) total-vm:655360kB, anon-rss:1843200kB, file-rss:90112kB, sh
# Time: 03:47 — overnight as suspected
# Process: PID 8821, command "node"
# anon-rss: 1.76 GB was in RAM when it was killed
# score 482 — OOM score at time of kill (higher = more likely to be killed)
```

2 **Read the full OOM event for context** — dmesg logs the memory state of the whole system at the moment of the kill.

```
$ dmesg -T | grep -A 30 "Out of memory" | head -40
[Mon Jun 16 03:47:21 2025] node invoked oom-killer: gfp_mask=0x..., order=0, oom_score_adj=0
[Mon Jun 16 03:47:21 2025] Mem-Info:
[Mon Jun 16 03:47:21 2025] active_anon:458752 inactive_anon:12288 isolated_anon:0
[Mon Jun 16 03:47:21 2025] active_file:256 inactive_file:384 isolated_file:0
# inactive_file (reclaimable page cache) is tiny: 384 pages = 1.5 MB
# The kernel had almost no cache left to reclaim before turning to swap/OOM
# This means the system was already very memory-pressured before the kill
[Mon Jun 16 03:47:21 2025] MemFree:    12288kB  ← Only 12 MB truly free at time of kill
```

3 **Also check journalctl** — on systemd systems, OOM kills appear in the journal too, often with more context about which service was affected.

```
$ journalctl -k --since "2025-06-16 03:40" --until "2025-06-16 04:00"
# -k = kernel messages only (same as dmesg)

$ journalctl -u myapp.service --since yesterday | grep -i "kill\|oom\|memory"
```

4 **Understand the OOM score — why was this process chosen?** The kernel assigns every process an `oom_score` from 0–1000. Higher = more likely to be killed. It's based primarily on memory usage as a fraction of total RAM, adjusted by `oom_score_adj`.

```
# Check the OOM score of running processes
$ for PID in $(ps -eo pid --no-headers); do
    printf "%6d %5d %s\n" $PID \
      $(cat /proc/$PID/oom_score 2>/dev/null) \
      "$(cat /proc/$PID/cmdline 2>/dev/null | tr '\0' ' ' | cut -c1-60)"
  done | sort -k2 -rn | head -10
  8821   482  node /opt/app/server.js
  2341   310  postgres: worker
   891    45  sshd: root
```

5 **Protect critical processes from OOM kills with oom_score_adj.** Setting this to -1000 makes the process immune; 0 is default; +1000 makes it the first target.

```
# Protect a running process (e.g., your database)
$ echo -500 | sudo tee /proc/2341/oom_score_adj
# For a systemd service — persistent across restarts:
$ sudo systemctl edit postgresql.service
# Add under [Service]:
# OOMScoreAdjust=-500

# Make a disposable worker MORE likely to be killed first (protecting everything else)
$ echo 500 | sudo tee /proc/8821/oom_score_adj
```

The OOM killer is a last resort — it only fires when the kernel genuinely cannot find any memory to allocate from any source. If you're seeing regular OOM kills, the fix is either more RAM, reduced process memory footprints, or better swap management — not adjusting oom_score_adj to protect things. That only changes who gets killed, not whether the kill happens.

Scenario 3 — A Service's Memory Keeps Growing

SCENARIO · CHAPTER 3 · SCENARIO 3

A long-running service uses 200 MB after restart, but after 48 hours it's at 3 GB. Nothing changed in the code.

1 **Confirm the growth is real, not just page cache being attributed to the process.** Watch RSS (not VSZ) over time — RSS is wha

```
# Watch RSS of a specific PID every 30 seconds
$ watch -n 30 'ps -p 8821 -o pid,rss,vsz,etime,comm | awk "{printf \"%s PID:%s RSS:%s MB VSZ:%s MB uptime:%s\n\",

# Or log it to a file to track the trend:
$ while true; do
    echo "$(date '+%H:%M:%S') $(ps -p 8821 -o rss= | awk '{printf "%.0f MB\n", $1/1024}')"
    sleep 60
  done | tee /tmp/mem_growth.log

# After an hour, check the trend:
09:00:00 204 MB
09:30:00 261 MB
10:00:00 318 MB
10:30:00 375 MB  ← Growing ~57 MB every 30 minutes — linear leak pattern
```

2 **Use pmap to inspect the process's memory map** — look for anonymous mappings that are growing, which indicates heap or mmap-based allocations not being freed.

```
$ pmap -x 8821 | tail -20
Address           Kbytes     RSS   Dirty Mode  Mapping
00007f8b2c000000  2097152  2097152  2097152 rw---   [ anon ]
00007f8b4c000000  524288   524288  524288 rw---   [ anon ]
...
----------------  -------  ------  ------
total kB          3276800  3145728  3014656
# Large anonymous mappings with RSS == Kbytes == Dirty means
# these pages are allocated, in memory, and have been modified —
# but nothing is freeing them. Classic heap leak pattern.

# Run pmap again 10 minutes later and compare anon totals:
$ pmap -x 8821 | grep anon | awk '{sum += $2} END {print sum/1024 " MB anon"}'
```

3 **Check for open file descriptors accumulating** — sometimes what looks like a memory leak is actually file handles or socket connections not being closed, each consuming a small amount of kernel memory.

```
# Count open file descriptors for the process
$ ls /proc/8821/fd | wc -l
4821
# 4,821 open file descriptors — suspicious for most apps.
# What are they?
$ lsof -p 8821 | awk '{print $5}' | sort | uniq -c | sort -rn | head -10
   4201 IPv4         ← Over 4000 open network connections — likely connection leak
    300 REG          ← Regular files
    200 sock         ← Sockets
```

4 **Document, report, and mitigate.** A genuine memory leak is a code bug that requires a fix. In the meantime:

- **Document the growth rate** — how long before it hits the danger zone? This defines your maintenance window.
- **Set up a scheduled restart** — `systemctl restart myapp.service` via cron at off-peak hours buys time while the fix is developed.
- **Set a memory limit via systemd** — `MemoryMax=2G` in the service unit will trigger an OOM kill of the leaking service (not the whole system) if it exceeds 2 GB, protecting other services.
- **Report to developers** — include your pmap output, the growth rate log, and the lsof fd count. This is the evidence needed to find the leak.

Memory leaks in interpreted languages (Node.js, Python, Ruby) are often event listener accumulation, cache objects that grow without bounds, or closures keeping references alive. In C/C++ services, tools like Valgrind or AddressSanitizer are used during development to find the leaking allocation.

OOM Score — Who Gets Killed First

`-1000`

Never kill

Completely protected from OOM killer. Set for init/systemd, critical infrastructure. Use with extreme care — if this process leaks, the system will OOM-kill everything else first.

`-500`

Strongly protected

Good for databases (PostgreSQL, MySQL, Redis). Unlikely to be killed unless the system is truly desperate. Set via systemd OOMScoreAdjust or /proc/PID/oom_score_adj.

`0`

Default

All processes start here. Final oom_score is calculated from this base plus memory usage. Large-RSS processes end up with higher scores.

`+500 to +1000`

Kill me first

Useful for disposable worker processes or batch jobs — if the system runs low, kill this before touching production services. Systemd sets +100 for most user services by default on some distros.

Page Cache — When to Clear It (Rarely)

The page cache is self-managing. The kernel evicts the oldest, least-used pages automatically when processes need RAM. You almost never need to clear it manually on a production server.

> **Clearing the page cache on a production server causes a performance cliff.** Every file access that was being served from RAM now hits disk — databases, web servers, and application servers all slow dramatically for minutes until the cache warms up again. Only clear it for benchmarking (to get a cold-cache baseline) or on a test system.

```
# If you genuinely need to clear the page cache (benchmarking, test systems ONLY):
$ sync                                  # Flush dirty pages to disk first
$ echo 1 | sudo tee /proc/sys/vm/drop_caches  # 1=page cache only
$ echo 2 | sudo tee /proc/sys/vm/drop_caches  # 2=dentries+inodes
$ echo 3 | sudo tee /proc/sys/vm/drop_caches  # 3=everything
# Effect is immediate but temporary — cache rebuilds as soon as files are accessed again.
# This does NOT help with process memory (AnonPages) — only file cache is affected.
```

> **If your monitoring shows "used memory" jumping after a cache clear, that's normal** — the graph is just showing the cache being rebuilt. The kernel is not "wasting" memory; it's making your system faster for the next time those files are read.

Quick Reference — Chapter 3 Commands

COMMAND	PURPOSE	KEY FLAGS / NOTES
`free -h`	Memory and swap — human-readable. Watch "available", not "free".	`-s 2` refresh every 2 seconds
`cat /proc/meminfo`	Full kernel memory accounting — MemAvailable, AnonPages, Committed_AS, Slab	Most reliable source; `grep MemAvailable` for the key figure
`vmstat 1`	Watch si/so columns — non-zero = active swapping (bad)	wa column shows % CPU time waiting for I/O (includes swap I/O)
`ps aux --sort=-%mem`	Processes sorted by memory usage (RSS). Quick top-consumers list.	`| head -10` · RSS is in KB
`smem -r -k`	Per-process USS/PSS/RSS/Swap — more accurate than ps for real memory cost	`-r` reverse sort · `-k` human sizes · may need install
`pmap -x PID`	Memory map of one process — find growing anonymous mappings (leaks)	`| grep anon` to filter heap · `| tail` for totals
`lsof -p PID`	Open files and sockets for one process — spot connection or fd leaks	`| wc -l` for count · `| awk '{print $5}' | sort | uniq -c` for type breakdown
`dmesg -T | grep -i oom`	Find OOM kill events — shows process killed, its RSS, and oom_score	Also: `journalctl -k | grep -i oom`
`cat /proc/PID/oom_score`	Current OOM kill score for a process (0–1000, higher = more likely to die)	Loop over all PIDs with `ps -eo pid` to find highest-scored processes
`echo N | tee /proc/PID/oom_score_adj`	Adjust OOM kill priority: -1000 (immune) to +1000 (kill first)	Persistent via systemd `OOMScoreAdjust=` in unit file
`sysctl vm.swappiness`	Check swap eagerness (default 60). Lower = prefer reclaiming cache over swapping.	`sysctl -w vm.swappiness=10` to change · persist in `/etc/sysctl.d/`
`watch -n 30 'ps -p PID -o rss='`	Monitor RSS of one process every 30s — track memory leak growth rate	Pipe to `tee /tmp/mem.log` to keep a history

CHAPTER 4 OF 8

Chapter 4 — Disk I/O & Storage Bottlenecks

Disk problems come in two distinct flavours that require completely different approaches. The first is space — the filesystem is full and nothing can be written. The second is I/O performance — the disk has space but is so busy that read and write operations queue up, causing the system to grind. This chapter covers both, including the trap where df and du disagree and the reason why is a hidden process.

> **What this chapter covers:** Why df and du can disagree — the deleted-but-open file trap. Reading iostat's extended output. Scenario 1: root filesystem at 100%. Scenario 2: inode exhaustion — disk has space but files can't be created. Scenario 3: high I/O wait — finding which process is hammering the disk. Log file management. ionice for throttling disk-intensive processes. Mount options that affect performance.

df vs du — Why They Sometimes Disagree

`df` reads disk usage from the filesystem's own accounting (the superblock). `du` walks the directory tree and adds up file sizes. These two methods give the same answer *unless* a file has been deleted from the directory tree while a process still holds it open — in which case `df` still counts the space (the inode and its blocks are still allocated) while `du` doesn't (the directory entry is gone, so the walk misses it).

```
The Deleted-But-Open File Problem

Normal state:
Directory entry  →  Inode  →  Data blocks on disk
(du finds this)     (df counts this)

After "rm large_log.log" while nginx still has it open:
~~Directory entry~~  →  Inode  →  Data blocks on disk
(du: file gone!)         (df: still counted! inode still allocated)

Effect: df says filesystem is 8 GB used.
        du / reports only 4 GB.
        Missing 4 GB = deleted-but-open files.

Fix: find the process holding the file open, restart it or
     send it SIGHUP to reopen its log files. The OS then
     releases the inode and the space is reclaimed.
```

```
# Step 1: spot the discrepancy
$ df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        50G   48G  2.0G  96% /

$ du -sh /* 2>/dev/null | sort -h | tail -5
 12G  /usr
  8G  /var
  4G  /home
  2G  /opt
# Total from du: ~26 GB. df says 48 GB used. ~22 GB unaccounted for.

# Step 2: find deleted-but-open files and their size
$ lsof | grep -i deleted | awk '{print $7, $1, $2, $NF}' | sort -rn | head -10
22548342912  nginx  1234  /var/log/nginx/access.log (deleted)
 4194304000  python 5678  /tmp/cache_dump.bin (deleted)
# nginx is holding a 21 GB deleted log file open. That's the missing space.

# Step 3a: send SIGHUP to nginx — it reopens its log files, releasing the old inode
$ kill -HUP $(pgrep nginx)    # or: nginx -s reopen

# Step 3b: if SIGHUP isn't enough, restart the service
$ systemctl restart nginx

# Verify space is reclaimed:
$ df -h /
Filesystem      Size  Used Avail Use%
/dev/sda1        50G   26G   24G  52%
```

> **Most well-behaved services (nginx, Apache, PostgreSQL, syslog) support SIGHUP** to reopen their log files without dropping connections or restarting. This is also how `logrotate` works — it renames the log file, sends SIGHUP to the service, and the service starts writing to a new file at the original path. The old file (now renamed) can then be compressed or deleted.

Reading iostat — The Columns That Matter

`iostat -x 1` gives per-device I/O statistics updated every second. It's the primary tool for identifying which disk is saturated and whether the problem is reads, writes, throughput, or latency.

```
$ iostat -xh 1
Device    r/s    w/s   rkB/s   wkB/s  await  r_await  w_await  aqu-sz  %util
sda      12.0    8.0   480.0   320.0   2.1    1.8      2.5      0.04     3.2
nvme0n1  45.0  320.0  1800.0 12800.0  4.8    2.1      5.2      0.82    42.0
sdb       2.0 1840.0     8.0  73600.0  180.0  12.0    182.0    24.8    98.6
# sda: healthy — low utilisation, low await
# nvme0n1: busy but not saturated — 42% util, await <5ms is fine for NVMe
# sdb: SATURATED — 98.6% util, queue depth 24.8, await 180ms (should be <10ms on SSD)
```

%util

UTILISATION

Percentage of time the device was busy. At 100% the device is saturated — I/O requests are queuing.

⚠ Above 80% sustained on HDD, or 95%+ on SSD — investigate

await

AVERAGE I/O WAIT (MS)

Average time from I/O request submission to completion. Includes queue time plus actual disk service time.

⚠ HDD: >20ms. SSD: >5ms. NVMe: >1ms. Consistently above these = saturation

aqu-sz

AVERAGE QUEUE DEPTH

Average number of I/O requests waiting in the device queue. The clearest saturation signal.

⚠ Sustained above 1–2 on a single disk = more requests than the device can handle

r/s, w/s

OPERATIONS / SECOND

Read and write IOPS. Spins disks are limited to ~100–200 IOPS. SSDs handle 10,000–100,000+. Tells you whether the workload is I/O-count bound.

⚠ HDD: r/s + w/s consistently near 150–200 = IOPS saturated

rkB/s, wkB/s

THROUGHPUT (KB/S)

Actual data transferred per second. Compare against the device's rated throughput. Large sequential files show high kB/s with low IOPS.

⚠ HDD: above ~100–150 MB/s. SSD: above rated speed. Sustained = throughput bound.

r_await / w_await

READ / WRITE AWAIT SEPARATELY

Splits await into read-side and write-side. If w_await is high but r_await is fine, writes are queuing. Useful for mixed workloads.

⚠ Asymmetry between r_await and w_await points to write-heavy saturation

iotop — Finding Which Process Is Doing the I/O

`iostat` tells you the device is saturated. `iotop` tells you who is responsible. It requires root (or `CAP_NET_ADMIN`) to read kernel I/O accounting.

```
$ iotop -o          # -o = only show processes with active I/O
Total DISK READ: 1.20 M/s | Total DISK WRITE: 73.60 M/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 9821 be/4  mysql     0.00 B/s  68.40 M/s   0.00 %  98.72 %  mysqld
 4812 be/4  www-data  1.20 M/s   5.20 M/s   0.00 %   1.02 %  php-fpm: pool www
# mysqld is responsible for 68 MB/s of writes and 98.7% of I/O time
# IO> column shows % of time the process was blocked waiting on I/O

# Batch mode — useful for logging or running from scripts
$ iotop -o -b -n 5  # -b = batch mode, -n 5 = 5 iterations then exit

# Once you have the PID, find what files it's accessing
$ lsof -p 9821 | grep -E "REG|DIR" | head -20
mysqld 9821 mysql  4u  REG  8,2  2147483648  /var/lib/mysql/ibdata1
mysqld 9821 mysql  5u  REG  8,2   524288000  /var/lib/mysql/ib_logfile0
# InnoDB is writing heavily to its redo log — normal under high write load,
# but if await is 180ms, either the disk is inadequate or innodb_flush_log_at_trx_commit=1
# is flushing on every transaction (safe but expensive)
```

Scenario 1 — Root Filesystem at 100%

SCENARIO · CHAPTER 4 · SCENARIO 1

df shows / at 100%. The application is throwing "No space left on device" errors.

1 **Confirm which filesystem is full and check for the df/du discrepancy first.** If they disagree, a deleted-but-open file may be the cause — the fastest fix with no risk of deleting the wrong thing.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        50G   50G     0  100% /
/dev/sdb1       500G  120G  380G  24% /data

$ lsof | grep -i deleted | awk '{printf "%s MB\t%s\t%s\n", $7/1048576, $1, $NF}' | sort -rn | head -5
21504 MB    nginx    /var/log/nginx/access.log (deleted)
# Found it immediately. nginx is holding 21 GB of deleted log space.
$ kill -HUP $(pgrep -x nginx)
$ df -h /
Filesystem      Size  Used Avail Use%
/dev/sda1        50G   29G   21G   58%  ← Immediate recovery, no deletions needed
```

2 **If lsof shows nothing significant, find the largest directories.** Work top-down — start at root, drill into the biggest directory at each level.

```
$ du -sh /* 2>/dev/null | sort -h
 12G  /usr
 38G  /var   # ← biggest, investigate this next
  4G  /home

$ du -sh /var/* 2>/dev/null | sort -h
  1G  /var/cache
  2G  /var/lib
 35G  /var/log   # ← 35 GB in logs

$ du -sh /var/log/* 2>/dev/null | sort -h | tail -10
  1G  /var/log/syslog
 30G  /var/log/myapp   # ← application log directory

$ ls -lh /var/log/myapp/
-rw-r--r-- 1 app app 30G Jun 14 14:22 debug.log
# Debug logging left enabled in production. Common cause.
```

3 **Use ncdu for interactive exploration** — faster than repeated du commands when you don't know where to look.

```
# ncdu may need: apt install ncdu / yum install ncdu
$ ncdu /var
# Opens a curses interface. Arrow keys navigate, Enter descends,
# d deletes (with confirmation), q quits.
# Shows size of each directory sorted largest first — much faster than du for exploration.
```

4 **Safe quick wins to recover space — in order of risk:**

- **Zero risk:** Clear the systemd journal — `journalctl --vacuum-size=500M`
- **Zero risk:** Clear package manager cache — `apt clean` or `yum clean all`
- **Zero risk:** Remove old kernel packages — `apt autoremove` (keeps current kernel)
- **Low risk:** Truncate (not delete) the offending log, leaving the file handle intact — `truncate -s 0 /var/log/myapp/debug.log`
- **Check before removing:** Core dumps in `/var/crash` or `/tmp` — `find /tmp /var/crash -name "core*" -size +100M`
- **Check before removing:** Old log rotations — `find /var/log -name "*.gz" -mtime +30`

5 **Truncate vs delete — why truncate is safer for open log files.** If you `rm` a log file that a running process has open, the process continues writing to the now-deleted inode — the space isn't freed (the deleted-but-open problem again). `truncate -s 0` zeros the file's content while leaving the inode intact — the process's file handle remains valid and the space is freed immediately.

```
# Safe: truncate the file while the process keeps writing to it
$ truncate -s 0 /var/log/myapp/debug.log
# The file now has 0 bytes. The running process continues writing to it normally.
# Alternatively: > /var/log/myapp/debug.log  (shell redirection truncate)

# Unsafe: rm while process is writing
# rm /var/log/myapp/debug.log  ← process keeps the deleted inode open; space not freed
```

6 **Root cause and prevention.** After recovering space, address why it filled up:

- Set up `logrotate` for application logs (or ensure it's configured correctly)
- Disable debug logging in production application config
- Set a journal size cap: `SystemMaxUse=500M` in `/etc/systemd/journald.conf`
- Set up disk space monitoring alerts (before it reaches 100%)

On systems where / is separate from /var, /tmp, and /home, a full /var won't affect the system's ability to run — only to write logs and package updates. On single-partition systems, a full / can prevent logins and crash running services. Always know your partition layout before an incident.

Scenario 2 — Inode Exhaustion

SCENARIO · CHAPTER 4 · SCENARIO 2

"No space left on device" — but df shows 40% free. Files can't be created anywhere.

1 **Check inode usage immediately** — this is the tell. Inodes are pre-allocated metadata slots (one per file or directory). When they run out, no new files can be created even if disk space is available.

```
$ df -i
Filesystem      Inodes   IUsed   IFree IUse% Mounted on
/dev/sda1      6553600 6553600       0  100% /
/dev/sdb1      6553600  412000 6141600    6% /data
# / has zero inodes remaining. This explains the error despite available disk space.
```

2 **Find which directory contains the most files.** Inode exhaustion is almost always caused by one directory containing millions of tiny files — mail queues, PHP session files, cache directories, or a logging system that creates one file per event.

```
# Count files per top-level directory (can take a while on a full system)
$ for dir in /*; do
    count=$(find "$dir" -xdev 2>/dev/null | wc -l)
    echo "$count $dir"
  done | sort -rn | head -10
5821432 /var
  412000 /usr
   98000 /home

# Drill into /var
$ find /var -xdev -printf '%h\n' 2>/dev/null | sort | uniq -c | sort -rn | head -10
5819200 /var/spool/postfix/deferred
   1200 /var/log
# The postfix deferred queue has 5.8 million messages. Classic inode exhaustion cause.
```

3 **Recovery — delete the tiny files.** With millions of files, `rm -rf /path/*` fails ("argument list too long"). Use find instead:

```
# For a mail queue — flush or delete deferred messages
$ postsuper -d ALL deferred   # Postfix: delete all deferred messages properly

# Generic: delete files in a directory with too many for rm to handle
$ find /var/spool/postfix/deferred -type f -delete

# For PHP session files (another common culprit):
$ find /var/lib/php/sessions -type f -mtime +1 -delete
# -mtime +1 = only files older than 1 day — leaves active sessions intact

# Verify recovery:
$ df -i /
Filesystem      Inodes   IUsed   IFree IUse%
/dev/sda1      6553600  734400 5819200   11%
```

4 **Prevention.** Add inode monitoring to your alerting alongside disk space — many monitoring systems skip it. Also consider: if your application legitimately creates millions of small files, use a directory structure that spreads them across subdirectories

(e.g., by first two characters of the filename: `ca/cache_abc123`) rather than a flat directory. Some filesystems (ext4 with `dir_index`) handle large directories better than others.

You cannot add inodes to an existing ext4 filesystem without reformatting. The inode count is set at creation time with `mkfs.ext4 -i bytes-per-inode`. If inode exhaustion is a recurring problem on a partition you can't reformat, consider moving the high-file-count directory to its own partition or filesystem (like tmpfs for session files).

Scenario 3 — High I/O Wait, Server Sluggish

SCENARIO · CHAPTER 4 · SCENARIO 3

vmstat shows wa=45%. The server is slow but CPU is mostly idle and memory is fine.

1. **Confirm it's disk I/O wait and not NFS or swap.** High wa covers all uninterruptible I/O — disk, NFS mounts, and swap all contribute.

```
$ vmstat 1 5
 r  b  swpd  free   buff  cache   si   so   bi    bo   in   cs  us sy  id  wa
 1  4     0  2.1G  200M  4.2G    0    0  1200  73800  2100  3400  8  4  43  45
# b=4: four processes in uninterruptible sleep (D state)
# si=0 so=0: no swap activity — not a memory problem
# bo=73800 KB/s: 72 MB/s of writes — high for a spinning disk
# id=43, wa=45: CPU is idle nearly half the time, but blocked on I/O the other half
```

2. **Identify the saturated device with iostat.**

```
$ iostat -xh 1 3
Device    r/s   w/s   rkB/s    wkB/s  await  aqu-sz  %util
nvme0n1   8.0  12.0   320.0    480.0    0.9    0.02   2.1
sdb       2.0 1840.0    8.0  73600.0  165.0   24.4   98.8
# sdb: the spinning HDD. 98.8% utilised, 165ms await (should be <20ms), queue 24.
# It's completely saturated by 73 MB/s of writes — well beyond its ~100-150 MB/s limit
# for sequential writes, and catastrophic for random writes.
```

3. **Find which process is responsible with iotop.**

```
$ iotop -o
Total DISK READ: 8.0 KB/s | Total DISK WRITE: 73.60 MB/s
  TID  PRIO  USER     DISK READ   DISK WRITE  IO>   COMMAND
12841 be/4  backup    0.00 B/s   73.50 MB/s  96.2% rsync -av /data /mnt/backup
 4812 be/4  www-data  8.00 KB/s   0.10 MB/s   1.8% php-fpm
# rsync backup job is consuming essentially all disk write bandwidth.
# It's writing to sdb (the backup mount) but the read side (reading /data from nvme0n1)
# is creating cache pressure that's causing page evictions on sdb.
```

4. **Throttle the backup job with ionice without stopping it.** `ionice` changes a process's I/O scheduling class — the disk equivalent of `renice` for CPU.

```
# Set rsync to idle I/O class — only gets disk access when nothing else needs it
$ ionice -c 3 -p 12841
# Class 3 = idle: process gets I/O only when no other process wants the disk
# The backup will slow down significantly but production I/O is protected

# Alternatively: best-effort with low priority (0=high, 7=low)
$ ionice -c 2 -n 7 -p 12841

# For future backups — launch rsync with low priority from the start:
$ ionice -c 3 nice -n 19 rsync -av /data /mnt/backup

# Verify the change reduced I/O wait:
$ vmstat 1 3
 r  b  swpd  free  si  so   bi     bo    wa
 0  0     0  2.3G   0   0  800   8400     5  ← wa dropped from 45% to 5%
```

5 **If the I/O is from a production service and not a background job** — the disk is genuinely undersized for the workload. Options to investigate:

- Move the high-I/O service to an NVMe SSD if it's currently on HDD
- Add a read cache (bcache, lvmcache) to the slow disk
- For databases: tune `innodb_buffer_pool_size` (MySQL) or `shared_buffers` (PostgreSQL) to serve more reads from RAM and reduce disk reads
- For write-heavy workloads: check if `sync()` / `fsync()` is being called too frequently — sometimes a config change (careful: tradeoff with durability) can dramatically reduce write IOPS

Always run iotop at the same time as iostat when diagnosing I/O — iostat identifies the saturated device, iotop identifies the responsible process. Neither alone gives the full picture.

Log Files — The Most Common Disk-Space Culprit

Logs fill disks more reliably than almost anything else. The systemd journal alone can consume dozens of gigabytes without size limits configured.

```
# How much space is the systemd journal using?
$ journalctl --disk-usage
Archived and active journals take up 8.3G in the filesystem.

# Immediate trim — keep only the most recent 500 MB
$ journalctl --vacuum-size=500M
# Or: keep only the last 2 weeks
$ journalctl --vacuum-time=2weeks

# Make limits permanent in /etc/systemd/journald.conf:
$ grep -E "SystemMaxUse|MaxRetention" /etc/systemd/journald.conf
SystemMaxUse=500M       # Hard cap on journal disk usage
MaxRetentionSec=2week   # Auto-delete entries older than 2 weeks
$ systemctl restart systemd-journald

# Find large traditional log files
$ find /var/log -type f -size +100M -exec ls -lh {} \;
-rw-r--r-- 1 root root  30G Jun 14 /var/log/myapp/debug.log
-rw-r--r-- 1 root root 4.2G Jun 12 /var/log/syslog

# Check logrotate configuration for an application
$ cat /etc/logrotate.d/nginx
/var/log/nginx/*.log {
    daily          # Rotate daily
    missingok      # Don't error if log is missing
    rotate 14      # Keep 14 days of logs
    compress       # Compress old logs (.gz)
    delaycompress  # Compress previous rotation, not current
    notifempty     # Don't rotate empty logs
    sharedscripts
    postrotate
        nginx -s reopen  # Signal nginx to reopen log files after rotation
    endscript
}

# Force an immediate logrotate run (for testing or emergency)
$ logrotate -f /etc/logrotate.d/nginx
```

Mount Options That Affect Performance

The options used when mounting a filesystem can make a measurable difference, particularly for read-heavy or small-file workloads. They're set in `/etc/fstab` or via the `mount` command.

`noatime`

No access time updates

By default, Linux updates the "last accessed" timestamp on every file read. On read-heavy workloads this turns every read into a write, doubling I/O. `noatime` disables this entirely.

`relatime`

Relative access time

A compromise: only updates atime if it's older than mtime (when the file was last modified). This is the default on modern Linux — most systems already have this behaviour without explicitly setting it.

Impact: significant — up to 30% I/O reduction on read-heavy small-file workloads. Safe for most servers. May affect backup tools that use atime to detect changed files.

Impact: moderate. Good balance — keeps atime meaningful for tools that use it while eliminating most unnecessary write traffic. The safe default.

errors=remount-ro

Remount read-only on error

If the filesystem encounters an error, remount it read-only instead of continuing to allow writes that could corrupt data. Common default for ext4.

Impact: safety improvement. No performance effect. Prevents data corruption on disk errors — recommended for all production filesystems.

nobarrier

Disable write barriers

Write barriers ensure data is physically on disk before acknowledging a write, protecting against corruption on power loss. Disabling them can improve write performance but risks data loss on power failure.

Impact: dangerous on real hardware. Only safe on VMs where the hypervisor provides equivalent guarantees. Do not use on physical servers without a battery-backed write cache.

```
# Check current mount options for a filesystem
$ findmnt -o TARGET,OPTIONS /
TARGET  OPTIONS
/       rw,relatime,errors=remount-ro

# Add noatime to /etc/fstab for the root filesystem:
$ grep " / " /etc/fstab
UUID=abc123  /  ext4  rw,relatime,errors=remount-ro  0  1
# Change relatime to noatime:
UUID=abc123  /  ext4  rw,noatime,errors=remount-ro  0  1

# Remount immediately without rebooting:
$ mount -o remount,noatime /
```

Quick Reference — Chapter 4 Commands

COMMAND	PURPOSE	KEY FLAGS / NOTES
`df -h`	Disk space per filesystem — human-readable	`-i` inode usage · `-T` show filesystem type · any filesystem at 100% = problem
`df -i`	Inode usage per filesystem — check when "no space" despite available space	100% inodes = no new files can be created regardless of disk space
`du -sh /* 2>/dev/null | sort -h`	Size of each top-level directory, sorted smallest to largest	Drill down iteratively: `du -sh /var/* | sort -h` etc.
`ncdu /path`	Interactive disk usage explorer — navigate, sort, delete	Much faster than repeated du. May need: `apt install ncdu`
`lsof | grep deleted`	Find deleted-but-open files still consuming disk space	Pipe to `awk '{print $7, $1, $NF}' | sort -rn` to show size + owner + filename
`truncate -s 0 file`	Zero a log file's content while leaving file handle intact	Safer than rm for files held open by running processes
`kill -HUP PID`	Signal a service to reopen its log files after rotation	Works for nginx, Apache, syslog, PostgreSQL and most well-behaved services
`iostat -xh 1`	Per-device I/O stats — find saturated devices	Watch %util (saturation), await (latency ms), aqu-sz (queue depth)
`iotop -o`	Per-process I/O — find which process is hammering the disk	`-b -n 5` batch mode · `-a` accumulated totals · requires root
`ionice -c 3 -p PID`	Set process to idle I/O class — only gets disk when nothing else needs it	`-c 2 -n 7` = best-effort low priority · use for backup jobs, batch work
`find /dir -type f -delete`	Delete all files in a directory when rm -rf fails (too many args)	Add `-mtime +N` to only delete files older than N days
`journalctl --disk-usage`	Show how much disk the systemd journal is using	`--vacuum-size=500M` trim immediately · `--vacuum-time=2weeks` by age
`logrotate -f /etc/logrotate.d/X`	Force an immediate logrotate run for a specific application	Check config with `logrotate --debug`
`findmnt -o TARGET,OPTIONS /`	Show current mount options for a filesystem	Add `noatime` in /etc/fstab + `mount -o remount,noatime /` to apply live

CHAPTER 5 OF 8

Chapter 5 — Network Performance

Network problems are the easiest bottleneck to miss. When CPU and memory look healthy, many administrators declare the server fine — but a 200ms DNS lookup on every database query, or a NIC saturated by a background rsync, can make an application feel broken without leaving any obvious trace in the usual tools. This chapter covers how to systematically rule network in or out, and how to find and fix it when it's the culprit.

> **What this chapter covers:** Bandwidth vs latency — two separate problems. Diagnosing with ping, mtr, and traceroute. Reading ss (the modern netstat). TCP connection states — ESTABLISHED, TIME_WAIT, CLOSE_WAIT and what each means. Scenario 1: app is slow but CPU/mem are idle. Scenario 2: thousands of TIME_WAIT connections. Scenario 3: NIC saturation — finding the process and throttling it. /proc/net/dev for NIC errors and drops.

Bandwidth vs Latency — Two Different Problems

Bandwidth (Throughput)

The maximum data transfer rate — how wide the pipe is. Measured in Mbps or GB/s. **Symptoms when limited:** large file transfers are slow, video streams buffer, bulk API calls take long. **Diagnosed with:** iftop, nethogs, /proc/net/dev. The NIC itself has a hard limit (1 Gbps, 10 Gbps etc.).

Latency (Round-Trip Time)

The time for a single packet to travel and return — how fast the pipe responds. Measured in milliseconds. **Symptoms when high:** interactive apps feel laggy, many small API calls are slow even though bandwidth is fine, database queries with many round-trips are sluggish. **Diagnosed with:** ping, mtr, curl timing.

Packet Loss

A percentage of packets that never arrive. Even 1% loss causes TCP to retransmit, adding latency and throttling throughput dramatically. **Symptoms:** connections work but are unreliable and slow, timeouts appear intermittently. **Diagnosed with:** mtr (shows per-hop loss%), ping with count.

DNS Resolution Time

Often invisible in monitoring but catastrophic in impact. If your app resolves a hostname on every request and DNS takes 200ms, that's 200ms of latency on every operation. **Symptoms:** app slow, server-to-server calls slow, intermittent timeouts. **Diagnosed with:** dig +stats, strace on a running process.

Connectivity Diagnostics — ping, traceroute, mtr

```
# ping — basic connectivity and round-trip time
$ ping -c 10 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=116 time=12.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=11.9 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=116 time=245.1 ms  ← spike
--- 8.8.8.8 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9013ms
rtt min/avg/max/mdev = 11.9/38.2/245.1/69.4 ms
# avg 38ms but max 245ms — one packet had a large spike. mdev (deviation) of 69ms is high.
# This jitter pattern suggests congestion somewhere in the path, not a constant problem.

# traceroute — which hop in the path is slow?
$ traceroute -n 8.8.8.8   # -n skips reverse DNS for faster output
 1  192.168.1.1      1.2 ms   1.1 ms   1.0 ms   ← your router
 2  10.0.0.1         4.8 ms   4.9 ms   4.7 ms   ← ISP edge
 3  * * *                                       ← hop drops ICMP (normal)
 4  72.14.215.100   185.0 ms 182.0 ms 190.0 ms  ← latency jump HERE
 5  8.8.8.8          12.3 ms  12.1 ms  12.4 ms   ← destination is fast
# Hop 4 is adding ~180ms — a peering point or transit link under load.
# Hop 5 (the destination) is fine. The problem is in the path, not the destination.
```

mtr — the best tool for diagnosing path problems

mtr (Matt's Traceroute) combines ping and traceroute into a live view that shows round-trip time *and* packet loss at every hop simultaneously. It's the single most useful tool for diagnosing whether a network problem is in your server, your network, or somewhere on the internet path.

```
$ mtr --report --report-cycles 20 -n 8.8.8.8
# --report: print a summary after 20 cycles instead of live display
# -n: no reverse DNS (faster)
Start: 2025-06-14T15:22:10+0100
HOST: myserver                    Loss%   Snt  Last   Avg  Best  Wrst StDev
  1. 192.168.1.1                   0.0%    20   1.2   1.1   1.0   1.5   0.1
  2. 10.0.0.1                      0.0%    20   4.8   4.7   4.6   5.1   0.2
  3. ???                         100.0%    20   0.0   0.0   0.0   0.0   0.0
  4. 72.14.215.100                 0.0%    20  12.0  12.1  11.8  12.6   0.2
  5. 8.8.8.8                       0.0%    20  12.3  12.2  12.0  12.8   0.2
# Hop 3 shows 100% loss — but hop 4 and 5 are fine with 0% loss.
# This means hop 3 just doesn't respond to ICMP probes. It IS forwarding packets.
# Genuine packet loss would show at hop 4 or 5, not just hop 3.
```

> **100% loss at an intermediate hop is not packet loss** — many routers de-prioritise or block ICMP TTL-exceeded messages (what traceroute/mtr uses) while still forwarding packets normally. If all subsequent hops are reachable with

> 0% loss, the "100%" hop is just filtering probes. Only worry if loss appears at your destination or persists across multiple subsequent hops.

Testing DNS resolution time

```
# How long does a DNS lookup take?
$ dig google.com | grep "Query time"
;; Query time: 2 msec   # Good — local DNS cache hit

$ dig google.com | grep "Query time"
;; Query time: 342 msec  # Bad — slow or overloaded DNS resolver

# Test a specific DNS server directly (bypasses system resolver)
$ dig @8.8.8.8 google.com | grep "Query time"

# For HTTP apps — break down where the time goes with curl:
$ curl -o /dev/null -s -w "DNS: %{time_namelookup}s\nConnect: %{time_connect}s\nTTFB: %{time_start
DNS:     0.342s   ← DNS lookup taking 342ms — this is the problem
Connect: 0.344s
TTFB:    0.489s
Total:   0.490s
# The curl timing breakdown instantly shows which phase is slow.
# Here DNS (name lookup) is taking 342ms — everything else is fine.
```

ss — The Modern netstat

`ss` (socket statistics) replaced `netstat` as the recommended tool for inspecting network connections. It's faster, more informative, and available on all modern Linux systems. The flags are similar to netstat but the output is richer.

```
# The most useful ss commands

# What's listening on which port, and which process owns it?
$ ss -tulpn
Netid  State   Recv-Q Send-Q  Local Address:Port   Peer Address:Port Process
tcp    LISTEN  0      128     0.0.0.0:22           0.0.0.0:*     users:(("sshd",pid=891))
tcp    LISTEN  0      511     0.0.0.0:80           0.0.0.0:*     users:(("nginx",pid=1234))
tcp    LISTEN  0      128     127.0.0.1:5432       0.0.0.0:*     users:(("postgres",pid=2341))
# -t TCP  -u UDP  -l listening  -p show process  -n no DNS lookup

# Summary statistics — all TCP states at a glance
$ ss -s
Total: 4821
TCP:   4701 (estab 241, closed 4120, orphaned 0, timewait 4112)
# 4112 TIME_WAIT out of 4701 total TCP sockets — high, worth investigating (Scenario 2)

# All established connections with process info
$ ss -tp state established
Recv-Q Send-Q  Local Address:Port    Peer Address:Port  Process
0      0       10.0.0.5:44321       10.0.0.10:5432     users:(("python3",pid=8821))
0      0       10.0.0.5:44322       10.0.0.10:5432     users:(("python3",pid=8821))
# python3 (PID 8821) has two connections to PostgreSQL (port 5432)

# Count connections per state
$ ss -tan | awk 'NR>1 {print $1}' | sort | uniq -c | sort -rn
   4112 TIME_WAIT
    241 ESTABLISHED
      8 LISTEN

# Find connections to/from a specific port or address
$ ss -tp dst 10.0.0.10:5432     # connections TO this PostgreSQL server
$ ss -tp sport :80              # connections FROM port 80 (web server outbound)
```

> **Recv-Q and Send-Q in ss output:** For LISTEN sockets, Recv-Q is the number of connections waiting to be accepted (should be near 0; a large value means the application isn't calling accept() fast enough). For established sockets, Send-Q is data buffered waiting to be sent to the remote end — a large Send-Q means the remote side is reading slowly or the connection is congested.

TCP Connection States

LISTEN NORMAL

Server socket waiting for incoming connections. Should exist for every service port. The Recv-Q shows the backlog queue — connections waiting to be accepted by the application.

ESTABLISHED NORMAL

Active two-way connection. Both sides can send data. The number of established connections reflects your actual active users or service-to-service connections right now.

TIME_WAIT WATCH

Connection has been closed. The kernel holds the socket for 60 seconds (2×MSL) to absorb any late-arriving packets. Normal in small numbers; thousands indicates high connection turnover or missing keep-alive.

CLOSE_WAIT APP BUG

SYN_SENT / SYN_RECV

FIN_WAIT1 / FIN_WAIT2

Remote end sent FIN (closed its side), but the local application hasn't called close() yet. A large and growing CLOSE_WAIT count is almost always an application bug — sockets not being closed after use.

TCP handshake in progress. SYN_SENT = local side waiting for remote to respond. SYN_RECV = server received SYN, waiting for ACK. Many SYN_RECV can indicate a SYN flood attack.

Connection teardown in progress — local side initiated the close. Brief transitional states. Many FIN_WAIT2 with no progression to TIME_WAIT can indicate the remote end is not responding to the close sequence.

iftop and nethogs — Bandwidth by Connection and by Process

iftop — bandwidth by connection pair

Shows which **source→destination pairs** are consuming bandwidth. Answers: "which remote host am I talking to most?" Useful when you suspect a specific remote host is involved.

`iftop -i eth0` — specify interface
`iftop -n` — don't resolve hostnames
`iftop -P` — show ports
`iftop -B` — show bytes not bits

Keys in interactive mode: `t` toggle TX/RX display · `s` show source · `d` show destination · `p` show ports · `q` quit

nethogs — bandwidth by process

Shows which **process** is consuming bandwidth. The network equivalent of iotop. Answers: "which application on this server is sending/receiving the most data?"

`nethogs eth0` — monitor one interface
`nethogs -d 1` — update every 1 second
`nethogs -b` — batch mode for scripts

Keys in interactive mode: `m` cycle display units (KB/MB/B) · `r` sort by received · `s` sort by sent · `q` quit

Requires root or `CAP_NET_ADMIN`.

```
# nethogs output — which process is using bandwidth right now?
$ nethogs -d 1 eth0
NetHogs version 0.8.5
  PID USER    PROGRAM                  DEV    SENT     RECEIVED
 8821 backup  /usr/bin/rsync           eth0   94.2     0.1      MB/s
  891 www     /usr/sbin/nginx          eth0    8.4      2.1      MB/s
 2341 mysql   /usr/sbin/mysqld         eth0    0.8      0.3      MB/s
# rsync (backup) is sending 94 MB/s — near the 1 Gbps NIC limit
# nginx is doing 8 MB/s — normal for a web server
# On a 1Gbps NIC: 125 MB/s max. rsync is consuming 75% of available bandwidth.
```

/proc/net/dev — NIC Errors and Drops

NIC-level errors are distinct from application-level bandwidth saturation. Hardware errors (CRC failures, framing errors) indicate a physical problem — bad cable, faulty NIC, misconfigured duplex. Drops indicate the kernel couldn't process packets fast enough.

```
$ cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo c
  eth0: 82341M 61234K    0    0    0     0          0    4821K 52341M  48923K    0    0    0     0
  eth1: 12341M 18234K  142   891    0    12          0       0  8234M  12821K    0    0    0     0
# eth0: clean — zero errors and drops on both receive and transmit
# eth1: 142 receive errors + 891 drops + 12 frame errors — investigate this NIC/cable

# More readable with ip -s link:
$ ip -s link show eth1
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
    RX: bytes    packets  errors  dropped  missed  mcast
        12341M   18234K   142     891      0       0
    TX: bytes    packets  errors  dropped carrier  collisions
        8234M    12821K   0       0       0        0
```

FIELD	MEANING	NON-ZERO MEANS...
`RX errors`	Packets received with hardware errors (CRC, framing, length)	**Physical problem: bad cable, NIC fault, duplex mismatch. Replace cable first.**
`RX dropped`	Received packets discarded by the kernel before processing	**Ring buffer overflow — NIC received faster than the kernel could process. Increase ring buffer size: `ethtool -G eth0 rx 4096`**
`RX missed`	Packets missed by the NIC hardware before they reached the ring buffer	**NIC is hardware-saturated. Interrupt coalescing or RSS (Receive Side Scaling) may help.**
`TX carrier`	Lost carrier signal during transmit — link went down mid-send	**Physical link instability — cable, switch port, or NIC issue**
`TX collisions`	Ethernet collisions (half-duplex only)	**Should be zero on modern full-duplex links. Non-zero = duplex mismatch with switch.**

Scenario 1 — App Is Slow but CPU and Memory Are

Idle

SCENARIO · CHAPTER 5 · SCENARIO 1

Web app response times are 2–3 seconds. htop shows CPU at 5% and memory is fine. Where to look?

1 **Confirm with vmstat that it's not disguised I/O wait.** Network blocking shows as idle CPU (not wa), making it look like the system is doing nothing when it's actually waiting on network responses.

```
$ vmstat 1 5
 r  b  swpd  free   si  so   bi   bo   wa   us  sy  id
 0  0     0  8.2G    0   0    0    0    0    5   2  93
# id=93: CPU is 93% idle. No I/O wait. No swap. System appears completely idle
# but the app is slow. The bottleneck is not a local resource — it's network or
# external service latency (database server, API, DNS).
```

2 **Test DNS resolution time — this is the most commonly missed culprit.**

```
$ time dig db-server.internal A
;; Query time: 380 msec
;; SERVER: 10.0.0.53#53
# DNS is taking 380ms. If the app resolves "db-server.internal" on every request,
# that's 380ms of latency before a single byte of database query is sent.
# Fix: add db-server.internal to /etc/hosts, or fix the internal DNS server.

$ cat /etc/hosts | grep db-server   # quick fix: static resolution
$ echo "10.0.0.10 db-server.internal" | sudo tee -a /etc/hosts
```

3 **Use curl timing to break down an HTTP request into phases.**

```
$ curl -o /dev/null -s -w \
  "DNS lookup:    %{time_namelookup}s\nTCP connect:   %{time_connect}s\nSSL handshake: %{time_appconnect}s\nTime
  https://api.example.com/health
DNS lookup:    0.002s   ← fast, cached
TCP connect:   0.015s   ← 13ms to connect, reasonable
SSL handshake: 1.240s   ← 1.2 seconds for TLS — this is the bottleneck
Time to first byte: 1.250s
Total:         1.252s
# TLS handshake taking 1.2 seconds. Possible causes:
# - Server is doing slow certificate validation (OCSP stapling not configured)
# - Client-side certificate revocation check over slow network
# - Session resumption not working (full handshake every time)
```

4 **Check for CLOSE_WAIT accumulation — a sign the app isn't closing connections.**

```
$ ss -tan | awk '{print $1}' | sort | uniq -c
    4 LISTEN
   12 ESTABLISHED
  820 CLOSE_WAIT
# 820 CLOSE_WAIT: the remote side has closed connections but the app hasn't.
# This means the app is holding onto dead connections. Each thread/worker waiting
# on a CLOSE_WAIT socket is blocked. This directly causes slow responses.
# Fix: find the bug in the application code that's not calling close() after use.

# Find which local process owns the CLOSE_WAIT sockets:
$ ss -tanp state CLOSE_WAIT | awk '{print $NF}' | sort | uniq -c | sort -rn | head -5
    820 users:(("node",pid=8821,fd=42))
```

5 **Check NIC errors — hardware problems can cause intermittent slowness.**

```
$ ip -s link show eth0 | grep -A2 "RX:"
    RX: bytes    packets  errors  dropped
        82341M   61234K   1842    234
# 1842 receive errors and 234 drops — hardware-level problem on this NIC.
# Check the cable, the switch port, and negotiate duplex settings.
# ethtool eth0 | grep -E "Speed|Duplex" — confirm link speed and duplex.
$ ethtool eth0 | grep -E "Speed|Duplex"
        Speed: 100Mb/s   ← Should be 1000Mb/s — NIC negotiated at wrong speed
        Duplex: Half     ← Half duplex — bad, should be Full. Explains collisions.
```

The curl timing breakdown is one of the most productive 30-second investments when diagnosing slow HTTP applications. DNS → Connect → TLS → TTFB each map to a specific layer you can investigate independently.

Scenario 2 — Thousands of TIME_WAIT Connections

SCENARIO · CHAPTER 5 · SCENARIO 2

ss -s shows 8,000 TIME_WAIT sockets. Is this a problem, and should it be tuned?

1. **Understand why TIME_WAIT exists before deciding to fight it.** TIME_WAIT is the TCP protocol's safety mechanism — after a connection closes, the kernel holds the port combination for 60 seconds to absorb any late-arriving packets that were delayed in transit. It prevents a new connection on the same port from receiving old data. Removing it entirely is dangerous.

2. **Determine whether it's actually causing a problem.** TIME_WAIT is only a problem if you're running out of ephemeral ports — the pool of local ports used for outgoing connections.

```
# What's the ephemeral port range?
$ cat /proc/sys/net/ipv4/ip_local_port_range
32768   60999
# 28,231 available local ports for outgoing connections.
# If more than ~28,000 TIME_WAIT sockets exist to the SAME destination,
# new connections to that destination will fail with "cannot assign address".

# Are we actually failing to create new connections?
$ dmesg -T | grep -i "port\|connect\|socket"
$ netstat -s | grep -i "failed\|refused\|exhausted"
```

3. **If TIME_WAIT is genuinely causing port exhaustion — tune carefully.**

```
# Option 1: Enable TIME_WAIT socket reuse (safe — only for outgoing connections)
$ sysctl -w net.ipv4.tcp_tw_reuse=1
# Allows reusing TIME_WAIT sockets for new outgoing connections when safe.
# This is the recommended setting for servers making many outgoing connections.

# Option 2: Widen the ephemeral port range
$ sysctl -w net.ipv4.ip_local_port_range="1024 65535"
# Gives ~64,000 local ports instead of ~28,000. Simple and safe.

# Option 3: Enable keep-alive on HTTP connections (fix the root cause)
# In nginx — ensure keep-alive is not disabled:
keepalive_timeout 65;    # in nginx.conf http {} block
# With keep-alive, the same TCP connection handles multiple HTTP requests,
# dramatically reducing connection turnover and TIME_WAIT accumulation.

# Make sysctl changes permanent:
$ echo "net.ipv4.tcp_tw_reuse=1" | sudo tee -a /etc/sysctl.d/99-network.conf
$ sysctl -p /etc/sysctl.d/99-network.conf
```

4. **Do not use tcp_tw_recycle.** This parameter was removed in Linux kernel 4.12 because it broke connections from clients behind NAT (a common scenario with load balancers and mobile networks). If you see advice recommending it for TIME_WAIT, the advice is outdated and potentially harmful.

8,000 TIME_WAIT sockets with a 28,000-port range is perfectly healthy — you have headroom. TIME_WAIT only warrants action when either the count approaches your port range limit, or you're seeing actual "cannot assign requested address" errors in application logs or dmesg.

Scenario 3 — One Process Is Saturating the NIC

SCENARIO · CHAPTER 5 · SCENARIO 3

The server's 1 Gbps NIC is pegged. Other services are suffering. A process is consuming nearly all bandwidth.

1 **Confirm NIC saturation and identify the interface.**

```
# Watch interface throughput in real time
$ watch -n 1 'cat /proc/net/dev | awk "/eth0/{print \"RX: \" $2/1048576 \" MB total | TX: \" $10/1048576 \" MB to

# Or more elegantly with ip:
$ ip -s -s link show eth0 | grep -A4 "TX:"
# Run twice 1 second apart to see the delta (bytes/second)

# iftop shows bandwidth by connection — which remote host is involved?
$ iftop -i eth0 -n -B -P
                   12.5MB          25.0MB          37.5MB          50MB     62.4MB
10.0.0.5  =>  192.168.50.20:873   89.4Mb   91.2Mb   87.8Mb
            <=                   0.12Mb    0.14Mb    0.13Mb
# Port 873 = rsync. Sending ~90 Mbps to 192.168.50.20 — the backup server.
# This is consuming ~90% of the 100Mbps link being used here.
```

2 **Confirm the responsible process with nethogs.**

```
$ nethogs -d 1 eth0
  PID USER    PROGRAM              SENT      RECEIVED
 9821 backup /usr/bin/rsync       94.2 MB/s  0.1 MB/s
  891 www     nginx                8.4 MB/s  2.1 MB/s
# Confirmed: PID 9821 (rsync backup) is the culprit
```

3 **Throttle rsync's bandwidth directly.** rsync has a built-in bandwidth limit flag — if you have control over how it's invoked, this is the cleanest solution.

```
# Kill the current rsync and restart it with a bandwidth limit
$ kill 9821
$ rsync -av --bwlimit=20480 /data /mnt/backup   # limit to 20 MB/s (20480 KB/s)
```

4 **For processes without built-in throttling — use trickle or tc.**

```
# trickle: simple per-process bandwidth shaping (no root required)
# apt install trickle  or  yum install trickle
$ trickle -u 20480 rsync -av /data /mnt/backup
# -u 20480 = limit upload to 20480 KB/s (20 MB/s)
# -d 20480 = limit download
# Works by intercepting socket calls — no kernel changes needed

# tc (traffic control): kernel-level interface shaping (affects ALL traffic on interface)
# Add a queuing discipline limiting the interface to 100 Mbit total
$ tc qdisc add dev eth0 root tbf rate 100mbit burst 32kbit latency 400ms
$ tc qdisc show dev eth0   # verify it's applied
$ tc qdisc del dev eth0 root   # remove the limit when done
# tc affects the entire interface — use trickle or rsync --bwlimit for per-process control
```

5 **Long-term fix: schedule bandwidth-heavy jobs during off-peak hours.**

```
# crontab -e — run backup at 2am with bandwidth limit
0 2 * * * /usr/bin/rsync -av --bwlimit=51200 /data /mnt/backup >> /var/log/backup.log 2>&1
# --bwlimit=51200 = 50 MB/s at 2am when the server is quiet
# Even at 50 MB/s, 1 TB backup completes in ~6 hours
```

If the high-bandwidth process is a production service rather than a backup job, investigate whether it's doing unnecessary data transfer (missing caching, missing compression, pulling full datasets when it only needs diffs) before throttling it — throttling a production service degrades its performance for users.

Quick Reference — Chapter 5 Commands

COMMAND	PURPOSE	KEY FLAGS / NOTES
`ping -c 10 host`	Basic connectivity test and round-trip time. Watch for packet loss and jitter (mdev).	High mdev = jitter = congestion somewhere in path
`mtr --report -n host`	Combined traceroute + ping — shows loss% at every hop over multiple cycles	`--report-cycles 20` for 20 probes · 100% loss at intermediate hop is usually normal
`dig hostname`	DNS lookup — check "Query time:" for DNS latency	`dig @8.8.8.8 host` to test a specific resolver
`curl -w "..." url`	Break HTTP request into phases: DNS / Connect / TLS / TTFB / Total	Use the timing format string from Scenario 1 step 3
`ss -tulpn`	Listening ports with owning process — the first check for "what's running"	`-t` TCP · `-u` UDP · `-l` listening · `-p` process · `-n` no DNS
`ss -s`	Connection state summary — quick view of ESTABLISHED, TIME_WAIT, CLOSE_WAIT counts	Many CLOSE_WAIT = app bug. Many TIME_WAIT = high connection turnover.
`ss -tan | awk '{print $1}' | sort | uniq -c`	Count connections per state	Add `state CLOSE_WAIT` to filter to one state
`ip -s link show eth0`	NIC statistics including RX/TX errors, drops, missed packets	Non-zero errors = physical problem. Non-zero drops = ring buffer overflow.
`ethtool eth0`	NIC link status — speed and duplex negotiation	Speed should be 1000Mb/s+, Duplex should be Full on modern links
`iftop -i eth0 -n`	Live bandwidth by source→destination connection pair	`-B` bytes · `-P` show ports · `-n` no DNS
`nethogs eth0`	Live bandwidth by process — the iotop equivalent for network	`-d 1` update every 1s · requires root
`rsync --bwlimit=20480`	Limit rsync bandwidth to 20 MB/s (20480 KB/s)	Built-in to rsync — cleanest solution when rsync is the culprit
`trickle -u 20480 cmd`	Per-process bandwidth cap for any command — no root needed	`-u` upload limit · `-d` download limit (in KB/s)
`sysctl net.ipv4.tcp_tw_reuse=1`	Allow reuse of TIME_WAIT sockets for new outgoing connections	Persist via `/etc/sysctl.d/` · safe, unlike the removed

COMMAND	PURPOSE	KEY FLAGS / NOTES
		tcp_tw_recycle

CHAPTER 6 OF 8

Chapter 6 — Process Management & Control

Understanding processes in depth is what separates an administrator who reacts to problems from one who controls them. This chapter covers finding, inspecting, and terminating processes safely — including the cases where the obvious approach (kill -9) doesn't work and you need to understand why.

> **What this chapter covers:** Reading ps output and process trees. pgrep and pstree for finding processes. Signals — SIGTERM, SIGKILL, SIGHUP, SIGUSR — what they mean and in what order to use them. Process groups and why killing the parent isn't always enough. pkill -P for process trees. strace and lsof -p for inspecting stuck processes. systemctl's signal control. Scenario 1: a script spawned 200 workers. Scenario 2: a process ignores SIGTERM. Scenario 3: kill -9 did nothing — uninterruptible D-state explained.

ps — Reading Process Output

`ps` snapshots the current process table. Unlike top/htop it doesn't update — it captures the state at the moment you run it, making it useful for scripting and for examining a specific process without the noise of a live display.

```
# The standard all-process listing
$ ps aux
USER       PID  %CPU %MEM    VSZ    RSS TTY      STAT START   TIME COMMAND
root         1   0.0  0.1 167852  9248 ?        Ss   Jun01   0:04 /sbin/init
postgres  2341   0.3  2.1 312148 87321 ?        Ss   Jun01  12:44 postgres: writer process
www-data  8821  98.0  0.4  12341  4182 ?        R    14:30   0:42 python3 worker.py
# VSZ = virtual memory size (total address space allocated)
# RSS = resident set size (memory actually in RAM)   ← the useful number
# STAT = process state (see state column below)
# TIME = total CPU time consumed since process started (not "right now")

# Forest view — shows parent/child relationships visually
$ ps faux
USER       PID %CPU %MEM   VSZ   RSS STAT COMMAND
root      1234  0.0  0.1  8234  4121 Ss   bash run_jobs.sh
root      1235  0.4  0.2  9341  4822 S     \_ python3 worker.py --id=1
root      1236  0.4  0.2  9341  4822 S     \_ python3 worker.py --id=2
root      1237  0.4  0.2  9341  4822 S     \_ python3 worker.py --id=3
# The \_ indentation shows these python3 workers are children of the bash script (PID 1234)

# Get a specific PID's details — including its parent
$ ps -o pid,ppid,stat,rss,comm -p 8821
  PID  PPID STAT   RSS COMM
 8821  1234 R    4182 python3
# PPID=1234: this python3 process's parent is PID 1234 (the bash script)

# Sort by CPU or memory usage
$ ps aux --sort=-%cpu | head -10   # top 10 by CPU
$ ps aux --sort=-%mem | head -10   # top 10 by memory
```

Process states — the STAT column

CODE	STATE	WHAT IT MEANS	ACTION IF STUCK
R	Running	Currently executing on CPU, or runnable and waiting for a CPU core. High %CPU in top correlates with R.	**Normal — investigate only if unexpected**
S	Sleeping	Waiting for an event (timer, network, user input). Most idle processes sit here. Interruptible — signals are delivered normally.	**Normal idle state**
D	Uninterruptible sleep	Waiting for a kernel I/O operation to complete. Cannot receive signals — SIGKILL does nothing. Usually brief; if persistent, the I/O source is hung.	**Fix the hung resource — often NFS or failing disk. May require reboot.**
Z	Zombie	Process has exited but parent hasn't called wait() to collect exit status. No resources used (no memory, no CPU). Just a row in the process table.	**Kill or restart the parent. kill -9 has no effect on the zombie itself.**
T	Stopped	Suspended — received SIGSTOP or Ctrl+Z. Not running, not consuming CPU. Waiting to be resumed (SIGCONT) or terminated.	Send SIGCONT to resume, or SIGKILL to terminate
s	Session leader	Lowercase modifier. This process is the leader of its session (usually the shell that started a job group).	Informational
l	Multi-threaded	Lowercase L. Process has multiple threads. Kill the process PID, not individual thread PIDs.	Informational

pgrep and pstree

```
# pgrep — find PIDs by process name (faster than ps | grep)
$ pgrep python3             # PIDs of all processes named python3
$ pgrep -c python3          # count of matching processes
$ pgrep -l python3          # PID + name
$ pgrep -a python3          # PID + full command line (like ps -f)
$ pgrep -f "worker.py"      # match against full command line, not just name
$ pgrep -u www-data python3 # only processes owned by www-data

# pstree — visual process tree
$ pstree -p          # full tree with PIDs
$ pstree -p 1234     # tree rooted at PID 1234
bash(1234)─┬─python3(1235)
           ├─python3(1236)
           └─python3(1237)
```

Signals — What They Are and Which to Use

A signal is a software interrupt sent to a process. The kernel delivers it asynchronously — the process can be in the middle of any operation when it arrives. Each signal has a default action, but a process can install a custom handler (catching the signal and doing something else) or explicitly ignore it. The one exception is SIGKILL — it cannot be caught, blocked, or ignored under any circumstances.

15 SIGTERM

Polite termination request. The default signal sent by `kill PID` and `systemctl stop`. A well-behaved process catches this, finishes in-flight work, flushes buffers, removes PID files, and exits cleanly. A process *can* ignore or delay responding to SIGTERM — it is just a request.

9 SIGKILL

Unconditional force kill. Cannot be caught, blocked, or ignored — the kernel enforces it directly. The process has no chance to clean up: open files are closed without flushing, database transactions are abandoned, PID files are left on disk. Use as a last resort after SIGTERM fails.

1 SIGHUP

Hangup / reload config. Originally meant "the terminal disconnected." Modern daemons (nginx, sshd, rsyslog) use it as "re-read your configuration file without restarting." Check the application's documentation before sending — some processes do treat it as terminate.

2 SIGINT

Interrupt (Ctrl+C). Sent to a foreground process when you press Ctrl+C in the terminal. Interruptible — a process can catch it. Many programs treat SIGINT the same as SIGTERM but exit immediately rather than draining work.

10/12 SIGUSR1 / SIGUSR2

User-defined — application-specific. No default behaviour. Each application decides what these mean. nginx: SIGUSR1 = reopen log files (use for log rotation). Apache: SIGUSR1 = graceful restart. PostgreSQL: SIGUSR1 = various. Always check the application's docs.

18/19 SIGCONT / SIGSTOP

Resume / Suspend. SIGSTOP (like Ctrl+Z) suspends a process — it enters T state and consumes no CPU. SIGCONT resumes it from where it stopped. Useful for temporarily pausing a CPU-intensive process. SIGSTOP also cannot be caught or ignored.

The correct order — try SIGTERM before SIGKILL

1. Try SIGTERM first (always)

`kill PID` — sends SIGTERM (signal 15 by default)

Wait **10–30 seconds**. A well-behaved process will:
— Finish the current request
— Flush write buffers
— Close database connections cleanly
— Remove PID/lock files
— Exit with a meaningful exit code

For services, `systemctl stop nginx` sends SIGTERM and waits up to 90 seconds before giving up.

2. Only then — SIGKILL

`kill -9 PID` or `kill -SIGKILL PID`

Use only after SIGTERM has not worked after a reasonable wait. Consequences:
— Partially written files may be corrupt
— Database WAL/journals may need recovery on restart
— PID files, lock files, and socket files may be left on disk
— Temporary files in /tmp may not be cleaned up

If SIGKILL also has no effect → the process is in **D state** (see Scenario 3).

```
# Sending signals
$ kill PID          # SIGTERM (15) — the polite default
$ kill -9 PID       # SIGKILL — force kill
$ kill -1 PID       # SIGHUP — reload config
$ kill -SIGTERM PID # same as kill PID, but explicit
$ kill -l           # list all signal names and numbers

# pkill — send signal by name instead of PID
$ pkill python3          # SIGTERM to all processes named python3
$ pkill -9 python3       # SIGKILL to all processes named python3
$ pkill -f "worker.py"   # match full command line
$ pkill -u www-data      # terminate all processes owned by www-data

# Verify a process is gone
$ kill -0 PID        # "signal 0" — no signal sent, just checks if PID exists
bash: kill: (8821) - No such process   ← process is gone
$ ps -p PID          # no output = process no longer exists
```

Process Groups — Killing a Whole Tree

Every process belongs to a **process group** (PGID). When you run a script, the script and all the processes it spawns typically share the same process group. This makes it possible to terminate an entire job tree in one command without having to enumerate every child PID.

```
Process group PGID=1234
┌──────────────────────────────────────┐
│                                      │
│  bash run_jobs.sh   PID=1234         │
│       │                              │
│       ├── python3 worker.py  1235    │
│       ├── python3 worker.py  1236    │
│       │        └── gzip       1298   │
│       └── python3 worker.py  1237    │
│                                      │
└──────────────────────────────────────┘

kill 1234           → SIGTERM to the bash script only (parent)
pkill -P 1234       → SIGTERM to direct children of PID 1234 (1235, 1236, 1237)
kill -- -1234       → SIGTERM to every process in PGID 1234 (all of the above)
kill -9 -- -1234    → SIGKILL to entire process group
```

```
# Find the process group ID (PGID) of a process
$ ps -o pid,pgid,ppid,comm -p 1235
  PID  PGID  PPID COMM
 1235  1234  1234 python3
# PGID=1234: same as the parent bash script. Killing PGID 1234 kills everything.

# Kill direct children of a parent PID (one level deep)
$ pkill -P 1234               # SIGTERM to all children of PID 1234
$ pkill -9 -P 1234            # SIGKILL to all children of PID 1234

# Kill entire process group (parent + all descendants at all levels)
$ kill -- -1234               # the -- separates options from the negative PGID
$ kill -9 -- -1234            # SIGKILL to entire group

# For a service: systemctl knows the full cgroup tree
$ systemctl stop myapp        # cleanly stops all processes in the service's cgroup
$ systemctl kill --signal=SIGHUP nginx   # send custom signal to a service
```

> **Why killing the parent often isn't enough:** If you kill the parent script with SIGTERM, the parent exits but does not automatically kill its children — they become orphaned and are re-parented to PID 1 (systemd). They keep running. Use `pkill -P` or `kill -- -PGID` to reach the whole tree.

strace and lsof -p — Inspecting a Stuck Process

strace — what system calls is the process making?

strace intercepts and logs every system call (read, write, open, connect, etc.) a process makes. Attach to a running process with `-p`. Useful for answering "why is this process hung — what is it waiting for?"

`strace -p PID` — attach to running PID
`strace -p PID -e trace=network` — only network calls
`strace -p PID -e trace=file` — only file-related calls
`strace -p PID -T` — show time spent in each call
`strace -p PID -tt` — show wall-clock timestamp per call

Important: strace adds overhead (~20–30% CPU cost) to the traced process. Don't leave it attached on production long-term.

lsof -p — what resources does the process hold?

lsof (list open files) lists every file descriptor a process has open — including regular files, directories, sockets, pipes, and devices.

`lsof -p PID` — all open files for one PID
`lsof -p PID | wc -l` — count open file descriptors
`lsof -p PID | grep IPv4` — network connections only
`lsof -p PID | grep REG` — regular files only
`lsof -p PID | grep deleted` — open-but-deleted files

For "too many open files" errors: `lsof -p PID | wc -l` vs `/proc/PID/limits | grep "open files"` to see the hard limit.

```
# strace — diagnosing why a process is stuck
$ strace -p 8821
strace: Process 8821 attached
read(5, ^             ← blocked on read() of file descriptor 5 — waiting for data

# Which file is fd=5?
$ ls -la /proc/8821/fd/5
lrwxrwxrwx 1 www-data www-data 0 Jun 14 14:30 /proc/8821/fd/5 -> socket:[234821]
# fd 5 is a socket — the process is blocked waiting for data from a network connection.
# This could be a database query that never returned, or a hung API call.

$ strace -p 8821 -T 2>&1 | head -30
futex(0x7f..., FUTEX_WAIT, 1, NULL)  = 0 <0.000001>
futex(0x7f..., FUTEX_WAIT, 1, NULL)  = 0 <0.000001>
# futex (fast mutex) calls taking ~0ms each = thread waiting on a lock.
# This is normal for multi-threaded idle processes.

# lsof — inventory of what a process has open
$ lsof -p 8821
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
python3 8821 www    cwd   DIR    8,1     4096       2 /opt/app
python3 8821 www    txt   REG    8,1    19504 2345678 /usr/bin/python3
python3 8821 www    mem   REG    8,1   167984 1234567 /lib/x86_64-linux-gnu/libc.so
python3 8821 www     0u   CHR    1,3      0t0       5 /dev/null
python3 8821 www     1u   REG    8,1    45890 9876543 /var/log/app/app.log
python3 8821 www     4u  IPv4 234821      0t0     TCP 10.0.0.5:44321->10.0.0.10:5432 (ESTABLISHED)
# fd 4 is a TCP connection to 10.0.0.10:5432 — PostgreSQL. Still ESTABLISHED.
```

Scenario 1 — A Script Spawned 200 Workers

SCENARIO · CHAPTER 6 · SCENARIO 1

A cron job or manual run spawned 200 python3 worker processes. The terminal is gone. How do you find them, confirm they belong to the right job, and stop them safely?

1 **Confirm what you're dealing with.**

```
# How many processes match?
$ pgrep -c -f "worker.py"
203

# See their full command lines — are they all the same job?
$ pgrep -af "worker.py" | head -10
1235 python3 /opt/app/worker.py --job-id=5a12b --queue=default
1236 python3 /opt/app/worker.py --job-id=5a12b --queue=default
1237 python3 /opt/app/worker.py --job-id=5a12b --queue=default
# Same --job-id. All 203 belong to the same batch job.

# Who is the parent? (look at one PID's PPID)
$ ps -o pid,ppid,stat,comm -p 1235
  PID  PPID STAT COMM
 1235  1234 S    python3
$ ps -o pid,ppid,stat,comm -p 1234
  PID  PPID STAT COMM
 1234     1 S    run_jobs.sh
# Parent PID is 1234 (a bash script), which itself is owned by PID 1 (systemd).
# The original shell that launched it has already exited.
```

2 **Check resource impact before deciding on an approach.**

```
$ ps aux --sort=-%cpu | grep worker.py | awk '{cpu+=$3; mem+=$4; count++} END {print count " workers using " cpu
203 workers using 812.4% CPU and 6.2% MEM
# 812% CPU across all 203 workers on a 16-core machine = 51% of total CPU capacity
# They're consuming significant resources. Safe to terminate the job.
```

3 **Terminate the parent first — give it a chance to clean up.**

```
# Send SIGTERM to the parent script — it may clean up children itself
$ kill 1234
$ sleep 10
$ pgrep -c -f "worker.py"
203
# Parent exited but workers are still running — they ignored the parent dying.
# The workers are now orphaned (adopted by PID 1 / systemd).
```

4 **Terminate all workers with pkill — match on the full command line.**

```
# SIGTERM to all matching workers first (polite)
$ pkill -TERM -f "worker.py --job-id=5a12b"
$ sleep 15   # wait for graceful shutdown
$ pgrep -c -f "worker.py"
12   # 12 processes didn't respond to SIGTERM

# SIGKILL the stragglers
$ pkill -9 -f "worker.py --job-id=5a12b"
$ pgrep -c -f "worker.py"
0   # all gone
```

5 **Alternative: if they all share a process group, one command kills the tree.**

```
# Check the process group ID
$ ps -o pgid= -p 1235 | head -1
1234

# Kill the entire process group in one shot
$ kill -- -1234        # SIGTERM to all 203 workers + parent at once
$ sleep 10
$ kill -9 -- -1234     # SIGKILL stragglers
```

Always use `-f` with pkill to match against the full command line — matching just the process name risks catching unrelated processes with the same name (e.g., pkill python3 would kill all Python processes on the server). The more specific your match, the safer.

Scenario 2 — Process Ignores SIGTERM

SCENARIO · CHAPTER 6 · SCENARIO 2

You ran kill 8821 thirty seconds ago. ps still shows the process running. What now?

1 **Check if the process is actually in D state — if so, SIGKILL won't help either.**

```
$ ps -o pid,stat,comm -p 8821
  PID STAT COMM
 8821 S    python3
# STAT=S (Sleeping) — it CAN receive signals. It's choosing not to respond to SIGTERM.
# This is different from D state (which physically cannot receive signals).
```

2 **Use strace to see what it's doing — is it stuck or deliberately ignoring?**

```
$ strace -p 8821 -T 2>&1 | head -20
read(5,  ← blocked waiting for data on fd 5 (a socket — database connection)
# The process is waiting on a slow database query. It installed a SIGTERM handler
# that sets a flag, but the flag is only checked AFTER the current read() returns.
# Once the database responds (or times out), the process will honour the SIGTERM.
# Wait longer — or check if the database is hung (and fix that instead).
```

3 **If it's genuinely ignoring SIGTERM and needs to be stopped now — use SIGKILL.**

```
$ kill -9 8821
$ sleep 2
$ kill -0 8821
bash: kill: (8821) - No such process
# SIGKILL worked — process is gone.
# Now check for leftover resources that it didn't clean up:

# PID file?
$ find /var/run -name "*.pid" -newer /proc/uptime 2>/dev/null

# Lock files?
$ find /var/lock -name "*.lock" 2>/dev/null

# Temp files?
$ ls -lt /tmp | head -20   # recently modified files in /tmp
```

4 **For services managed by systemd — let systemctl handle the sequence for you.**

```
# systemctl stop sends SIGTERM, waits TimeoutStopSec (default 90s), then sends SIGKILL
$ systemctl stop myapp
$ systemctl status myapp   # check result — should say "inactive (dead)"

# If 90 seconds is too long to wait, reduce the timeout for this stop:
$ systemctl stop --timeout=10 myapp

# Check how long systemd waited (the logs will show):
$ journalctl -u myapp --since "5 minutes ago" | grep -E "stop|kill|timeout"
```

Many well-written services intentionally delay SIGTERM — they finish an in-flight transaction or drain a queue before exiting. This is correct behaviour. The question to ask before reaching for SIGKILL is: "is this process making progress, or is it genuinely stuck?" strace answers that in seconds.

Scenario 3 — kill -9 Did Nothing

SCENARIO · CHAPTER 6 · SCENARIO 3

kill -9 8821 ran with no error. ps still shows the process. It's still there five minutes later. What is happening?

1 **Confirm D state — SIGKILL is undeliverable to a process in uninterruptible sleep.**

```
$ ps -o pid,stat,comm -p 8821
  PID STAT COMM
 8821 D    python3
# STAT=D: Uninterruptible sleep. The process is executing a kernel I/O operation
# and cannot be interrupted until the kernel gives control back.
# SIGKILL has been queued — it will be delivered the moment the process
# leaves D state. If it never leaves D state, the kill never lands.

# How long has it been in D state?
$ ps -o pid,stat,etimes,comm -p 8821
  PID STAT ELAPSED COMM
 8821 D       1842 python3
# ELAPSED=1842 seconds (~30 minutes) in D state — this I/O will never complete.
```

2 **Identify what I/O the process is waiting on.**

```
# Check what the kernel backtrace shows — what syscall is stuck?
$ cat /proc/8821/wchan
nfs_wait_on_request
# wchan = "wait channel" — the kernel function where the process is blocked.
# "nfs_wait_on_request" = waiting for an NFS server to respond. Classic D-state cause.

# Other common wchan values:
#   jbd2_log_wait_commit  → waiting for ext4 journal commit (disk problem)
#   blk_wait_io           → waiting for block device I/O (disk problem)
#   __refrigerator        → process is being frozen (systemd suspend/cgroup freeze)
#   futex_wait            → waiting on a mutex (normal for threads)

# Also check dmesg for related messages
$ dmesg -T | grep -E "NFS|nfs|hung|timeout" | tail -20
[Jun14 14:01] nfs: server 192.168.50.100 not responding, timed out
[Jun14 14:05] nfs: server 192.168.50.100 not responding, timed out
# NFS server at 192.168.50.100 is unreachable. The process is stuck
# waiting for it — and will stay stuck until the NFS situation resolves.
```

3 **Try to resolve the underlying resource — the process will then exit on its own.**

```
# Option 1: If the NFS server came back, the process will resume and the
# queued SIGKILL will be delivered. Monitor with watch:
$ watch -n 2 'ps -o pid,stat,comm -p 8821'

# Option 2: Lazy unmount the NFS filesystem — tells the kernel to disconnect
# the mount once all current I/O is released (safer than umount -f)
$ umount -l /mnt/nfs     # -l = lazy unmount
$ sleep 5
$ ps -o stat -p 8821 | head -2
STAT
S   # back to Sleeping — queued SIGKILL now delivered, process gone in seconds

# Option 3: Force unmount (can cause data loss on any open files on the mount)
$ umount -f -l /mnt/nfs   # -f force + -l lazy — use only when data loss is acceptable
```

4 **If the resource cannot be recovered — a reboot is the only remaining option.**

```
# Confirm nothing else depends on this stuck process before rebooting
$ systemctl list-units --state=failed          # anything already failed?
$ who                                            # anyone else logged in?
$ lsof /mnt/nfs 2>/dev/null | grep -v COMMAND   # other processes using the mount?

# Schedule a clean reboot (waits for current logins to end or timeout)
$ shutdown -r +5 "NFS mount hung, rebooting in 5 minutes. Please save your work."
```

5 **Prevention — mount NFS with a timeout so this can't happen again.**

```
# In /etc/fstab, change a hard NFS mount to soft with a timeout:
# BAD (default — hard mount, hangs indefinitely if NFS dies):
192.168.50.100:/exports /mnt/nfs nfs defaults 0 0

# BETTER — soft mount with timeout (returns EIO to the process after 30 deciseconds):
192.168.50.100:/exports /mnt/nfs nfs soft,timeo=30,retrans=3,_netdev 0 0
# soft   → returns error to the process instead of hanging forever
# timeo  → timeout per retry in deciseconds (30 = 3 seconds per retry)
# retrans=3 → retry 3 times before giving up
# _netdev → wait for network before mounting (important on boot)
```

D-state processes stuck on local disk I/O (not NFS) are rarer and usually indicate a failing drive or a kernel bug. Check `dmesg` for I/O errors, run `smartctl -a /dev/sdX` for SMART data, and check if the filesystem has remounted read-only due to errors (`dmesg | grep "remounting read-only"`).

Quick Reference — Chapter 6 Commands

COMMAND	PURPOSE	NOTES
`ps faux`	Process list with forest/tree view showing parent→child relationships	`ps -o pid,ppid,stat,comm -p PID` for a specific process with PPID
`pgrep -c -f "pattern"`	Count processes matching a full command line pattern	`-a` show full command line · `-l` PID + name · `-u user` filter by owner
`pstree -p PID`	Visual process tree rooted at a specific PID	Omit PID to see the full system tree
`kill PID`	SIGTERM (15) — polite termination request. Try first, always.	`kill -0 PID` checks existence without sending a signal
`kill -9 PID`	SIGKILL — unconditional force kill. Use only after SIGTERM fails.	Cannot kill D-state processes. Look for leftover PID/lock files after.
`kill -1 PID`	SIGHUP — reload configuration without restarting (for daemons)	Check application docs — not universal. nginx/sshd/rsyslog all support it.
`pkill -f "pattern"`	SIGTERM by full command line match — safer than name match alone	`pkill -9 -f` for SIGKILL · `pkill -P PPID` for direct children only
`kill -- -PGID`	SIGTERM to entire process group — kills parent + all descendants	`kill -9 -- -PGID` for SIGKILL · find PGID with `ps -o pgid= -p PID`
`systemctl stop service`	SIGTERM → wait TimeoutStopSec (90s) → SIGKILL. Covers the full cgroup.	`--timeout=10` to override wait · `systemctl kill --signal=SIGHUP` for custom signal
`strace -p PID`	Attach to running process and show every system call in real time	`-T` show time per call · `-e trace=network` filter to network calls only · adds CPU overhead
`lsof -p PID`	List all open files, sockets, and devices held by a process	`lsof -p PID | wc -l` count FDs · `grep deleted` find open-but-deleted files
`cat /proc/PID/wchan`	Which kernel function a D-state process is waiting on	`nfs_wait_on_request` = NFS hang · `jbd2_log_wait_commit` = disk journal · `blk_wait_io` = block I/O
`umount -l /mnt/path`	Lazy unmount — detaches mount once all current I/O completes	Often frees D-state processes stuck on NFS · `-f` adds force (risk of data loss)

CHAPTER 7 OF 8

Chapter 7 — System-Wide Tuning & Kernel Parameters

The Linux kernel exposes hundreds of tunable parameters that control how it manages memory, handles network connections, and allocates resources between processes. Most of the time defaults are fine. But when defaults aren't fine — a high-traffic web server, a database under load, or a system constantly throwing "too many open files" — understanding how to find, change, and (critically) revert these parameters safely is essential.

> **What this chapter covers:** sysctl — reading, changing, and persisting kernel parameters. The four parameter namespaces: vm.*, net.*, fs.*, kernel.*. ulimit — per-process resource limits, the soft/hard limit distinction. Persistent limits via /etc/security/limits.d/ and systemd LimitNOFILE. tuned-adm profiles for pre-packaged tuning. sar for viewing historical system data. Scenario 1: diagnosing and fixing "too many open files." Scenario 2: reverting a sysctl change that made things worse.

sysctl — Reading and Writing Kernel Parameters

Kernel tunables are exposed as files under `/proc/sys/`. `sysctl` is a convenient interface for reading and writing them. Changes made at runtime take effect immediately but are **lost on reboot** unless written to a config file. This is actually useful: a bad change won't survive a reboot, giving you a natural safety net.

```
# ── Reading parameters ──────────────────────────────────────
$ sysctl vm.swappiness          # read one parameter
vm.swappiness = 60

$ sysctl -a                     # list ALL parameters (very long output)
$ sysctl -a | grep tcp_mem      # filter for TCP memory settings
$ sysctl -a 2>/dev/null | grep vm\. | sort    # all vm.* params sorted

# Read directly from /proc/sys (same result)
$ cat /proc/sys/vm/swappiness
60

# ── Writing parameters at runtime ──────────────────────────
# These take effect immediately but are LOST on reboot
$ sysctl -w vm.swappiness=10
vm.swappiness = 10

# Or write directly to /proc/sys
$ echo 10 > /proc/sys/vm/swappiness

# Verify the change applied
$ sysctl vm.swappiness
vm.swappiness = 10

# ── Making changes persistent ──────────────────────────────
# /etc/sysctl.conf — the traditional file (works, but gets cluttered)
# /etc/sysctl.d/   — drop-in directory (preferred — keeps changes organised)

$ cat /etc/sysctl.d/99-mytuning.conf
vm.swappiness = 10
net.core.somaxconn = 1024
fs.file-max = 2097152

# Reload from all config files (applies persistent config to running kernel)
$ sysctl --system         # reads all files in /etc/sysctl.d/ and /etc/sysctl.conf
$ sysctl -p /etc/sysctl.d/99-mytuning.conf   # reload just one file
```

> **Naming convention:** The directory path under `/proc/sys/` maps to the sysctl parameter name — dots replace slashes. So `/proc/sys/vm/swappiness` ↔ `vm.swappiness`, and `/proc/sys/net/ipv4/tcp_fin_timeout` ↔ `net.ipv4.tcp_fin_timeout`.

Key Parameters to Know

vm.* — Virtual Memory

`vm.swappiness` (default **60**)
How aggressively to swap. 0–10 for servers with plenty of RAM that must avoid swap latency. 60 for general use. *0 does not disable swap* — it just deprioritises it strongly.

`vm.dirty_ratio` (default **20**)
Max percentage of RAM that can be dirty (unwritten to disk) before processes are throttled to wait for writes. Too low = frequent pauses; too high = large write spikes on flush.

`vm.dirty_background_ratio` (default **10**)
When background flushing *starts*. Set lower than dirty_ratio. The kernel starts writing dirty pages quietly in the background once this is exceeded.

`vm.vfs_cache_pressure` (default **100**)
How aggressively to reclaim inode and dentry cache. 50 = keep cache longer (good for file-heavy workloads); 200 = reclaim more aggressively.

net.* — Network Stack

`net.core.somaxconn` (default **128**)
Max listen queue depth. A web server under high connection rate needs this raised (1024–65535). If your backlog exceeds this, new connections are silently dropped.

`net.ipv4.ip_local_port_range` (default **32768–60999**)
Ephemeral port range for outgoing connections. Widen to 1024–65535 on servers making many outgoing connections (proxies, databases). Fixes port exhaustion.

`net.ipv4.tcp_fin_timeout` (default **60**)
How long FIN_WAIT2 state lasts in seconds. Reducing to 15–30 frees sockets faster under heavy connection churn.

`net.ipv4.tcp_tw_reuse` (default **0**)
Allow TIME_WAIT sockets to be reused for *outgoing* connections. Set to 1 on busy outbound services. Safe — *never* re-enable the removed tcp_tw_recycle.

`net.core.rmem_max / wmem_max`
Max TCP receive / send buffer size. Raise on high-bandwidth hosts (10 GbE+) to allow larger windows: `net.core.rmem_max=16777216`.

fs.* — Filesystem

`fs.file-max` (default ~**800,000+**)
System-wide maximum total open file descriptors across all processes. Rarely the bottleneck — per-process ulimit usually hits first. Check with `cat /proc/sys/fs/file-nr` (used / unused / max).

`fs.inotify.max_user_watches` (default **8192**)
Maximum inotify watches per user. IDEs, build tools, and development servers (webpack, vite) watch thousands of files. If exhausted: "ENOSPC: System limit for number of file watchers reached." Fix: raise to 524288.

`fs.inotify.max_user_instances` (default **128**)
How many inotify instances a single user can have open. Raise to 256–512 on developer workstations.

kernel.* — Kernel Behaviour

`kernel.pid_max` (default **32768**)
Maximum PID number. On a server spawning many short-lived processes, PID space wraps around — raise to 4194304 (the Linux maximum) to avoid PID collision edge cases.

`kernel.panic` (default **0**)
Seconds to wait before auto-rebooting after a kernel panic. 0 = don't reboot (shows the panic for investigation). Servers often set 10 to auto-recover from transient kernel crashes.

`kernel.shmmax` (default varies)
Maximum single shared memory segment in bytes. PostgreSQL requires this to be larger than shared_buffers. Common fix for "FATAL: could not create shared memory segment."

`kernel.nmi_watchdog` (default **1**)
Hardware watchdog that detects CPU lockups. Set 0 only if needed to reduce PMU counter usage for specific profiling tools.

```
# Check what the kernel is actually using right now for open files
$ cat /proc/sys/fs/file-nr
3712    0    9223372036854775807
# Column 1: open file descriptors in use system-wide (3,712)
# Column 2: unused but allocated slots (ignore, always 0 on modern kernels)
# Column 3: system maximum (fs.file-max)

# Check conntrack table (if running iptables/nftables with connection tracking)
$ cat /proc/sys/net/netfilter/nf_conntrack_count   # current tracked connections
$ cat /proc/sys/net/netfilter/nf_conntrack_max     # maximum allowed
# If count is close to max, connections are silently dropped with "nf_conntrack: table full"
# Fix: sysctl -w net.netfilter.nf_conntrack_max=524288
```

ulimit — Per-Process Resource Limits

While `sysctl fs.file-max` is a system-wide ceiling, `ulimit` controls per-process (and per-user-session) limits. They operate in layers:

KERNEL MAX `fs.file-max` — the absolute ceiling for the entire system. No single process can exceed this across all its FDs combined.

HARD LIMIT Set in `/etc/security/limits.conf` or `/etc/security/limits.d/`. Only root can raise the hard limit. Unprivileged users can only *lower* the hard limit or raise their soft limit up to the hard limit.

SOFT LIMIT The actual working limit enforced on the process. A process can raise its own soft limit up to the hard limit without root. This is what applications typically hit when they report "too many open files."

SYSTEMD OVERRIDE For services managed by systemd, `LimitNOFILE=` in the unit file overrides `limits.conf` entirely. This is the correct way to raise limits for modern services — limits.conf is not read by systemd services by default.

```
# ulimit in the current shell session
$ ulimit -a                  # show all limits for current shell
$ ulimit -n                  # show soft open-files limit
1024
$ ulimit -Hn                 # show hard open-files limit
1048576

# Raise the soft limit for the current shell (up to the hard limit, no root needed)
$ ulimit -n 65536

# Other useful limits
$ ulimit -u          # max user processes
$ ulimit -v          # max virtual memory (kB)
$ ulimit -s          # stack size (kB)

# Read a running process's actual limits (ignores current shell)
$ cat /proc/8821/limits
Limit                     Soft Limit   Hard Limit   Units
Max open files            1024         1048576      files
Max processes             128453       128453       processes
Max address space         unlimited    unlimited    bytes
# This process's soft open-files limit is still 1024 — even if you raised
# the shell limit later, this process was already launched with 1024.
```

Persistent limits — /etc/security/limits.d/

```
# /etc/security/limits.d/99-myapp.conf
# Format: domain   type   item   value
# domain: username, @groupname, or * (all users)

# Raise open file limits for the app user
appuser   soft   nofile   65536
appuser   hard   nofile   65536

# Raise for all users in the www-data group
@www-data soft   nofile   32768
@www-data hard   nofile   32768

# Raise the process limit system-wide (for servers with many workers)
*         soft   nproc    65536
*         hard   nproc    65536

# IMPORTANT: limits.conf only takes effect on NEW login sessions.
# A running process does not pick up changes until it restarts.
# To verify a new login sees the new limits:
$ su - appuser -c "ulimit -n"
65536
```

systemd service limits

```
# For services managed by systemd, limits.conf is NOT read.
# Set limits in the service unit file (or a drop-in override).

# Option 1: Add to /etc/systemd/system/myapp.service
[Service]
LimitNOFILE=65536
LimitNPROC=65536

# Option 2: Create a drop-in override (better — doesn't touch the original unit)
$ systemctl edit myapp     # opens editor for a drop-in override

# Contents of the drop-in (systemctl edit creates this automatically):
[Service]
LimitNOFILE=65536

# Reload systemd and restart the service for changes to take effect
$ systemctl daemon-reload
$ systemctl restart myapp

# Verify the service has the new limit
$ systemctl show myapp | grep LimitNOFILE
LimitNOFILE=65536
$ cat /proc/$(systemctl show -p MainPID --value myapp)/limits | grep "open files"
Max open files            65536        65536        files
```

tuned-adm — Pre-Packaged Tuning Profiles

`tuned` is a daemon that applies a set of kernel parameters, CPU governor settings, and disk scheduler settings as a named profile. Rather than researching and setting dozens of individual sysctl values, you pick a profile that matches your workload type. It is installed by default on RHEL/CentOS/Fedora and available on Debian/Ubuntu.

`throughput-performance`

Maximises CPU throughput and I/O bandwidth. Sets CPU governor to **performance**, disables power saving, tunes network buffers for bulk data. **Best for: batch jobs, database servers, data processing.**

`latency-performance`

Minimises response latency. Disables all CPU power saving, disables transparent huge pages, tunes for low-latency I/O. **Best for: trading systems, real-time applications, gaming servers.**

`balanced`

Moderate tuning across power efficiency and performance. The default on most distributions. **Best for: general-purpose servers, development machines.**

`powersave`

Minimises energy consumption. CPU governor set to **powersave**, reduces disk write frequency. **Best for: idle servers, edge devices, cost-sensitive deployments.**

`virtual-guest`

Tuned for running inside a hypervisor. Disables unnecessary host-level tuning, reduces overhead for virtualised I/O. **Best for: any VM or VPS.**

`network-latency`

Focuses on network response time — tunes TCP parameters, disables offload features that add latency, enables RPS/RFS for multi-core NIC handling. **Best for: high-traffic web/API servers.**

```
# Install tuned if not present
$ apt install tuned          # Debian/Ubuntu
$ dnf install tuned          # RHEL/Fedora
$ systemctl enable --now tuned

# Working with profiles
$ tuned-adm active           # show current profile
Current active profile: balanced
$ tuned-adm list             # all available profiles
$ tuned-adm recommend        # recommend a profile for this hardware
virtual-guest

# Apply a profile (takes effect immediately, survives reboot)
$ tuned-adm profile throughput-performance
Switching to profile 'throughput-performance'

# Verify what parameters it actually changed
$ tuned-adm profile_info throughput-performance
$ cat /etc/tuned/throughput-performance/tuned.conf   # if you have the profile installed

# Revert to default (or any other profile)
$ tuned-adm profile balanced
```

sar — Historical System Activity Data

`sar` (System Activity Reporter) is part of the `sysstat` package. When sysstat is installed and its collection service is running, it records CPU, memory, disk, and network statistics every 10 minutes to files in `/var/log/sysstat/`. This lets you look back at what was happening hours or days ago — invaluable when someone reports "the server was slow this morning" and you weren't there.

```
# Install (if not present)
$ apt install sysstat        # Debian/Ubuntu
$ dnf install sysstat        # RHEL/Fedora
$ systemctl enable --now sysstat

# ── Live sampling ──────────────────────────────────────────
$ sar -u 1 5        # CPU utilisation — sample every 1s, 5 times
$ sar -r 1 5        # memory utilisation — 1s interval, 5 samples
$ sar -b 1 5        # I/O statistics — transfers, reads, writes per second
$ sar -n DEV 1 5    # network — packets and bytes per interface per second

# ── Historical data from today ─────────────────────────────
$ sar -u                    # CPU history from midnight to now (default)
$ sar -r                    # memory history from midnight to now
$ sar -u -s 08:00 -e 12:00  # CPU between 08:00 and 12:00 today
Linux 6.1.0  (myserver)  06/14/2026  _x86_64_  (8 CPU)

08:00:01 AM  CPU  %user  %nice  %system  %iowait  %steal  %idle
08:10:01 AM  all   12.3    0.0      4.1     28.4     0.0   55.2
08:20:01 AM  all   14.1    0.0      5.2     31.7     0.0   49.0
# High %iowait between 08:00-08:20 — something was hammering disk at that time.
# Correlate with sar -b and sar -d to identify which disk.

# ── Historical data from a previous day ────────────────────
$ sar -u -f /var/log/sysstat/sa13    # CPU data from the 13th of this month
$ ls /var/log/sysstat/               # see what data files exist
sa12  sa13  sa14

# Memory — key columns in sar -r output:
#   %memused  — percentage of RAM in use (including buffers/cache)
#   kbbuffers — memory used as kernel buffers
#   kbcached  — memory used as page cache (reclaimable)
#   %commit   — percentage of RAM+swap committed to processes
$ sar -r -s 02:00 -e 06:00   # investigate a memory spike at 3am
```

> **sysstat not collecting?** Check `grep ENABLED /etc/default/sysstat` — on Debian/Ubuntu it ships with `ENABLED="false"` and you must change it to `"true"` and restart the service. Without this, the collection timers are installed but do nothing.

Scenario 1 — "Too Many Open Files" Errors

SCENARIO · CHAPTER 7 · SCENARIO 1

An application log shows "OSError: [Errno 24] Too many open files" or "accept: too many open files." The service is degraded — it's rejecting connections. Diagnose where the limit is being hit and fix it permanently.

1 **Find the process and check its actual open file count vs its limit.**

```
# Find the PID of the service
$ pgrep -a myapp
8821 /usr/bin/myapp --config /etc/myapp/config.toml

# How many file descriptors is it actually using right now?
$ ls /proc/8821/fd | wc -l
1021

# What is its soft limit?
$ cat /proc/8821/limits | grep "open files"
Max open files            1024         1048576      files
# The process has 1,021 FDs open against a soft limit of 1,024.
# It's about to hit the wall — or already hitting it.
# The hard limit is 1,048,576 so there is plenty of room to raise the soft limit.
```

2 **Understand what those file descriptors are — are there any leaks?**

```
# Break down open FDs by type
$ lsof -p 8821 | awk '{print $5}' | sort | uniq -c | sort -rn
    847 IPv4            ← 847 open TCP connections
     98 REG             ← 98 regular files
     44 PIPE            ← 44 pipes
     32 sock            ← 32 Unix domain sockets

# 847 TCP connections from one process is a lot.
# Check their states — are they all active, or is there a CLOSE_WAIT buildup?
$ lsof -p 8821 | grep IPv4 | awk '{print $NF}' | sort | uniq -c
    763 (ESTABLISHED)
     84 (CLOSE_WAIT)
# 84 CLOSE_WAIT connections indicate the app is not closing connections properly.
# That's a code-level bug. Fix the bug long-term, but raise the limit short-term.
```

3 **Immediate relief — if this is a systemd service, add LimitNOFILE to the unit.**

```
# Create a drop-in override
$ systemctl edit myapp
# Add the following in the editor that opens:
[Service]
LimitNOFILE=65536

# Apply and restart
$ systemctl daemon-reload
$ systemctl restart myapp

# Verify the new limit is in effect
$ cat /proc/$(systemctl show -p MainPID --value myapp)/limits | grep "open files"
Max open files            65536        65536        files
```

4 **If this is a non-systemd process or a user-launched application — use limits.d/**

```
# Add a drop-in limits file (takes effect on next login for that user)
$ cat > /etc/security/limits.d/99-myapp.conf << 'EOF'
appuser   soft   nofile   65536
appuser   hard   nofile   65536
EOF

# Test as that user (in a new login session)
$ su - appuser -c "ulimit -n"
65536
```

5 **Also check the system-wide ceiling (rarely the bottleneck, but worth verifying).**

```
$ cat /proc/sys/fs/file-nr
9841    0    9223372036854775807
# 9,841 open FDs system-wide out of a virtually unlimited ceiling. Not the issue.

# If you ever do hit the system-wide limit (very unusual on modern systems):
$ sysctl -w fs.file-max=2097152
$ echo "fs.file-max = 2097152" > /etc/sysctl.d/99-fs-limits.conf
```

The most common "too many open files" cause is not a limit that is too low — it's a leak. Connection pools not being returned, file handles not being closed, or CLOSE_WAIT accumulation. Raising the limit is the right first step to restore service, but investigate the root cause so you're not raising limits indefinitely.

Scenario 2 — A sysctl Change Made Things Worse

SCENARIO · CHAPTER 7 · SCENARIO 2

You set vm.dirty_ratio=3 to reduce write latency on a database server. Now applications are pausing more frequently — the tight dirty limit is causing constant write throttling. Revert the change without rebooting.

1 **Confirm the current (bad) value and find the default to revert to.**

```
$ sysctl vm.dirty_ratio
vm.dirty_ratio = 3

# What is the kernel default? The safest source is the kernel documentation.
# For common parameters, the defaults are well known:
#   vm.dirty_ratio          → 20  (default on most systems)
#   vm.dirty_background_ratio → 10 (default on most systems)
#   vm.swappiness           → 60
#   net.core.somaxconn      → 4096 (was 128 before kernel 5.4)
#
# You can also check an unmodified system of the same kernel version,
# or look at /usr/lib/sysctl.d/50-default.conf or equivalent.
```

2 **Revert the runtime value immediately — no reboot needed.**

```
# Set the runtime value back to the default
$ sysctl -w vm.dirty_ratio=20
vm.dirty_ratio = 20

# Verify it applied
$ sysctl vm.dirty_ratio
vm.dirty_ratio = 20
# The running kernel is now using the reverted value.
# Applications will see improved behaviour immediately — no restart needed.
```

3 **Remove or correct the persistent config file so it doesn't come back after reboot.**

```
# Find which config file set this
$ grep -r "dirty_ratio" /etc/sysctl.conf /etc/sysctl.d/ 2>/dev/null
/etc/sysctl.d/99-custom.conf:vm.dirty_ratio=3

# Remove or correct the bad line
$ sed -i '/dirty_ratio=3/d' /etc/sysctl.d/99-custom.conf

# Or, if reverting the entire file, remove it altogether
$ rm /etc/sysctl.d/99-custom.conf

# Verify nothing in the config files would re-apply the bad value on reboot
$ grep -r "dirty_ratio" /etc/sysctl.conf /etc/sysctl.d/ 2>/dev/null
# (no output) — the bad value is gone from all config files
```

4 **Test the revert by doing a dry-run of what would happen on reboot.**

```
# Reload all sysctl config files (simulates what happens at boot)
$ sysctl --system
* Applying /usr/lib/sysctl.d/50-default.conf ...
* Applying /etc/sysctl.d/10-network.conf ...
* Applying /etc/sysctl.d/99-custom.conf ...   ← no longer sets dirty_ratio
* Applying /etc/sysctl.conf ...

# Verify the final state is what you want
$ sysctl vm.dirty_ratio vm.dirty_background_ratio
vm.dirty_ratio = 20
vm.dirty_background_ratio = 10
# Both back to defaults. A reboot would produce the same result.
```

5 **If you want to tune this correctly — understand what the parameters actually control before changing them.**

```
# Monitor dirty memory in real time to understand your workload before tuning
$ watch -n 1 'grep -E "Dirty|Writeback" /proc/meminfo'
Dirty:              45312 kB    ← currently dirty, not yet written
Writeback:            256 kB    ← currently being written to disk
# If Dirty rarely exceeds a few hundred MB, vm.dirty_ratio=20 is already fine.
# Only tune dirty_ratio downward if Dirty regularly hits tens of GB and causes
# unacceptable pause latency when the kernel forces a flush.

# A safer approach than lowering dirty_ratio: use ionice to deprioritise
# I/O-heavy background processes rather than tightening the global dirty limit.
```

The key safety property of sysctl: changes made with `sysctl -w` survive until the next reboot, but no longer. If you make a bad change and can't figure out the correct value, rebooting restores whatever your config files specify. Make changes to the runtime kernel first; only write to config files once you've confirmed the change is an improvement.

Quick Reference — Chapter 7 Commands

COMMAND	PURPOSE	NOTES
`sysctl -a | grep vm\.`	List all virtual memory kernel parameters	Use `sysctl parameter.name` to read one value. Dot notation maps to `/proc/sys/` path.
`sysctl -w vm.swappiness=10`	Change a kernel parameter at runtime (lost on reboot)	Verify with `sysctl vm.swappiness`. Safe — rebooting undoes it unless in a config file.
`sysctl --system`	Reload all config files from /etc/sysctl.d/ (simulates boot)	`sysctl -p /etc/sysctl.d/file.conf` to reload just one file
`cat /proc/PID/limits`	Show a running process's actual resource limits	More reliable than `ulimit -a` which shows the current shell, not the target process
`ulimit -n`	Show soft open-files limit for current shell session	`ulimit -Hn` for hard limit · `ulimit -a` for all limits · `ulimit -n 65536` to raise
`cat /proc/sys/fs/file-nr`	System-wide: used / unused / max open file descriptors	Rarely the bottleneck — per-process ulimit hits first
`systemctl edit service`	Create a drop-in override for a systemd unit (add LimitNOFILE=)	Follow with `systemctl daemon-reload && systemctl restart service`
`tuned-adm recommend`	Ask tuned what profile best suits this hardware	`tuned-adm profile throughput-performance` to apply · `tuned-adm active` to check current
`sar -u 1 5`	CPU utilisation — sample every 1 second, 5 times	`sar -r` memory · `sar -b` I/O · `sar -n DEV` network · `sar -f /var/log/sysstat/sa13` historical
`grep -r "param" /etc/sysctl.d/`	Find which config file sets a particular kernel parameter	Run before modifying to know where to clean up. Also check `/etc/sysctl.conf`.

CHAPTER 8 OF 8

Chapter 8 — Further Reading & Resources

The tools covered in this course — top, iostat, ss, vmstat, strace, sysctl — are the everyday instruments every Linux administrator should reach for first. But performance work goes deeper. This chapter introduces the next layer: kernel-level profiling with `perf`, modern tracing with eBPF and bpftrace, the most valuable books and resources in the field, and a complete one-liner cheat sheet covering every topic from all eight chapters.

> **What this chapter covers:** perf stat, perf top, perf record, and flame graphs for CPU profiling. eBPF architecture and why it matters. bpftrace one-liners. BCC ready-made tools (execsnoop, opensnoop, biolatency, tcpconnect, and more). Brendan Gregg's resources. Essential books. Key man pages. Full diagnostic one-liner cheat sheet organised by topic.

perf — Kernel-Level CPU Profiling

`perf` is the Linux kernel's built-in profiling tool. Unlike top or htop — which show what processes are running — `perf` shows what those processes are *doing at the CPU instruction level*. It can tell you which functions inside a program are consuming CPU, whether the bottleneck is in your code or in a library, and even whether cache misses or branch mispredictions are the root cause of high CPU usage.

> **Installation:** `apt install linux-perf linux-tools-common` (Debian/Ubuntu) or `dnf install perf` (RHEL/Fedora). You may also need `linux-tools-$(uname -r)`. Run as root or with `CAP_SYS_ADMIN` capability.

`perf stat`

Run a command and count hardware events

Runs a program and prints hardware event counts when it finishes: CPU cycles, instructions, cache references, cache misses, branch instructions, branch misses. **Useful for comparing two implementations** of the same function. `perf stat -e cache-misses,instructions ./myprogram`

`perf top`

Live CPU profiling — like htop but shows functions

Shows a continuously updated list of the kernel functions and user-space functions consuming the most CPU cycles system-wide. **The fastest way to answer "what is the CPU actually executing?"** No recording needed. Press `a` to annotate a function with its assembly.

`perf record`

Capture a CPU profile to a file

Records samples to `perf.data`. Attach to a running process with `-p PID`, or profile a command directly. `-g` enables call graph recording (stack traces) — required for flamegraphs. `perf record -g -p 8821 sleep 30` profiles PID 8821 for 30 seconds.

`perf report`

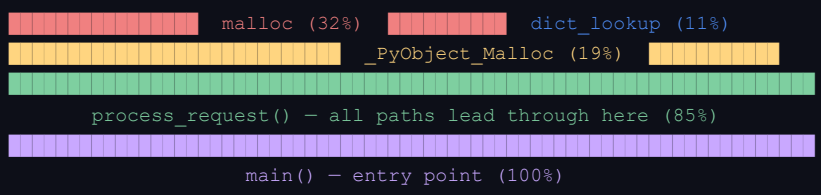
Browse a recorded profile interactively

Opens `perf.data` in an interactive TUI. Shows samples sorted by overhead %. Expand a function to see its callers and callees. Press `a` to annotate with source code (if debug symbols are available). `perf report --stdio` for plain text output.

```
# Quick hardware event summary for the currently running system
$ perf stat -a sleep 5
 Performance counter stats for 'system wide':
    39,842,156,430  cycles            #   7.97 GHz
    28,391,004,221  instructions      #   0.71  insn per cycle
     1,203,441,882  cache-references
       312,441,112  cache-misses      #  25.96% of all cache refs
# 26% cache miss rate is high — suggests a memory-intensive workload with poor locality

# Profile a running process for 30 seconds with call graphs
$ perf record -g -p 8821 sleep 30
[ perf record: Woken up 12 times to write data ]
[ perf record: Captured and wrote 3.241 MB perf.data (84241 samples) ]

$ perf report --stdio | head -30
# Overhead  Command   Shared Object      Symbol
    32.41%  python3   libc-2.31.so       [.] malloc
    18.72%  python3   python3.9          [.] _PyObject_Malloc
    11.44%  python3   python3.9          [.] dict_lookup
# malloc and _PyObject_Malloc together = 51% of CPU time in memory allocation.
# This process is spending more time allocating memory than doing actual work.
```

Flame Graphs — visualising the whole call stack at once

A flame graph collapses thousands of perf samples into a single SVG. Each box is a function. Width represents time spent. The call stack grows upward — a wide box at the top means that function is directly consuming CPU. Narrow boxes below it are its callers.




```
Width = time. Height = call depth. Wide flat top = hot function. Narrow spike = deep call chain.
Read bottom-up: where your code spends time bubbles to the top.
```

```
# Generate a flame graph (requires Brendan Gregg's FlameGraph scripts)
$ git clone https://github.com/brendangregg/FlameGraph
$ perf record -F 99 -g -p 8821 sleep 30
$ perf script | ./FlameGraph/stackcollapse-perf.pl | ./FlameGraph/flamegraph.pl > flame.svg
$ xdg-open flame.svg   # or copy to a web server and view in browser

# Off-CPU flame graph — shows time spent WAITING (not on CPU)
# Useful for I/O latency, lock contention, sleep() calls
$ perf record -e sched:sched_stat_sleep -e sched:sched_switch -g -p 8821 sleep 30
$ perf inject --stats -i perf.data | ./FlameGraph/stackcollapse-perf.pl --offcpu | ./FlameGraph/fl
```

eBPF and bpftrace — Modern Kernel Tracing

eBPF (extended Berkeley Packet Filter) allows small, safe programs to run inside the Linux kernel in response to events — system calls, function calls, network packets, scheduler events — without modifying kernel source code or loading a kernel module. The kernel verifies the program before running it, guaranteeing it can't crash the system. This makes eBPF the foundation of modern Linux observability.

bpftrace is a high-level scripting language for eBPF — it compiles one-liners and short scripts into eBPF programs and attaches them to kernel and user-space events. Think of it as `awk` for the kernel.

```
# Install bpftrace
$ apt install bpftrace      # Debian/Ubuntu 20.04+
$ dnf install bpftrace      # RHEL 8+

# List available probepoints (tracepoints + kprobes + uprobes)
$ bpftrace -l | head -30
tracepoint:syscalls:sys_enter_read
tracepoint:syscalls:sys_enter_write
kprobe:tcp_sendmsg

# ── One-liners ──────────────────────────────────────────────

# Count system calls by process name (for 10 seconds)
$ bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); } interval:s:10 { print(@);

# Trace every file opened on the system (filename + PID)
$ bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s opened %s\n", comm, str(args->fil

# Distribution of read() sizes — shows if reads are small (many syscalls) or large
$ bpftrace -e 'tracepoint:syscalls:sys_exit_read /args->ret > 0/ { @bytes = hist(args->ret); }'

# Disk I/O latency histogram (microseconds)
$ bpftrace -e 'tracepoint:block:block_rq_issue { @start[args->dev, args->sector] = nsecs; }
  tracepoint:block:block_rq_complete /@start[args->dev, args->sector]/
  { @us = hist((nsecs - @start[args->dev, args->sector]) / 1000);
    delete(@start[args->dev, args->sector]); }'

# TCP connection attempts by destination port
$ bpftrace -e 'kprobe:tcp_connect { @[((struct sock *)arg0)->__sk_common.skc_dport] = count(); }'
```

BCC — Ready-Made eBPF Tools

BCC (BPF Compiler Collection) ships with dozens of production-ready tracing tools. Install with `apt install bpfcc-tools` or `dnf install bcc-tools`. On Ubuntu the binaries are suffixed `-bpfcc`.

`execsnoop`

Every new process spawned on the system in real time — PID, parent, full command line. Invaluable for spotting rogue cron jobs or compiler spam.

`opensnoop`

Every file open() call — which process opened which file. Filter by PID with `-p PID`. Great for finding what config files an application actually reads.

`biolatency`

Block I/O latency histogram. Shows the distribution of disk request latencies in microseconds — immediately reveals if a disk is spiking to high latencies.

`biotop`

Like iotop but uses eBPF — top processes by disk I/O, updated live. Lower overhead than iotop. Shows reads and writes separately.

`tcpconnect`

Every outgoing TCP connection — destination IP and port. Shows you what a process is connecting to without a network capture.

`tcpaccept`

Every incoming TCP connection accepted — source IP, destination port, PID. Useful for auditing what services are accepting connections.

`tcpretrans`

TCP retransmissions in real time — source, destination, state. Retransmissions are a smoking gun for network packet loss.

`cachetop`

Page cache hit rate by process — how much of each process's I/O is served from the page cache vs going to disk.

`runqlat`

CPU run-queue latency histogram — how long processes wait on the run queue before getting CPU time. High latency here means CPU saturation.

`profile`

CPU profiler (like perf top) using eBPF — samples the call stack at 99Hz and reports the top functions. Lower overhead than perf for long-running profiles.

`memleak`

Tracks outstanding memory allocations — surfaces objects that were malloc'd but never freed. Run against a suspect process to confirm a memory leak.

`funclatency`

Latency histogram for any kernel or user-space function. `funclatency vfs_read` shows how long filesystem reads take.

> **eBPF kernel version requirement:** bpftrace needs kernel 4.9+; most BCC tools work best on 4.18+; advanced features (BTF, CO-RE) require 5.8+. On older kernels, fall back to the traditional tools covered in chapters 1–7. Check with `uname -r`.

Brendan Gregg's Resources

Brendan Gregg (Senior Principal Engineer at Intel, formerly Netflix) is the most influential figure in Linux performance analysis. He created the USE method, the flamegraph visualisation, and the Linux Performance Observability Tools diagram — the field's most-referenced one-page reference.

- **Linux Performance Observability Tools diagram** — a single diagram showing which tool covers which layer of the Linux stack (hardware → kernel → user space), organised by CPU/memory/file/network/system call. Available at brendangregg.com/linuxperf.html. Print this and keep it on your desk.
- **The USE Method** — Utilisation / Saturation / Errors. For every resource, check if it's over-utilised, if there's a queue forming (saturation), and if there are error counts. Covered in Chapter 1 of this course.
- **FlameGraph repository** — github.com/brendangregg/FlameGraph — the Perl scripts used to generate flamegraph SVGs from perf, DTrace, or bpftrace data.
- **brendangregg.com** — extensive blog posts on perf, eBPF, and performance analysis methodology. The posts are long, detailed, and worth reading in full.

Essential Books

ESSENTIAL

Systems Performance: Enterprise and the Cloud

Brendan Gregg — 2nd edition, 2020

The definitive reference on Linux and Solaris performance analysis. Covers every layer: CPU, memory, file systems, disks, network, and the cloud. The second half is an exhaustive reference for every tool and metric. **If you buy one book on this topic, this is the one.** Dense but comprehensive — read the chapter introductions, then use it as a reference for specific topics.

ESSENTIAL

BPF Performance Tools

Brendan Gregg — 2019

Companion to Systems Performance, focused entirely on eBPF, BCC, and bpftrace. Documents every BCC tool with examples and explains what each one is measuring at the kernel level. **The definitive eBPF observability reference.** Available free to read at the author's site.

DEEP DIVE

Linux Kernel Development

Robert Love — 3rd edition, 2010

Explains how the Linux kernel works internally: the scheduler, virtual memory, the VFS layer, device drivers, synchronisation. Older (pre-cgroups/eBPF era) but the fundamentals haven't changed. **Helps you understand why sysctl parameters have the effect they do.**

REFERENCE

The Linux Programming Interface

Michael Kerrisk — 2010

Comprehensive reference for Linux system calls and the C library — signals, processes, threads, files, sockets, memory mapping. Directly relevant when using strace output: knowing what each syscall does helps you interpret what a stuck process is waiting on. **The man pages, explained.**

Essential Man Pages

MAN PAGE	WHAT IT COVERS	WHY IT MATTERS FOR PERFORMANCE WORK
`man 5 proc`	The /proc filesystem — every file under /proc/ documented	Explains what each field in /proc/meminfo, /proc/PID/status, /proc/stat, /proc/net/dev actually means. The authoritative source when a metric is ambiguous.
`man sysctl.conf`	Format and semantics of /etc/sysctl.conf and /etc/sysctl.d/ files	File format, ordering rules, how wildcards work, and which files override which.
`man 5 limits.conf`	PAM limits configuration — the format for /etc/security/limits.conf	All supported resource types (nofile, nproc, memlock, etc.) with valid value ranges.
`man iostat`	All iostat flags and column definitions	Precise definition of %util, await, aqu-sz, and the difference between -x and basic output.
`man ss`	Socket statistics — all filter expressions and output fields	The filter syntax for selecting sockets by state, address, or port. More powerful than the man page initially suggests.
`man tc`	Traffic control — queueing disciplines, classes, filters	Very large man page covering the full tc syntax for bandwidth shaping and throttling.
`man perf`	Linux perf tool — events, record options, report format	List of available hardware events (`perf list`) and all record/report flags.
`man bpftrace`	bpftrace language reference — probes, builtins, functions	Full language spec: all probe types (kprobe/uprobe/tracepoint/usdt), builtin variables (comm, pid, tid, nsecs), and map functions (count/hist/avg).
`man 7 signal`	Signal overview — all signal numbers, default actions, and semantics	Complete table of every signal, whether it can be caught/blocked, and what it does by default. Useful when you don't remember a signal number.
`man free`	free command output field definitions	Clarifies the "available" column vs "free" — a common source of confusion when reading memory output.

Diagnostic One-Liner Cheat Sheet

The following one-liners cover the most common diagnostic tasks from all eight chapters. Keep this as a reference for the next time something is slow and you need to start somewhere.

CPU — CHAPTERS 1 & 2	
`uptime`	Load average (1/5/15 min). Divide by nproc — above 1.0 = saturation
`top -b -n 1 | head -20`	One-shot CPU snapshot sorted by %CPU (no interactive mode)
`mpstat -P ALL 1 3`	Per-core CPU breakdown — spot single-core saturation hidden by averages
`ps aux --sort=-%cpu | head -10`	Top 10 processes by CPU consumption right now
`pgrep -c -f "pattern"`	Count processes matching a full command-line pattern
`ps faux | grep -A10 "script.sh"`	Show process tree rooted at a specific process
`renice +10 -p PID`	Throttle a CPU-hungry process without killing it (nice +10 = lower priority)
`taskset -cp 0,1 PID`	Pin a process to CPU cores 0 and 1 to isolate it from other workloads
`ps -o pid,stat,comm -p PID`	Check if a process is D (uninterruptible), Z (zombie), or S (sleeping)

MEMORY — CHAPTER 3	
`free -h`	Memory overview — read "available" not "free" for usable RAM
`vmstat 1 5`	Memory + swap + I/O every second (5 samples) — si/so = swap in/out
`grep -E "Mem|Swap|Dirty|Commit" /proc/meminfo`	Key meminfo fields — committed memory, dirty pages, swap usage
`ps aux --sort=-%mem | head -10`	Top 10 processes by RSS (actual RAM usage)
`smem -r | head -10`	Top 10 by PSS (proportional memory — accounts for shared libraries correctly)
`dmesg -T | grep -i "oom|killed"`	Check if the OOM killer has fired recently
`watch -n 2 'grep -E "Dirty|Writeback" /proc/meminfo'`	Monitor dirty page accumulation in real time
`cat /proc/PID/oom_score`	OOM kill priority for a specific process (0=immune, 1000=kill first)
`sysctl vm.swappiness`	Check current swap aggressiveness setting (default 60)

DISK I/O & STORAGE — CHAPTER 4	
`df -h`	Disk space by filesystem — spot filesystems near 100%
`df -i`	Inode usage — "disk full" errors with free space = inode exhaustion
`du -sh /* 2>/dev/null | sort -rh | head -15`	Find the largest top-level directories consuming space
`iostat -x 1 5`	Disk I/O — watch %util, await, aqu-sz for saturation
`iotop -o -b -n 3`	Which processes are doing I/O right now (requires root)
`lsof | grep deleted | awk '$NF ~ /\(deleted\)/'`	Files deleted but still held open (df vs du discrepancy)
`find / -xdev -printf '%h\n' | sort | uniq -c | sort -rn | head -5`	Find directories with most files (inode exhaustion investigation)
`journalctl --disk-usage`	How much space systemd journal is consuming
`truncate -s 0 /var/log/bigfile.log`	Empty a log file without breaking open file handles

NETWORK — CHAPTER 5	
`ss -tulpn`	Listening TCP/UDP ports with process names (no DNS lookup)
`ss -s`	Socket summary — total counts by state (TIME_WAIT, ESTABLISHED, etc.)
`ss -tan | awk '{print $1}' | sort | uniq -c | sort -rn`	Count TCP connections by state
`ping -c 5 host && mtr --report host`	Latency baseline then per-hop breakdown — spot where packet loss occurs
`dig host +stats | grep "Query time"`	DNS resolution time — is DNS adding latency?
`curl -w "@curl-format.txt" -o /dev/null -s URL`	HTTP timing breakdown: DNS / TCP connect / TLS / TTFB / total
`nethogs eth0`	Bandwidth by process — which process is saturating the NIC
`cat /proc/net/dev | column -t`	NIC error counters — RX errors, dropped, overrun, TX errors
`ethtool eth0 | grep -E "Speed|Duplex|Link"`	NIC link speed and duplex — spot half-duplex mismatches

PROCESS MANAGEMENT — CHAPTER 6	
`ps faux`	Full process list with parent→child tree (forest view)
`pstree -p PID`	Process tree rooted at a specific PID with all PIDs shown
`ps -o pid,ppid,pgid,stat,comm -p PID`	Process details including parent PID and process group
`kill -0 PID`	Check if a PID exists without sending a real signal
`pkill -TERM -f "full-command-pattern"`	SIGTERM by full command line — safer than name-only match
`kill -- -PGID`	SIGTERM to entire process group (parent + all descendants)
`cat /proc/PID/wchan`	What kernel function a D-state process is stuck in
`strace -p PID -T 2>&1 | head -20`	What system calls a running process is making (with timing)
`lsof -p PID | wc -l`	Count open file descriptors for a process
`systemctl edit service`	Create a drop-in systemd override (e.g. add LimitNOFILE=65536)

KERNEL TUNING — CHAPTER 7	
`sysctl -a | grep "param"`	Find kernel parameters matching a pattern
`sysctl -w vm.swappiness=10`	Change a kernel parameter at runtime (lost on reboot)
`sysctl --system`	Reload all sysctl config files (simulate what happens at boot)
`cat /proc/sys/fs/file-nr`	System-wide open FD count: used / unused / max
`cat /proc/PID/limits | grep "open files"`	Process's actual open file limit (soft and hard)
`ulimit -Hn`	Hard open-files limit for the current shell session
`grep -r "nofile" /etc/security/limits.d/ /etc/security/limits.conf`	Find persistent ulimit configuration
`tuned-adm recommend`	Ask tuned which profile suits this hardware
`sar -u -s 08:00 -e 12:00`	Historical CPU usage between 08:00 and 12:00 today
`sar -r -f /var/log/sysstat/sa13`	Historical memory data from the 13th of the month

ADVANCED PROFILING — CHAPTER 8

`perf top`	Live CPU profiler — which kernel/user functions are consuming cycles
`perf stat -a sleep 5`	System-wide hardware event counts for 5 seconds
`perf record -g -p PID sleep 30 && perf report`	Record 30s CPU profile with call graphs and browse results
`bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args->filename)); }'`	Trace every file open on the system in real time
`execsnoop-bpfcc`	Show every new process spawned (BCC tool — requires bpfcc-tools)
`biolatency-bpfcc`	Block I/O latency histogram — reveals disk response time distribution
`tcpconnect-bpfcc`	Every outgoing TCP connection —

	destination IP and port
`runqlat-bpfcc`	CPU run-queue latency — how long processes wait to get CPU time
`memleak-bpfcc -p PID 30`	Track memory allocations for 30 seconds — confirm a memory leak

> **Where to go from here:** The tools in this course are the foundation. The workflow is always the same — form a hypothesis about which resource is the bottleneck, use the relevant tool to confirm it, resolve the root cause, and verify the fix. perf and eBPF take you deeper when the standard tools don't show enough detail. Brendan Gregg's *Systems Performance* is the natural next step when you want to understand not just what the tools show, but why the Linux kernel behaves the way it does.