

Linux

Aayush Bajaj

July 10, 2026

Contents

1	kernel vs userland	1
2	the process model: fork, exec, wait	2
3	the filesystem hierarchy	3
4	permissions	4
5	processes and signals	4
6	pipes and redirection	5
7	package management	5
8	the boot chain	5
9	see also	5

unix is less an operating system than a worldview: everything is a file, every program does one thing, and text streams are the universal interface. linux is the worldview's most successful implementation — a monolithic kernel started by a finnish undergraduate in 1991, now running most of the internet, every android phone, and the top 500 supercomputers without exception. this page is the trunk; the sharpened tools each get their own branch:

macos is the other unix on this desk — same syscall dialect, different kernel lineage. every demo on this page runs on both

- `shell/s` — the REPL for the operating system itself
- `vim` — text editing as a language
- `emacs` — the other church
- `regular expressions` — the pattern language every unix tool speaks
- `gdb` — watching processes from the inside

1 kernel vs userland

one line divides the machine: the CPU's privilege boundary. the **kernel** runs privileged — it owns the hardware, the scheduler, the page tables, the filesystems, the network stack. **userland** is everything else: your shell, your editor, your database, all sandboxed into per-process virtual address spaces and allowed to touch reality only by asking.

the asking mechanism is the **system call**: a controlled trap into the kernel (`read`, `write`, `open`, `fork`, `execve`, `mmap`, ...). a syscall is a function call that changes privilege level, which makes it expensive relative to a normal call — hence buffered I/O in libc, and hence decades of engineering to keep hot paths out of the kernel (Tanenbaum, Andrew S., 2008).

the memory system is the part worth holding in your head (the original stub of this page was three words: *pagetables*, *TLB*, *cache* — fair summary): each process sees a flat virtual address space; **page tables** map virtual pages to physical frames; the **TLB** caches those translations because walking a four-level page table on every access would be ruinous; and the hardware **caches** (L1/L2/L3) sit under all of it hiding DRAM latency. the deep dive lives in *memory*.

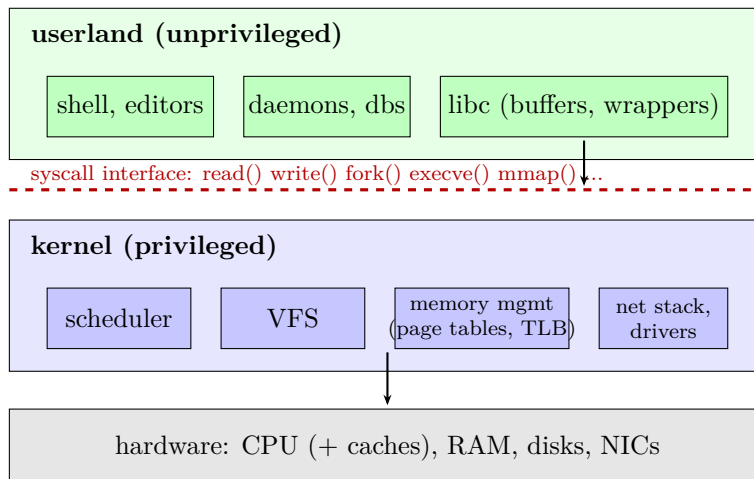


Figure 1: the privilege boundary: userland asks, the kernel decides, the hardware obeys.

2 the process model: fork, exec, wait

unix creates processes by *cloning*: `fork()` duplicates the calling process — same code, same (copy-on-write) memory, same open file descriptors — and returns twice: the child sees `0`, the parent sees the child's pid. the child then usually calls `execve()`, which replaces its entire program image with a new one while keeping the pid and file descriptors. the parent calls `wait()` to collect the child's exit status. every shell command you have ever run is this trio (Tanenbaum, Andrew S., 2008).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(void) {
7      printf("parent: pid=%d\n", getpid());
8      fflush(stdout);                      /* don't let the child inherit the
9      ↪  buffer */
10
11     pid_t pid = fork();                  /* one process in, two out */
12     if (pid == 0) {                      /* child sees 0 */
13         printf("child : pid=%d ppid=%d\n", getpid(), getppid());
14         fflush(stdout);
15         execlp("echo", "echo", "child : exec replaced me with echo", (char
16         ↪  *)NULL);
17         perror("execlp");                /* reached only if exec failed */
18         exit(1);

```

```

17     }
18                                     /* parent sees the child's pid */
19     int status;
20     waitpid(pid, &status, 0);         /* reap the child: no zombie */
21     printf("parent: child %d exited with status %d\n", pid,
22            ↪ WEXITSTATUS(status));
23     return 0;

```

compiled with `cc -Wall -o fork_demo fork_demo.c` and run:

```

parent: pid=74830
child : pid=75103 ppid=74830
child : exec replaced me with echo
parent: child 75103 exited with status 0

```

the `fflush` before `fork()` is a genuine footgun, not pedantry: `fork` duplicates the `stdio` buffer too, and if `stdout` is redirected to a pipe or file (fully buffered), unflushed output prints *twice*. (Kernighan, Brian W. and Ritchie, Dennis M., 1988)

a child that exits before being `wait()`-ed becomes a **zombie**: dead, but its exit status still occupies a process-table slot until the parent collects it. a child whose parent dies first is an **orphan**, adopted and reaped by `init` (pid 1).

the c idioms here — and the discipline of checking every return value — are straight out of kernighan and ritchie

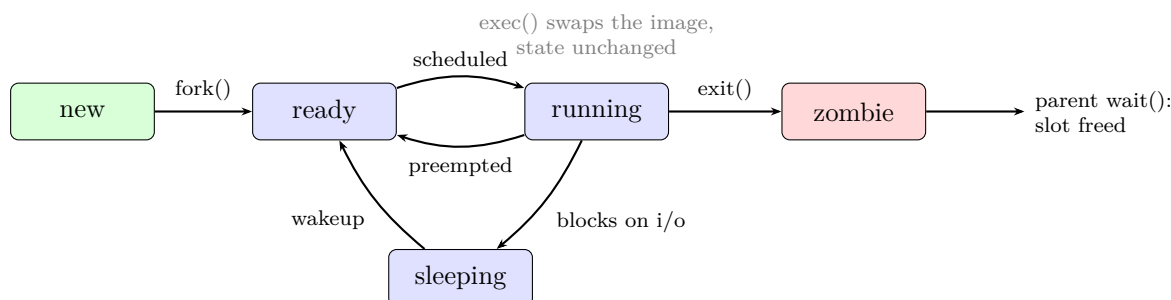


Figure 2: process lifecycle: the scheduler shuttles between ready and running; exit leads through zombie until the parent waits.

3 the filesystem hierarchy

one rooted tree, no drive letters; disks, USB sticks, and even RAM-backed filesystems are *mounted* onto directories. the directories that matter:

dir	contents
<code>/bin, /sbin</code>	essential binaries (nowadays usually symlinks into <code>/usr</code>)
<code>/usr</code>	the bulk of userland: <code>/usr/bin</code> , <code>/usr/lib</code> , <code>/usr/share</code>
<code>/etc</code>	system-wide configuration, plain text almost throughout
<code>/var</code>	things that grow: logs (<code>/var/log</code>), spools, caches
<code>/home</code>	user home directories (<code>/root</code> for root)
<code>/tmp</code>	scratch space, usually wiped at boot
<code>/dev</code>	device nodes: disks (<code>sda</code>), terminals (<code>tty*</code>), <code>null</code> , <code>random</code>
<code>/proc</code>	virtual: the kernel's live view of processes (<code>/proc/1234/</code>)
<code>/sys</code>	virtual: kernel objects, device tree, tunables
<code>/boot</code>	kernel images and the bootloader's files

`/proc` deserves the double-take: `cat /proc/self/status` is a file read that fabricates its contents from kernel state on demand. “everything is a file” is not a slogan,

it is an API decision.

4 permissions

every file carries an owner, a group, and nine permission bits — read/write/execute for user, group, other. each triple is one octal digit ($r = 4$, $w = 2$, $x = 1$):

```
$ touch report.txt && chmod 754 report.txt && ls -l report.txt
-rwxr-xr-- report.txt
```

754 = 111101100₂: owner **rw**x (4 + 2 + 1), group **r**-x (4 + 1), other **r**-- (4). the common defaults decode the same way: **644** for files (owner edits, world reads), **755** for directories and executables — where **x** on a *directory* means “may traverse into it”, a distinction that bites exactly once. new files start from **666** (files) or **777** (directories) masked by your **umask**, conventionally **022**.

the tenth bit worth knowing is **setuid** (**chmod u+s**, shown as **s** in place of the owner’s **x**):

```
$ chmod u+s report.txt && ls -l report.txt
-rwsr-xr-- report.txt
```

a **setuid** *executable* runs with its owner’s identity instead of the caller’s — the mechanism that lets **passwd** (owned by root) edit **/etc/shadow** for you. it is also the classic privilege-escalation surface, which is why modern systems keep the **setuid** inventory as short as possible (Tanenbaum, Andrew S., 2008).

5 processes and signals

ps aux lists processes; **top** (or **htop**) watches them; **kill** is misnamed — it *sends signals*, of which termination is merely the default. the vocabulary:

signal	number	default action	meaning
SIGHUP	1	terminate	terminal hung up; daemons repurpose it as “reload config”
SIGINT	2	terminate	ctrl-c — polite interruption
SIGQUIT	3	core dump	ctrl-\ — interruption with evidence
SIGKILL	9	terminate	uncatchable, unblockable; the kernel just ends you
SIGSEGV	11	core dump	invalid memory access — the debugger’s dinner bell
SIGTERM	15	terminate	catchable “please shut down cleanly” — what kill sends by default
SIGTSTP	—	stop	ctrl-z — suspend, resumable
SIGCONT	—	continue	resume a stopped process

(the stop/continue numbers vary between platforms, so they are spelled by name.) a process can catch or ignore everything except **SIGKILL** and **SIGSTOP** — the kernel’s two non-negotiables. escalation etiquette: **SIGTERM** first, give it a moment, **SIGKILL** only for the unresponsive, because a **SIGKILL**ed process gets no chance to flush buffers or remove lock files.

job control is signals wearing shell syntax: **sleep 100 &** backgrounds a job, **ctrl-z** stops the foreground one, **jobs** lists them, **fg %1** / **bg %1** resume them in fore/background, and **kill %1** addresses them by job number. **nohup** (or **disown**) detaches a job from the terminal so **logout**’s **SIGHUP** misses it.

6 pipes and redirection

the unix philosophy in one operator: `|` connects one process's stdout to another's stdin through a kernel buffer, both processes running *concurrently*. programs become composable because they agree on the dumbest possible interface — a stream of bytes, usually lines of text. word-frequency analysis without writing a program:

```
$ printf 'to be or not to be that is the question\n' \
  | tr ' ' '\n' | sort | uniq -c | sort -rn | head -4
2 to
2 be
1 the
1 that
```

read it left to right: split words onto lines, sort so duplicates are adjacent (`uniq` only sees neighbours — the classic trap), count runs, sort numerically descending, keep four. five single-purpose tools, zero code. the same grammar handles files: `>` redirects stdout (truncating), `>>` appends, `2>` redirects stderr, `2>&1` merges the two streams, `<` feeds stdin. the heavyweight stream tools — `sed` for editing in flight, `awk` for field-wise computation — are a page of their own over at [regular expressions](#) (Dougherty, Dale and Robbins, Arnold, 1997).

7 package management

a package manager is a transactional installer with a dependency solver: it fetches signed archives from repositories, resolves the dependency graph, installs in the right order, and — crucially — knows how to undo. the family tree splits by lineage: debian/ubuntu use `apt` over `.deb`, red hat/fedora use `dnf` over `.rpm`, arch uses `pacman`, and macos fills the gap with `homebrew` or `macports`. the discipline they all buy you is the same: the package database *is* the inventory of your system, so nothing is installed that cannot be listed, verified, or removed.

*this site's build once broke
because ninja was installed
outside any package manager's
memory — unpinned
dependencies are debts, not
tools*

8 the boot chain

power-on hands control to firmware (BIOS historically, UEFI now), which finds and runs a **bootloader** (GRUB, systemd-boot) — from the MBR under legacy BIOS, from the EFI system partition under UEFI; the bootloader loads the kernel image plus an `initramfs` — a small in-RAM root filesystem with just enough drivers to find the real one. the kernel initialises hardware, mounts the real root, and starts exactly one userland process: `init`, pid 1, which on nearly every modern distribution is `systemd`. `systemd` then brings up the rest of the system as a dependency graph of units — services, mounts, sockets, timers — in parallel, which is both why modern linux boots fast and why `systemctl status` is the first debugging command of the decade. everything running traces its ancestry to pid 1 — the process tree is literally a tree.

9 see also

- [internet networks](#) — what the kernel's other half is doing
- [databases](#) — fsync and pages, seen from above
- [memory](#) — pagetables, TLB, cache: the stub's three words, expanded

- **version control** — the other tool that made unix development bearable

references

Dougherty, Dale and Robbins, Arnold (1997). *Sed & Awk*, O'Reilly Media.

Kernighan, Brian W. and Ritchie, Dennis M. (1988). *The C Programming Language*, Prentice Hall.

Tanenbaum, Andrew S. (2008). *Modern Operating Systems*, Pearson.