Chapter 15: Linux

Operating Systems

UNIX and Linux history | Design goals and interfaces | The shell and pipelines

Kernel structure and modules | Processes, signals, threads | Scheduling: CFS and EEVDF

Booting and systemd | Memory: paging, zones, SLUB | I/O: devices, blk-mq, networking

File systems: VFS, ext4 | Security: permissions, containers | Android

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_15_Linux.html

Learning Outcomes

History and family

Trace the line from UNIX through BSD, MINIX and GNU to Linux; place distributions and Android.

Interfaces and the shell

Separate the library / system-call / kernel layers; show how pipelines are built with fork, pipe and dup2.

Processes and scheduling

Explain the kernel steps of fork and execve, signal delivery, task_struct, clone() flags and NPTL; what changed in the move from CFS to EEVDF.

Memory

Explain the address-space layout, 4/5-level page tables, memory zones, the SLUB allocator, the page cache and the OOM killer.

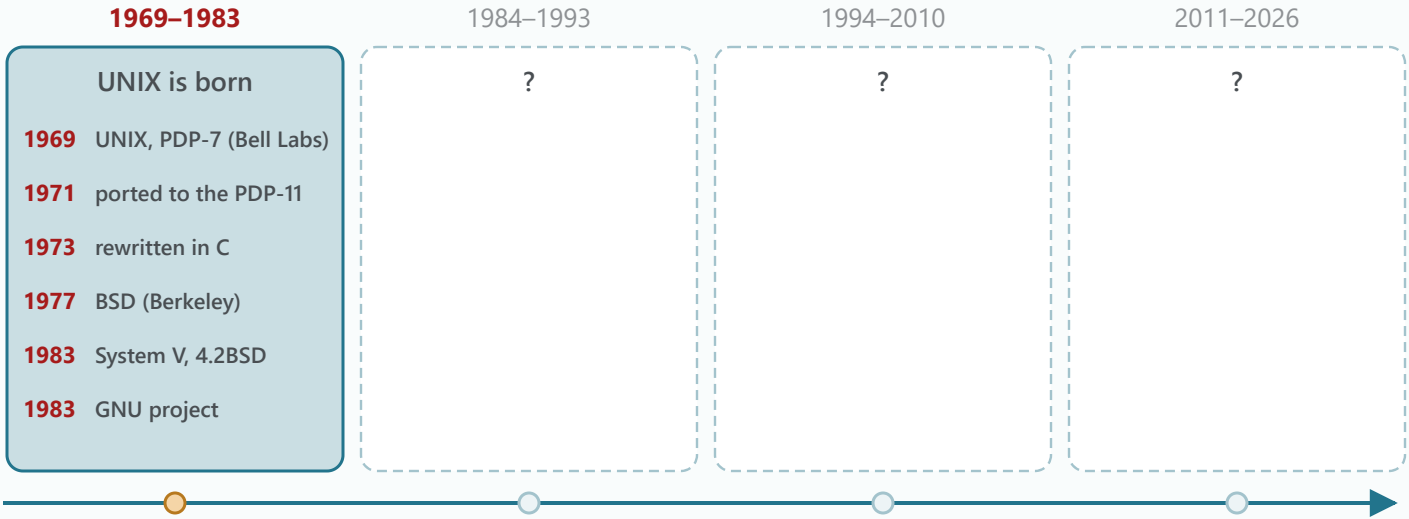
I/O and files

Device files, blk-mq, the network stack; the VFS, i-nodes, ext4 extents and journaling, links and /proc.

Security

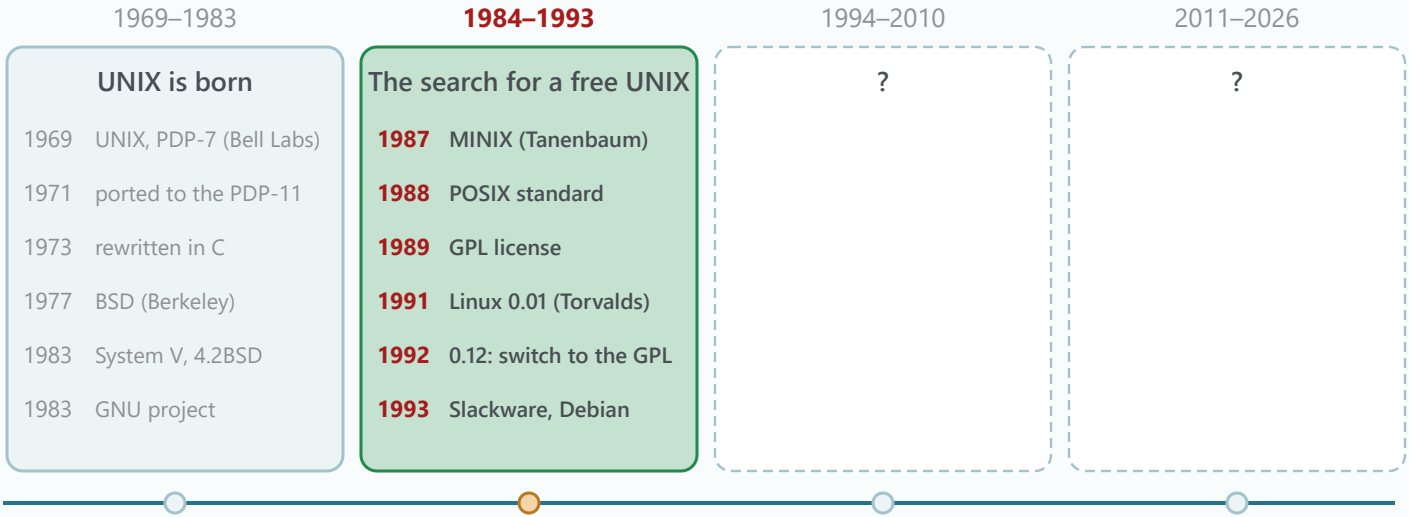
Compute rwx/setuid/umask; recognize capabilities, namespaces + cgroups, SELinux/AppArmor, seccomp and the Android sandbox.

Timeline from UNIX to Linux



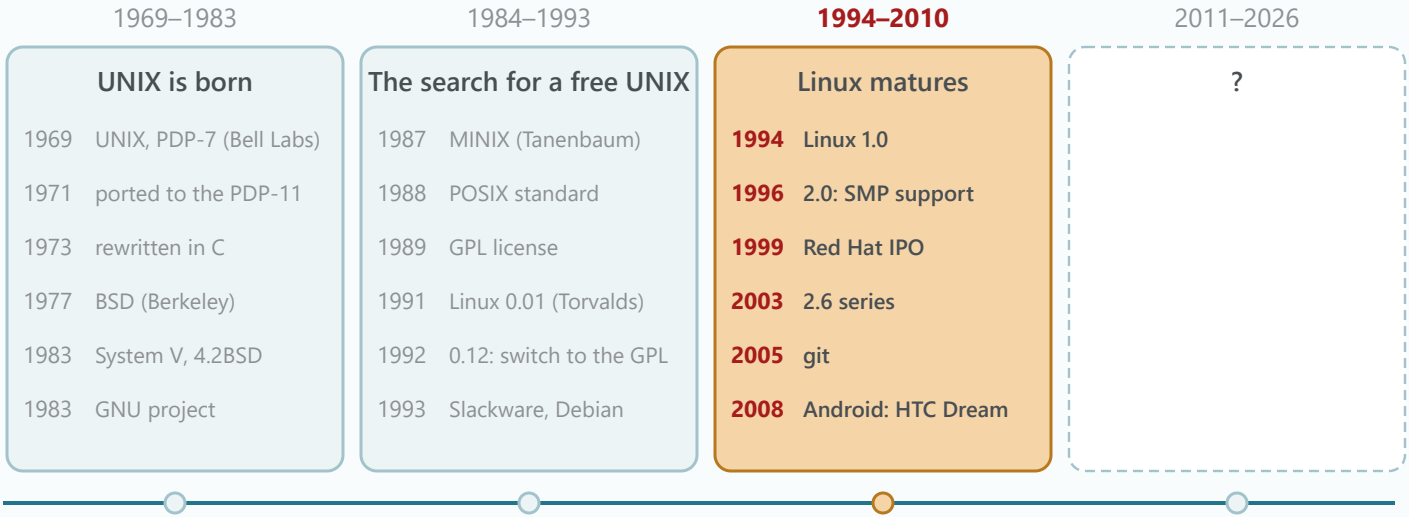
Ken Thompson and Dennis Ritchie wrote UNIX in assembly on a PDP-7 in 1969; **rewriting it in C** in 1973 made it portable. AT&T gave the source to universities, and **BSD** was born at Berkeley.

Timeline from UNIX to Linux



As the commercial UNIXes (System V, SunOS, AIX…) diverged, **POSIX** defined a common interface. Tanenbaum wrote **MINIX** for teaching; in 1991 Linus Torvalds developed his own kernel on MINIX and combined it with the GNU tools.

Timeline from UNIX to Linux



Linux 1.0 (1994); multiprocessing with 2.0; a preemptible kernel and the O(1)/CFS schedulers with 2.6. In 2005, after the BitKeeper dispute, Torvalds wrote **git**. The first Android phone shipped in 2008.

Timeline from UNIX to Linux

1969–1983 — UNIX is born

1969	UNIX, PDP-7 (Bell Labs)
1971	ported to the PDP-11
1973	rewritten in C
1977	BSD (Berkeley)
1983	System V, 4.2BSD
1983	GNU project

1984–1993 — The search for a free UNIX

1987	MINIX (Tanenbaum)
1988	POSIX standard
1989	GPL license
1991	Linux 0.01 (Torvalds)
1992	0.12: switch to the GPL
1993	Slackware, Debian

1994–2010 — Linux matures

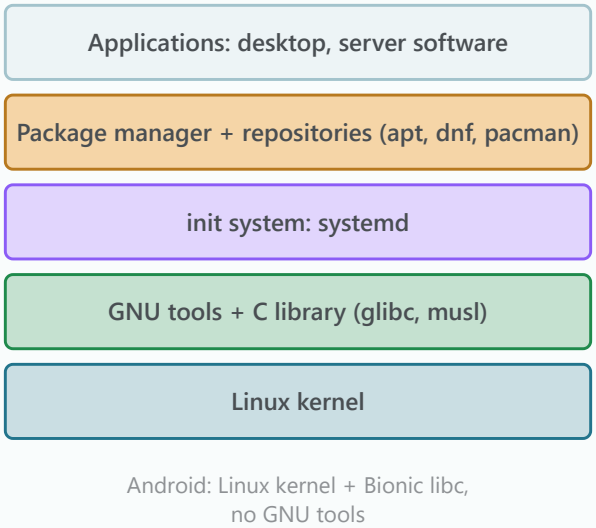
1994	Linux 1.0
1996	2.0: SMP support
1999	Red Hat IPO
2003	2.6 series
2005	git
2008	Android: HTC Dream

2011–2026 — Linux everywhere

2011	**version 3.0**
2017	**Top500: 100% Linux**
2022	**6.0; Rust in 6.1**
2023	**6.6: EEVDF**
2024	**6.12: sched_ext, RT**
2026	**6.x → 7.0 series**

Since 2017 all of the world's 500 fastest supercomputers run Linux. The 6.x series brought Rust support, EEVDF and sched_ext; version jumps carry no meaning (7.0 follows 6.19).

GNU, Linux and Distributions



Family	Examples	Package
Debian	Debian, Ubuntu, Mint, Pardus	.deb / apt
Red Hat	Fedora, RHEL, Rocky, Alma	.rpm / dnf
Arch	Arch, Manjaro	pacman
SUSE	openSUSE, SLES	zypper
Lightweight	Alpine (musl + BusyBox)	apk

Release model: a major release every ~9–10 weeks (a 2-week merge window + rc1…rc7). The last release of each year becomes **LTS**: 6.6, 6.12, 6.18…

Strictly, “Linux” is only the kernel's name; the system users install is a **distribution** (GNU/Linux).

Design Goals and the UNIX Philosophy

Simple
A few well-defined concepts: process, file, pipe.

Elegant
Small tools that do one thing combine to do big jobs.

Consistent
“Everything is a file”: disks, terminals and pipes all use the same read/write.

Powerful
Shell scripts, pipelines, regular expressions.

Flexible
Text streams are the universal interface; tools work together without knowing about each other.

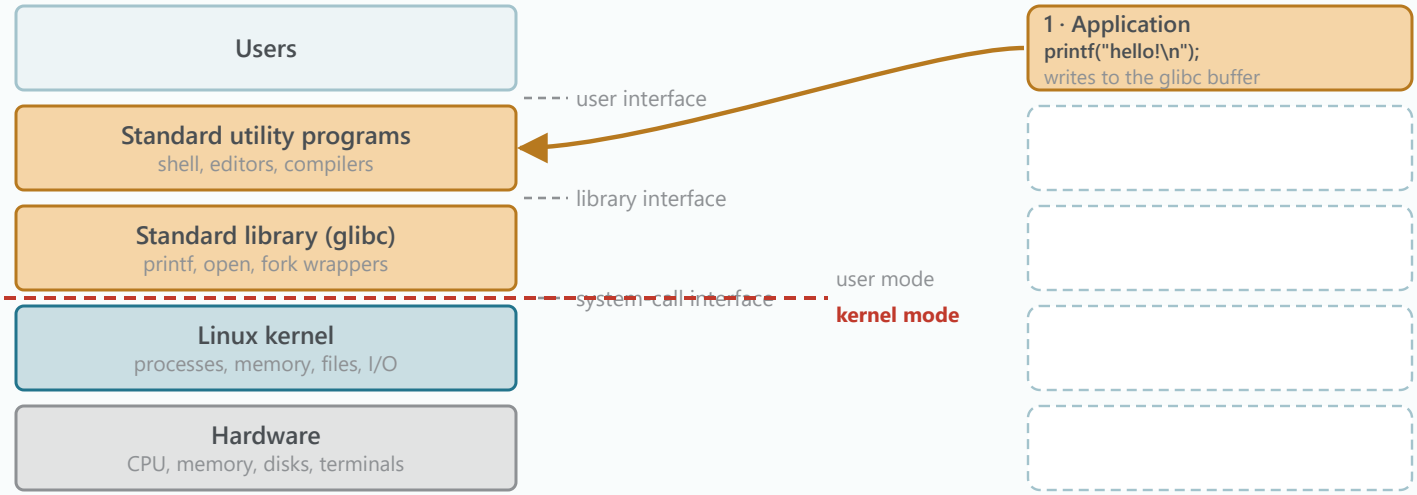
For programmers
Designed by programmers for programmers: little “magic”, silent success.

The UNIX philosophy (McIlroy): Write programs that do one thing and do it well, and that work together. Write programs to handle text streams, because that is a universal interface.



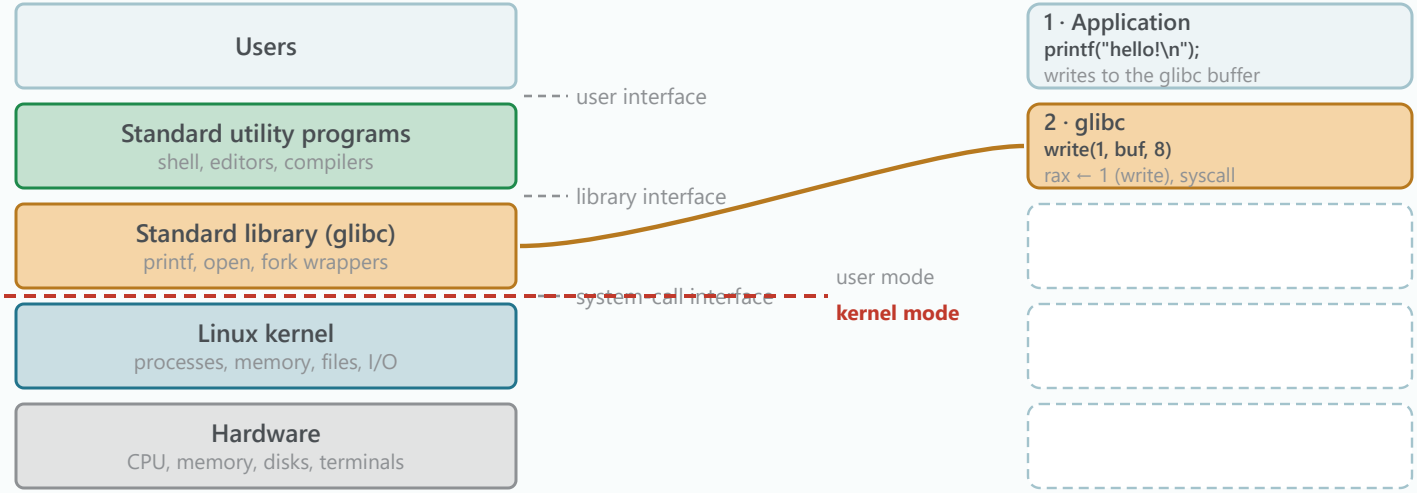
POSIX standardizes this interface: the same source compiles on Linux, BSD and macOS.

Layers and Interfaces: The Journey of a printf



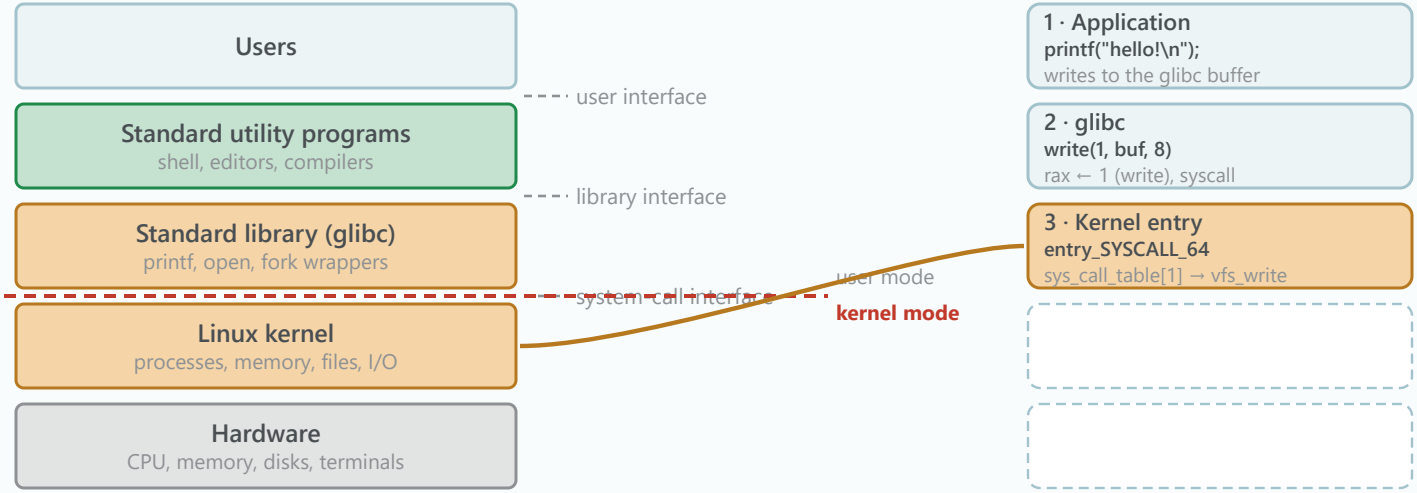
The program calls `printf`. This is not a system call yet: glibc writes the text into a buffer in user space.

Layers and Interfaces: The Journey of a printf



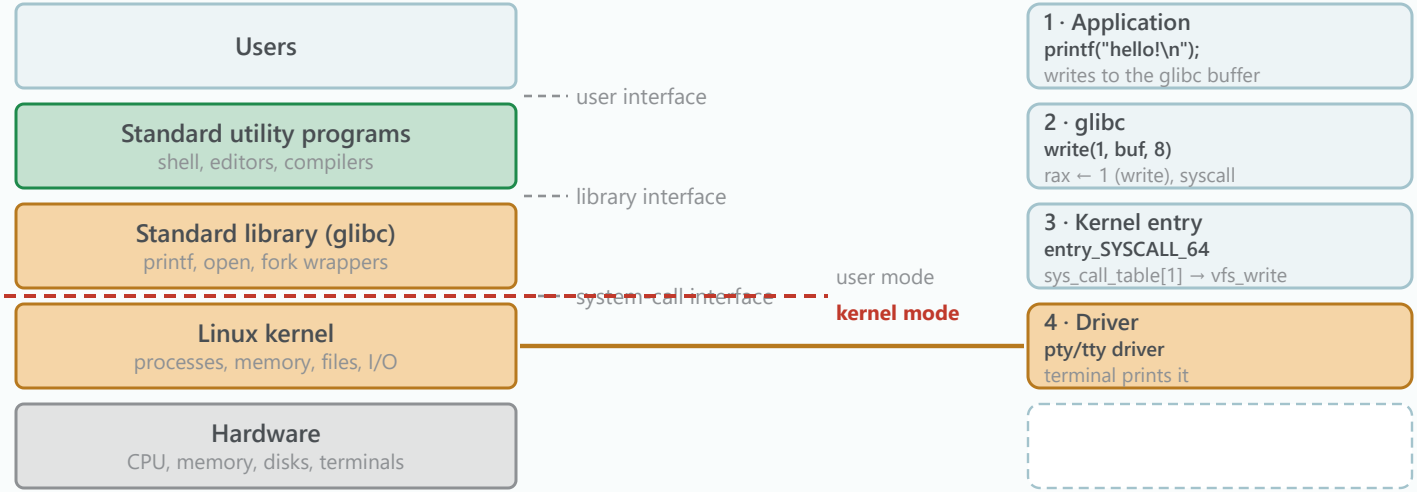
When the buffer is flushed, glibc's `write()` wrapper runs: it puts the system-call number (write = 1 on x86-64) in `rax` and the arguments in registers, and executes the `syscall` instruction.

Layers and Interfaces: The Journey of a printf



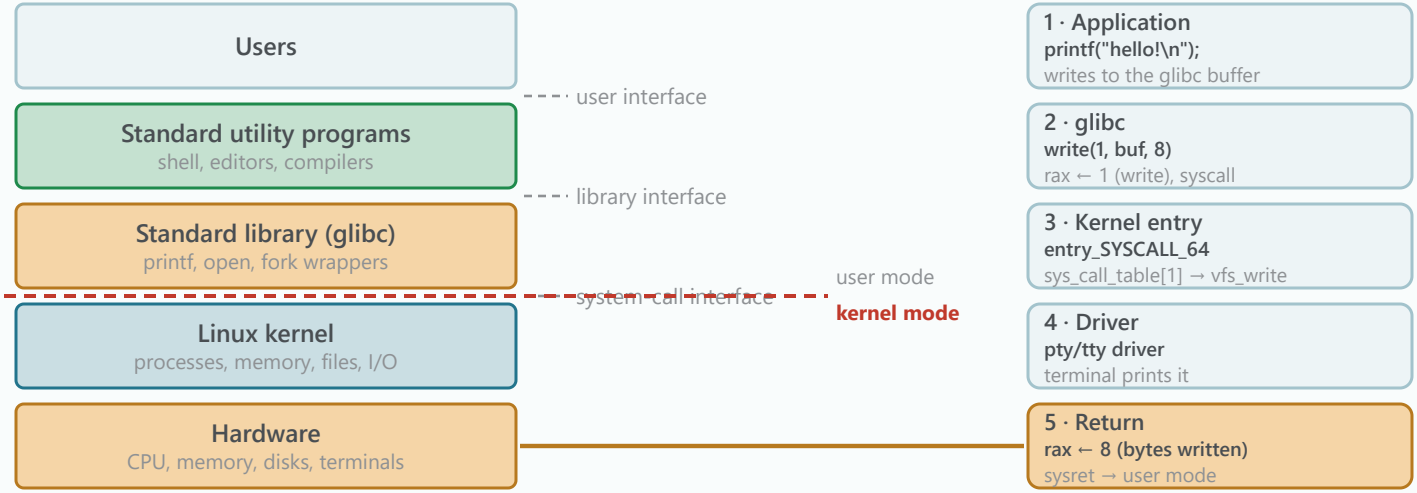
The CPU switches to kernel mode. The entry code uses the number to pick the handler from `sys_call_table`; the VFS finds the object behind file descriptor 1.

Layers and Interfaces: The Journey of a printf



fd 1 is attached to a pseudoterminal (pty); the tty layer passes the data to the terminal emulator. Had it been a disk, the path would go to the page cache and the block layer.

Layers and Interfaces: The Journey of a printf



The return value comes back in `rax` (8 bytes), and `sysret` returns to user mode. On error, glibc turns the negative value into `errno` and returns −1.

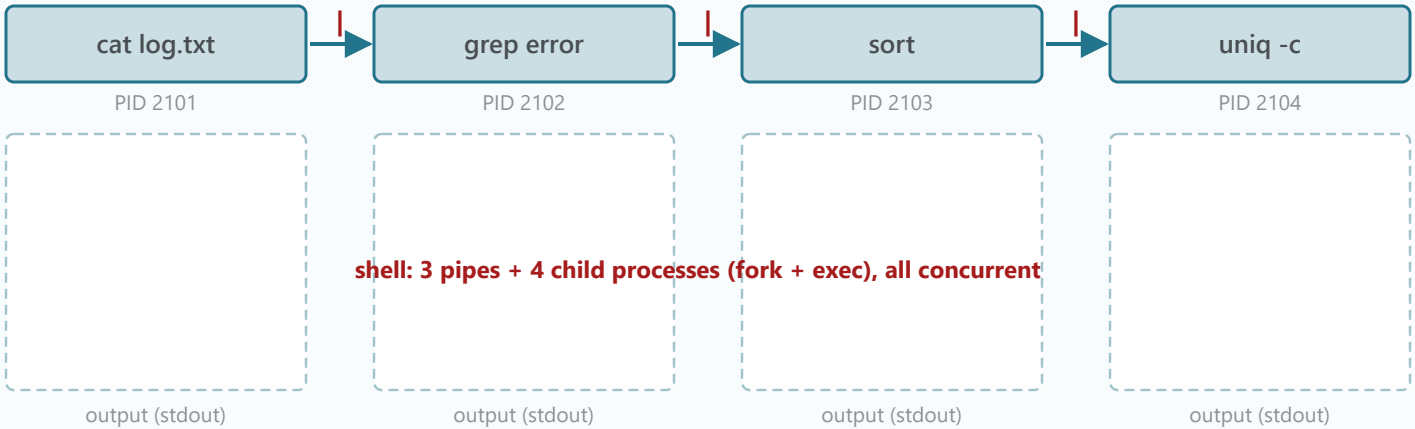
Basic Commands and Redirection

Category	Commands	Note
Files / directories	`ls cd pwd mkdir rm -r cp mv touch find tar`	`touch` updates the timestamp; creates an empty file if none exists
Text (filters)	`cat less head tail grep sort uniq wc echo`	read stdin, write stdout: they combine in pipelines
Processes / system	`ps top kill chmod chown df du sync`	`sync` flushes dirty buffers **to disk** (syncing between machines: `rsync`)
Network	`ssh scp ping curl wget traceroute`	`ip addr` , `ip route` , `ss` , `nft` ; they replaced the old `ifconfig` , `route` , `netstat` and `iptables`
Information	`man uname free lsblk`	`/proc` is not a command but a pseudo file system: `cat /proc/meminfo`

```
# redirection: 0 = stdin, 1 = stdout, 2 = stderr
sort < input.txt > output.txt 2> errors.log   # < reads, > writes (overwrite)
echo "line" >> log.txt                        # >> appends
make 2>&1 | tee build.log &                   # stderr→stdout, pipe, background
```

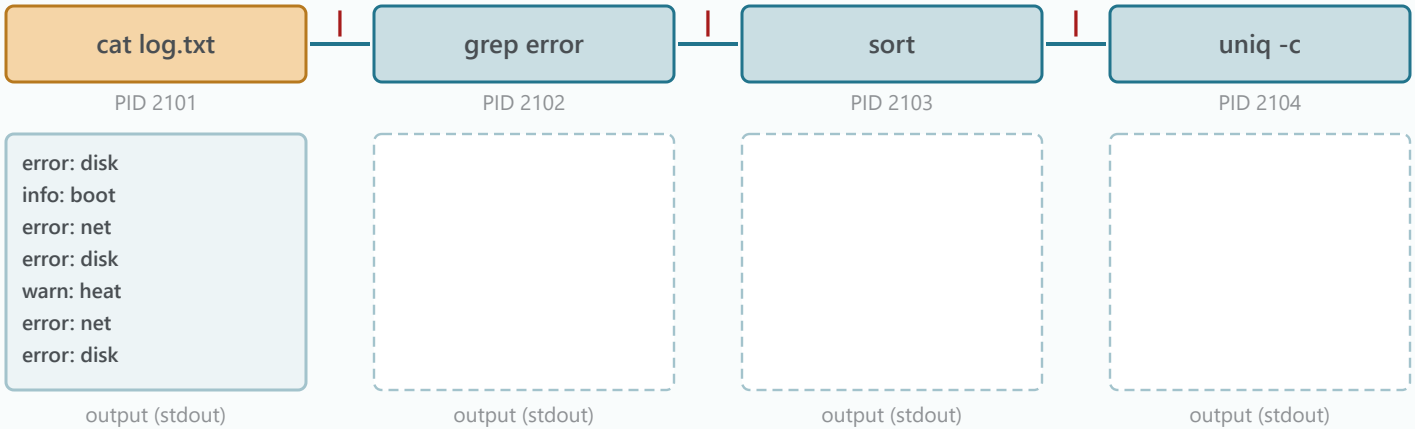


Pipelines: Data Flows Through the Stages



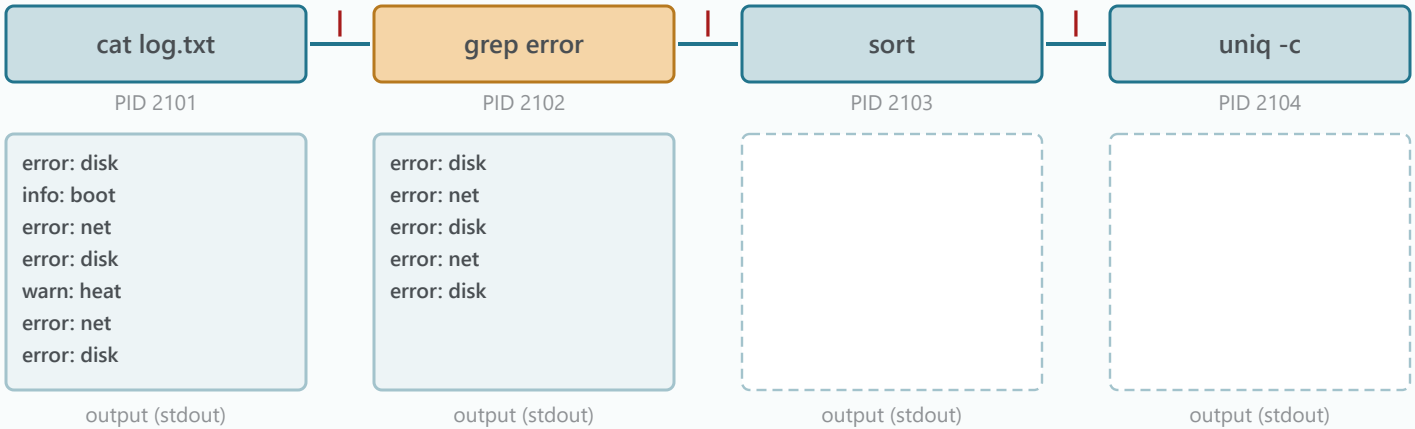
The shell parses the line and creates 3 pipes and 4 processes. The processes run **at the same time**; each knows only its own stdin/stdout.

Pipelines: Data Flows Through the Stages



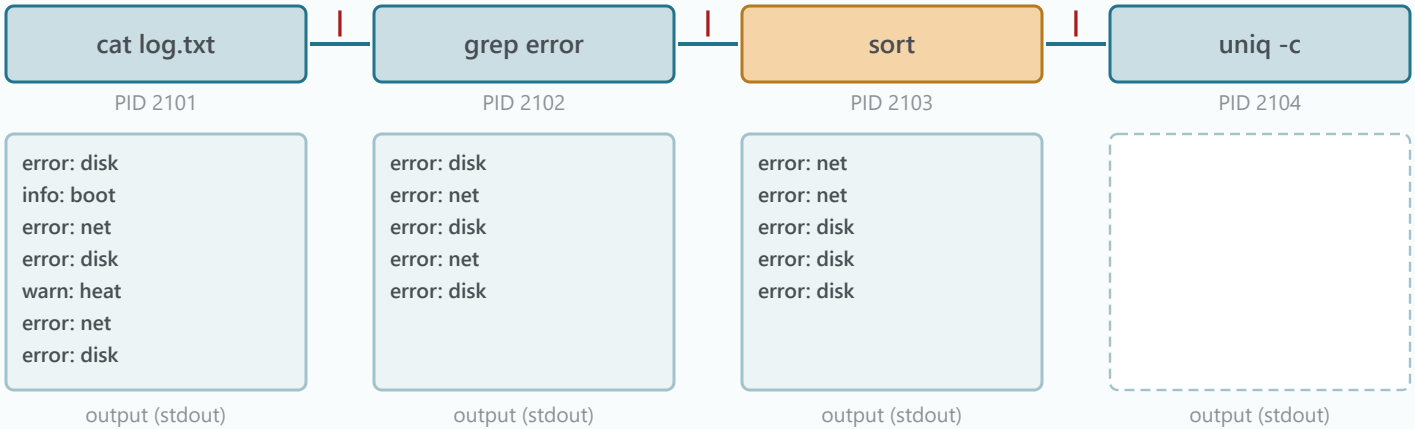
`cat` writes the file's 7 lines into the pipe. If the pipe buffer (64 KiB by default) fills, the writer **blocks**: producer–consumer.

Pipelines: Data Flows Through the Stages



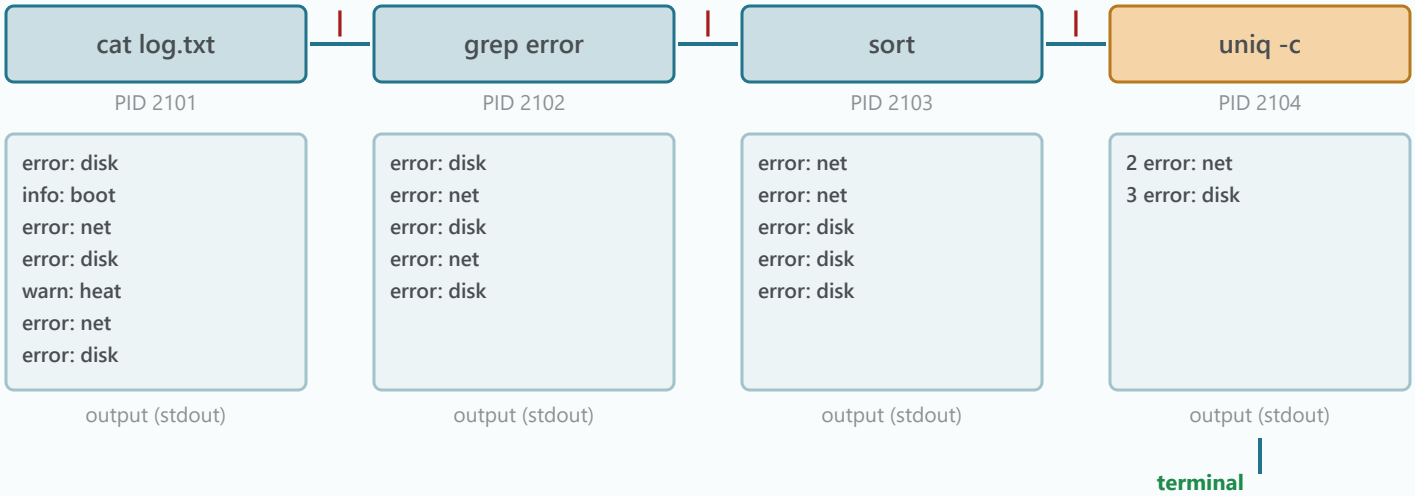
`grep` reads line by line and passes the 5 matching lines on at once. A filter is stateless; it does not hold up the flow.

Pipelines: Data Flows Through the Stages



`sort`, however, cannot output anything before reading **all of its input** (up to EOF): the last line could change the first position.

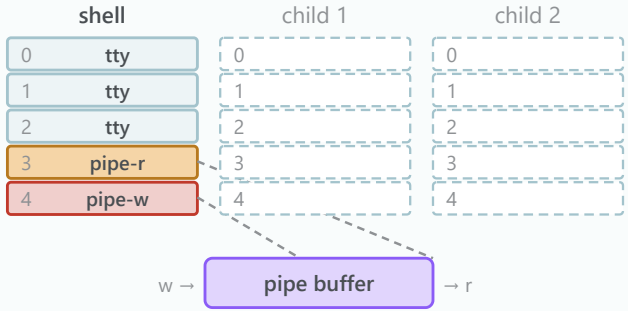
Pipelines: Data Flows Through the Stages



`uniq -c` counts only **adjacent** repeats, which is why sort comes first. Result: 2 × “error: net”, 3 × “error: disk”.

How Does the Shell Build a Pipeline?

```
▶ int fd[2];
▶ pipe(fd);                 /* 3: read, 4: write */
3 if (fork() == 0) {     /* child 1 */
4     dup2(fd[1], 1);
5     close(fd[0]); close(fd[1]);
6     execlp("ls", "ls", NULL);
7 }
8 if (fork() == 0) {     /* child 2 */
9     dup2(fd[0], 0);
10    close(fd[0]); close(fd[1]);
11    execlp("wc", "wc", "-l", NULL);
12 }
13 /* shell: close both ends */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`pipe()` creates a buffer in the kernel and two ends: the lowest free numbers, **3 (read)** and **4 (write)**.

Step 1 / 6

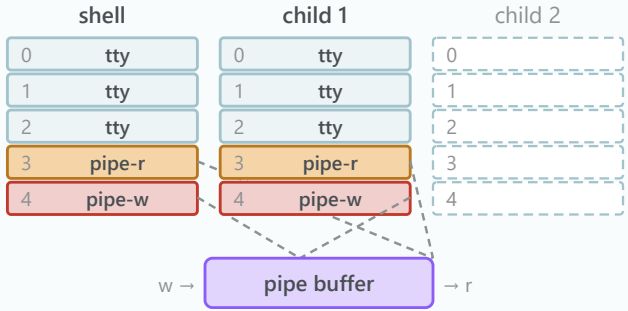
VARIABLES

`fd[0] = 3` `fd[1] = 4`

OUTPUT

How Does the Shell Build a Pipeline?

```
1  int fd[2];
2  pipe(fd);                /* 3: read, 4: write */
▶  if (fork() == 0) {       /* child 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {       /* child 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* shell: close both ends */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`fork()` : the child gets a **copy** of the descriptor table; both processes point to the same pipe.

Step 2 / 6

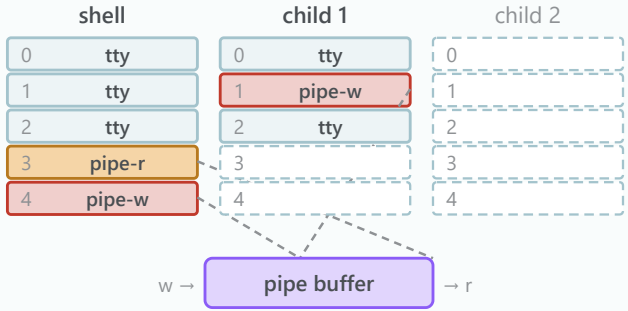
VARIABLES

`fd[0] = 3` `fd[1] = 4` `child1 = PID 2201`

OUTPUT

How Does the Shell Build a Pipeline?

```
1  int fd[2];
2  pipe(fd);                  /* 3: read, 4: write */
3  if (fork() == 0) {     /* child 1 */
▶      dup2(fd[1], 1);
▶      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* child 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* shell: close both ends */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`dup2(4, 1)` : stdout is now the pipe's write end. The extra ends are closed.

Step 3 / 6

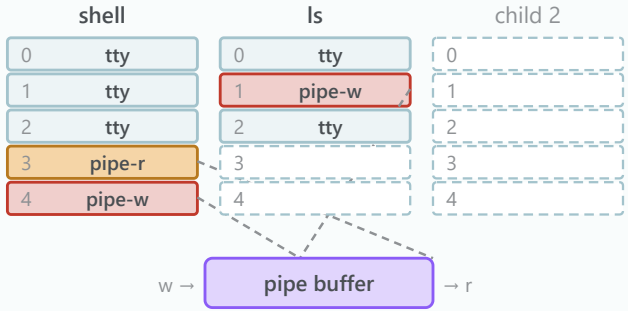
VARIABLES

`fd[0] = 3` `fd[1] = 4` `child1 = PID 2201`

OUTPUT

How Does the Shell Build a Pipeline?

```
1  int fd[2];
2  pipe(fd);                  /* 3: read, 4: write */
3  if (fork() == 0) {     /* child 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
▶      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* child 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
13 /* shell: close both ends */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



`execlp` replaces the image with `ls`; the descriptor table is **kept**. ls writes into the pipe without knowing it.

Step 4 / 6

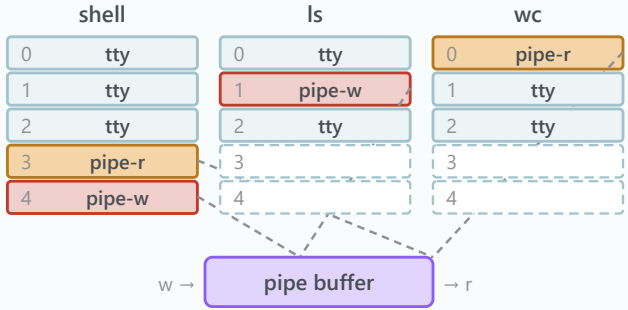
VARIABLES

`fd[0] = 3` `fd[1] = 4` `child1 = ls`

OUTPUT

How Does the Shell Build a Pipeline?

```
1  int fd[2];
2  pipe(fd);                  /* 3: read, 4: write */
3  if (fork() == 0) {     /* child 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
▶  if (fork() == 0) {     /* child 2 */
▶      dup2(fd[0], 0);
▶      close(fd[0]); close(fd[1]);
▶      execlp("wc", "wc", "-l", NULL);
12 }
13 /* shell: close both ends */
14 close(fd[0]); close(fd[1]);
15 wait(NULL); wait(NULL);
```



The second child connects its stdin to the pipe's read end and becomes `wc -l`.

Step 5 / 6

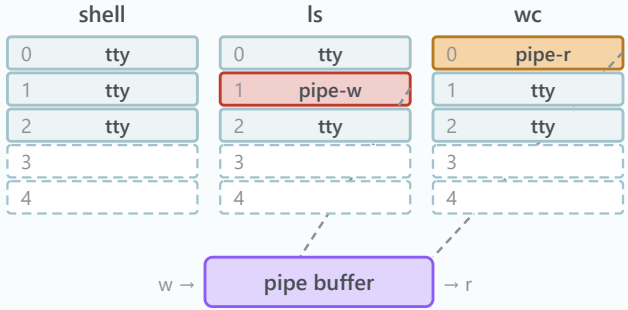
VARIABLES

`fd[0] = 3` `fd[1] = 4` `child1 = ls` `child2 = wc`

OUTPUT

How Does the Shell Build a Pipeline?

```
1  int fd[2];
2  pipe(fd);                /* 3: read, 4: write */
3  if (fork() == 0) {     /* child 1 */
4      dup2(fd[1], 1);
5      close(fd[0]); close(fd[1]);
6      execlp("ls", "ls", NULL);
7  }
8  if (fork() == 0) {     /* child 2 */
9      dup2(fd[0], 0);
10     close(fd[0]); close(fd[1]);
11     execlp("wc", "wc", "-l", NULL);
12 }
▶ /* shell: close both ends */
▶ close(fd[0]); close(fd[1]);
▶ wait(NULL); wait(NULL);
```



The shell closes both ends. Otherwise the write end would stay open, and wc would **never see EOF** and wait forever. Then it waits for both children.

Step 6 / 6

VARIABLES

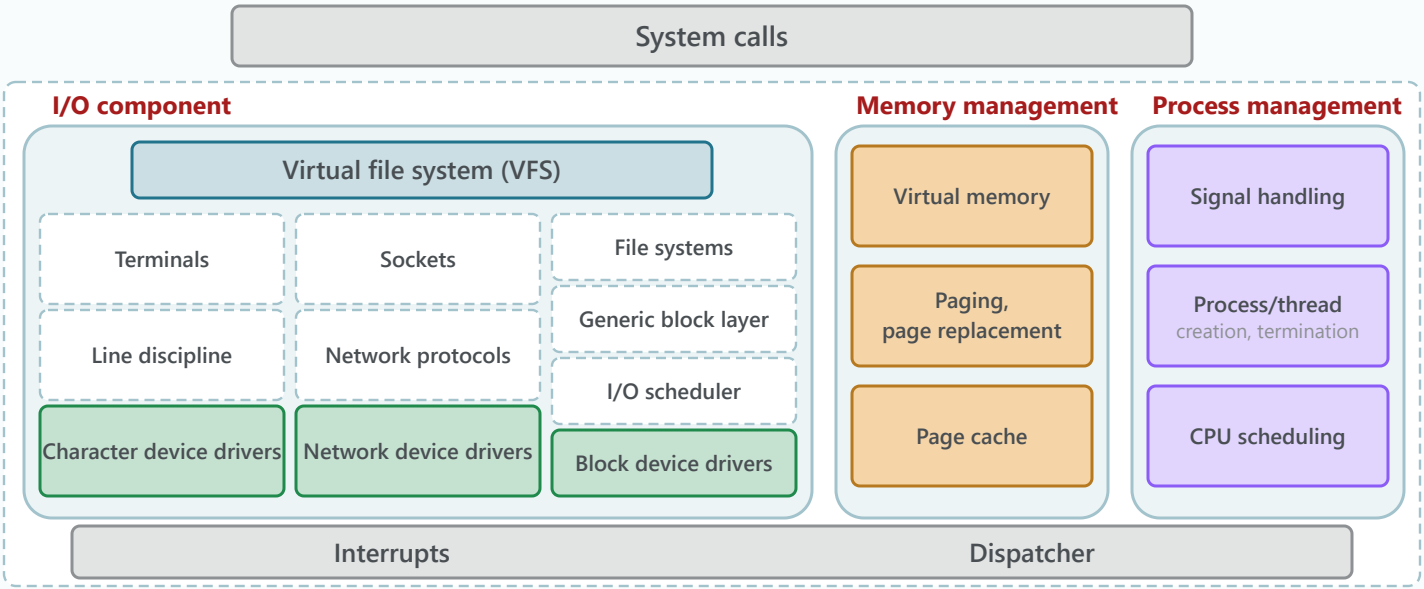
`fd[0] = –` `fd[1] = –` `child1 = done` `child2 = done`

OUTPUT

```
7
```

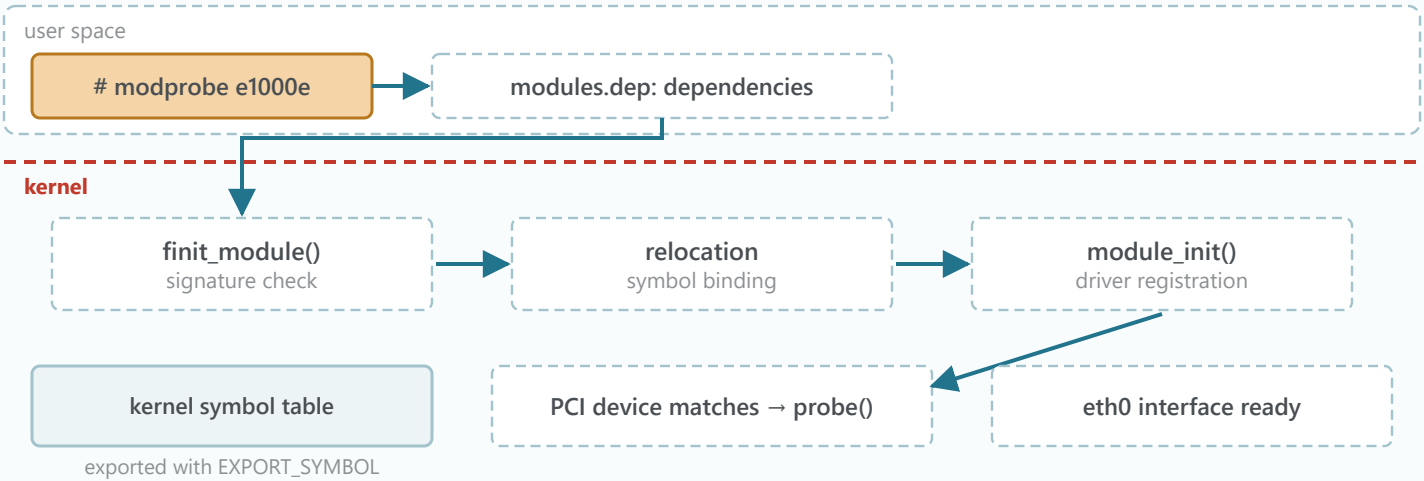
KERNEL STRUCTURE

Structure of the Linux Kernel



The system-call interface on top, interrupts and the dispatcher (context switching) at the bottom. All components run **in a single address space**: Linux is a monolithic kernel. The classic 4.4BSD kernel split into the same three components.

Loadable Kernel Modules (LKMs)

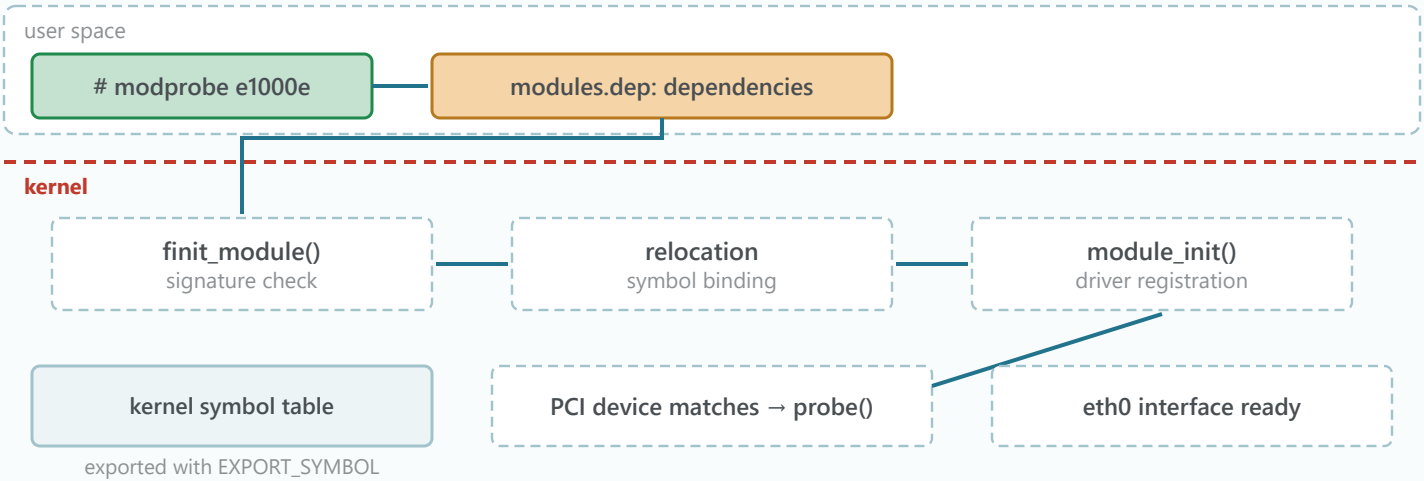


Monolithic but **modular**: most drivers are loadable kernel modules (`.ko`). The administrator requests one with `modprobe` , or udev loads it automatically when the device appears.

Step 1 / 6

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

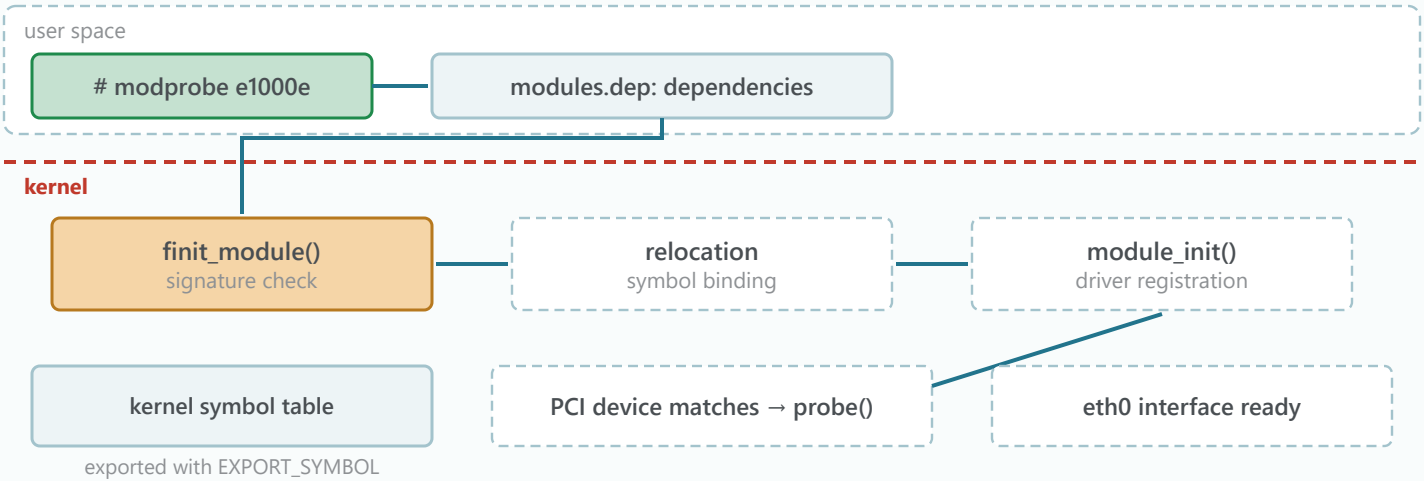
Loadable Kernel Modules (LKMs)



`modprobe` finds the dependencies in `/lib/modules/$(uname -r)/modules.dep` and loads them first (`insmod` does not).

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

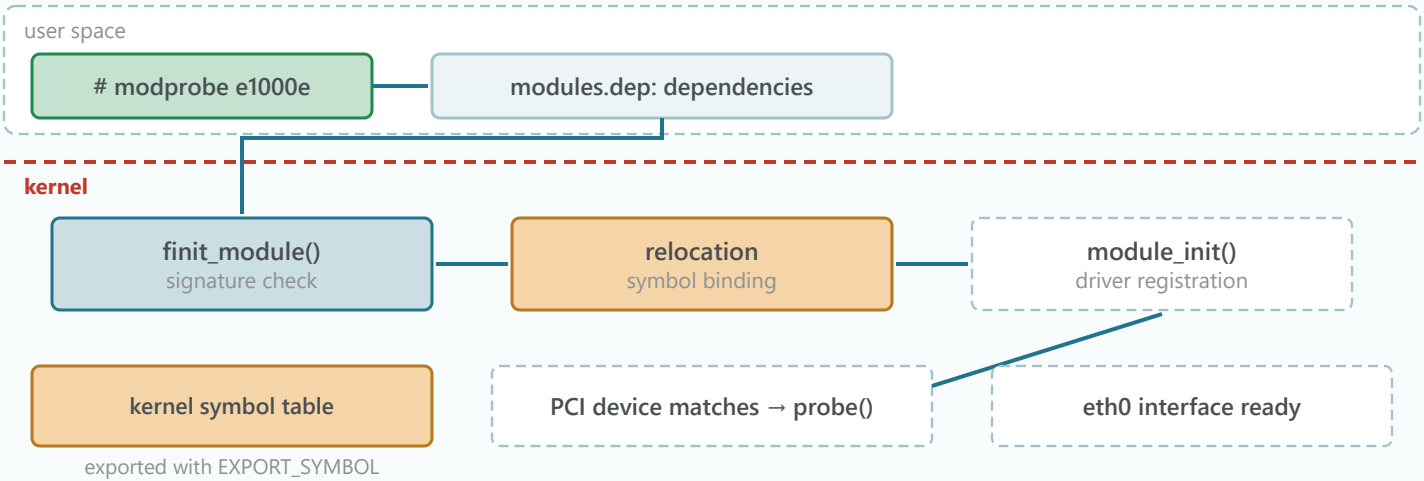
Loadable Kernel Modules (LKMs)



The `finit_module()` system call copies the module into the kernel; with Secure Boot on, its **signature** is checked. It requires the CAP_SYS_MODULE capability.

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

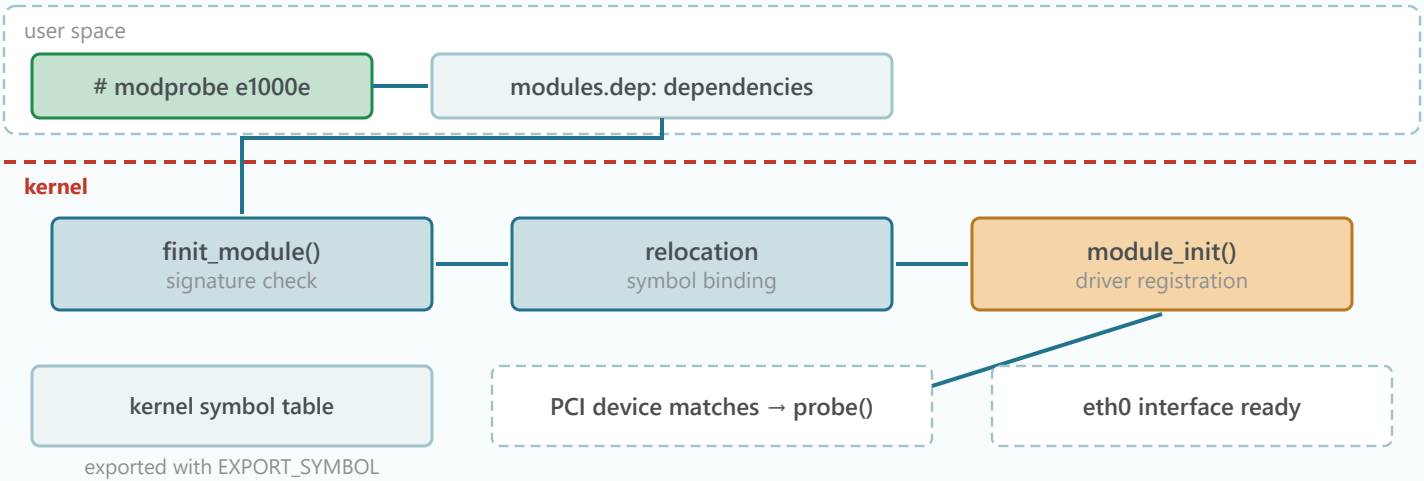
Loadable Kernel Modules (LKMs)



The kernel applies the module's relocations and binds the symbols it uses to kernel functions exported with `EXPORT_SYMBOL`.

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

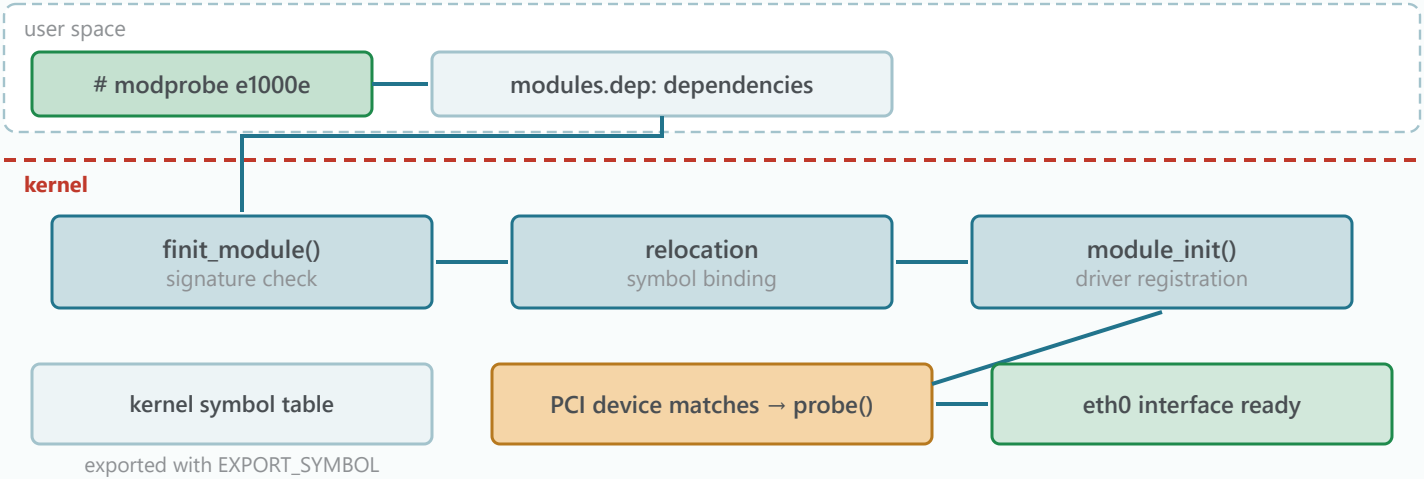
Loadable Kernel Modules (LKMs)



The `module_init` function runs: the driver registers itself with the PCI subsystem (`pci_register_driver`).

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

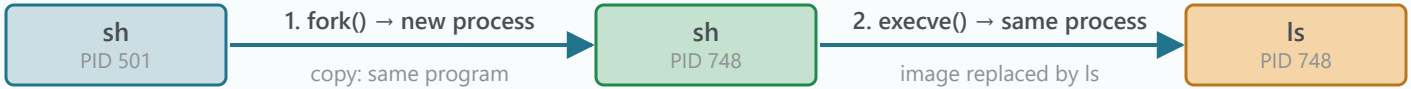
Loadable Kernel Modules (LKMs)



`probe()` is called for the matching device and the network interface appears. `lsmod` lists modules, `rmmod` removes one. A module runs with kernel privilege: its bug can crash the whole system.

The source tree is ~40 million lines, mostly drivers. The kernel has had Rust support since 6.1, and the first Rust driver arrived in 6.8; eBPF runs small, safe, verified programs in the kernel.

What Do fork and exec Do in the Kernel?



fork() in the kernel

- Allocate a new `task_struct`, fill it from the parent, assign a new **PID**
- Allocate a kernel stack; copy registers (return value 0 in the child)
- Copy the page tables and mark writable pages **read-only** (COW)
- Share open files: the `struct file` reference count goes up
- Signal handlers, cwd and credentials are copied

execve() in the kernel

- The ELF header is read; the old `mm_struct` is dropped, a new one is built and the segments are `mmap`ped
- A new `cred` is prepared according to the setuid/setgid bits
- PC ← `ld.so` (the dynamic linker) or the program's entry point
- A failed `execve` returns −1; by convention the child exits with `_exit(127)`

Old UNIX really copied data and stack on fork; Linux copies only the page tables (COW). Since exec usually follows fork at once, very few pages are ever copied.

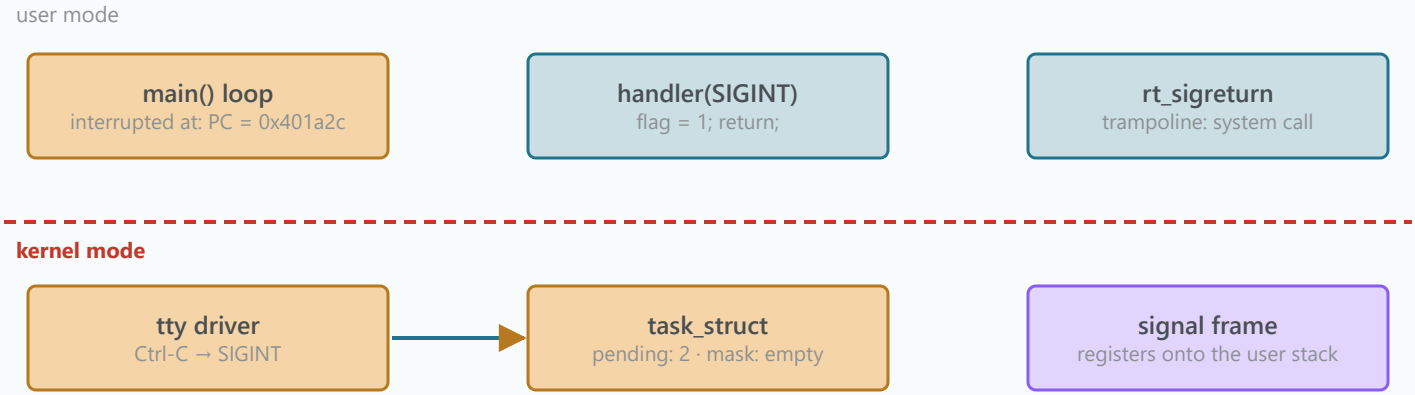
POSIX Signals

A signal is a software interrupt that **notifies** a process of an event. It can be **synchronous** (SIGSEGV, SIGFPE: caused by the process's own instruction) or **asynchronous** (SIGINT, SIGTERM: from outside). For each signal a process chooses the default action, ignoring it, or a **handler** (cleanup, logging).

Signal	No	When is it raised?	Default
SIGHUP	1	The terminal hung up; for daemons, “reload your configuration”	terminate
SIGINT	2	Ctrl-C (to the foreground process group)	terminate
SIGQUIT	3	Ctrl-\	core dump
SIGILL	4	Illegal machine instruction	core dump
SIGABRT	6	`abort()`: the process terminates itself abnormally	core dump
SIGFPE	8	Arithmetic error such as integer division by zero	core dump
SIGKILL	9	`kill -9`; **cannot be caught or ignored**	terminate
SIGSEGV	11	Invalid memory access	core dump
SIGPIPE	13	Writing to a pipe with no readers	terminate
SIGALRM	14	`alarm()` timer expired	terminate
SIGTERM	15	The **default** of the `kill` command; a polite request to terminate	terminate
SIGUSR1/2	10/12	User-defined	terminate
SIGCHLD	17	A child stopped or terminated	ignore
SIGSTOP / SIGCONT	19/18	Stop (uncatchable) / continue; Ctrl-Z → SIGTSTP	stop / continue

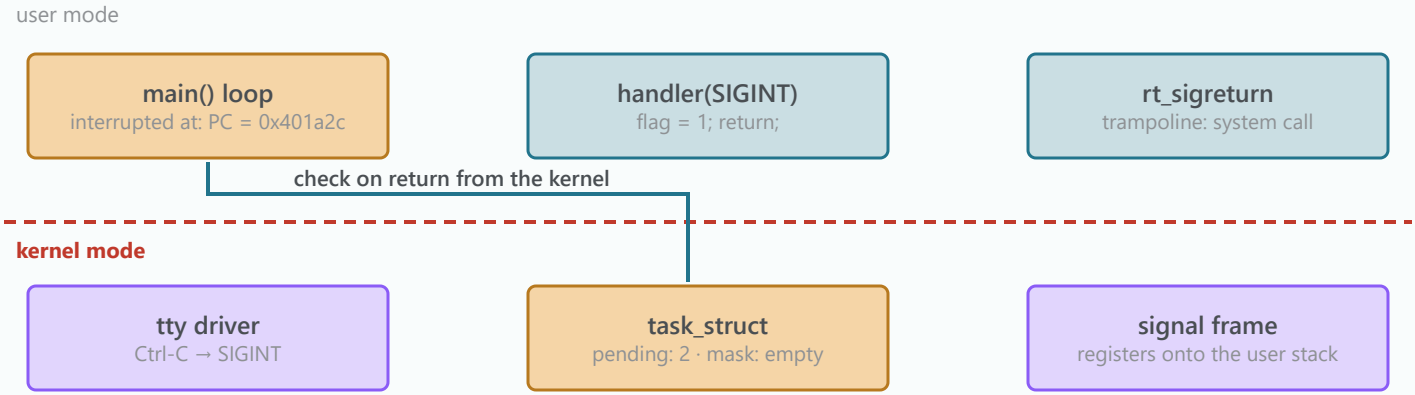
Numbers are for x86/ARM Linux. SIGKILL and SIGTERM can both be sent with the `kill` command; the difference is whether they can be caught.

Signal Delivery



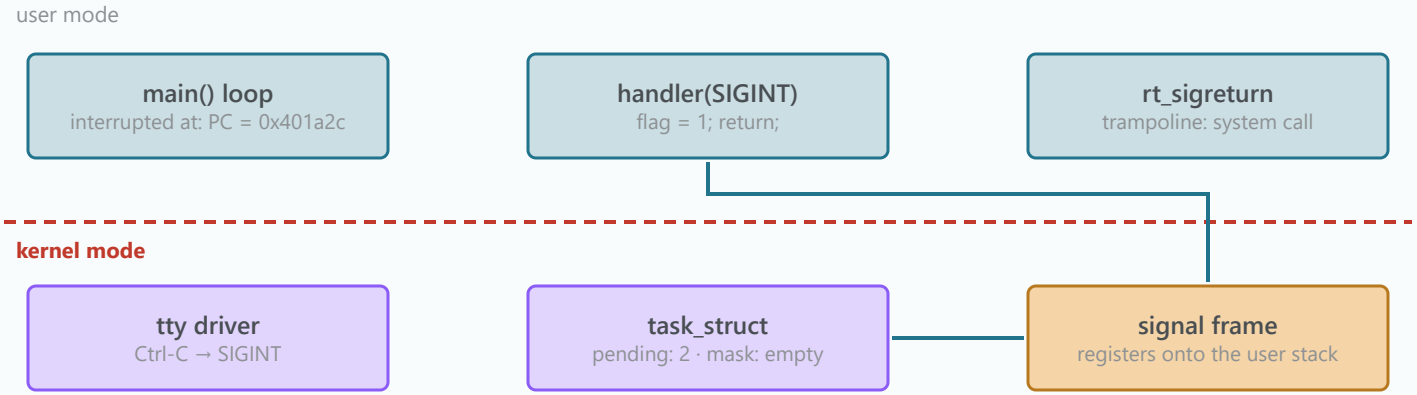
The user presses Ctrl-C: the tty line discipline raises **SIGINT** for the foreground process group. The kernel merely sets bit 2 in the target task's **pending** signal set.

Signal Delivery



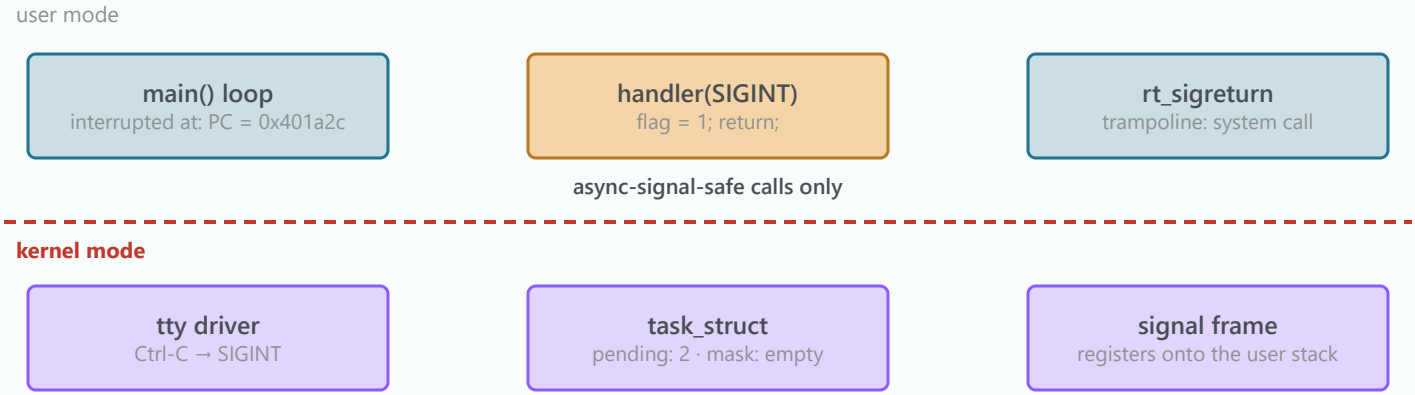
The signal is not handled at once. When the process **returns to user mode** from a system call or an interrupt, the kernel checks for signals that are pending and unmasked.

Signal Delivery



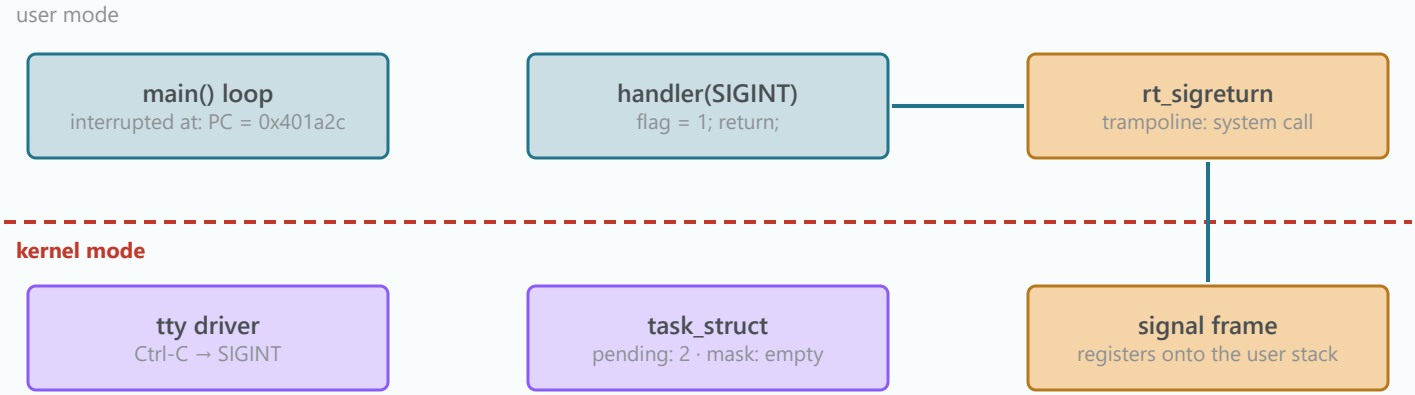
If a handler was installed with `sigaction`, the kernel writes a **signal frame** (saved registers + a return trampoline) onto the user stack and sets the PC to the handler.

Signal Delivery



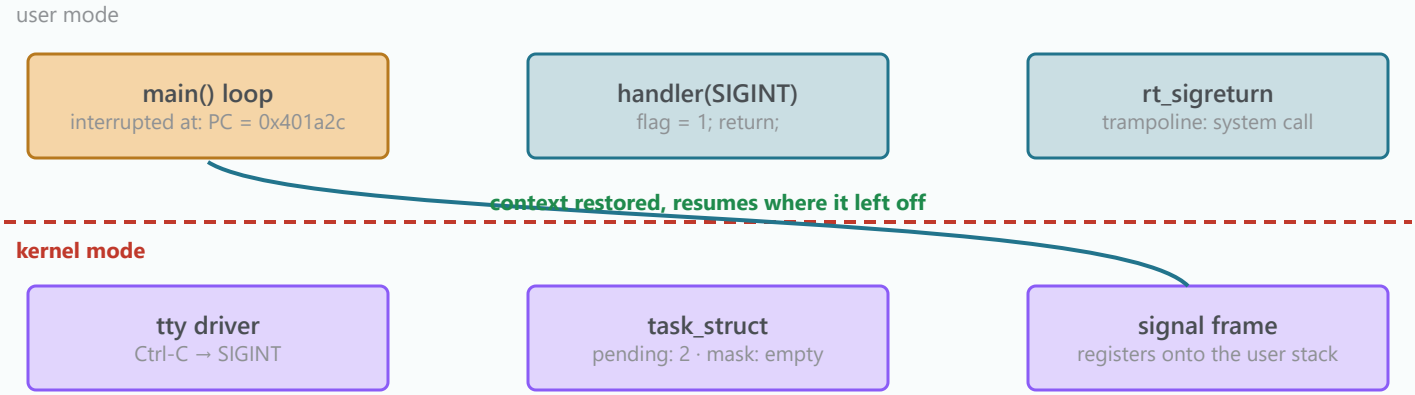
The handler runs in user mode. Only async-signal-safe functions are safe inside it (e.g. `write`, a `volatile sig_atomic_t` flag); `printf` and `malloc` are not.

Signal Delivery



When the handler returns, the trampoline makes the `rt_sigreturn` system call and the kernel restores the registers from the frame. (sigreturn does not **enter** the handler; it **returns** from it.)

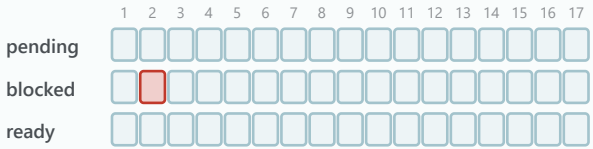
Signal Delivery



The process resumes at the interrupted instruction. Without a handler the default action would apply: termination for SIGINT.

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->pend &= ~(1UL << (sig - 1));
7          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
10 }
▶ /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGINT blocked

Before entering a critical section the process blocked SIGINT with sigprocmask (bit 2 of the blocked set). Signals are a bit set in each task.

Step 1 / 8

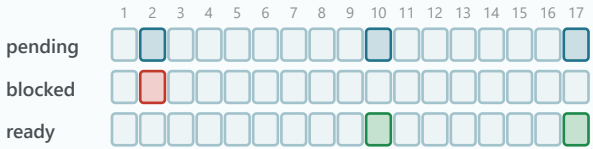
VARIABLES

`pend = {}` `blk = {2}`

OUTPUT

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->pend &= ~(1UL << (sig - 1));
7          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
10 }
▶ /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

Ctrl-C, kill -USR1, child exited

Three signals arrive while the process sleeps: Ctrl-C (2), kill -USR1 (10) and a child's exit (17). The kernel only sets bits in the pending set; nothing runs yet.

Step 2 / 8

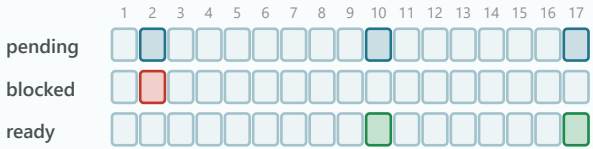
VARIABLES

`pend = {2,10,17}` `blk = {2}`

OUTPUT

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
▶  void check_signals(task *t) {
▶      unsigned long h = t->pend & ~t->blk;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->pend &= ~(1UL << (sig - 1));
7          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
10 }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

ready = pending & ~blocked = {10, 17}

As the process returns to user mode the kernel checks: ready = pending & ~blocked. SIGINT is pending but blocked; ready to deliver are **10** and **17**.

Step 3 / 8

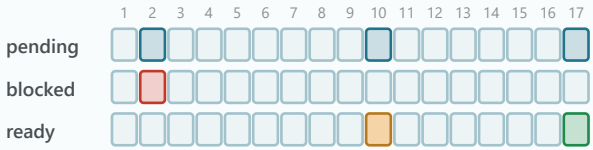
VARIABLES

`h = {10,17}`

OUTPUT

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
▶      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->pend &= ~(1UL << (sig - 1));
▶          deliver(t, sig);   /* handler/default */
▶          h = t->pend & ~t->blk;
9      }
10 }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGUSR1 → handler runs

The lowest ready bit (ctz): **10**. It is cleared from pending and delivered: SIGUSR1 → the handler runs. The ready set is recomputed.

Step 4 / 8

VARIABLES

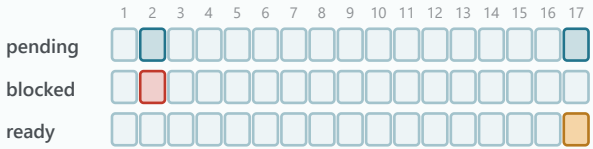
`sig = 10` `pend = {2,17}`

OUTPUT

```
delivered: 10 (SIGUSR1)
```

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
▶      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->pend &= ~(1UL << (sig - 1));
▶          deliver(t, sig);   /* handler/default */
▶          h = t->pend & ~t->blk;
9      }
10 }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGCHLD → default: ignore

The lowest ready bit (ctz): **17**. It is cleared from pending and delivered: SIGCHLD → default: ignore. The ready set is recomputed.

Step 5 / 8

VARIABLES

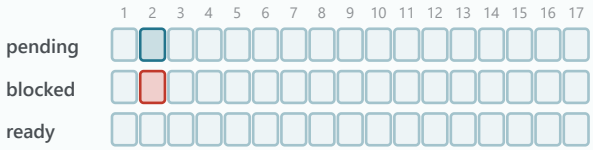
`sig = 17` `pend = {2}`

OUTPUT

```
delivered: 10 (SIGUSR1)
delivered: 17 (SIGCHLD)
```

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
▶      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->pend &= ~(1UL << (sig - 1));
7          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
▶  }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

ready empty: loop ends, SIGINT still pending

The ready set is empty, the loop ends and the process continues in user mode. SIGINT is not lost: it stays in the pending set.

Step 6 / 8

VARIABLES

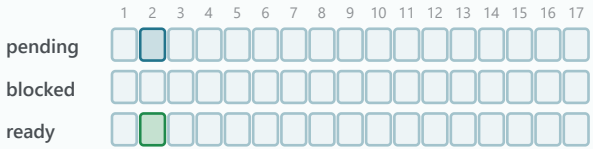
`h = {}` `pend = {2}`

OUTPUT

```
delivered: 10 (SIGUSR1)
delivered: 17 (SIGCHLD)
```

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
▶      unsigned long h = t->pend & ~t->blk;
4      while (h) {
5          int sig = __builtin_ctzl(h) + 1;
6          t->pend &= ~(1UL << (sig - 1));
7          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
10 }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
▶  /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

unblocked: ready = {2}

The critical section is over and SIGINT is unblocked. The same check runs on return from sigprocmask: ready = {2}.

Step 7 / 8

VARIABLES

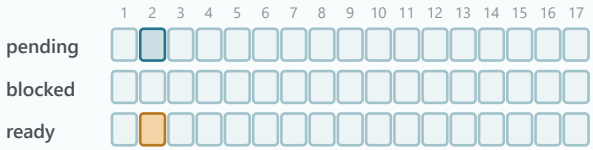
`blk = {}` `h = {2}`

OUTPUT

```
delivered: 10 (SIGUSR1)
delivered: 17 (SIGCHLD)
```

Choosing Pending Signals: Code Trace

```
1  /* on return to user mode (kernel) */
2  void check_signals(task *t) {
3      unsigned long h = t->pend & ~t->blk;
4      while (h) {
▶          int sig = __builtin_ctzl(h) + 1;
▶          t->pend &= ~(1UL << (sig - 1));
▶          deliver(t, sig);   /* handler/default */
8          h = t->pend & ~t->blk;
9      }
10 }
11 /* before: sigprocmask(SIG_BLOCK, {SIGINT}) */
12 /* after:  sigprocmask(SIG_UNBLOCK, {SIGINT}) */
```



2 SIGINT · 10 SIGUSR1 · 17 SIGCHLD

SIGINT → default: terminate

SIGINT is delivered; with no handler installed the default action applies: the process terminates. Blocking a signal defers it; it does not ignore it. Had the same signal arrived several times, it would have merged into one bit.

Step 8 / 8

VARIABLES

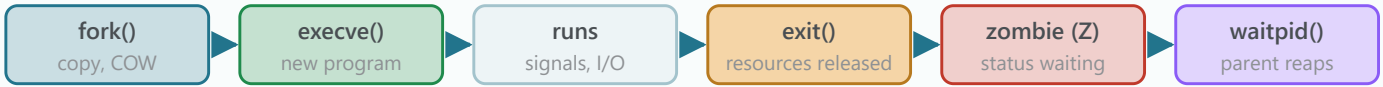
`sig = 2` `pend = {}`

OUTPUT

```
delivered: 10 (SIGUSR1)
delivered: 17 (SIGCHLD)
delivered: 2 (SIGINT → default:
terminate)
```

Process-Management System Calls

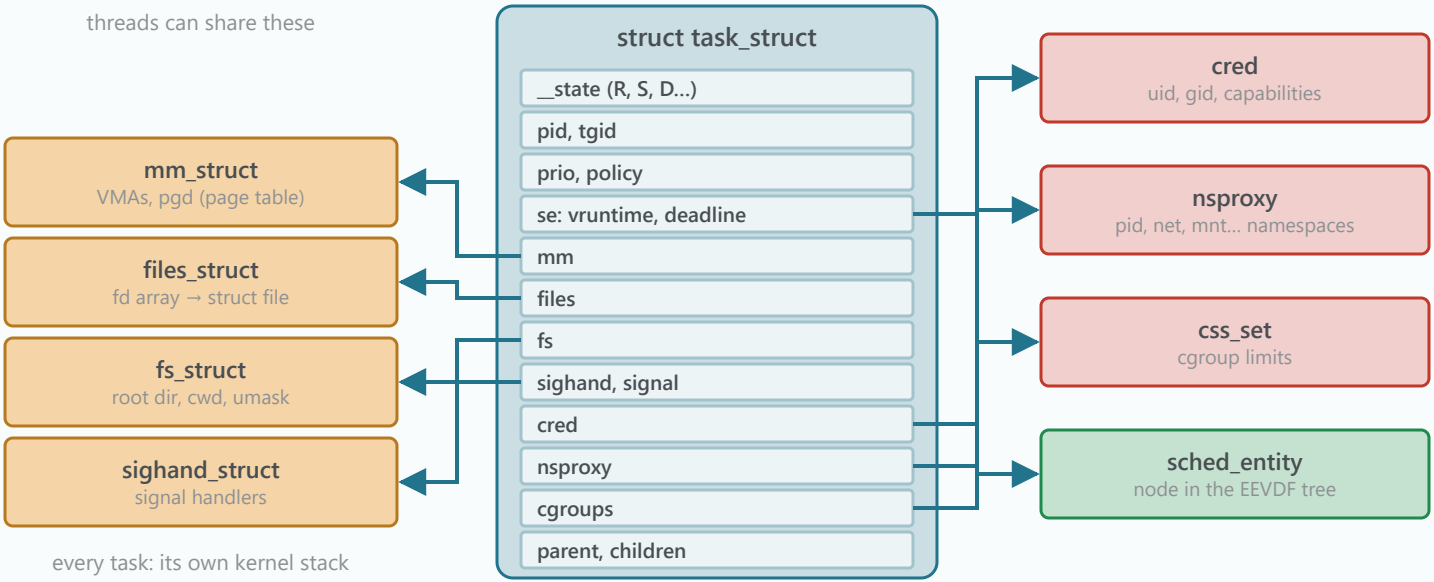
System call	Description
`fork()`	Creates a new process by copying the caller (COW)
`clone()` , `clone3()`	A new task with flags choosing what is shared (threads, containers)
`execve()`	Replaces the current process image with a new program
`waitpid()`	Waits for a child to finish and gets its exit status (reaps the zombie)
`exit()` / `_exit()`	Terminates the caller (exit flushes stdio buffers)
`kill()`	**Sends a signal** to a process; the signal need not mean termination
`sigaction()`	Sets the handler or action for a signal
`sigprocmask()`	Reads/changes the signal mask (blocked signals)
`sigpending()`	Returns the signals pending because they are blocked
`sigsuspend()`	Temporarily changes the mask and sleeps until a signal arrives
`rt_sigreturn()`	Restores the saved context on return from a handler
`alarm()` / `pause()`	SIGALRM after n seconds / sleep until a signal arrives



signals via kill()/sigaction() and timing via alarm()/pause() can step in at any stage

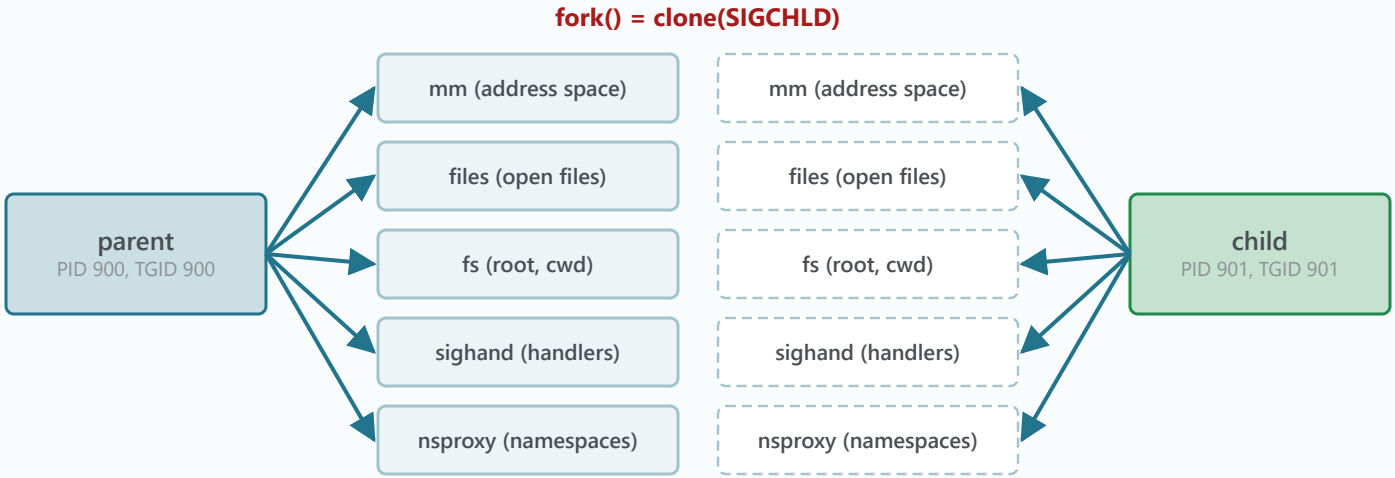
A system call is the only way for a user program to ask the kernel for service; Linux x86-64 has ~460 of them. The process tree is shown by `pstree -p` or `ps -ef --forest`.

The Process Descriptor: task_struct



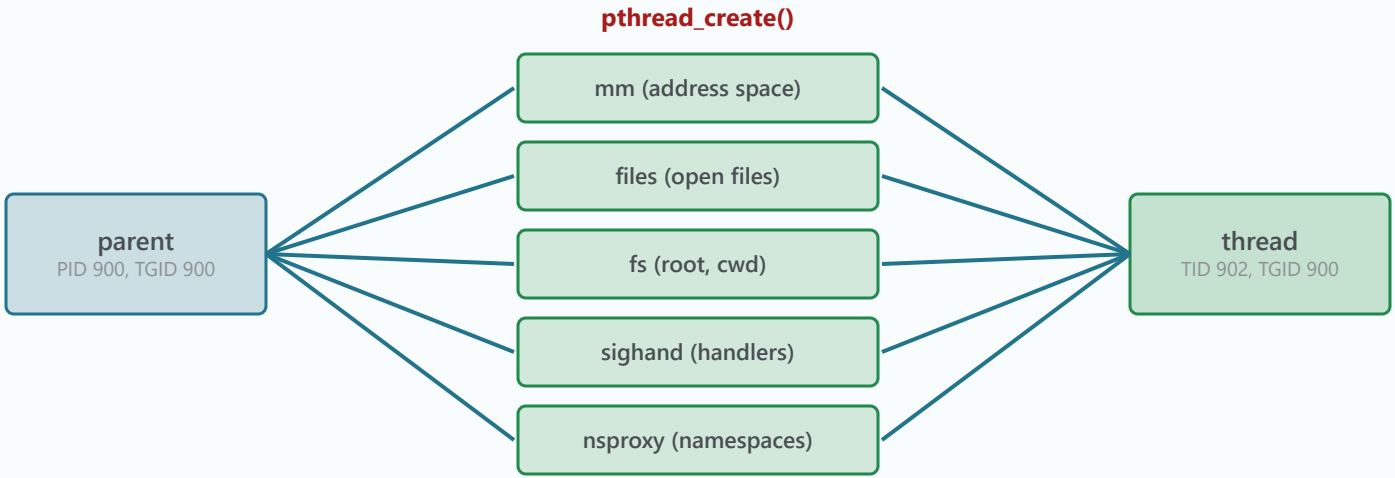
In Linux there is no structural difference between a process and a thread: both are a **task** (task_struct). The difference is which substructures they **share**. The PID users see is really the `tgid` (thread group ID). With `__state` D (uninterruptible sleep) a task does not die at once even on SIGKILL, which is why most waits now use TASK_KILLABLE.

clone() and Threads



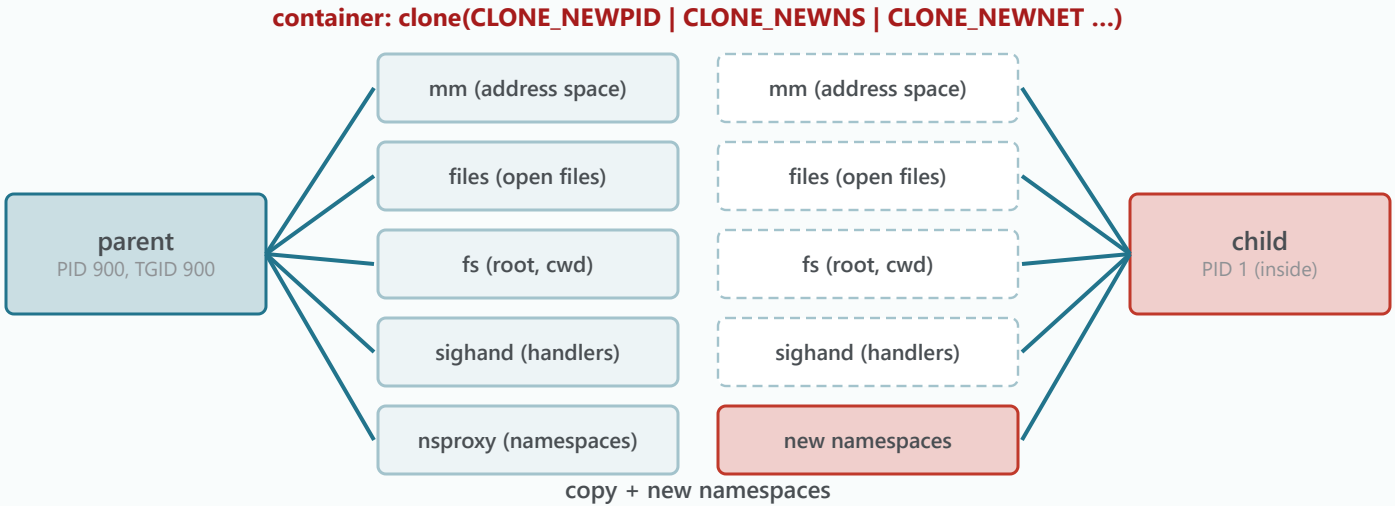
In glibc, `fork()` is a `clone()` called with just the `SIGCHLD` flag: the child gets a **copy** of every structure (memory via COW). Its own PID and TGID: a new process.

clone() and Threads



`pthread_create()` → `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD | …)`. Everything is shared; the new task joins the same TGID. **NPTL**: every thread is a kernel task (the 1:1 model).

clone() and Threads



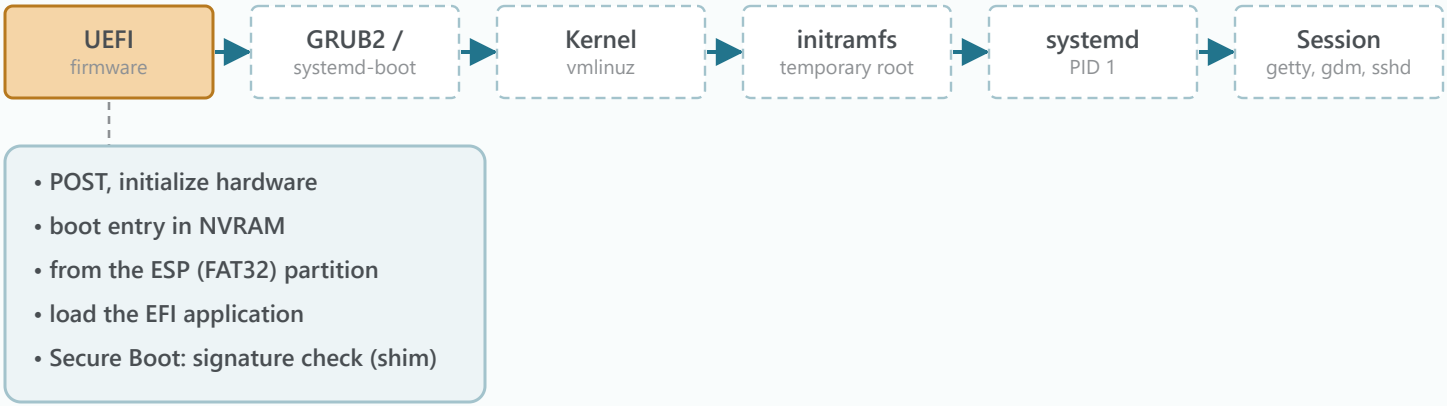
Namespace flags (`CLONE_NEWPID` , `CLONE_NEWNS` , `CLONE_NEWNET` ...) give the child a new view: inside, it believes it is PID 1. This is the basis of containers.

From CFS to EEVDF: Fair Sharing

Version	Selection rule of the fair class	Data structure
2.4 (2001)	O(n): every pick scans all ready tasks	a single list
2.6.0–2.6.22 (2003)	O(1): a list per priority, active and expired arrays	array of 140 lists + bitmap
2.6.23–6.5 (2007)	CFS: smallest vruntime	red-black tree
6.6+ (2023)	EEVDF: earliest virtual deadline among eligible tasks ($v \le V$)	the same tree, augmented with subtree data

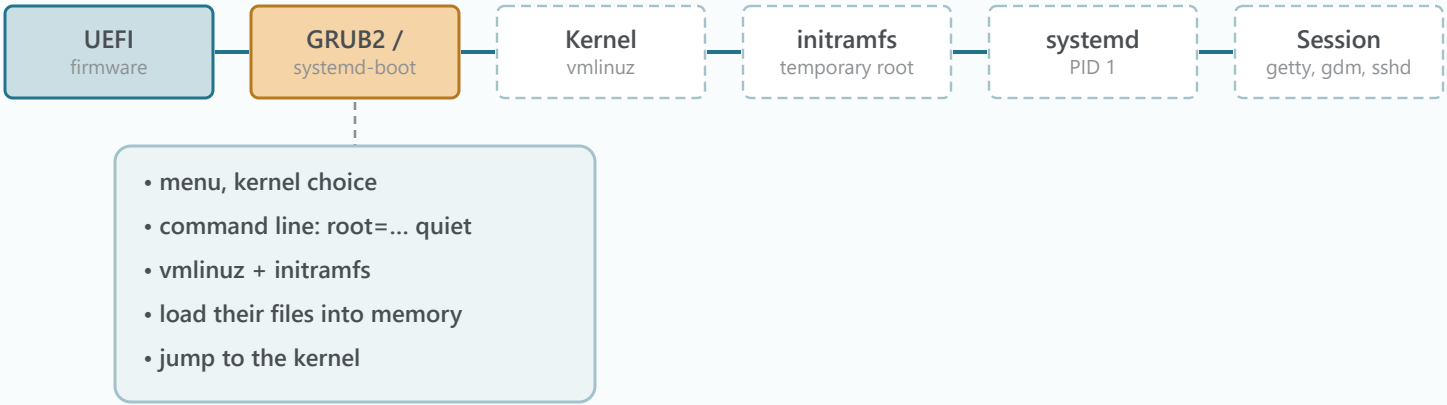
- Unchanged: the vruntime in `sched_entity`, the nice → weight table (`sched_prio_to_weight`, nice 0 → 1024) and the tree in the run queue.
- CFS's latency knobs (`sched_latency`, wakeup granularity) gave way to a single base slice (`base_slice_ns`); the virtual deadline is $d = v + \text{slice} \times 1024 / \text{weight}$.
- A latency-sensitive task can ask for a shorter slice with `sched_setattr()` (6.12+): a short slice means an early deadline and being picked more often.

Booting: From UEFI to systemd



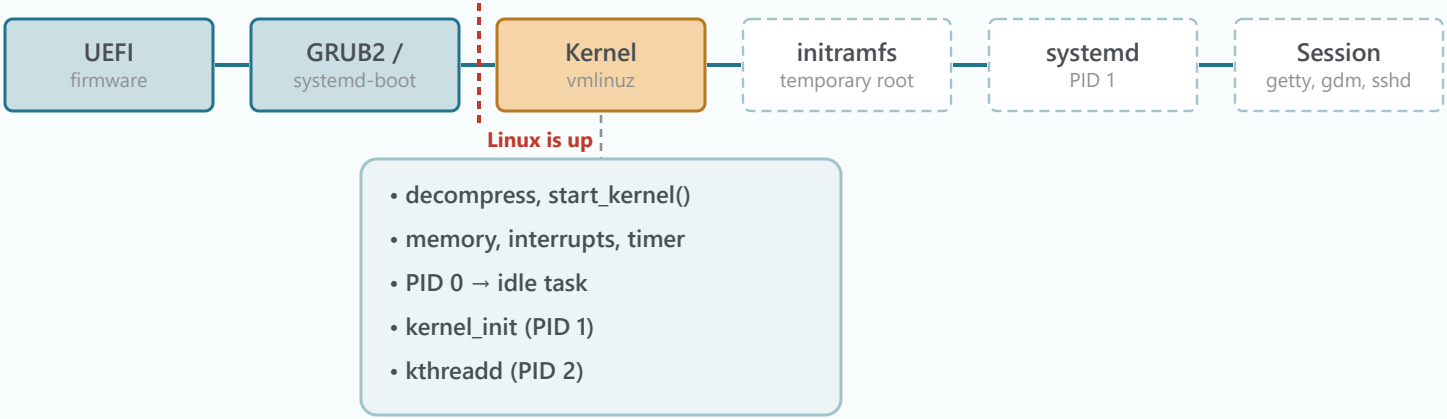
At power-on the **UEFI** firmware tests the hardware and loads the EFI application from the ESP partition. With Secure Boot on, each stage verifies the next one's signature.

Booting: From UEFI to systemd



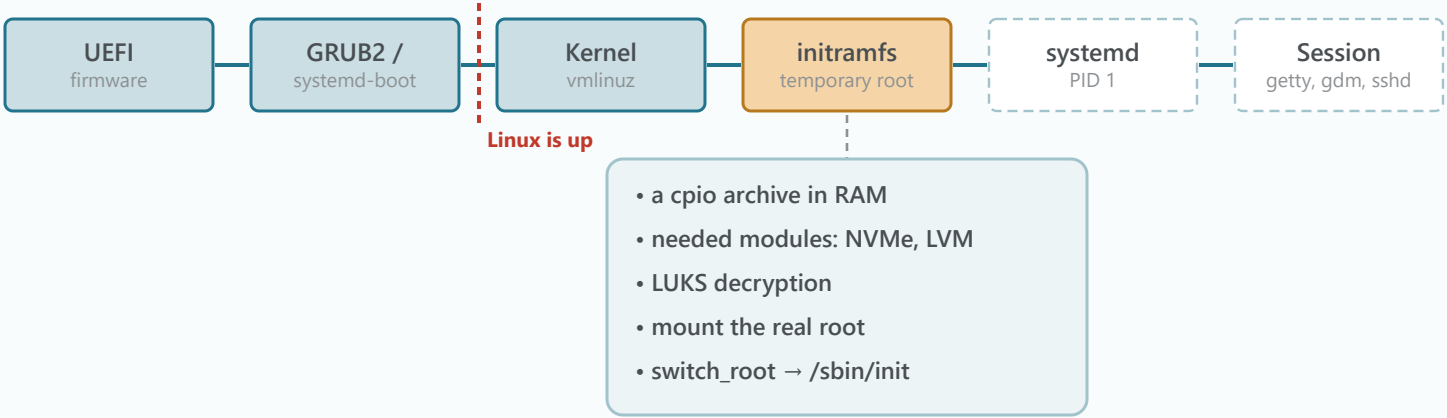
The bootloader loads the kernel image (`vmlinuz`) and the `initramfs` into memory and passes the command-line parameters. The kernel can also be started directly as an EFI application (UKI).

Booting: From UEFI to systemd



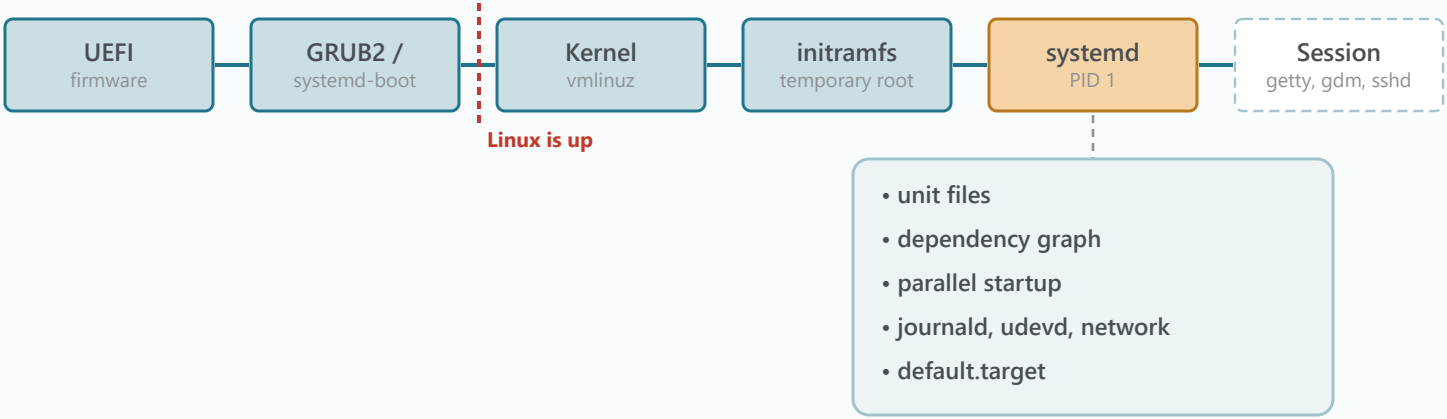
The kernel decompresses itself, and `start_kernel()` sets up memory management, interrupts and the scheduler. The first task becomes PID 0, the idle task; `kernel_init` (PID 1) and `kthreadd` (PID 2) are created.

Booting: From UEFI to systemd



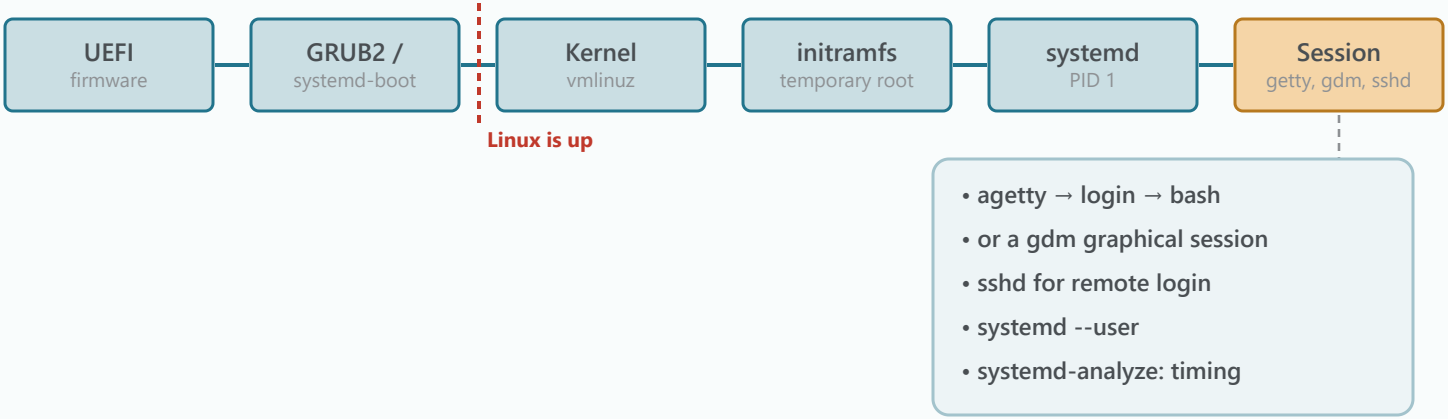
The drivers needed to mount the root file system may not be built into the kernel (NVMe, RAID, encrypted disks). The **initramfs** provides a temporary root in RAM, mounts the real root and does a `switch_root`.

Booting: From UEFI to systemd



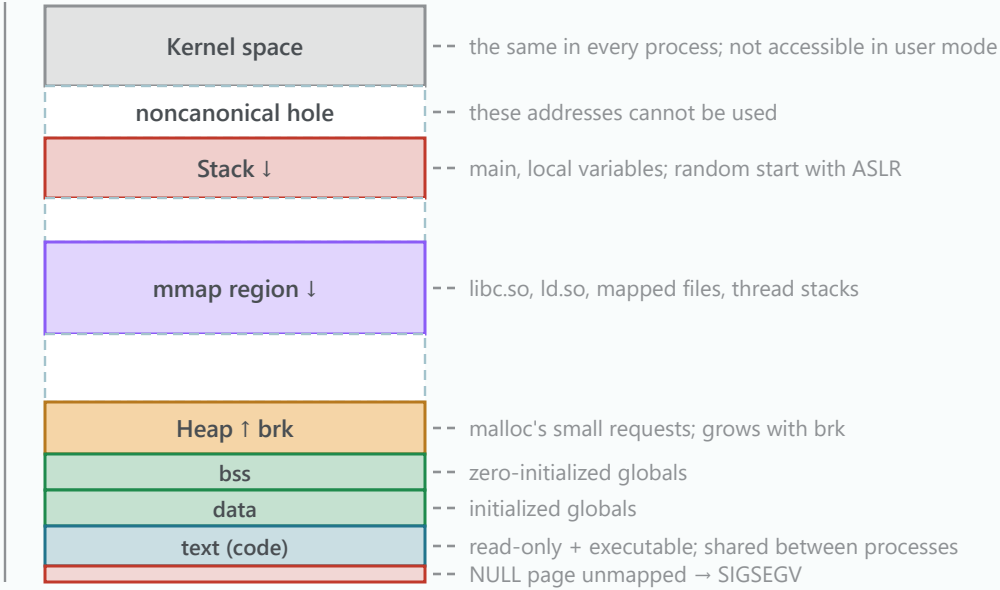
PID 1 is now **systemd**: it starts services **in parallel** according to the dependencies in the unit files and reaches the target (`multi-user.target` or `graphical.target`).

Booting: From UEFI to systemd



On the console `agetty` → `login` → shell; on the desktop gdm; remotely sshd. The classic UNIX chain init → getty → login → sh is still recognizable.

The Virtual Address Space of a Process



x86-64, 4 levels (48 bits):

user 2^{47} B = 128 TiB

kernel 128 TiB

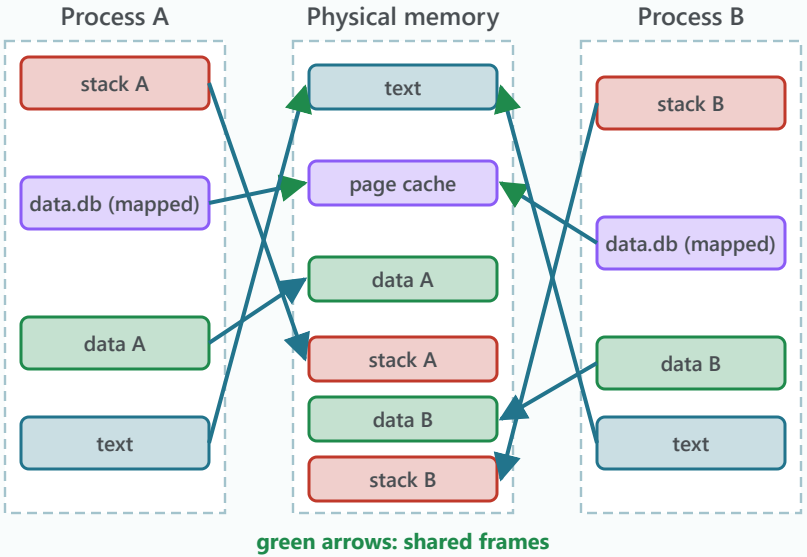
5 levels, LA57 (57 bits):

user 2^{56} B = 64 PiB

cat /proc/PID/maps

Each region is described by a **VMA** (vm_area_struct); VMAs are kept in a "maple tree" inside the `mm_struct`. The stack and mmap grow down and the heap grows up; the gap between them is only virtual and costs no physical memory.

Sharing, Mapped Files and Memory Calls

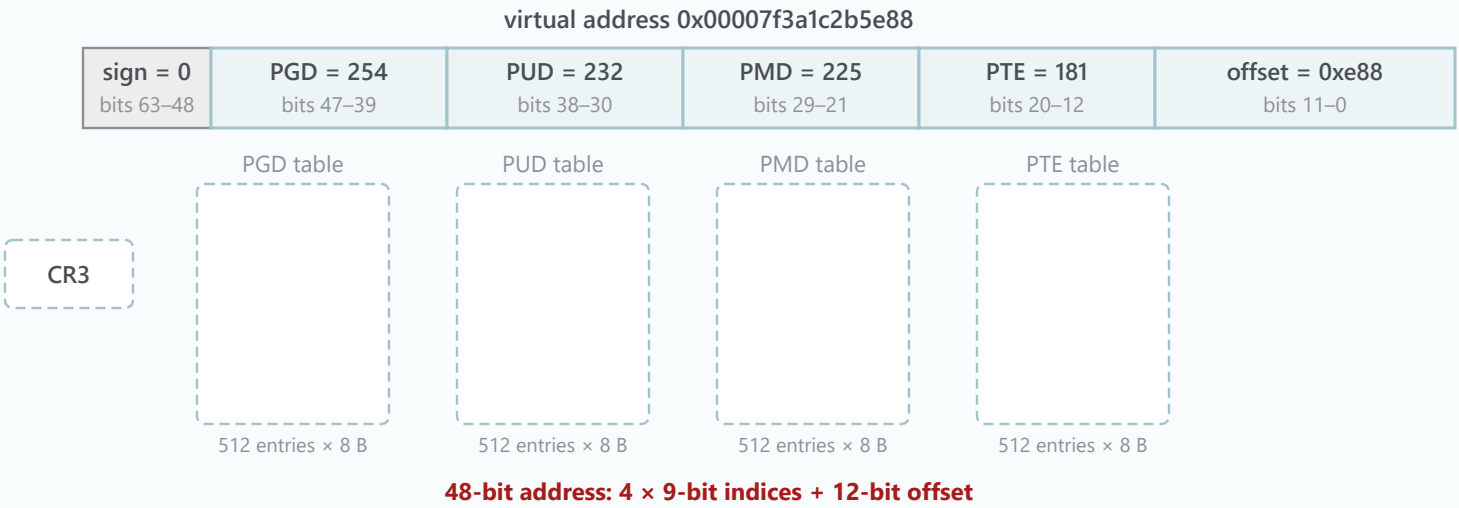


green arrows: shared frames

Call	Function
`brk` / `sbrk`	Grows/shrinks the heap (data segment) by moving its end
`mmap`	Maps a file, a device or anonymous memory into the address space
`munmap`	Removes a mapping and returns the memory to the system
`mprotect`	Changes a region's permissions (r/w/x)
`madvise`	Gives the kernel a usage hint (e.g. MADV_DONTNEED)
`msync`	Writes a shared mapping's dirty pages to the file

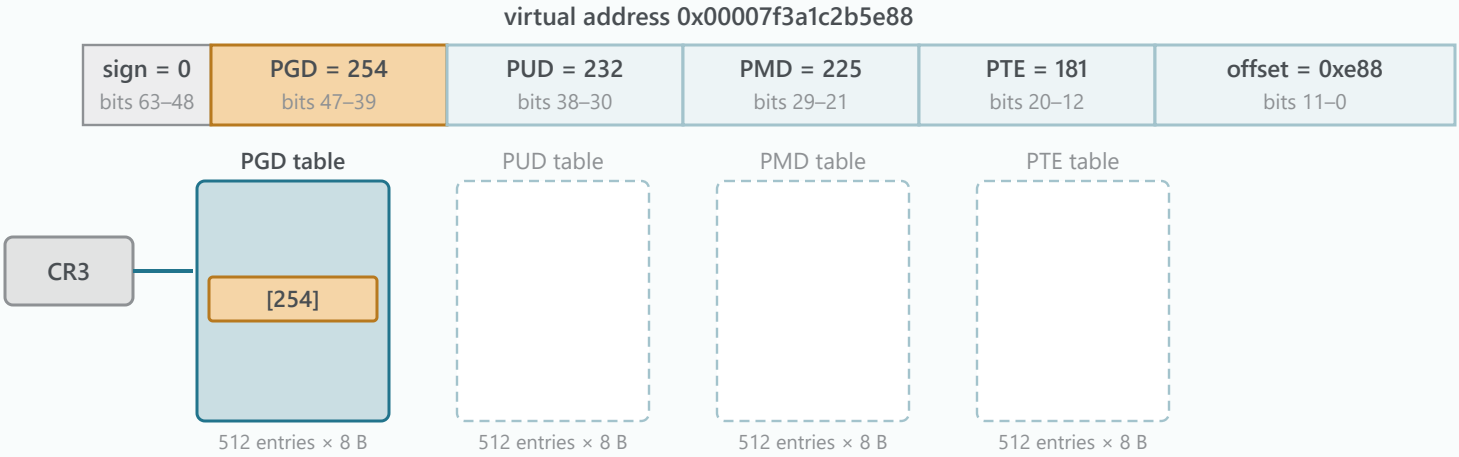
MAP_SHARED: writes are visible in the file and to other processes.
MAP_PRIVATE: copy on write; the file does not change. glibc `malloc` serves small requests from the heap and requests ≥ 128 KiB directly with `mmap`.

Page-Table Walk: 4 and 5 Levels



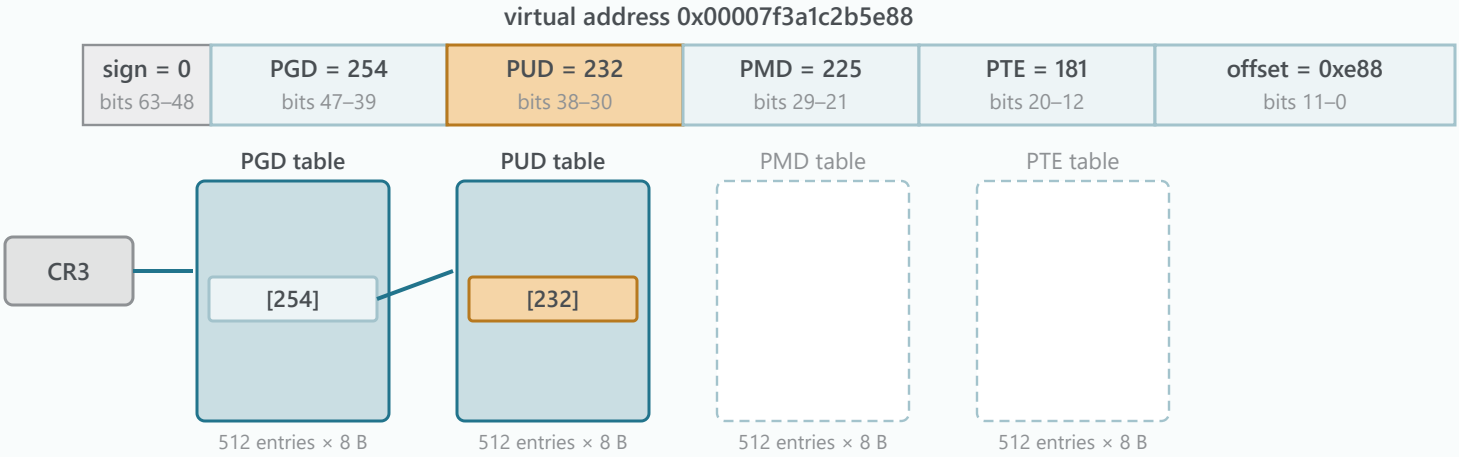
On x86-64 a 4 KiB page → a 12-bit offset. The remaining 36 bits are split across four tables of 512 entries (9 bits) each. Bits 63–48 copy bit 47 (a canonical address).

Page-Table Walk: 4 and 5 Levels



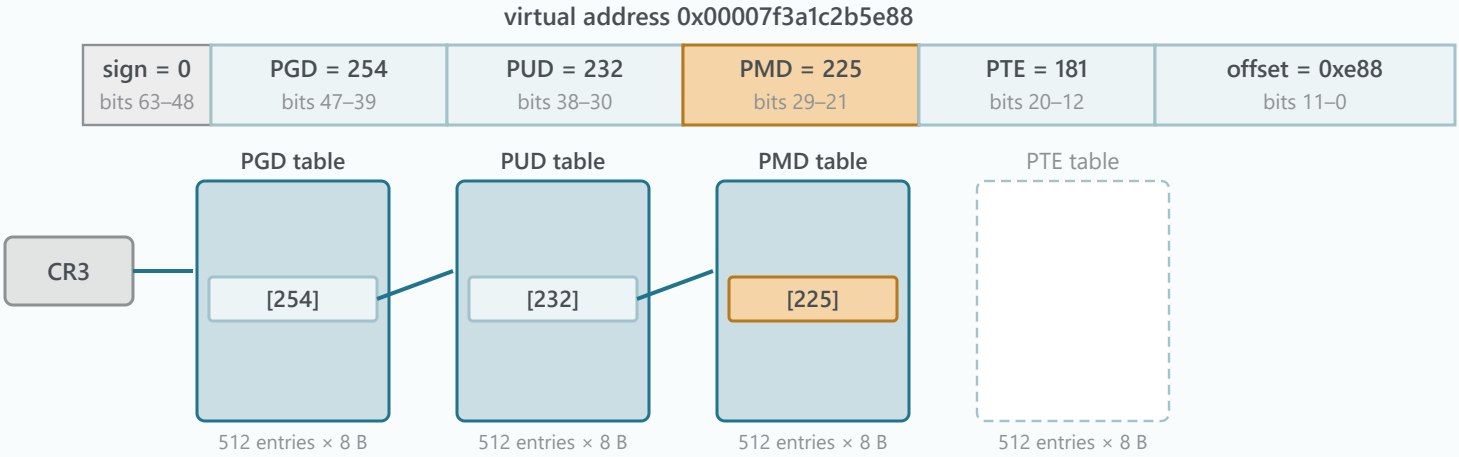
The CR3 register points to the process's top-level table (PGD); CR3 changes on a context switch. Index 254 → the address of the PUD table.

Page-Table Walk: 4 and 5 Levels



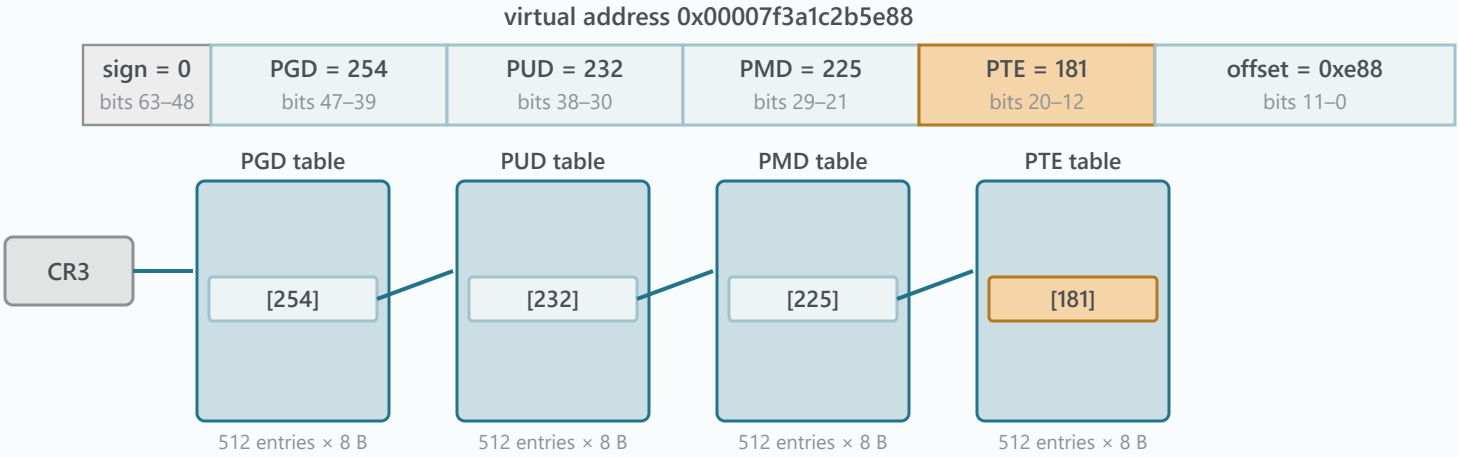
PUD[232] points to a PMD table. (A 1 GiB huge page could also end here.)

Page-Table Walk: 4 and 5 Levels



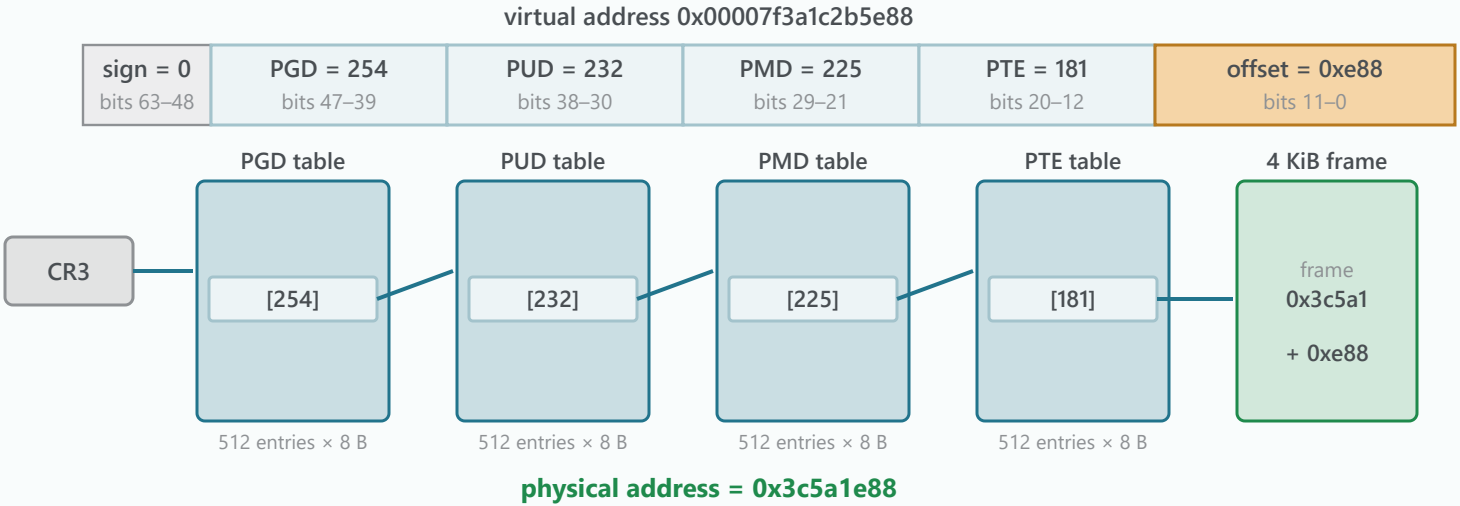
PMD[225] points to a PTE table. (A 2 MiB huge page / THP ends here.)

Page-Table Walk: 4 and 5 Levels



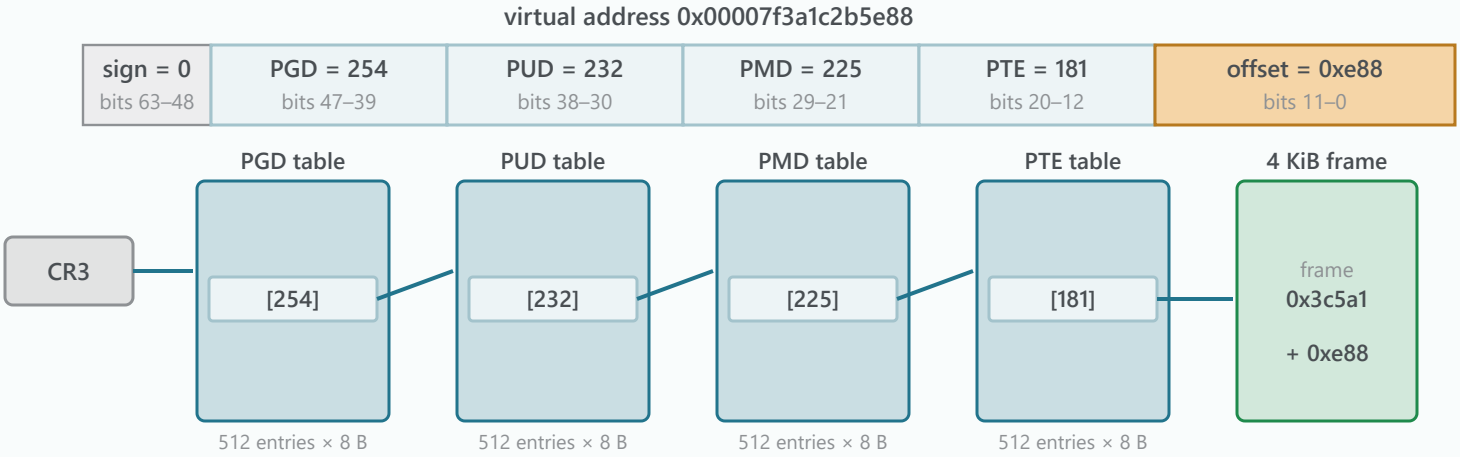
PTE[181]: the present bit, frame number, and R/W, U/S, NX, dirty (D) and accessed (A) bits. Present = 0 means a **page fault**.

Page-Table Walk: 4 and 5 Levels



The frame number is joined with the offset: physical address = frame × 4096 + offset.

Page-Table Walk: 4 and 5 Levels

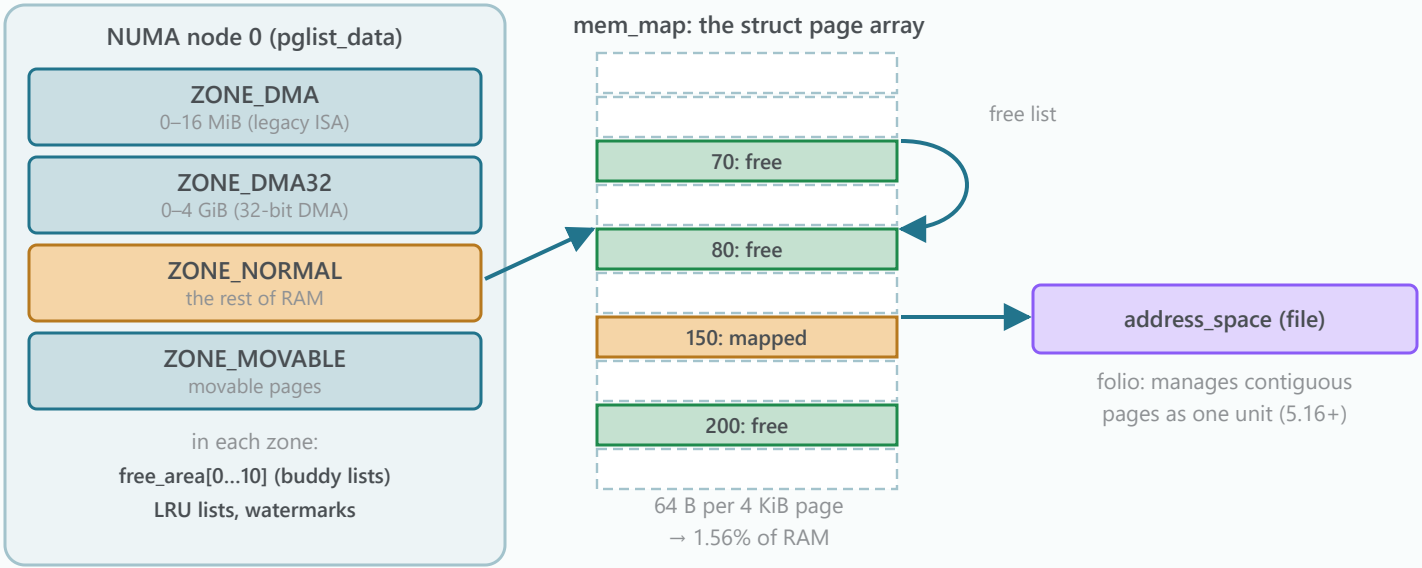


physical address = 0x3c5a1e88

TLB: the next access skips 4 memory reads

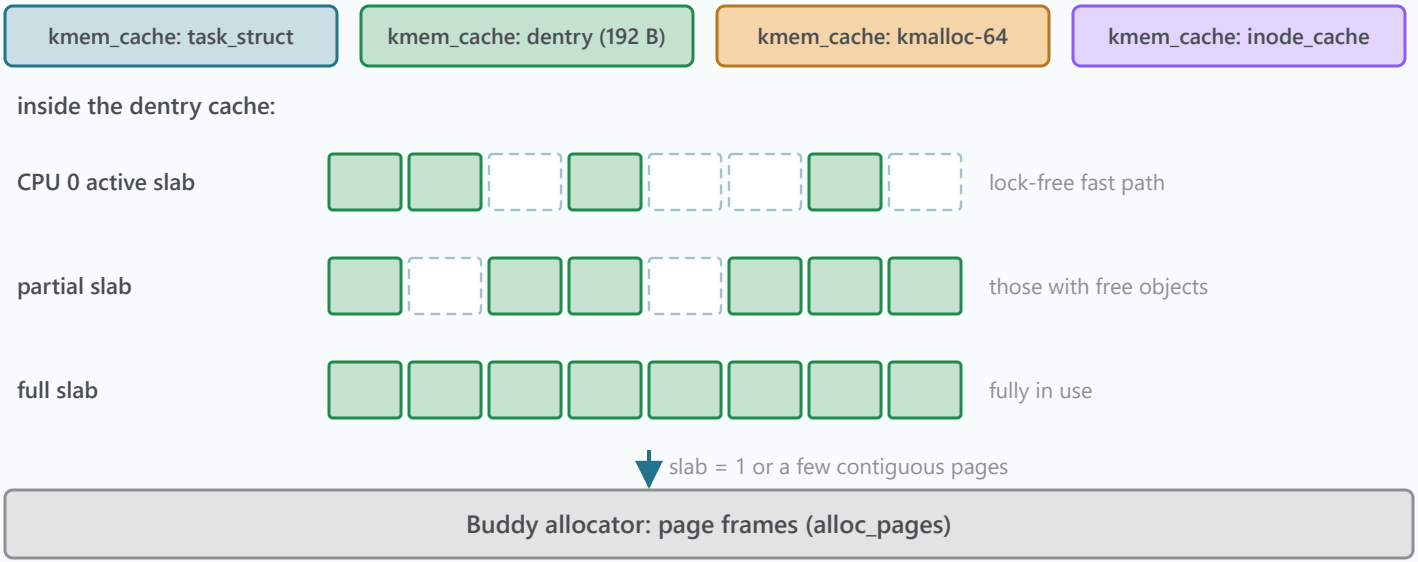
Four extra reads on every access are expensive: the **TLB** caches the translation. Five-level paging (LA57, Linux 4.14+) inserts a **P4D** level for 57-bit addresses. Linux's generic model has 5 levels; when the hardware uses fewer, levels are folded.

Representing Physical Memory



Linux describes physical memory as a **node** → **zone** → **page** hierarchy. On a multisocket machine each CPU prefers to allocate from its own node (NUMA). 64-bit systems have no ZONE_HIGHMEM; it survives only in 32-bit kernels. Each zone's `free_area` array holds the buddy allocator's free lists for orders 0–10 (4 KiB–4 MiB); see `/proc/buddyinfo`.

The SLUB Object Allocator



The kernel allocates many small objects of the same size (task_struct, dentry, inode); allocating a page for each would be wasteful. **SLUB** keeps object caches; `kmalloc` uses size classes from 8 B to 8 KiB. SLOB (6.4) and SLAB (6.8) were removed: SLUB is the only allocator. For large areas that need not be physically contiguous, `vmalloc`.

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
2      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
5      c->freelist = *(void **)obj;   /* next */
6      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
▶  /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → +0 → +1024 → +2048 → +3072 → NULL

A new slab: a 4 KiB page cut into four 1 KiB objects. Free objects form a list; the “next” pointer is stored not in a separate structure but inside the free object itself.

Step 1 / 7

VARIABLES

`freelist = +0`

OUTPUT

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
▶      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
▶      c->freelist = *(void **)obj;   /* next */
▶      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → +1024 → +2048 → +3072 → NULL

slab_alloc: the head of the list (+0) is taken, and freelist advances to the next pointer stored in it (+1024). No lock, no page allocation: a few instructions.

Step 2 / 7

VARIABLES

`obj = +0` `freelist = +1024`

OUTPUT

```
a = +0
```

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
▶      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
▶      c->freelist = *(void **)obj;   /* next */
▶      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → +2048 → +3072 → NULL

b gets +1024; freelist points to +2048.

Step 3 / 7

VARIABLES

`obj = +1024` `freelist = +2048`

OUTPUT

```
a = +0   b = +1024
```

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
2      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
5      c->freelist = *(void **)obj;   /* next */
6      return obj;
7  }
▶  void slab_free(kmem_cache *c, void *obj) {
▶      *(void **)obj = c->freelist;   /* inside the object */
▶      c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → +0 → +2048 → +3072 → NULL

a is freed: the old head (+2048) is written into the object's first 8 bytes and freelist becomes +0. The list works like a stack (last in, first out).

Step 4 / 7

VARIABLES

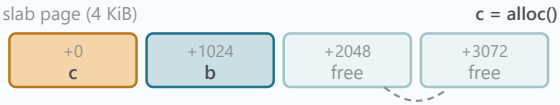
`obj = +0` `freelist = +0`

OUTPUT

```
a = +0    b = +1024
```

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
▶      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
▶      c->freelist = *(void **)obj;   /* next */
▶      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → +2048 → +3072 → NULL

c takes the +0 that a just released: most likely still in the CPU cache, so a “hot” object.

Step 5 / 7

VARIABLES

`obj = +0` `freelist = +2048`

OUTPUT

```
a = +0   b = +1024   c = +0
```

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
▶      void *obj = c->freelist;      /* head of list */
3      if (!obj)
4          obj = new_slab(c);        /* slow path */
▶      c->freelist = *(void **)obj;   /* next */
▶      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



freelist → NULL

d takes +2048 and e takes +3072. The next pointer inside +3072 was NULL: the slab is full and freelist is empty.

Step 6 / 7

VARIABLES

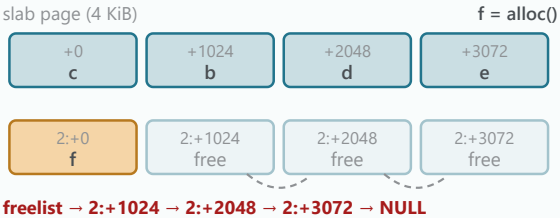
`obj = +3072` `freelist = NULL`

OUTPUT

```
a = +0    b = +1024    c = +0
d = +2048    e = +3072
```

The SLUB Free List: Code Trace

```
1  void *slab_alloc(kmem_cache *c) {
▶      void *obj = c->freelist;      /* head of list */
▶      if (!obj)
▶          obj = new_slab(c);        /* slow path */
▶      c->freelist = *(void **)obj;   /* next */
▶      return obj;
7  }
8  void slab_free(kmem_cache *c, void *obj) {
9      *(void **)obj = c->freelist;   /* inside the object */
10     c->freelist = obj;
11 }
12 /* a,b = alloc; free(a); c,d,e,f = alloc */
```



For f the list is empty: the **slow path**. A new page comes from the page (buddy) allocator, is cut into four objects, and the first goes to f. The common case fast, the rare case correct.

Step 7 / 7

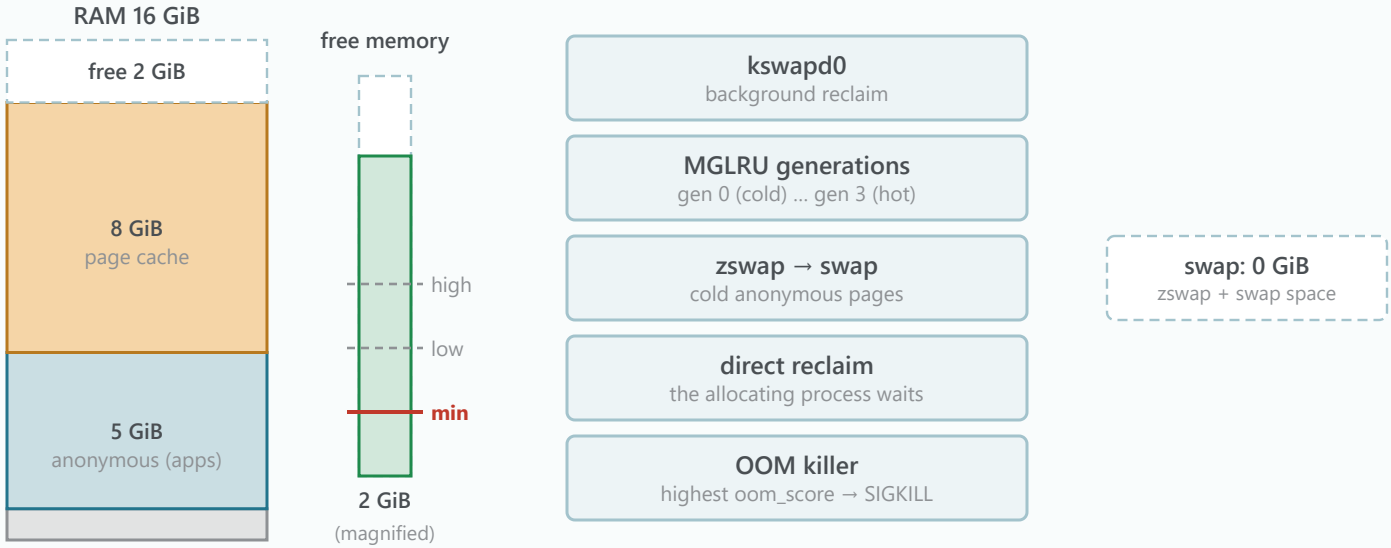
VARIABLES

`obj = 2:+0` `freelist = 2:+1024`

OUTPUT

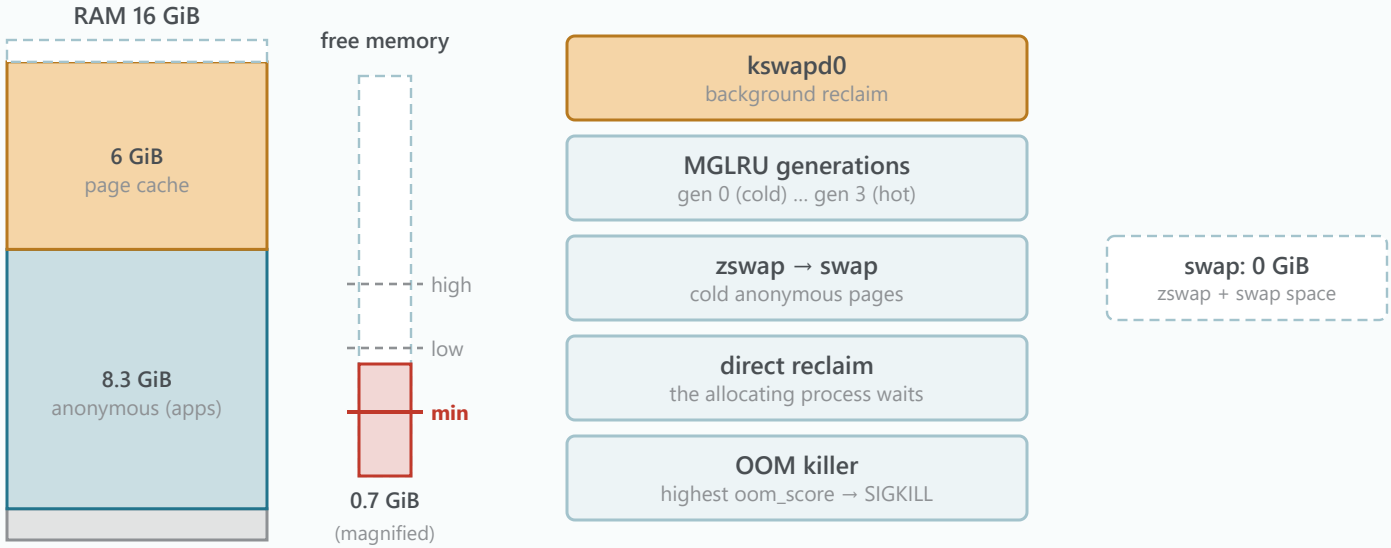
```
a = +0   b = +1024   c = +0
d = +2048   e = +3072   f = 2:+0 (new
slab)
```

Page Cache, Reclaim and OOM



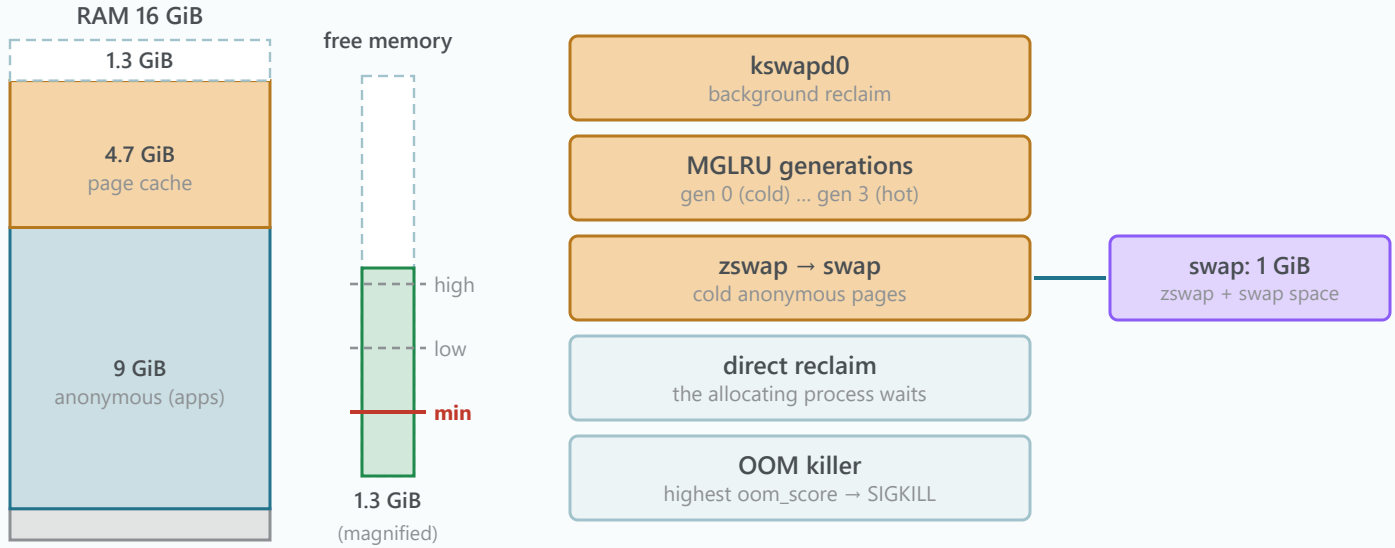
Free memory is wasted memory: Linux uses unused RAM as the **page cache** for file data. “available” in `free -h` also counts the reclaimable part of this cache.

Page Cache, Reclaim and OOM



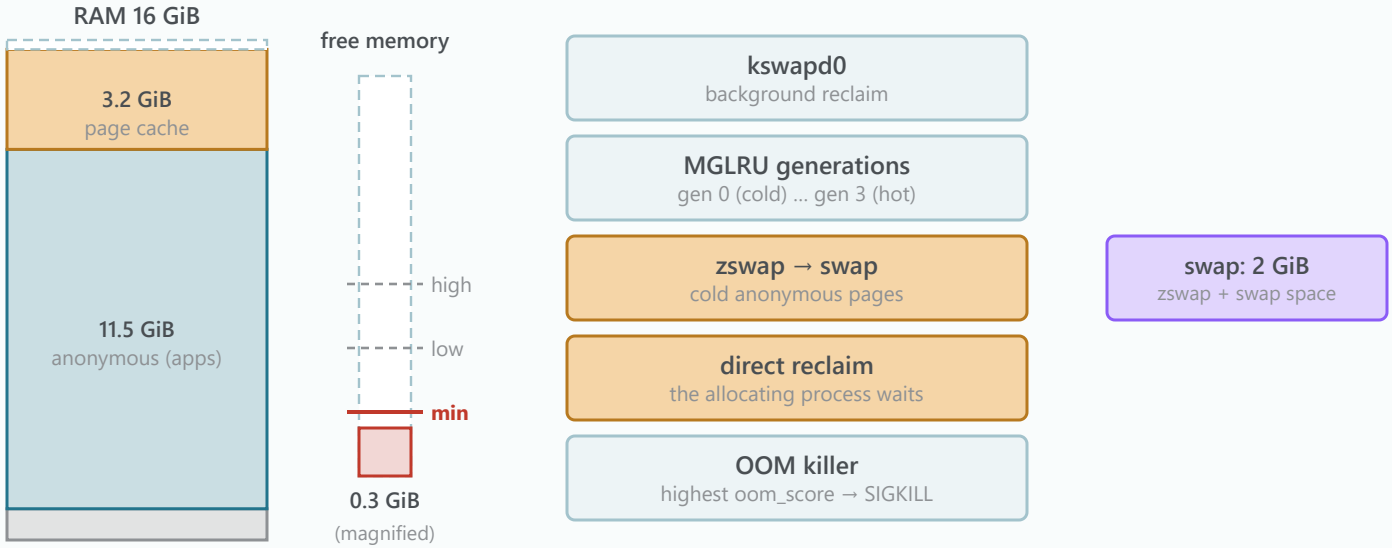
The application grew and free memory dropped below the **low** watermark: `kswapd` wakes and frees clean cache pages in the background (dirty ones are written to disk first).

Page Cache, Reclaim and OOM



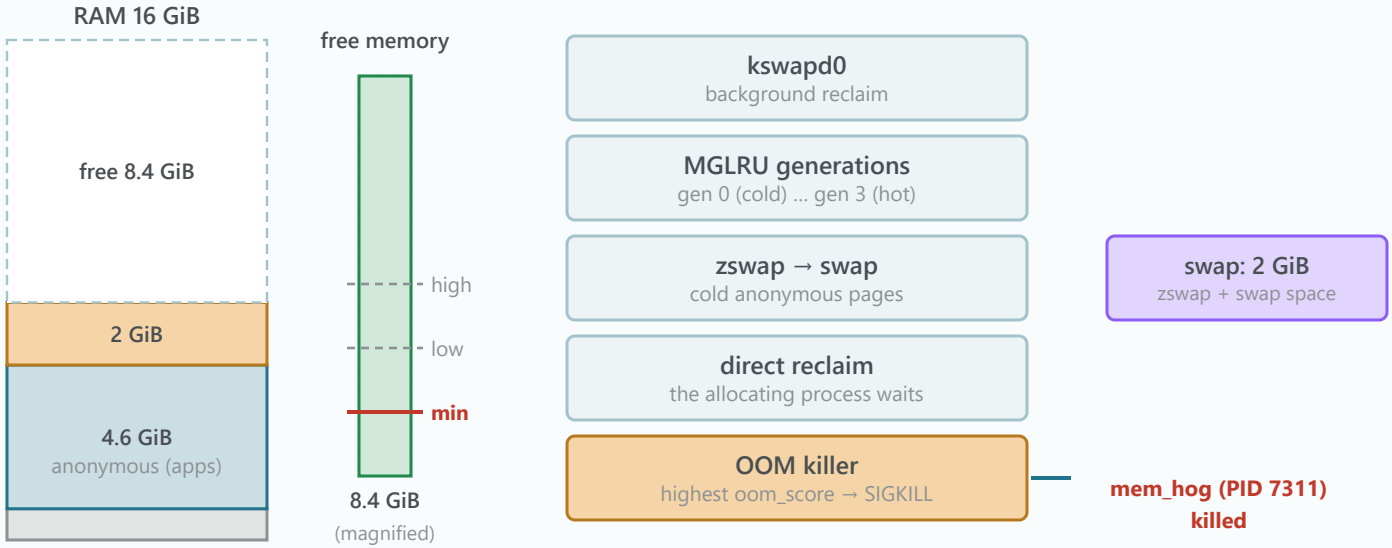
MGLRU (6.1) sorts pages into access generations; the coldest generation goes first. Cold **anonymous** pages have no file: **zswap** compresses them in RAM and, if needed, writes them to swap.

Page Cache, Reclaim and OOM



Allocation outpaces reclaim and free memory is below the **min** watermark: the allocating process does **direct reclaim** itself and stalls (latency rises).

Page Cache, Reclaim and OOM

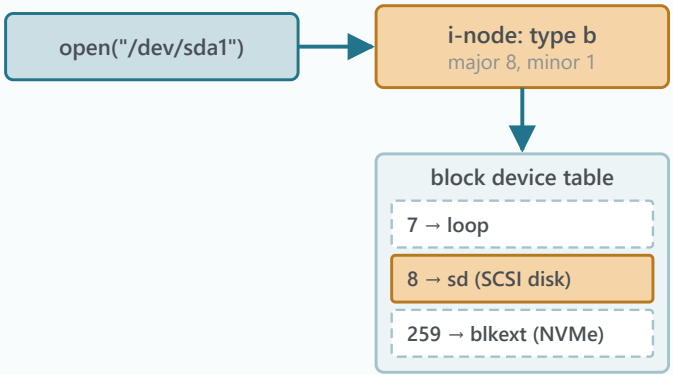


If nothing else works, the **OOM killer** kills the process with the highest `oom_score` (usually the largest) with SIGKILL; `oom_score_adj` can influence the choice. If a cgroup limit is exceeded, the victim is picked from that group only.

Devices Are Files: Major and Minor Numbers

```
$ ls -l /dev     # selected lines
brw-rw---- 1 root disk   8, 0 sda
brw-rw---- 1 root disk   8, 1 sda1
brw-rw---- 1 root disk 259, 0 nvme0n1
crw-rw-rw- 1 root root   1, 3 null
crw-rw-rw- 1 root tty    5, 0 tty
```

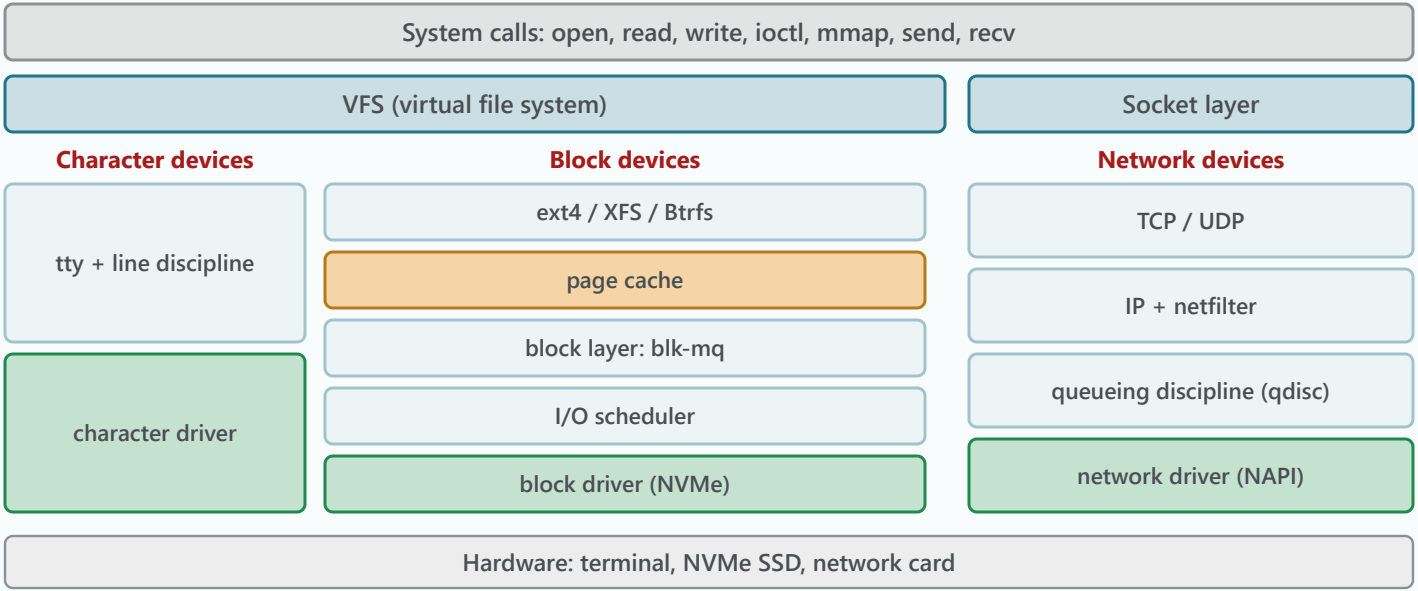
- **c** character device: byte stream, usually uncached (terminals, `/dev/null`, mice)
- **b** block device: fixed-size blocks, random access; uses the page cache and I/O scheduler
- Network interfaces (`eth0`) aren't in `/dev` ; they're used via sockets
- devtmpfs + **udev** create nodes dynamically as devices appear



minor: 0 = whole disk, 1 = partition 1
16 minors per disk: sdb = 16

"major, minor" replaces the size column. The **major** number selects the driver, the **minor** the unit it manages. Applications reach disks and terminals with the same `open/read/write` .

Structure of the Linux I/O System



The three device types go down three separate paths. Classic BSD had a similar split; System V's **STREAMS** module stack never entered the mainline Linux kernel. On the network path data travels in `sk_buff` structures; XDP/eBPF can process packets at the driver level.

The Driver Interface: file_operations and Terminals

```
static const struct file_operations my_fops = {
    .owner          = THIS_MODULE,
    .open           = my_open,
    .release        = my_release,     /* close */
    .read           = my_read,
    .write          = my_write,
    .unlocked_ioctl = my_ioctl,
    .poll           = my_poll,
    .mmap           = my_mmap,
};
cdev_init(&cdev, &my_fops);  /* major:minor → fops */
```

Old UNIX entries	Linux equivalent
`k_open, tty_open, pr_open, ip_open`	`.open`
`k_close, tty_close, pr_close…`	`.release`
`mem_read, k_read, tty_read`	`.read`
`mem_write, tty_write, pr_write…`	`.write`
`k_ioctl, tty_ioctl, pr_ioctl…`	`.unlocked_ioctl`

Terminal I/O (termios): `tcgetattr/tcsetattr` read and write terminal parameters: canonical mode (line editing, ICANON) or raw mode, ECHO. `cfset/cfget[io]speed` set the input/output baud rate; today mostly for serial ports, since terminals are pseudoterminals (pty).

The classic textbook table lists separate entry points per driver (k_*, tty_*, pr_*, ip_*); Linux gathers them into one structure of function pointers. The VFS routes every `read()` to the driver's `.read`.

INPUT/OUTPUT

The Block Layer: blk-mq and I/O Schedulers

The Linux block layer **blk-mq** is multiqueue: each CPU writes to its own software queue, and requests pass through an optional I/O scheduler (`none`, `mq-deadline`, `bfq`, `kyber`) before being spread over the device's hardware queues. The active scheduler is read and changed per device in `/sys/block/<device>/queue/scheduler`.

Networking: Sockets and Connection Setup



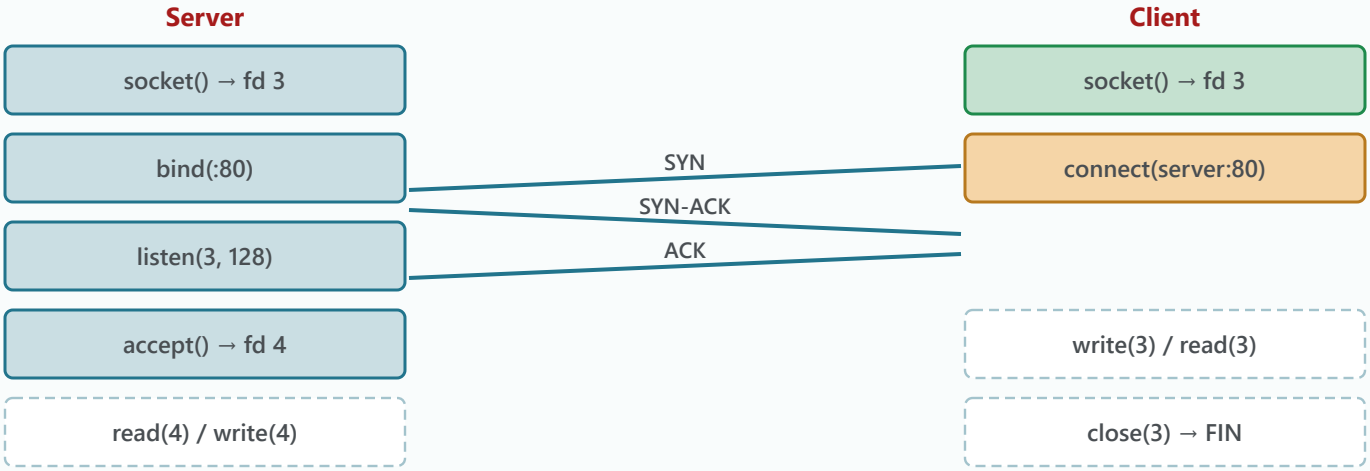
The server creates a socket (fd 3), binds it to port 80 (`bind`) and opens a listen queue with `listen` . A socket is a file descriptor too.

Networking: Sockets and Connection Setup



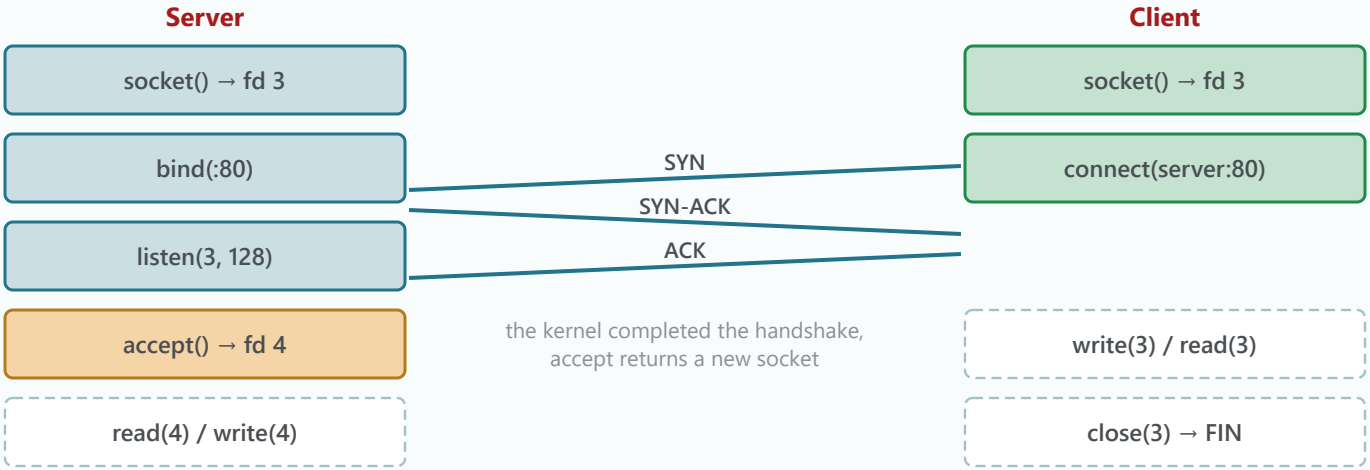
The server blocks in `accept`. The client creates its own socket.

Networking: Sockets and Connection Setup



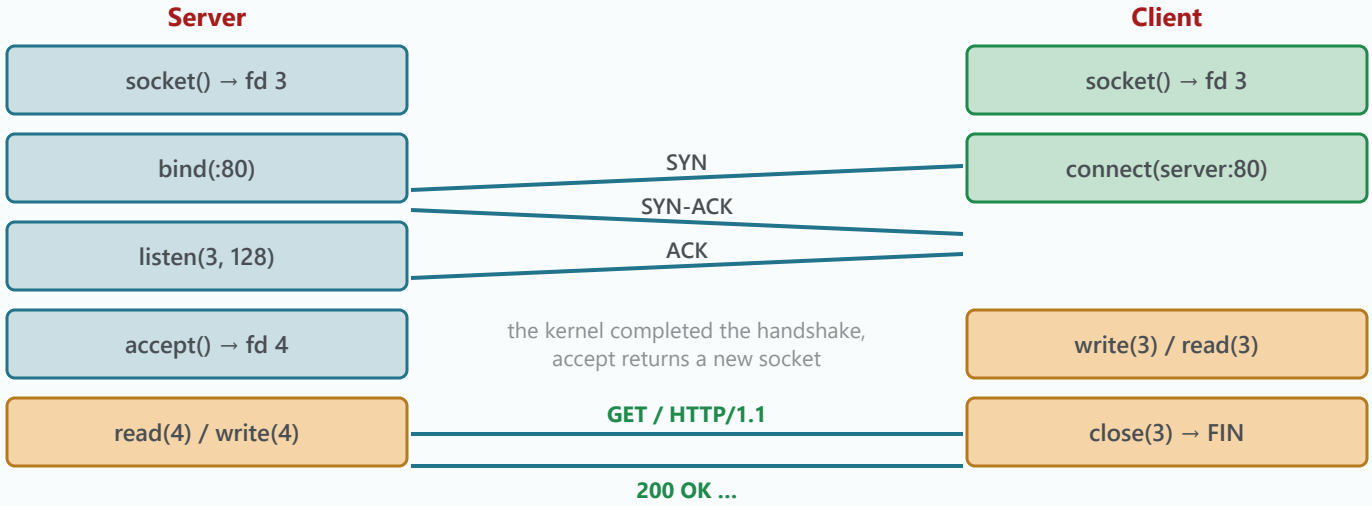
With `connect`, the client kernel performs the **three-way handshake** (SYN, SYN-ACK, ACK); the application never sees these packets.

Networking: Sockets and Connection Setup



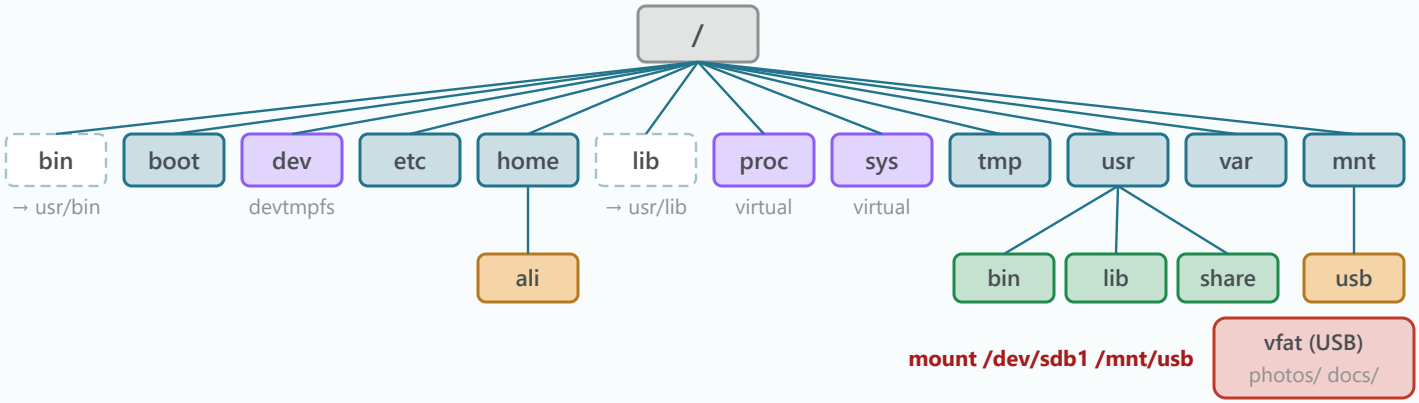
Once connected, `accept` returns a **new** socket (fd 4); fd 3 keeps listening. Servers with many clients manage thousands of sockets with `epoll` or threads.

Networking: Sockets and Connection Setup



Data flows with `write` / `read`: the kernel puts it in `sk_buff`s, adds TCP/IP headers, and the driver writes it to the card's ring. `close` sends a FIN.

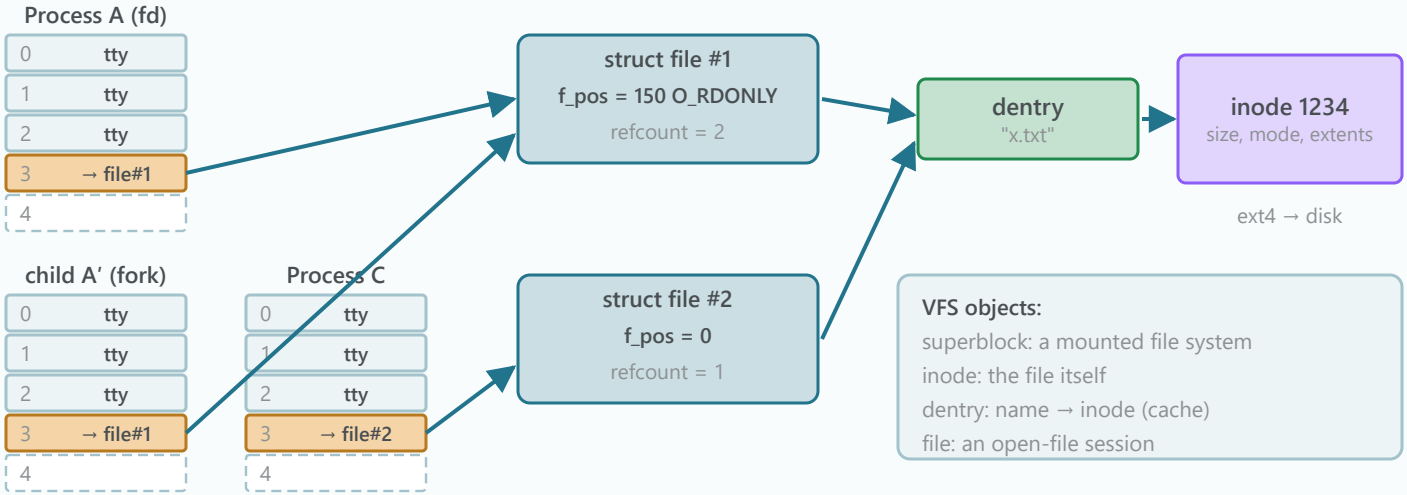
The Directory Tree and Mounting



- `/usr/bin` : commands (`/bin` is now a link to it)
- `/etc` : text configuration files
- `/dev` : device files
- `/usr/lib` : shared libraries, `/usr/lib/modules` : kernel modules
- `/usr` : read-only, shareable software (bin, lib, share)
- `/home` : user directories; `/root` : the administrator's
- `/var` : variable data (logs, spools, caches)
- `/proc` , `/sys` : the kernel's pseudo file systems
- `/boot` : the kernel and initramfs
- `/tmp` : temporary (tmpfs on most distributions)

Instead of Windows-style drive letters there is a single tree: another file system is **mounted** on any directory. The layout follows the FHS standard; most distributions moved /bin and /lib under /usr (“usrmerge”).

The VFS: Descriptors, Open Files and i-nodes



In Linux an open file is a `struct file` (`f_pos`, flags, reference count); the per-process table lives in `task_struct → files_struct`, and after `fork()` both processes point to the same `struct file`. The objects' operation tables (`super_operations`, `inode_operations`, `file_operations`) route each call to ext4's, XFS's, Btrfs's, NFS's or procfs's own function; dentries are kept in the dcache.

File and Directory System Calls, stat

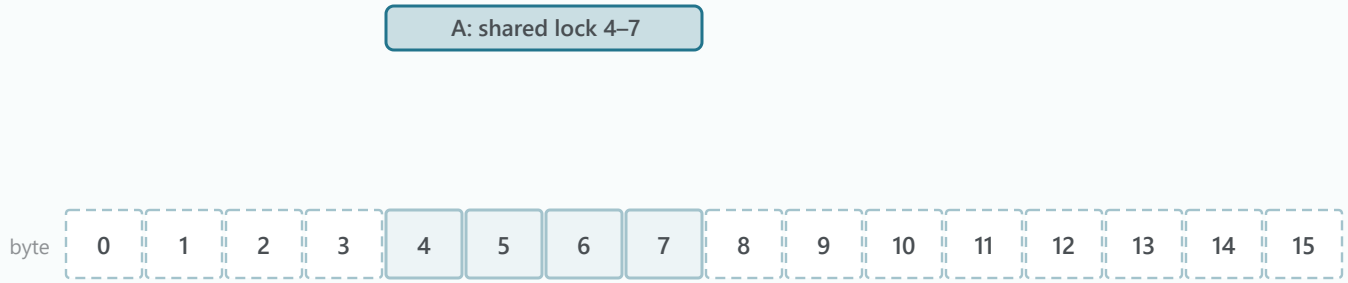
File call	Function
`open` , `creat`	Open / create, return an fd
`close`	Close the descriptor
`read` , `write`	Read / write at the position
`lseek`	Move the position
`stat` , `fstat` , `lstat`	Get information (lstat: the link itself)
`pipe` , `dup2`	Create a pipe / duplicate an fd
`fcntl`	Locks, flags, etc.
`mkdir` , `rmdir`	Create / remove a directory (if empty)
`link` , `unlink`	Add / remove a name
`rename` , `chdir`	Move (atomically) / change cwd
`getdents64`	Read directory entries

stat field	Meaning
`st_dev`	Device the file resides on
`st_ino`	i-node number
`st_mode`	Type + permission bits
`st_nlink`	Number of hard links
`st_uid` , `st_gid`	Owner and group
`st_size`	Size in bytes
`st_atime`	Last access time
`st_mtime`	Last content change
`st_ctime`	Last i-node change (not creation!)
`st_blocks`	Number of 512 B blocks allocated



`opendir/readdir/closedir` are not system calls but library functions built on `getdents64` . The creation time is available only through the newer `statx()` (stx_btime).

File Locking



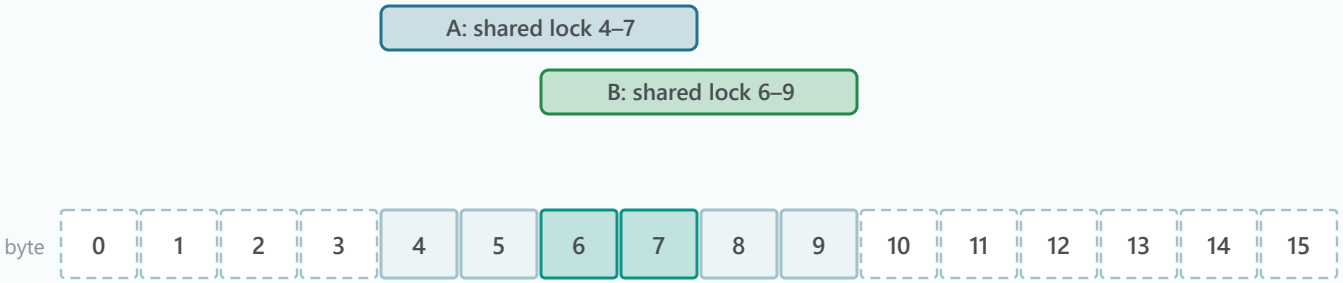
shared (read) locks may overlap; an exclusive (write) lock may overlap no lock

Process A puts a **shared** (read) lock on bytes 4–7: `fcntl(fd, F_SETLK, &fl)` with `l_type = F_RDLCK`, `l_start = 4`, `l_len = 4`.

Step 1 / 4

These locks are **advisory**: a process that never asks for the lock can still write. For whole files there is `flock()`; for thread-safe use, OFD locks bound to the open file.

File Locking

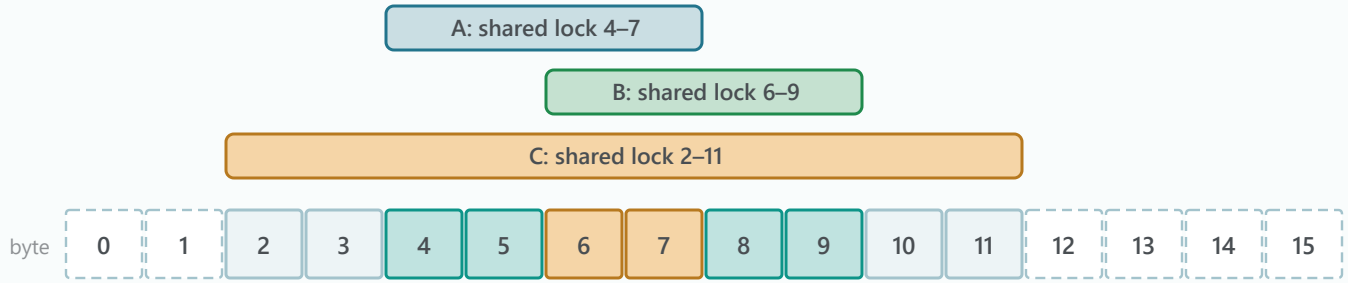


shared (read) locks may overlap; an exclusive (write) lock may overlap no lock

B asks for a shared lock on 6–9: shared locks may overlap freely, so the request is granted.

These locks are **advisory**: a process that never asks for the lock can still write. For whole files there is `flock()`; for thread-safe use, OFD locks bound to the open file.

File Locking

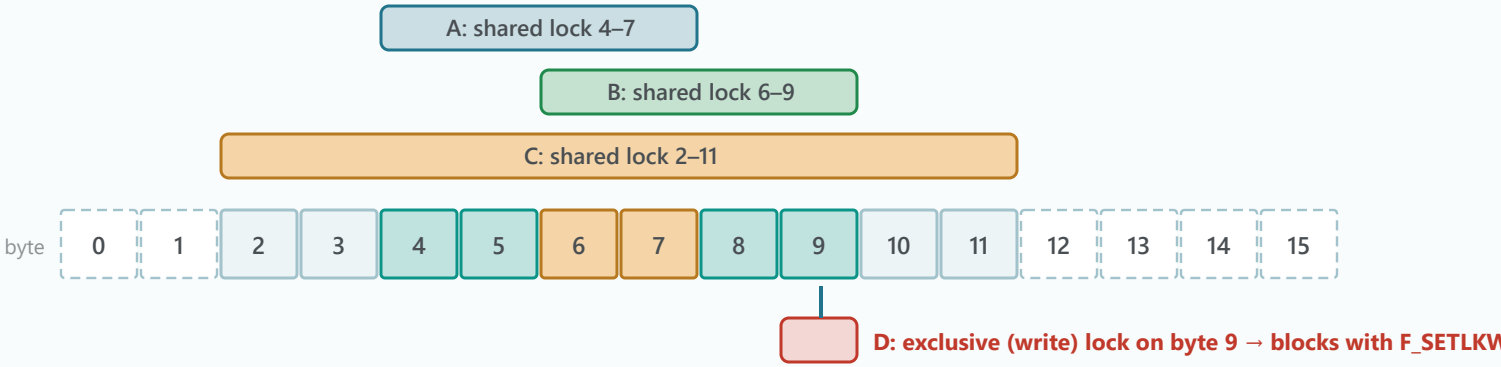


shared (read) locks may overlap; an exclusive (write) lock may overlap no lock

C puts a shared lock on 2–11; bytes 6–7 are now covered by three locks.

These locks are **advisory**: a process that never asks for the lock can still write. For whole files there is `flock()`; for thread-safe use, OFD locks bound to the open file.

File Locking



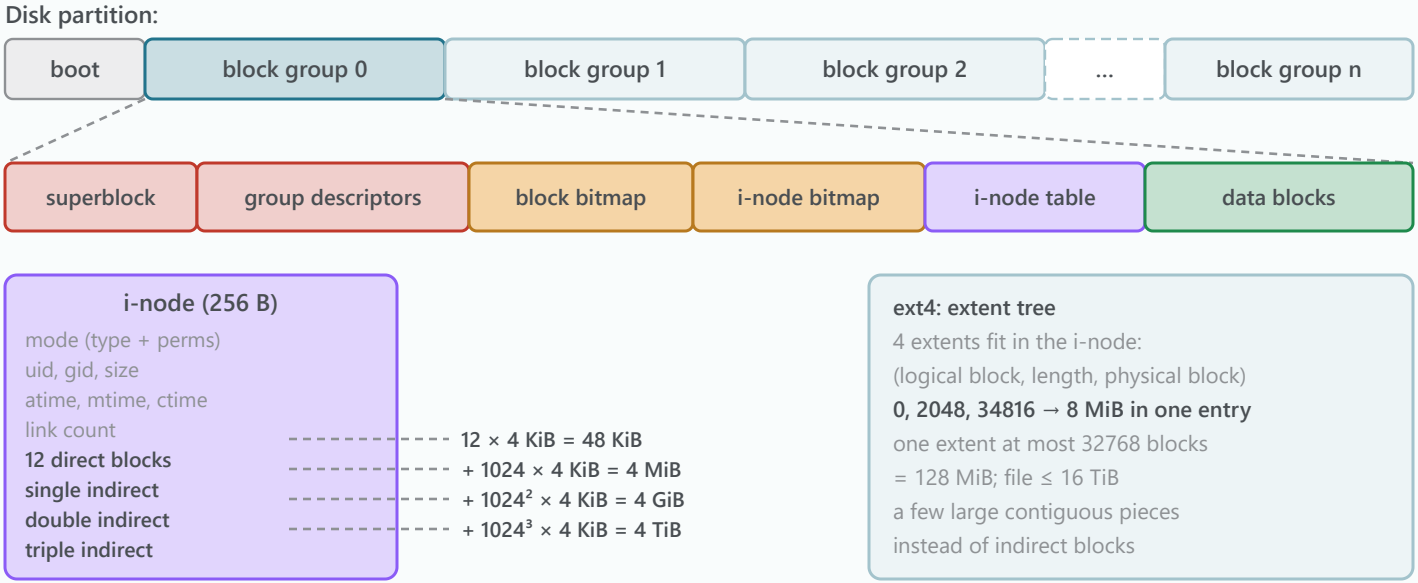
shared (read) locks may overlap; an exclusive (write) lock may overlap no lock

D asks for an **exclusive** (F_WRLCK) lock on byte 9: it conflicts with B's and C's locks. `F_SETLK` fails at once; `F_SETLKW` waits until the locks go away. If the kernel detects a wait cycle it returns `EDEADLK`.

Step 4 / 4

These locks are **advisory**: a process that never asks for the lock can still write. For whole files there is `flock()`; for thread-safe use, OFD locks bound to the open file.

ext2/ext4 Disk Layout and the i-node



ext2 split the classic UNIX layout (boot block, superblock, i-nodes, data) into **block groups**: an i-node and its data are kept close in the same group, and the superblock is backed up. With indirect blocks the theoretical limit is ~4 TiB (2 TiB in ext2 because of the i_blocks field). The root directory's i-node number is always 2.

ext4 Journaling



One `write()` + `fsync()` changes several blocks: the data (D) plus **metadata** such as the i-node, the block bitmap and the extent. A crash after writing one but not another leaves the file system inconsistent.

Step 1 / 5

Other modes: **data=journal** (data goes to the journal too; safest, slowest) and **data=writeback** (no ordering guarantee). XFS also journals metadata; Btrfs stays consistent with copy on write instead of a journal.

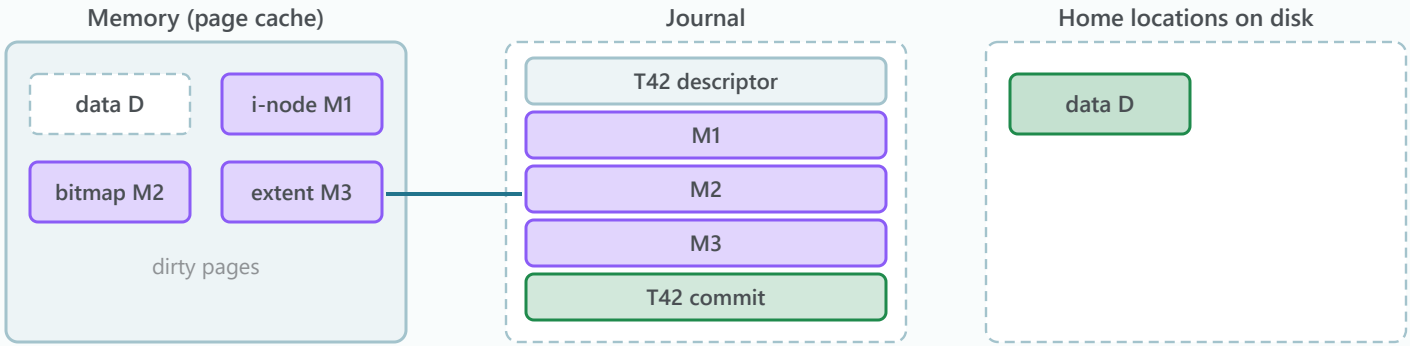
ext4 Journaling



In the default **data=ordered** mode the data blocks are written to their home locations first, so metadata never points at a block of garbage.

Other modes: **data=journal** (data goes to the journal too; safest, slowest) and **data=writeback** (no ordering guarantee). XFS also journals metadata; Btrfs stays consistent with copy on write instead of a journal.

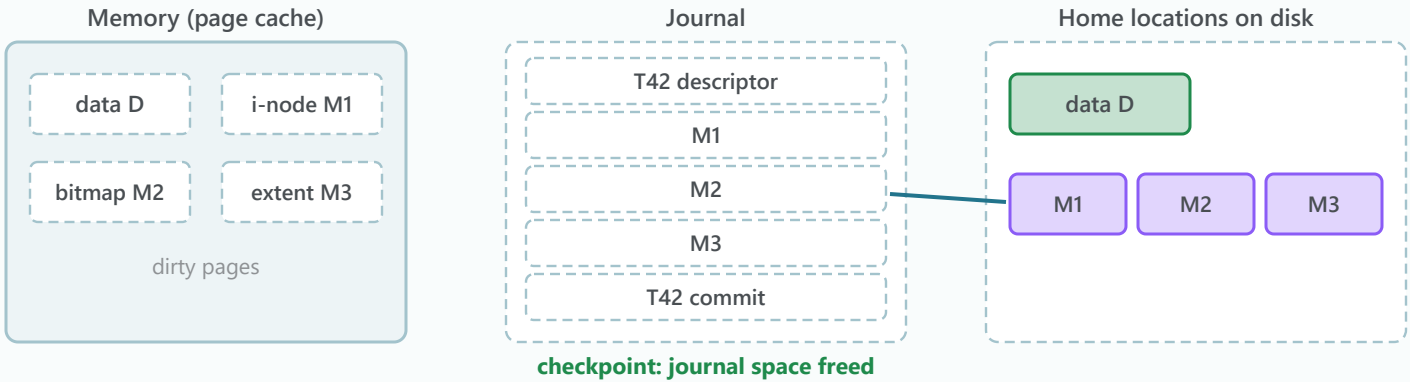
ext4 Journaling



The metadata blocks are written to the **journal** as one transaction (T42), ending with a **commit** record. Once the commit reaches the disk, the transaction is durable.

Other modes: **data=journal** (data goes to the journal too; safest, slowest) and **data=writeback** (no ordering guarantee). XFS also journals metadata; Btrfs stays consistent with copy on write instead of a journal.

ext4 Journaling



Checkpoint: the metadata is written to its home locations and the journal space is reused (a circular buffer).

Other modes: **data=journal** (data goes to the journal too; safest, slowest) and **data=writeback** (no ordering guarantee). XFS also journals metadata; Btrfs stays consistent with copy on write instead of a journal.

ext4 Journaling



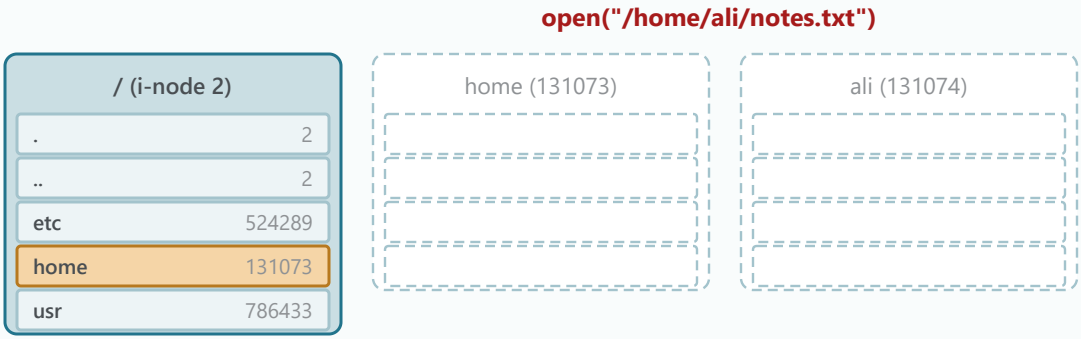
crash after commit → T42 is replayed at mount

crash before commit → T42 is ignored; the old consistent state

After a crash, instead of a full `fsck` scan the journal is read: committed transactions are replayed, incomplete ones discarded. Consistency in seconds.

Other modes: **data=journal** (data goes to the journal too; safest, slowest) and **data=writeback** (no ordering guarantee). XFS also journals metadata; Btrfs stays consistent with copy on write instead of a journal.

Directories and Path Resolution



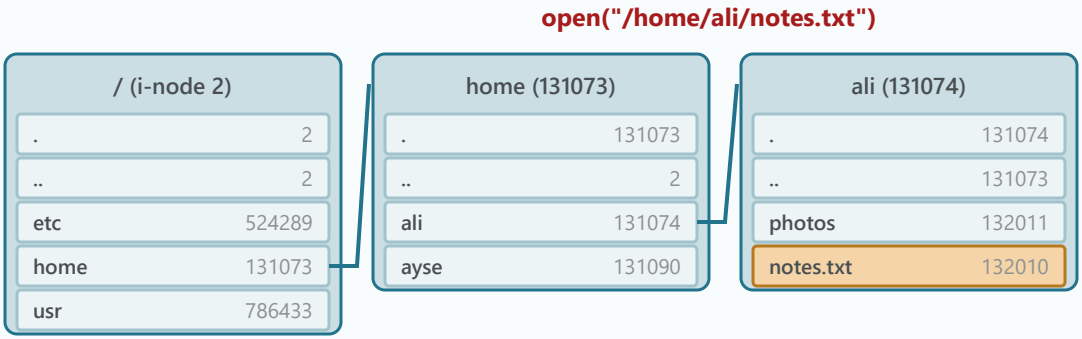
An absolute path starts at the root: the root directory's i-node (2) is read and its data blocks are searched for "home" → 131073. The **x** (search) permission is checked for every directory.

Directories and Path Resolution



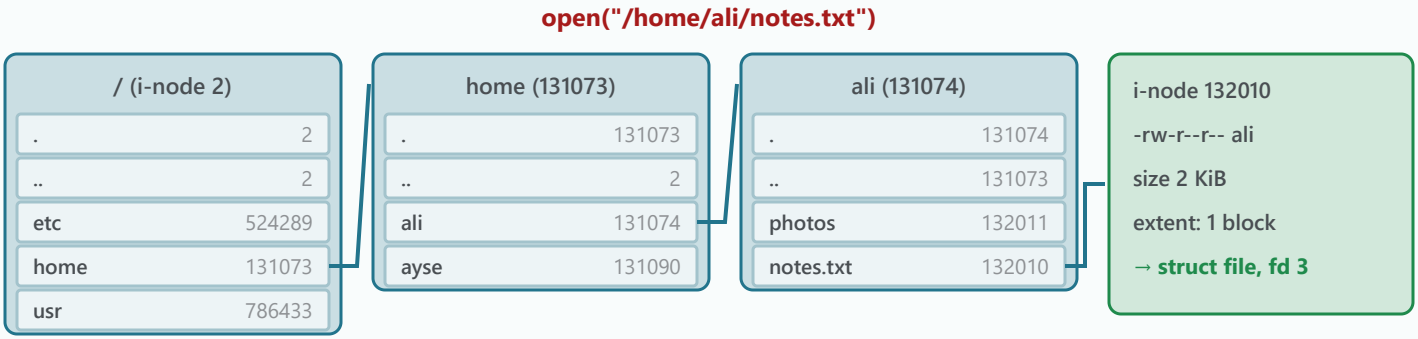
"ali" is looked up in the data of the "home" directory → 131074.

Directories and Path Resolution



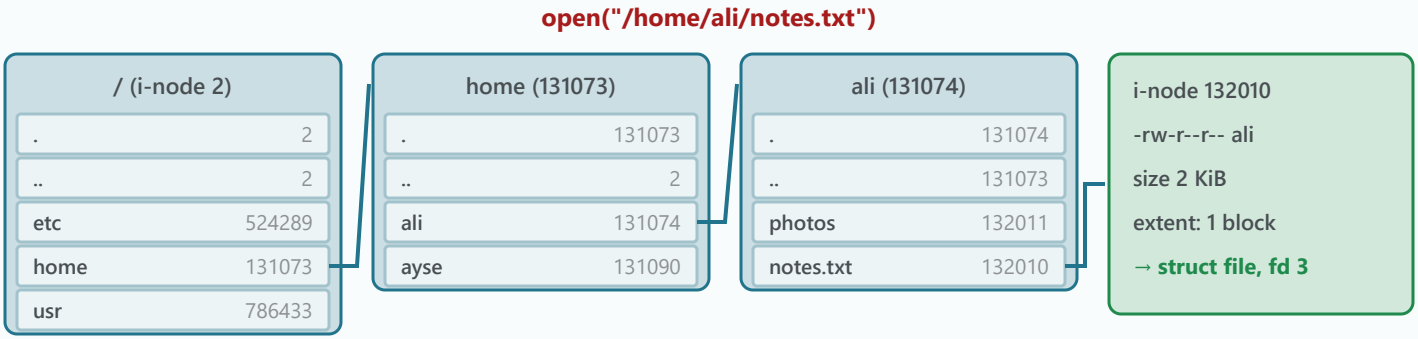
"notes.txt" in the "ali" directory → 132010. (A relative path would start at the process's cwd.)

Directories and Path Resolution



The last component's i-node is read, permissions are checked, and a struct file is created and an fd returned.

Directories and Path Resolution

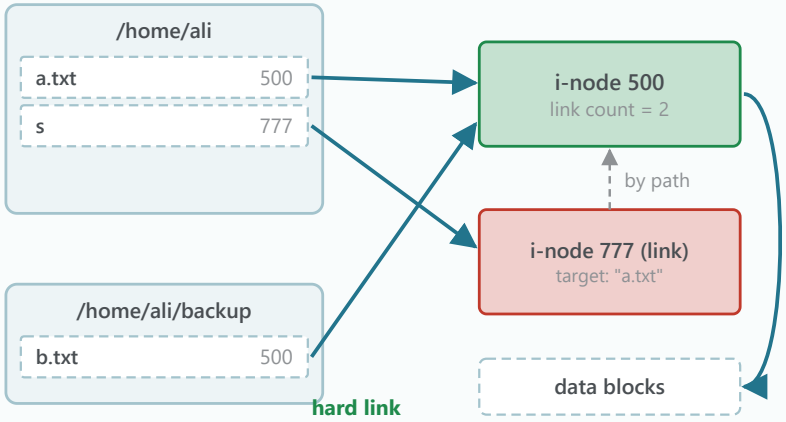


directory entry (ext2/4): i-node no · record length · name length · type · name

a deleted entry is merged into the previous one's record length; large directories use an htree (hash tree)

If every step needed a disk read it would be slow: the **dentry cache** (dcache) keeps name → i-node mappings in memory, so hot paths resolve without touching the disk.

Hard and Symbolic Links

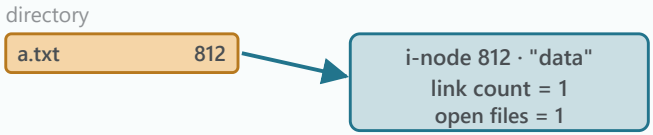



```
$ ln a.txt backup/b.txt    # hard
$ ln -s a.txt s            # symbolic
$ ls -li
500 -rw-r--r-- 2 ali ali 9 a.txt
777 lrwxrwxrwx 1 ali ali 5 s -> a.txt
```

- Hard link: **the same i-node**, and the link count goes up. `rm a.txt` removes only the name; the data is freed when the count reaches 0 (and no open fd remains).
- A hard link cannot cross file-system boundaries or point to a directory.
- Symbolic link: holds the target's **path**; it can point to another file system or a directory. If the target is deleted, the link **dangles**.

Link Count and unlink: Code Trace

```
▶ int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
▶ write(fd, "data", 4);
3  link("a.txt", "b.txt");      /* hard link */
4  symlink("a.txt", "s.txt");   /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



a.txt created, 4 bytes written

open allocates a new i-node (812) and adds the entry “a.txt → 812” to the directory: link count 1. fd 3 is open: open-file count 1.

Step 1 / 9

VARIABLES

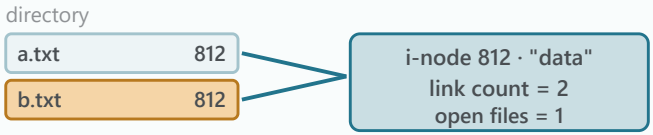
`fd = 3` `links = 1` `open = 1`

OUTPUT

```
fd = 3
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
▶ link("a.txt", "b.txt");      /* hard link */
4  symlink("a.txt", "s.txt");   /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



b.txt: a second name for the same i-node

link only writes a new directory entry: b.txt → **812**. No data is copied; the link count is 2. a.txt and b.txt are equals; neither is the “original”.

Step 2 / 9

VARIABLES

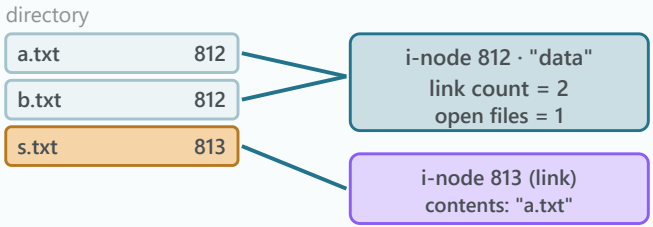
`fd = 3` `links = 2` `open = 1`

OUTPUT

```
fd = 3
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");      /* hard link */
▶  symlink("a.txt", "s.txt");   /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



s.txt: a separate file holding the path

symlink creates a separate i-node (813) whose contents are just the path “a.txt”. 812's link count does not change.

Step 3 / 9

VARIABLES

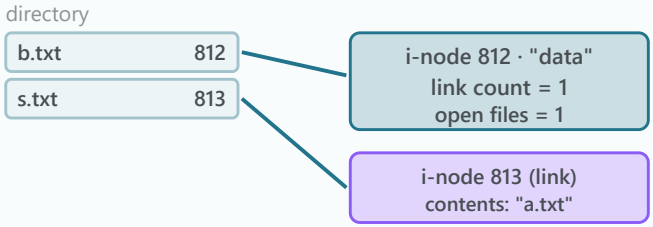
`fd = 3` `links = 2` `open = 1`

OUTPUT

```
fd = 3
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");       /* hard link */
4  symlink("a.txt", "s.txt");    /* symbolic */
▶  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



name a.txt removed, data remains

unlink("a.txt") removes the name and drops the link count to 1. The data is still reachable through b.txt.

Step 4 / 9

VARIABLES

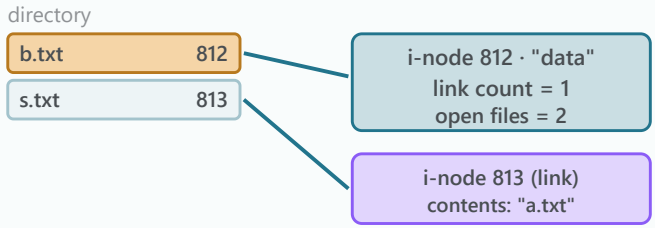
`fd = 3` `links = 1` `open = 1`

OUTPUT

```
fd = 3
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");       /* hard link */
4  symlink("a.txt", "s.txt");    /* symbolic */
5  unlink("a.txt");
▶  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



b.txt opened

b.txt opens fine: r1 = **4**, open-file count 2.

Step 5 / 9

VARIABLES

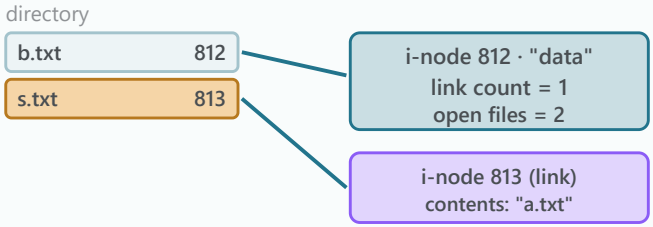
`r1 = 4` `links = 1` `open = 2`

OUTPUT

```
fd = 3
r1 = 4
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");       /* hard link */
4  symlink("a.txt", "s.txt");    /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
▶  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
9  close(r1);
10 close(fd);
```



s.txt → "a.txt" → missing

Opening s.txt, the kernel follows the path inside it (“a.txt”); that name is gone. The symbolic link **dangles**: r2 = −1, ENOENT.

Step 6 / 9

VARIABLES

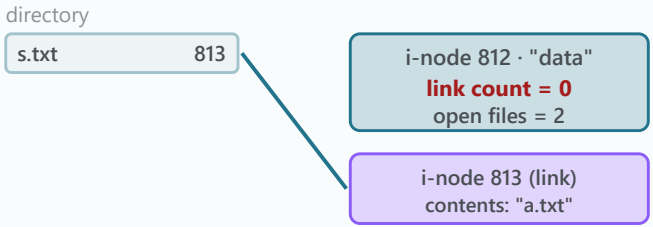
`r2 = -1` `errno = ENOENT`

OUTPUT

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");       /* hard link */
4  symlink("a.txt", "s.txt");    /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
▶  unlink("b.txt");
9  close(r1);
10 close(fd);
```



no names left, but two fds open

The last name is gone too: link count **0**. But two descriptors are open, so the i-node and data are not freed yet; ls no longer shows the file.

Step 7 / 9

VARIABLES

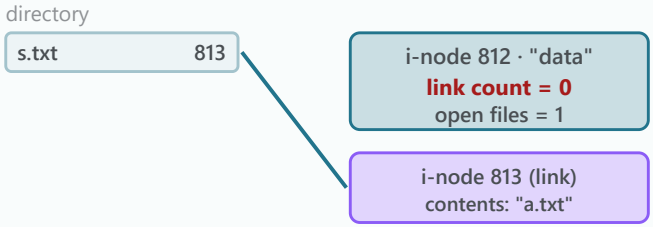
`links = 0` `open = 2`

OUTPUT

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Link Count and unlink: Code Trace

```
1  int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2  write(fd, "data", 4);
3  link("a.txt", "b.txt");      /* hard link */
4  symlink("a.txt", "s.txt");   /* symbolic */
5  unlink("a.txt");
6  int r1 = open("b.txt", O_RDONLY);
7  int r2 = open("s.txt", O_RDONLY);
8  unlink("b.txt");
▶  close(r1);
10 close(fd);
```



r1 closed

close(r1): open-file count 1. Reads and writes through fd 3 still work.

Step 8 / 9

VARIABLES

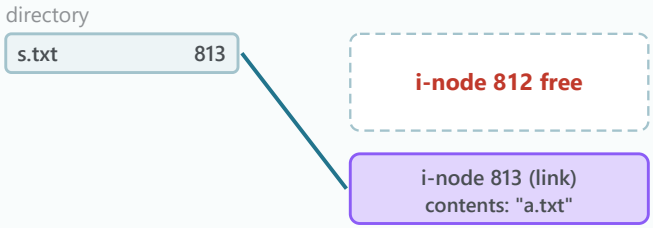
`links = 0` `open = 1`

OUTPUT

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Link Count and unlink: Code Trace

```
1 int fd = open("a.txt", O_CREAT|O_RDWR, 0644);
2 write(fd, "data", 4);
3 link("a.txt", "b.txt");       /* hard link */
4 symlink("a.txt", "s.txt");    /* symbolic */
5 unlink("a.txt");
6 int r1 = open("b.txt", O_RDONLY);
7 int r2 = open("s.txt", O_RDONLY);
8 unlink("b.txt");
9 close(r1);
▶ close(fd);
```



links 0 and open 0: blocks freed

The last descriptor is closed: both counters are 0, so the i-node and data blocks are freed. The “open, then unlink at once” idiom for temporary files relies on this.

Step 9 / 9

VARIABLES

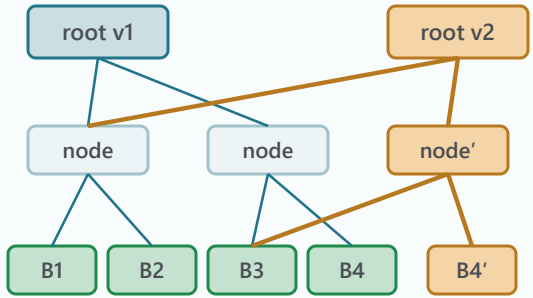
`links = 0` `open = 0`

OUTPUT

```
fd = 3
r1 = 4
r2 = -1 (ENOENT)
```

Beyond ext4: XFS, Btrfs and Others

File system	Key feature	Typical use
ext4	Journaling, extents, mature with a fast fsck	Default on most distributions
XFS	Parallel I/O, large files, delayed allocation	RHEL default, large servers
Btrfs	Copy on write, snapshots, checksums, compression, subvolumes	Fedora, openSUSE default
F2FS	Flash-friendly, log-structured	Android phones
tmpfs	Entirely in RAM (page cache + swap)	/tmp, /run, /dev/shm
NFS, SMB	Network file systems	Shared home directories



when B4 changes only the path is copied;
root v1 = a snapshot

Thanks to the VFS they all work with the same system calls; `findmnt` or `df -T` shows which file system is mounted where.

/proc and /sys: Windows into the Kernel

```
$ cat /proc/loadavg
0.42 0.37 0.30 2/1234 98765
$ grep -E "MemTotal|MemAvailable" /proc/meminfo
MemTotal:       16303452 kB
MemAvailable:    9876544 kB
$ ls /proc/self
cmdline  cwd  environ  exe  fd  maps  status  …
$ cat /sys/block/nvme0n1/queue/scheduler
[none] mq-deadline kyber bfq
$ sysctl vm.swappiness        # = /proc/sys/vm/swappiness
vm.swappiness = 60
```

/proc

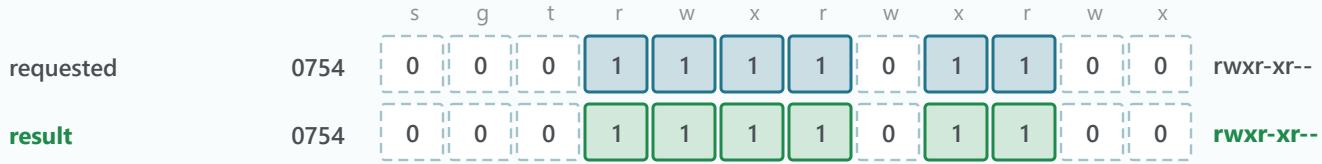
Per-process directories (`/proc/PID/status`, `maps`, `fd/`) and kernel statistics. `ps`, `top` and `free` read their data here. `/proc/sys`: tunable kernel parameters.

/sys (sysfs)

The device model: every device, driver and bus is a directory; **one value per file**. `/sys/fs/cgroup` is the cgroup v2 interface.

chmod and umask Arithmetic

$ chmod 754 script.sh

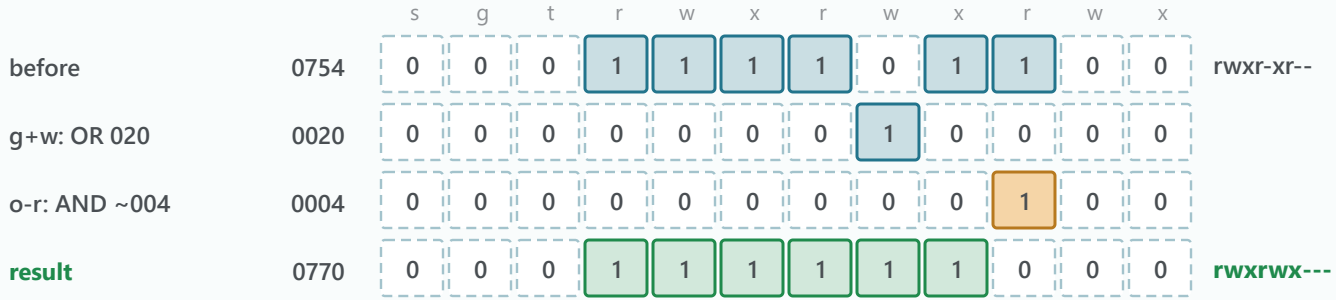


Each octal digit is three bits: 7 = 111 = rwx, 5 = 101 = r-x, 4 = 100 = r--.

Check: `stat -c "%a %A" file` shows the octal and symbolic modes together; `umask -S` prints the mask symbolically.

chmod and umask Arithmetic

$ chmod g+w,o-r script.sh

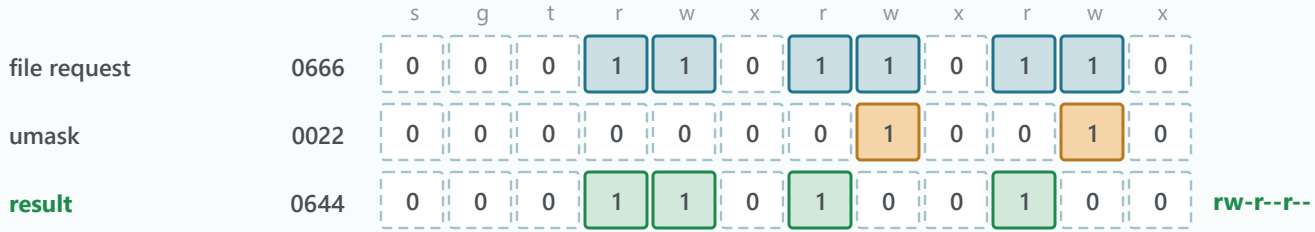


Symbolic mode changes the existing bits: the group write bit is added and the others' read bit removed → 0770.

Check: `stat -c "%a %A" file` shows the octal and symbolic modes together; `umask -S` prints the mask symbolically.

chmod and umask Arithmetic

$ umask 022 ; touch a ; mkdir d

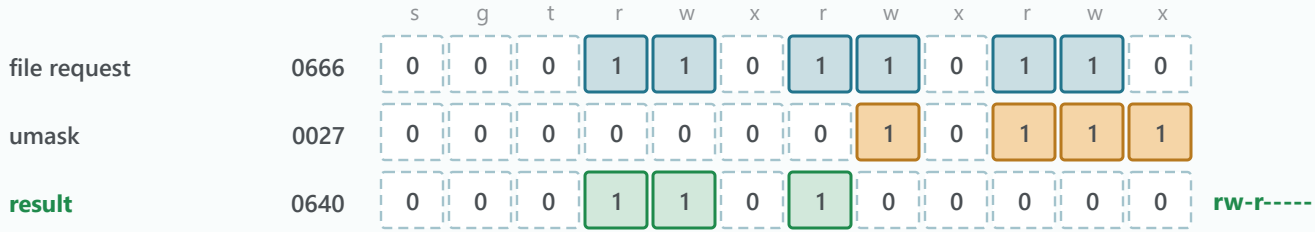


A new file asks for 666 and a directory for 777; the bits in umask are **cleared**: 666 & ~022 = 0644, directory 777 & ~022 = 0755. (Bit clearing, not subtraction!)

Check: `stat -c "%a %A" file` shows the octal and symbolic modes together; `umask -S` prints the mask symbolically.

chmod and umask Arithmetic

$ umask 027 ; touch a ; mkdir d

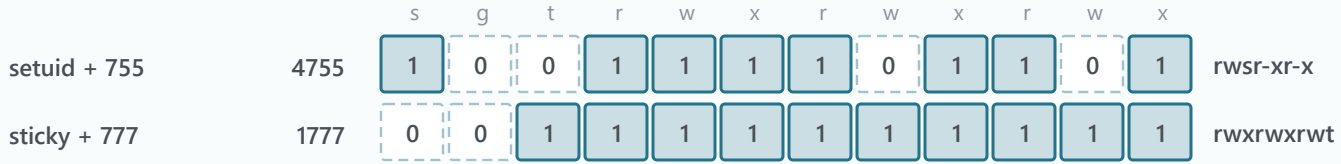


A stricter umask: file 666 & ~027 = 0640 (rw-r-----), directory 777 & ~027 = 0750 (rwxr-x---). Others see nothing.

Check: `stat -c "%a %A" file` shows the octal and symbolic modes together; `umask -S` prints the mask symbolically.

chmod and umask Arithmetic

$ chmod 4755 passwd ; chmod 1777 /tmp



4xxx setuid: the running process takes the file owner's identity → rwsr-xr-x. **2**xxx setgid (on a directory: new files inherit the group). **1**xxx sticky bit: everyone writes in /tmp, but a file can be deleted or renamed only by the file's owner, the directory's owner or root (a suitably privileged process) → rwxrwxrwt.

Check: `stat -c "%a %A" file` shows the octal and symbolic modes together; `umask -S` prints the mask symbolically.

SECURITY

Access Control, root and Capabilities

1. euid = 0 (root) → allow (execution needs at least one **x** bit)
2. euid = file owner → look **only** at the owner bits
3. egid or a supplementary group = file group → group bits
4. otherwise the other bits

The first matching class wins: for a `----rw-rw-` file, its owner ali **cannot read** it while others can.

Capabilities: with `setcap cap_net_raw+ep /usr/bin/ping`, `ping` opens a raw socket without being setuid root.

Call	Function
`chmod`	Changes a file's permission mode
`chown`	Changes owner and group
`access`	Tests access with the **real** uid/gid
`getuid`, `geteuid`	Real / effective uid
`getgid`, `getegid`	Real / effective gid
`setuid`, `setresuid`	Changes the uid (dropping privilege)
`setgid`	Changes the gid
`umask`	Permission mask for new files

User and group IDs live in `/etc/passwd` and `/etc/group`; password hashes are kept in `/etc/shadow`, readable only by root. `sudo` grants authorized users root rights per command.

File Permission Check: Code Trace

```
▶ int has_perm(cred *c, inode *f, int req) {
2     int bit, m = f->mode;        /* r4 w2 x1 */
3     if (c->euid == 0)            /* root */
4         return !(req & 1) || (m & 0111);
5     if (c->euid == f->uid)
6         bit = (m >> 6) & 7;      /* owner */
7     else if (in_group(c, f->gid))
8         bit = (m >> 3) & 7;      /* group */
9     else
10        bit = m & 7;             /* other */
11    return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: — · —

report.txt: owner ali, group team, mode 0640 = rw-r-----. veli is in the team group; can is not.

Step 1 / 7

VARIABLES

`mode = 0640`

OUTPUT

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(req & 1) || (m & 0111);
▶      if (c->euid == f->uid)
▶          bit = (m >> 6) & 7;      /* owner */
7      else if (in_group(c, f->gid))
8          bit = (m >> 3) & 7;      /* group */
9      else
10         bit = m & 7;             /* other */
▶      return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: ali (uid 1000) · w

allow

ali owns the file: the **owner** bits are chosen, bit = 6 (rw-). 6 & 2 == 2: **allowed**.

Step 2 / 7

VARIABLES

`euid = 1000` `req = w` `class = owner` `result = allow`

OUTPUT

```
ali w: allow
```

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(req & 1) || (m & 0111);
▶      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* owner */
▶      else if (in_group(c, f->gid))
▶          bit = (m >> 3) & 7;      /* group */
9      else
10         bit = m & 7;             /* other */
▶      return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: veli (uid 1001) · w

deny (EACCES)

veli is in the team group: the **group** bits are chosen, bit = 4 (r--). 4 & 2 ≠ 2: **denied**.

Step 3 / 7

VARIABLES

`euid = 1001` `req = w` `class = group` `result = deny`

OUTPUT

```
ali w: allow   veli w: deny
```

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(req & 1) || (m & 0111);
▶      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* owner */
▶      else if (in_group(c, f->gid))
▶          bit = (m >> 3) & 7;      /* group */
9      else
10         bit = m & 7;             /* other */
▶      return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: veli (uid 1001) · r

allow

veli is in the team group: the **group** bits are chosen, bit = 4 (r--). 4 & 4 == 4: **allowed**.

Step 4 / 7

VARIABLES

`euid = 1001` `req = r` `class = group` `result = allow`

OUTPUT

```
ali w: allow   veli w: deny
veli r: allow
```

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(req & 1) || (m & 0111);
▶      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* owner */
▶      else if (in_group(c, f->gid))
8          bit = (m >> 3) & 7;      /* group */
9      else
▶          bit = m & 7;             /* other */
▶      return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: can (uid 1002) · r

deny (EACCES)

can is neither the owner nor in the group: the **other** bits are chosen, bit = 0 (---). 0 & 4 ≠ 4: **denied**.

Step 5 / 7

VARIABLES

`euid = 1002` `req = r` `class = other` `result = deny`

OUTPUT

```
ali w: allow   veli w: deny
veli r: allow   can r: deny
```

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
▶      if (c->euid == 0)            /* root */
▶          return !(req & 1) || (m & 0111);
5      if (c->euid == f->uid)
6          bit = (m >> 6) & 7;      /* owner */
7      else if (in_group(c, f->gid))
8          bit = (m >> 3) & 7;      /* group */
9      else
10         bit = m & 7;             /* other */
11     return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0640



request: root (uid 0) · r

euid 0: permission bits ignored

allow

root (euid 0): for reading and writing the permission bits are not checked at all; **allowed**. Only execution requires at least one x bit.

Step 6 / 7

VARIABLES

`euid = 0` `req = r` `class = root` `result = allow`

OUTPUT

```
ali w: allow   veli w: deny
veli r: allow   can r: deny
root r: allow
```

File Permission Check: Code Trace

```
1  int has_perm(cred *c, inode *f, int req) {
2      int bit, m = f->mode;        /* r4 w2 x1 */
3      if (c->euid == 0)            /* root */
4          return !(req & 1) || (m & 0111);
▶      if (c->euid == f->uid)
▶          bit = (m >> 6) & 7;      /* owner */
7      else if (in_group(c, f->gid))
8          bit = (m >> 3) & 7;      /* group */
9      else
10         bit = m & 7;             /* other */
▶      return (bit & req) == req;
12 }
```

report.txt: owner ali, group team, mode 0066



request: ali (uid 1000) · r

deny (EACCES)

Take mode 0066 (----rw-rw-): can and veli may read, but ali falls into the owner class, which has no bits: **denied**. The first matching class wins; the other bits are never consulted.

Step 7 / 7

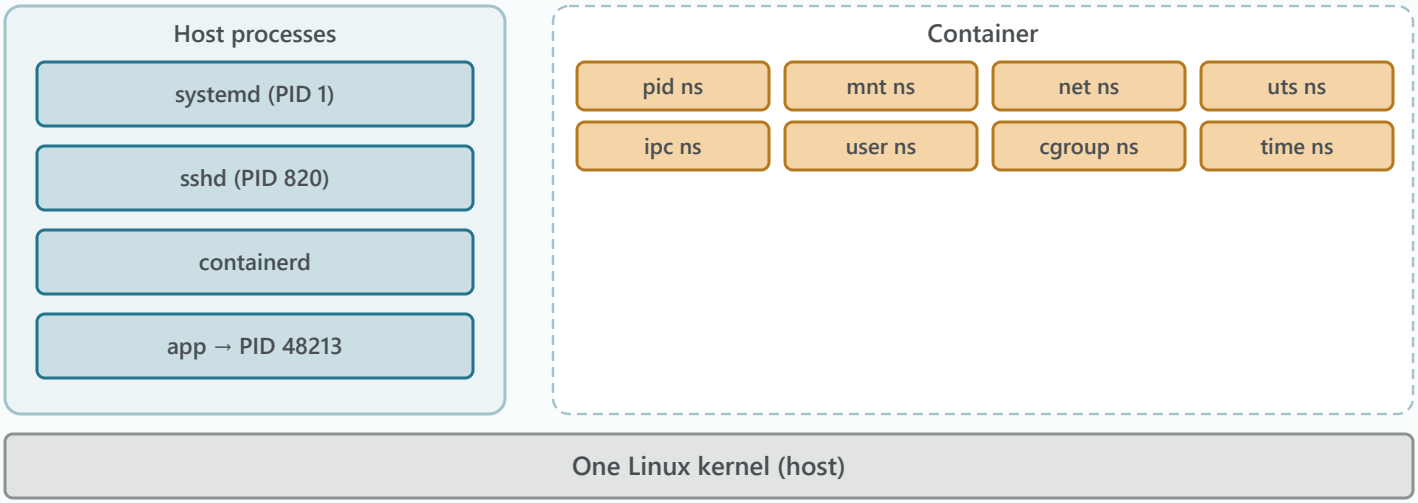
VARIABLES

`euid = 1000` `req = r` `class = owner` `result = deny`

OUTPUT

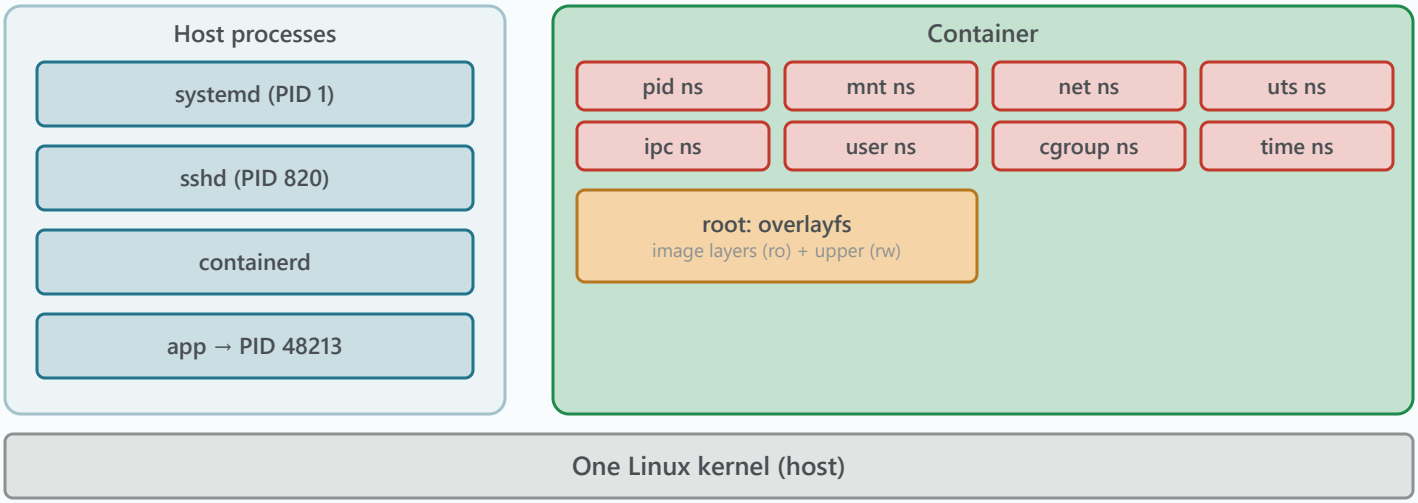
```
ali w: allow   veli w: deny
veli r: allow   can r: deny
root r: allow   0066 ali r: deny
```

Containers: Namespaces + cgroups



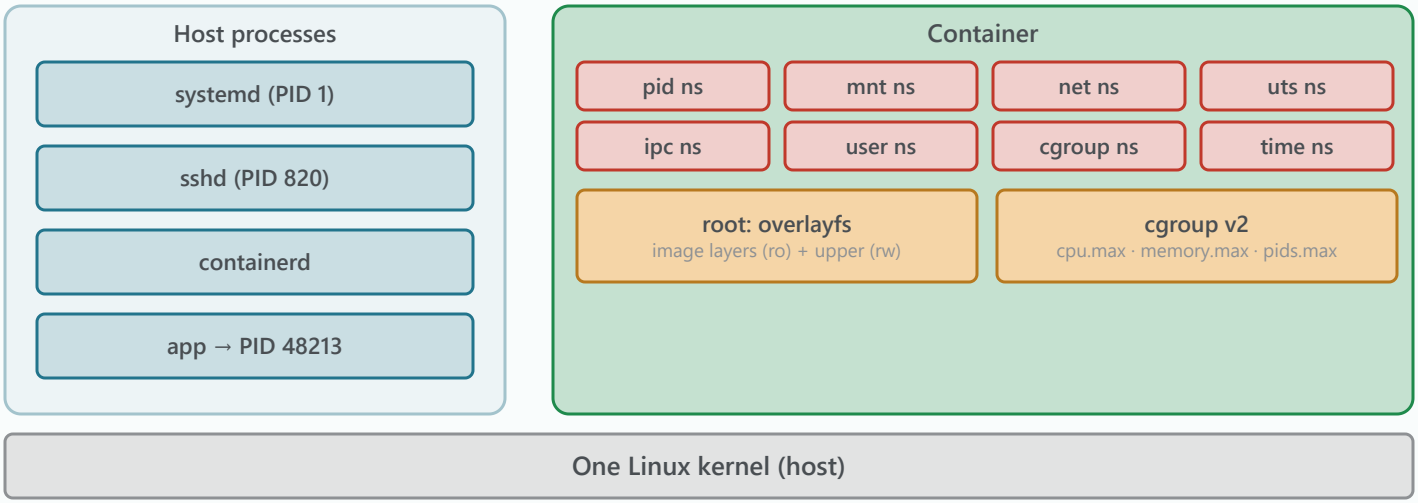
The runtime (runc, crun) creates new **namespaces** with `clone()` / `unshare()`: its own PID numbers, mount tree, network stack, host name, IPC, user mapping (root inside is an ordinary uid outside), cgroup and clock views.

Containers: Namespaces + cgroups



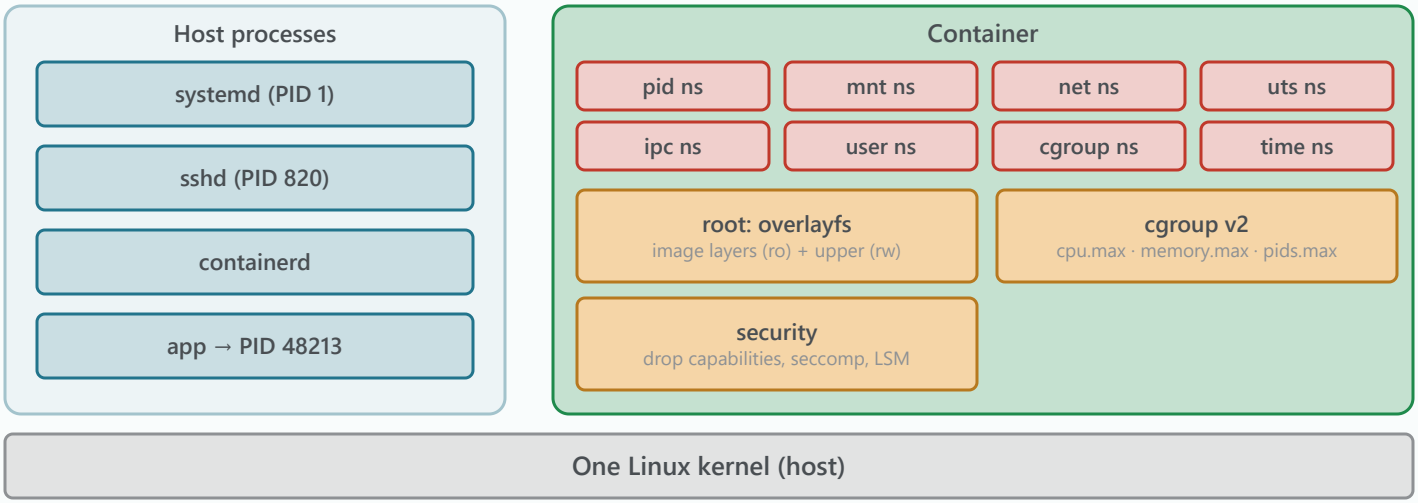
Inside the mount namespace, `pivot_root` switches to the image's root file system. The root is usually **overlayfs**: read-only image layers + a writable upper layer private to the container.

Containers: Namespaces + cgroups



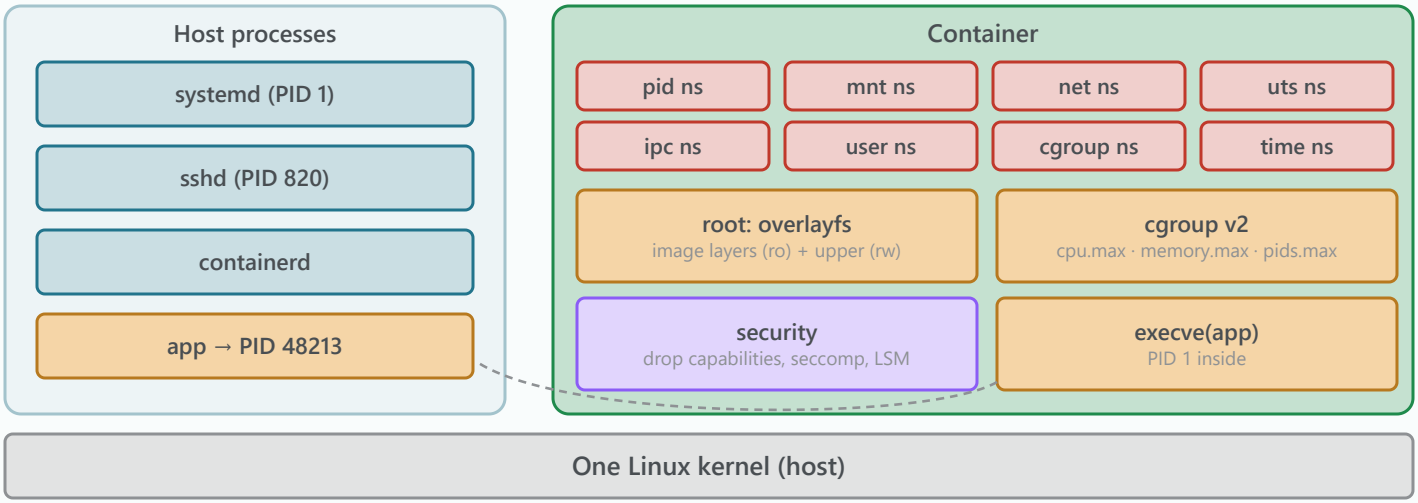
cgroup v2 limits resources: `cpu.max = "50000 100000"` (half a CPU), `memory.max = 512M`, `pids.max = 100`. Namespaces decide what is **seen**; cgroups decide how much is **used**.

Containers: Namespaces + cgroups



The attack surface is shrunk: unneeded capabilities are dropped, and `no_new_privs`, a seccomp filter (dozens of risky system calls blocked) and an AppArmor/SELinux profile are applied.

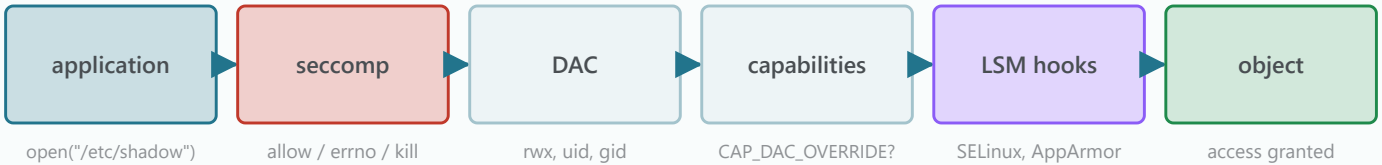
Containers: Namespaces + cgroups



Finally `execve`: the application is PID 1 inside and PID 48213 on the host. All share **the same kernel**: much lighter than a VM, but a kernel bug affects every container.

SECURITY

seccomp and Linux Security Modules (LSM)



if any stage says “no”: EPERM / EACCES (seccomp may kill the process)

SELinux

Label-based mandatory access control (MAC): every process and file has a type, and the policy says what each type may do to which. RHEL, Fedora, Android.

AppArmor

Path-based profiles: “this program may access only these files”. Easy to write. Ubuntu, SUSE, Debian.

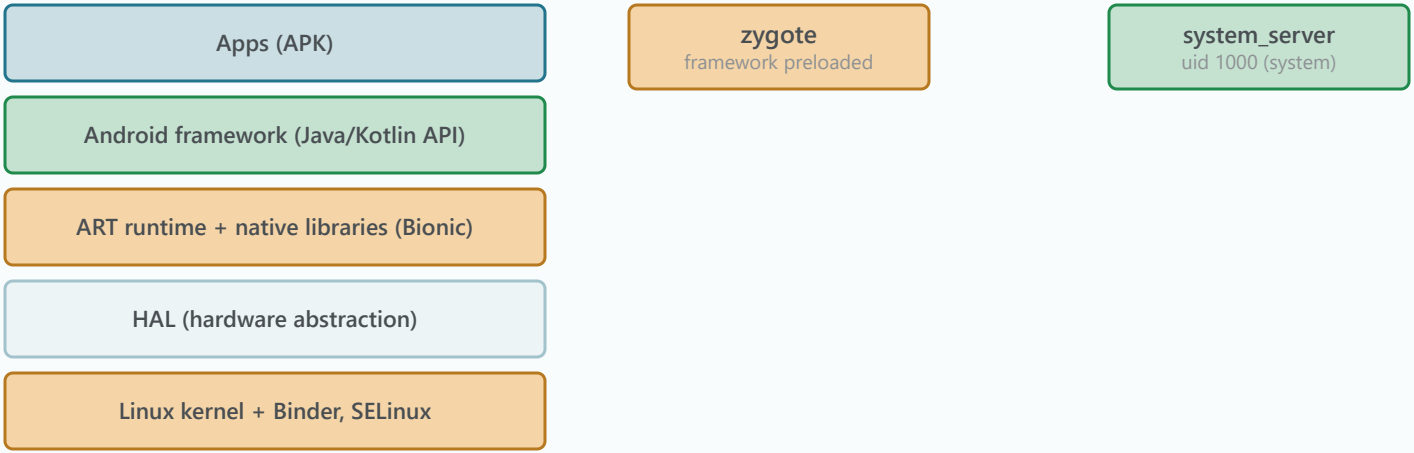
Landlock (5.13+)

An unprivileged process sandboxes **itself**: “from now on I may write only under ./data”.

seccomp-BPF

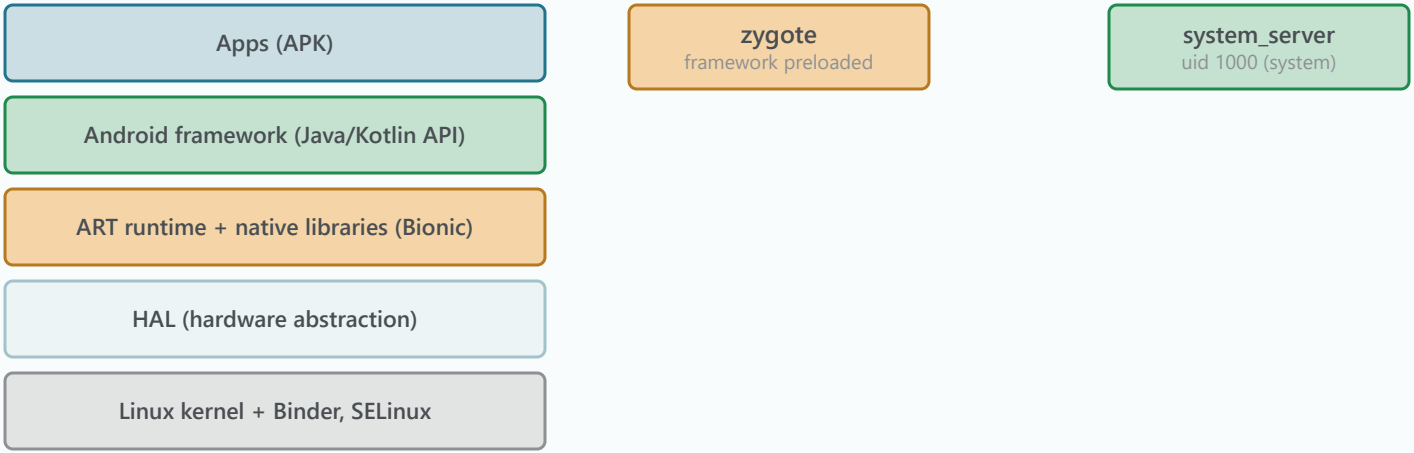
Filters the system calls a process may make; used by Chrome, systemd services, Docker and Android apps.

Android: Binder, Zygote and the Sandbox



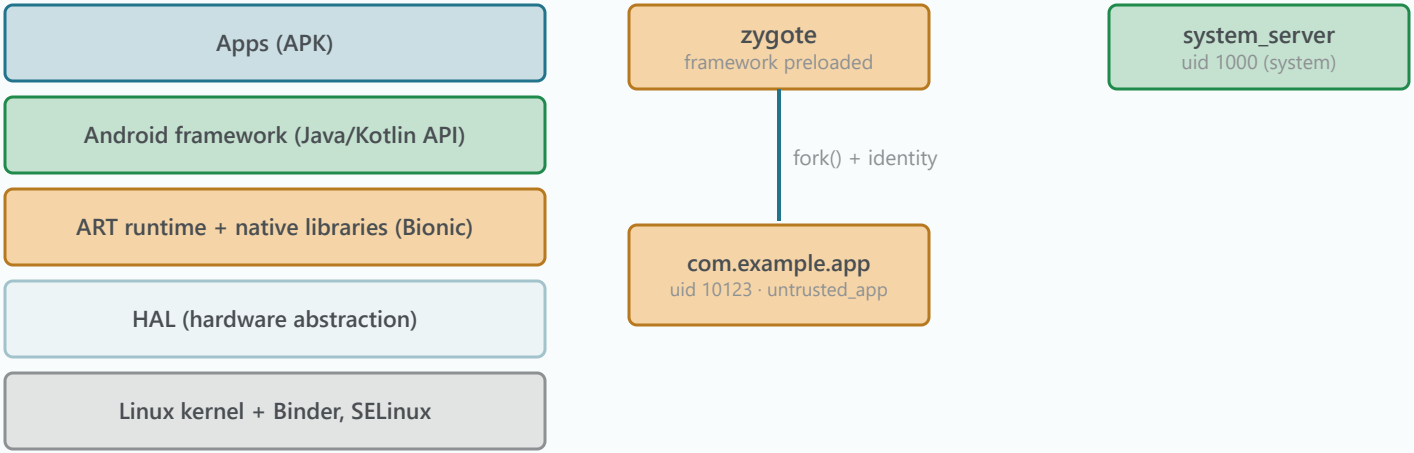
Android runs on the Linux kernel but has no GNU user space: the C library is **Bionic**, and apps run on ART. Added to the kernel: **Binder** IPC, power management (wakelocks), mandatory SELinux.

Android: Binder, Zygote and the Sandbox



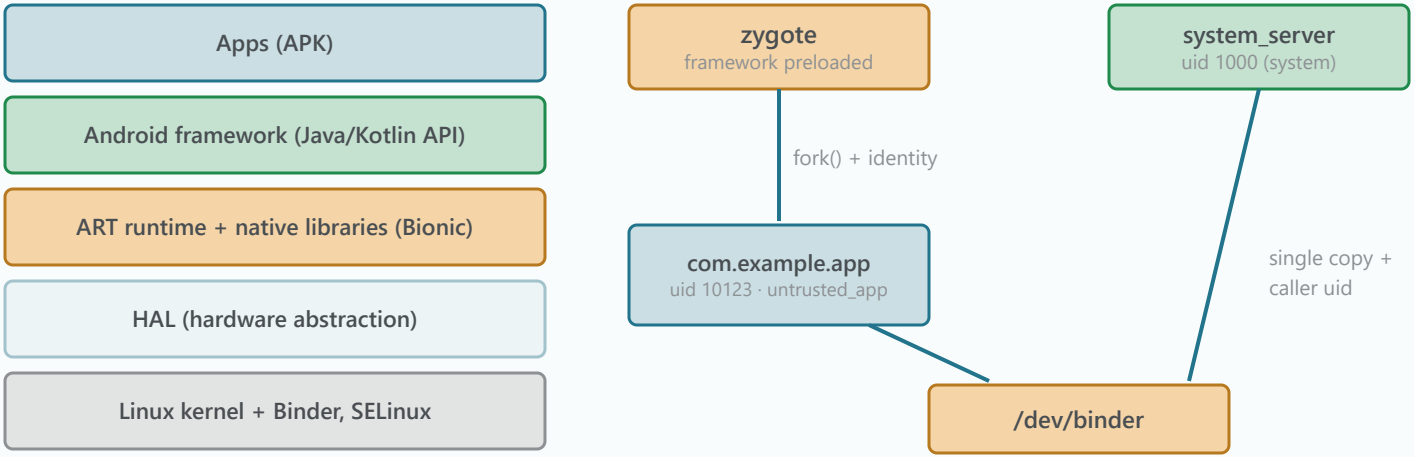
At boot init starts the **zygote** process; zygote loads the framework classes and resources once and waits.

Android: Binder, Zygote and the Sandbox



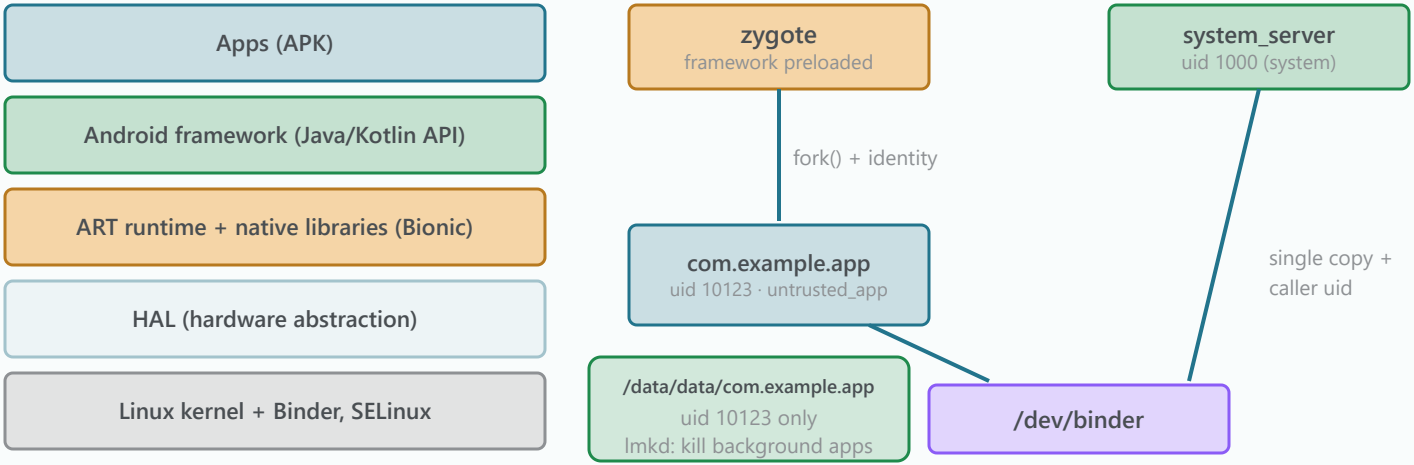
When an app launches, zygote `fork()`s: the loaded pages are shared via **COW**, which speeds up startup. The child switches to the app's **unique uid**, SELinux context and seccomp filter.

Android: Binder, Zygote and the Sandbox



The app calls a service (location, camera) through **Binder**: the kernel moves the data into the receiver's mapped buffer with **a single copy** and passes along the caller's uid, which the service uses to check permissions.

Android: Binder, Zygote and the Sandbox



Sandbox: every app is a separate Linux user, and its data is open only to itself. When memory runs low, the user-space `lmkd` uses PSI signals to kill unimportant background apps.

Summary: The Course's Concepts in Linux

Topic	Concept	In Linux
Structures	System calls, monolithic kernel	syscall, LKM, eBPF
Processes	PCB, creation, IPC	task_struct, fork/exec/wait, signals, pipes
Threads	Kernel threading model	clone(), NPTL 1:1
Scheduling	Fair sharing, real time	EEVDF, FIFO/RR/DEADLINE, sched_ext
Synchronization	Locks, condition variables	futex, spinlock, RCU
Deadlocks	Detection and prevention	lockdep, fcntl EDEADLK
Memory	Allocation, fragmentation	zones, buddy lists, SLUB
Virtual memory	Paging, replacement	4/5-level tables, COW, MGLRU, zswap, OOM
File systems	i-nodes, directories, journaling	VFS, ext4, XFS, Btrfs
I/O	Drivers, scheduling	device files, blk-mq, io_uring
Multiprocessors	Load balancing, NUMA	per-CPU queues, scheduling domains
Security	Access control, isolation	rwx, capabilities, namespaces, SELinux, seccomp
Design	Separating mechanism and policy	cgroups, sched_ext: policy in user space

SUMMARY

Review Questions

1. Which structures does `fork()` copy in the kernel, and which does it share? What does COW save?
2. How does the shell set up `ls | wc -l`? What happens if the parent does not close the pipe ends?
3. What is the difference between a process and a thread in Linux? Explain with clone flags.
4. What changed in the fair class in Linux 6.6, and what stayed the same?
5. What are the four 4-level table indices and the offset of the address `0x00007f3a1c2b5e88`?
6. Why is SLUB better than allocating pages directly for objects like `task_struct`?
7. With `umask 027`, what permissions do a new file and a new directory get?
8. What is the difference between a hard link and a symbolic link? What happens when the target is deleted?
9. How is consistency restored after a crash with ext4 ordered journaling?
10. Container = namespaces + cgroups: what does each provide? How does it differ from a VM?

▶ **Answers**

References

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 10: Case Study 1: Unix, Linux and Android.
- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Chapter 20: The Linux System.
- R. Love, *Linux Kernel Development*, 3rd ed., Addison-Wesley, 2010.
- M. Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010; man7.org manual pages.
- The Linux Kernel Documentation, docs.kernel.org — scheduler (EEVDF, sched-ext), admin-guide/mm (MGLRU, zswap), block (blk-mq), filesystems (vfs, ext4), userspace-api (seccomp, landlock).
- Android Open Source Project, source.android.com — architecture, Binder, the application sandbox.
- D. M. Ritchie, K. Thompson, “The UNIX Time-Sharing System”, *Communications of the ACM*, 1974.

The End

Chapter 15: Linux

Previous: Chapter 14: Operating-System Design · This is the last chapter of the course.