Chapter 14: Operating-System Design

Operating Systems

- Why is design hard?
- Goals of an operating system
- Interface principles and paradigms
- The system-call interface
- Choosing a structure: isolation and speed
- Mechanism, policy, orthogonality
- Naming and binding time
- Implementation strategies
- Useful techniques
- Performance: trade-offs, caching, hints
- Project management and Brooks's law
- Testing and debugging
- Trends: cloud, Rust, eBPF

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_14_Operating_System_Design.html

Learning Outcomes

Challenges and goals

Explain what makes OS design hard and the four basic goals of an operating system.

Interface design

Discuss the principles of simplicity, completeness and efficiency, and the execution and data paradigms, with system-call examples.

System structure

Weigh structural options by the isolation–performance trade-off; treat layer order, safe extension and kernel threads as design decisions.

Design principles

Apply the separation of mechanism and policy, orthogonality, two-level naming, binding time, and static versus dynamic structures.

Techniques and performance

Use indirection, reentrancy, space–time trade-offs, caching, hints and speeding up the common case, with examples.

Projects, testing and trends

Explain Brooks's law; choose kernel testing and debugging tools; assess the trends of virtualization, multicore, Rust, eBPF and unikernels.

Why Is Designing an OS Hard?

1 · Sheer size

Linux is ≈ 40 million lines; no one grasps it all.

2 · Concurrency

Devices, interrupts and threads all at once: races, deadlocks.

3 · Hostile users

Some want to break the system or steal data; every input is suspect.

4 · Sharing despite isolation

Users must be kept apart, yet able to share information with whom they choose.

5 · Long life

Lives 20–30 years; future hardware and applications must be anticipated.

6 · Generality

No one knows how it will be used: from phones to supercomputers.

7 · Portability

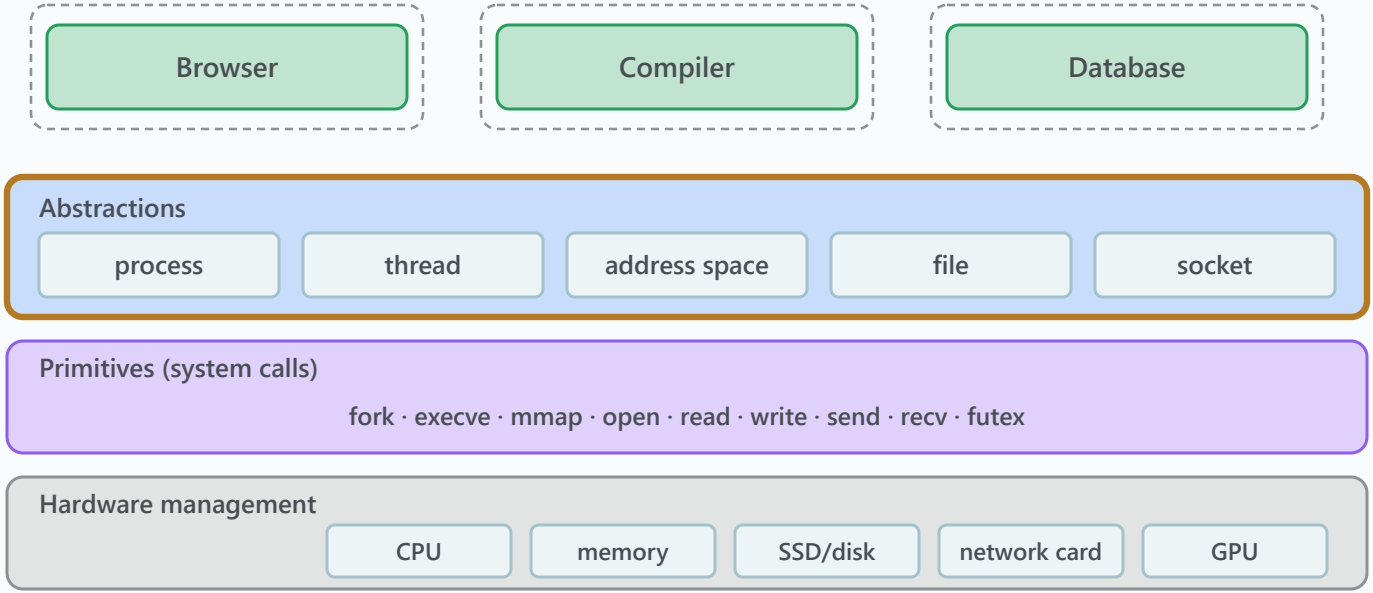
Different processors and platforms: x86-64, ARM64, RISC-V…

8 · Backward compatibility

Old programs must keep running; new ideas are bound by old constraints.

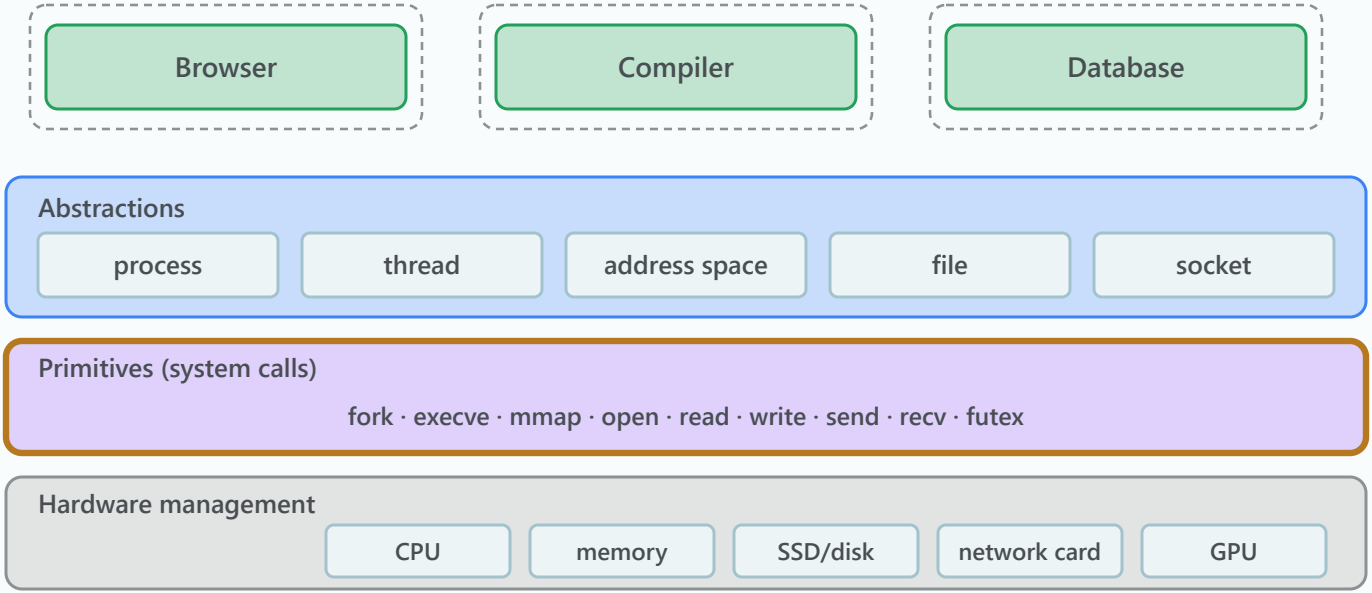
Tanenbaum's eight factors. The need for long-term maintenance follows naturally from items 5 and 8.

Goals of an Operating System



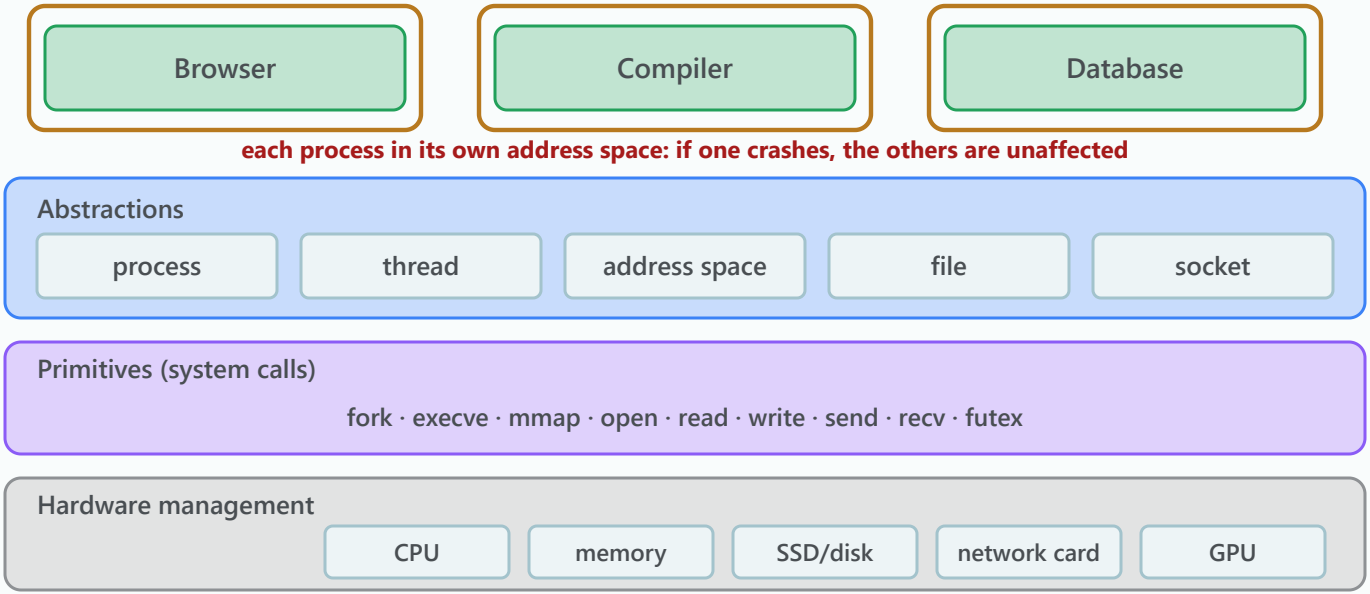
1. **Define abstractions**: concepts such as processes, address spaces, files and sockets hide the complexity of the hardware. Finding the right abstraction is the hardest part of design.

Goals of an Operating System



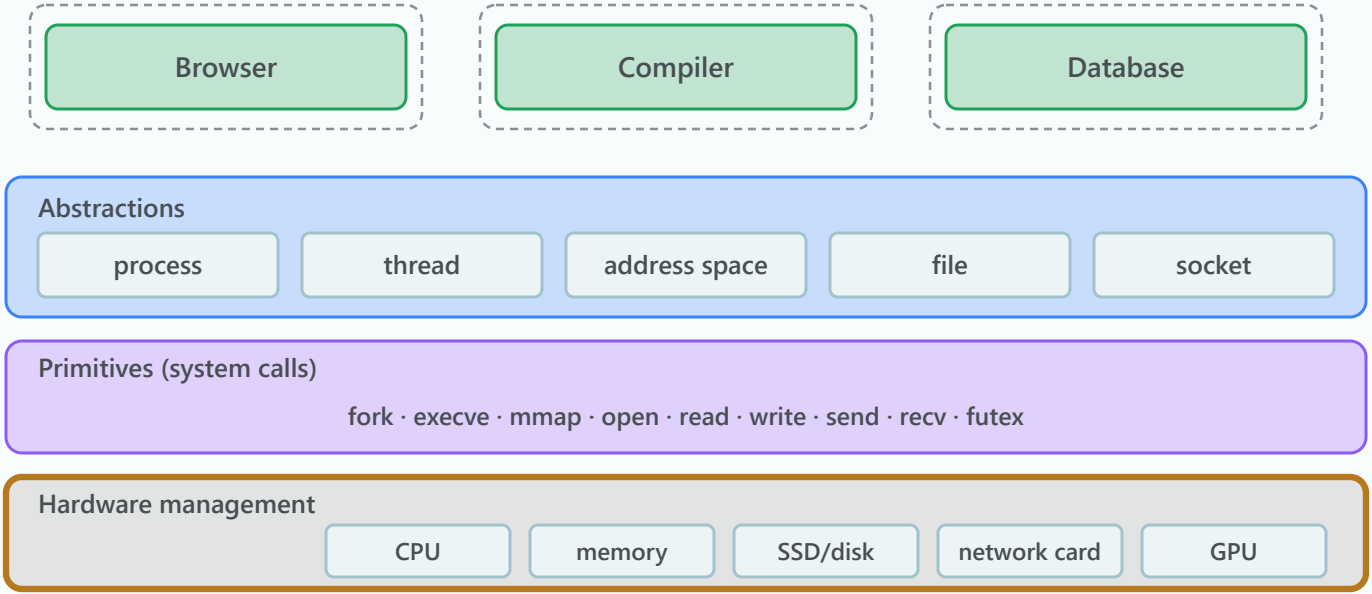
2. Provide primitive operations: the operations on those abstractions, i.e. system calls. E.g. `open/read/write/close` for files.

Goals of an Operating System



3. Ensure isolation: users and processes must not interfere with each other, yet controlled sharing (file permissions, shared memory) must be possible.

Goals of an Operating System



4. Manage the hardware: interrupts, timers, the MMU, device controllers; most low-level work happens in this layer.

Guiding Principles for Interface Design

1 · Simplicity

"Perfection is reached not when there is no longer anything to add, but when there is **no longer anything to take away**." (A. de Saint-Exupéry). A few well-chosen concepts are easy to learn, implement and verify.

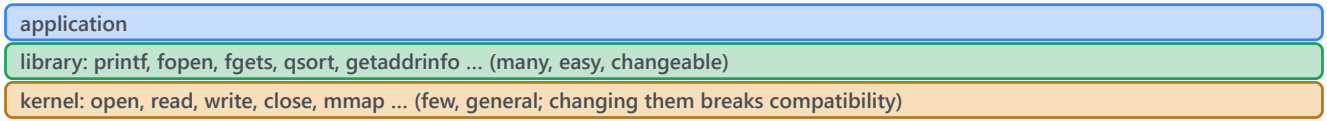
2 · Completeness

"Everything should be as simple as possible, but **no simpler**." (attributed to Einstein). The interface must let users do everything they need. Much work with few mechanisms: the MINIX 3 kernel offers only a few communication primitives (send, receive, sendrec, notify).

3 · Efficiency

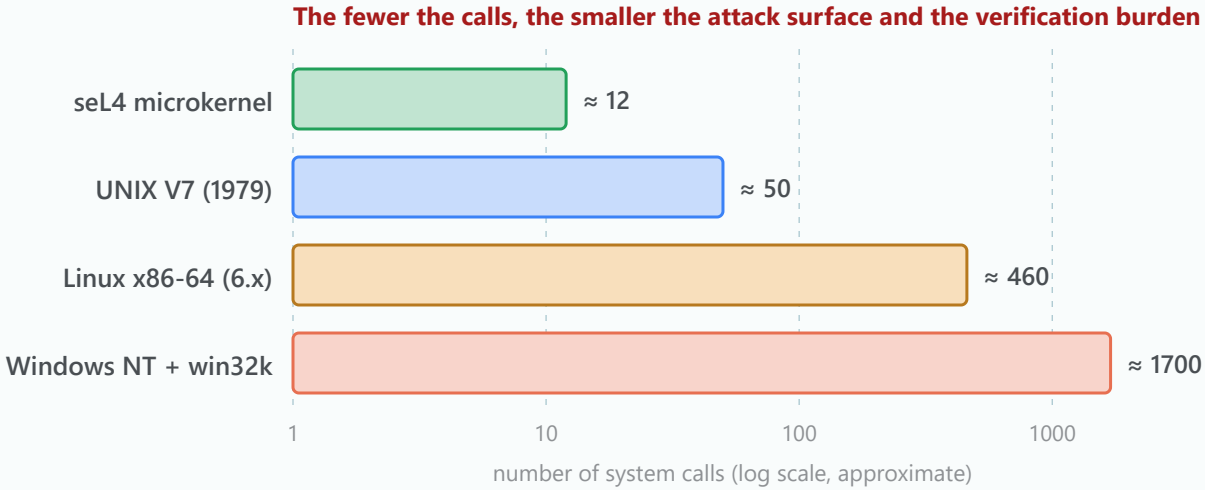
If a feature or system call cannot be implemented efficiently, it is probably **not worth having**. Programmers should be able to guess a call's rough cost: `lseek` is cheap, `read` may be expensive.

Rule of thumb: **each system call should do one thing well**. Adding features is easy; removing them is nearly impossible (backward compatibility).



INTERFACE DESIGN

“Small Is Beautiful”



- **Small is beautiful**: a few general calls beat many special ones.
- seL4's kernel is about 10,000 lines and **formally verified**; that is impossible for a kernel of millions of lines.
- Linux gains a few calls every year (e.g. `io_uring_*`, `landlock_*`, `mseal`).
- On Windows the documented interface is the Win32 API; kernel calls change from release to release.

INTERFACE DESIGN

User-Interface Paradigms

Command line (CLI)

Shells, text commands, pipes. Still indispensable for automation and remote administration.

WIMP / GUI

Windows, Icons, Menus, Pointer. Mouse buttons should mean the same thing in every application.

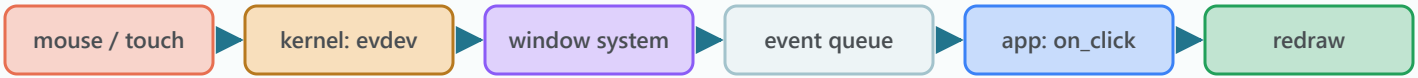
Touch and mobile

Gestures (swipe, pinch), full-screen apps, a notification center.

Voice, chat and agents

Voice assistants and natural-language interfaces; a new entry point into OS services.

The user interface is often not part of the kernel, but the **paradigm** shapes the whole system design: a GUI needs an event-driven execution model, a fast window system and a consistent application framework. What matters is **consistency**: the same action should give the same result everywhere.



latency budget: touch to screen ≈ 16 ms (60 Hz); a long handler freezes the whole UI

Execution Paradigms

(a) Algorithmic code

```
int main(void)
{
    int x, y;
    init();
    do_something();
    read(fd, &x, sizeof x);
    do_something_else();
    write(fd, &y, sizeof y);
    keep_going();
    exit(0);
}
```

The **program** sets the flow and calls the OS for services. Compilers, sorters, batch jobs.

(b) Event-driven code

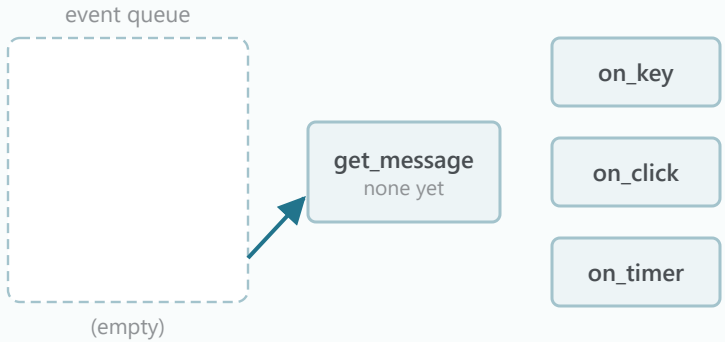
```
int main(void)
{
    msg_t msg;
    init();
    while (get_message(&msg)) {
        switch (msg.type) {
        case 1: ... ; break;
        case 2: ... ; break;
        case 3: ... ; break;
        }
    }
}
```

External events (keys, clicks, packets, timers) set the flow. GUIs, servers, microkernel servers.

The choice of execution paradigm shapes the system calls too: the algorithmic model wants a blocking `read`; the event-driven model wants a "give me the next event" call (`get_message`, `epoll_wait`, `io_uring_enter`).

How Does an Event Loop Work?

```
▶ int main(void) {
▶     msg_t m;
▶     init();
4     while (get_message(&m)) {
5         switch (m.type) {
6         case KEY:   on_key(&m);   break;
7         case CLICK: on_click(&m); break;
8         case TIMER: on_timer(&m); break;
9         }
10    }
11    return 0;
12 }
```



The program starts; `init()` sets up the window and resources. The queue is empty.

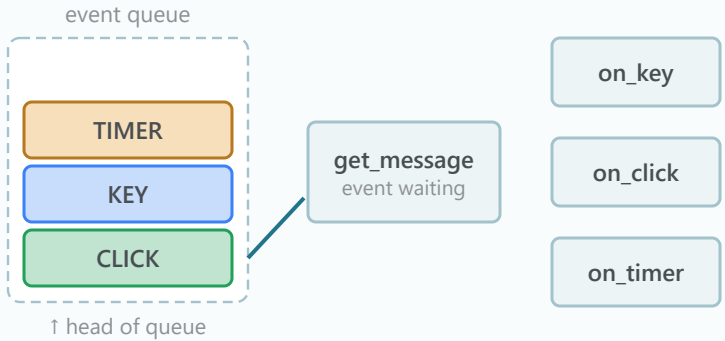
Step 1 / 8

VARIABLES

`m.type = –` `queue = []`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
▶      while (get_message(&m)) {
5          switch (m.type) {
6          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



The user clicks, presses a key, and a timer fires. The OS/window system collects the events and appends them to the application's queue.

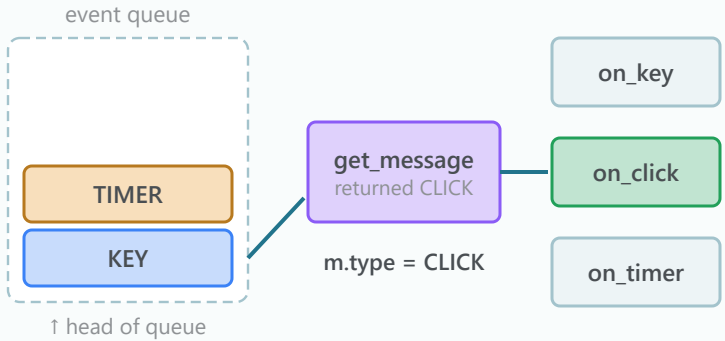
Step 2 / 8

VARIABLES

`m.type = –` `queue = [CLICK, KEY, TIMER]`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
4      while (get_message(&m)) {
▶          switch (m.type) {
6          case KEY:   on_key(&m);   break;
▶          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



`get_message` returns the CLICK at the head of the queue; the `switch` routes it to `on_click`.

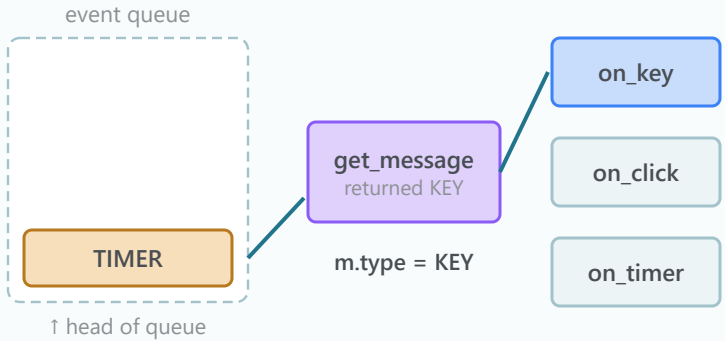
Step 3 / 8

VARIABLES

`m.type = CLICK` `queue = [KEY, TIMER]`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
▶      while (get_message(&m)) {
5          switch (m.type) {
▶          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



The handler finishes quickly and the loop goes around. The next event is KEY: `on_key` runs.

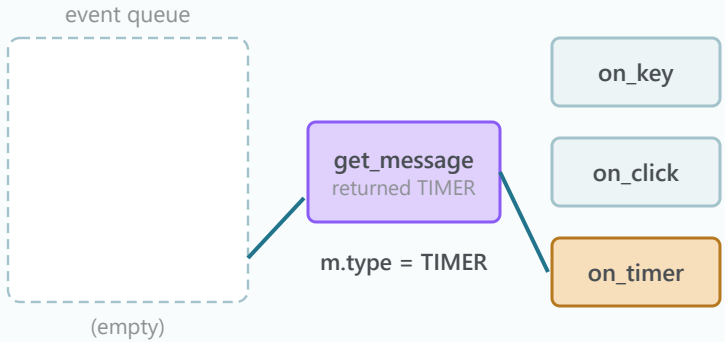
Step 4 / 8

VARIABLES

`m.type = KEY` `queue = [TIMER]`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
▶      while (get_message(&m)) {
5          switch (m.type) {
6          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
▶          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



The TIMER event goes to `on_timer` (e.g. blinking the cursor). A long-running handler freezes the whole UI; long jobs go to a separate thread.

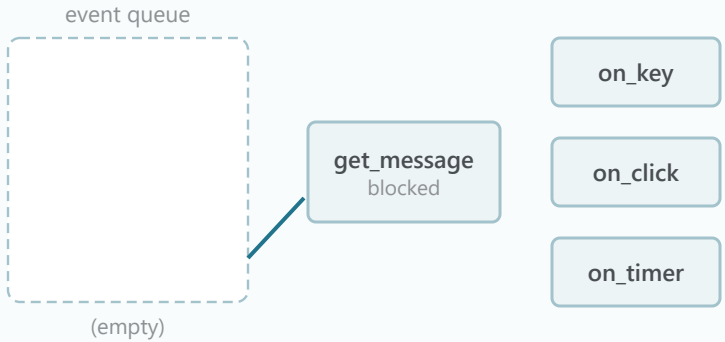
Step 5 / 8

VARIABLES

`m.type = TIMER` `queue = []`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
▶      while (get_message(&m)) {
5          switch (m.type) {
6          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



The queue is empty: `get_message` **blocks**, the process sleeps and uses no CPU. A new event wakes it.

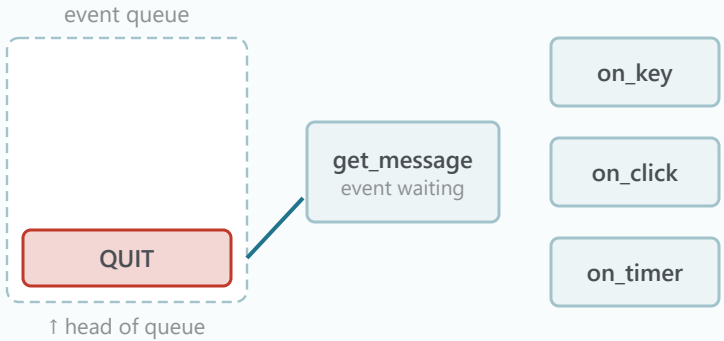
Step 6 / 8

VARIABLES

`m.type = TIMER` `queue = []`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
▶      while (get_message(&m)) {
5          switch (m.type) {
6          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
11     return 0;
12 }
```



The user closes the window: a QUIT event arrives in the queue.

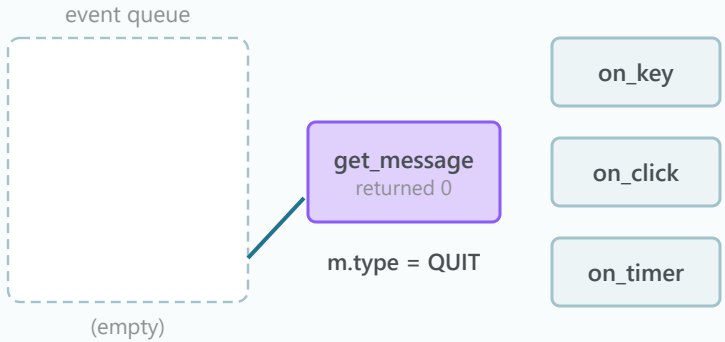
Step 7 / 8

VARIABLES

`m.type = –` `queue = [QUIT]`

How Does an Event Loop Work?

```
1  int main(void) {
2      msg_t m;
3      init();
4      while (get_message(&m)) {
5          switch (m.type) {
6          case KEY:   on_key(&m);   break;
7          case CLICK: on_click(&m); break;
8          case TIMER: on_timer(&m); break;
9          }
10     }
▶      return 0;
12 }
```



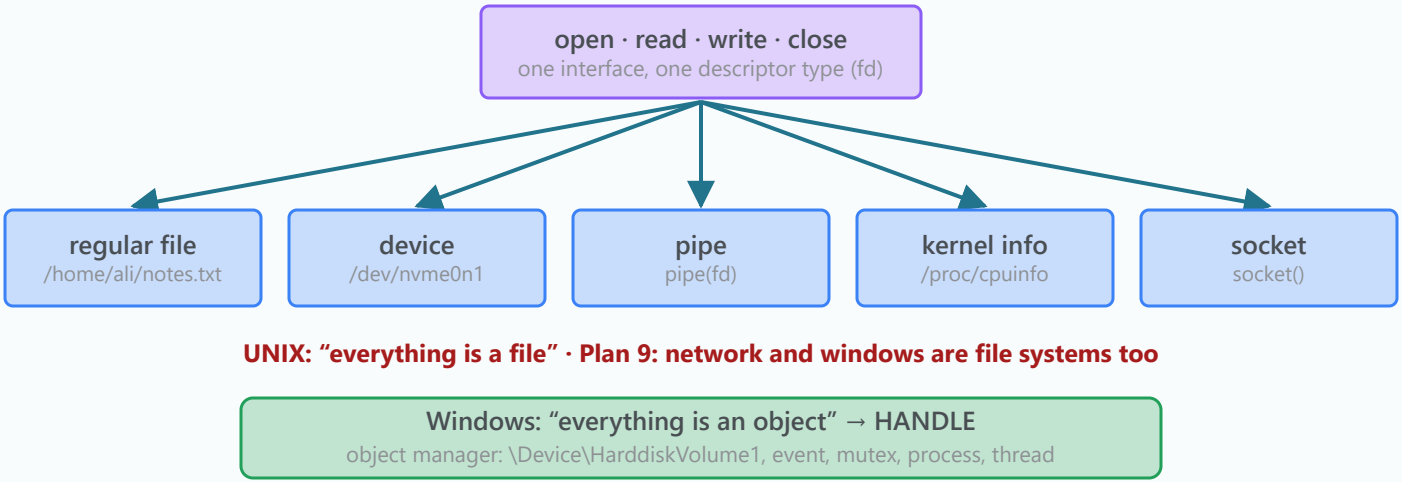
For QUIT `get_message` returns 0, the loop ends and the program exits. nginx/Node.js (epoll) and microkernel servers use the same pattern.

Step 8 / 8

VARIABLES

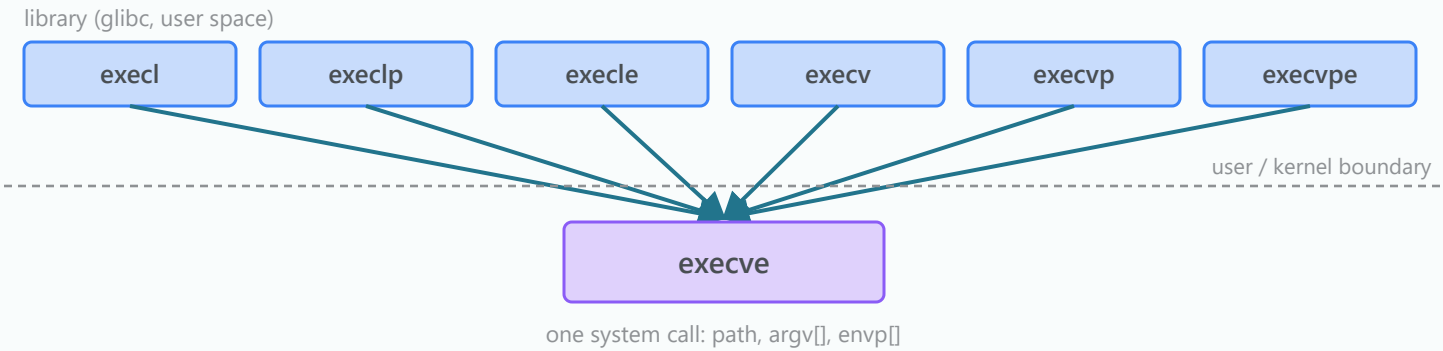
`m.type = QUIT` `queue = []`

Data Paradigms



- A common data paradigm lets the same tools (`cat` , `dd` , redirection) work on every resource: `cat /proc/loadavg` .
- The price: some devices do not fit the file model, and everything piles up behind “escape hatches” like `ioctl` .

The System-Call Interface



- **Separation**: few, general calls, with convenience functions in the library. Adding a convenience never changes the kernel.
- **Independent components**: UNIX splits process creation in two: `fork` + `execve`. Redirection, `chdir` and so on can happen in between.
- On Windows, `CreateProcess` takes 10 parameters in one call: powerful but complex.
- The modern trend: `posix_spawn` (avoids the cost of fork) and **batched-call** interfaces (`io_uring` : many requests in one crossing).

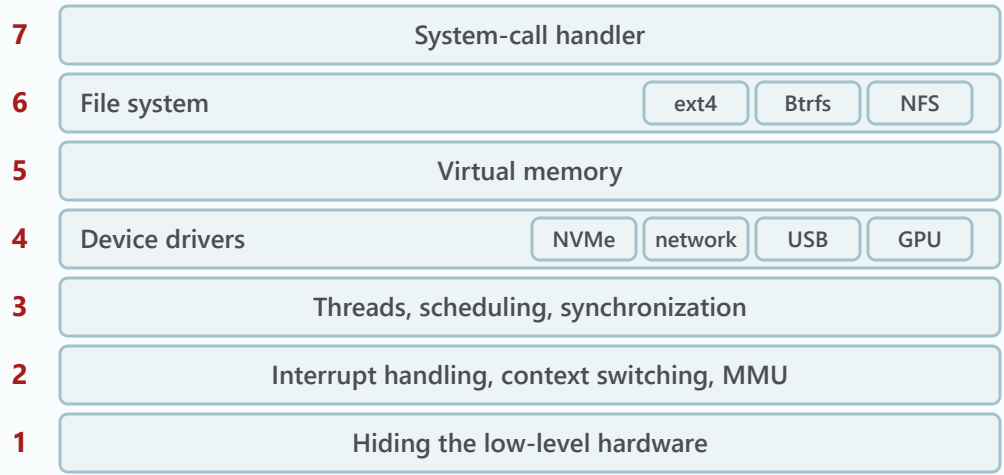
System Structure Options

Design decision	Gain	Cost	Example
Call everything directly in one address space	Speed: one trap, direct calls	No fault isolation: one driver bug brings down the system	Linux, FreeBSD
Restrict dependencies to one direction (layers)	Layer-by-layer testing; changes stay local	Choosing the layer order; the cost of every crossing	THE, MULTICS
Put services in separate address spaces	A crashed server is restarted; a small, verifiable kernel	IPC, context switches and copies on every request	MINIX 3, QNX, seL4, Zircon
Separate protection from management	Application-specific abstractions (libOS)	Hard to share between applications; libOS complexity	Aegis/ExOS, unikernels
Load code at run time	Adapt without recompiling	Extension safety: needs a verifier or a safe language	SPIN, Linux eBPF
Compromise on the pure model	Microkernel layout, monolithic speed	Part of the isolation is lost	Windows NT, macOS XNU

The basic trade-off is **isolation versus performance**: every crossing of an isolation boundary costs a trap, a context switch or a copy. The options are not exclusive: Linux is monolithic, but extensible with modules and eBPF, and it uses kernel threads.



Layered Systems: A Seven-Layer Design

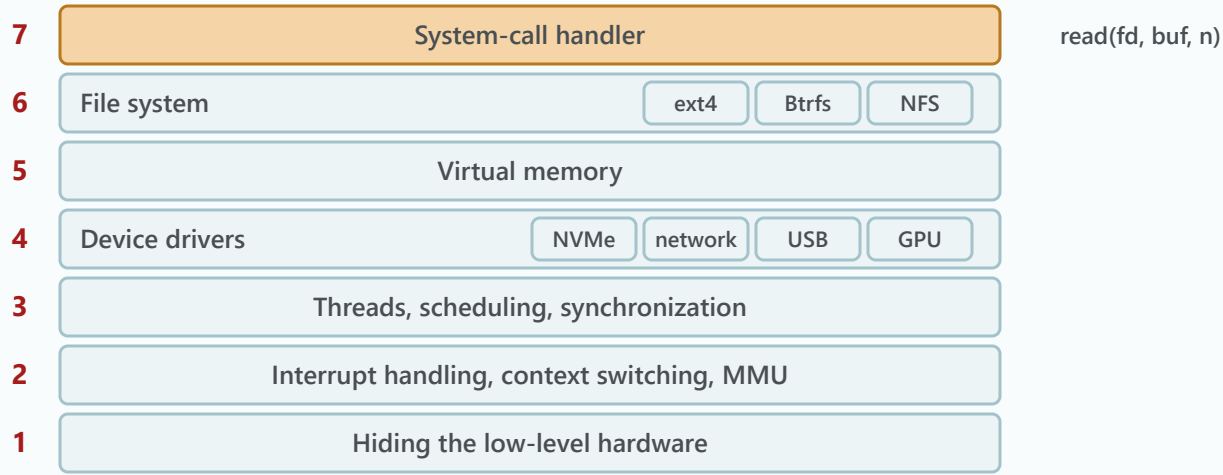


The **seven-layer** design Tanenbaum proposes. Layer 1 hides hardware differences; each upper layer works only with the abstractions the one below offers.

Step 1 / 7

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

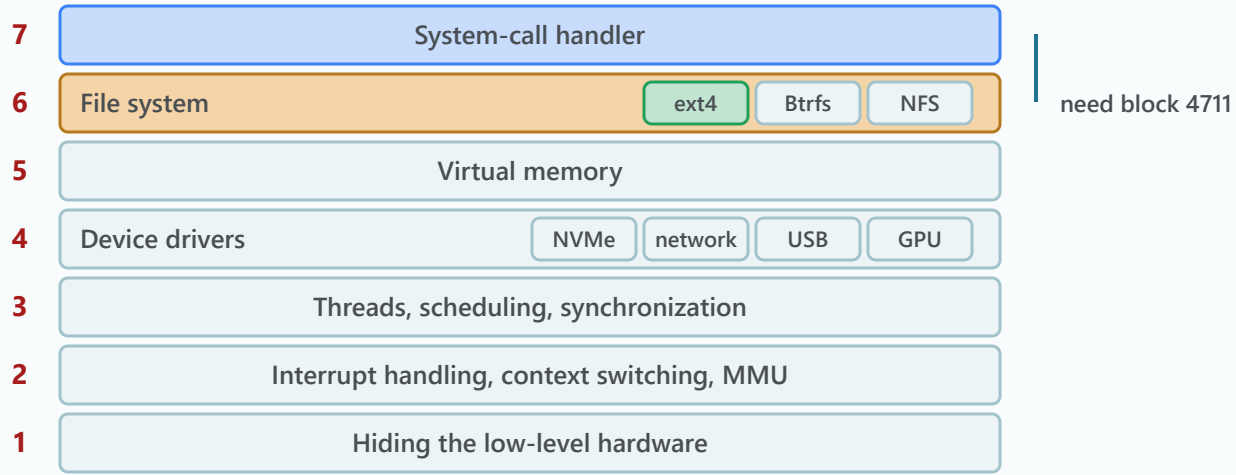
Layered Systems: A Seven-Layer Design



The application calls `read`; the trap arrives at layer 7. Parameters are checked here (is the fd valid, is the buffer in user space?).

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

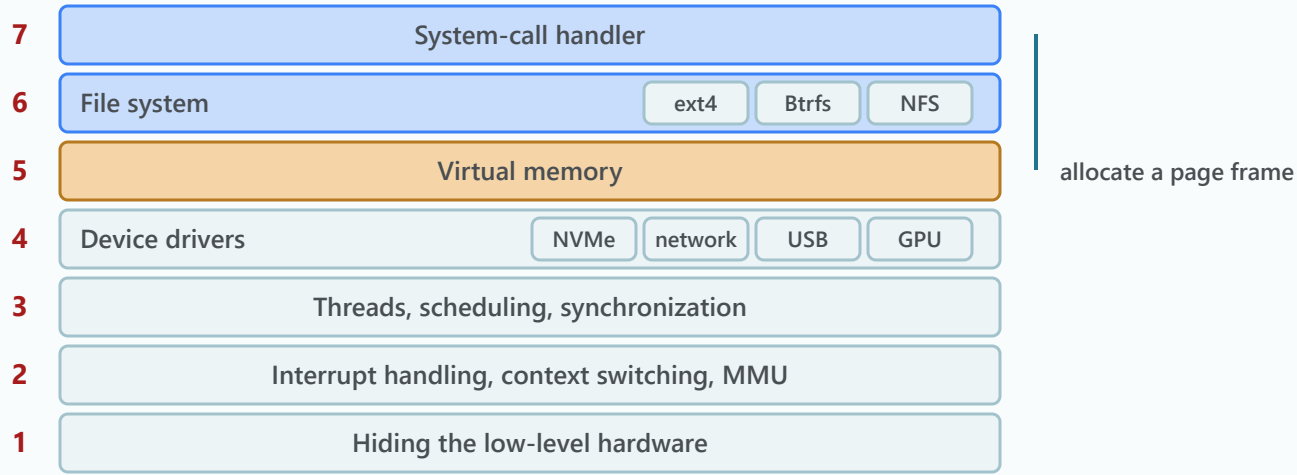
Layered Systems: A Seven-Layer Design



The file system (ext4) maps the file offset to a disk block. If the block is not cached, it is requested from the layers below.

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

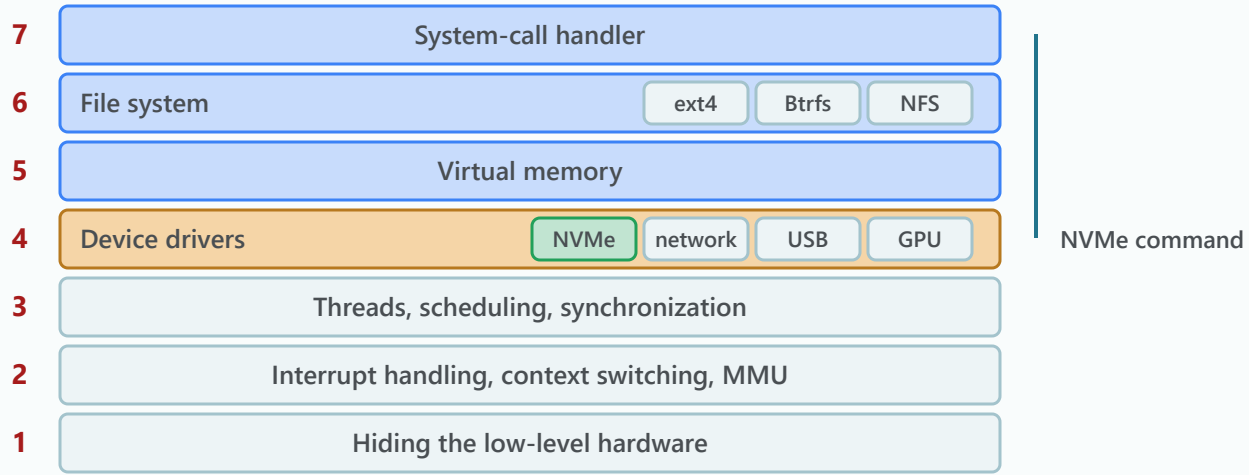
Layered Systems: A Seven-Layer Design



Virtual memory allocates the page-cache frame for the data and validates the mapping of the user buffer.

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

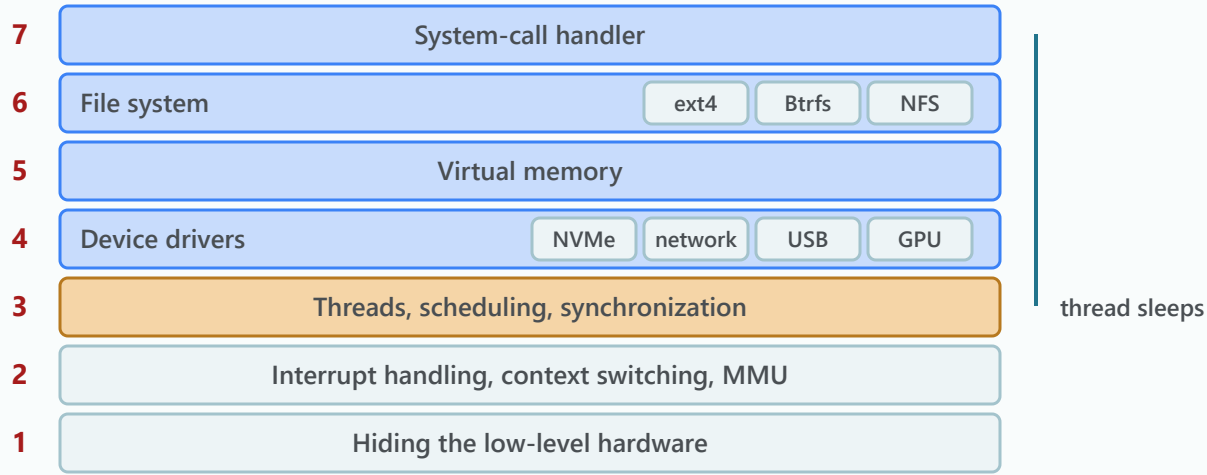
Layered Systems: A Seven-Layer Design



The NVMe driver writes the read command into the device's queue.

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

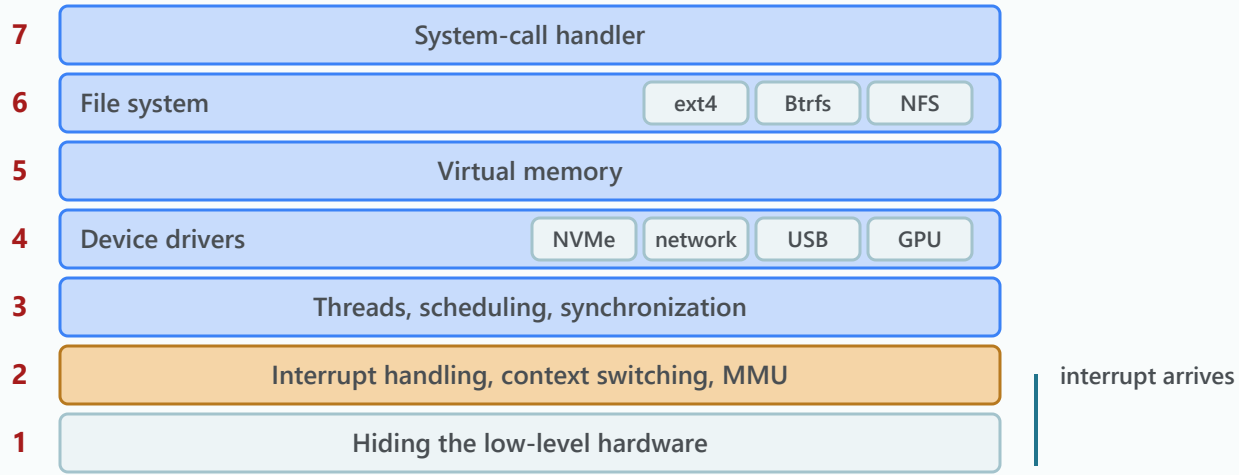
Layered Systems: A Seven-Layer Design



The calling thread **blocks** until the I/O completes; the scheduler runs another thread.

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

Layered Systems: A Seven-Layer Design

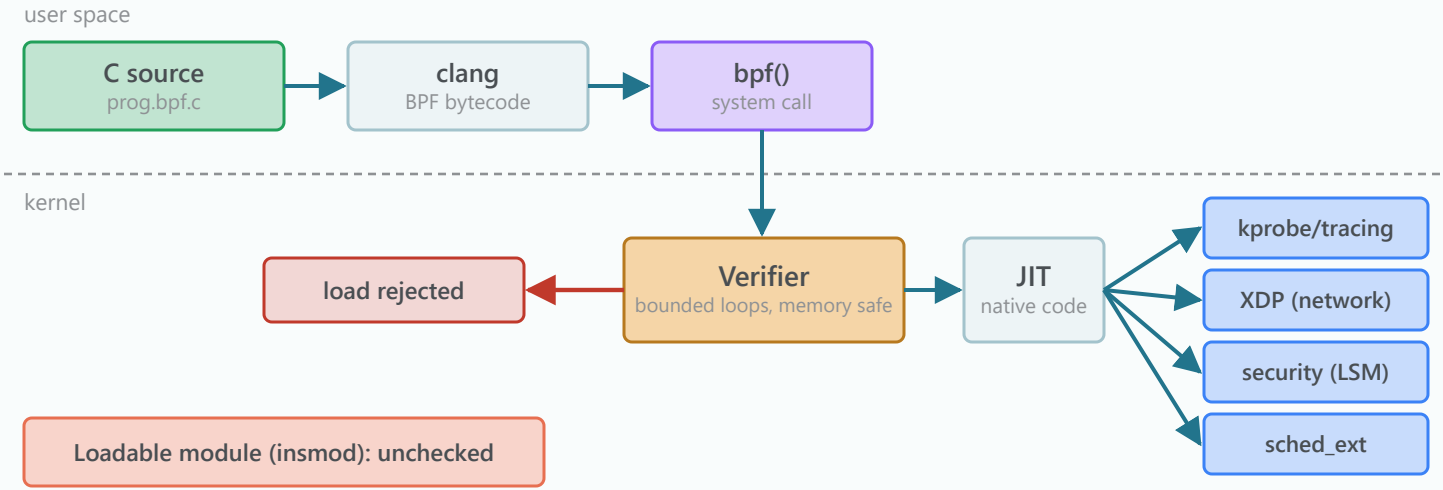


When done, the device raises an **interrupt**; layer 2 handles it and makes the sleeping thread ready. The result travels back up through the layers.

Step 7 / 7

The layer order follows from dependencies: virtual memory writes pages to disk, so it sits above the drivers; the file system allocates page frames, so it sits above virtual memory. Layer 1, which hides the hardware, provides portability; the price is the cost of every layer crossing.

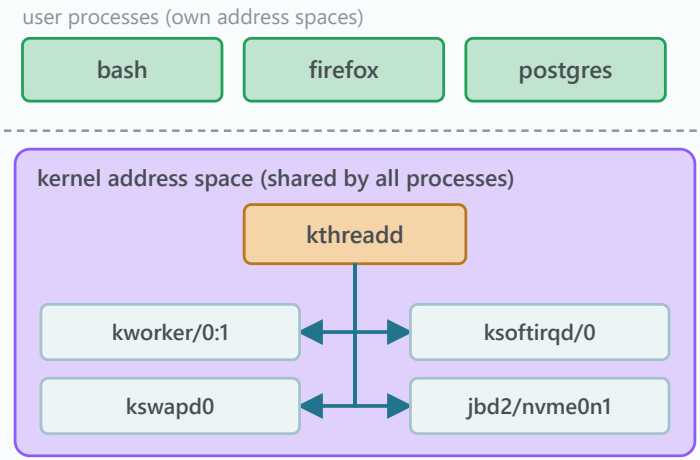
Extensible Systems



- The kernel provides the core functions; new components are added **at run time** as needs change.
- Third parties can add features to the system; compatibility and **security** are the biggest issues.
- A classic module has full privileges: one bug crashes the kernel. SPIN (1995) confined extensions with a safe language (Modula-3).
- Today Linux has **eBPF**: small, verified programs for tracing, networking, security and, since 6.12, the scheduler (sched_ext).

SYSTEM STRUCTURE

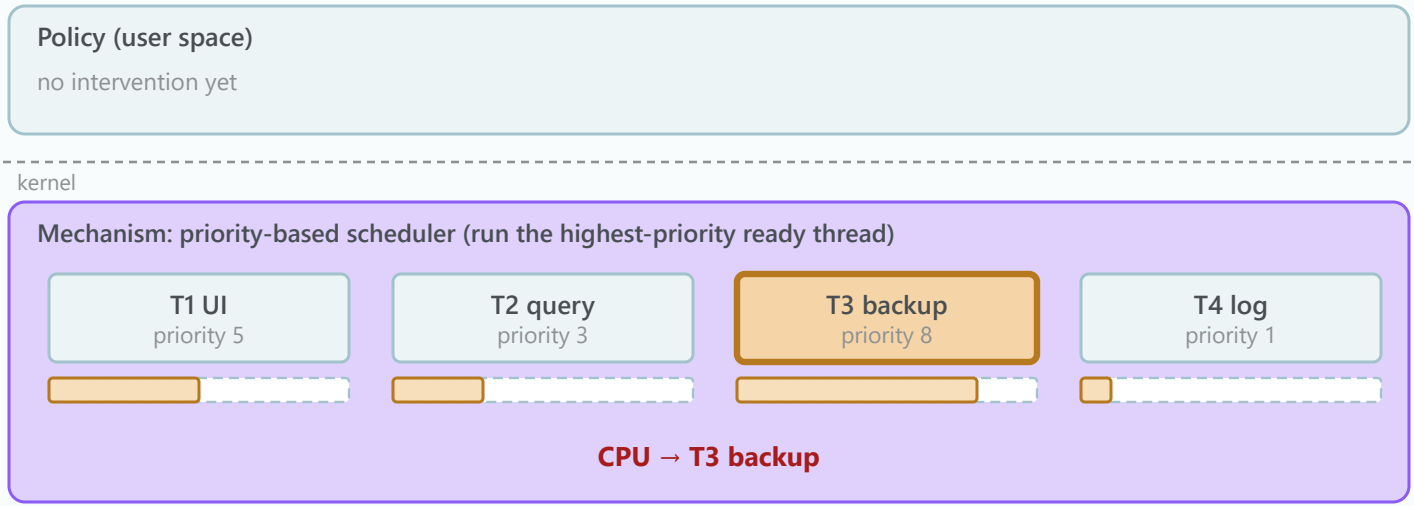
Kernel Threads



- Created and **managed** by the kernel (`kthread_create`); they have no user address space and run only in kernel memory.
- They are **scheduled** like ordinary threads and can block and sleep: long jobs that cannot be done in an interrupt handler are deferred here.
- Context switches are cheap: the address space (page table) does not change.
- Examples: deferred work (kworker), soft interrupts (ksoftirqd), page reclaim (kswapd), journal writes (jbd2).
- They show up in square brackets in `ps -ef` output: `[kswapd0]`.

User processes **cannot create** kernel threads; system calls can only hand work to threads the kernel already runs (e.g. the `io_uring` worker pool).

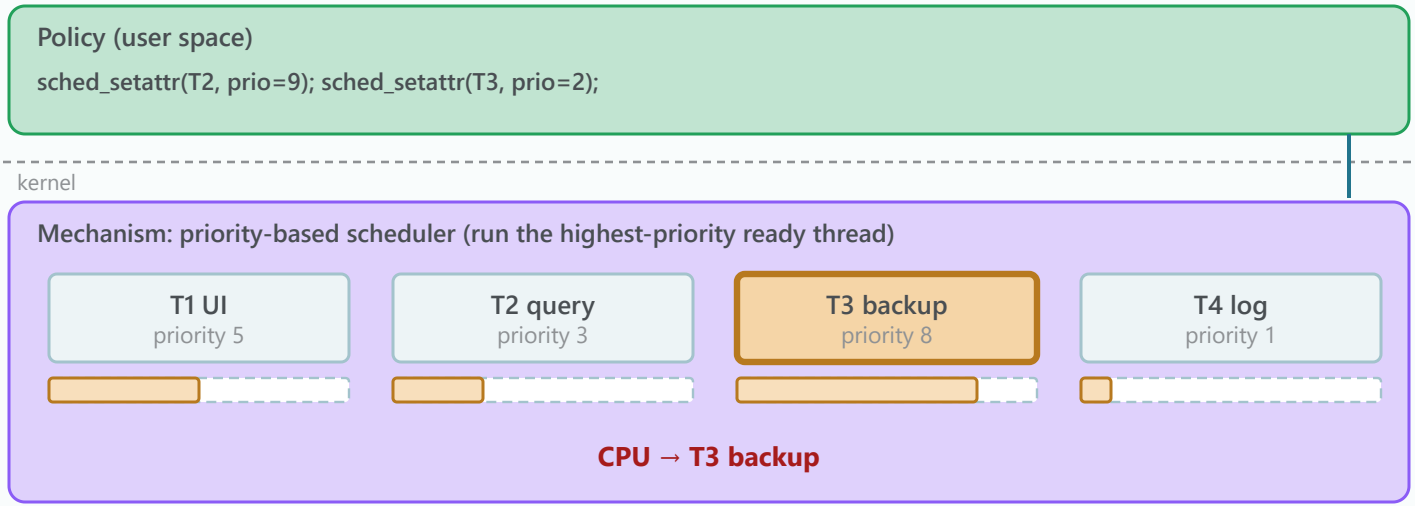
Separating Mechanism and Policy



The kernel offers only a **mechanism**: “run the highest-priority ready thread”. Right now T3 (backup, priority 8) is chosen.

Other examples: paging (mechanism: MMU, frame allocation; policy: which page to evict), CPU frequency (mechanism: the cpufreq driver; policy: the `schedutil` / `powersave` governor), security (mechanism: LSM hooks; policy: SELinux rules).

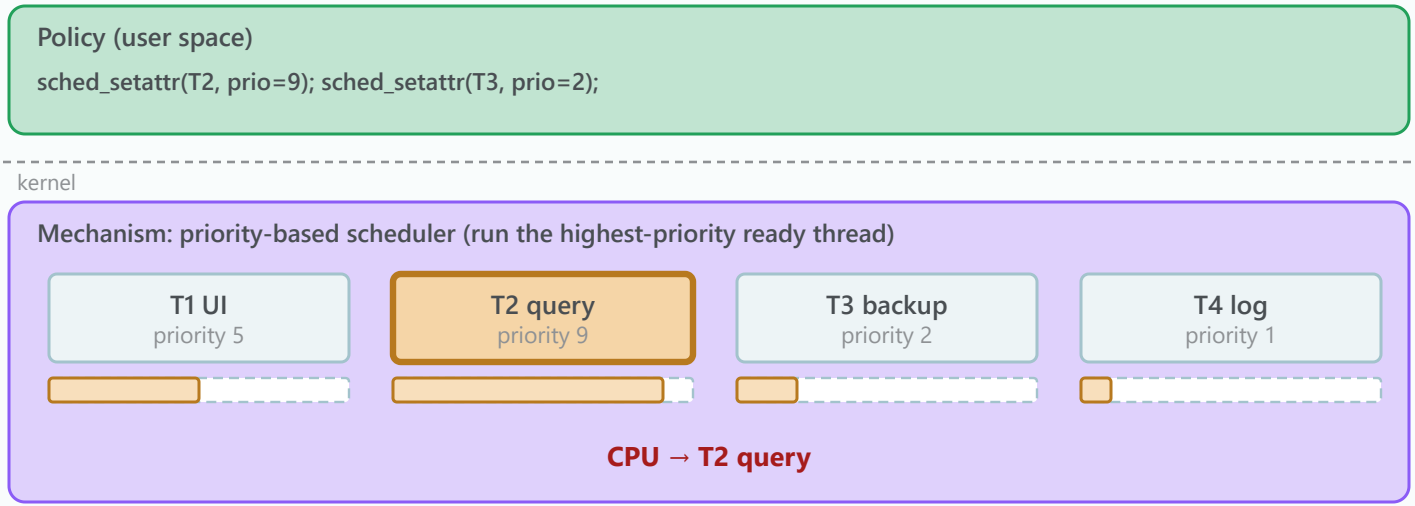
Separating Mechanism and Policy



The database process knows its own threads best: it decides that queries matter and the backup can wait. That is a **policy**, handed down through a system call.

Other examples: paging (mechanism: MMU, frame allocation; policy: which page to evict), CPU frequency (mechanism: the cpufreq driver; policy: the `schedutil` / `powersave` governor), security (mechanism: LSM hooks; policy: SELinux rules).

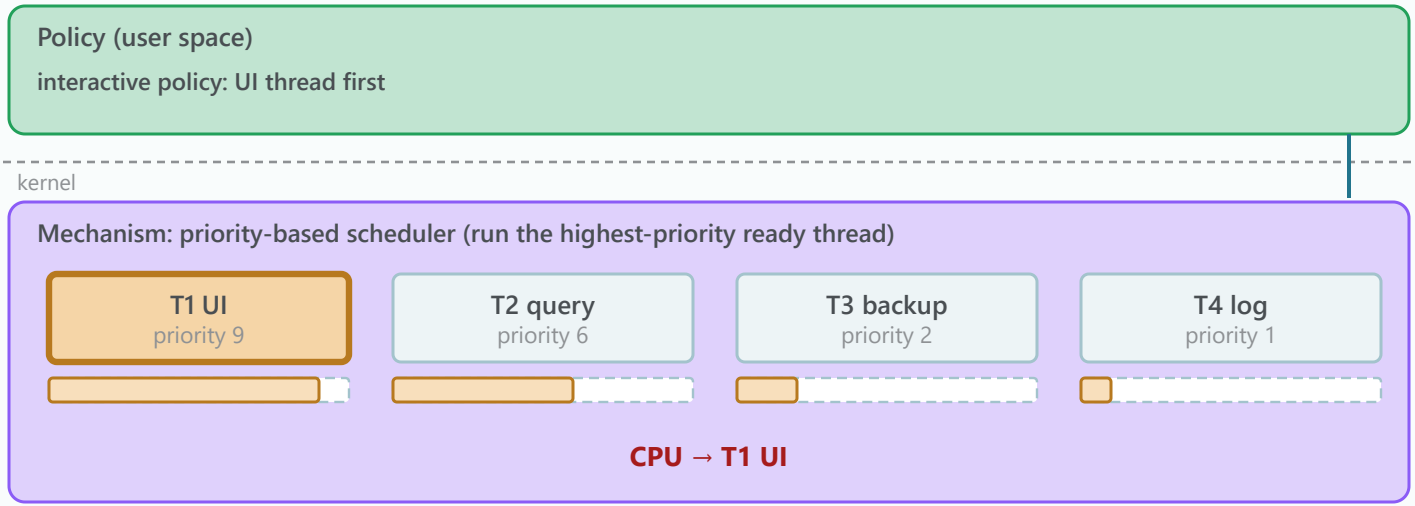
Separating Mechanism and Policy



Without any change to kernel code, T2 is now chosen. The same mechanism with a different policy gives different behavior.

Other examples: paging (mechanism: MMU, frame allocation; policy: which page to evict), CPU frequency (mechanism: the cpufreq driver; policy: the `schedutil` / `powersave` governor), security (mechanism: LSM hooks; policy: SELinux rules).

Separating Mechanism and Policy



Another application applies another policy. **sched_ext** in Linux 6.12 goes a step further: the entire scheduling policy can be loaded as an eBPF program.

Other examples: paging (mechanism: MMU, frame allocation; policy: which page to evict), CPU frequency (mechanism: the cpufreq driver; policy: the `schedutil` / `powersave` governor), security (mechanism: LSM hooks; policy: SELinux rules).

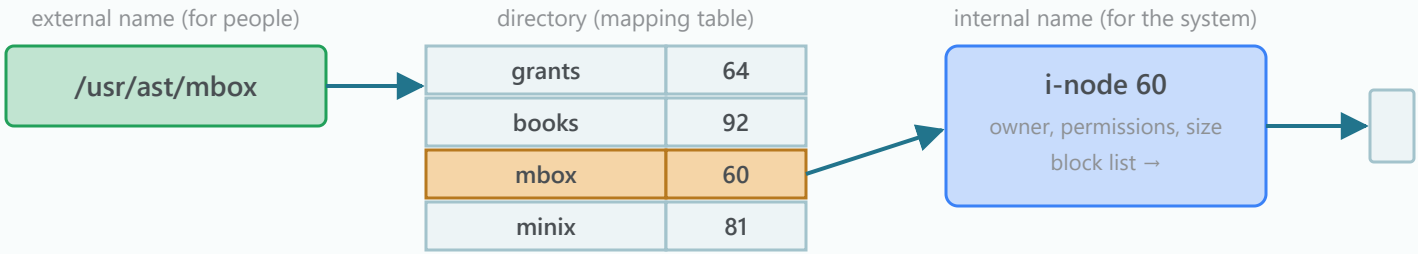
Orthogonality

	fork()	in between pthread_create()	
CLONE_VM address space	copy	copy	share
CLONE_FS root/working dir	copy	share	share
CLONE_FILES open files	copy	share	share
CLONE_SIGHAND signal handlers	copy	copy	share
CLONE_THREAD same thread group	copy	copy	share

process ← one call: clone(flags) → thread

- **Orthogonality**: the ability to combine separate concepts **independently**. Changing one does not affect another.
- Linux `clone`: separate flags choose what is shared. Processes and threads are two ends of the same mechanism; containers are built with the `CLONE_NEW*` flags too.
- Windows: a process is a **resource container** and a thread is the **unit of scheduling**; the two concepts are separate.
- C has an example too: `static` (lifetime) is chosen independently of the type.

Naming: Two-Level Names



External name	Mapped by	Internal name
`/usr/ast/mbox`	directory	i-node number
`www.giresun.edu.tr`	DNS	IP address → (ARP) MAC address
file descriptor `3`	per-process file table	`struct file` in the kernel
virtual address	page table	physical frame
`\Device\Harddisk0`	object manager	kernel object (HANDLE)

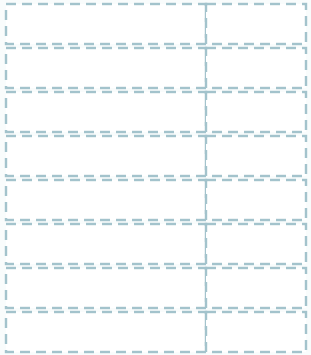
Names are chosen so **people** can remember them (ASCII/Unicode strings); inside, the system uses a fast, unique number. The mapping in between lets an object move without changing its name.

Name Resolution Step by Step

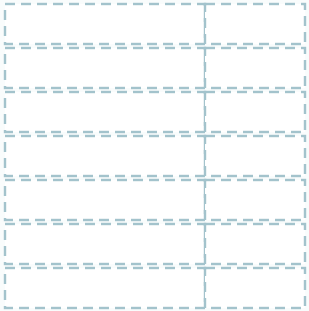
/ | usr/ | ast/ | mbox

disk reads: 0

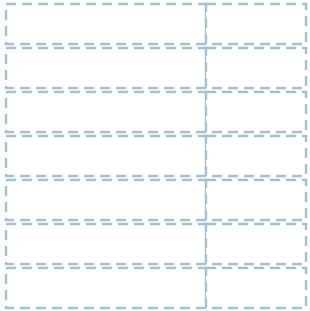
i-node 1 → root directory · block 1



i-node 6 → /usr · block 132

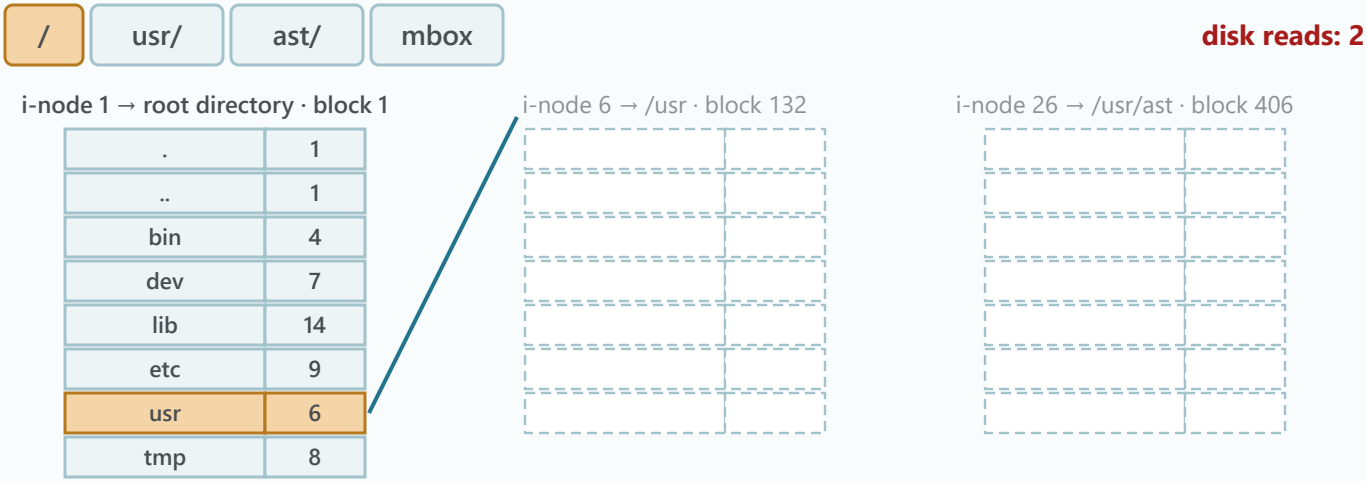


i-node 26 → /usr/ast · block 406



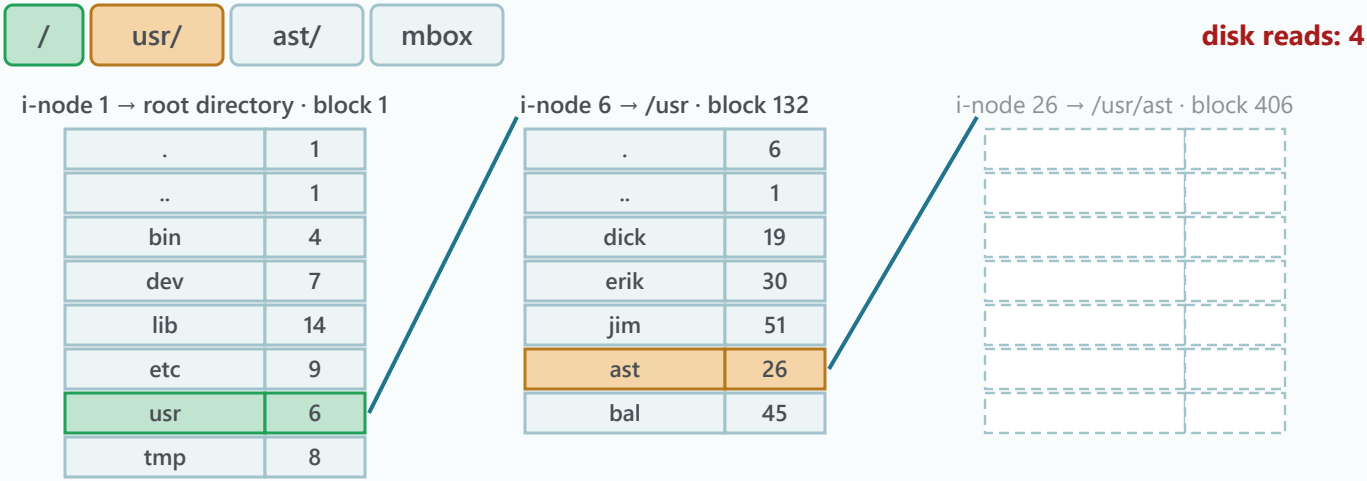
The kernel receives `open("/usr/ast/mbox")`. The path is split into components; if nothing is cached, each component needs disk reads.

Name Resolution Step by Step



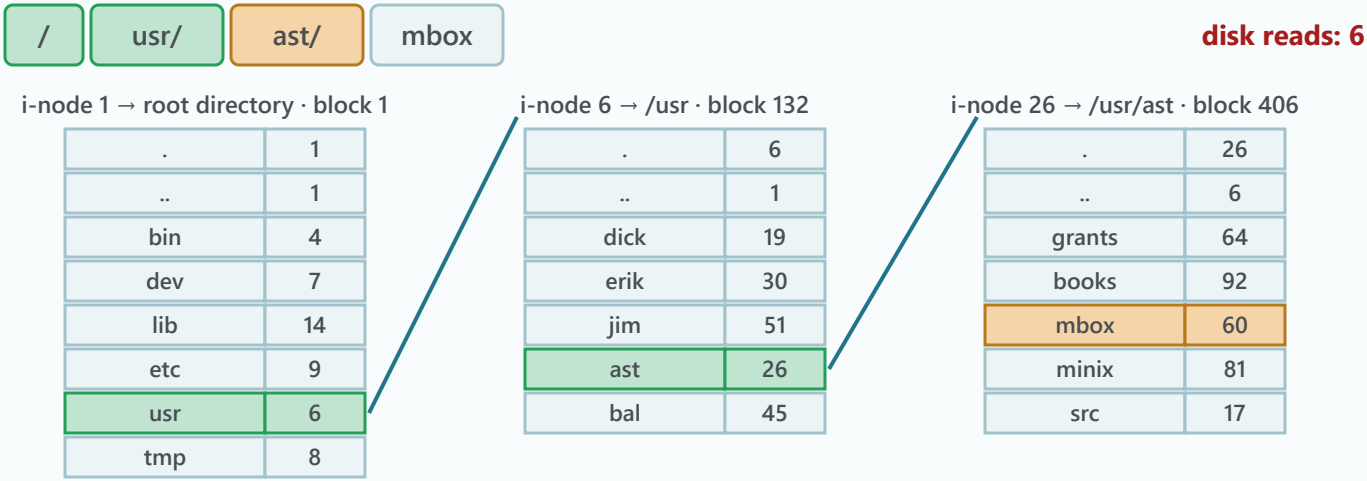
The root directory's i-node (i-node 1) is read, then the root directory's data (block 1). A linear search: `usr` → i-node 6. **2 reads**.

Name Resolution Step by Step



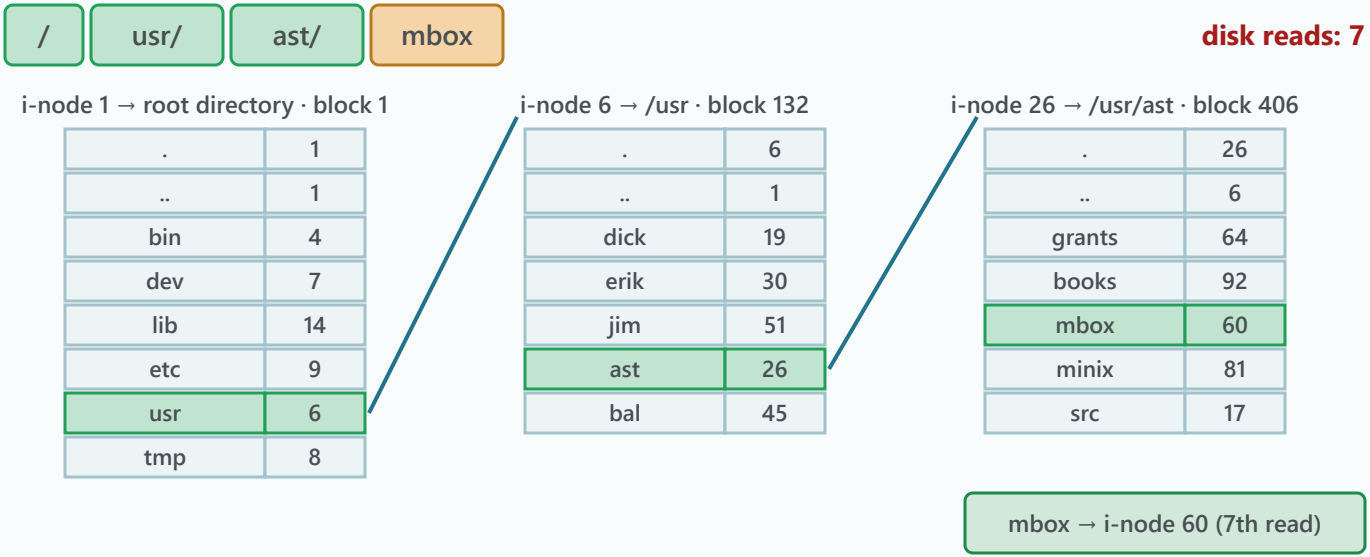
i-node 6 for /usr is read, then the /usr directory (block 132). `ast` → i-node 26. **4 reads**.

Name Resolution Step by Step



i-node 26 for /usr/ast is read, then directory block 406. `mbox` → i-node 60. **6 reads**.

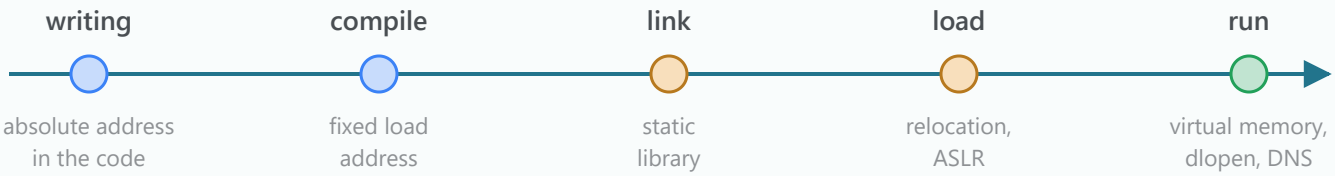
Name Resolution Step by Step



Finally the file's own i-node (60) is read. Just **finding** the file took 7 disk accesses: hence caching is essential.

DESIGN PRINCIPLES

Binding Time



Early binding

The link between a name and an object is fixed in advance. **Simple** and fast, but **less flexible**: the program runs only at a given address and must be recompiled when a library changes.

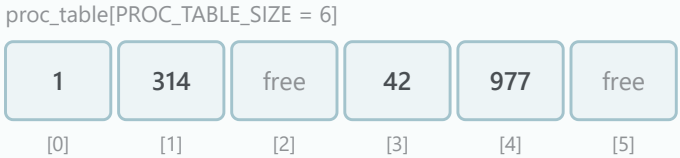
Late binding

The link is made at the last possible moment. **More complex** (extra indirection, run-time cost) but **more flexible**: pages can load anywhere, and the program is unchanged when a library is updated.

Binding time asks “at what moment is a name bound to a real object?” Operating systems generally choose late binding; virtual memory, for example, re-resolves a page's physical location on every access (with the TLB's help).

Static and Dynamic Structures

```
found = 0;
for (p = &proc_table[0];
     p < &proc_table[PROC_TABLE_SIZE];
     p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```



target pid = 42

A static process table: six entries, some free (pid 0). We look for pid 42. `found` is cleared.

Step 1 / 5

VARIABLES

`pid = 42` `found = 0` `p = –`

A **static** table is simple and needs no allocator, but when it fills no new process can be created (`NPROC` in old UNIX). A **dynamic** structure (linked list, hash table, Linux's xarray) is flexible, and lookups in a hash table or xarray are near O(1), but it brings an allocator, fragmentation and complex locking. A middle ground: tunable table sizes (`/proc/sys/kernel/pid_max`).

Static and Dynamic Structures

```
1 found = 0;
▶ for (p = &proc_table[0];
3       p < &proc_table[PROC_TABLE_SIZE];
4       p++) {
▶       if (p->proc_pid == pid) {
6           found = 1;
7           break;
8       }
9 }
```



target pid = 42

The pointer starts at the head of the table. `proc_table[0]` has pid 1: no match.

Step 2 / 5

VARIABLES

`pid = 42` `found = 0` `p = &proc_table[0]`

A **static** table is simple and needs no allocator, but when it fills no new process can be created (`NPROC` in old UNIX). A **dynamic** structure (linked list, hash table, Linux's xarray) is flexible, and lookups in a hash table or xarray are near O(1), but it brings an allocator, fragmentation and complex locking. A middle ground: tunable table sizes (`/proc/sys/kernel/pid_max`).

Static and Dynamic Structures

```
1 found = 0;
2 for (p = &proc_table[0];
3      p < &proc_table[PROC_TABLE_SIZE];
▶      p++) {
▶     if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }
```



`p++` advances to the next **structure** (not the next byte). 314 ≠ 42.

Step 3 / 5

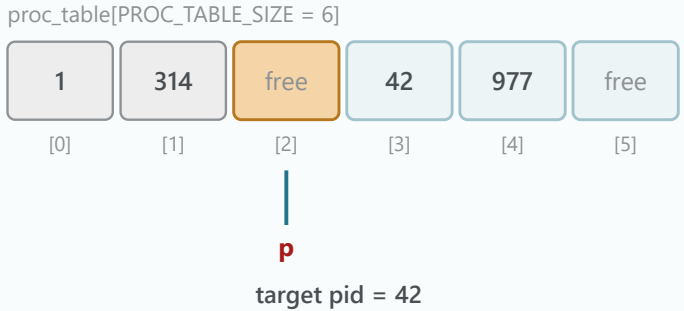
VARIABLES

`pid = 42` `found = 0` `p = &proc_table[1]`

A **static** table is simple and needs no allocator, but when it fills no new process can be created (`NPROC` in old UNIX). A **dynamic** structure (linked list, hash table, Linux's xarray) is flexible, and lookups in a hash table or xarray are near O(1), but it brings an allocator, fragmentation and complex locking. A middle ground: tunable table sizes (`/proc/sys/kernel/pid_max`).

Static and Dynamic Structures

```
1 found = 0;
2 for (p = &proc_table[0];
3      p < &proc_table[PROC_TABLE_SIZE];
      p++) {
      if (p->proc_pid == pid) {
6         found = 1;
7         break;
8     }
9 }
```



Free entries (pid 0) are scanned one by one too: the price of linear search.

Step 4 / 5

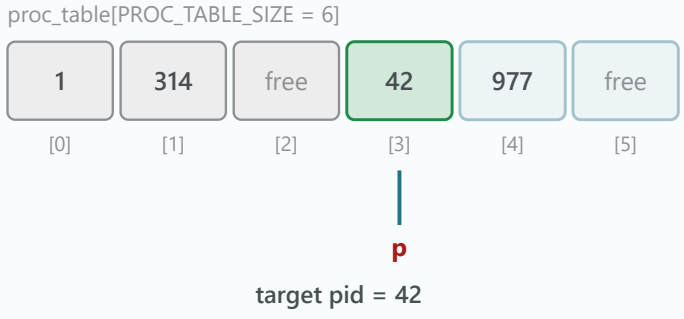
VARIABLES

`pid = 42` `found = 0` `p = &proc_table[2]`

A **static** table is simple and needs no allocator, but when it fills no new process can be created (`NPROC` in old UNIX). A **dynamic** structure (linked list, hash table, Linux's xarray) is flexible, and lookups in a hash table or xarray are near O(1), but it brings an allocator, fragmentation and complex locking. A middle ground: tunable table sizes (`/proc/sys/kernel/pid_max`).

Static and Dynamic Structures

```
1 found = 0;
2 for (p = &proc_table[0];
3      p < &proc_table[PROC_TABLE_SIZE];
4      p++) {
      if (p->proc_pid == pid) {
          found = 1;
          break;
8     }
9 }
```



A match: `found = 1`, and `break` leaves the loop. In the worst case the whole table is scanned: O(n). For a small, fixed table this simple code is enough.

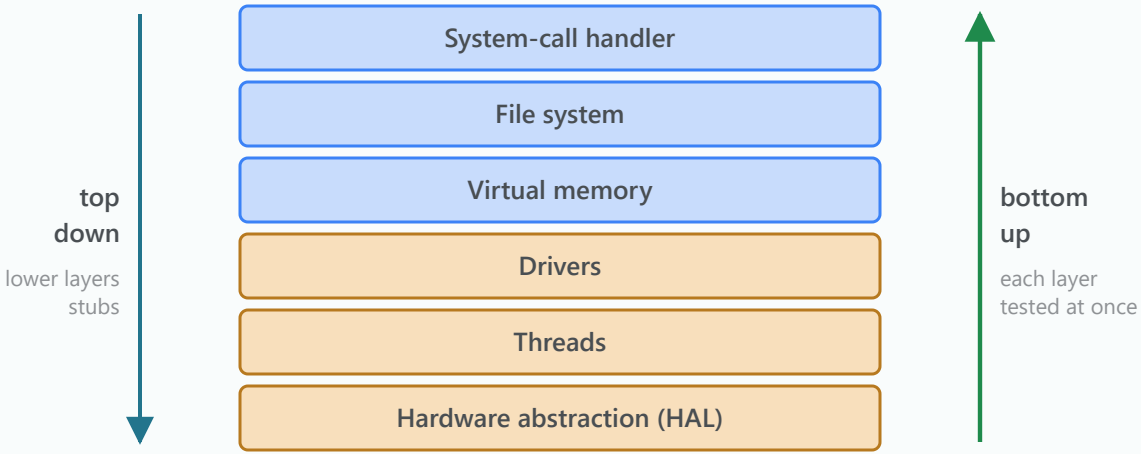
Step 5 / 5

VARIABLES

`pid = 42` `found = 1` `p = &proc_table[3]`

A **static** table is simple and needs no allocator, but when it fills no new process can be created (`NPROC` in old UNIX). A **dynamic** structure (linked list, hash table, Linux's xarray) is flexible, and lookups in a hash table or xarray are near O(1), but it brings an allocator, fragmentation and complex locking. A middle ground: tunable table sizes (`/proc/sys/kernel/pid_max`).

Top-Down and Bottom-Up Implementation



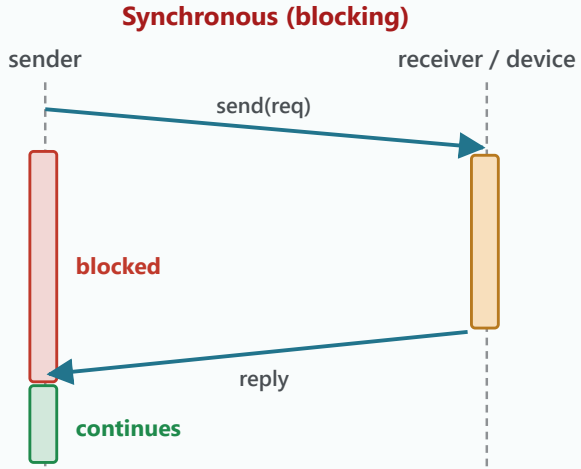
Top-down

System-call handlers are written first and the lower layers are left as **stubs**. The interface firms up early, but realistic testing is hard until the whole system exists.

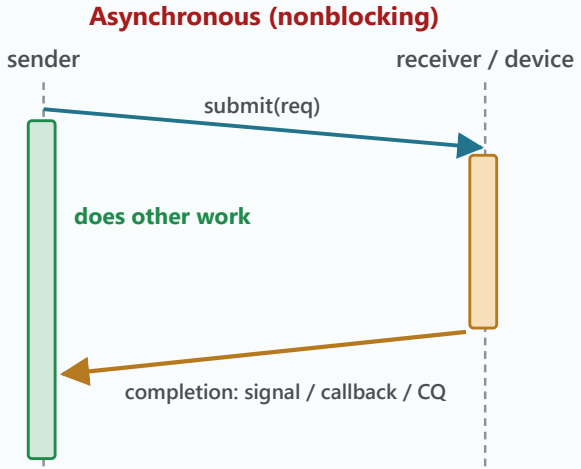
Bottom-up

First the layer that hides the hardware, then interrupts, the timer, threads… Each layer is **tested** as it is finished. Tanenbaum recommends this route for implementation, while design should be thought out top-down.

Synchronous and Asynchronous Communication



- **Synchronous**: easy to program, the flow reads sequentially; but a thread waits for each request.
- In microkernels IPC is usually synchronous (L4, seL4): no buffering needed, very fast.



- **Asynchronous**: one thread can drive thousands of requests; but races, error handling and buffering get harder.
- Linux `io_uring`: submission (SQ) and completion (CQ) rings; requests can be submitted even without a system call.

Hiding the Hardware

Conditional compilation by CPU type

```
#include "config.h"
void init(void)
{
#if (CPU == X86_64)
    /* x86-64 init code */
#endif
#if (CPU == ARM64)
    /* ARM64 init code */
#endif
#if (CPU == RISCV64)
    /* RISC-V init code */
#endif
}
```

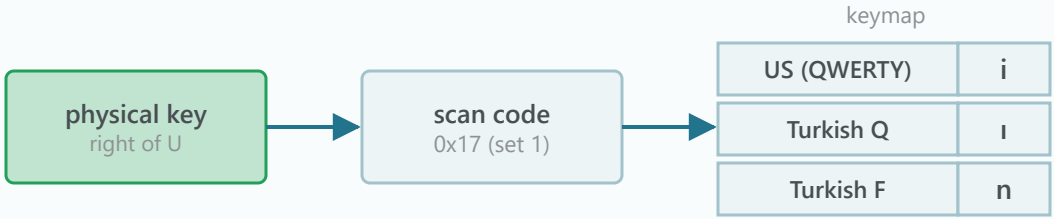
Conditional compilation by word length

```
#include "config.h"
#include <stdint.h>
#if (WORD_LENGTH == 32)
typedef int Register;
#elif (WORD_LENGTH == 64)
typedef long Register;
#else  /* today: portable */
typedef uintptr_t Register;
#endif
Register R0, R1, R2, R3;
```

Collecting hardware-dependent code in a few files is the key to portability: the **HAL** (Hardware Abstraction Layer) in Windows, the `arch/x86`, `arch/arm64` and `arch/riscv` directories in Linux. Porting to a new processor means rewriting only that part. Use conditional compilation sparingly: `#ifdef` s scattered everywhere make code unreadable.

The code updates Tanenbaum's Pentium/UltraSPARC example.

Indirection

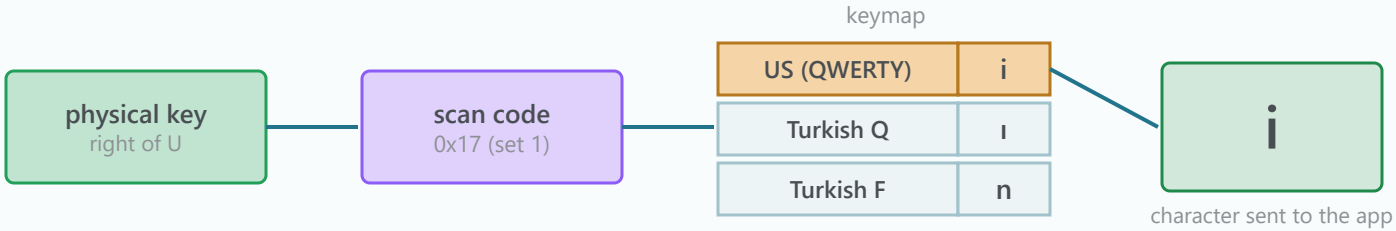


The user presses the key to the right of U. The keyboard reports the key's **position**, not a character.

Step 1 / 4

“All problems in computer science can be solved by another level of indirection.” (David Wheeler). In operating systems: virtual memory (page tables), file descriptors, symbolic links, i-nodes, device classes.

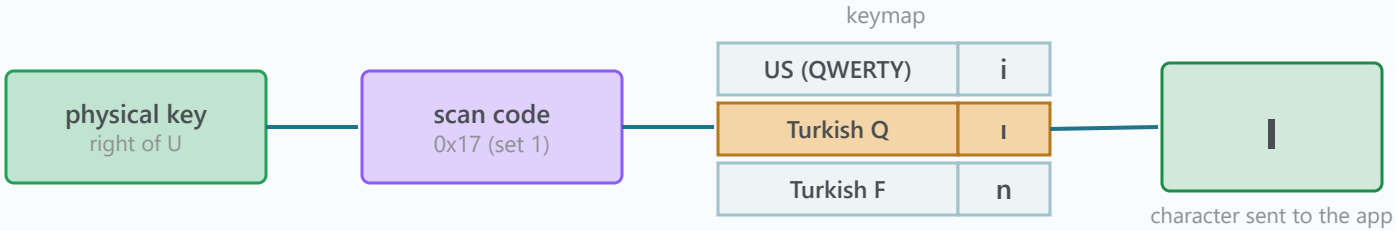
Indirection



The keyboard sends scan code 0x17 (USB keyboards send a HID usage code, which the driver maps to a common code). With the US keymap it becomes `i`.

“All problems in computer science can be solved by another level of indirection.” (David Wheeler). In operating systems: virtual memory (page tables), file descriptors, symbolic links, i-nodes, device classes.

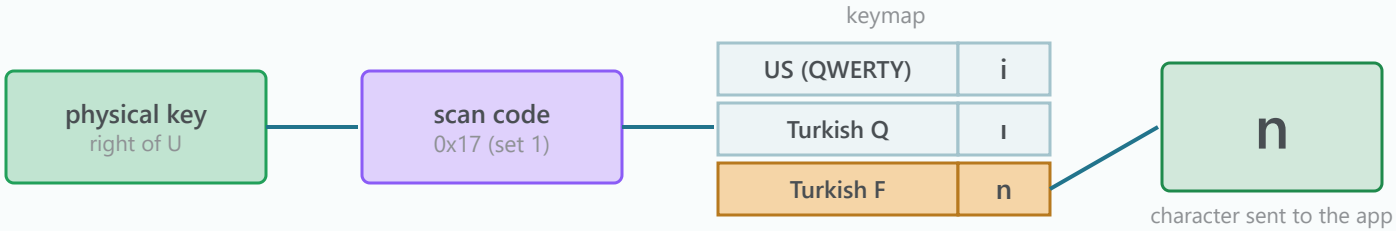
Indirection



Only the **table** is swapped for Turkish Q: the same key now produces `ı`. Neither hardware nor application changed.

“All problems in computer science can be solved by another level of indirection.” (David Wheeler). In operating systems: virtual memory (page tables), file descriptors, symbolic links, i-nodes, device classes.

Indirection



With the Turkish F keymap the same key gives `n`. One extra step (indirection) buys flexibility; the cost is a table lookup.

Step 4 / 4

“All problems in computer science can be solved by another level of indirection.” (David Wheeler). In operating systems: virtual memory (page tables), file descriptors, symbolic links, i-nodes, device classes.

Reusability and Reentrancy

Reusability

Using the same code in different places: a single **bitmap** module can track free disk blocks, memory frames and PIDs. In Linux: the shared `list.h`, red-black trees, xarray.

Reentrancy

A piece of code must be able to run **twice** at the same time: an interrupt may call the same function while it executes, or another core runs the same code. The rule: no shared static variables, and locks where needed.

```
char *strtok(char *s, const char *d)
{
    static char *last; /* shared state! */
    ...
}

/* reentrant version: state lives in the caller */
char *strtok_r(char *s, const char *d,
               char **saveptr);
```

The `static` variable inside `strtok` breaks when two threads, or an interrupt handler, use it at the same time.

strtok and Reentrancy: Code Trace

```
▶ char *my_strtok(char *s, char sep) {
▶     static char *rest;      /* shared by all! */
3     if (s) rest = s;
4     if (!rest || !*rest) return NULL;
5     char *tok = rest;
6     while (*rest && *rest != sep) rest++;
7     if (*rest) *rest++ = '\0';
8     return tok;
9 }
10 /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```

a

a	l	i	,	v	e	l	i	\0

static rest: **NULL** main
a = "ali,veli"

The main program is splitting comma-separated names. The function's state lives in one static pointer: rest records where the next call should resume.

Step 1 / 6

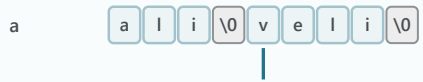
VARIABLES

`rest = NULL`

OUTPUT

strtok and Reentrancy: Code Trace

```
1  char *my_strtok(char *s, char sep) {
2      static char *rest;      /* shared by all! */
▶      if (s) rest = s;
4      if (!rest || !*rest) return NULL;
▶      char *tok = rest;
▶      while (*rest && *rest != sep) rest++;
▶      if (*rest) *rest++ = '\0';
▶      return tok;
9  }
10 /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```



static rest: **a[4] (v)** main

call 1 → "ali"

First call: rest = a. It advances to the comma, writes \0 over it, and rest points to the start of “veli”. Returns **ali**.

Step 2 / 6

VARIABLES

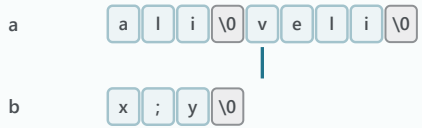
`s = a` `rest = a+4` `ret = ali`

OUTPUT

```
main: ali
```

strtok and Reentrancy: Code Trace

```
1  char *my_strtok(char *s, char sep) {
2      static char *rest;      /* shared by all! */
3      if (s) rest = s;
4      if (!rest || !*rest) return NULL;
5      char *tok = rest;
6      while (*rest && *rest != sep) rest++;
7      if (*rest) *rest++ = '\0';
8      return tok;
9  }
▶ /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```



static rest: **a[4] (v)** interrupt arrives
b = "x;y"

At exactly this moment an interrupt arrives; the handler starts splitting its own string (b = “x;y”) with the same function.

Step 3 / 6

VARIABLES

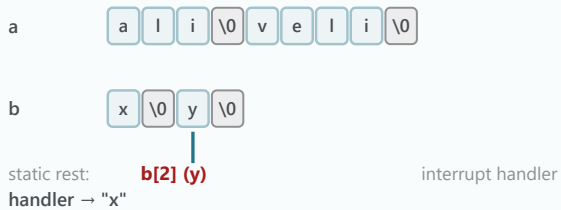
`rest = a+4`

OUTPUT

```
main: ali
```

strtok and Reentrancy: Code Trace

```
1  char *my_strtok(char *s, char sep) {
2      static char *rest;      /* shared by all! */
▶      if (s) rest = s;
4      if (!rest || !*rest) return NULL;
▶      char *tok = rest;
▶      while (*rest && *rest != sep) rest++;
▶      if (*rest) *rest++ = '\0';
▶      return tok;
9  }
10 /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```



The handler's call with s = b overwrites the static **rest**: it now points inside b (“y”). main's place is lost.

Step 4 / 6

VARIABLES

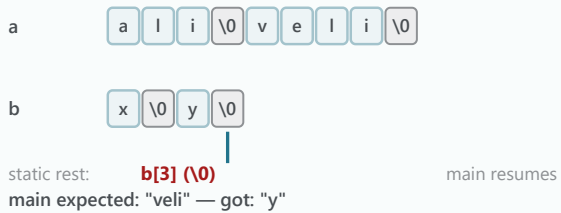
`s = b` `rest = b+2` `ret = x`

OUTPUT

```
main: ali
irq: x
```

strtok and Reentrancy: Code Trace

```
1  char *my_strtok(char *s, char sep) {
2      static char *rest;      /* shared by all! */
▶      if (s) rest = s;
▶      if (!rest || !*rest) return NULL;
▶      char *tok = rest;
▶      while (*rest && *rest != sep) rest++;
7      if (*rest) *rest++ = '\0';
▶      return tok;
9  }
10 /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```



main passes NULL to resume, but rest points into b: it returns **y**, not the expected “veli”. The bug shows up rarely and depends on timing.

Step 5 / 6

VARIABLES

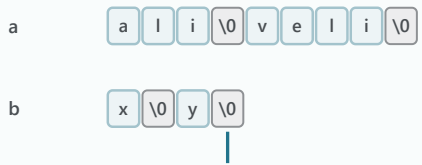
`s = NULL` `rest = b+3` `ret = y`

OUTPUT

```
main: ali
irq: x
main: y  (wrong)
```

strtok and Reentrancy: Code Trace

```
1  char *my_strtok(char *s, char sep) {
▶      static char *rest;      /* shared by all! */
3      if (s) rest = s;
4      if (!rest || !*rest) return NULL;
5      char *tok = rest;
6      while (*rest && *rest != sep) rest++;
7      if (*rest) *rest++ = '\0';
8      return tok;
9  }
10 /* main:  my_strtok(a, ',')  a = "ali,veli" */
11 /* irq:   my_strtok(b, ';')  b = "x;y"      */
12 /* main:  my_strtok(NULL, ',')              */
```



Root cause: the state between calls lives in a static (shared) variable. The reentrant strtok_r keeps it in a pointer the caller supplies, so every caller has its own rest.

Step 6 / 6

VARIABLES

`rest = b+3`

OUTPUT

```
main: ali
irq: x
main: y  (wrong)
```

Brute Force and Checking for Errors First

Brute force

- The simple, "slow-looking" solution is often enough: a linear search of a 50-entry table can be **faster and less buggy** than a hash table.
- Use a complex algorithm only when measurements demand it: when n is large or the code is on a hot path.
- Sorting: insertion sort for small arrays, merge sort/heapsort for large ones.

Check for errors first

Bad: checking after acquiring resources

```
lock(&tbl_lock);
buf = kmalloc(n, GFP_KERNEL);
if (n > MAX || !access_ok(u, n)) {
    kfree(buf);
    unlock(&tbl_lock);
    return -EINVAL;
}
```

Good: check first, then acquire

```
if (n > MAX || !access_ok(u, n))
    return -EINVAL; /* check first */
buf = kmalloc(n, GFP_KERNEL);
if (!buf)
    return -ENOMEM;
lock(&tbl_lock);
...
```

Validating parameters before taking locks and memory removes the undo code on the error path (and forgotten-`unlock` bugs).

Performance: Why Slow, and What to Speed Up?

Why are systems slow?

Feature bloat: every release adds features and cannot remove old ones. Backward-compatibility layers, security measures (Spectre/Meltdown patches raised system-call costs, for example), indirection added for generality.

What should be optimized?

Only the parts **measurements** show to be bottlenecks. “Good enough is good enough”: doubling code complexity for a 5% gain is rarely worth it.

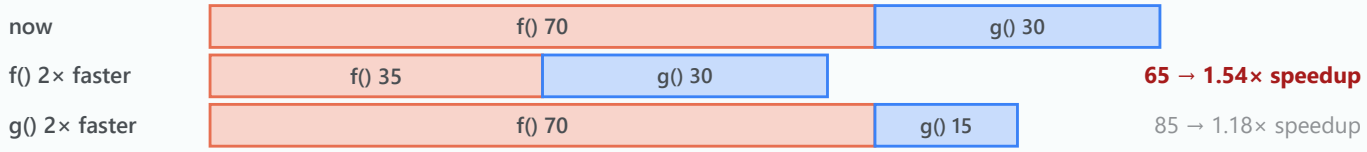
First correct, then fast

“Premature optimization is the root of all evil.” (D. Knuth). Measurement tools: `perf`, eBPF-based `bpftrace`, flame graphs.

Classic techniques

Space–time trade-offs · caching · hints · exploiting locality · optimizing the common case.

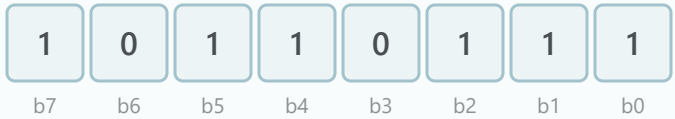
Hardware got thousands of times faster over the decades, but perceived speed did not rise as much: software growth ate much of the gain.



Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
▶ int bit_count(int byte)
3 {
▶     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 0

The sample byte is 0xB7 (183). The loop tests each bit in turn; the counter starts at zero.

Step 1 / 9

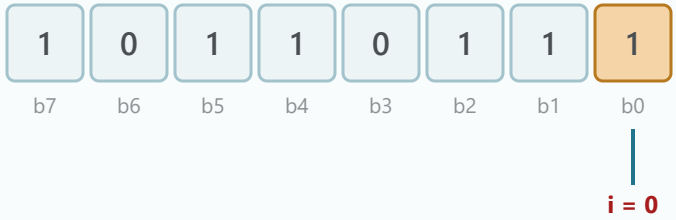
VARIABLES

`byte = 0xB7` `i = –` `count = 0`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
▶             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 1

`(byte >> 0) & 1` = 1: the counter goes up.

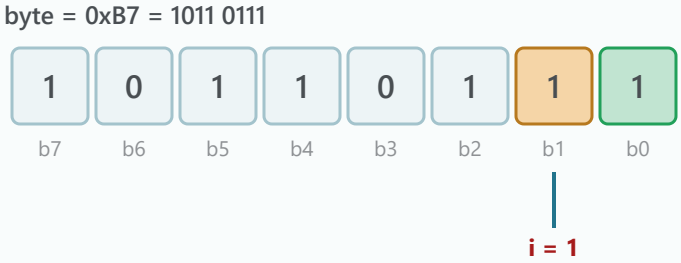
Step 2 / 9

VARIABLES

`byte = 0xB7` `i = 0` `count = 1`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
▶             count++;
8     return count;
9 }
```



count = 2

Shift, mask, compare: a few instructions and a branch per bit.

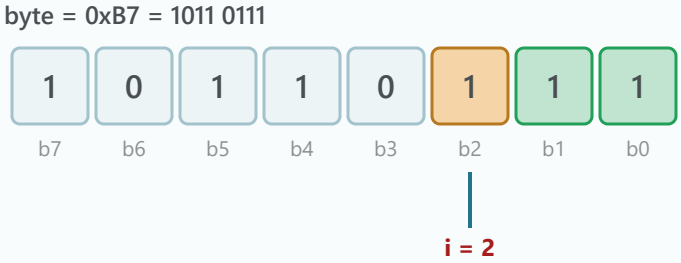
Step 3 / 9

VARIABLES

byte = 0xB7 | i = 1 | count = 2

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
▶             count++;
8     return count;
9 }
```



Bit 2 is 1 too.

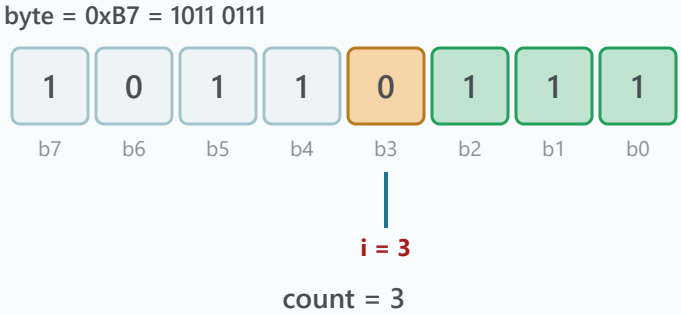
Step 4 / 9

VARIABLES

`byte = 0xB7` `i = 2` `count = 3`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```



count = 3

Bit 3 is zero: the branch is not taken and the counter stays.

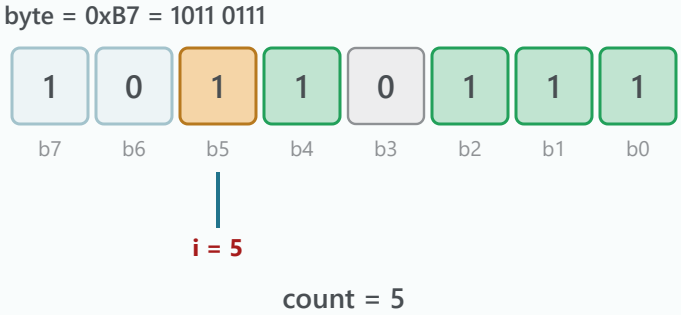
Step 5 / 9

VARIABLES

`byte = 0xB7` `i = 3` `count = 3`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
▶             count++;
8     return count;
9 }
```



Bits 4 and 5 are 1 too.

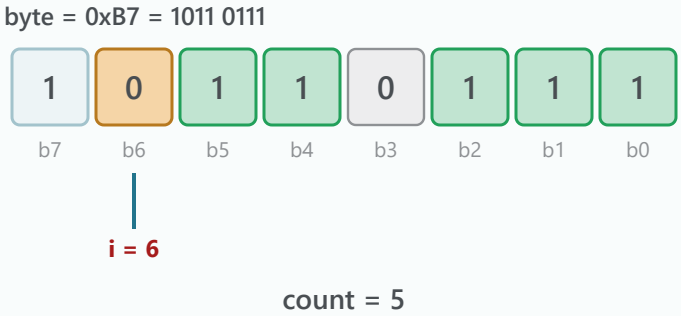
Step 6 / 9

VARIABLES

`byte = 0xB7` `i = 5` `count = 5`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
7             count++;
8     return count;
9 }
```



Bit 6 is zero.

Step 7 / 9

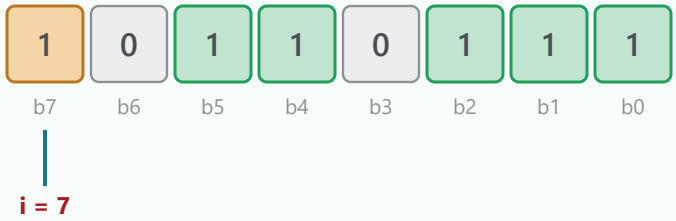
VARIABLES

`byte = 0xB7` `i = 6` `count = 5`

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
▶     for (i = 0; i < BYTE_SIZE; i++)
▶         if ((byte >> i) & 1)
▶             count++;
8     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 6

The last bit is 1. The loop ran 8 times.

Step 8 / 9

VARIABLES

byte = 0xB7 | i = 7 | count = 6

Space–Time Trade-offs: (a) Counting Bits with a Loop

```
1 #define BYTE_SIZE 8
2 int bit_count(int byte)
3 {
4     int i, count = 0;
5     for (i = 0; i < BYTE_SIZE; i++)
6         if ((byte >> i) & 1)
7             count++;
▶     return count;
9 }
```

byte = 0xB7 = 1011 0111



count = 6

The result is 6. Almost no memory, but about 8 iterations × a few instructions per byte. To go faster we must **spend space to save time**.

Step 9 / 9

VARIABLES

`byte = 0xB7` `i = 8` `count = 6`

Space–Time Trade-offs: (b) Macro and (c) Table

```
/* (b) macro: no loop */
#define bit_count(b) ((b&1) + ((b>>1)&1) + \
    ((b>>2)&1) + ... + ((b>>7)&1))

/* (c) table: 256 bytes */
char bits[256] =
    {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, ...};
#define bit_count(b) (int) bits[b]
```

Method	Memory	Work per byte
(a) loop	~0	8 iterations: shift, mask, branch
(b) macro	~0	~24 instructions, no branch
(c) 256-entry table	256 B	1 memory read
16-entry table (nibble)	16 B	2 reads + an add
POPCNT instruction	0	1 instruction (`__builtin_popcount`)

bits[0..31] (computed in Python: bin(i).count("1"))

0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	2	3	2	3	3	4	2	3	3	4	3	4	4	5
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

bits[0xB7] = bits[183] = 6

Counting Bits with a Nibble Table: Code Trace

```
1 char nib[16];            /* 16 bytes */
2 void build_table(void) {
▶     nib[0] = 0;
4     for (int i = 1; i < 16; i++)
5         nib[i] = (i & 1) + nib[i >> 1];
6 }
7 int bit_cnt(unsigned char b) {
8     return nib[b & 0x0F] + nib[b >> 4];
9 }
```



A 16-byte table instead of a 256-byte one: one count for each 4-bit value. 0 has no bits set.

Step 1 / 7

VARIABLES

`nib[0] = 0`

OUTPUT

Counting Bits with a Nibble Table: Code Trace

```
1  char nib[16];            /* 16 bytes */
2  void build_table(void) {
3      nib[0] = 0;
▶      for (int i = 1; i < 16; i++)
▶          nib[i] = (i & 1) + nib[i >> 1];
6  }
7  int bit_cnt(unsigned char b) {
8      return nib[b & 0x0F] + nib[b >> 4];
9  }
```



i = 1 … 3: the bit count of i = its last bit + the bit count of (i ≫ 1), which is always filled in already. E.g. nib[3] = 1 + nib[1] = **2**.

Step 2 / 7

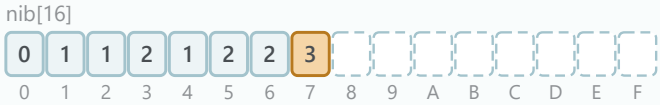
VARIABLES

`i = 3` `nib[3] = 2`

OUTPUT

Counting Bits with a Nibble Table: Code Trace

```
1  char nib[16];            /* 16 bytes */
2  void build_table(void) {
3      nib[0] = 0;
▶      for (int i = 1; i < 16; i++)
▶          nib[i] = (i & 1) + nib[i >> 1];
6  }
7  int bit_cnt(unsigned char b) {
8      return nib[b & 0x0F] + nib[b >> 4];
9  }
```



i = 4 … 7: the bit count of i = its last bit + the bit count of (i ≫ 1), which is always filled in already. E.g. nib[7] = 1 + nib[3] = **3**.

Step 3 / 7

VARIABLES

`i = 7` `nib[7] = 3`

OUTPUT

Counting Bits with a Nibble Table: Code Trace

```
1  char nib[16];            /* 16 bytes */
2  void build_table(void) {
3      nib[0] = 0;
▶      for (int i = 1; i < 16; i++)
▶          nib[i] = (i & 1) + nib[i >> 1];
6  }
7  int bit_cnt(unsigned char b) {
8      return nib[b & 0x0F] + nib[b >> 4];
9  }
```

nib[16]

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

i = 8 … 15: the bit count of i = its last bit + the bit count of (i ≫ 1), which is always filled in already. E.g. nib[15] = 1 + nib[7] = **4**.

Step 4 / 7

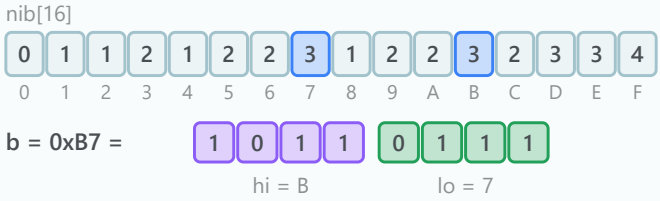
VARIABLES

`i = 15` `nib[15] = 4`

OUTPUT

Counting Bits with a Nibble Table: Code Trace

```
1 char nib[16];            /* 16 bytes */
2 void build_table(void) {
3     nib[0] = 0;
4     for (int i = 1; i < 16; i++)
5         nib[i] = (i & 1) + nib[i >> 1];
6 }
▶ int bit_cnt(unsigned char b) {
▶     return nib[b & 0x0F] + nib[b >> 4];
9 }
```



bit_cnt(0xB7): the byte splits into two halves: low 7 = 0111, high B = 1011. No loop, no branch.

Step 5 / 7

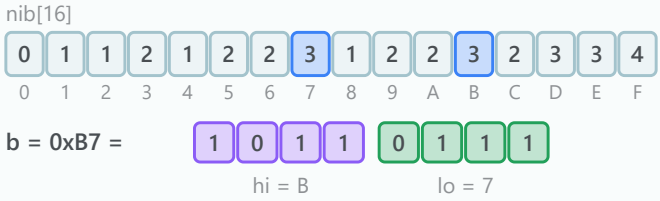
VARIABLES

`b = 0xB7` `lo = 7` `hi = B`

OUTPUT

Counting Bits with a Nibble Table: Code Trace

```
1 char nib[16];            /* 16 bytes */
2 void build_table(void) {
3     nib[0] = 0;
4     for (int i = 1; i < 16; i++)
5         nib[i] = (i & 1) + nib[i >> 1];
6 }
7 int bit_cnt(unsigned char b) {
▶     return nib[b & 0x0F] + nib[b >> 4];
9 }
```



nib[7] + nib[B] = 3 + 3 = 6

Two memory reads and an add: 3 + 3 = **6**. The loop version needed 8 iterations for the same byte.

Step 6 / 7

VARIABLES

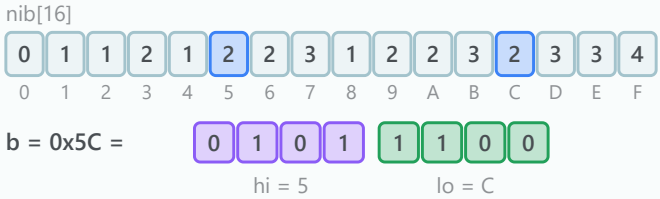
`b = 0xB7` `nib[7] = 3` `nib[B] = 3`

OUTPUT

```
bit_cnt(0xB7) = 6
```

Counting Bits with a Nibble Table: Code Trace

```
1 char nib[16];            /* 16 bytes */
2 void build_table(void) {
3     nib[0] = 0;
4     for (int i = 1; i < 16; i++)
5         nib[i] = (i & 1) + nib[i >> 1];
6 }
▶ int bit_cnt(unsigned char b) {
▶     return nib[b & 0x0F] + nib[b >> 4];
9 }
```



nib[C] + nib[5] = 2 + 2 = 4

bit_cnt(0x5C): nib[C] + nib[5] = 2 + 2 = **4**. 16 bytes of memory, 2 reads: a middle point between the 256-entry table and the loop.

Step 7 / 7

VARIABLES

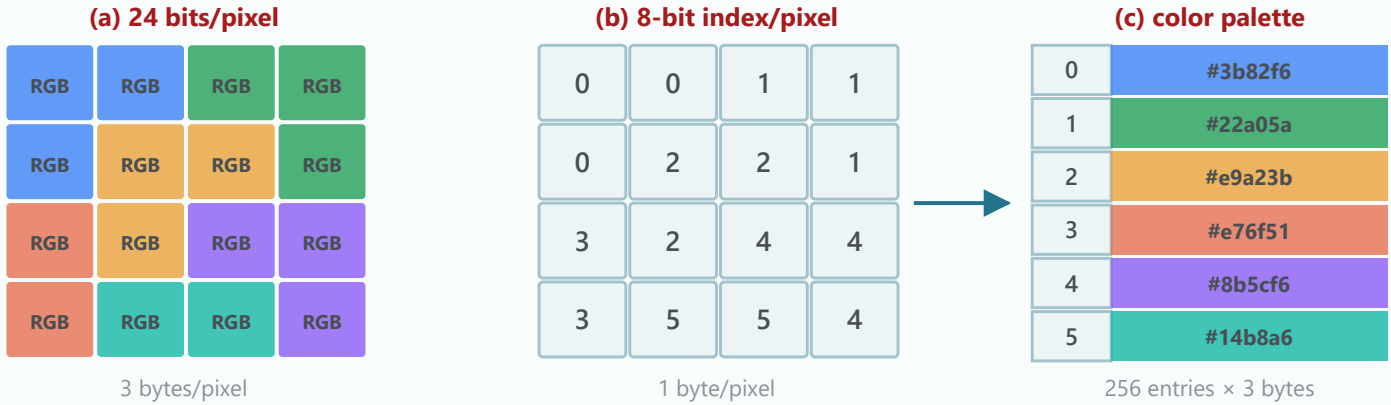
`b = 0x5C` `nib[C] = 2` `nib[5] = 2`

OUTPUT

```
bit_cnt(0xB7) = 6
bit_cnt(0x5C) = 4
```

PERFORMANCE

Space–Time Trade-offs: A Color Palette



1024 × 768 image	Size (bytes)	Ratio
(a) 24 bits/pixel	2,359,296 (2.25 MiB)	1
(b)+(c) 8-bit index + palette	786.432 + 768 = 787.200	≈ 1/3

Space drops about threefold; the price is a **table lookup** (time) whenever a pixel is displayed, and a limit of 256 colors. GIF and PNG-8 use this method (GIF also compresses with LZW). Sizes were computed in Python.

PERFORMANCE

Caching

Path → i-node cache

Path	i-node
`/usr`	6
`/usr/ast`	26
`/usr/ast/mbox`	60
`/usr/ast/books`	92
`/usr/bal`	45
`/usr/bal/paper.ps`	85

- Without a cache, finding `/usr/ast/mbox` takes: the root i-node, the root directory, the /usr i-node, the /usr directory, the /usr/ast i-node, the /usr/ast directory → **6 disk reads** (+ the file's i-node).
- If the full path is cached: **one memory lookup**, zero disk accesses.
- If only `/usr/ast` is cached, the search continues from there: a partial hit is a gain too.
- Linux: the directory-entry cache (**dcache**), the i-node cache and the page cache. That is why free RAM looks "used".
- The price of caching is consistency: if a file is deleted or renamed, its entry must be invalidated.

General rule: if the same result is computed again and again, **store it**. Caches in an OS: the TLB, CPU caches, the page cache, the dcache, ARP and DNS caches.

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
▶      int k = n, ino = ROOT, reads = 0;
▶      while (k > 0 && !in_cache(c, k, &ino))
▶          k--;                /* longest prefix */
6      for (int j = k; j < n; j++) {
7          ino = dir_lookup(ino, c[j]);
8          reads += 2;         /* i-node + block */
9          cache_add(c, j + 1, ino);
10     }
11     return ino;
12 }
```

path: usr | bal | thesis.tex

disk reads: 0

cache (path → i-node)

/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/bal	45
/usr/bal/paper.ps	85

/usr/bal/thesis.tex not cached

The cache holds five paths. We look up /usr/bal/thesis.tex: the full path is not cached.

Step 1 / 7

VARIABLES

`k = 3` `ino = ROOT` `reads = 0`

OUTPUT

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
▶      while (k > 0 && !in_cache(c, k, &ino))
▶          k--;              /* longest prefix */
6      for (int j = k; j < n; j++) {
7          ino = dir_lookup(ino, c[j]);
8          reads += 2;       /* i-node + block */
9          cache_add(c, j + 1, ino);
10     }
11     return ino;
12 }
```

path: usr | bal | thesis.tex

disk reads: 0

cache (path → i-node)

/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/bal	45
/usr/bal/paper.ps	85

partial hit: /usr/bal → 45

Drop the last component: /usr/bal is cached, ino = **45**. No need to start from the root: two components for free.

Step 2 / 7

VARIABLES

`k = 2` `ino = 45` `reads = 0`

OUTPUT

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
4      while (k > 0 && !in_cache(c, k, &ino))
5          k--;              /* longest prefix */
▶      for (int j = k; j < n; j++) {
▶          ino = dir_lookup(ino, c[j]);
▶          reads += 2;       /* i-node + block */
▶          cache_add(c, j + 1, ino);
10     }
11     return ino;
12 }
```

path: usr | bal | thesis.tex

disk reads: 2

cache (path → i-node)

/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/bal	45
/usr/bal/paper.ps	85
/usr/bal/thesis.tex	131

from disk: /usr/bal directory → thesis.tex

The one remaining component comes from disk: i-node 45 and its directory block are read, thesis.tex → **131**. The result is added to the cache.

Step 3 / 7

VARIABLES

`j = 2` `ino = 131` `reads = 2`

OUTPUT

```
/usr/bal/thesis.tex → 131 (2 reads)
```

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
▶      while (k > 0 && !in_cache(c, k, &ino))
5          k--;                /* longest prefix */
6      for (int j = k; j < n; j++) {
7          ino = dir_lookup(ino, c[j]);
8          reads += 2;         /* i-node + block */
9          cache_add(c, j + 1, ino);
10     }
▶      return ino;
12 }
```

path: usr | bal | thesis.tex

disk reads: 0

cache (path → i-node)

/usr	6	/usr/bal/thesis.tex	131
/usr/ast	26		
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

full hit: no disk access at all

When the same path is opened again the full path is cached: neither loop runs, **0** disk reads.

Step 4 / 7

VARIABLES

`k = 3` `ino = 131` `reads = 0`

OUTPUT

```
/usr/bal/thesis.tex → 131 (2 reads)
/usr/bal/thesis.tex → 131 (0 reads)
```

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
▶      while (k > 0 && !in_cache(c, k, &ino))
▶          k--;              /* longest prefix */
6      for (int j = k; j < n; j++) {
7          ino = dir_lookup(ino, c[j]);
8          reads += 2;       /* i-node + block */
9          cache_add(c, j + 1, ino);
10     }
11     return ino;
12 }
```

path: etc | passwd

disk reads: 0

cache (path → i-node)

/usr	6	/usr/bal/thesis.tex	131
/usr/ast	26		
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

neither /etc/passwd nor /etc is cached

/etc/passwd: neither the full path nor /etc is cached; k drops to 0 and resolution starts at the root i-node (ROOT = 1).

Step 5 / 7

VARIABLES

`k = 0` `ino = ROOT` `reads = 0`

OUTPUT

```
/usr/bal/thesis.tex → 131 (2 reads)
/usr/bal/thesis.tex → 131 (0 reads)
```

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
4      while (k > 0 && !in_cache(c, k, &ino))
5          k--;              /* longest prefix */
▶      for (int j = k; j < n; j++) {
▶          ino = dir_lookup(ino, c[j]);
▶          reads += 2;       /* i-node + block */
▶          cache_add(c, j + 1, ino);
10     }
11     return ino;
12 }
```

path: etc | passwd

disk reads: 2

cache (path → i-node)

/usr	6	/usr/bal/thesis.tex	131
/usr/ast	26	/etc	14
/usr/ast/mbox	60		
/usr/bal	45		
/usr/bal/paper.ps	85		

etc in the root directory → 14

The root i-node and root directory block are read: etc → **14**; /etc enters the cache.

Step 6 / 7

VARIABLES

`j = 0` `ino = 14` `reads = 2`

OUTPUT

```
/usr/bal/thesis.tex → 131 (2 reads)
/usr/bal/thesis.tex → 131 (0 reads)
```

Name Resolution with a Path Cache: Code Trace

```
1  /* c[]: "usr","bal","thesis.tex"; n = 3 */
2  int resolve(char *c[], int n) {
3      int k = n, ino = ROOT, reads = 0;
4      while (k > 0 && !in_cache(c, k, &ino))
5          k--;              /* longest prefix */
▶      for (int j = k; j < n; j++) {
▶          ino = dir_lookup(ino, c[j]);
▶          reads += 2;       /* i-node + block */
▶          cache_add(c, j + 1, ino);
10     }
▶      return ino;
12 }
```

path: etc | passwd

disk reads: 4

cache (path → i-node)

/usr	6	/usr/bal/thesis.tex	131
/usr/ast	26	/etc	14
/usr/ast/mbox	60	/etc/passwd	212
/usr/bal	45		
/usr/bal/paper.ps	85		

passwd in the etc directory → 212

The /etc i-node and directory block: passwd → **212**. On the uncached path, 2 reads per component: 4 in all. In Linux this cache is the dcache.

Step 7 / 7

VARIABLES

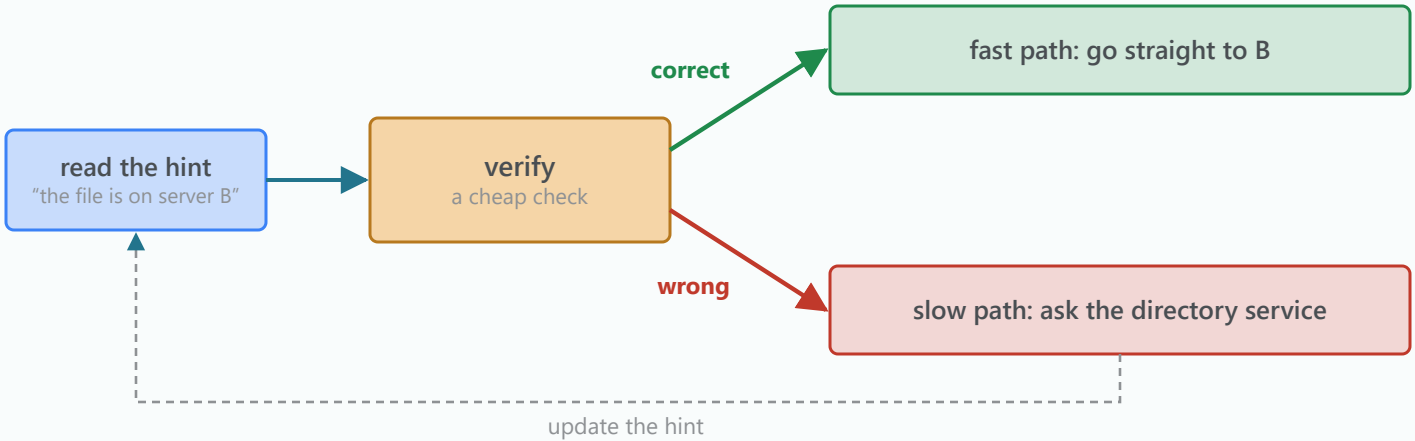
`j = 1` `ino = 212` `reads = 4`

OUTPUT

```
/usr/bal/thesis.tex → 131 (2 reads)
/usr/bal/thesis.tex → 131 (0 reads)
/etc/passwd → 212 (4 reads)
```

PERFORMANCE

Hints



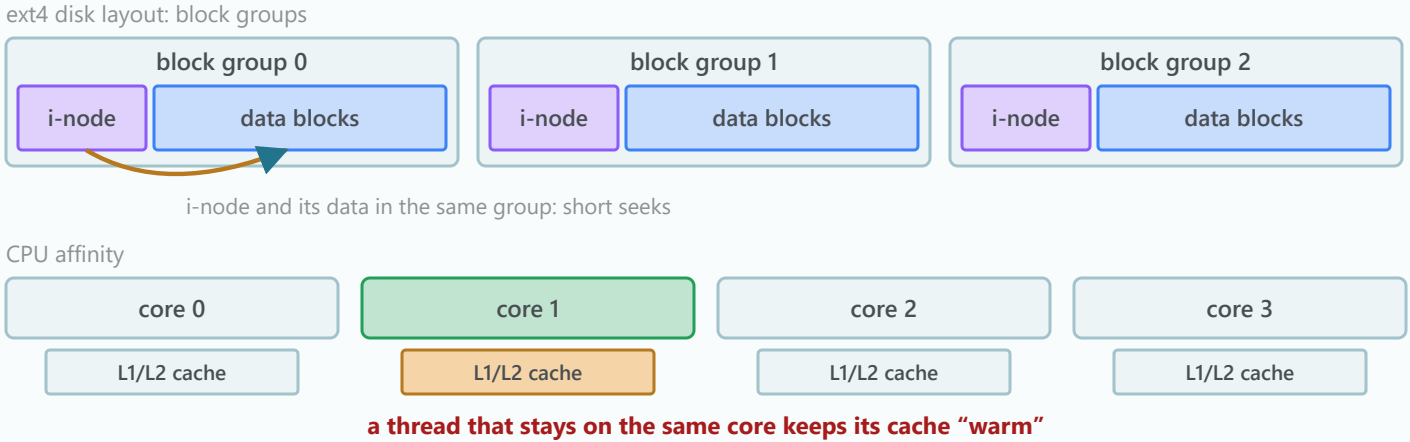
How it differs from a cache

A cached value must always be correct. **A hint may be wrong**, so no invalidation is needed, but a cheap check before use is a must.

Examples

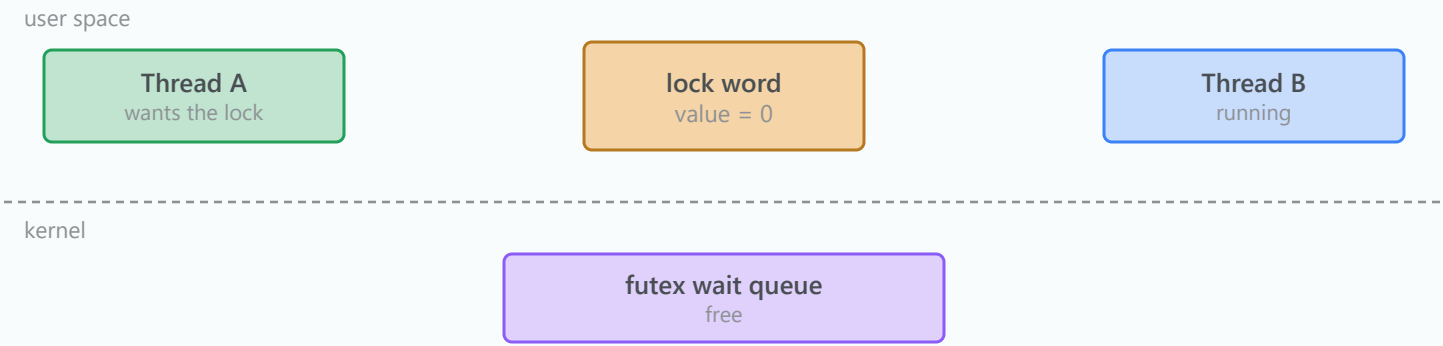
"The file is probably on server B" in a distributed file system; redirect caches on the web; branch prediction; an access hint from an application to the kernel via `madvise(MADV_SEQUENTIAL)`.

Exploiting Locality



- **Temporal locality**: what was used recently will be used again (the basis of caching). **Spatial locality**: what is nearby will be used too (read-ahead, huge pages).
- Put data used together side by side: FFS cylinder groups, ext4 block groups; on NUMA, allocate memory from the thread's own node.

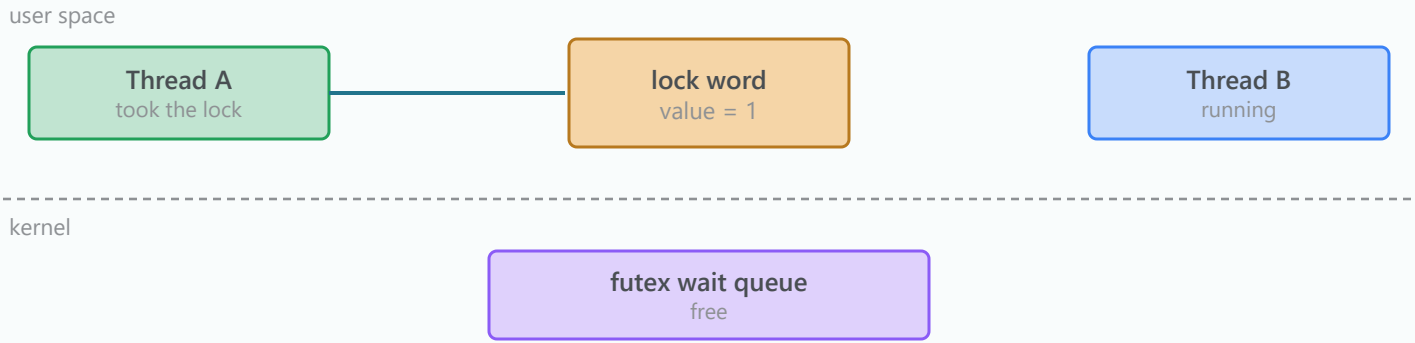
Optimizing the Common Case



Most locks are taken **uncontended**. If that is the common case, it is the case to speed up.

The general principle (Lampson): **make the common case fast and the rare case correct**. Other examples: system-call fast paths, `clock_gettime` via the vDSO without entering the kernel, TLB hits.

Optimizing the Common Case

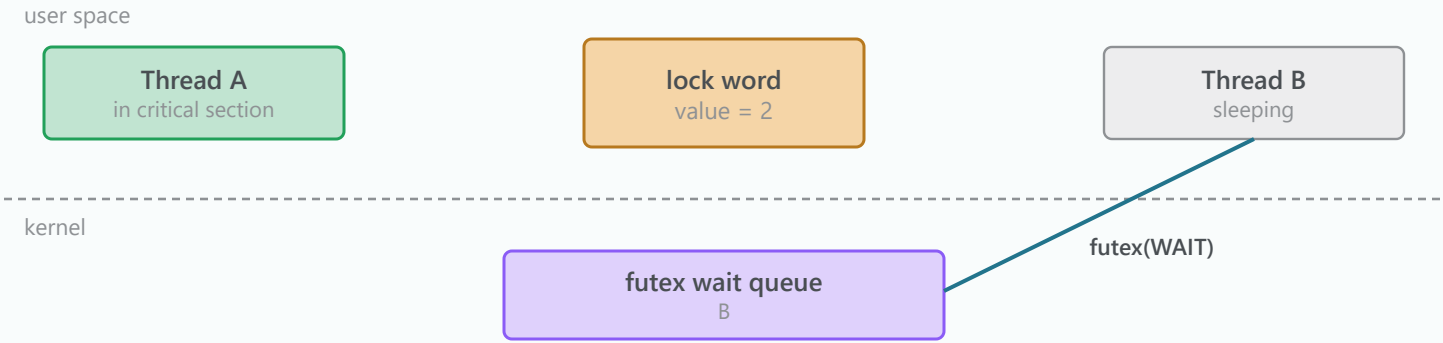


CAS 0→1 succeeded: no system call

A uses an atomic compare-and-swap (CAS) to turn 0 into 1. **The kernel is never entered**: a few nanoseconds.

The general principle (Lampson): **make the common case fast and the rare case correct**. Other examples: system-call fast paths, `clock_gettime` via the vDSO without entering the kernel, TLB hits.

Optimizing the Common Case

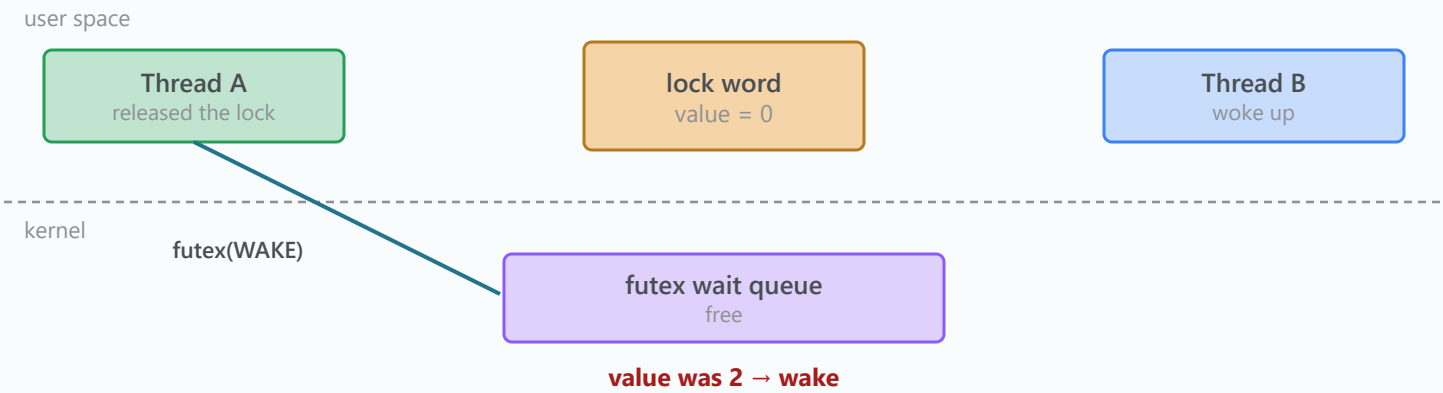


CAS failed → 2 "waiters exist"

The rare case: B also wants the lock and its CAS fails. It sets the value to 2 and sleeps in the kernel with `futex(FUTEX_WAIT)`. Only here is the slow path taken.

The general principle (Lampson): **make the common case fast and the rare case correct**. Other examples: system-call fast paths, `clock_gettime` via the vDSO without entering the kernel, TLB hits.

Optimizing the Common Case



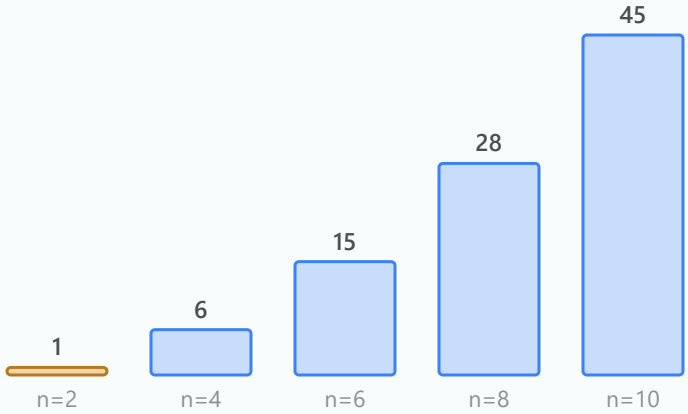
On release, if the old value was 2, A calls `futex(FUTEX_WAKE)`; had it been 1, A would not call it at all. B wakes and takes the lock.

The general principle (Lampson): **make the common case fast and the rare case correct**. Other examples: system-call fast paths, `clock_gettime` via the vDSO without entering the kernel, TLB hits.

The Mythical Man-Month



n = 2 people → 1 communication path



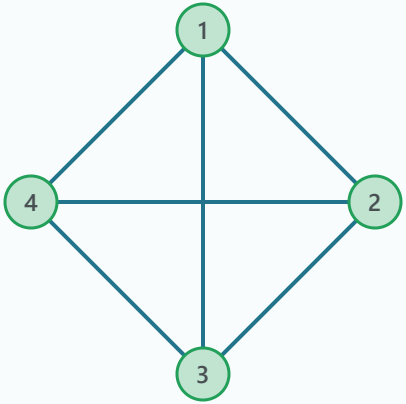
paths = n(n−1)/2

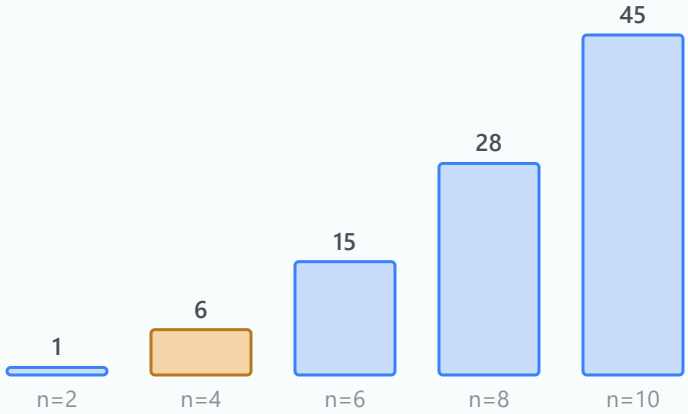
The “man-month” is a misleading unit: it assumes work can be divided among people. Two people have just **1** communication path.

Step 1 / 4

Brooks's time split from his OS/360 experience: 1/3 planning, 1/6 coding, 1/4 module testing, 1/4 system testing. Writing code is only one-sixth of the job.

The Mythical Man-Month



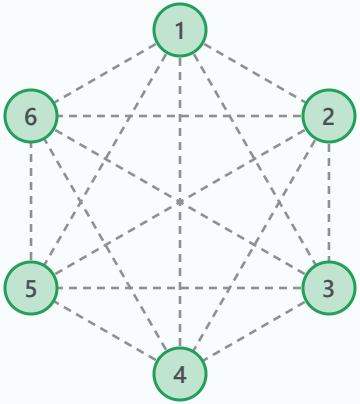


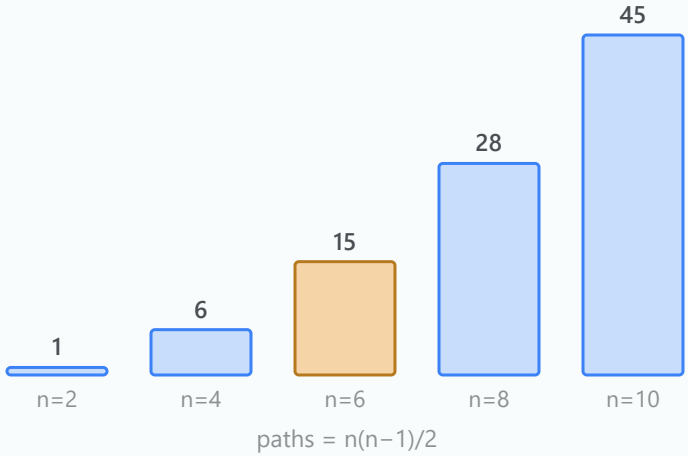
n = 4 people → 6 communication paths

4 people have **6** paths. Every newcomer must coordinate with everyone; meetings, documents and interface agreements multiply.

Brooks's time split from his OS/360 experience: 1/3 planning, 1/6 coding, 1/4 module testing, 1/4 system testing. Writing code is only one-sixth of the job.

The Mythical Man-Month



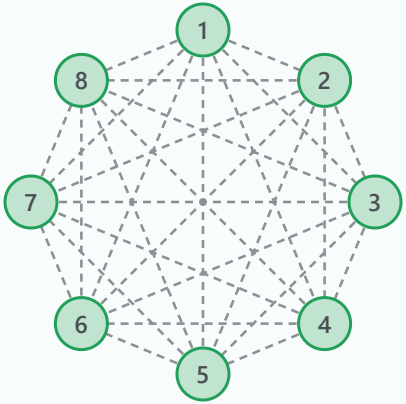


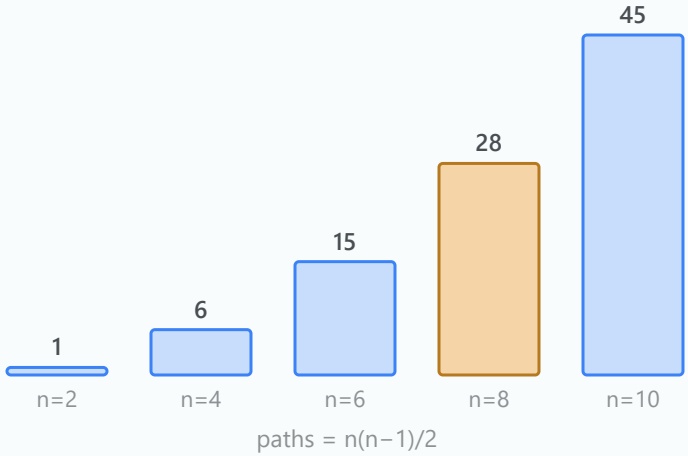
n = 6 people → 15 communication paths

6 people have **15** paths. Training someone added to a late project also eats the existing team's time.

Brooks's time split from his OS/360 experience: 1/3 planning, 1/6 coding, 1/4 module testing, 1/4 system testing. Writing code is only one-sixth of the job.

The Mythical Man-Month



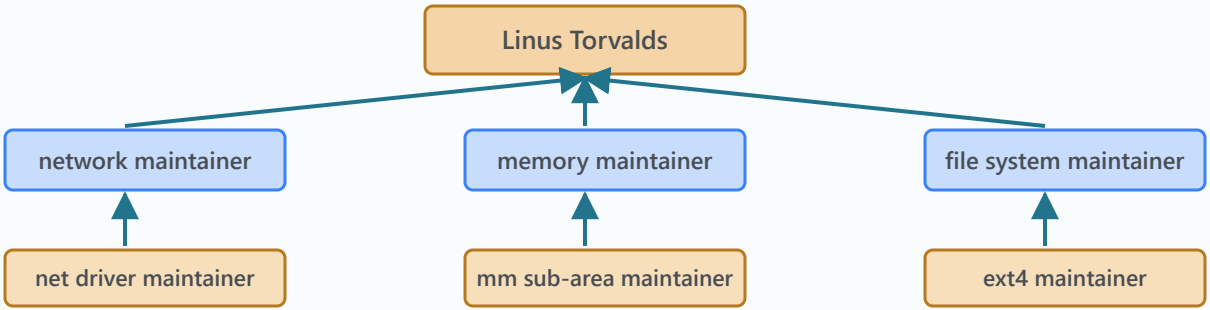


n = 8 people → 28 communication paths

8 people have **28** paths. **Brooks's law**: “Adding manpower to a late software project makes it later.”

Brooks's time split from his OS/360 experience: 1/3 planning, 1/6 coding, 1/4 module testing, 1/4 system testing. Writing code is only one-sixth of the job.

Team Structure, Experience and “No Silver Bullet”



patches from thousands of contributors flow upward (~4–5k developers a year)

Team structure

Mills/Brooks's **surgical team**: the chief programmer designs and codes, the others support. Conceptual integrity stays with a few people.

The role of experience

Good programmers are many times more productive than average. **“Plan to throw one away”**; beware the second-system effect.

No silver bullet

Brooks (1986): tools reduce only **accidental** complexity; **essential** complexity remains. AI-assisted coding does not remove design responsibility either.

Why Is Testing OS Code Hard?

Why is it hard?

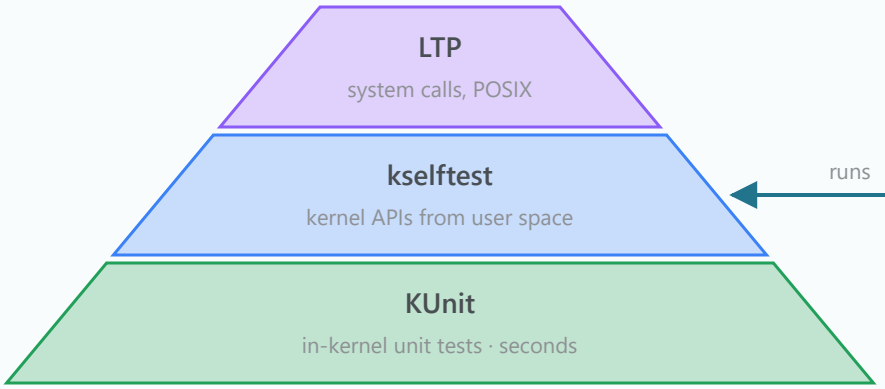
- **Privileged mode:** there is no higher layer to catch the error; one bad pointer in the kernel brings down **the whole system**, test harness included, with an oops/panic.
- **Concurrency:** interrupts, preemption and multiple cores create countless interleavings; a bug may show up only under a rare timing.
- **Nondeterminism:** "heisenbugs" do not repeat on every run; even adding a `printk` can change the timing and hide the bug.
- **Diversity:** thousands of drivers, several architectures (x86-64, arm64, RISC-V) and tens of thousands of Kconfig options; not every combination can be tried.
- **Long-lived state:** leaks and counter overflows appear only after days of running.

Kinds of tests

Kind	What it tests (kernel example)
Unit	A single function or module, in isolation (KUnit)
Integration	Subsystems working together, the user-space interface (kselftest)
System	The whole OS, on a real or virtual machine (LTP, boot tests)
Regression	Did a fixed bug come back? Rerun on every change (KernelCI)
Stress / load	Overload, long runs, resource scarcity (stress-ng, rcutorture)

A failed OS test often means losing the machine itself, so tests run in VMs with timeouts and automatic reboots.

The Kernel Test Pyramid



runs

KernelCI
continuous integration

- many trees and branches
- many architectures, Kconfigs
- QEMU + real board labs
- runs KUnit, kselftest, LTP
- auto-bisects regressions

similar: 0-day bot, syzbot

↑ realism and cost rise · ↓ speed and test count rise

KUnit

Tests are built with the kernel and call internal functions directly; they run in seconds under UML or QEMU: `kunit.py run`. Output is in KTAP format.

kselftest

Under `tools/testing/selftests`; tests a running kernel's system calls and interfaces such as cgroup, bpf and networking from the outside: `make kselftest`.

LTP

The Linux Test Project: thousands of tests of system-call semantics, POSIX conformance, file systems and stress scenarios. Distributions use it before releases.

Dynamic Checkers: KASAN, KCSAN, lockdep

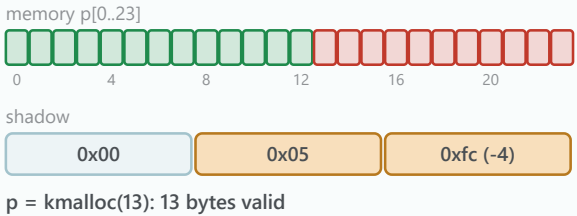
Tool	Bug it catches	How it works
KASAN `CONFIG_KASAN`	use-after-free; heap, stack and global array overflows; double free	The compiler puts a check before every memory access; **shadow memory** (1 byte per 8 bytes) records which bytes are valid. Reports include the allocation and free stack traces.
KCSAN `CONFIG_KCSAN`	Data races: concurrent, unlocked accesses where at least one is a write	Sets a **watchpoint** on random accesses and waits briefly; if another core touches the same address meanwhile, it reports.
UBSAN `CONFIG_UBSAN`	Undefined behavior: signed overflow, division by zero, invalid shifts, array bounds	Run-time checks inserted by the compiler.
lockdep `CONFIG_PROVE_LOCKING`	Lock-order violations (AB-BA), wrong lock use in interrupt context	Keeps a "before-after" graph between lock **classes**; on seeing a cycle it warns **before** a deadlock happens.
kmemleak `CONFIG_DEBUG_KMEMLEAK`	Memory leaks	Scans memory like a garbage collector; reports allocated blocks no pointer refers to.

```
BUG: KASAN: slab-use-after-free in mydrv_release+0x54/0x90
Read of size 8 at addr ffff888012a4c010 by task cat/1432
```

Cost: KASAN slows the kernel several times and devotes about 1/8 of memory to shadow; KCSAN and lockdep also add noticeable overhead. So they are enabled in **development, test and CI kernels**. In production the sampling, low-overhead **KFENCE** can be used.

KASAN Shadow-Memory Check: Code Trace

```
/* 1 shadow byte per 8-byte region:
   0: all valid, 1..7: first k bytes,
   < 0: redzone or freed */
void kasan_check(void *a, int size) {
    int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
                           + SHADOW_OFS);
    if (s == 0) return;            /* fast path */
    if ((int)((uintptr_t)a & 7) + size - 1 >= s)
        report(a, s);
}
/* p = kmalloc(13); p[3]; p[12]; p[13]; */
/* kfree(p); p[0]; */
```



p = kmalloc(13): 13 bytes valid

kmalloc(13) returns a 16-byte slot. Shadow: the first 8 bytes fully valid (0), the first **5** bytes of the next region valid (13 − 8), the rest a redzone (0xfc).

Step 1 / 6

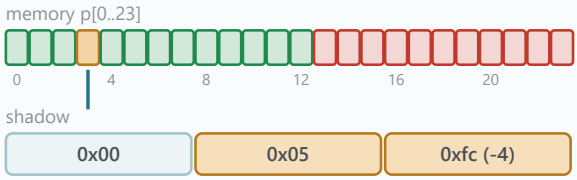
VARIABLES

`shadow = 00 05 fc`

OUTPUT

KASAN Shadow-Memory Check: Code Trace

```
1  /* 1 shadow byte per 8-byte region:
2     0: all valid, 1..7: first k bytes,
3     < 0: redzone or freed */
4  void kasan_check(void *a, int size) {
▶      int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
▶                             + SHADOW_OFS);
▶      if (s == 0) return;             /* fast path */
8      if ((int)((uintptr_t)a & 7) + size - 1 >= s)
9          report(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```



p[3]: shadow 0 → fast path

p[3]: the shadow byte for region p+3 ≫ 3 = 0 is read: 0, so the check ends with one comparison.

Step 2 / 6

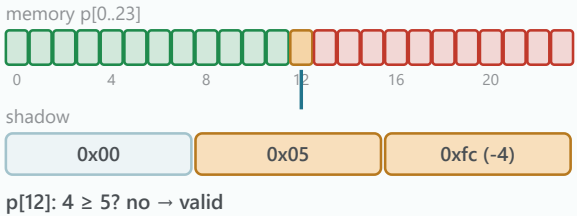
VARIABLES

`a = p+3` `s = 0`

OUTPUT

KASAN Shadow-Memory Check: Code Trace

```
1  /* 1 shadow byte per 8-byte region:
2     0: all valid, 1..7: first k bytes,
3     < 0: redzone or freed */
4  void kasan_check(void *a, int size) {
▶      int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
▶                             + SHADOW_OFS);
▶      if (s == 0) return;            /* fast path */
▶      if ((int)((uintptr_t)a & 7) + size - 1 >= s)
9          report(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```



p[12]: shadow 5; offset within the region 12 & 7 = 4. 4 ≥ 5 is false: the last valid byte, access allowed.

Step 3 / 6

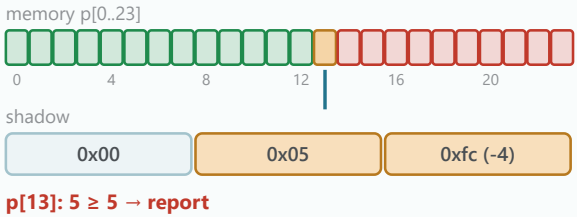
VARIABLES

`a = p+12` `s = 5` `a&7 = 4`

OUTPUT

KASAN Shadow-Memory Check: Code Trace

```
1  /* 1 shadow byte per 8-byte region:
2     0: all valid, 1..7: first k bytes,
3     < 0: redzone or freed */
4  void kasan_check(void *a, int size) {
5      int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6                             + SHADOW_OFS);
7      if (s == 0) return;              /* fast path */
▶      if ((int)((uintptr_t)a & 7) + size - 1 >= s)
▶          report(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```



p[13]: 5 ≥ 5: one byte past the array. In a real kernel this byte might belong to another object and the bug would stay silent; KASAN reports the first invalid access.

Step 4 / 6

VARIABLES

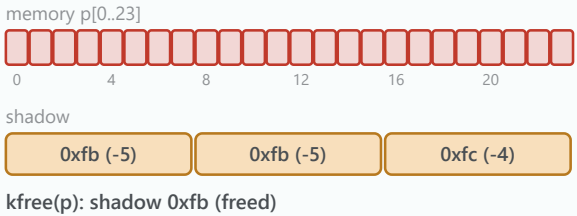
`a = p+13` `s = 5` `a&7 = 5`

OUTPUT

```
BUG: KASAN: slab-out-of-bounds p+13
```

KASAN Shadow-Memory Check: Code Trace

```
1  /* 1 shadow byte per 8-byte region:
2     0: all valid, 1..7: first k bytes,
3     < 0: redzone or freed */
4  void kasan_check(void *a, int size) {
5      int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
6                             + SHADOW_OFS);
7      if (s == 0) return;            /* fast path */
8      if ((int)((uintptr_t)a & 7) + size - 1 >= s)
9          report(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
▶ /* kfree(p); p[0]; */
```



kfree(p): the object's shadow is marked 0xfb (freed); the slot sits in quarantine for a while so it is not reused at once, which would hide the bug.

Step 5 / 6

VARIABLES

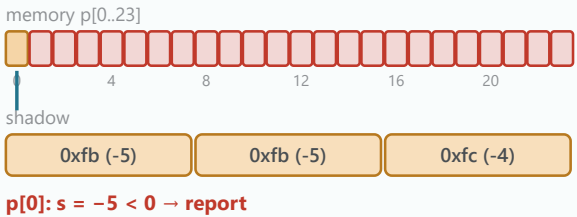
`shadow = fb fb fc`

OUTPUT

```
BUG: KASAN: slab-out-of-bounds p+13
```

KASAN Shadow-Memory Check: Code Trace

```
1  /* 1 shadow byte per 8-byte region:
2     0: all valid, 1..7: first k bytes,
3     < 0: redzone or freed */
4  void kasan_check(void *a, int size) {
▶      int8_t s = *(int8_t *)(((uintptr_t)a >> 3)
▶                             + SHADOW_OFS);
▶      if (s == 0) return;             /* fast path */
▶      if ((int)((uintptr_t)a & 7) + size - 1 >= s)
▶          report(a, s);
10 }
11 /* p = kmalloc(13); p[3]; p[12]; p[13]; */
12 /* kfree(p); p[0]; */
```



p[0]: s = −5. Since the `(int)` comparison is signed, a negative shadow satisfies "≥ s" at any offset: a **use-after-free** report, with the allocation and free stack traces.

Step 6 / 6

VARIABLES

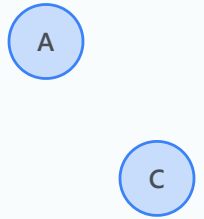
`a = p+0` `s = −5`

OUTPUT

```
BUG: KASAN: slab-out-of-bounds p+13
BUG: KASAN: use-after-free p+0
```

lockdep's Lock-Order Graph: Code Trace

```
1  void lock_acquire(task *t, int L) {
2      for (int i = 0; i < t->n; i++) {
3          int H = t->held[i];
4          if (path_exists(L, H))    /* L → … → H */
5              warn("possible deadlock", H, L);
6          add_edge(H, L);     /* H before L */
7      }
8      t->held[t->n++] = L;
9  }
10 void lock_release(task *t) { t->n--; }
▶  /* T1: A,B   T2: B,C   T3: C,A */
```



lockdep tracks lock classes, not instances, and accumulates the relation “this lock was taken while that one was held” in a graph. The graph starts empty.

Step 1 / 7

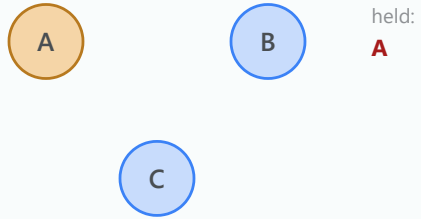
VARIABLES

`edges = 0`

OUTPUT

lockdep's Lock-Order Graph: Code Trace

```
▶ void lock_acquire(task *t, int L) {
2     for (int i = 0; i < t->n; i++) {
3         int H = t->held[i];
4         if (path_exists(L, H))    /* L → … → H */
5             warn("possible deadlock", H, L);
6         add_edge(H, L);     /* H before L */
7     }
▶     t->held[t->n++] = L;
9 }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



T1 takes A: no other lock is held, so the loop does not run; A goes onto the held list.

Step 2 / 7

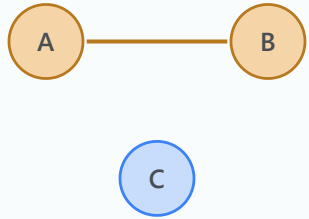
VARIABLES

`task = T1` `L = A` `n = 1`

OUTPUT

lockdep's Lock-Order Graph: Code Trace

```
1  void lock_acquire(task *t, int L) {
▶      for (int i = 0; i < t->n; i++) {
▶          int H = t->held[i];
▶          if (path_exists(L, H))   /* L → … → H */
5              warn("possible deadlock", H, L);
▶          add_edge(H, L);    /* H before L */
7      }
▶      t->held[t->n++] = L;
9  }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



held:
A, B

T1 takes B while holding A: no path from B to A, no warning. Edge **A → B** is added. Then both locks are released.

Step 3 / 7

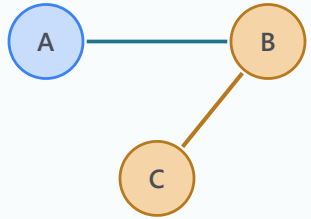
VARIABLES

`task` = **T1** `L` = **B** `H` = **A**

OUTPUT

lockdep's Lock-Order Graph: Code Trace

```
1  void lock_acquire(task *t, int L) {
▶      for (int i = 0; i < t->n; i++) {
▶          int H = t->held[i];
▶          if (path_exists(L, H))   /* L → … → H */
5              warn("possible deadlock", H, L);
▶          add_edge(H, L);    /* H before L */
7      }
▶      t->held[t->n++] = L;
9  }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



held:
B, C

Later T2 takes C while holding B: edge **B → C**. The graph now has the order A → B → C.

Step 4 / 7

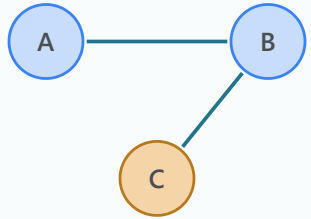
VARIABLES

`task = T2` `L = C` `H = B`

OUTPUT

lockdep's Lock-Order Graph: Code Trace

```
▶ void lock_acquire(task *t, int L) {
2     for (int i = 0; i < t->n; i++) {
3         int H = t->held[i];
4         if (path_exists(L, H))    /* L → … → H */
5             warn("possible deadlock", H, L);
6         add_edge(H, L);     /* H before L */
7     }
▶     t->held[t->n++] = L;
9 }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



held:
C

At another time T3 first takes C; a single lock, no problem.

Step 5 / 7

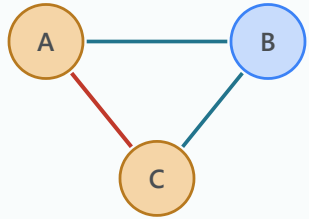
VARIABLES

`task = T3` `L = C` `n = 1`

OUTPUT

lockdep's Lock-Order Graph: Code Trace

```
1  void lock_acquire(task *t, int L) {
▶      for (int i = 0; i < t->n; i++) {
▶          int H = t->held[i];
▶          if (path_exists(L, H))   /* L → … → H */
▶              warn("possible deadlock", H, L);
6          add_edge(H, L);    /* H before L */
7      }
8      t->held[t->n++] = L;
9  }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



held:
C, A

A → B → C → A: a cycle!

T3 wants A while holding C: there is a path from A to C (A → B → C). The new edge C → A closes a **cycle**: lockdep warns.

Step 6 / 7

VARIABLES

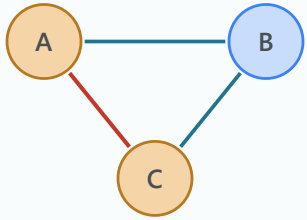
`task = T3` `L = A` `H = C`

OUTPUT

```
WARNING: possible circular locking
dependency
  C → A, but A → B → C
```

lockdep's Lock-Order Graph: Code Trace

```
1  void lock_acquire(task *t, int L) {
2      for (int i = 0; i < t->n; i++) {
3          int H = t->held[i];
4          if (path_exists(L, H))    /* L → … → H */
▶              warn("possible deadlock", H, L);
▶          add_edge(H, L);     /* H before L */
7      }
8      t->held[t->n++] = L;
9  }
10 void lock_release(task *t) { t->n--; }
11 /* T1: A,B   T2: B,C   T3: C,A */
```



held:
C, A

warning: before any deadlock

The three tasks never ran at the same time, and no real deadlock happened. But if T1, T2 and T3 ever run together, each waits for the next one's lock. lockdep shows this in a single run, before it happens.

Step 7 / 7

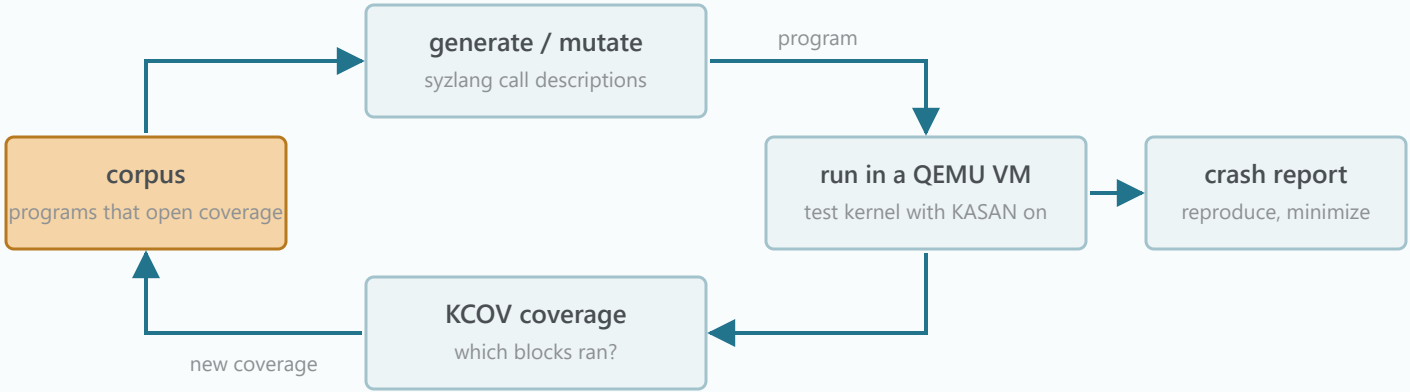
VARIABLES

`task = T3` `L = A` `H = C`

OUTPUT

```
WARNING: possible circular locking
dependency
  C → A, but A → B → C
```

Fuzzing: The syzkaller Loop



syzkaller knows system-call signatures from **syzlang** descriptions: which argument is a file descriptor, which a flag, which a pointer. The corpus starts empty or with seed programs.

Step 1 / 6

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

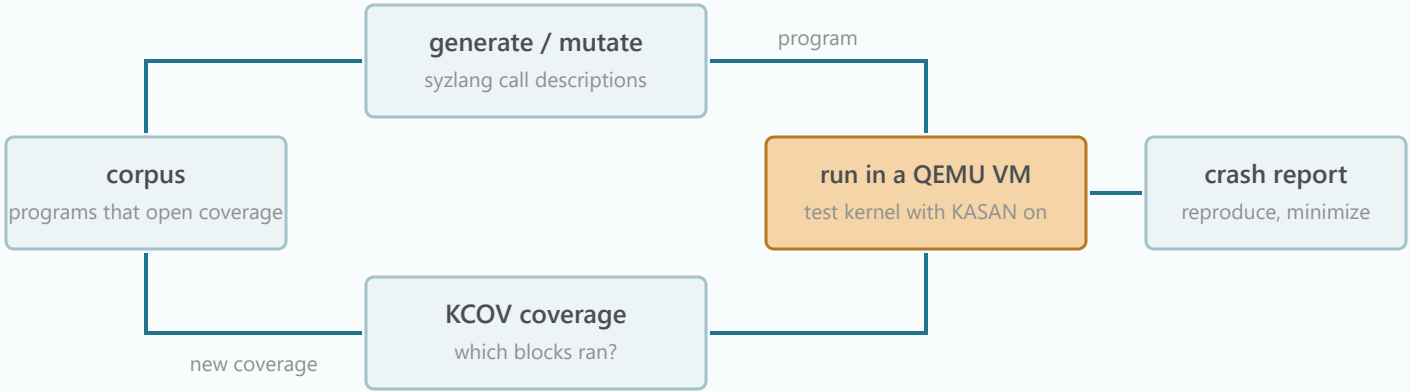
Fuzzing: The syzkaller Loop



The generator picks a program from the corpus and mutates it, or makes a new one: e.g. `openat → ioctl → mmap → close`. Arguments match their types, so calls are not rejected at once with `EINVAL`.

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

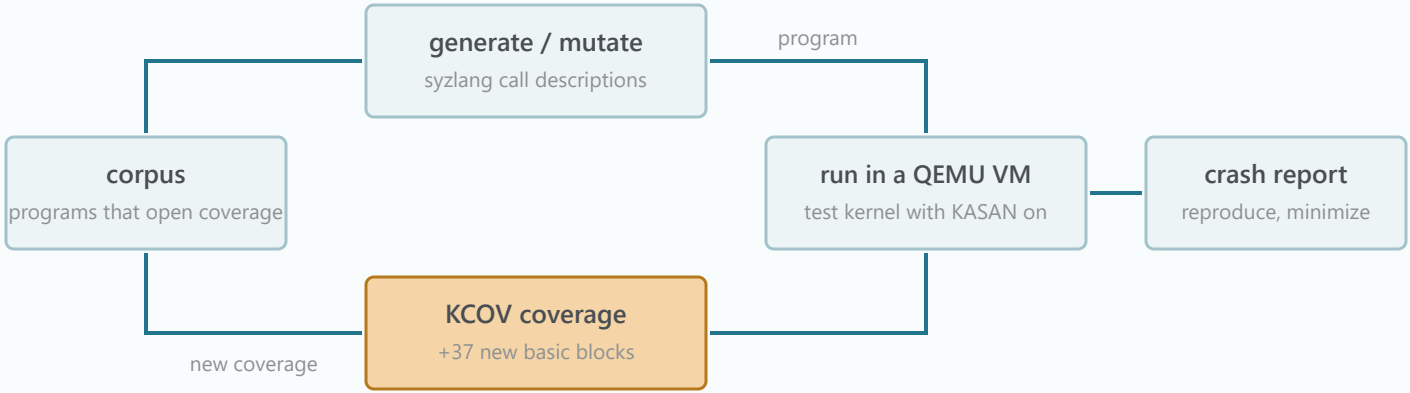
Fuzzing: The syzkaller Loop



The program runs in QEMU VMs booting a test kernel with KASAN/KCSAN/lockdep on. Even if a VM crashes the host is unaffected; `syz-manager` restarts the VM.

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

Fuzzing: The syzkaller Loop



KCOV records the basic blocks (PC values) executed in the kernel. This **coverage feedback** turns random trial into a guided search.

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

Fuzzing: The syzkaller Loop



A program that opens a new code path joins the corpus and seeds later mutations. The loop never stops, with hundreds of programs per second in each VM.

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

Fuzzing: The syzkaller Loop



On a crash or KASAN report, syzkaller reproduces the program, minimizes it and often produces a C reproducer. **syzbot** reports it to developers automatically.

A fuzzer alone sees only crashes; checkers like KASAN make silent memory corruption visible at the first invalid access. That is why the two are used together.

Observation and Tracing Tools

Tool	What it gives	Example
printk	Log levels 0 (EMERG) … 7 (DEBUG); written to a ring buffer, those above the threshold go to the console	`pr_warn("queue full: %d\n", n);`
dmesg	Reads the ring buffer; with `kernel.dmesg_restrict=1` only CAP_SYSLOG holders can read it (prevents address leaks)	`dmesg -w --level=err,warn`
dynamic debug	Turns `pr_debug` lines on at run time, without recompiling	`echo 'module e1000e +p' > …/dynamic_debug/control`
ftrace	The in-kernel tracer: function and function-graph tracing, tracepoint events (tracefs)	`trace-cmd record -p function_graph -g vfs_read`
perf	Hardware counters and sampling profiles: hot spots, cache misses, scheduler latency	`perf record -g -- ./app`, `perf report`
eBPF / bpftrace	Small programs that pass the verifier attach to kprobes and tracepoints; safe observation in production	the one-liner below
kdump / crash	On panic, kexec switches to a spare kernel and saves a memory dump (`/proc/vmcore`) for offline analysis	`crash vmlinux vmcore`

```
# which process opens which file?
bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s %s\n", comm, str(args.filename)); }'
```

Observe first, then guess: logs and traces help form hypotheses. But heavy `printk` can change timing and hide a race, which is why low-overhead tracers (ftrace, eBPF) are preferred.

Kernel Debugging with QEMU and gdb

```
# terminal 1: -s → gdb server :1234
#             -S → CPU halts at start
qemu-system-x86_64 -kernel bzImage \
  -append "console=ttyS0 nokaslr" \
  -nographic -s -S
# terminal 2
gdb vmlinux
(gdb) target remote :1234
(gdb) hbreak start_kernel
(gdb) continue
(gdb) bt
(gdb) lx-dmesg  # kernel gdb script
```

- `vmlinux` : the uncompressed kernel with symbols (`CONFIG_DEBUG_INFO`); `nokaslr` keeps addresses matching the symbols.
- **kgdb**: the same gdb session on real hardware over a serial line (`kgdboc=ttyS0,115200`).

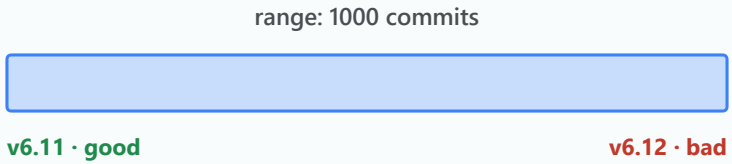
```
BUG: kernel NULL pointer dereference,
     address: 0000000000000010
Oops: 0000 [#1] PREEMPT SMP NOPTI
CPU: 2 PID: 1432 Comm: cat
Tainted: G           O  6.12.0
RIP: 0010:mydrv_read+0x2a/0x80 [mydrv]
Call Trace:
 vfs_read+0xa4/0x2f0
 ksys_read+0x63/0xe0
 do_syscall_64+0x82/0x170
```

- **RIP**: function + offset / function size; `scripts/faddr2line` gives the source line.
- **Tainted O**: an out-of-tree module is loaded. Read the call trace bottom-up: `read` → VFS → driver.
- An **oops** kills that process and the system may carry on; with `panic_on_oops=1` it is a straight **panic**.

Windows: kernel-mode debugging with **WinDbg** (over the network, USB or a VM) and `!analyze -v` on blue-screen dumps; **Driver Verifier** (`verifier.exe`) stresses drivers with special pool, IRQL and resource-leak checks to catch bugs early.

Hunting Regressions: git bisect

```
▶ git bisect start
▶ git bisect bad v6.12
▶ git bisect good v6.11
# #500: build, boot, test
git bisect bad
git bisect good   # #250
git bisect good   # #375
git bisect bad    # #437
git bisect run ./test.sh
git bisect reset
```



The known good (v6.11) and bad (v6.12) ends are given. The first bad commit is among these 1000; git checks out the middle one, **#500**.

Step 1 / 7

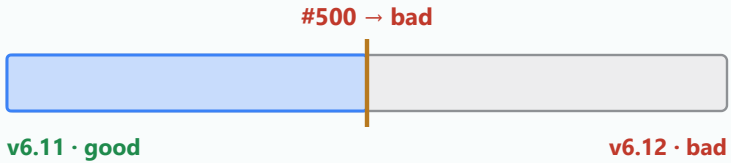
VARIABLES

`range = (0, 1000]` `left = 1000` `tests = 0`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1  git bisect start
2  git bisect bad v6.12
3  git bisect good v6.11
▶  # #500: build, boot, test
▶  git bisect bad
6  git bisect good    # #250
7  git bisect good    # #375
8  git bisect bad     # #437
9  git bisect run ./test.sh
10 git bisect reset
```



candidates left: 500 commits · tests: 1

#500 is built and tested: the bug is there, `git bisect bad`. It began at #500 or earlier; the right half is eliminated.

Step 2 / 7

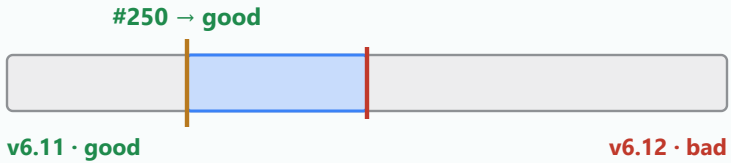
VARIABLES

`range = (0, 500]` `left = 500` `tests = 1`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1  git bisect start
2  git bisect bad v6.12
3  git bisect good v6.11
4  # #500: build, boot, test
5  git bisect bad
▶  git bisect good    # #250
7  git bisect good    # #375
8  git bisect bad     # #437
9  git bisect run ./test.sh
10 git bisect reset
```



candidates left: 250 commits · tests: 2

No bug at #250: `good`. The first bad commit lies between 251 and 500.

Step 3 / 7

VARIABLES

`range = (250, 500]` `left = 250` `tests = 2`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1  git bisect start
2  git bisect bad v6.12
3  git bisect good v6.11
4  # #500: build, boot, test
5  git bisect bad
6  git bisect good    # #250
▶  git bisect good    # #375
8  git bisect bad     # #437
9  git bisect run ./test.sh
10 git bisect reset
```



candidates left: 125 commits · tests: 3

#375 is good too. Each answer halves the candidates: 1000 → 500 → 250 → 125.

VARIABLES

`range = (375, 500]` `left = 125` `tests = 3`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1  git bisect start
2  git bisect bad v6.12
3  git bisect good v6.11
4  # #500: build, boot, test
5  git bisect bad
6  git bisect good   # #250
7  git bisect good   # #375
▶  git bisect bad    # #437
9  git bisect run ./test.sh
10 git bisect reset
```



candidates left: 62 commits · tests: 4

#437 is bad: the range is 376–437. Instead of going on by hand, we can leave the testing to a script.

Step 5 / 7

VARIABLES

`range = (375, 437]` `left = 62` `tests = 4`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1  git bisect start
2  git bisect bad v6.12
3  git bisect good v6.11
4  # #500: build, boot, test
5  git bisect bad
6  git bisect good   # #250
7  git bisect good   # #375
8  git bisect bad    # #437
▶  git bisect run ./test.sh
10 git bisect reset
```



found in 10 tests = $\lceil \log_2 1000 \rceil$

`git bisect run` carries out the remaining steps itself: 406 good, 421 bad, 413 bad, 409 good, 411 good, 412 bad. **10 tests** in all: $2^{10} = 1024 \geq 1000$.

VARIABLES

`range = (411, 412]` `left = 1` `tests = 10`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means “this commit cannot be tested, skip it”.

Hunting Regressions: git bisect

```
1 git bisect start
2 git bisect bad v6.12
3 git bisect good v6.11
4 # #500: build, boot, test
5 git bisect bad
6 git bisect good   # #250
7 git bisect good   # #375
8 git bisect bad    # #437
9 git bisect run ./test.sh
▶ git bisect reset
```



git bisect reset → back to the working branch

Result: "#412 is the first bad commit". `reset` returns to the starting branch. Even with a million commits only $\lceil \log_2 10^6 \rceil = 20$ tests would be needed.

Step 7 / 7

VARIABLES

`range = (411, 412]` `left = 1` `tests = 10`

If `./test.sh` exits with 0 the commit is good; 1–127 (except 125) means bad; 125 means "this commit cannot be tested, skip it".

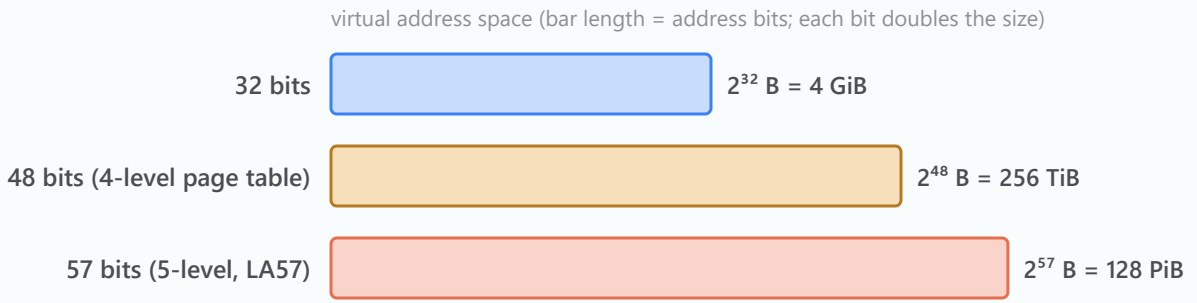
Trends in Operating Systems

Trend	Impact on the OS
Virtualization and the cloud	The guest kernel cooperates with the hypervisor through virtio; in a multitenant cloud the isolation boundary moves to the hypervisor. Containers: light but weak isolation; micro-VMs (Firecracker): VM isolation, millisecond boot
Multicore chips	Scalable locks, RCU, NUMA awareness, heterogeneous cores
Large address spaces	64 bits, 5-level page tables; single-address-space ideas
Seamless data access	Transparent access across network, cloud storage and local disk; NVMe, CXL memory
Battery-powered systems	DVFS, race-to-idle, Android Doze, energy-aware scheduling (EAS)
Embedded and sensor nodes	Small real-time kernels: Zephyr, FreeRTOS, Contiki-NG; kilobytes of memory
Safe languages and a programmable kernel	Rust in Linux (since 6.1), eBPF and sched_ext, unikernels

Networking, parallel and distributed systems and multimedia are now a normal part of every system, so they appear inside the relevant rows rather than as separate trends.

TRENDS

Multicore and Large Address Spaces



Multicore chips

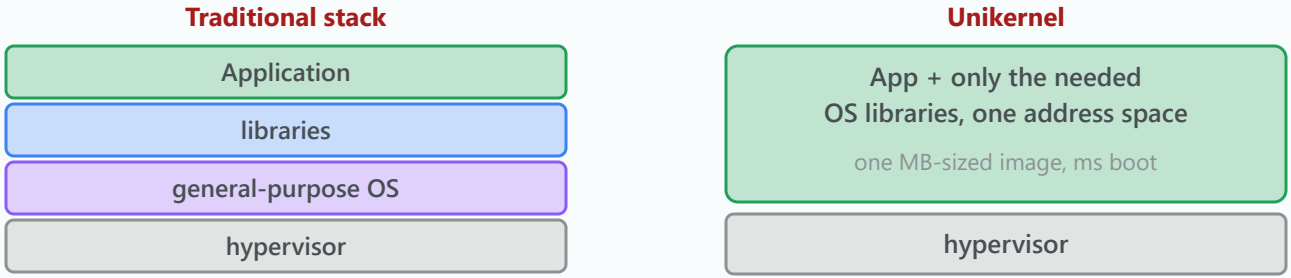
128–192 cores per socket in servers, mixed big/little (big.LITTLE) cores in phones. One big kernel lock does not scale: fine-grained locks, RCU and per-core data structures are needed.

Large address spaces

Mapping every file into memory (mmap) or single-address-space systems become possible. The price: deeper page-table walks, more TLB pressure.

TRENDS

Rust, eBPF and Unikernels



Kernels in Rust

Rust support entered mainline with Linux **6.1** (December 2022), followed by the first Rust drivers. The goal: prevent memory bugs such as use-after-free and overflows at compile time. The Windows kernel has Rust components too.

eBPF

Extending the kernel with small verified programs, without recompiling: tracing, networking (XDP), security, scheduling (sched_ext).

Unikernel

The cloud version of the exokernel's libOS idea: MirageOS, Unikraft. For single-application microservices.

Summary

Topic	Key idea
Challenges	Size, concurrency, hostile users, long life, compatibility
Goals	Abstractions, primitives, isolation, hardware management
Interface	Simple, complete, efficient; few, general system calls
Structure	Isolation versus performance; layer order, safe extension, kernel threads
Principles	Mechanism/policy, orthogonality, two-level names, late binding
Techniques	Hiding hardware, indirection, reentrancy, checking errors first
Performance	Measure; space–time trade-offs, caching, hints, locality, the common case
Project	Brooks's law, small strong teams, no silver bullet
Testing	KUnit, kselftest, LTP; KASAN, KCSAN, lockdep; syzkaller; ftrace, eBPF; QEMU+gdb; git bisect
Trends	Cloud, multicore, energy, Rust, eBPF, unikernels

SUMMARY

Exercise: Testing and Debugging

1. With which tools, and in what order, would you catch an occasional use-after-free in a driver? Why is syzkaller run with a KASAN-enabled kernel?
2. There are 12,000 commits between v6.11 and v6.12, and a performance regression appeared in v6.12. At most how many tests does `git bisect` need? Which exit codes should the test script use?
3. A race disappears when you add a `printk`. Why, and which tools would you investigate with next?

▶ **Answers**

Review Questions

1. Explain four of the eight factors that, according to Tanenbaum, make OS design hard.
2. Discuss the principles of simplicity, completeness and efficiency, each with a system-call example.
3. Compare the algorithmic and event-driven execution paradigms. Why is a long-running handler a problem in an event loop?
4. In the seven-layer design, why does virtual memory sit above the device drivers?
5. What happens when a driver crashes in a microkernel? Why is the same thing more dangerous in a monolithic kernel?
6. Explain separating mechanism from policy with the scheduler example. How does sched_ext apply this principle?
7. How many disk accesses does resolving /usr/ast/mbox take without caching? How does a cache differ from a hint?
8. Compare the loop, macro and table methods of the bit-counting example in terms of space and time.
9. Why is the futex a good example of optimizing the common case?
10. Justify Brooks's law with the number of communication paths: how many paths does a 10-person team have?

▶ **Answers**

References

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 12: Operating System Design.
- B. W. Lampson, “Hints for Computer System Design”, *ACM SOSP*, 1983 (and the 2020 update, “Hints and Principles for Computer System Design”).
- F. P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering*, 20th anniversary ed., Addison-Wesley, 1995 — including “No Silver Bullet” (1986).
- J. H. Saltzer, M. F. Kaashoek, *Principles of Computer System Design: An Introduction*, Morgan Kaufmann, 2009.
- D. R. Engler, M. F. Kaashoek, J. O’Toole, “Exokernel: An Operating System Architecture for Application-Level Resource Management”, SOSP 1995.
- G. Klein et al., “seL4: Formal Verification of an OS Kernel”, SOSP 2009; J. N. Herder et al., MINIX 3 reliability studies.
- Linux kernel documentation: Rust (docs.kernel.org/rust), BPF and sched_ext; H. Franke et al., “Fuss, Futexes and Furwocks”, 2002.

The End

Chapter 14: Operating-System Design

Previous: Chapter 13: Security · Next: Chapter 15: Linux