# Chapter 2: Operating-System Structures

## Operating Systems

Services | Process, address space, file | Dual mode | System calls

POSIX and Win32 | Monolithic | Layered | Microkernel | Modular | Hybrid

Exokernel | Virtual machines | Linking and booting

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_02_Operating_System_Structures.html

# Learning Outcomes

### Services and concepts

Explain the services an OS provides and the concepts of process, address space, file, mounting and pipe.

### System calls

Trace step by step the switch from user to kernel mode, parameter passing and the basic POSIX calls.

### Kernel structures

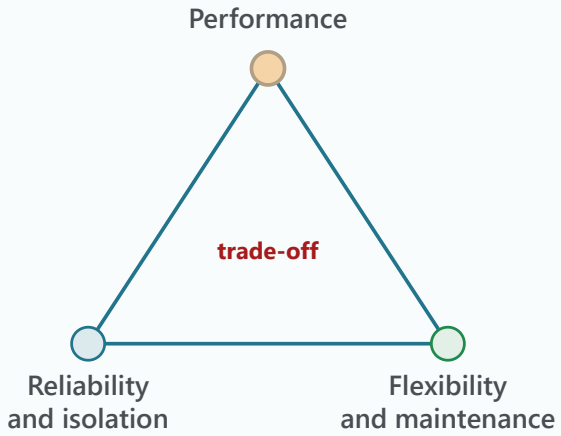
Compare monolithic, layered, microkernel, modular, hybrid and exokernel structures, with their strengths and weaknesses.

### Virtualization and booting

Describe virtual machines and containers, the linker/loader and the boot steps.

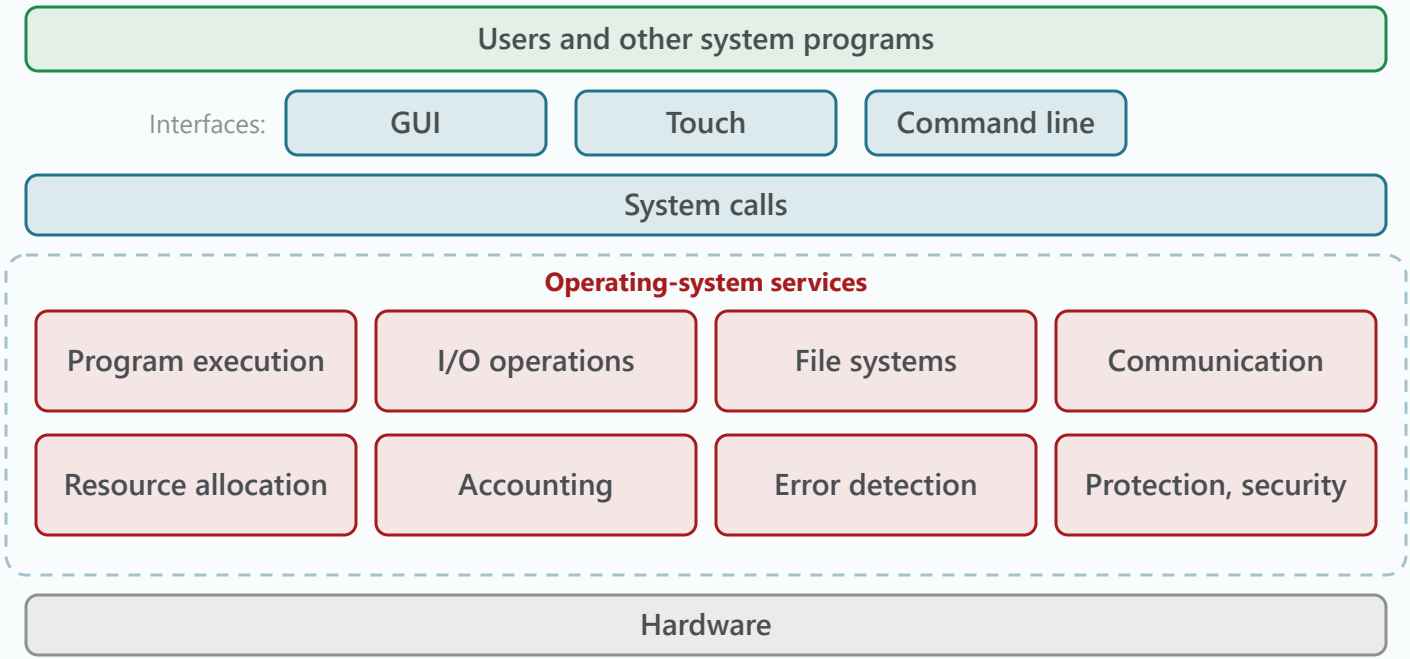
# Why Does Structure Matter?

- An OS is made of many components: process, memory, file, I/O and network management, and more.
- The strategy for **where** these components run (kernel / user mode) and **how they communicate** (function call / message) is called the **OS structure**.
- Structure directly affects how we manage complexity, make changes, debug and port.




There is no single right structure: each offers a particular trade-off and suits particular uses (desktop, server, embedded, safety-critical).

# Operating-System Services




User interfaces and programs reach the services through system calls (adapted from Silberschatz, Figure 2.1).

# Whom Do the Services Serve?

### Helping users and programs

- **Interface:** command line, GUI, touch, voice.
- **Program execution:** load, run, terminate normally or abnormally.
- **I/O operations:** a program reaches a device through the OS, not directly.
- **File system:** create, delete, read, write, search, permissions.
- **Communication:** shared memory or message passing (local / network).
- **Error detection:** hardware, device and program errors.

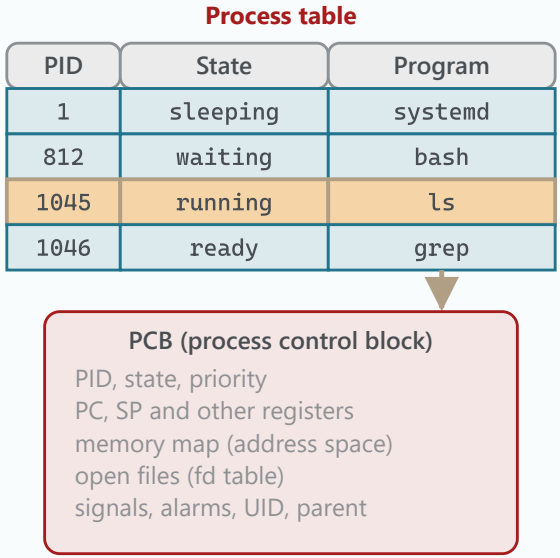
### Keeping the system efficient and secure

- **Resource allocation:** CPU, memory and storage are shared among many processes.
- **Accounting:** who used how much of what; logs.
- **Protection:** processes cannot harm each other or the kernel.
- **Security:** authentication, authorization, defense against outside attacks.

Most users see the OS not through system calls but through **system programs**: `ls`, `cp`, `ps`, package managers, compilers, file managers…

# Process

- A process is **a program in execution**; the program is a passive file on disk, the process its active instance.
- Each process has an **address space** (code, data, stack), registers (PC, SP), open files, alarms, related processes and other resources.
- The OS keeps this information in the **process table**; a suspended process resumes where it left off thanks to its PCB.
- Processes can talk to each other through **IPC** (interprocess communication).



BASIC CONCEPTS

# Process Tree

A process can create **child processes** with `fork()`; the children create their own children. The result is a tree.

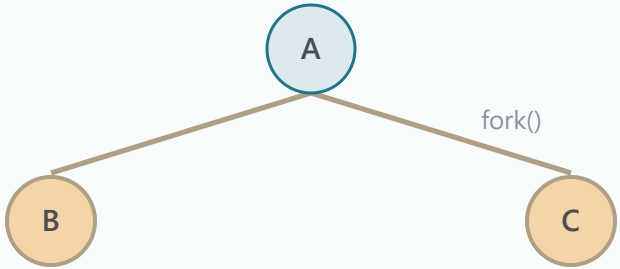



PID 1 (systemd) is the root of the tree

At first only process A exists. On Linux the root of the whole tree is `systemd` (or `init`), PID 1.

# Process Tree

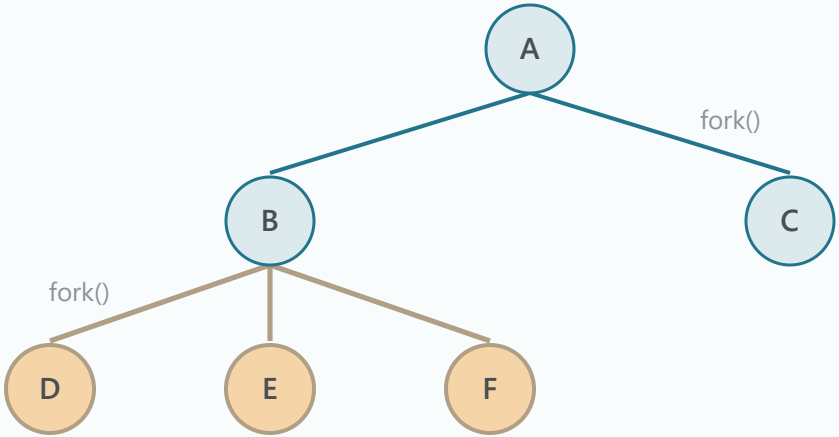
A process can create **child processes** with `fork()`; the children create their own children. The result is a tree.




Process A calls `fork()` twice: B and C are A's children (A is their parent).

# Process Tree

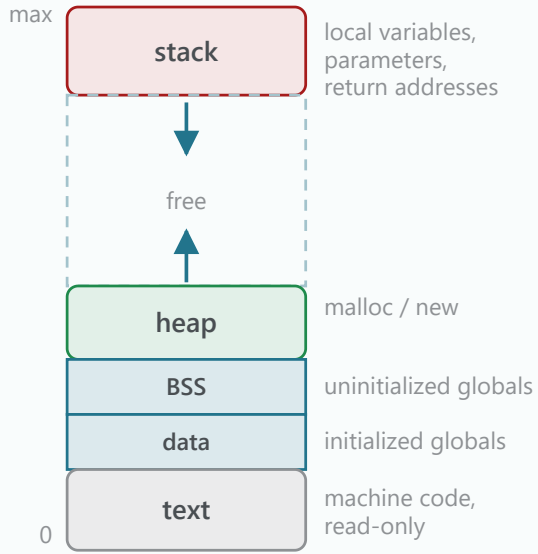
A process can create **child processes** with `fork()`; the children create their own children. The result is a tree.




Process B creates three children: D, E and F. The `pstree -p` command shows this tree.

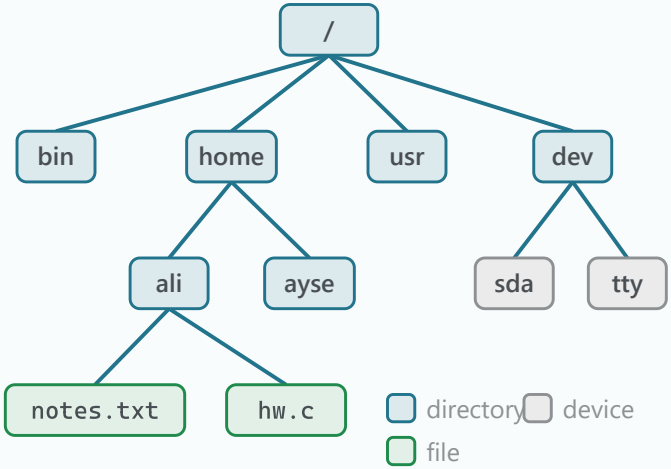
# Address Space and Memory Segments

- **Address space:** the set of addresses a process can use (0 … max). On x86-64 Linux user space is 128 TiB.
- Many processes are in memory at once; the OS protects them from each other and from the kernel.
- **Text:** machine code; read-only, shared by processes running the same program and by all threads.
- **Data + BSS:** global and static variables. **Heap:** memory allocated dynamically at run time. Both are shared by the process's threads.
- **Stack:** parameters, local variables, return addresses; *each thread has its own stack*.
- **Virtual memory** allows address spaces larger than physical memory.

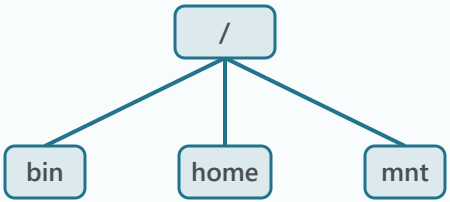


# Files and the Directory Hierarchy

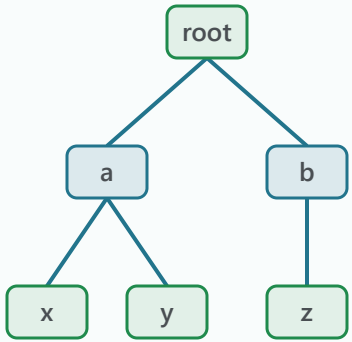
- **File:** a named unit that stores data persistently (text, images, video, programs…).
- The OS hides disk blocks and device details and offers a clean abstraction.
- Files live in a **directory tree**. A path name can be absolute ( `/home/ali/notes.txt` ) or relative to the working directory ( `ali/notes.txt` ).
- An opened file gets a small integer, the **file descriptor** (fd); 0, 1, 2 = stdin, stdout, stderr.
- **Special files:** devices look like files. Block devices ( `/dev/sda` , `/dev/nvme0n1` ) and character devices ( `/dev/tty` , serial port).



# Mounting a File System




?

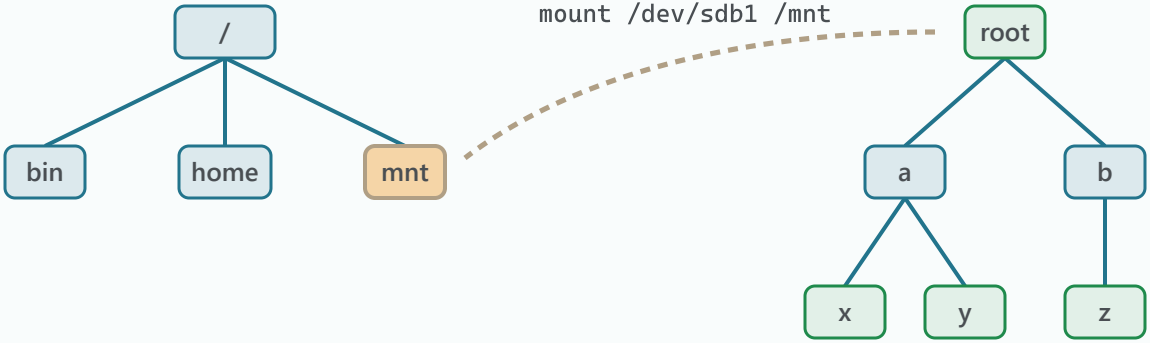



Root file system (/dev/nvme0n1p2)

USB stick (/dev/sdb1)

Before mounting, the file system on the USB stick is a separate tree; no path name reaches it.

# Mounting a File System

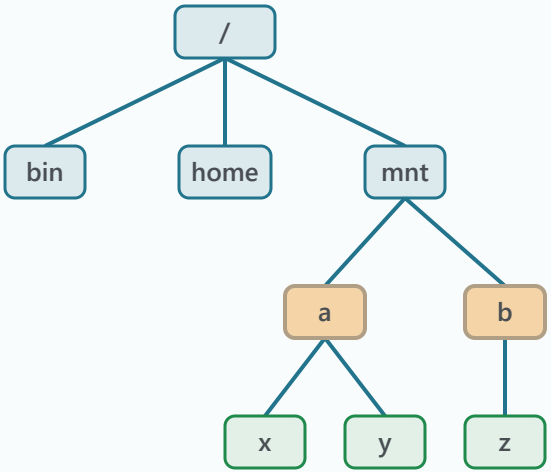



Root file system (/dev/nvme0n1p2)

USB stick (/dev/sdb1)

`mount` attaches the root of the new file system to a **mount point** in the existing tree (the empty directory `/mnt`).

# Mounting a File System




Root file system (/dev/nvme0n1p2)

Now a single tree:
/mnt/a/x
/mnt/b/z

Once mounted, the USB files are part of the hierarchy. Windows usually uses drive letters (C:, D:) instead.

# The `mount` Command and Its Options

```
$ lsblk                              # list disks and partitions
$ sudo mount /dev/sdb1 /mnt          # mount sdb1 on /mnt
$ sudo mount -o ro /dev/sdb1 /mnt    # mount read-only
$ sudo mount -o remount,rw /mnt      # remount read-write
$ findmnt /mnt                       # show mount info
$ sudo umount /mnt                   # unmount
```

### `ro` / `rw`

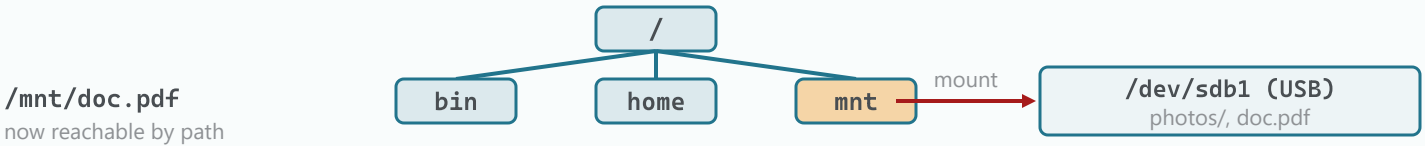
Mount read-only or read-write. `ro` is the safe choice for recovery and forensics.

### `/etc/fstab`

File systems to mount automatically at boot are listed here (or in systemd mount units).

### On the desktop

A plugged-in USB stick is mounted automatically under `/run/media/…` by `udisks2`.

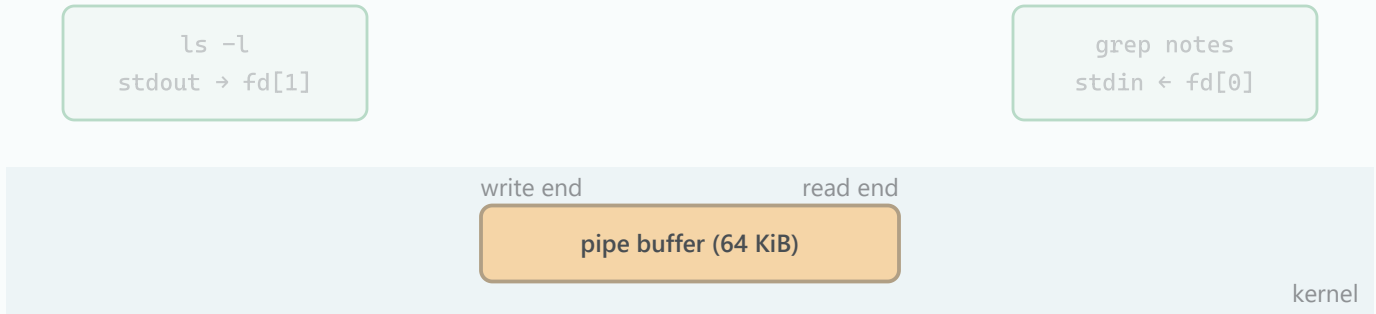


BASIC CONCEPTS

# Pipe

- A **one-way** byte channel buffered in the kernel that connects two processes; they use it like a file, with `read` / `write`.
- **Anonymous:** created with `pipe()`, between parent and child. **Named (FIFO):** created with `mkfifo`, has a name in the file system, usable by unrelated processes.

```
$ ls -l | grep "notes"
```

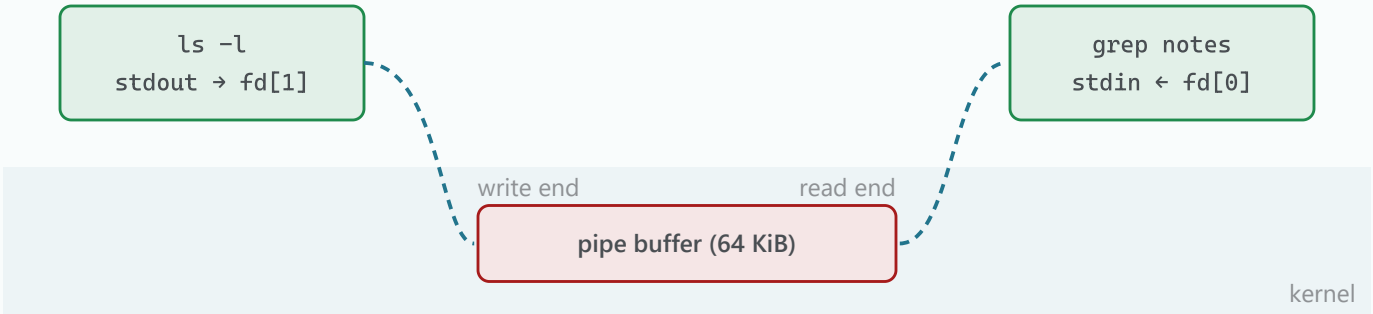



The shell first calls `pipe(fd)`: the kernel opens a buffer with two ends (fd[0] read, fd[1] write).

# Pipe

- A **one-way** byte channel buffered in the kernel that connects two processes; they use it like a file, with `read` / `write`.
- **Anonymous:** created with `pipe()`, between parent and child. **Named (FIFO):** created with `mkfifo`, has a name in the file system, usable by unrelated processes.

```
$ ls -l | grep "notes"
```

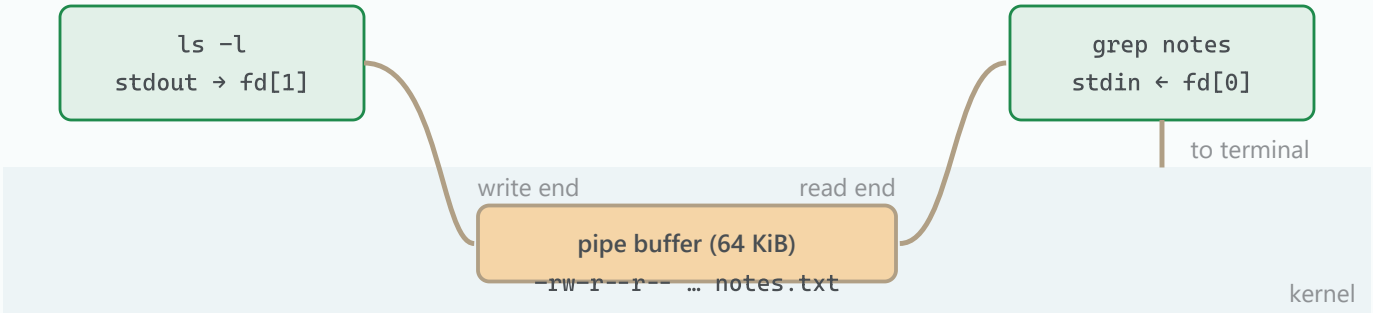



The shell forks twice; with `dup2`, ls's stdout goes to the write end and grep's stdin comes from the read end.

# Pipe

- A **one-way** byte channel buffered in the kernel that connects two processes; they use it like a file, with `read` / `write`.
- **Anonymous:** created with `pipe()`, between parent and child. **Named (FIFO):** created with `mkfifo`, has a name in the file system, usable by unrelated processes.

```
$ ls -l | grep "notes"
```

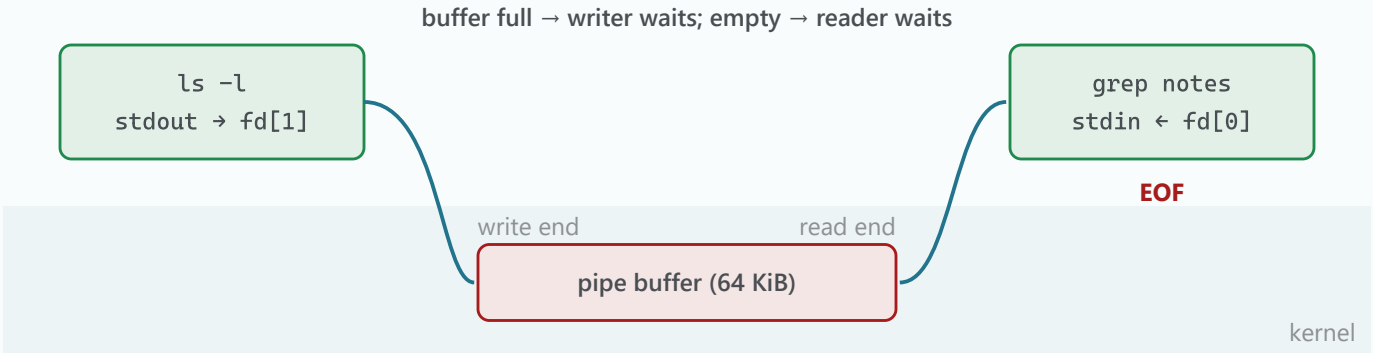



ls writes its listing into the pipe; grep reads it and prints only the lines containing “notes”.

# Pipe

- A **one-way** byte channel buffered in the kernel that connects two processes; they use it like a file, with `read` / `write`.
- **Anonymous:** created with `pipe()`, between parent and child. **Named (FIFO):** created with `mkfifo`, has a name in the file system, usable by unrelated processes.

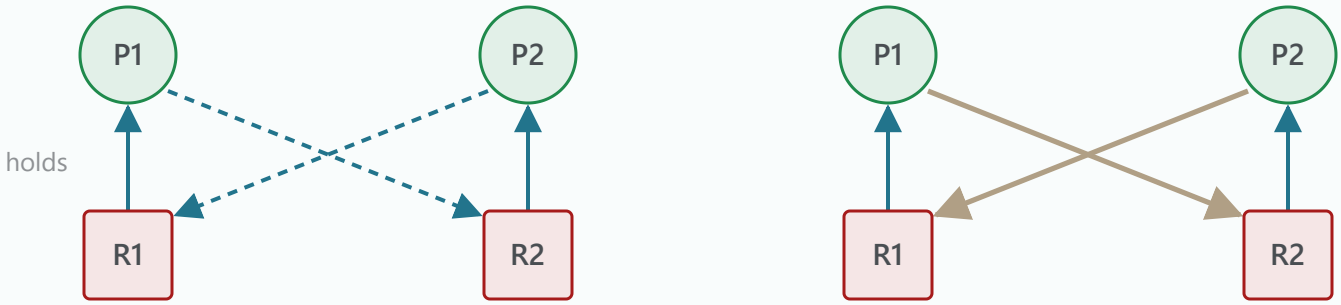
```
$ ls -l | grep "notes"
```




Flow control is in the kernel: the writer blocks when the buffer is full, the reader when it is empty. When ls finishes and the write end closes, grep sees end of file (EOF).

# Deadlock

- If every process in a set waits for a resource **held by another process** in the set, none can proceed: deadlock.
- Like cars entering an intersection from four directions and waiting for each other.




(a) Potential: requests not yet made

(b) Actual: cycle P1→R2→P2→R1→P1

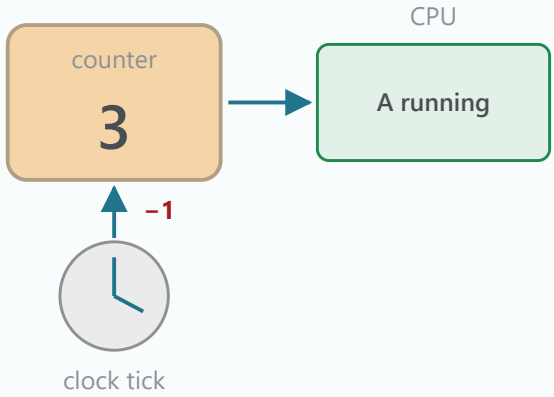
Arrow from box to process: the resource is held by that process. Arrow from process to box: the process requests it (dashed = will request later).

# Protection with Dual Mode

- The OS protects itself and processes through the hardware's **user / kernel mode** split: I/O, setting the timer, disabling interrupts and changing page tables are possible only in kernel mode.
- If a privileged instruction is attempted in user mode, the hardware traps and the OS kills the process. The kernel is entered only through system calls, interrupts and exceptions.
- x86 has rings 0–3, but Linux and Windows use only 0 and 3; ARM has EL0–EL3, with EL2 reserved for virtualization.
- Mode separation alone is not enough: a **timer** is needed to take the CPU back, and an **MMU** to keep process memories apart.

# Timer

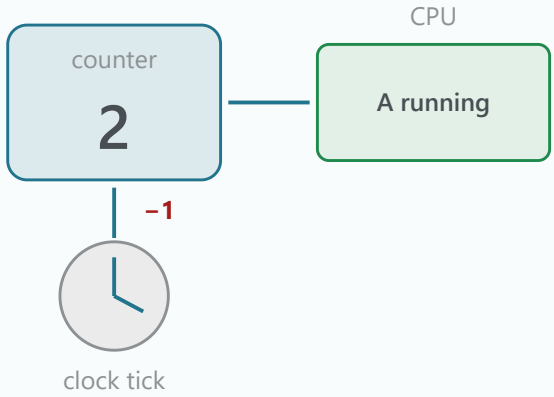
- Goal: stop a process from looping forever and **never giving up** the CPU.
- The OS loads a value into a counter; this is a **privileged** instruction.
- The counter decrements on every tick of the physical clock; **at zero an interrupt** fires and the OS regains control.
- The OS preempts a process that has used up its time slice and gives the CPU to another (preemptive scheduling). The process is killed only if a CPU-time limit (e.g. `RLIMIT_CPU`) is exceeded.
- Linux's periodic tick is 100–1000 Hz; *tickless* (NO_HZ) mode, which stops ticks when idle, saves energy.




Before running A, the OS loads 3 into the counter.

# Timer

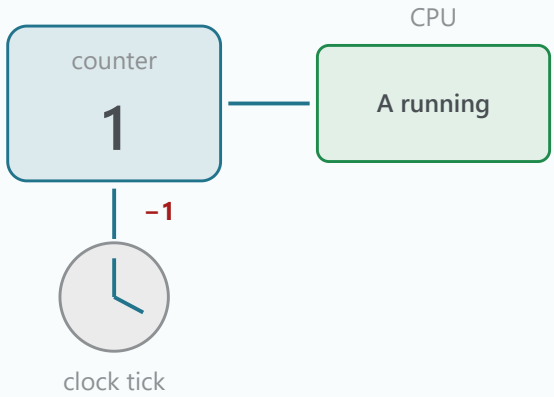
- Goal: stop a process from looping forever and **never giving up** the CPU.
- The OS loads a value into a counter; this is a **privileged** instruction.
- The counter decrements on every tick of the physical clock; **at zero an interrupt** fires and the OS regains control.
- The OS preempts a process that has used up its time slice and gives the CPU to another (preemptive scheduling). The process is killed only if a CPU-time limit (e.g. `RLIMIT_CPU`) is exceeded.
- Linux's periodic tick is 100–1000 Hz; *tickless* (NO_HZ) mode, which stops ticks when idle, saves energy.




Each clock tick decrements the counter: 2.

# Timer

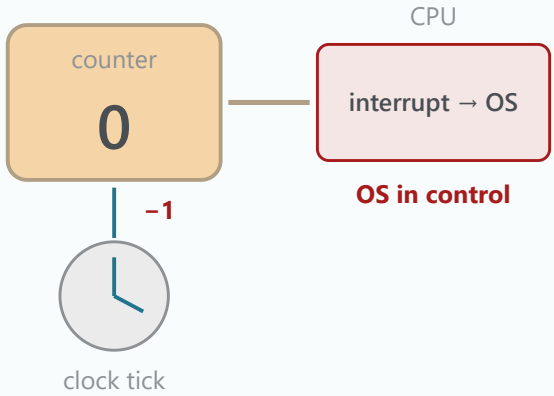
- Goal: stop a process from looping forever and **never giving up** the CPU.
- The OS loads a value into a counter; this is a **privileged** instruction.
- The counter decrements on every tick of the physical clock; **at zero an interrupt** fires and the OS regains control.
- The OS preempts a process that has used up its time slice and gives the CPU to another (preemptive scheduling). The process is killed only if a CPU-time limit (e.g. `RLIMIT_CPU`) is exceeded.
- Linux's periodic tick is 100–1000 Hz; *tickless* (NO_HZ) mode, which stops ticks when idle, saves energy.




Counter 1; A is still on the CPU.

# Timer

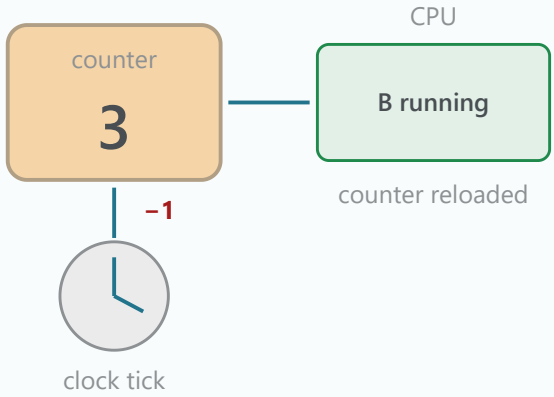
- Goal: stop a process from looping forever and **never giving up** the CPU.
- The OS loads a value into a counter; this is a **privileged** instruction.
- The counter decrements on every tick of the physical clock; **at zero an interrupt** fires and the OS regains control.
- The OS preempts a process that has used up its time slice and gives the CPU to another (preemptive scheduling). The process is killed only if a CPU-time limit (e.g. `RLIMIT_CPU`) is exceeded.
- Linux's periodic tick is 100–1000 Hz; *tickless* (NO_HZ) mode, which stops ticks when idle, saves energy.




Counter 0: timer interrupt. The CPU switches to kernel mode and the interrupt handler runs.

# Timer

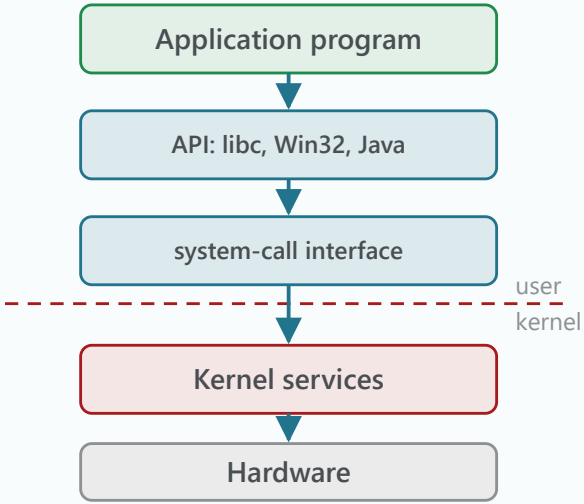
- Goal: stop a process from looping forever and **never giving up** the CPU.
- The OS loads a value into a counter; this is a **privileged** instruction.
- The counter decrements on every tick of the physical clock; **at zero an interrupt** fires and the OS regains control.
- The OS preempts a process that has used up its time slice and gives the CPU to another (preemptive scheduling). The process is killed only if a CPU-time limit (e.g. `RLIMIT_CPU`) is exceeded.
- Linux's periodic tick is 100–1000 Hz; *tickless* (NO_HZ) mode, which stops ticks when idle, saves energy.



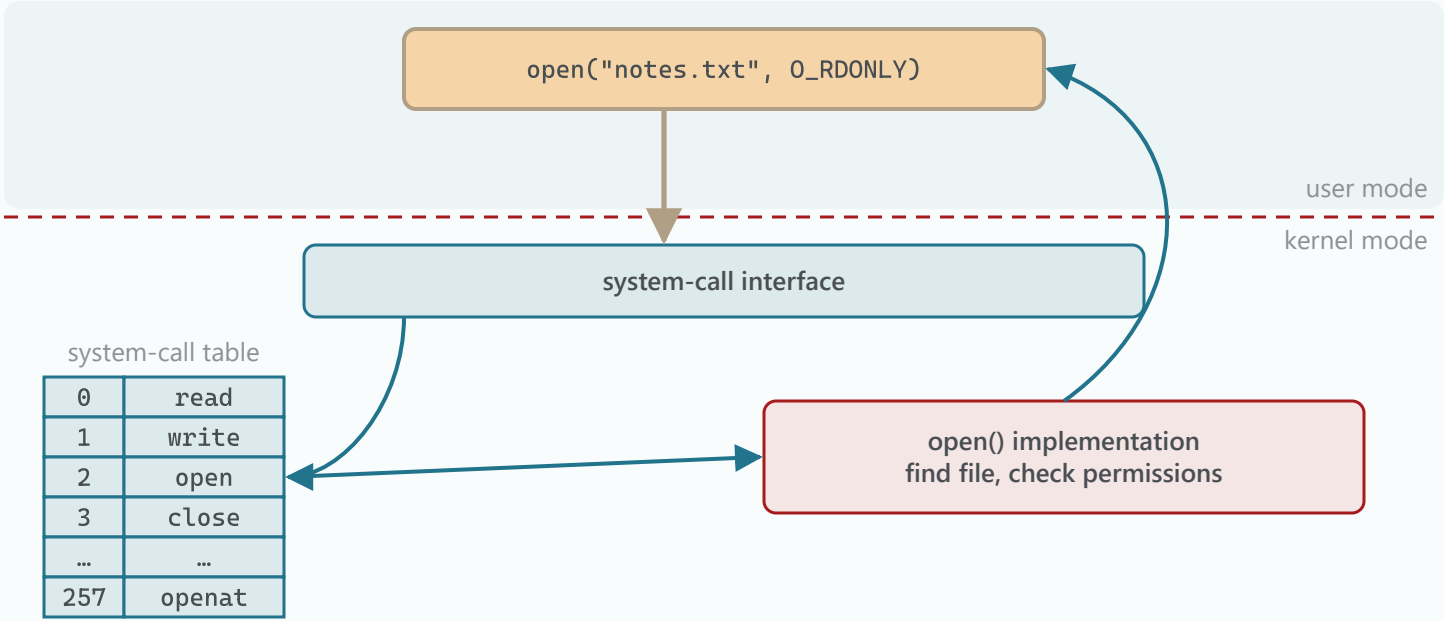

A's slice is used up, so the scheduler picks B and reloads the counter.

# What Is a System Call?

- The **defined interface** programs use to reach OS services (files, memory, processes, time, network…).
- *Requested* by the application, *executed in kernel mode* by the kernel.
- Programmers usually reach calls through an **API**, not directly: POSIX (glibc, musl), the Windows API, the Java API. The API gives portability and convenience.
- The set of calls varies by system: Linux on x86-64 has about 370. The core ideas are shared.
- `strace` (Linux), `dtruss` (macOS) and Process Monitor (Windows) show the calls a program makes.

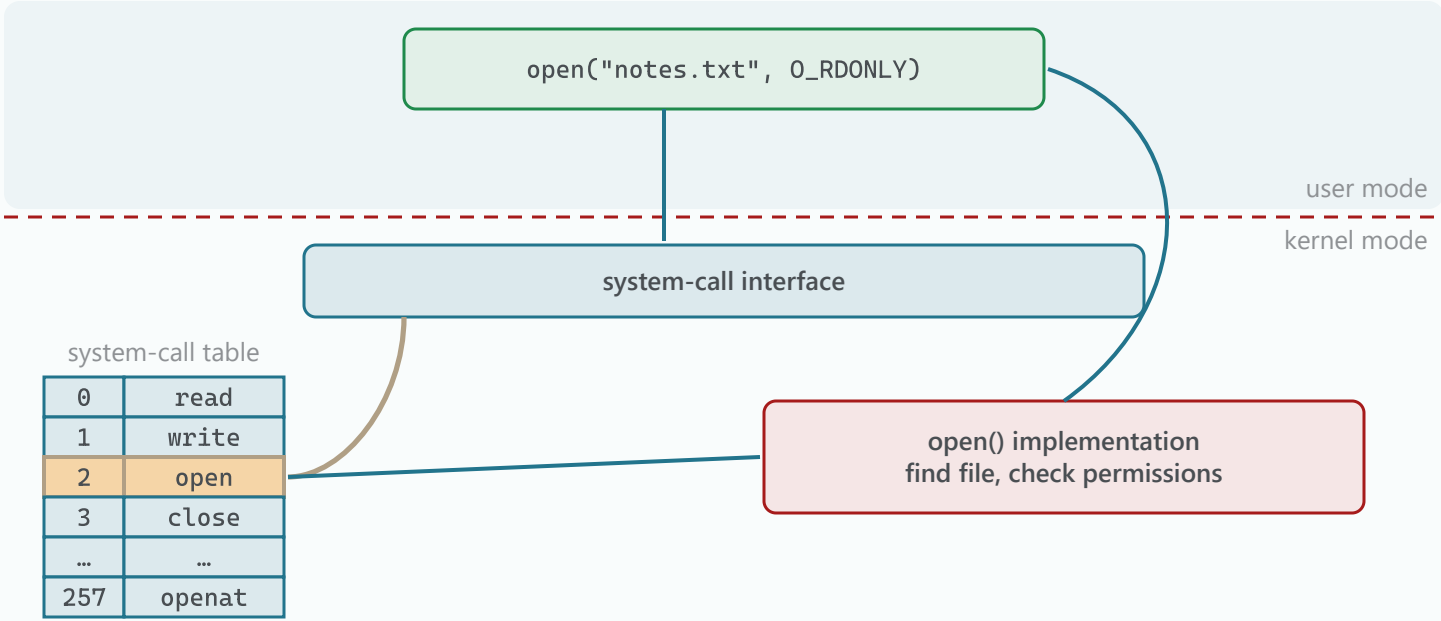


# API → System-Call Interface → Kernel



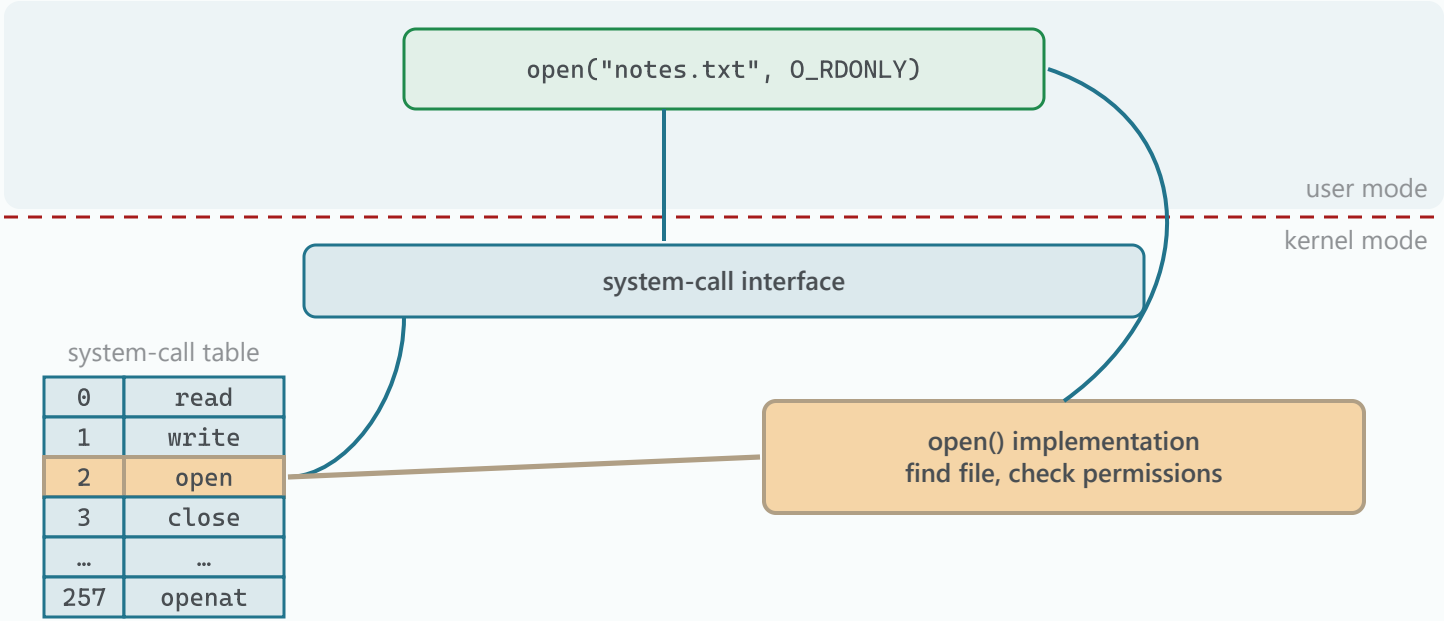

The application calls the `open()` wrapper in libc; the wrapper puts the call number in a register and executes the trap instruction.

# API → System-Call Interface → Kernel



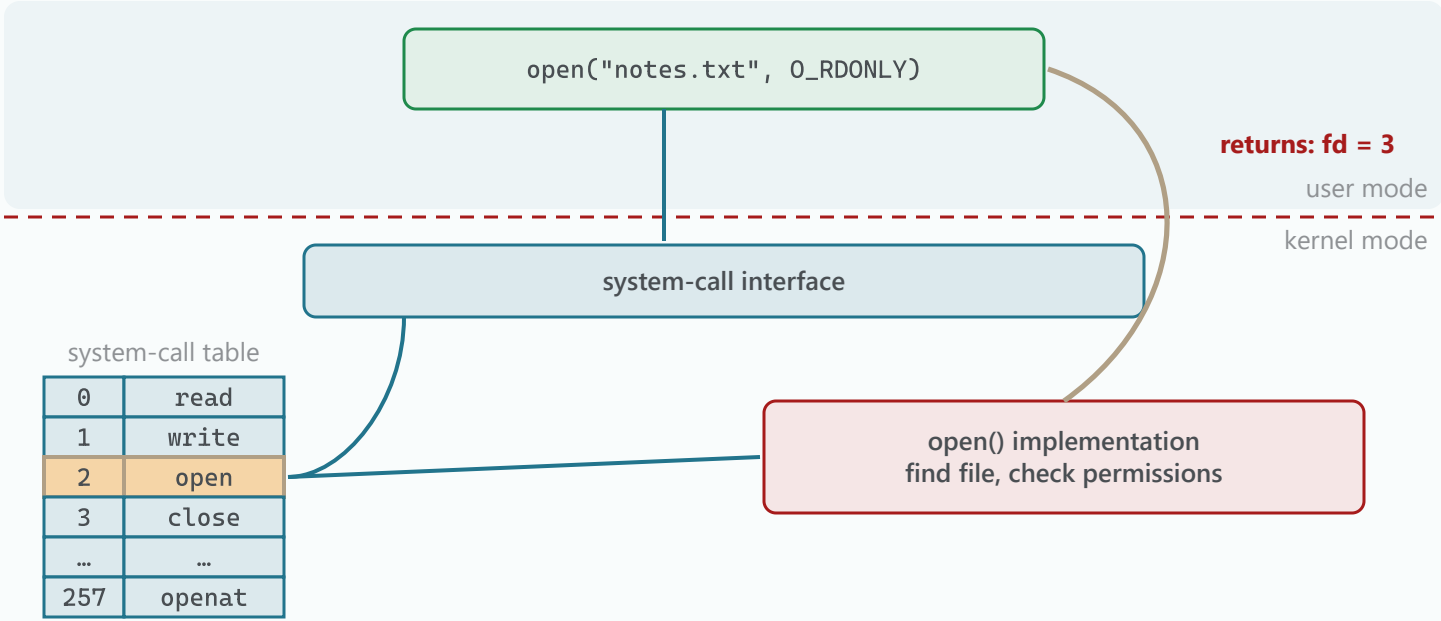

The kernel's system-call interface uses the number as an index into the **system-call table** (x86-64 Linux: open = 2).

# API → System-Call Interface → Kernel




The pointer in the table leads to the kernel implementation: the path is resolved, permissions are checked, an open-file entry is created.

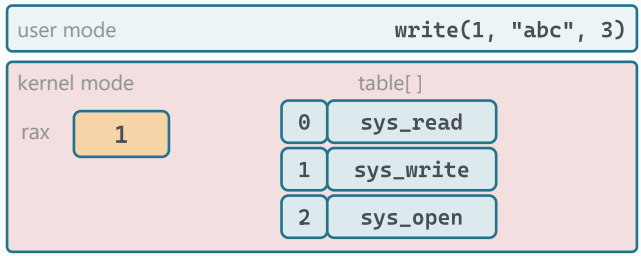
# API → System-Call Interface → Kernel




The result (the lowest free descriptor: 3, since 0, 1, 2 are taken) returns to the user. Note: current glibc actually implements `open()` with `openat` (257).

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
▶ long syscall_entry(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




The libc wrapper puts the call number in `rax` and executes `syscall`. The hardware enters kernel mode and jumps to the **single entry point**: **no = 1** (write on Linux x86-64).

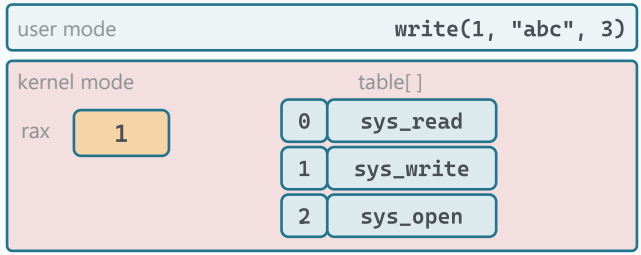
Step 1 / 9

VARIABLES

`no = 1`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




0 ≤ 1 < 3: the number is valid.

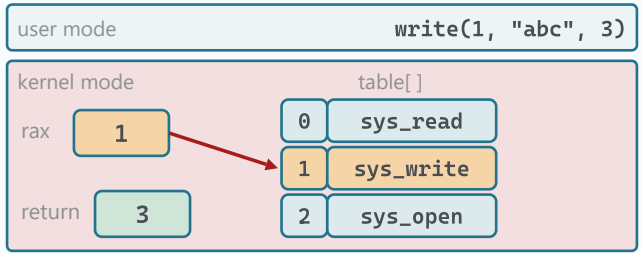
Step 2 / 9

**VARIABLES**

`no = 1`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
▶   return table[no](a, b, c);
9 }
```




`table[1]` is a function pointer: **sys_write** is called and returns **3**, the number of bytes written.

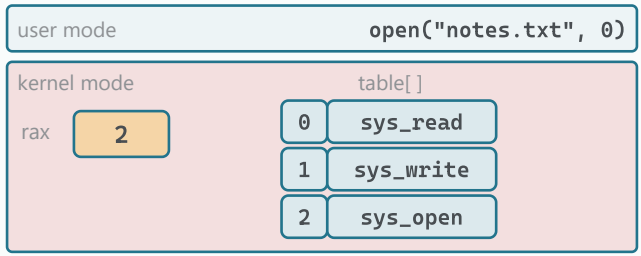
Step 3 / 9

VARIABLES

`no = 1` `ret = 3`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
▶ long syscall_entry(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




A new call: `open("notes.txt", 0)`, **no = 2**.

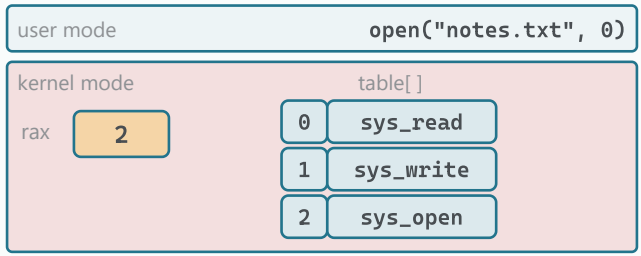
Step 4 / 9

**VARIABLES**

`no = 2`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




0 ≤ 2 < 3: the number is valid.

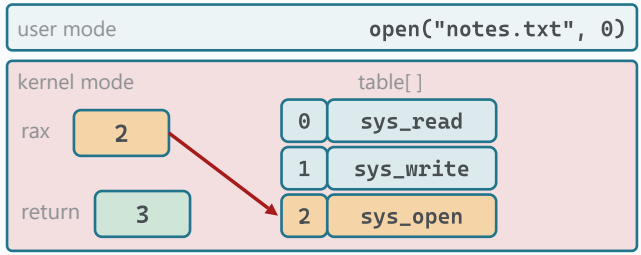
Step 5 / 9

**VARIABLES**

`no = 2`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
▶   return table[no](a, b, c);
9 }
```




`table[2]` is a function pointer: **sys_open** is called and returns **3**, the new descriptor.

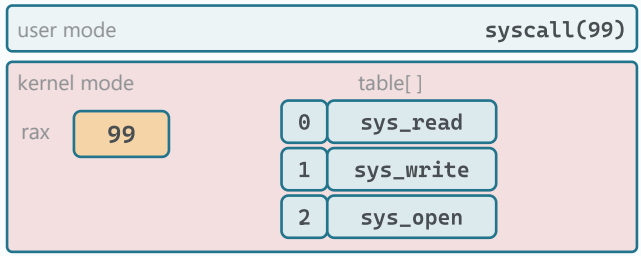
Step 6 / 9

**VARIABLES**

`no = 2` `ret = 3`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
▶ long syscall_entry(long no, long a,
▶                    long b, long c) {
6   if (no < 0 || no >= 3)
7     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




A new call: `syscall(99)`, **no = 99**.

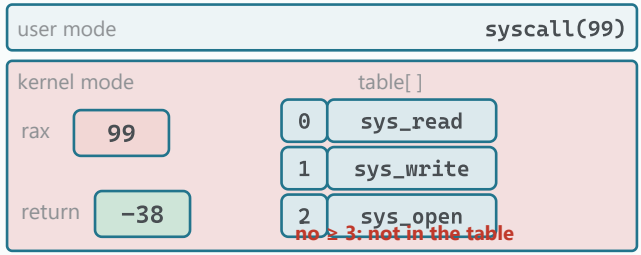
Step 7 / 9

**VARIABLES**

`no = 99`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
▶   if (no < 0 || no >= 3)
▶     return -ENOSYS;     /* bad number */
8   return table[no](a, b, c);
9 }
```




The number is outside the table. Without the check, `table[99]` would jump to a random kernel address. The call is rejected with **−ENOSYS (−38)**.

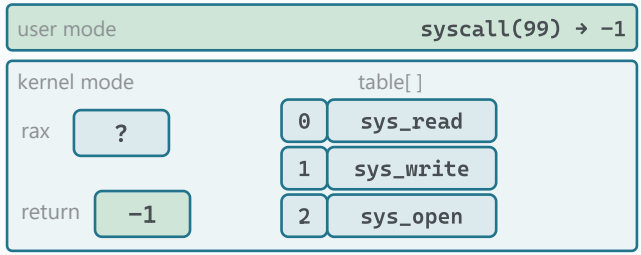
Step 8 / 9

**VARIABLES**

`no = 99` `ret = −38`

# Code Trace: The System-Call Table

```
1 typedef long (*syscall_t)(long, long, long);
2 syscall_t table[3] = {sys_read, sys_write,
3                       sys_open};
4 long syscall_entry(long no, long a,
5                     long b, long c) {
6   if (no < 0 || no >= 3)
▶     return -ENOSYS;      /* bad number */
8   return table[no](a, b, c);
9 }
```




libc turns the negative value into `errno = ENOSYS` and returns **−1** to the program. The kernel never uses a value from user space **without validating it**.

Step 9 / 9

VARIABLES

`no = 99` `ret = −1` `errno = ENOSYS`

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

Even getting a name from the user takes two calls: write a prompt to the screen ( `write` ), read from the keyboard ( `read` ).

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

The same is repeated for the output file.

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

If `open` fails (no such file, no permission) it returns −1; the program prints an error and exits.

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

`open(…, O_WRONLY | O_CREAT | O_EXCL)` : fails if the file already exists.

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

The real work: a `read` / `write` loop until `read` returns 0 (end of file).

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

Every open file is closed with `close`.

# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

The user is informed.

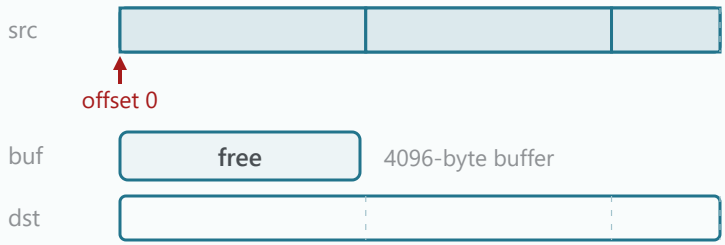
# Example: Copying One File to Another

| | |
|---|---|
| Prompt for and read the input file name | `write, read` |
| Prompt for and read the output file name | `write, read` |
| Open the input file (if missing, report and exit) | `open` |
| Create the output file (if it exists, report and exit) | `open + O_CREAT` |
| Loop: read from input, write to output | `read, write` |
| Close the files | `close` |
| Print “done” on the screen | `write` |
| Terminate normally | `exit` |

The process ends with `exit`. Even a simple program like this makes dozens of calls; count them with `strace -c cp a b`.

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
▶  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1`

"src" is a 10,000-byte file. `open` returns the lowest free descriptor: 0, 1, 2 are taken, so **in = 3**; offset 0.

Step 1 / 12

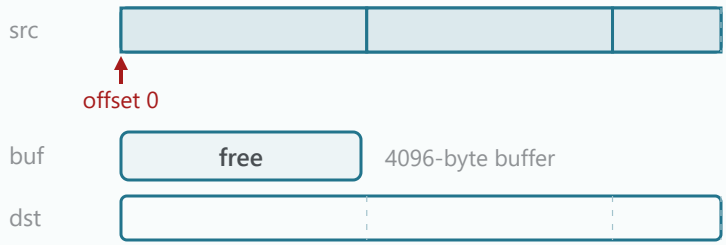
**VARIABLES**

`in = 3` `out = ?` `n = ?` `total = 0`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
▶ 3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1`

An empty destination file is created: **out = 4**.

Step 2 / 12

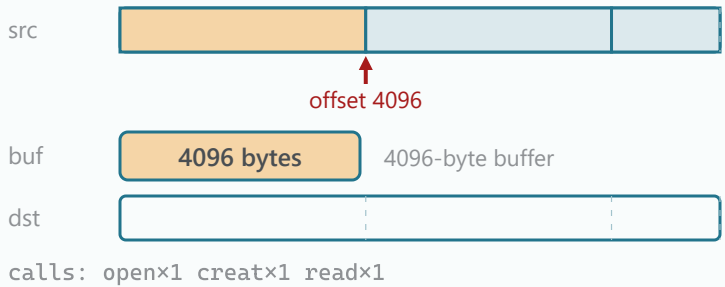
**VARIABLES**

`in = 3` `out = 4` `n = ?` `total = 0`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
▶  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




calls: open×1 creat×1 read×1

`read` enters the kernel and asks for at most 4096 bytes: **n = 4096**. The kernel advances the file offset by 4096.

Step 3 / 12

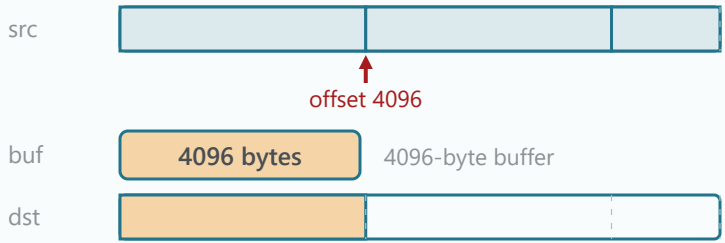
**VARIABLES**

`in = 3` `out = 4` `n = 4096` `total = 0`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×1 write×1`

The **n** bytes in the buffer (as many as were read, not 4096) are written to the destination.

Step 4 / 12

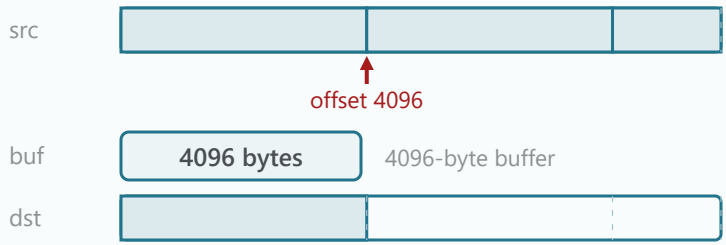
**VARIABLES**

`in = 3` `out = 4` `n = 4096` `total = 0`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
▶    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×1 write×1`

total = 4096. Each iteration costs two system calls: one `read`, one `write`.

Step 5 / 12

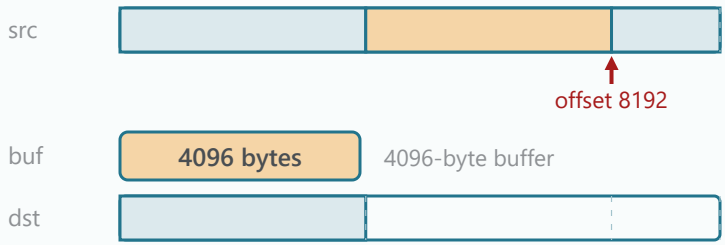
**VARIABLES**

`in = 3` `out = 4` `n = 4096` `total = 4096`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×2 write×1`

`read` again: **n = 4096**, offset 8192.

Step 6 / 12

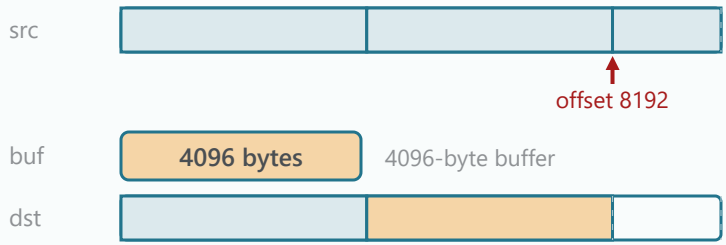
**VARIABLES**

`in = 3` `out = 4` `n = 4096` `total = 4096`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
▶    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×2 write×2`

4096 bytes are written, **total = 8192**.

Step 7 / 12

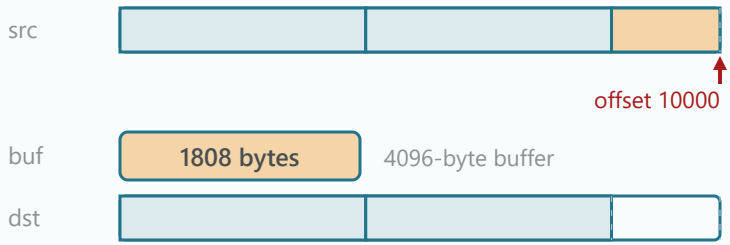
**VARIABLES**

`in = 3` `out = 4` `n = 4096` `total = 8192`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×3 write×2`

Only 10,000 − 8192 = 1808 bytes are left: `read` returns fewer than requested, **n = 1808**. This is not an error.

Step 8 / 12

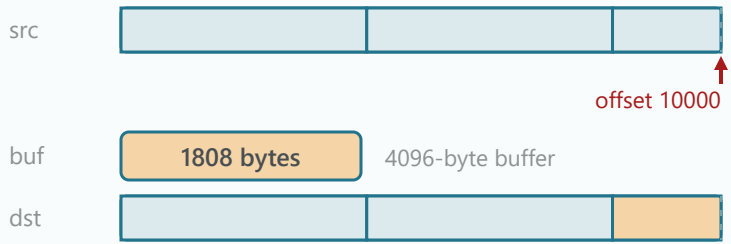
**VARIABLES**

`in = 3` `out = 4` `n = 1808` `total = 8192`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
▶    write(out, buf, n);
▶    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×3 write×3`

1808 bytes are written, **total = 10000**.

Step 9 / 12

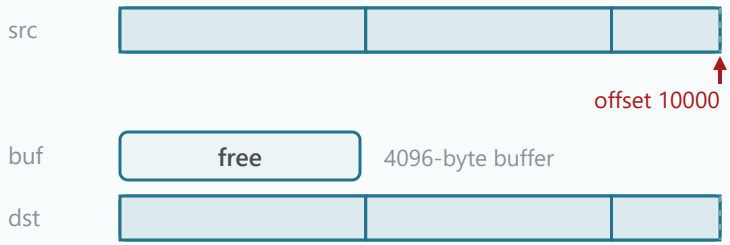
**VARIABLES**

`in = 3` `out = 4` `n = 1808` `total = 10000`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
▶ while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
9  close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×4 write×3`

The offset is at end of file: `read` returns **0** (EOF). The condition is false and the loop ends.

Step 10 / 12

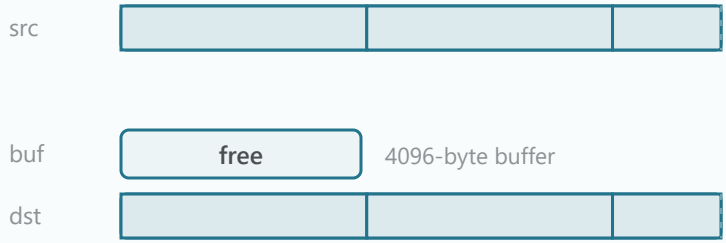
**VARIABLES**

`in = 3` `out = 4` `n = 0` `total = 10000`

**OUTPUT**

# Code Trace: Copying with `read` / `write`

```
1  char buf[4096];
2  int in = open("src", O_RDONLY);
3  int out = creat("dst", 0644);
4  ssize_t n;  long total = 0;
5  while ((n = read(in, buf, 4096)) > 0) {
6    write(out, buf, n);
7    total += n;
8  }
▶ close(in);  close(out);
10 printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×4 write×3 close×2`

Both descriptors are closed; 3 and 4 can be reused.

Step 11 / 12

**VARIABLES**

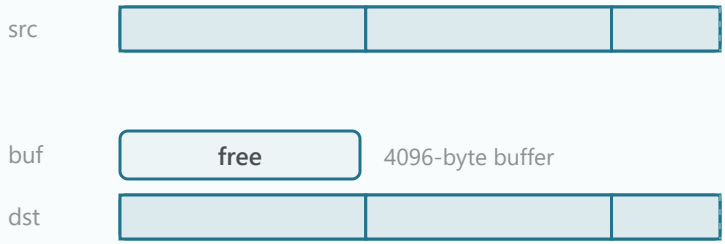
`in = 3` `out = 4` `n = 0` `total = 10000`

**OUTPUT**

```
```

# Code Trace: Copying with `read` / `write`

```
1 char buf[4096];
2 int in = open("src", O_RDONLY);
3 int out = creat("dst", 0644);
4 ssize_t n;  long total = 0;
5 while ((n = read(in, buf, 4096)) > 0) {
6   write(out, buf, n);
7   total += n;
8 }
9 close(in);  close(out);
▶ printf("%ld bytes\n", total);
```




`calls: open×1 creat×1 read×4 write×3 close×2 +write`

Output: **10000 bytes**. 12 system calls in all (printf also does a `write`). With a 1-byte buffer, reading and writing alone would take about 20,000 calls.

Step 12 / 12

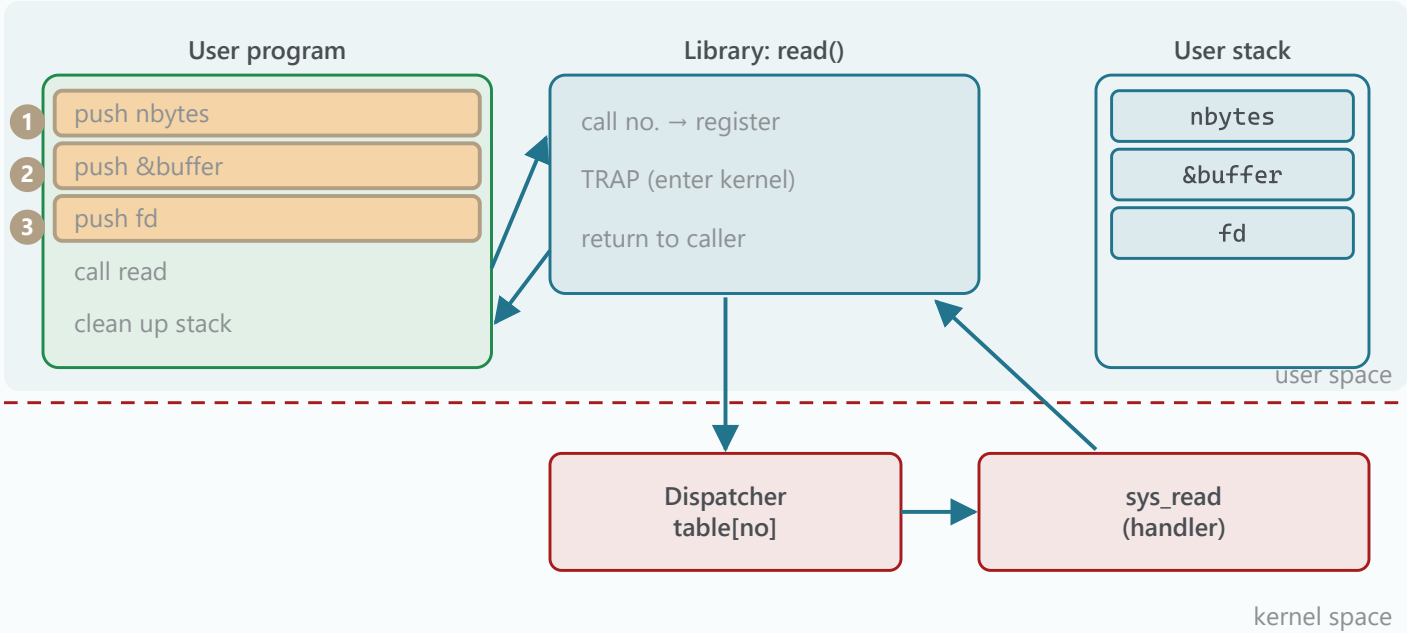
**VARIABLES**

`in = 3` `out = 4` `n = 0` `total = 10000`

**OUTPUT**

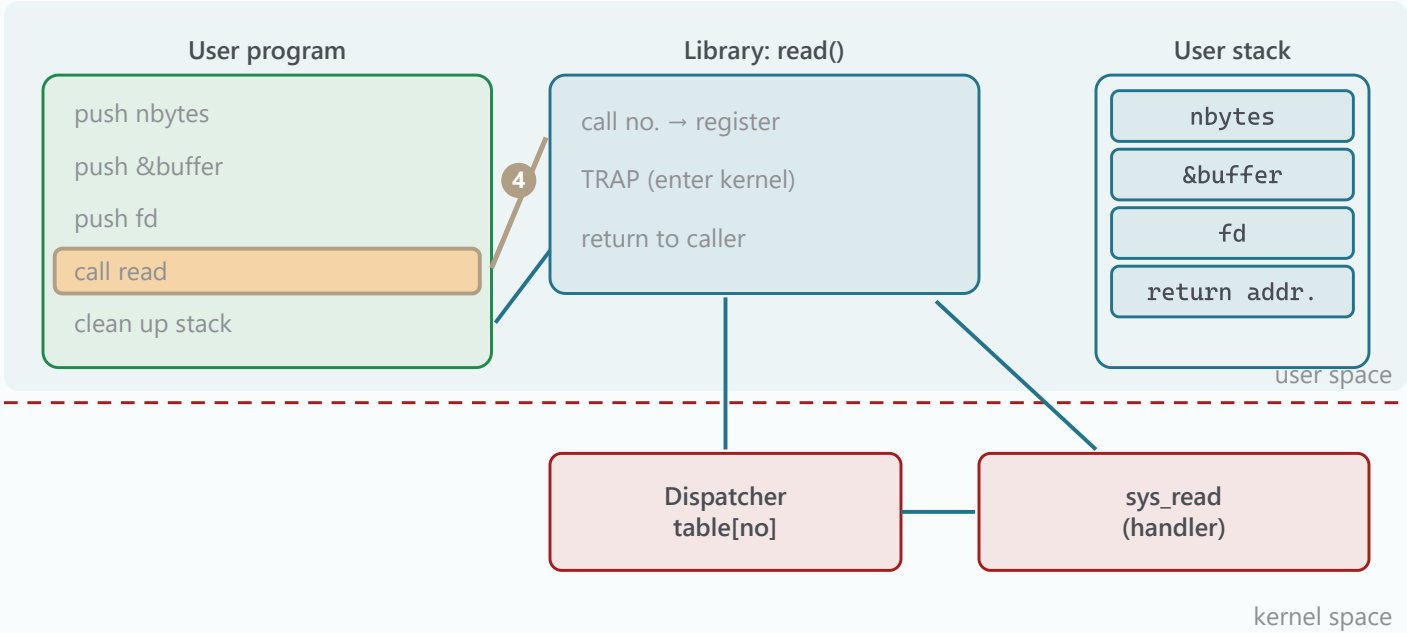
```
10000 bytes
```

# The 11 Steps of `read(fd, buffer, nbytes)`



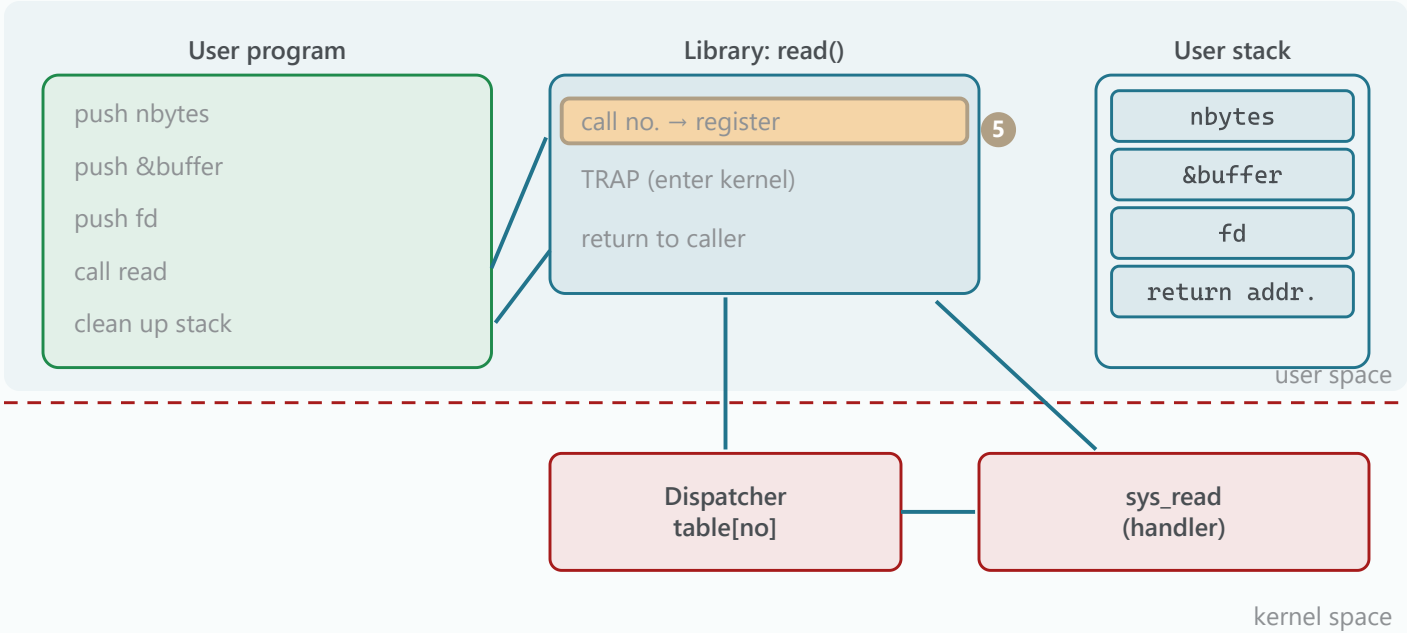

1–3: Parameters are pushed on the stack (the historical 32-bit x86 convention in Tanenbaum's example). On x86-64 the first six parameters go in registers.

# The 11 Steps of `read(fd, buffer, nbytes)`



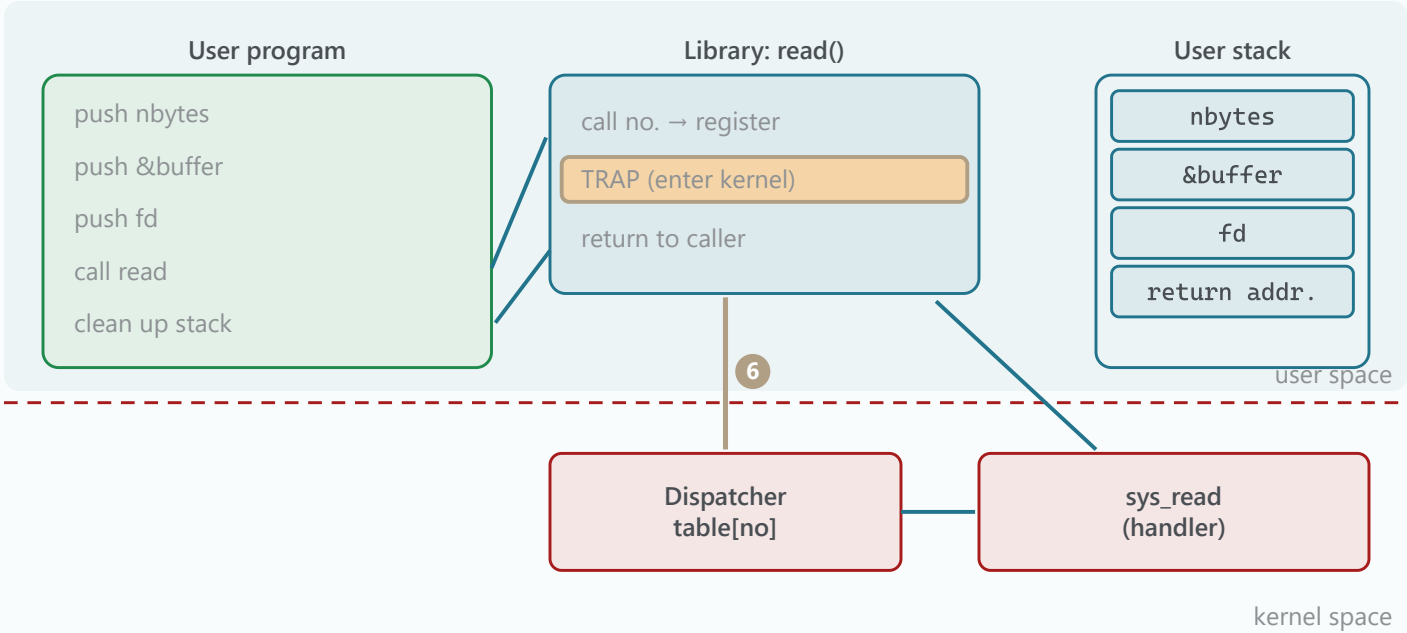

4: The library's `read()` is called like an ordinary function; the return address goes on the stack.

# The 11 Steps of `read(fd, buffer, nbytes)`



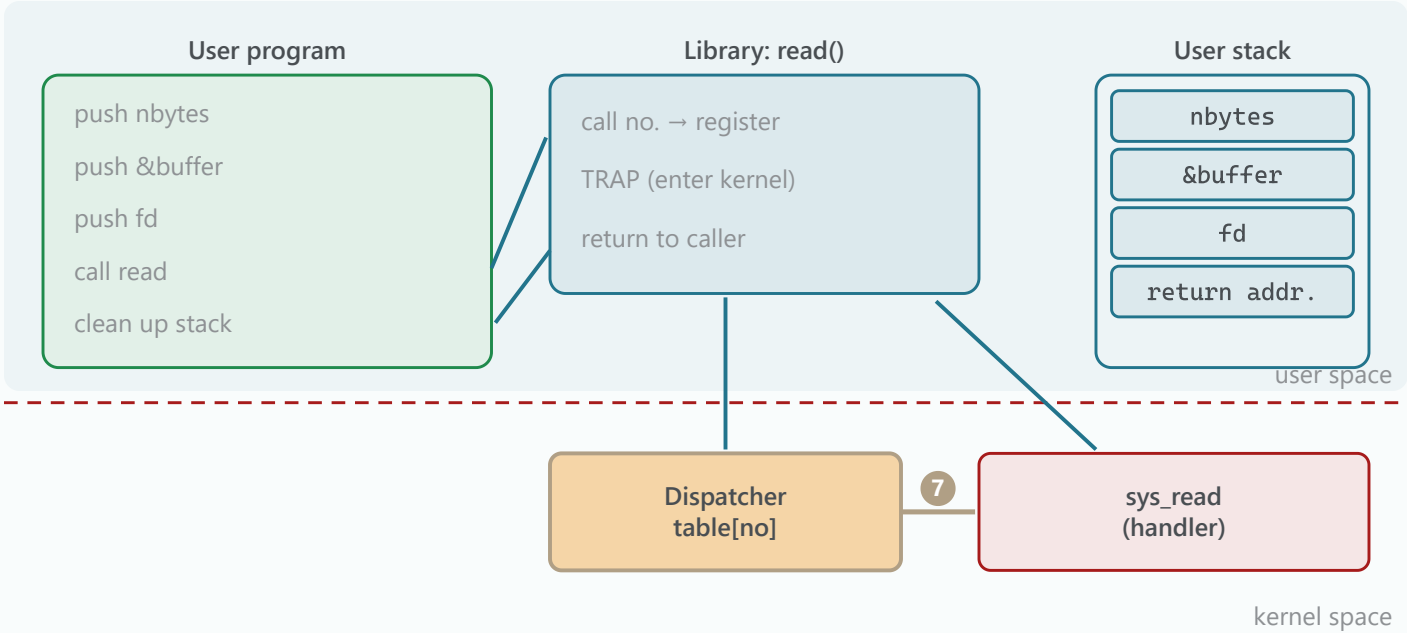

5: The library puts the call number in the register the kernel expects (Linux x86-64: `rax = 0`).

# The 11 Steps of `read(fd, buffer, nbytes)`



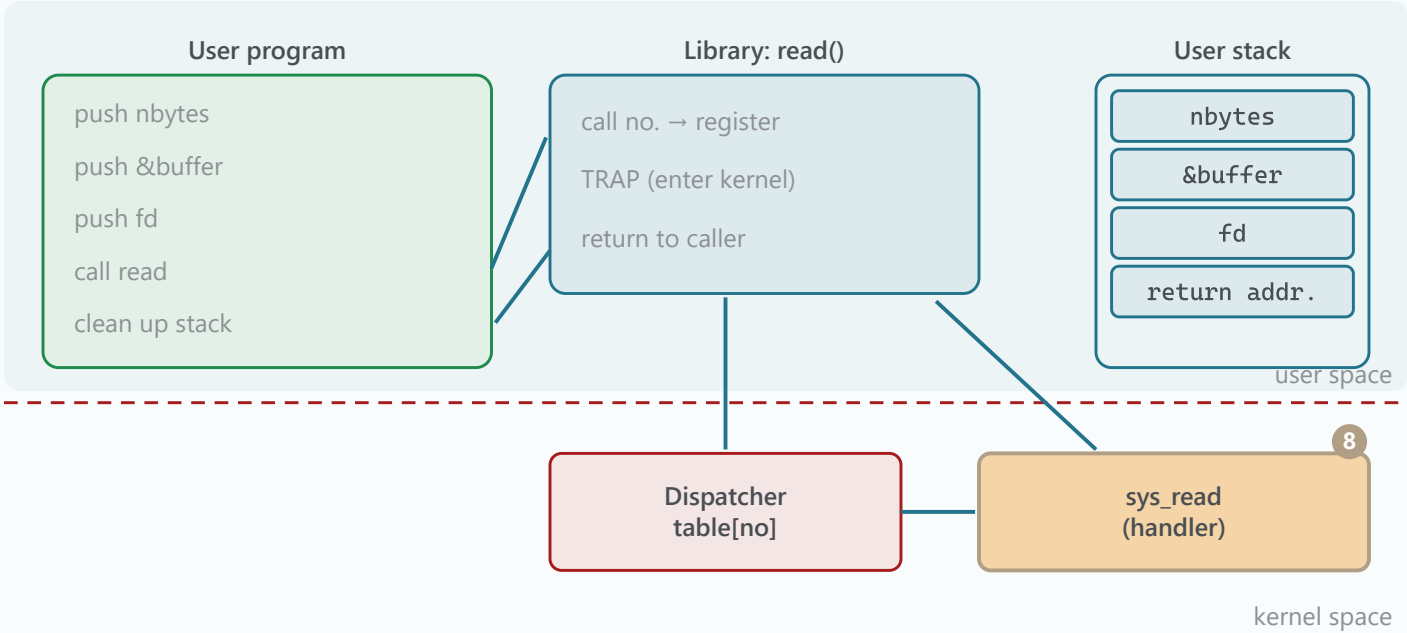

6: The trap instruction (x86-64: `syscall`) runs; the CPU enters kernel mode and jumps to a fixed kernel address.

# The 11 Steps of `read(fd, buffer, nbytes)`



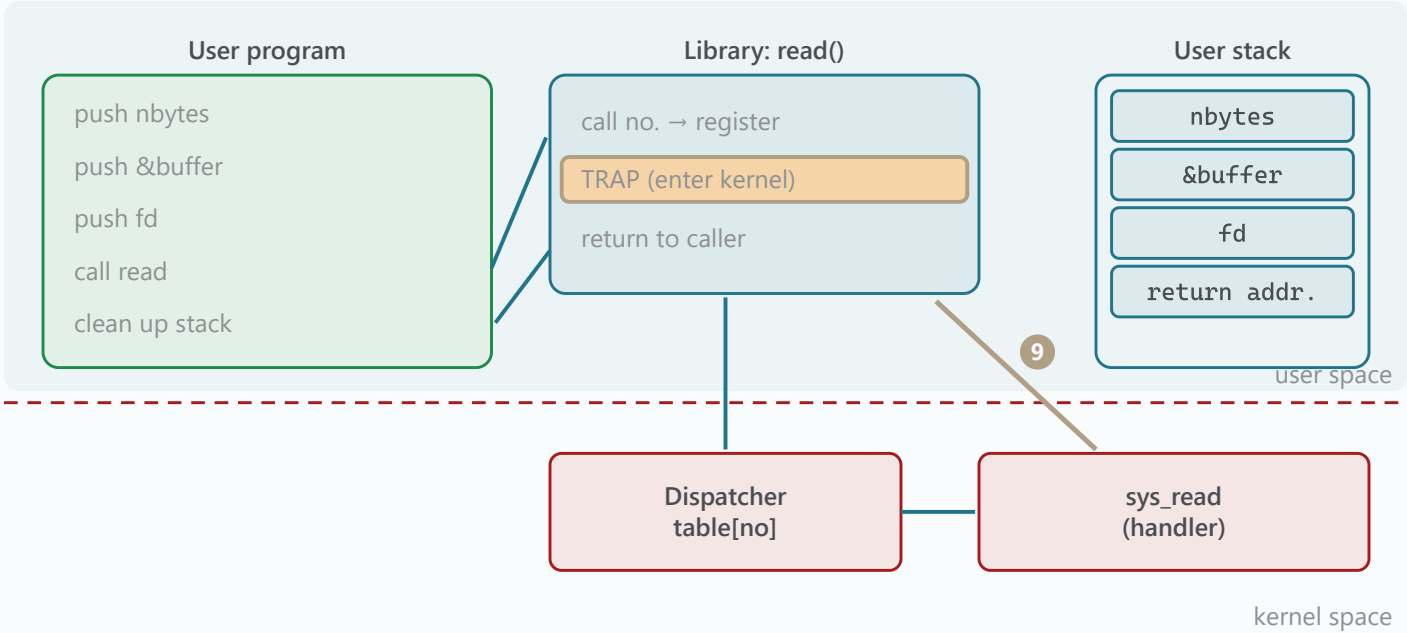

7: The kernel looks at the call number and dispatches to the right handler through the system-call table.

# The 11 Steps of `read(fd, buffer, nbytes)`



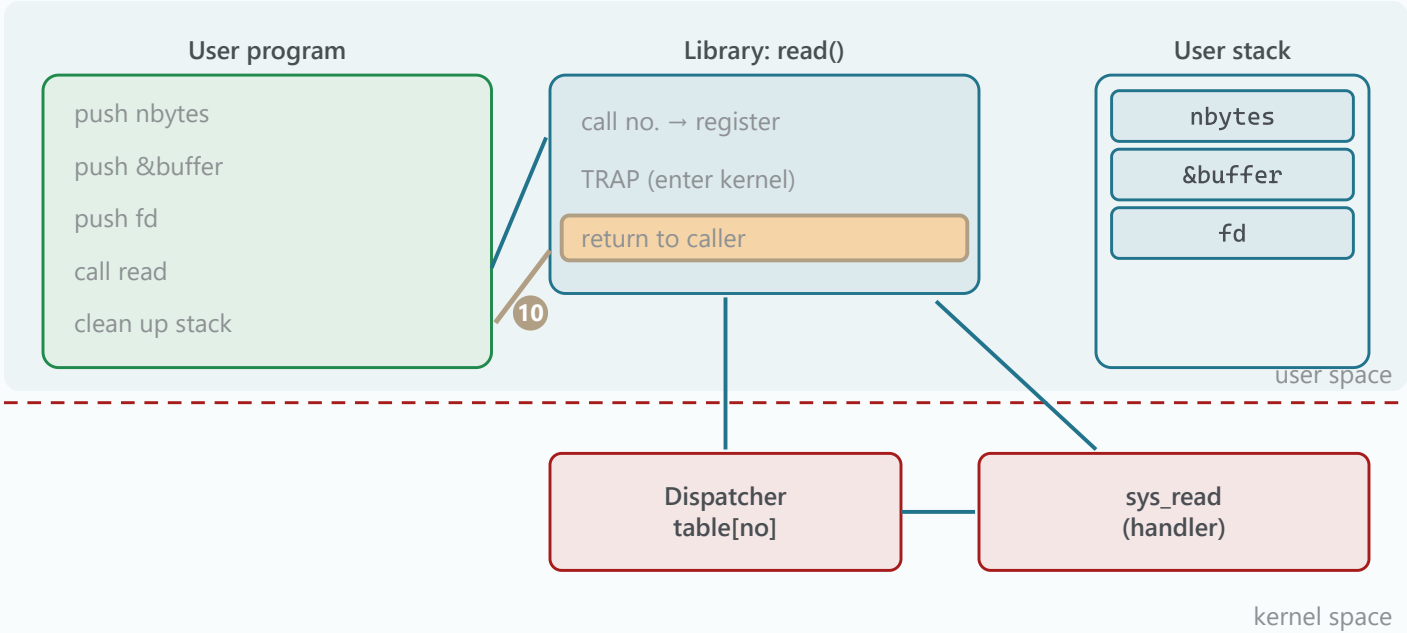

8: The handler copies up to nbytes bytes from fd into buffer. If no data is ready, the process blocks and the CPU goes to another process.

# The 11 Steps of `read(fd, buffer, nbytes)`



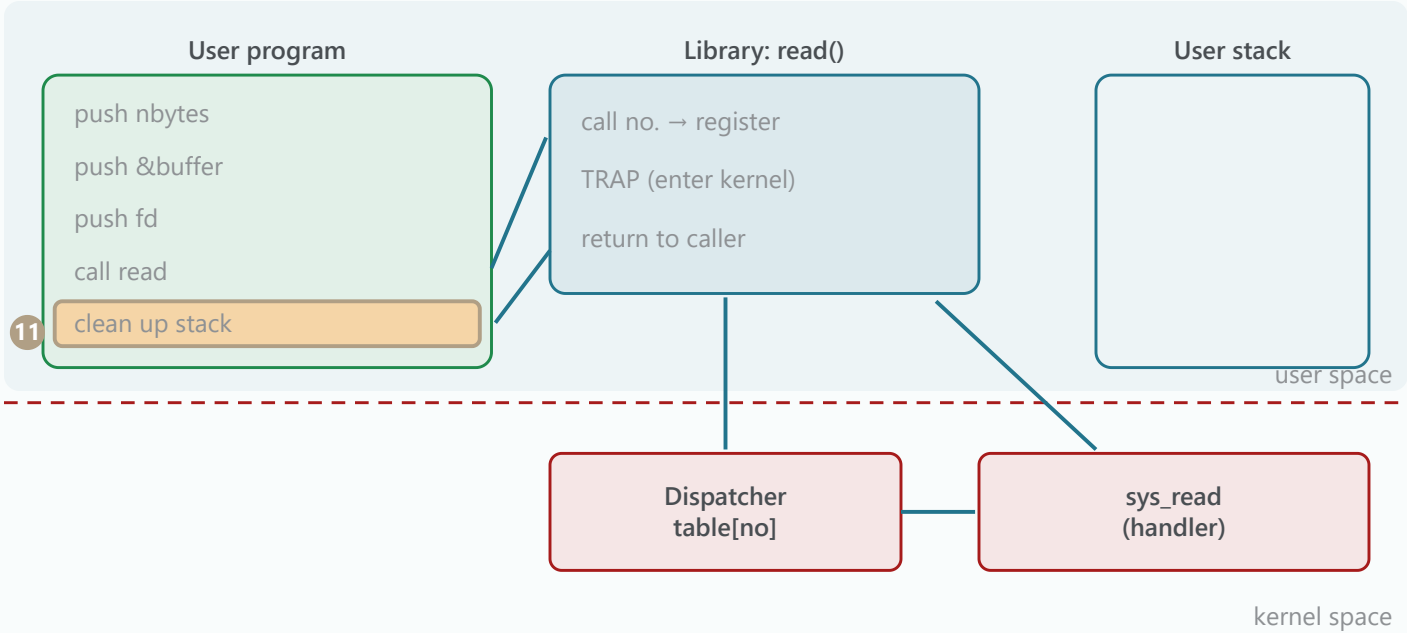

9: When done, control returns to user mode at the instruction after the trap; the result (bytes read) is in a register.

# The 11 Steps of `read(fd, buffer, nbytes)`




10: The library's `read()` returns to the user program.

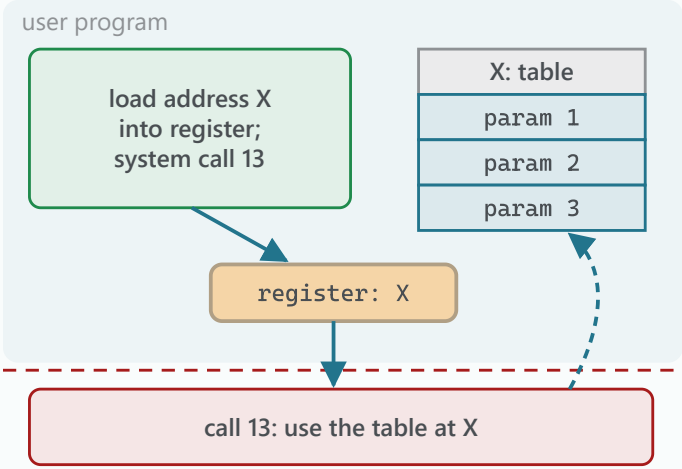
# The 11 Steps of `read(fd, buffer, nbytes)`




11: The program pops the parameters off the stack (SP is incremented) and continues with the next instruction.

# Passing Parameters to System Calls

1. **Registers:** the fastest way; Linux passes up to 6 parameters like this.
2. **Block / table in memory:** parameters in memory, their address in a register (figure). E.g. `clone3(struct clone_args *)`.
3. **Stack:** the program pushes the parameters, the OS pops them.




| x86-64 Linux | Role |
|---|---|
| `rax` | call number; result on return |
| `rdi, rsi, rdx` | 1st, 2nd, 3rd parameter |
| `r10, r8, r9` | 4th, 5th, 6th parameter (`r10` instead of `rcx` from the C convention) |
| `syscall` | enter kernel; `rcx` and `r11` clobbered |
| ARM64 | `x8` = number, `x0–x5` parameters, `svc #0` |

On error the kernel returns a value between −1 and −4095 (−errno); libc stores it in `errno` and returns −1 to the caller.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
▶      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = ?` `n = ?`

**CALL STACK**

`main()`

**OUTPUT**

`msg` points to the string in the read-only data segment.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
▶      size_t len = strlen(msg);        // 17 bytes
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = ?`

**CALL STACK**

`main()`

**OUTPUT**

“Hello, ” 7 + “Türkiye” 8 (ü takes 2 bytes in UTF-8) + “!” and “\n” 2 = **17** bytes. A hard-coded 15 would cut the output to “Hello, Türkiye”.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = ?`

**CALL STACK**

`write(1, msg, 17) [libc]`

`main()`

**OUTPUT**

The libc wrapper: `rax=1` (write), `rdi=1`, `rsi=msg`, `rdx=17`; then `syscall`.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = ?`

**CALL STACK**

- `sys_write [kernel]`
- `write [libc]`
- `main()`

**OUTPUT**

```
Hello, Türkiye!
```

In kernel mode, `sys_write` passes the 17 bytes to the terminal behind fd 1.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
▶      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = 17`

**CALL STACK**

`main()`

**OUTPUT**

```
Hello, Türkiye!
```

Back in user mode: `n = 17`, the number of bytes written. It could be smaller (a partial write).

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
7      ssize_t n = write(1, msg, len);  // 1: stdout
▶      if (n == -1)
9          return 1;                    // error
10     return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = 17`

**CALL STACK**

`main()`

**OUTPUT**

```
Hello, Türkiye!
```

`n` is not −1, so no error. On error it would be −1 and `errno` would be set.

# Code Trace: Writing to the Screen with `write()`

```
1  #include <unistd.h>      // write()
2  #include <string.h>      // strlen()
3
4  int main(void) {
5      const char *msg = "Hello, Türkiye!\n";
6      size_t len = strlen(msg);        // 17 bytes
7      ssize_t n = write(1, msg, len);  // 1: stdout
8      if (n == -1)
9          return 1;                    // error
▶      return 0;
11 }
```

**VARIABLES**

`msg = "Hello, Türkiye!\n"`

`len = 17` `n = 17`

**CALL STACK**

`main()`

**OUTPUT**

```
Hello, Türkiye!
```

`main` returns 0; libc ends the process with `exit_group(0)`.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
▶      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf` = **[100 bytes, uninitialized]**

`n` = **?**

**CALL STACK**

**main()**

**OUTPUT**

`buf` is allocated on the stack; its contents are uninitialized.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf = [100 bytes, uninitialized]`

`n = ?`

**CALL STACK**

`sys_read [kernel]`

`read(0, buf, 100)`

`main()`

**OUTPUT**

fd 0 = stdin (keyboard). No line yet: the process **blocks** and the scheduler runs another process.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf = [100 bytes, uninitialized]`

`n = ?`

**CALL STACK**

`sys_read [kernel]`

`read(0, buf, 100)`

`main()`

**OUTPUT**

```
hello
```

The user types “hello” and presses Enter; the terminal driver echoes the characters and buffers the line.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
▶      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf = "hello\n…" (6 bytes)`

`n = 6`

**CALL STACK**

`main()`

**OUTPUT**

```
hello
```

Once the line is complete, 6 bytes are copied into `buf`, the process wakes up and read returns 6. No `'\0'` is appended.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
▶      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf = "hello\n…" (6 bytes)`

`n = 6`

**CALL STACK**

`main()`

**OUTPUT**

```
hello
```

`n = 6`, no error; `perror` is skipped.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
▶      write(1, buf, n);    // exactly n bytes
12     return 0;
13 }
```

**VARIABLES**

`buf = "hello\n…" (6 bytes)`

`n = 6`

**CALL STACK**

`write(1, buf, 6)`

`main()`

**OUTPUT**

```
hello
hello
```

write outputs exactly 6 bytes. Since buf is not a C string, `printf("%s", buf)` would be wrong.

# Code Trace: `read()` and Blocking

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(void) {
5      char buf[100];
6      ssize_t n = read(0, buf, sizeof(buf));
7      if (n == -1) {
8          perror("read error");
9          return 1;
10     }
11     write(1, buf, n);    // exactly n bytes
▶      return 0;
13 }
```

**VARIABLES**

`buf = "hello\n…" (6 bytes)`

`n = 6`

**CALL STACK**

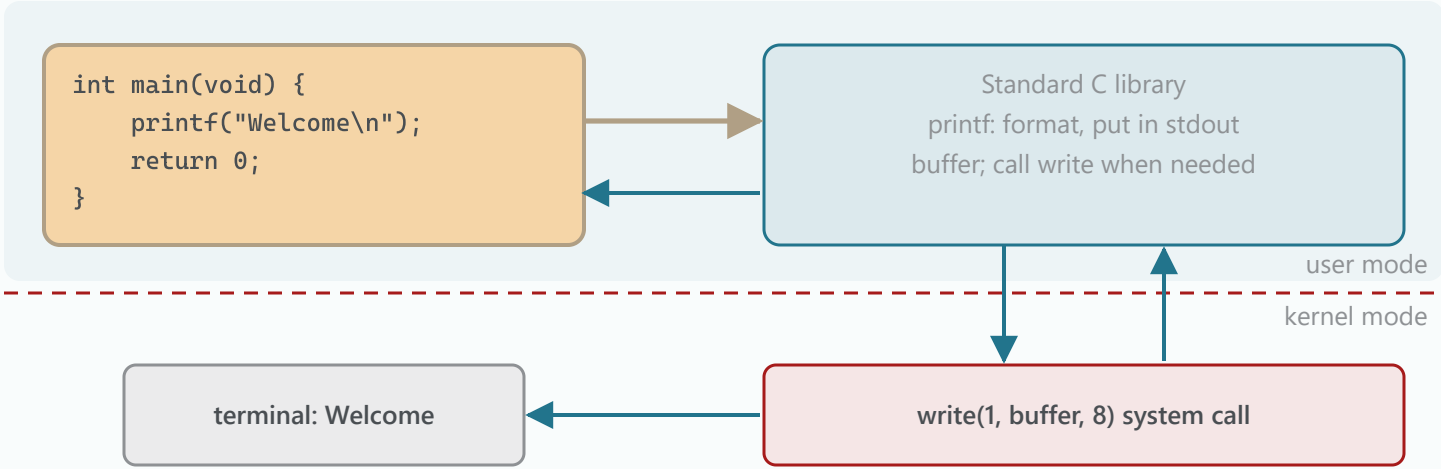
`main()`

**OUTPUT**

```
hello
hello
```

The program exits with 0.

# The Standard C Library and System Calls

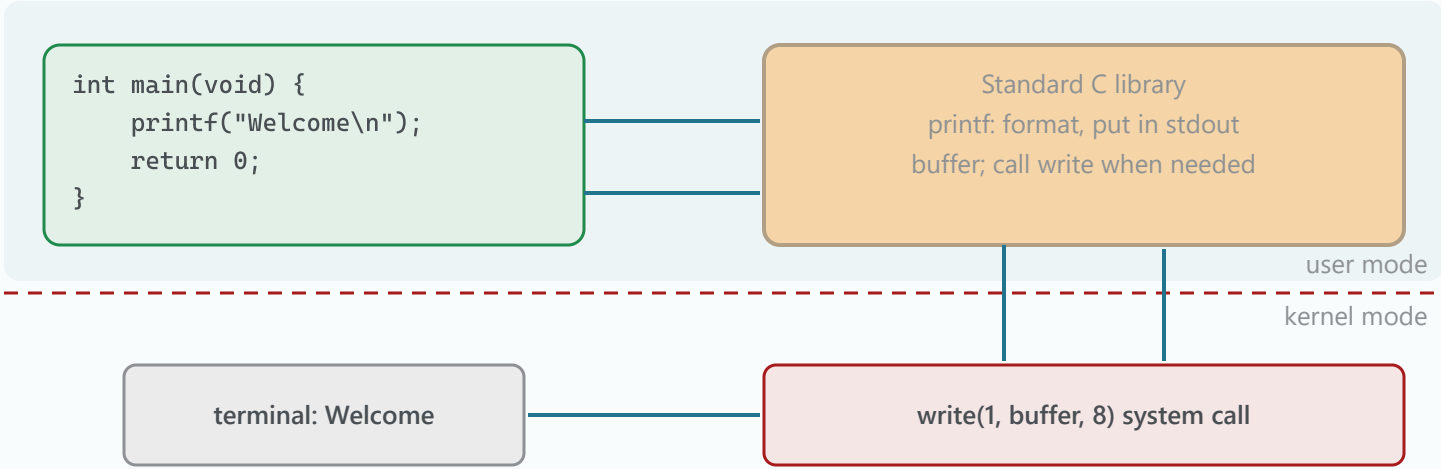



The program calls `printf`; it is not a system call but a library function running in user mode.

Step 1 / 4

Headers: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Implementations: glibc, musl, UCRT on Windows. Functions like `strlen` live entirely in the library; ones like `write` wrap a system call.

# The Standard C Library and System Calls

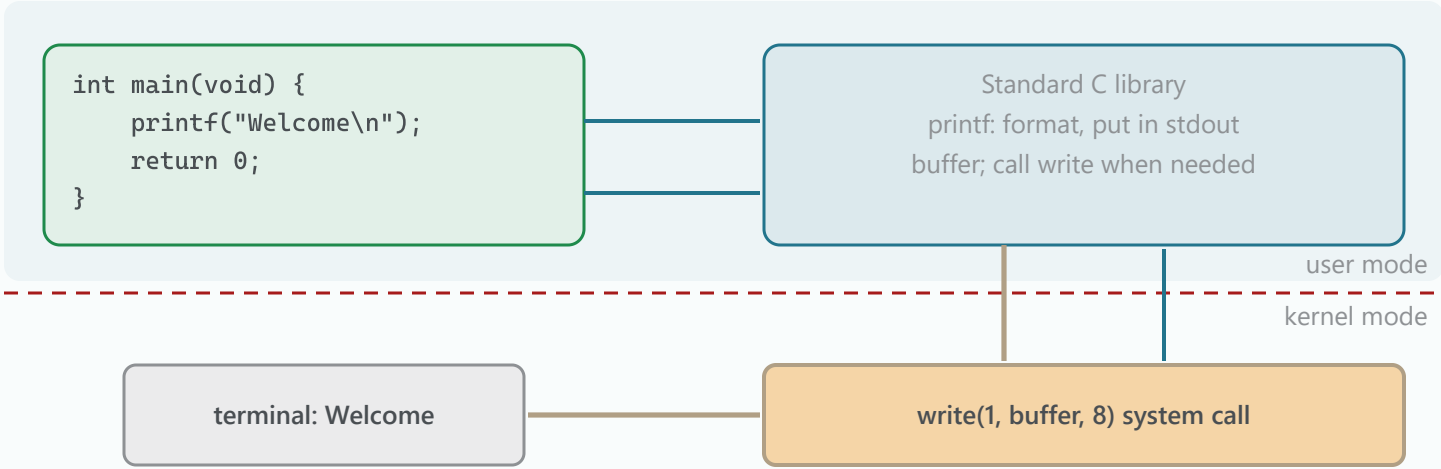



libc formats the text and puts it in the stdout buffer. On a terminal the buffer is **line**-buffered; redirected to a file or pipe it is **fully** buffered.



Headers: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Implementations: glibc, musl, UCRT on Windows. Functions like `strlen` live entirely in the library; ones like `write` wrap a system call.

# The Standard C Library and System Calls

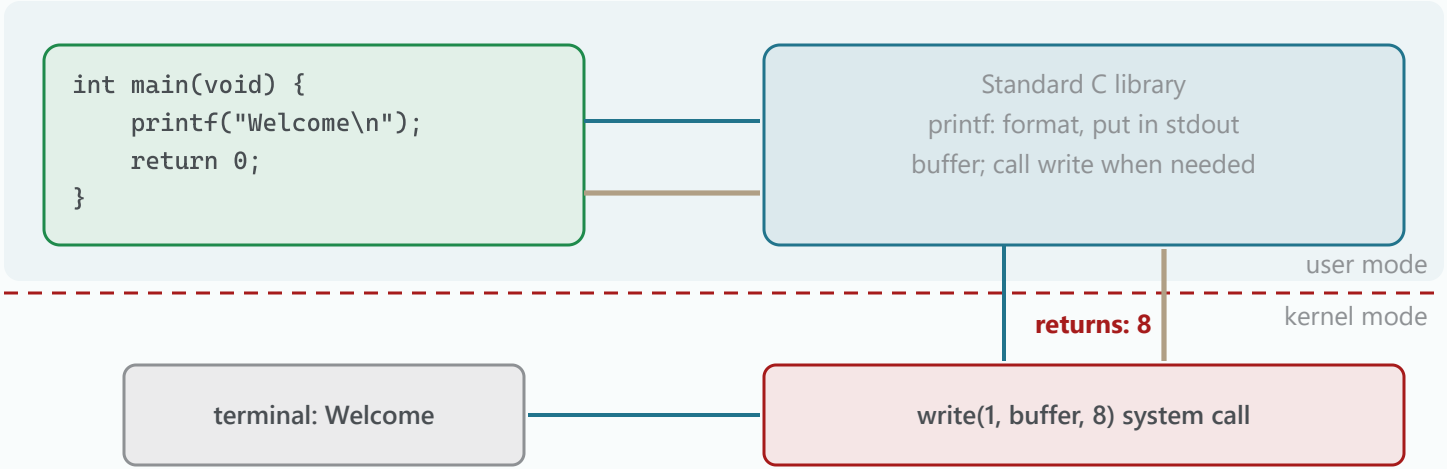



Because “\n” arrived, libc makes the system call `write(1, buffer, 8)`; the kernel passes the bytes to the terminal.



Headers: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Implementations: glibc, musl, UCRT on Windows. Functions like `strlen` live entirely in the library; ones like `write` wrap a system call.

# The Standard C Library and System Calls




write returns 8 and printf returns to its caller. `strace ./welcome` shows only write, not printf.



Headers: `stdio.h` (printf, fopen), `stdlib.h` (malloc, exit), `string.h` (strlen, memcpy), `math.h`, `time.h`; POSIX `unistd.h` (fork, read, write). Implementations: glibc, musl, UCRT on Windows. Functions like `strlen` live entirely in the library; ones like `write` wrap a system call.

# Types of System Calls

**Process control**

create, terminate, load, wait, signal

`fork, execve, exit, waitpid, kill`

**File management**

create, delete, open, close, read, write, attributes

`open, read, write, close, stat`

**Device management**

request / release device, read, write, control

`ioctl, read, write, mmap`

**Information maintenance**

time, date, system and process information

`getpid, uname, clock_gettime`

**Communications**

connect, send/receive messages, shared memory

`pipe, socket, sendmsg, shm_open`

**Protection**

get / set permissions, user identity

`chmod, chown, umask, setuid`

Windows equivalents: CreateProcess, ReadFile, DeviceIoControl, GetSystemTime, CreatePipe, SetFileSecurity…
(Silberschatz, Figure 2.8)

# POSIX Calls: Process Management

| Call | Description |
|---|---|
| `pid = fork()` | Creates a child process that is a copy of the parent. |
| `pid = waitpid(pid, &statloc, options)` | Waits for a child to terminate; writes its exit status to `statloc`. |
| `s = execve(name, argv, envp)` | Replaces the process's memory image with a new program; does not return on success. |
| `exit(status)` | Terminates the process and passes the status to the parent. |




On error the calls return **−1** and set `errno`. `pid`: process ID, `s`: return code. On Windows a single `CreateProcess` call does the work of fork + execve.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                /* forever */
▶      type_prompt();            /* show prompt */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* parent */
8          waitpid(-1, &status, 0);
9      } else {                  /* child */
10         execve(command, parameters, 0);
11         perror("execve");     /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = ?` `status = ?`

**CALL STACK**

`shell (812)`

**OUTPUT**

```
$
```

The shell prints a prompt and waits for the user.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                /* forever */
4      type_prompt();            /* show prompt */
▶      read_command(command, parameters);
6
7      if (fork() != 0) {        /* parent */
8          waitpid(-1, &status, 0);
9      } else {                  /* child */
10         execve(command, parameters, 0);
11         perror("execve");     /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

**CALL STACK**

`shell (812)`

**OUTPUT**

```
$ ls
```

The user types `ls`; the command and its parameters are parsed.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                  /* forever */
4      type_prompt();              /* show prompt */
5      read_command(command, parameters);
6
▶      if (fork() != 0) {          /* parent */
8          waitpid(-1, &status, 0);
9      } else {                    /* child */
10         execve(command, parameters, 0);
11         perror("execve");       /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→parent = 1045` `fork→child = 0`

**CALL STACK**

- **child (1045)**
- shell (812)

**OUTPUT**

```
$ ls
```

`fork` makes a copy of the shell. In the parent the result is 1045 (≠ 0); in the child it is 0.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                  /* forever */
4      type_prompt();              /* show prompt */
5      read_command(command, parameters);
6
7      if (fork() != 0) {          /* parent */
▶          waitpid(-1, &status, 0);
9      } else {                    /* child */
10         execve(command, parameters, 0);
11         perror("execve");       /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→parent = 1045` `fork→child = 0`

**CALL STACK**

- **child (1045)**
- shell (812) [waiting]

**OUTPUT**

```
$ ls
```

The parent waits for any child to finish with `waitpid(-1, …)`; it blocks.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                  /* forever */
4      type_prompt();              /* show prompt */
5      read_command(command, parameters);
6
7      if (fork() != 0) {          /* parent */
8          waitpid(-1, &status, 0);
9      } else {                    /* child */
▶          execve(command, parameters, 0);
11         perror("execve");       /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = ?`

`fork→parent = 1045` `fork→child = 0`

**CALL STACK**

`ls (1045)`

`shell (812) [waiting]`

**OUTPUT**

```
$ ls
notes.txt  hw.c
```

The child replaces itself with the ls program via `execve` (the PID stays the same); ls prints its output. Real shells search PATH for the command.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                /* forever */
4      type_prompt();            /* show prompt */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* parent */
▶          waitpid(-1, &status, 0);
9      } else {                  /* child */
10         execve(command, parameters, 0);
11         perror("execve");     /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = 0`

**CALL STACK**

`shell (812)`

**OUTPUT**

```
$ ls
notes.txt  hw.c
```

ls finishes with `exit(0)`; the parent's waitpid returns and `status` holds the exit status.

# Code Trace: A Simple Shell

```
1  #define TRUE 1
2
3  while (TRUE) {                /* forever */
▶      type_prompt();            /* show prompt */
5      read_command(command, parameters);
6
7      if (fork() != 0) {        /* parent */
8          waitpid(-1, &status, 0);
9      } else {                  /* child */
10         execve(command, parameters, 0);
11         perror("execve");     /* on failure */
12         _exit(127);
13     }
14 }
```

**VARIABLES**

`command = "/bin/ls"`

`parameters = ["ls", NULL]` `status = 0`

**CALL STACK**

**shell (812)**

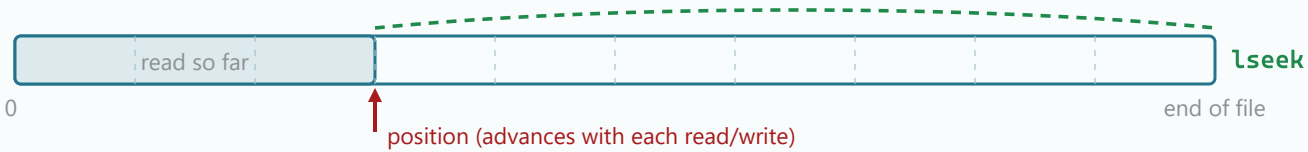
**OUTPUT**

```
$ ls
notes.txt  hw.c
$
```

The loop starts over and a new prompt is printed.

# POSIX Calls: File Management

| Call | Description |
|---|---|
| `fd = open(file, how, …)` | Opens a file for reading, writing or both (creating it if needed). |
| `s = close(fd)` | Closes an open file. |
| `n = read(fd, buffer, nbytes)` | Reads up to nbytes bytes from the file into the buffer. |
| `n = write(fd, buffer, nbytes)` | Writes data from the buffer to the file. |
| `position = lseek(fd, offset, whence)` | Moves the file position pointer ( `SEEK_SET` , `SEEK_CUR` , `SEEK_END` ). |
| `s = stat(name, &buf)` | Gets the file's status information (size, permissions, times). |




fd: file descriptor, n: byte count, position: offset from the start of the file. Current Linux also offers `openat` , `pread` / `pwrite` , `statx` .

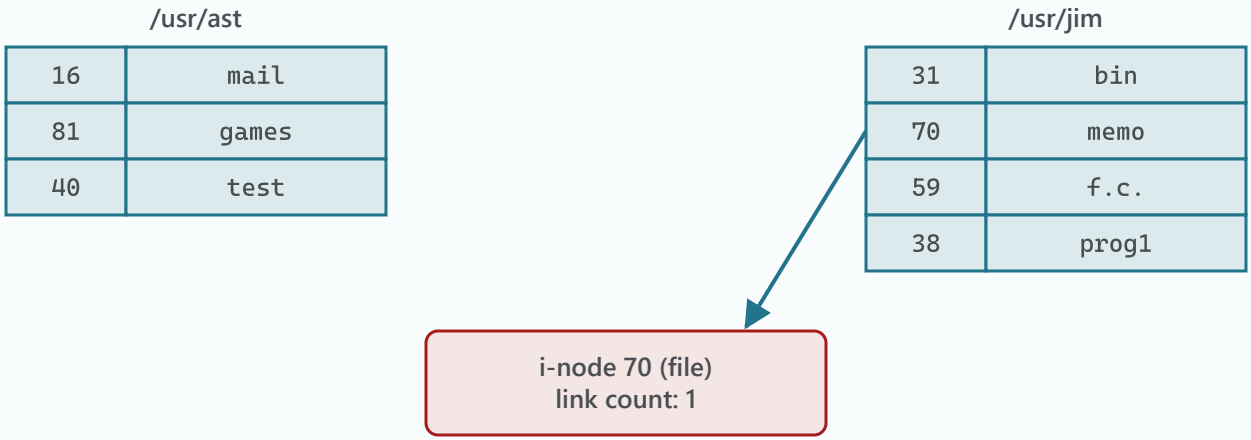
# POSIX Calls: Directory Management and Others

| Call | Description |
|---|---|
| `s = mkdir(name, mode)` | Creates a new directory. |
| `s = rmdir(name)` | Removes an empty directory. |
| `s = link(name1, name2)` | Creates a new directory entry name2 pointing to the file name1 refers to. |
| `s = unlink(name)` | Removes a directory entry. |
| `s = mount(special, name, flag)` | Mounts a file system (Linux: `mount(src, target, type, flags, data)`). |
| `s = umount(special)` | Unmounts a file system. |
| `s = chdir(dirname)` | Changes the working directory. |
| `s = chmod(name, mode)` | Changes a file's protection (permission) bits. |
| `s = kill(pid, signal)` | Sends a signal to a process (not only to kill it). |
| `seconds = time(&seconds)` | Seconds elapsed since 1 January 1970 00:00 UTC (the Unix epoch). |

A 32-bit `time_t` overflows on 19 January 2038 (the year-2038 problem); current systems use a 64-bit `time_t`.

# `link` and `unlink`: One File, Two Names

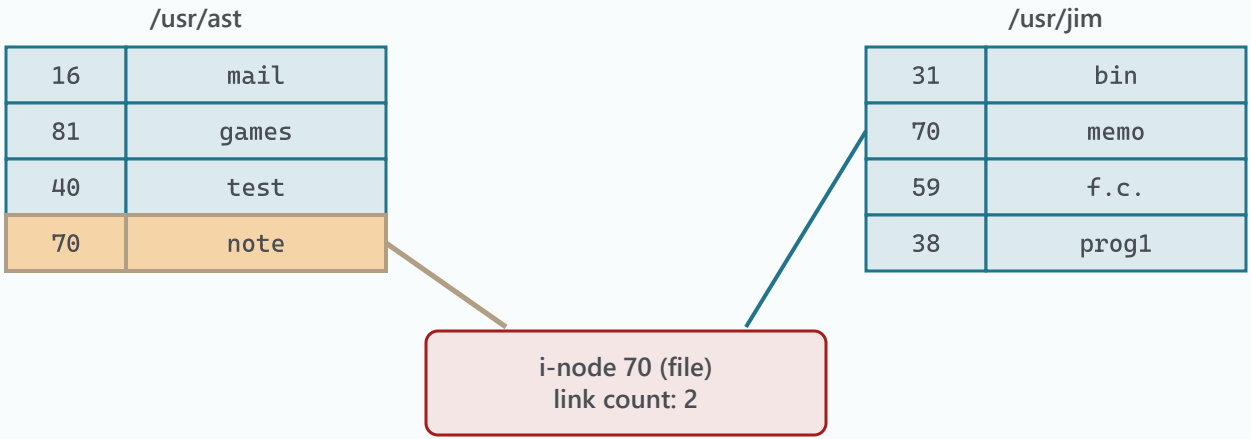
In UNIX a directory is a table of (i-node number, name) pairs. The file's real information lives in the i-node (inode).




Start: `/usr/jim/memo` refers to i-node 70; link count 1.

# `link` and `unlink`: One File, Two Names

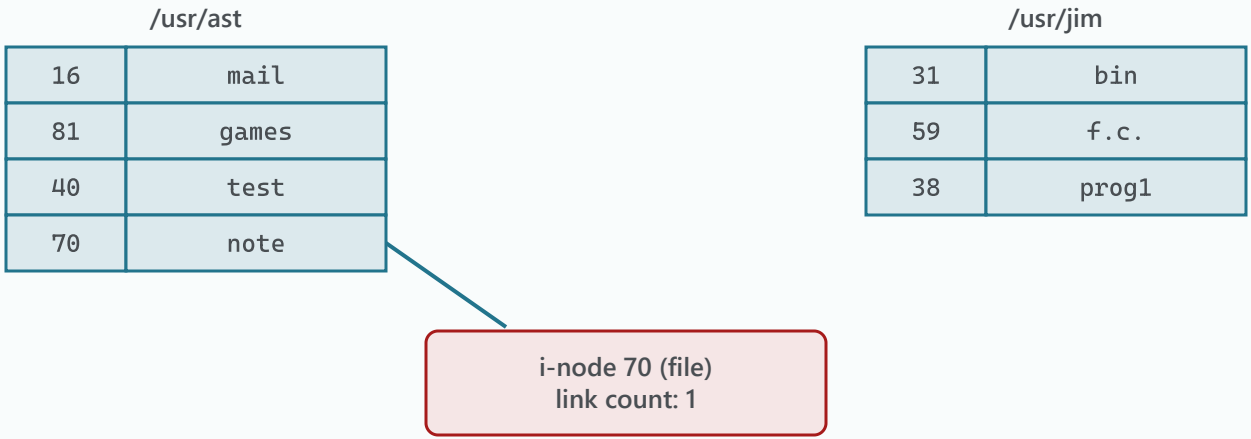
In UNIX a directory is a table of (i-node number, name) pairs. The file's real information lives in the i-node (inode).




`link("/usr/jim/memo", "/usr/ast/note")`: a new entry referring to the same i-node (70) is added to /usr/ast. No data is copied; the count becomes 2.

# `link` and `unlink`: One File, Two Names

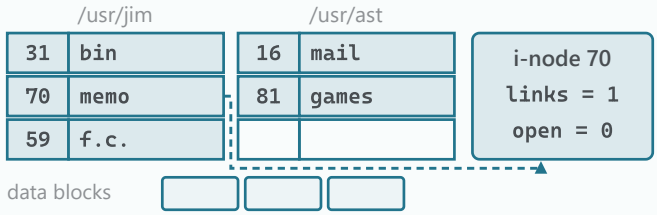
In UNIX a directory is a table of (i-node number, name) pairs. The file's real information lives in the i-node (inode).




`unlink("/usr/jim/memo")` removes only the entry; the file lives on as `note`. When the count reaches 0 (and no one holds it open) its blocks are freed.

# Code Trace: Link Count and an Open File

```
▶ int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




A directory is just (i-node number, name) pairs. `memo` refers to i-node 70; link count 1, file size 85 bytes.

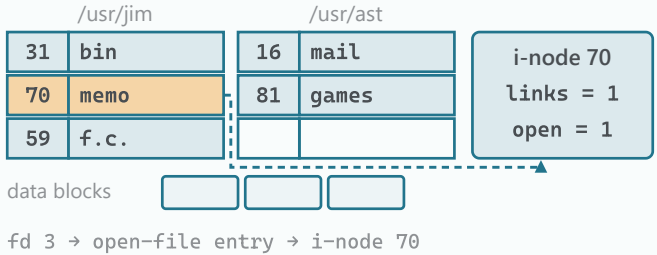
Step 1 / 7

**VARIABLES**

`fd = ?` `n = ?`

# Code Trace: Link Count and an Open File

```
▶ int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




`open` resolves the path and opens i-node 70: **fd = 3**, the i-node's open count is 1.

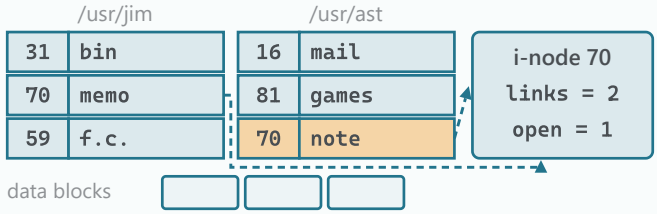
Step 2 / 7

**VARIABLES**

`fd = 3` `n = ?`

# Code Trace: Link Count and an Open File

```
1 int fd = open("/usr/jim/memo", O_RDONLY);
▶ link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




`fd 3 → open-file entry → i-node 70`

`link` adds a new entry in /usr/ast referring to **the same i-node**. No data is copied; links = 2.

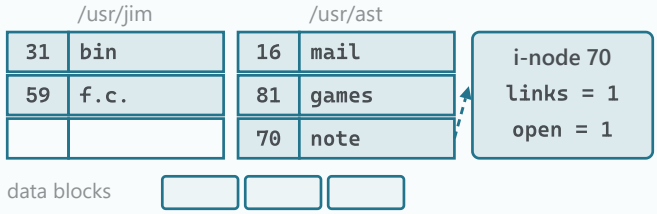
Step 3 / 7

**VARIABLES**

`fd = 3` `n = ?`

# Code Trace: Link Count and an Open File

```
1 int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
▶ unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




`fd 3 → open-file entry → i-node 70`

`unlink` removes only the directory entry; the file lives on as `note`. Links = 1.

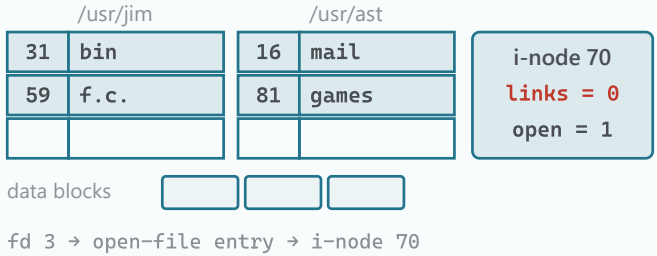
Step 4 / 7

**VARIABLES**

`fd = 3` `n = ?`

# Code Trace: Link Count and an Open File

```
1 int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
▶ unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




The last name is gone too: **links = 0**. The file has no path any more, but since fd 3 holds it open, the i-node and blocks are **not deleted**.

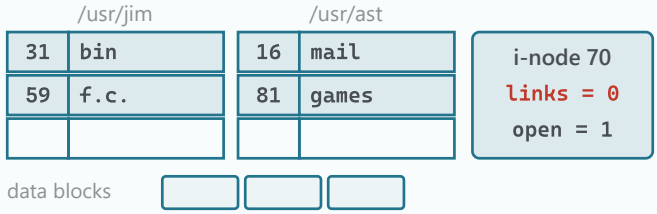
Step 5 / 7

**VARIABLES**

`fd = 3` `n = ?`

# Code Trace: Link Count and an Open File

```
1 int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
▶ n = read(fd, buf, 100);     /* still readable */
6 close(fd);
```




`fd 3 → open-file entry → i-node 70`

`read` goes straight to the i-node through the descriptor; no name is needed: **n = 85**.

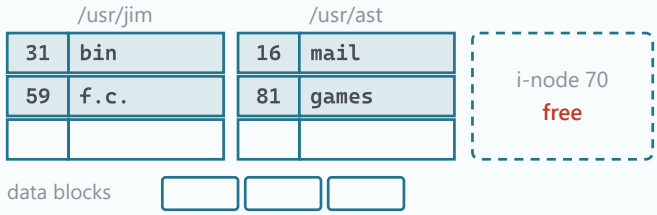
Step 6 / 7

**VARIABLES**

`fd = 3` `n = 85`

# Code Trace: Link Count and an Open File

```
1 int fd = open("/usr/jim/memo", O_RDONLY);
2 link("/usr/jim/memo", "/usr/ast/note");
3 unlink("/usr/jim/memo");
4 unlink("/usr/ast/note");
5 n = read(fd, buf, 100);     /* still readable */
▶ close(fd);
```




The last descriptor is closed: links = 0 and open = 0. Only now does the kernel **free** the i-node and data blocks. Temporary files are deleted while open this way.

Step 7 / 7

**VARIABLES**

`fd = 3` `n = 85`

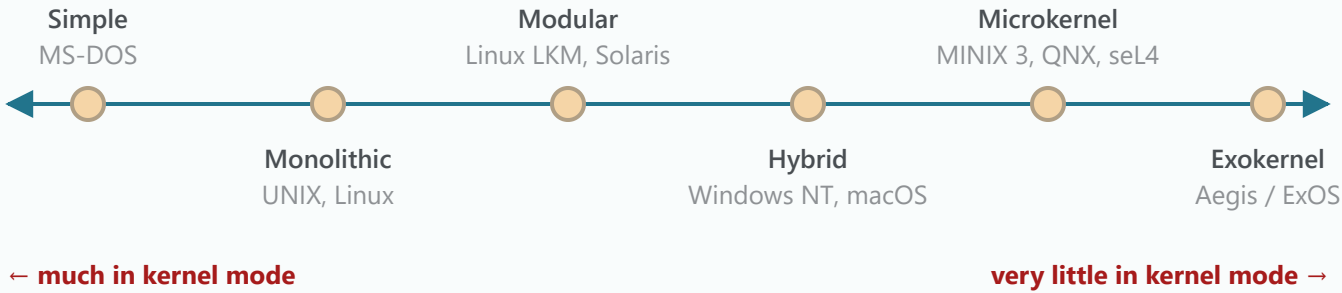
# Win32 API Equivalents of UNIX Calls

| UNIX | Win32 |
|---|---|
| `fork` | `CreateProcess` |
| `waitpid` | `WaitForSingleObject` |
| `execve` | (none: CreateProcess) |
| `exit` | `ExitProcess` |
| `open` | `CreateFile` |
| `close` | `CloseHandle` |
| `read` | `ReadFile` |
| `write` | `WriteFile` |
| `lseek` | `SetFilePointerEx` |
| `stat` | `GetFileAttributesEx` |

| UNIX | Win32 |
|---|---|
| `mkdir` | `CreateDirectory` |
| `rmdir` | `RemoveDirectory` |
| `link` | `CreateHardLink` |
| `unlink` | `DeleteFile` |
| `mount` | `SetVolumeMountPoint` |
| `umount` | `DeleteVolumeMountPoint` |
| `chdir` | `SetCurrentDirectory` |
| `chmod` | (none: ACLs, SetFileSecurity) |
| `kill` | (none: TerminateProcess) |
| `time` | `GetSystemTime` |

The Win32 API has thousands of functions; the actual kernel transition is done by the `Nt…` functions in `ntdll.dll` (e.g. `NtCreateFile`). Their numbers change between versions and are undocumented, so programs use the API. The `link` and `mount` equivalents listed as “none” in Tanenbaum 3e exist today.

# Operating-System Structures: Overview



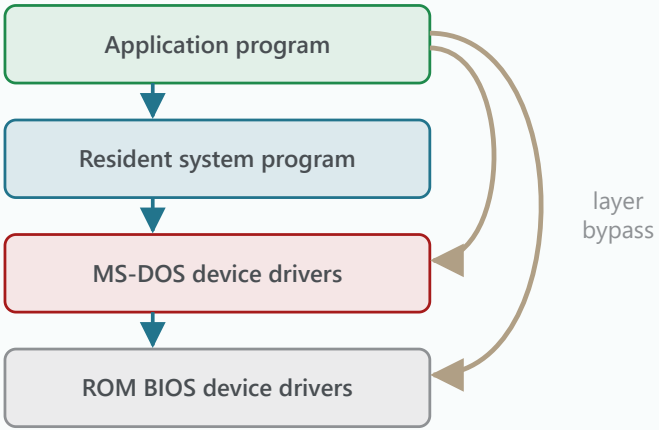

### Layered approach

Not a placement but an organizing principle: components are split into hierarchical layers (THE, MULTICS rings).

### Virtual machines

A separate axis: the hardware itself is virtualized and several OSes run on top of it.

# Simple Structure: MS-DOS



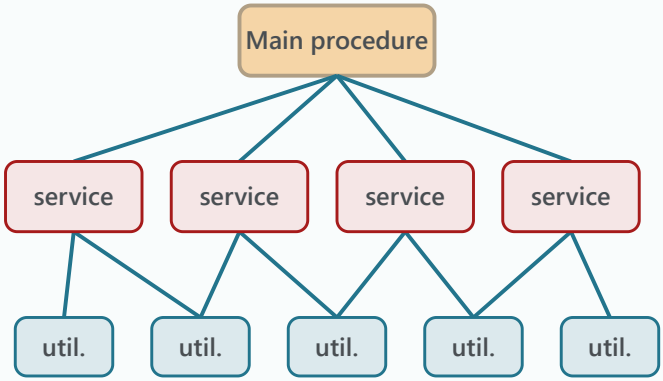

- No clear structure: it was written to be small and simple, for limited hardware.
- Interfaces and levels of functionality are not well separated; applications can reach BIOS routines and the hardware **directly**.
- The Intel 8088 had no dual mode, so there was no hardware protection: one faulty program could **crash the whole system**.
- No clear boundaries between modules and no data hiding.
- Silberschatz treats this as a “simple structure”, separate from monolithic kernels.

# Monolithic Systems

The whole OS is linked as **a single program** running in kernel mode. Any procedure can call any other directly.

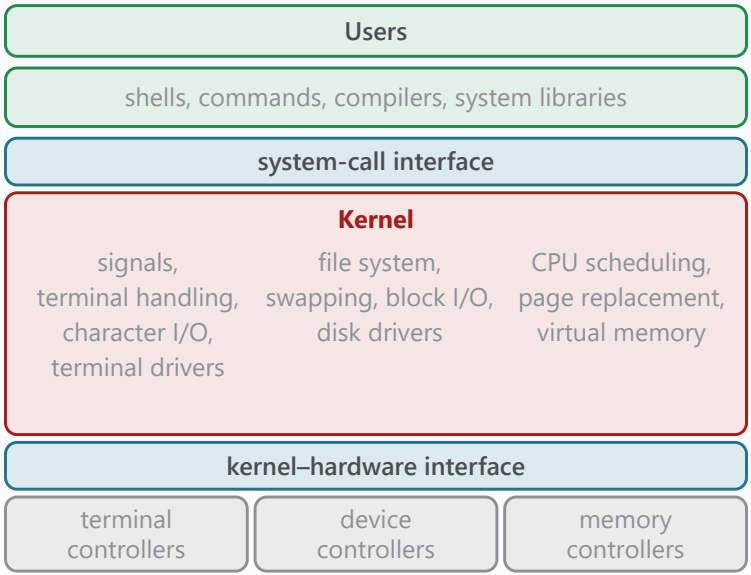
1. **Main program:** looks at the call number and invokes the requested service procedure.
2. **Service procedures:** carry out the system calls.
3. **Utility procedures:** work shared by several services (e.g. fetching data from user memory).

Most monolithic kernels today also accept drivers and file systems as **loadable modules** at run time.




all one binary, one address space, kernel mode

# Traditional UNIX and Linux: The Monolithic Kernel



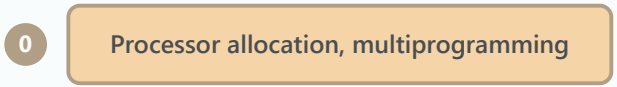

## Strengths

+ Direct function calls: no IPC, no mode switches, **fast**.
+ Relatively simple, integrated design and implementation.

## Weaknesses

– One driver bug can crash **the whole kernel**.
– A huge code base (Linux passed 40 million lines in 2025): debugging and maintenance are hard.

# Layered Approach




Layer 0: switches between processes on interrupts and timer expiry; layers above need not think about the single CPU.



The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.

### Pros

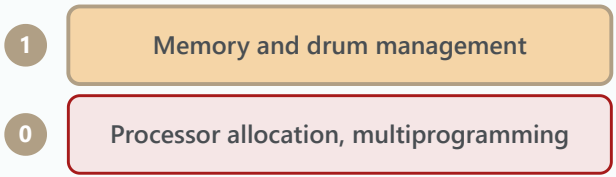
+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Layered Approach

The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.




Layer 1: allocates memory to processes and moves pages between the drum (the disk of the day) and memory.



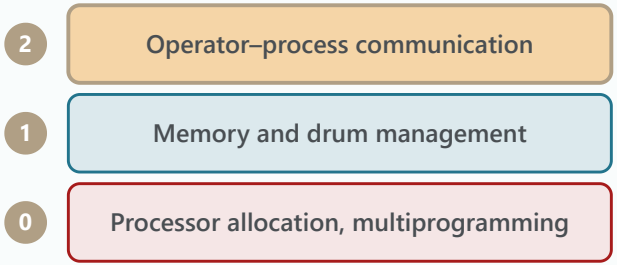
### Pros

+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Layered Approach




Layer 2: handles communication between each process and the operator console.



The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.

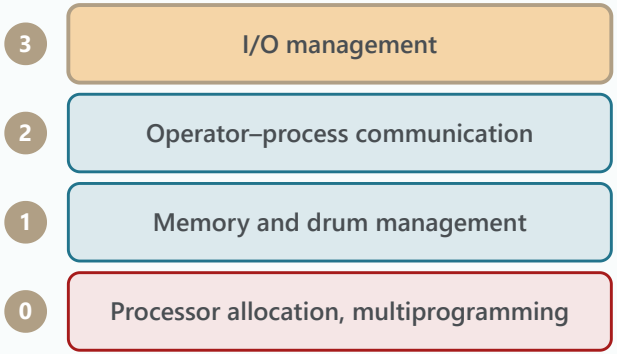
### Pros

+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Layered Approach




Layer 3: manages I/O devices; layers above work with abstract devices.



The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.

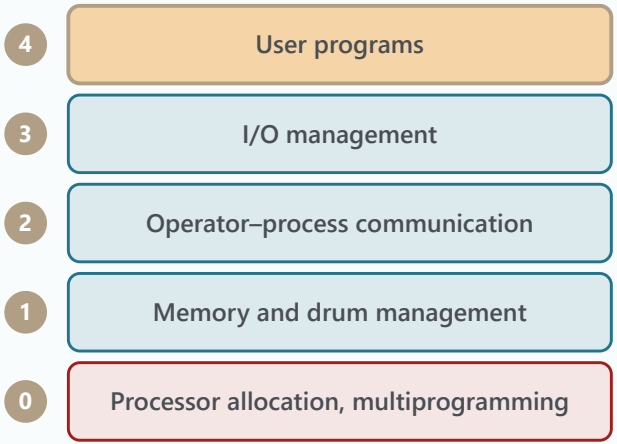
### Pros

+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Layered Approach




Layer 4: user programs; they need not deal with memory, console or I/O details.



The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.

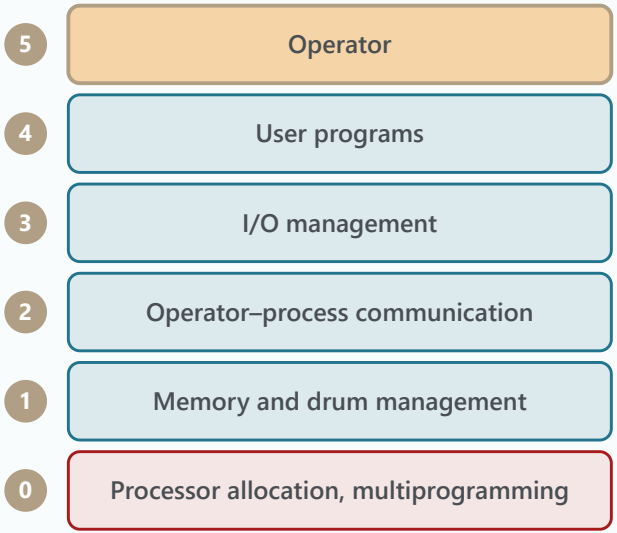
### Pros

+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Layered Approach




Layer 5: the system operator. Dijkstra's THE system (1968) had these six layers.



The OS is divided into layers: the bottom layer is closest to the hardware, the top to the user. Each layer uses the services of **only the layers below it**.

### Pros

+ Debugging and verification proceed layer by layer; a bug is sought in the relevant layer.
+ Abstraction and data hiding come easily.

### Cons

– Defining the layers properly is hard (dependencies often run both ways).
– Each layer crossing adds overhead. The idea lives on today in network stacks.

# Microkernels

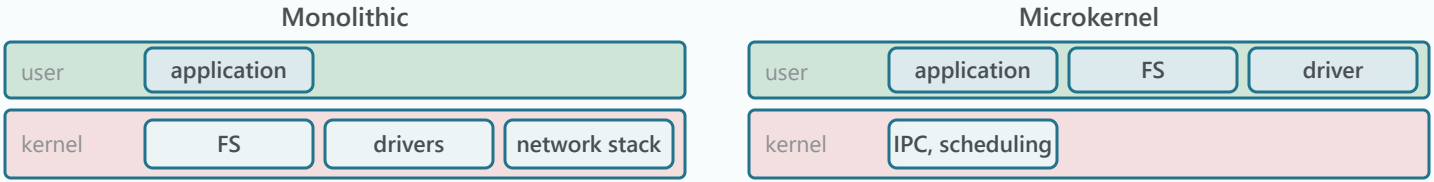
- Everything nonessential is moved out of the kernel; only **IPC (message passing)**, basic scheduling, address-space management and interrupt handling remain.
- Drivers, file systems and the network stack run as **user-mode server processes**; new services are added in user space without changing the kernel.
- Examples: Mach, MINIX 3, QNX Neutrino (automotive), the L4 family, the formally verified **seL4**, and the **Zircon** kernel of Google Fuchsia. macOS, by contrast, uses XNU, a **hybrid** derived from Mach.
- **Mechanism in the kernel, policy outside:** the kernel can run processes by priority; a user-mode component decides how priorities are assigned.

### Pros

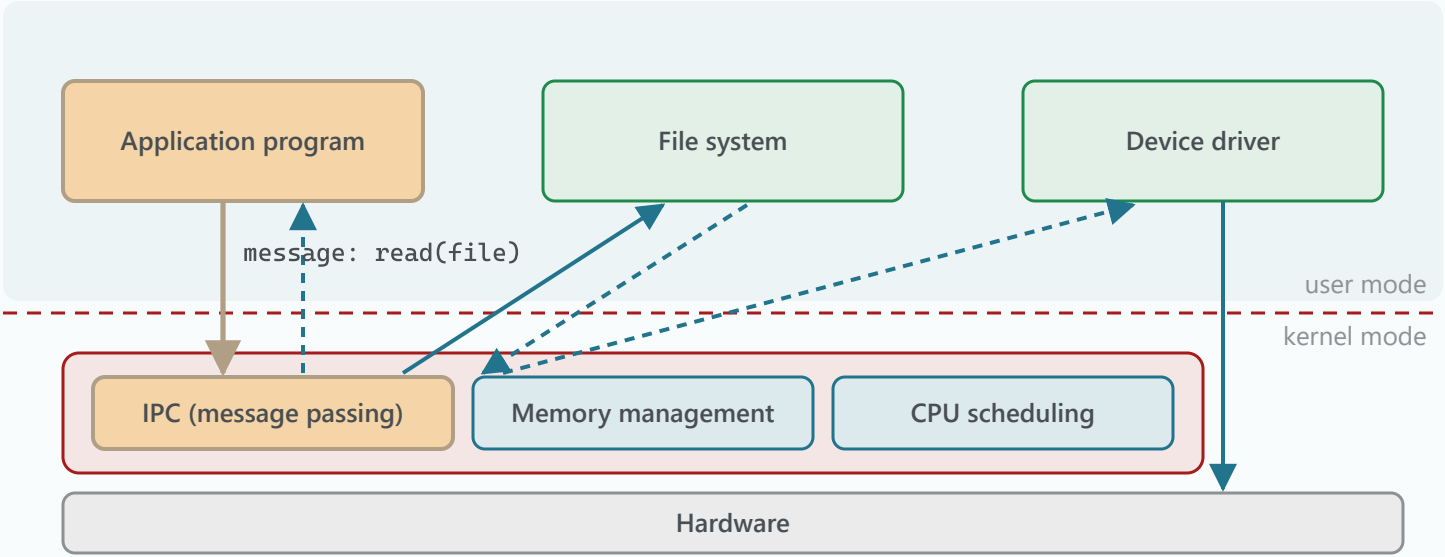
+ A crashed driver does not bring the system down; it can be restarted.
+ Small kernel: easy to test, verify and port.
+ More secure: a small attack surface.

### Cons

− Messages and mode switches add overhead; L4 designs greatly reduced it.

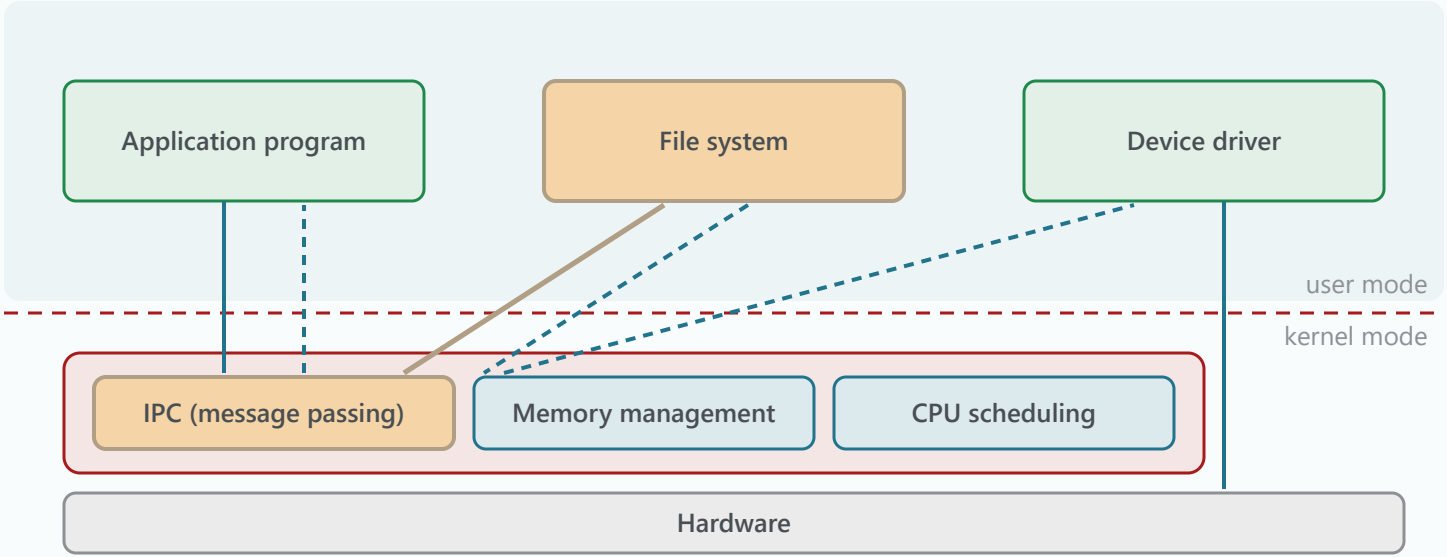


# Reading a File on a Microkernel



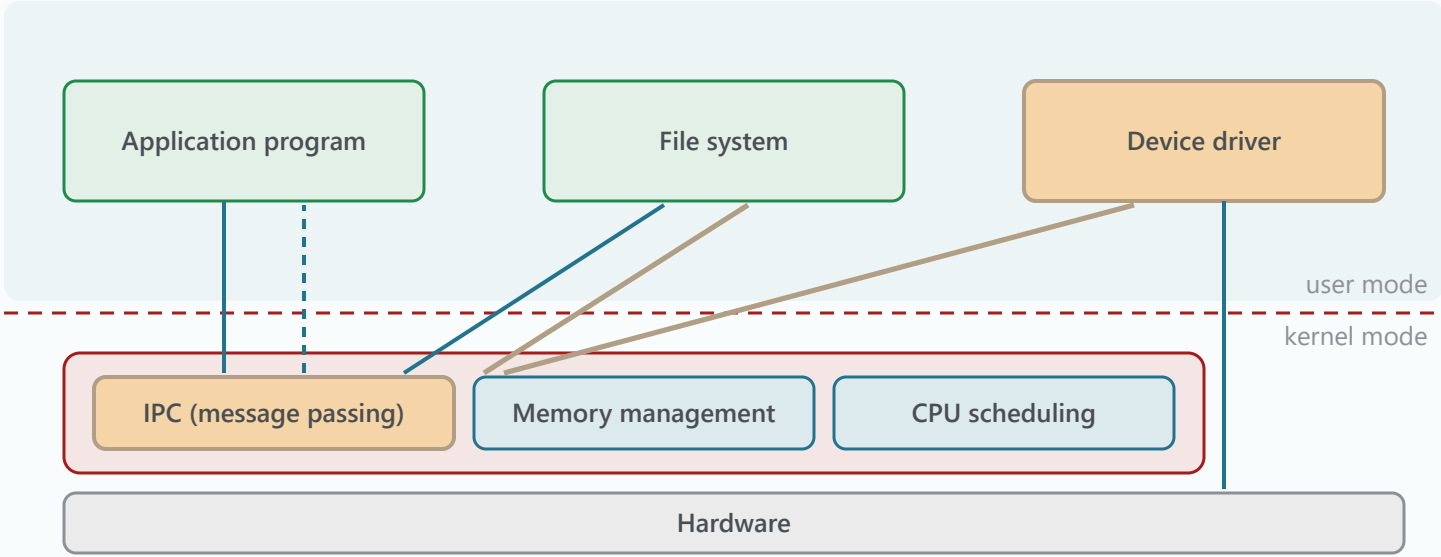

The application sends a **message** to the file-system server; the message passes through the kernel's IPC mechanism.

# Reading a File on a Microkernel



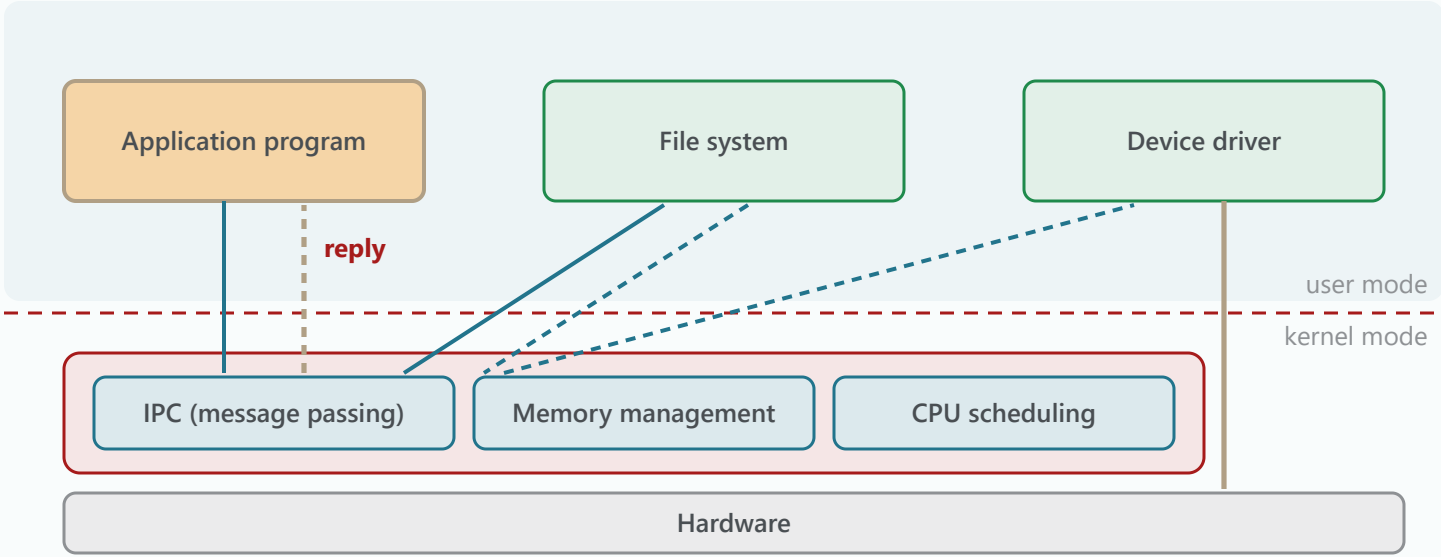

The kernel delivers the message to the file-system server in user mode.

# Reading a File on a Microkernel




The file system works out which block to read and sends a message, again through the kernel, to the disk driver.

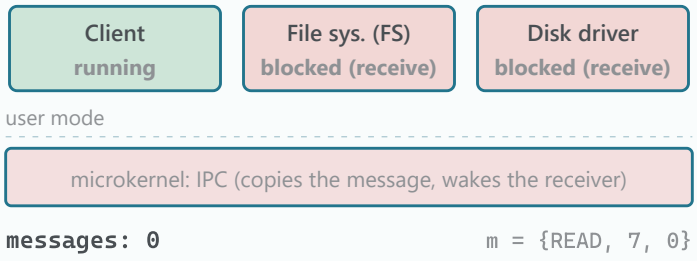
# Reading a File on a Microkernel




The driver reaches the hardware through I/O the kernel permits; replies come back the same way. In a monolithic kernel this is a single chain of function calls.

# Code Trace: `send` / `receive` on a Microkernel

```
1  /* client */
▶ 2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




FS and the disk driver are each a user-mode process; both wait for work in `receive`. The client writes its request into a **message** structure.

Step 1 / 8

**VARIABLES**

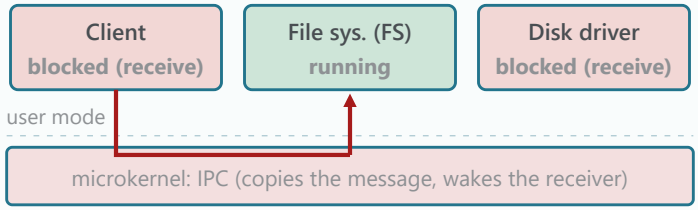
`m.type = READ` `m.file = 7` `m.offset = 0`

**CALL STACK**

`client`

# Code Trace: `send` / `receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
▶  send(FS, &m);
▶  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




`messages: 1` `m → FS`

`send` traps into the kernel: the kernel **copies** the message into FS's buffer and wakes FS. The client blocks in `receive` for the reply.

Step 2 / 8

**VARIABLES**

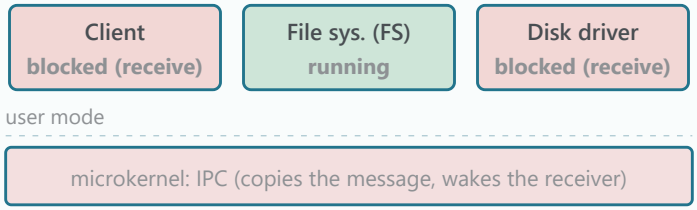
`m.type = READ` `m.file = 7` `m.offset = 0`

**CALL STACK**

**FS**

# Code Trace: `send` / `receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);       /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
▶    receive(ANY, &m);
▶    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




FS's `receive(ANY)` returns; the sender is recorded in `m.source`. FS finds that offset 0 of file 7 is in **block 120**.

Step 3 / 8

VARIABLES

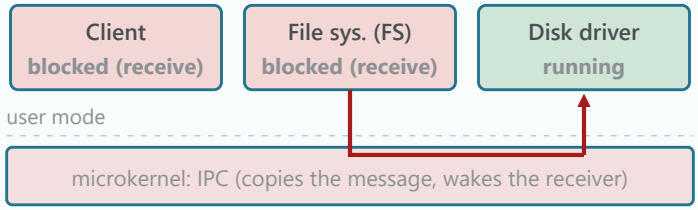
`m.source = client` `block = 120`

CALL STACK

**FS**

# Code Trace: `send`/`receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
▶    send(DISK, &block);
▶    receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




**messages: 2** block 120 → DISK

FS cannot touch the disk directly either: it sends a second message to the driver and waits for the reply.

Step 4 / 8

**VARIABLES**

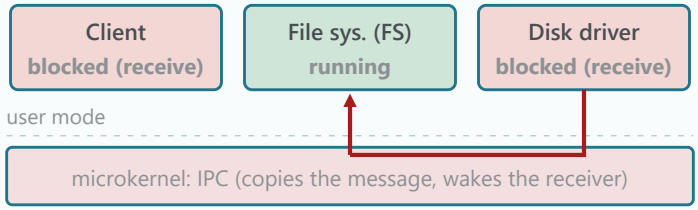
`m.source` = **client** `block` = **120**

**CALL STACK**

**disk driver**

# Code Trace: `send` / `receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
▶    receive(DISK, &data);
▶    m.result = data;
12   send(m.source, &m);
13 }
```




**messages: 3** data → FS

The driver reads the block through I/O the kernel permits and returns the data to FS in a **third** message. FS prepares the reply.

Step 5 / 8

**VARIABLES**

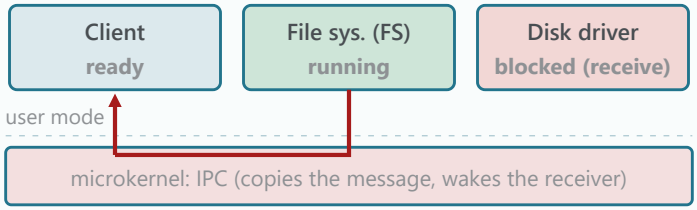
`m.source = client` `block = 120` `m.result = data`

**CALL STACK**

**FS**

# Code Trace: `send`/`receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
▶    send(m.source, &m);
13 }
```




**messages: 4** m → client

The fourth message carries the reply to the client; the client becomes **ready**.

Step 6 / 8

VARIABLES

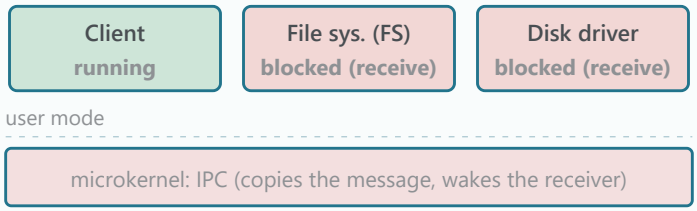
`m.source` = **client** `m.result` = **data**

CALL STACK

**FS**

# Code Trace: `send`/`receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
4  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
▶    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




`messages: 4`

FS goes back to the top of its loop and waits for new work.

Step 7 / 8

**VARIABLES**

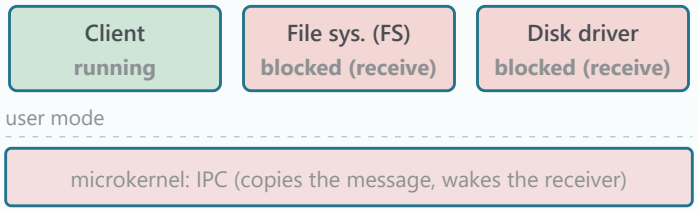
`m.result = data`

**CALL STACK**

**FS**

# Code Trace: `send` / `receive` on a Microkernel

```
1  /* client */
2  m.type = READ; m.file = 7; m.offset = 0;
3  send(FS, &m);
▶  receive(FS, &m);        /* wait for reply */
5  /* file-system server (FS) */
6  while (1) {
7    receive(ANY, &m);
8    block = find_block(m.file, m.offset);
9    send(DISK, &block);
10   receive(DISK, &data);
11   m.result = data;
12   send(m.source, &m);
13 }
```




`messages: 4`

The client's `receive` returns: the data is in `m.result`. One read took **4 messages**, each with a user ↔ kernel switch; a monolithic kernel would finish the same job with function calls inside one system call.

Step 8 / 8

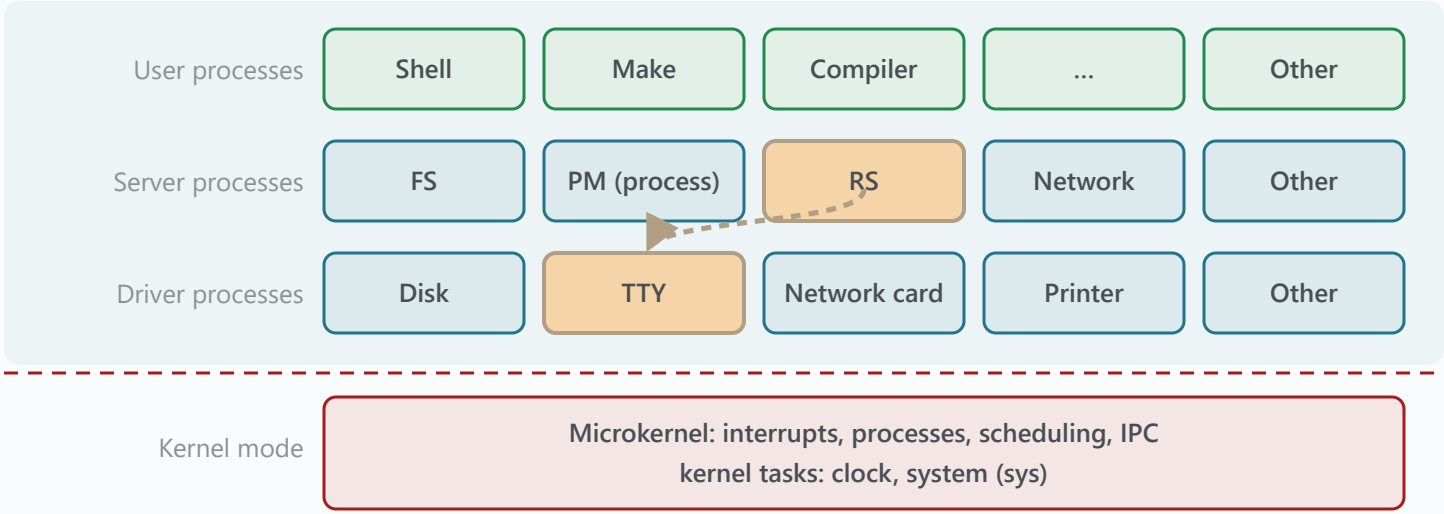
**VARIABLES**

`m.result = data`

**CALL STACK**

`client`

# MINIX 3: A Microkernel Example



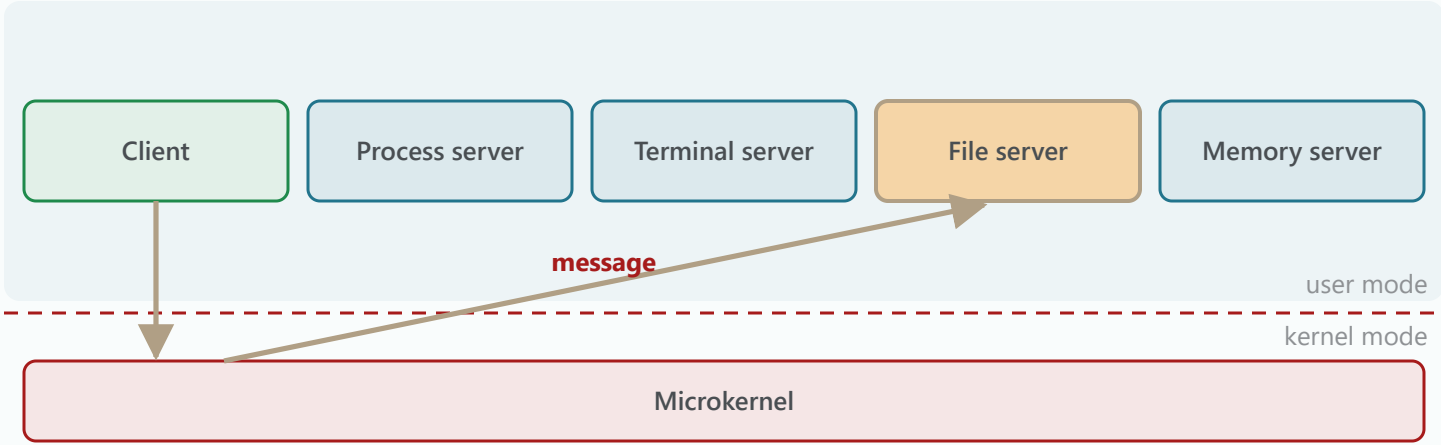

### Reincarnation server (RS)

Monitors the drivers' health and restarts a crashed driver (e.g. TTY) without the user noticing: a *self-healing* system.

### Split privileges

Drivers are in user mode; they reach I/O ports only through kernel calls, to the extent allowed. A faulty driver cannot corrupt memory.

# The Client–Server Model

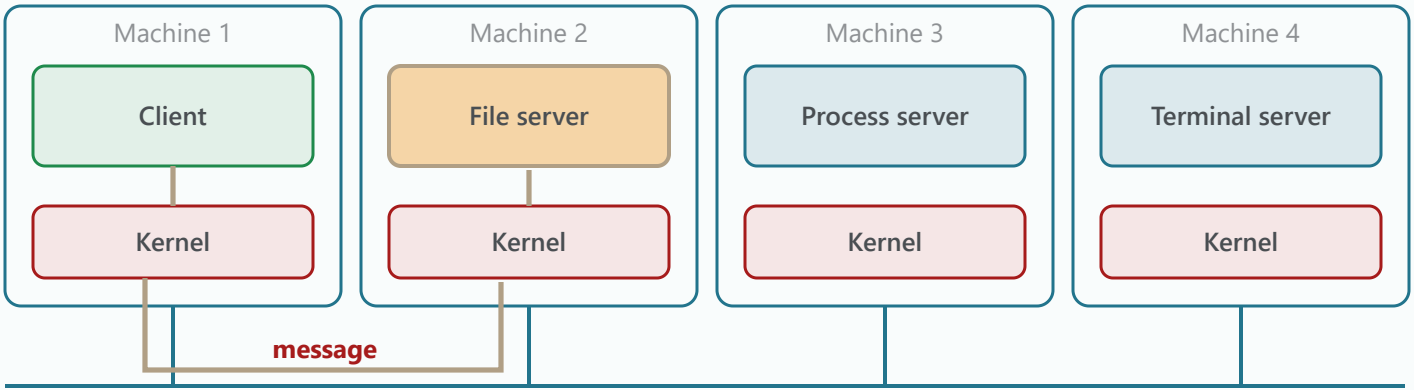



On a microkernel, processes fall into two classes: **clients** that request services and **servers** that provide them. They communicate only by messages.

Step 1 / 2

Message passing is not secure by itself: security depends on the kernel checking which process may send messages to which server.

# The Client–Server Model




A client need not know whether a message is handled locally or on another machine on the network: the model extends naturally to **distributed systems** (today's web and microservice architectures).



Message passing is not secure by itself: security depends on the kernel checking which process may send messages to which server.

# The Tanenbaum–Torvalds Debate (1992)

### Andrew Tanenbaum: microkernel

- In his post “LINUX is obsolete” he called the monolithic design an outdated 1970s approach.
- A microkernel is more modular, flexible and portable; damage from a bug in one component is contained.
- As hardware gets faster, the overhead of message passing becomes negligible.

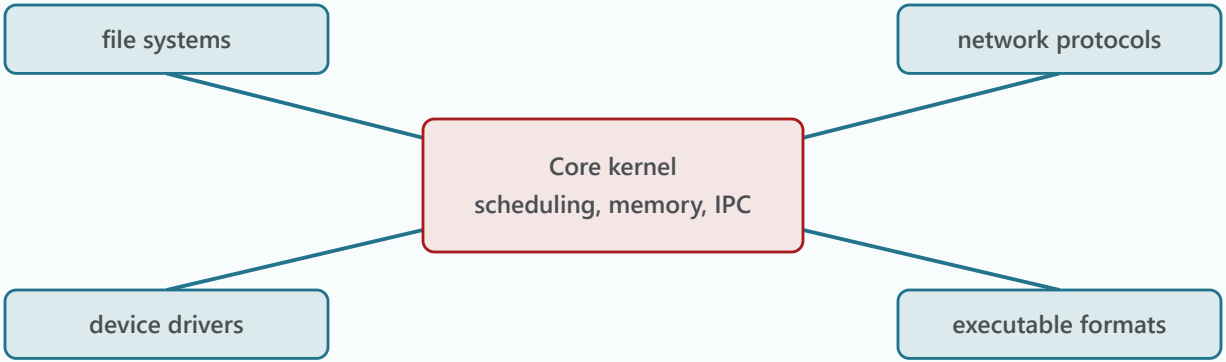
### Linus Torvalds: monolithic

- A monolithic kernel is simpler to design and implement; direct function calls are faster.
- No message passing between processes is needed; in practice, code that *works* is what counts.
- The open-source development model let Linux evolve quickly.

Both sides proved partly right: Linux stayed monolithic but gained flexibility through loadable modules, user-space drivers (FUSE, VFIO) and eBPF; microkernels spread in safety-critical and embedded systems (QNX, seL4).

# Modular Structure: Loadable Kernel Modules

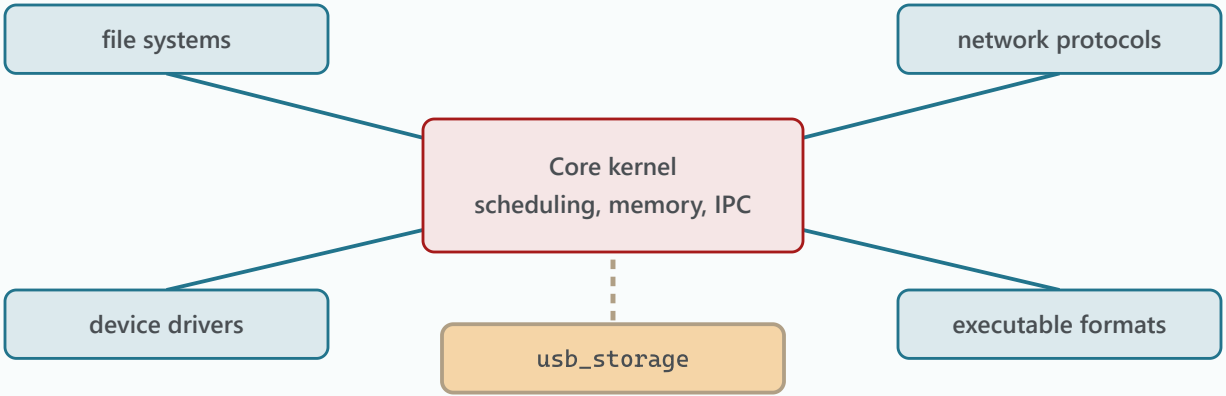
- The kernel holds only core components; other services are loaded at boot or **at run time as modules** (Linux LKM, FreeBSD, Solaris).
- Like layering (defined interfaces) but more flexible: any module can call any other. Like a microkernel, but no messages are needed: modules run **in kernel mode**.




At boot the kernel and the required modules from initramfs are loaded. `lsmod` lists the loaded modules.

# Modular Structure: Loadable Kernel Modules

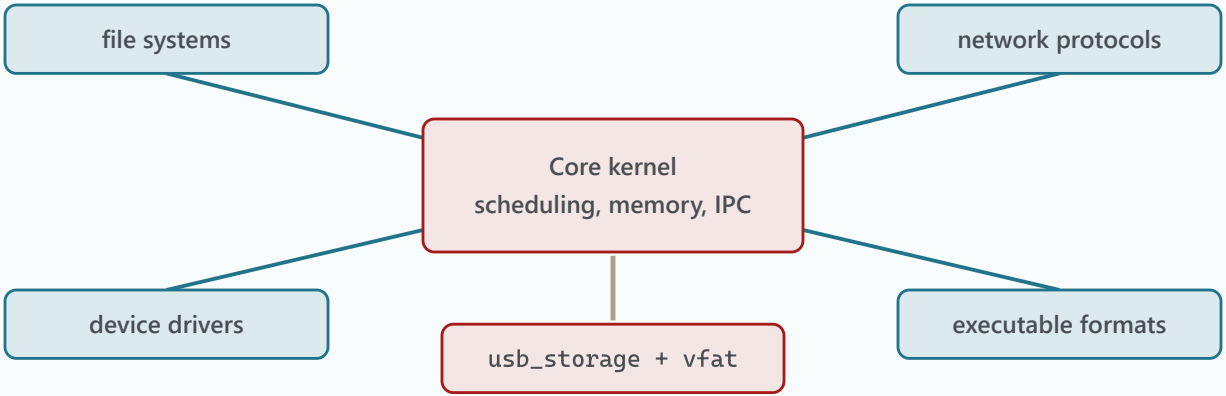
- The kernel holds only core components; other services are loaded at boot or **at run time as modules** (Linux LKM, FreeBSD, Solaris).
- Like layering (defined interfaces) but more flexible: any module can call any other. Like a microkernel, but no messages are needed: modules run **in kernel mode**.




A USB stick is plugged in: a udev event loads the driver (with its dependencies) via `modprobe usb-storage`; the `vfat` module is loaded when the stick is mounted.

# Modular Structure: Loadable Kernel Modules

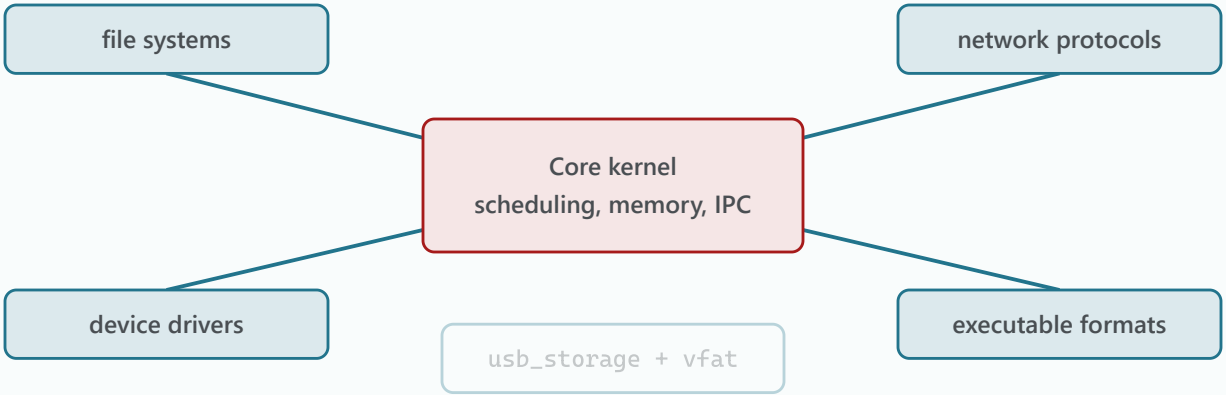
- The kernel holds only core components; other services are loaded at boot or **at run time as modules** (Linux LKM, FreeBSD, Solaris).
- Like layering (defined interfaces) but more flexible: any module can call any other. Like a microkernel, but no messages are needed: modules run **in kernel mode**.




The module registers itself with the kernel and works through direct function calls: fast, but **no isolation**; a faulty module can crash the kernel.

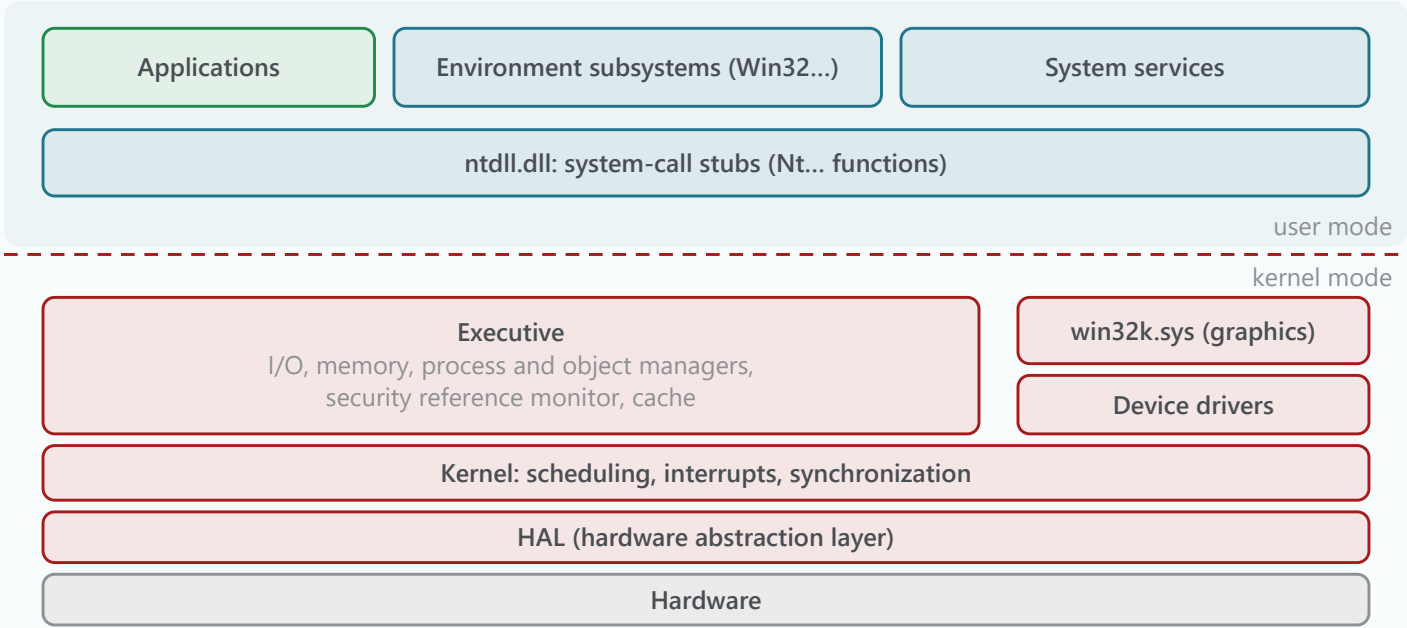
# Modular Structure: Loadable Kernel Modules

- The kernel holds only core components; other services are loaded at boot or **at run time as modules** (Linux LKM, FreeBSD, Solaris).
- Like layering (defined interfaces) but more flexible: any module can call any other. Like a microkernel, but no messages are needed: modules run **in kernel mode**.



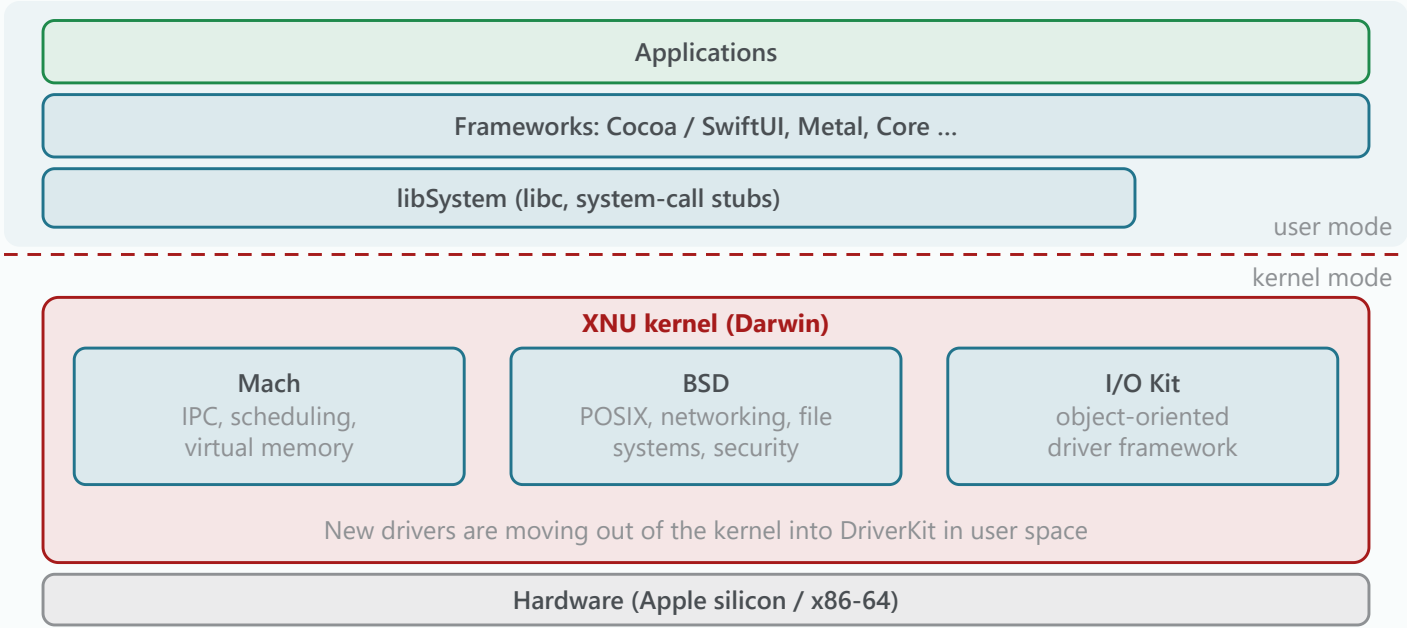

When its use count is 0, `modprobe -r` (or `rmmod`) unloads the module. Modules must match the kernel version.

# Hybrid Kernels: Windows and macOS



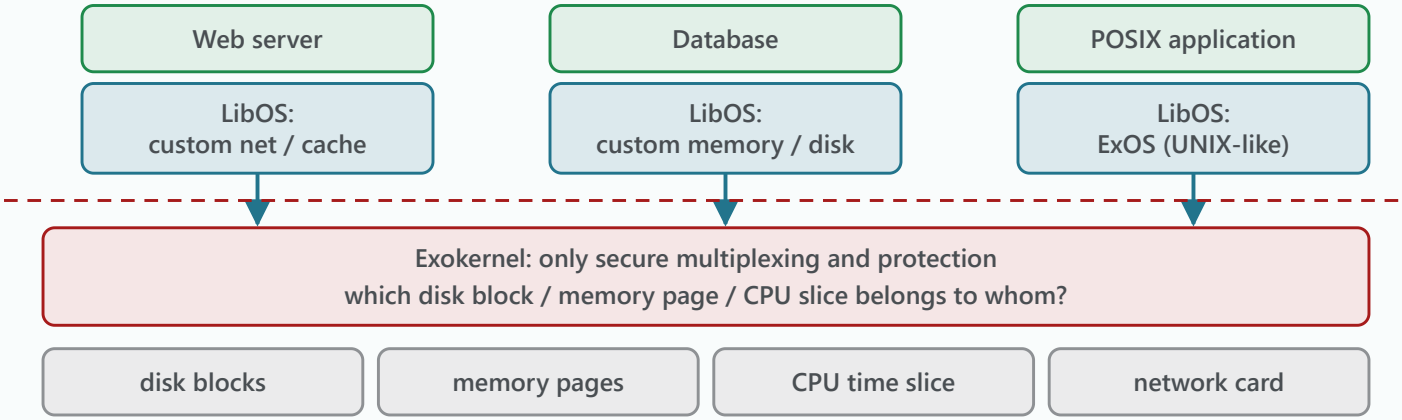

**Windows NT:** microkernel ideas (subsystems, object/message-based design) are kept, but for performance the executive, graphics and drivers were moved into kernel mode.

# Hybrid Kernels: Windows and macOS




**macOS / iOS XNU:** the Mach microkernel and the BSD layer share one kernel address space: microkernel modularity + monolithic speed.

# Exokernels




- A research architecture by Engler and Kaashoek at MIT (1995): the kernel **Aegis** and the library OS **ExOS**. Not a commercial OS.
- A VM gets a **copy** of the machine; an exokernel hands out a **share** of a resource and checks ownership, with no remapping layer needed.
- **Separates management from protection:** the kernel only tracks who owns which resources; abstractions (files, virtual memory) come from application-level **LibOSes**.
- High performance through application-specific memory management and caching, but applications are complex to build. The idea lives on in **unikernels** and kernel-bypass I/O such as DPDK / SPDK.

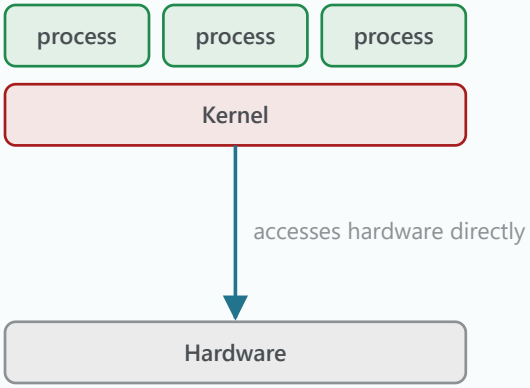
# Comparing the Structures

| Structure | What is in the kernel? | Strength | Weakness | Example |
|---|---|---|---|---|
| Simple | everything, unrestricted | small, simple | no protection | MS-DOS |
| Monolithic | all services | speed | no fault isolation | UNIX, Linux |
| Layered | hierarchical layers | easy to verify | overhead, hard to define | THE, MULTICS |
| Microkernel | IPC, scheduling, memory | reliability | message overhead | MINIX 3, QNX, seL4 |
| Modular | kernel + modules | flexibility, speed | module = full privilege | Linux, FreeBSD |
| Hybrid | micro + selected services | balance | complexity | Windows, macOS |
| Exokernel | protection only | app-specific speed | hard to develop | Aegis/ExOS |

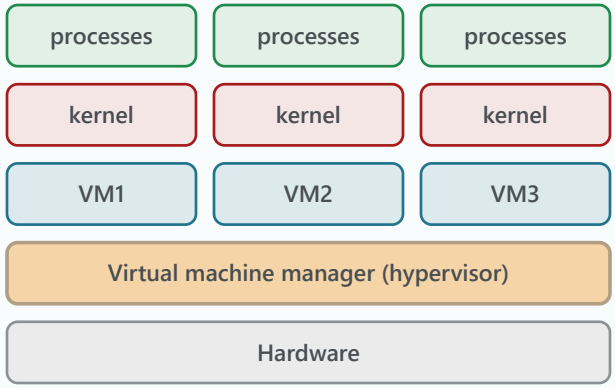
Real systems are not pure: Linux is monolithic + modular, Windows and macOS are hybrid, and nearly all are layered to some degree.

# The Virtual-Machine Concept

- A hypervisor (VMM) creates virtual machines on one physical machine, each **looking like an independent computer**.
- Each guest OS believes it has its own CPU, RAM, disk and I/O devices; different OSes can run concurrently on the same hardware.
- The idea goes back to IBM VM/370 (1972); today it underpins cloud computing: server consolidation, isolation, snapshots, live migration.
- A hypervisor runs on the hardware (**type 1**) or on a host OS (**type 2**). A **container** carries no kernel of its own; it isolates processes that share one kernel.

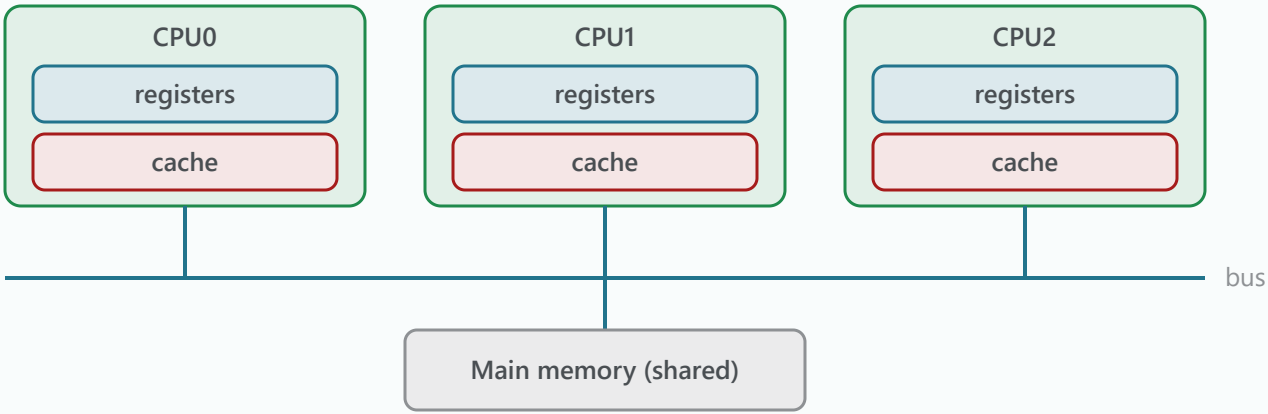



(a) Nonvirtual machine



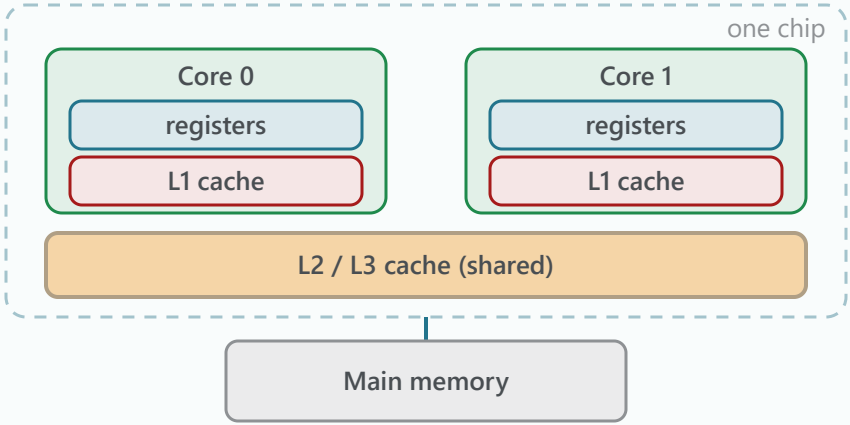

(b) Virtual machines

# Symmetric Multiprocessing and Multicore Architecture



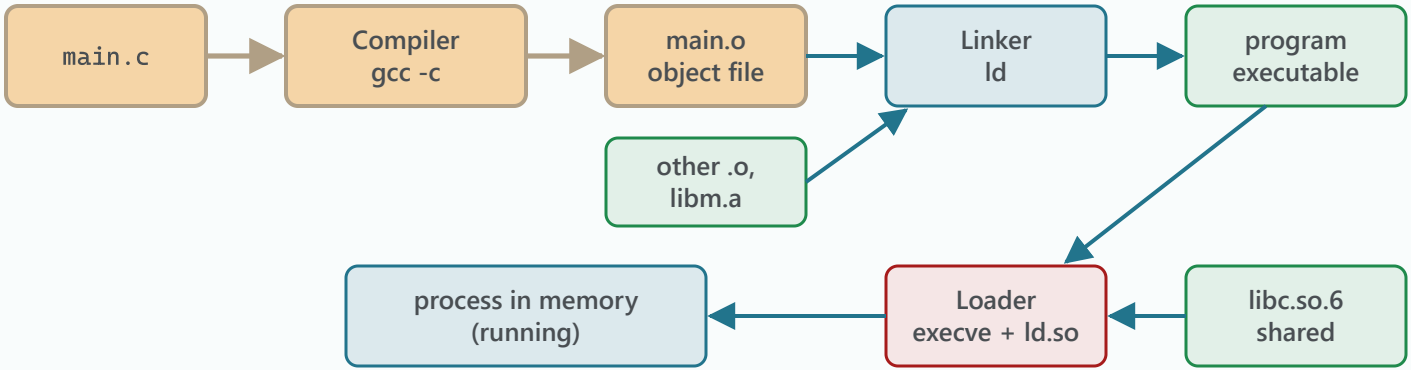

**SMP:** each CPU has its own registers and cache and shares memory; every CPU can do any work, the OS included. Kernel data structures must be protected by locks (Linux's “big kernel lock” was removed entirely in 2011).

# Symmetric Multiprocessing and Multicore Architecture



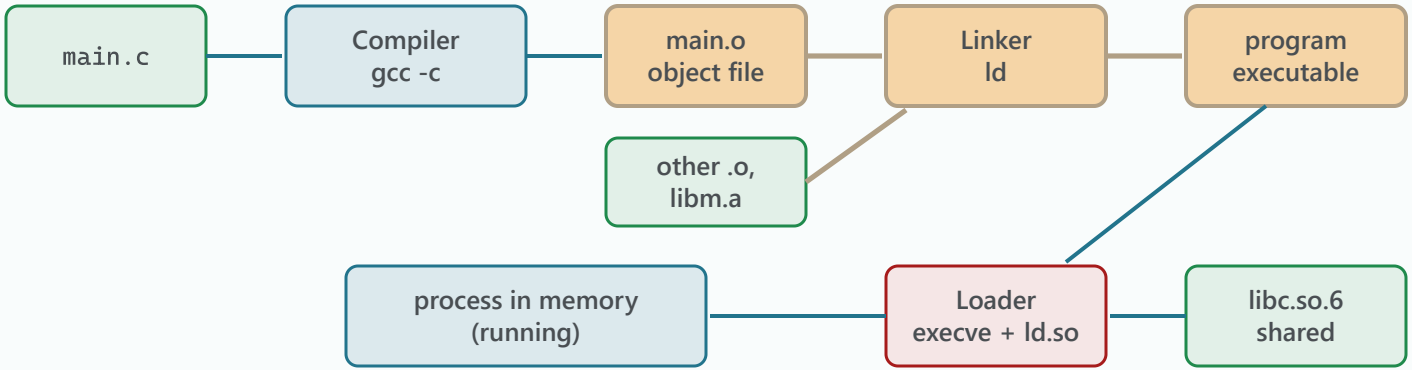

**Multicore chip:** cores on the same chip share nearby caches and communicate faster with less energy. P/E (performance/efficiency) cores, common in 2026, give the scheduler new work.

# Compiling, Linking and Loading



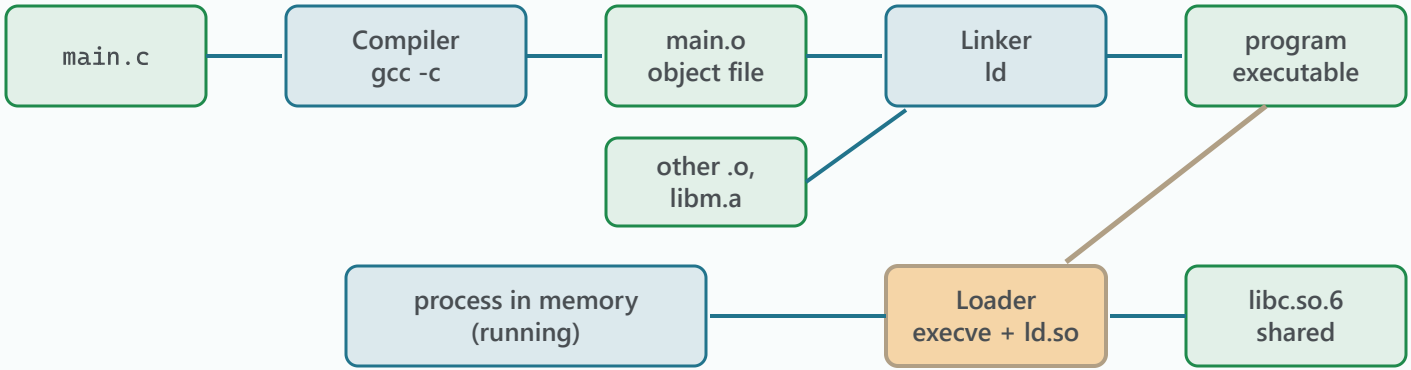

The compiler turns source code into a **relocatable object file**: addresses are not final yet and external symbols (e.g. `printf`) are unresolved.

# Compiling, Linking and Loading



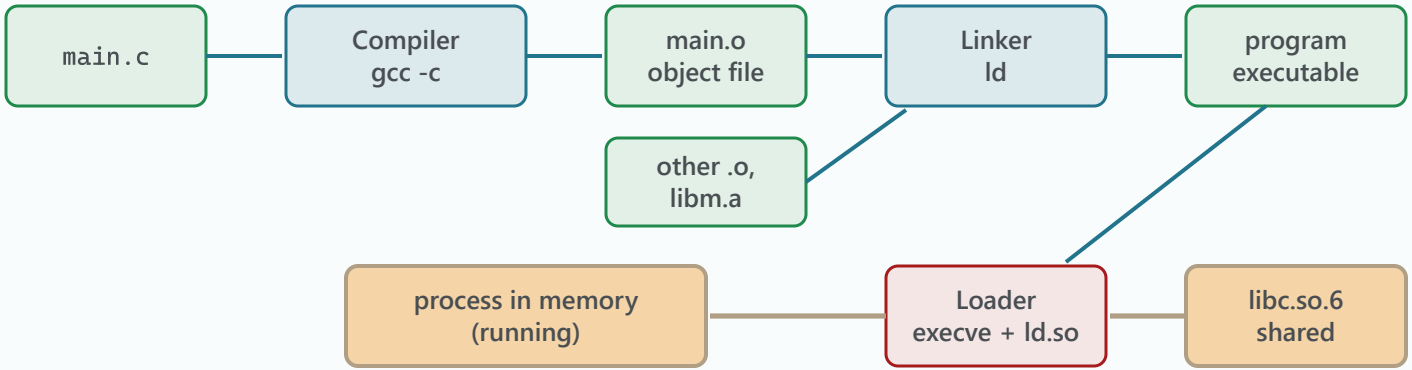

The **linker** combines object files and static libraries, resolves symbols, fixes up addresses and produces an executable.

# Compiling, Linking and Loading




The shell calls `execve`; the kernel's **loader** reads the file header, maps the segments into the address space and starts the dynamic linker (`ld-linux.so`).

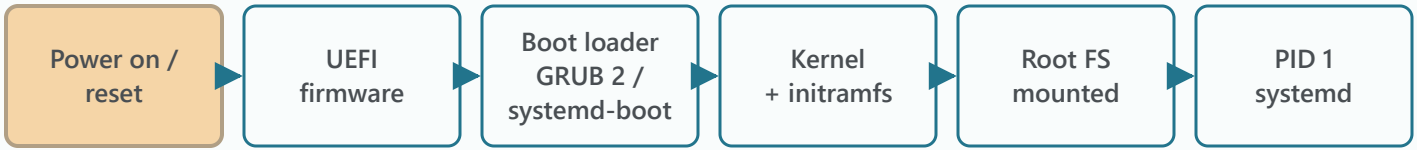
# Compiling, Linking and Loading




Shared libraries ( `libc.so.6` ) are linked at run time and shared among processes. Formats are OS-specific: ELF (Linux), PE (Windows), Mach-O (macOS); so programs cannot move directly between OSes.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.

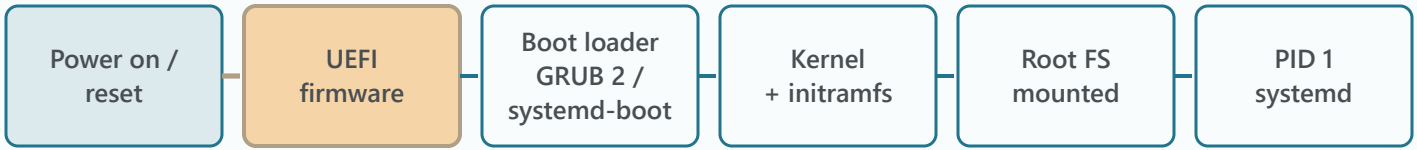



The CPU starts running firmware at a fixed address.

At power-on the CPU resets and starts executing firmware in flash from a **fixed address** set by the hardware.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.

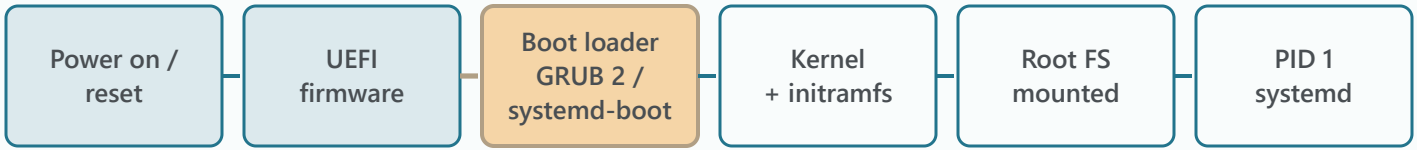



POST tests and initializes the hardware; the boot loader on the EFI system partition is found.
Secure Boot verifies the signature of every stage.

UEFI tests memory and devices with the power-on self-test (**POST**), then, following the order in NVRAM (NVMe, USB, network/PXE), loads the boot loader from the EFI system partition (FAT32) on a GPT disk. With Secure Boot on, its signature is verified.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.



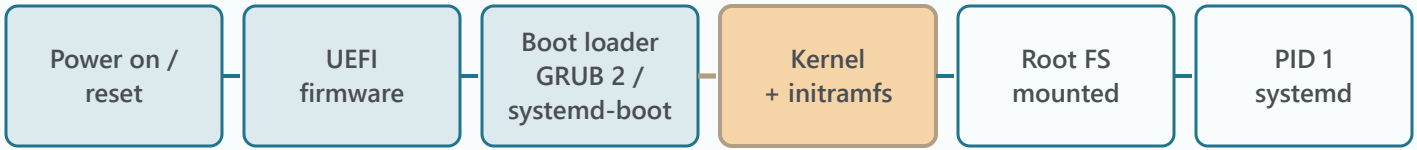

The boot loader offers a choice of kernels and loads the kernel and initramfs into memory.

Older systems: BIOS + MBR.

The boot loader shows a menu, puts the chosen kernel and the initial RAM disk (initramfs) in memory and jumps to the kernel.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.

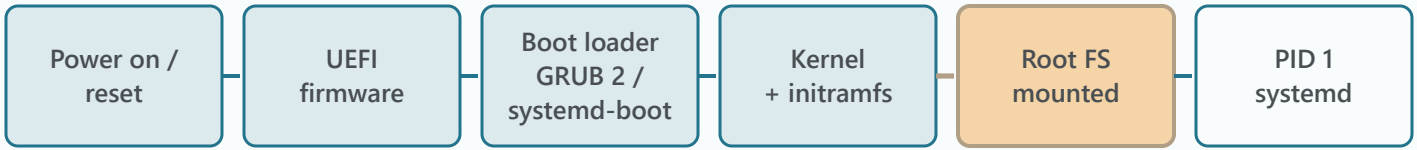



The kernel decompresses itself, detects hardware, loads drivers; initramfs provides a temporary root file system.

The kernel starts memory management, the scheduler and drivers; scripts in initramfs load the modules needed.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.

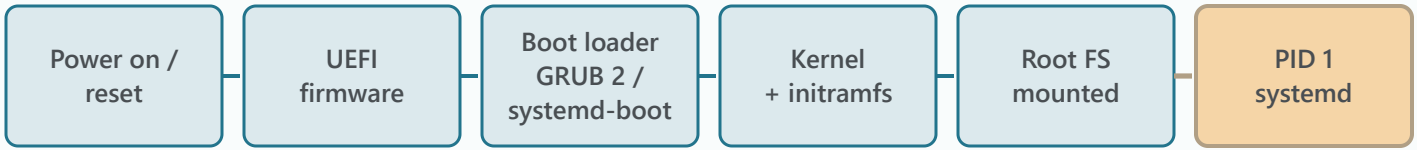



The real root file system is mounted (e.g. after unlocking an encrypted disk).

The real root file system is mounted and initramfs is discarded.

# Booting the Operating System

Booting is everything from loading the kernel into memory to starting the first user process. The old BIOS (1981, 16-bit real mode) ran the MBR code in the disk's first sector; UEFI loads a file ( `.efi` ), not a sector. On Linux `systemd-analyze` shows the time of each stage.




The first user process starts; it launches services in parallel and opens the login screen.

The kernel starts PID 1 (systemd), which brings up services, the network and the login screen. From then on every process descends from PID 1. On Windows: winload.efi → ntoskrnl.exe → smss.exe → winlogon.exe.

SUMMARY

# Review Questions

1. Why can't a user-mode program execute I/O instructions directly? What happens if it tries?
2. What is the difference between `printf` and `write`? Which one is a system call?
3. Why is `fork()` "called once, returns twice"? What does it return in the parent and in the child?
4. What happens if a driver crashes on a microkernel? On a monolithic kernel?
5. Do loadable modules make Linux a microkernel? Why?
6. What is the first user process the kernel starts at boot? Why does it matter?

▶ **Show answers**

# References


- Silberschatz, Galvin and Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 (Chapters 2 and 18).
- Tanenbaum and Bos, *Modern Operating Systems*, 4th ed., Pearson, 2014 (Sections 1.5–1.7 and Chapter 7).
- Arpaci-Dusseau and Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), version 1.10, pages.cs.wisc.edu/~remzi/OSTEP.
- Linux man pages: `syscalls(2)`, `syscall(2)`, `fork(2)`, `pipe(7)`, `mount(8)`.
- Engler, Kaashoek and O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management", SOSP 1995.

# The End

## Chapter 2: Operating-System Structures

Previous: Chapter 1: Introduction · Next: Chapter 3: Processes