

C, string.h

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

Announcements

- Homework 10 available, due next Monday
- Lab 12 tomorrow
- Final exam: 7pm Dec 12, Physics 338 (different room!)
 - Cumulative, see practice tests
 - Exam conflict form in email

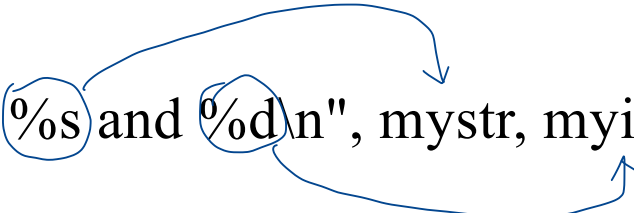
printf

Could you write printf?

printf

```
int printf(const char *format, ...);
```

```
printf("hi: (%s) and (%d)\n", mystr, myint);
```



You'd scan the format string one character at a time.

When you see %, you parse the specifier.

That tells you the type of the next argument.

You fetch the argument and convert it (e.g., convert int → its decimal characters).

You output everything to some destination.

“Where does the output actually go?”

printf ultimately sends characters to `stdout`, which is a file-like object.

To do that, `printf` must eventually call **the system call** `write`.

All the processes running on the machine : `ps -A | less`

How many things are running ? `ps -A | wc -l` (All of them have a view of)

→ memory as if they were the only program running in memory.

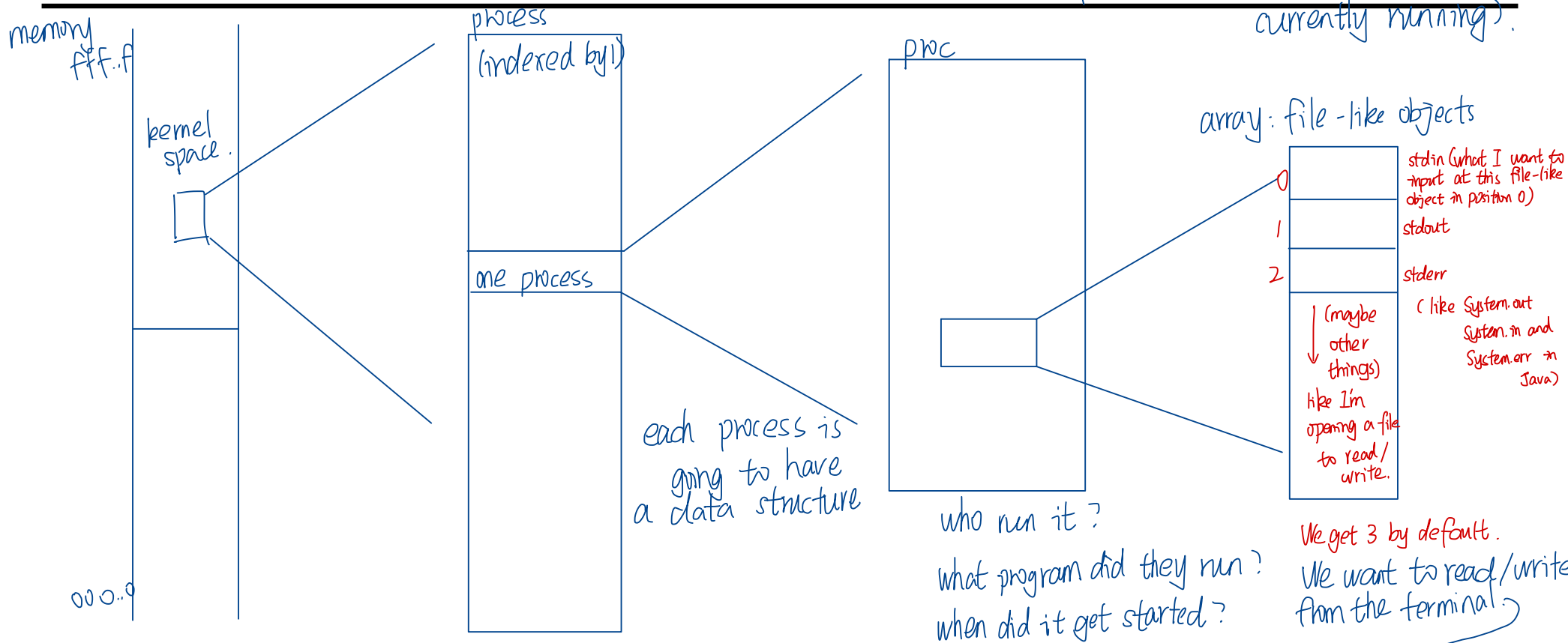
Processes

Process - approximately what we think of as a “running program”

- Operating System effectively has a giant array of processes started since computer turned on
- Try `ps -A`
- Has access to all memory (but only its own!)
- Operating System maintains data structure about each process
 - What program is running, who ran it, when it started, ...
 - Array of “file like objects”

Processes

(a big array of processes. I can see information and stores information about all the processes are currently running).



↳ (The operating system is handling all of that)

stdio.h manual page

1. What is `<stdio.h>`?

`<stdio.h>` is the **standard buffered input/output library** in C.

1. **Standard** (commonly accepted, default, expected way to do IO)
2. **Buffered** (IO is not instantaneous; it is collected in buffers)
3. **Input/Output** (reading from user / sending characters to the user)

2. Why is IO “weird”? The role of buffering

IO feels strange, especially in assignments, because **it is buffered**.

Why buffering exists

When you type something:

- OS must detect the keystroke
- translate the keyboard code to a character based on locale
- figure out which program should receive the input
- deliver that input to the program
- the program must ask the OS for it

This is expensive.

So instead of handling characters **one by one**, `<stdio.h>`:

- reads **a chunk** at once (maybe hundreds or thousands of bytes)
- stores it in a **buffer**
- future reads simply take characters from this buffer

This makes IO faster and more efficient.

Why buffering causes weird behavior

- Input may not appear until *Enter* is pressed
- Sometimes input “lags” or seems not to arrive
- Reads may get an entire line at once
- Output may not appear until the buffer flushes

This is normal.

3. Why is `FILE` hidden? (Encapsulation in C Standard Library)

`<stdio.h>` defines a type:

```
typedef struct _IO_FILE FILE;
```

...but the **contents of the structure are not shown**.

- This lets library authors change how `FILE` works internally **without breaking your code**
- This is similar to **Java encapsulation**: private fields + public methods
- As a programmer, **you are not supposed to know the internals**
- You only receive a **pointer to `FILE`** and use library functions to operate on it

If you think you need to know what's inside `FILE`, you're probably doing something wrong.

4. What's actually inside `<stdio.h>`?

4.1 Standard streams

These macros are provided automatically:

```
stdin
stdout
stderr
```

- `stdin` – default input (keyboard)
- `stdout` – default output (prints to console)
- `stderr` – error output (unbuffered or line-buffered)

Many functions operate on these by default if no `FILE*` is provided.

4.2 The large set of IO functions

Most start with `f` and take a `FILE*`:

- `fopen`
- `fclose`
- `fread`
- `fwrite`
- `fprintf`
- `fscanf`
- `fseek`
- `ftell`
- `feof, ferror`
- etc.

Whenever you see a function that takes a `FILE*`, it means:

“Which file / which stream do you want to operate on?”

Functions that do NOT start with `f`

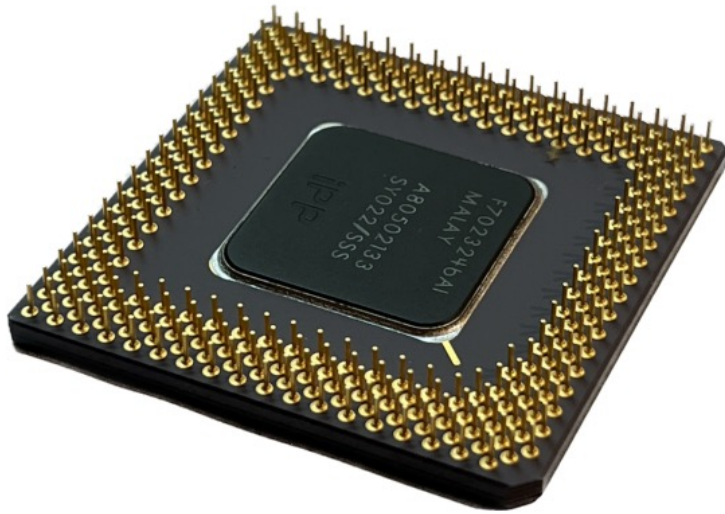
Examples:

- `printf` → prints to **`stdout`**
- `putchar` → sends one character to **`stdout`**
- `getchar` → reads one character from **`stdin`**

If there is **no `FILE*` parameter**, it is using a **default stream**.

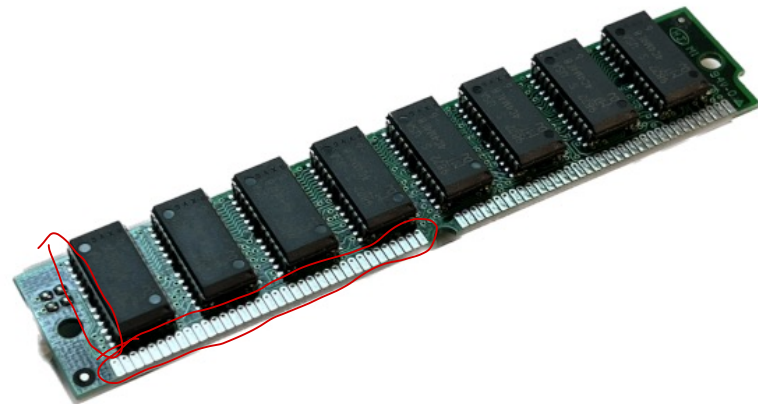
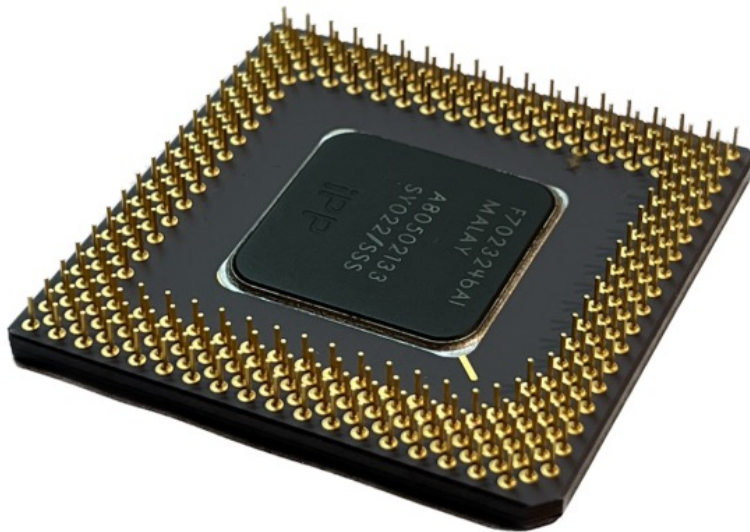
5. Why the library is designed this way

- The C Standard Library provides a **simple, expected, portable** way to do IO.
- Encapsulation lets the library evolve without breaking your code.
- IO is buffered because interacting with the OS character-by-character is extremely expensive.



CPU. (20 years old) .

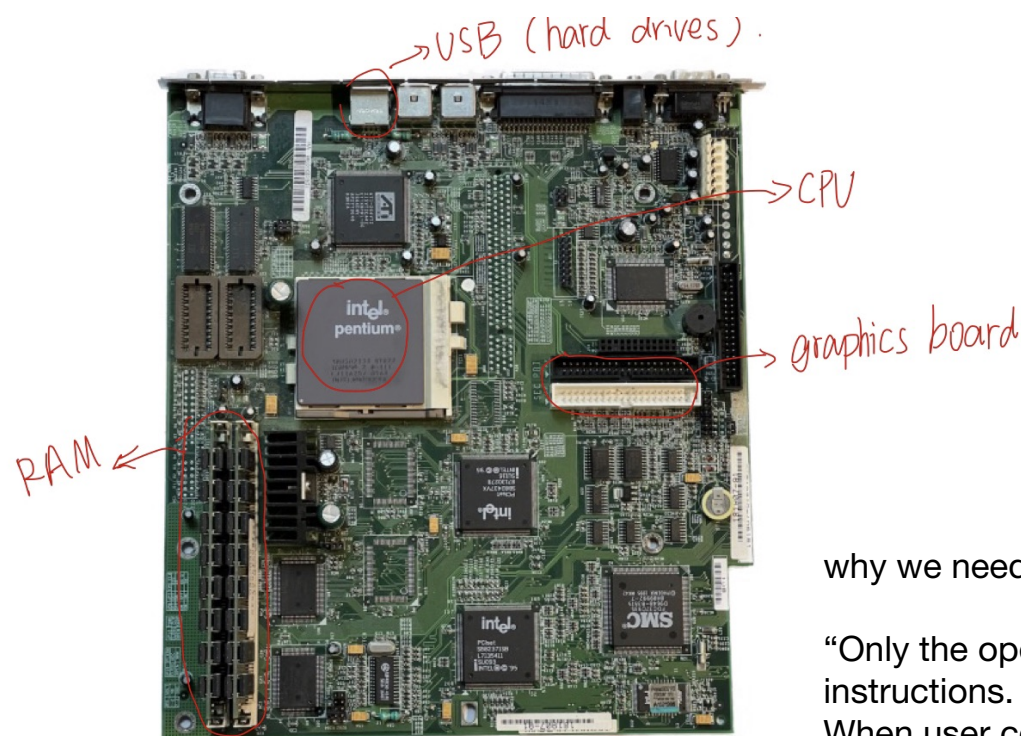
Those pins exist because the CPU must communicate with the outside world. Some pins carry power, but most are used to talk to other parts of the computer.



RAM. Notice it also has rows of pins.

Some of the CPU's pins are specifically for communicating with memory. This is how the CPU reads instructions, loads data, and stores results.

But remember, not all pins are for memory — the CPU still needs to talk to many other things.”



Here is a full motherboard.

You can see the CPU, the RAM slots, and connectors for hard drives, USB, network cards, graphics, and more.

The CPU must communicate with all of these components — not just memory.

And here's the key point:

We have never discussed instructions that let us talk to these devices.

Those instructions exist, but they're extremely dangerous.

A user-level program is not allowed to execute them — the hardware blocks it — because a tiny bug could accidentally format your hard drive or send invalid commands to a device.

why we need system calls:

“Only the operating system (the kernel) is allowed to use those instructions.

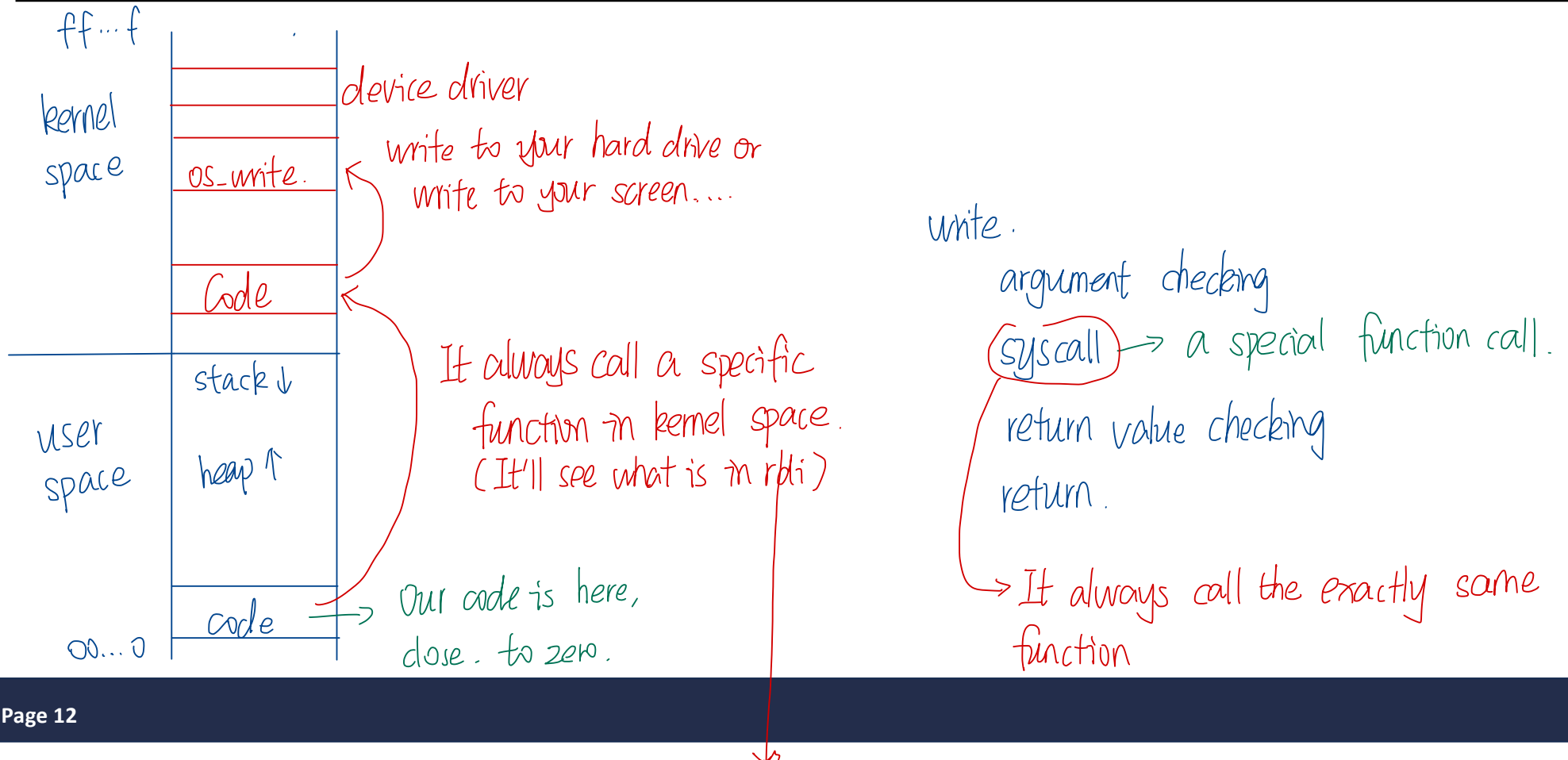
When user code needs to talk to a device (like printing to the terminal), it must call into the kernel through a system call, like `write()`.”

kernel is allowed to call these special assembly instructions that do things like write to the hard drive, write to the terminal, open/close the files, that kind of thing.

Syscalls

How do I get the kernel to do things?

how do we, as user programs, get the kernel to do things for us?



But we are not allowed to read/write in the kernel space. Syscall is allowed to do this. When I run this instruction, it becomes the operation system.

System Call Mechanism Explanation

The way we ask the kernel to do something is through a system call. And the key system call for output — the one that `printf` ultimately uses — is called `write`.

A system call is implemented as a normal function in assembly, but with one important instruction inside:

`syscall`

That instruction performs a very special kind of control transfer. It's like calling a function, but instead of calling another function in user space, it jumps into kernel space, which we normally cannot access.

When the CPU executes `syscall`, it jumps to a fixed entry point in the kernel.

The kernel then looks at the first argument, which is stored in `%rdi` — because that's the calling convention.

`%rdi` contains a number that tells the kernel which service you want.

For example, the number for `write` means:

"Dear kernel, please write these bytes to this file descriptor."

So the kernel checks that number, looks it up in a giant table of allowed operations, and then calls the correct internal function that can actually talk to the hardware.

So the system call mechanism is the one legal doorway into kernel space.

It's the only way user code can trigger the privileged instructions that control devices.

1. Why device drivers exist

Different devices behave differently.

A hard drive from Seagate, a hard drive from Samsung, a USB mouse, a graphics card — each one needs its own very specific sequence of bits to operate it.

That means the kernel can't hard-code special logic for every possible device in the world.

2. What device drivers actually do

Device drivers are pieces of code inside the kernel that know exactly what sequence of bits a particular device needs.

They act as the translation layer between the operating system and the hardware.

In other words:

OS → abstract request

Driver → concrete hardware-specific commands

3. Why drivers are risky

Device drivers live in kernel space. That means they run with full privileges.

If the driver has a bug, it can crash your entire system.

It's not like a user program that just segfaults — the whole OS can go down.

4. Why drivers cause most system crashes

There are only a few major operating systems, and many people contribute to them and check their code.

But there are thousands upon thousands of hardware devices in the world, each needing its own driver.

So statistically, most crashes come from buggy device drivers — not the core OS.

5. Two philosophies: Windows vs. macOS

Windows — open hardware ecosystem

Windows allows anyone to write and install device drivers.

Pros:

Users can plug in almost any hardware device.

Cons:

Many drivers come from small vendors.

Lower quality control → more system crashes.

This is why “blue screens” often trace back to driver problems.

macOS — controlled hardware ecosystem

Apple takes the opposite approach.

They strictly control which hardware is allowed, and they control or verify all the drivers.

Pros:

Far fewer driver-related crashes.

Very stable OS behavior.

Cons:

Users cannot freely choose arbitrary hardware.