



I/O and peripherals

Advanced Operating Systems

Vittorio Zaccaria | Politecnico di Milano | '25/26

Scan below QR code for course website



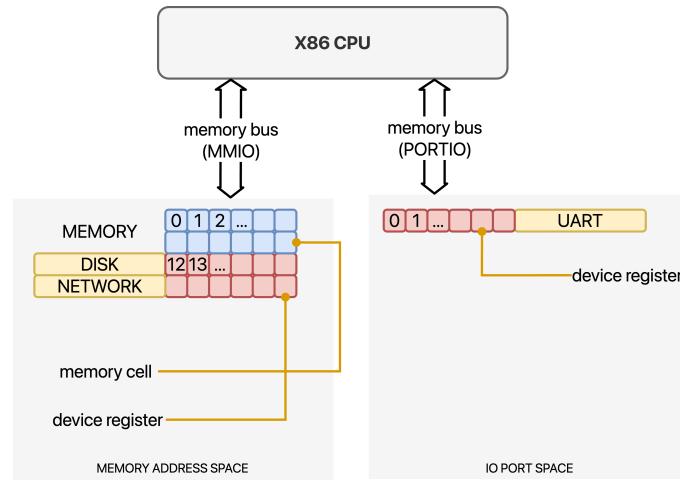
System architecture and IO

Introduction

A **bus** is a connection channel between the CPU and the peripherals.

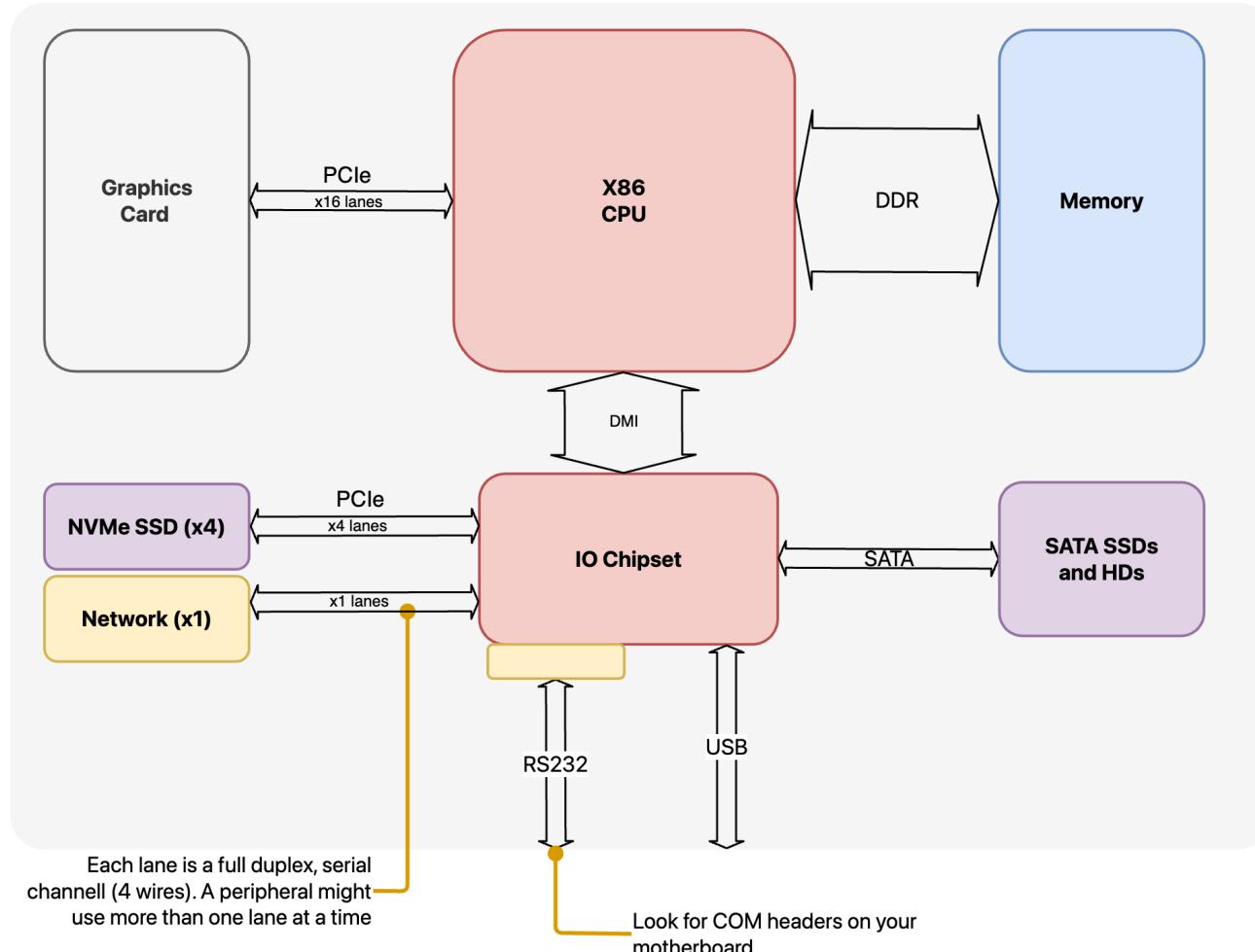
Communication between the CPU and devices can happen over two buses, the **memory bus** and the **port bus**.

- **Memory bus:** Communication happens through loads and stores; here device registers to send/receive data are exposed as memory addresses on the address space of the CPU. This is also called **Memory-mapped IO**.
- **Port bus:** Uses special instructions and special addressing modes (*port indices*). This is called **Port-based IO**.



Introduction

Modern system device architecture (intel Z270 motherboard)



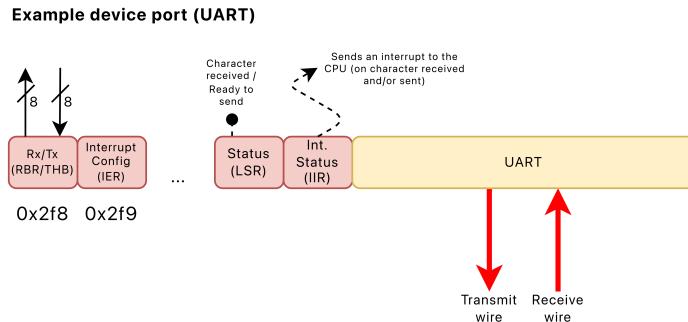
CPU to device communication

Port I/O Instructions

- Port bus I/O operations are done through **special I/O instructions**.
- Some devices are assigned a `port` number in the I/O address space which names the device and must be used with the instructions
- For example on `intel` you use the `in` and `out` (**privileged**)

Port I/O Instructions » Example legacy UART

Example: legacy x86 systems have a universal asynchronous receiver-transmitter (UART) whose register indices start on the port bus at 0x2f8.



To read a character received through the receive wire, you would use the following assembly code.

```
in RBR, %al    ;; RBR = 0x2f8
```

Port I/O Instructions » Available ports

```
% cat /proc/ioports
0000-0cf7 : PCI Bus 0000:00
 0000-001f : dma1
 0020-0021 : pic1
 0040-0043 : timer0
 0050-0053 : timer1
 0060-0060 : keyboard
 0064-0064 : keyboard
 0070-0071 : rtc0
 0080-008f : dma page reg
 00a0-00a1 : pic2
 00c0-00df : dma2
...
03f8-03ff : serial
```

Memory-mapped IO

Gained importance as `in / out` instructions are inefficient and limited in their use of registers. Additionally, there is a restriction of only 2^{16} port numbers.

In fact, devices registers can be operated on with loads/stores to specific physical addresses (also called memory mapped registers).

Here we are writing `0x80` into a device register positioned at address `0xc0100` :

```
volatile int32_t *device_control
    = (int32_t *) (0xc0100);
*device_control = 0x80;
```

Memory-mapped IO » Available mapped devices

Example, the ethernet device e1000

```
% cat /proc/iomem
...
08000000-feffff : PCI Bus 0000:00
    ...
    feb80000-feb9ffff : 0000:00:03.0
    feb80000-feb9ffff : e1000
    ...
...
```

Device to CPU communication

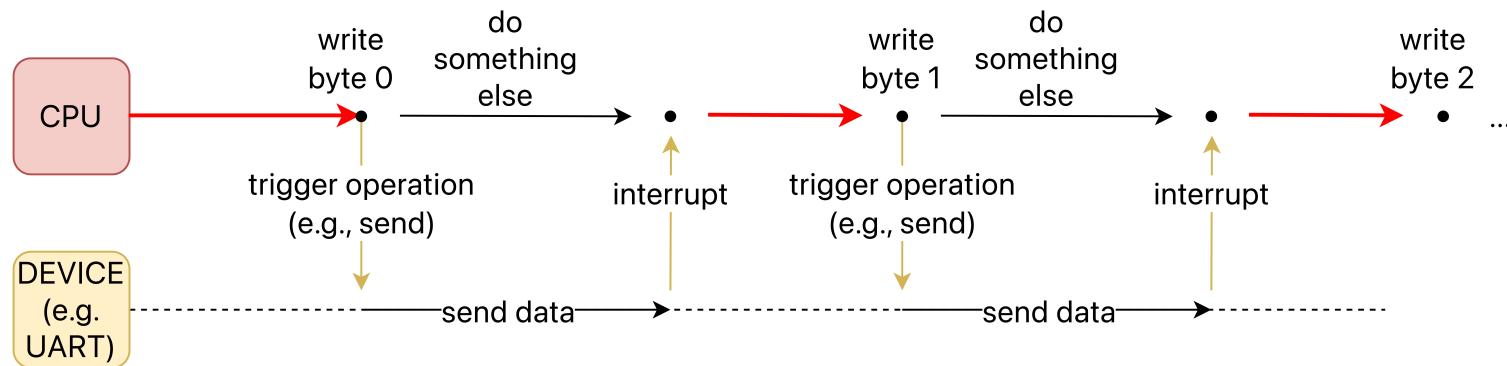
Polling

- Polling is a method of interaction with devices where the processor waits until the device requires attention.
- This can result in wasted processor cycles if the device is slow, but it is inexpensive if the device is fast.

```
// Here we are using in/out instructions
// to wait until the UART is ready to send;
while((inb(LSR) & RDY)==0) ;
outb(c, THR); // send character
```

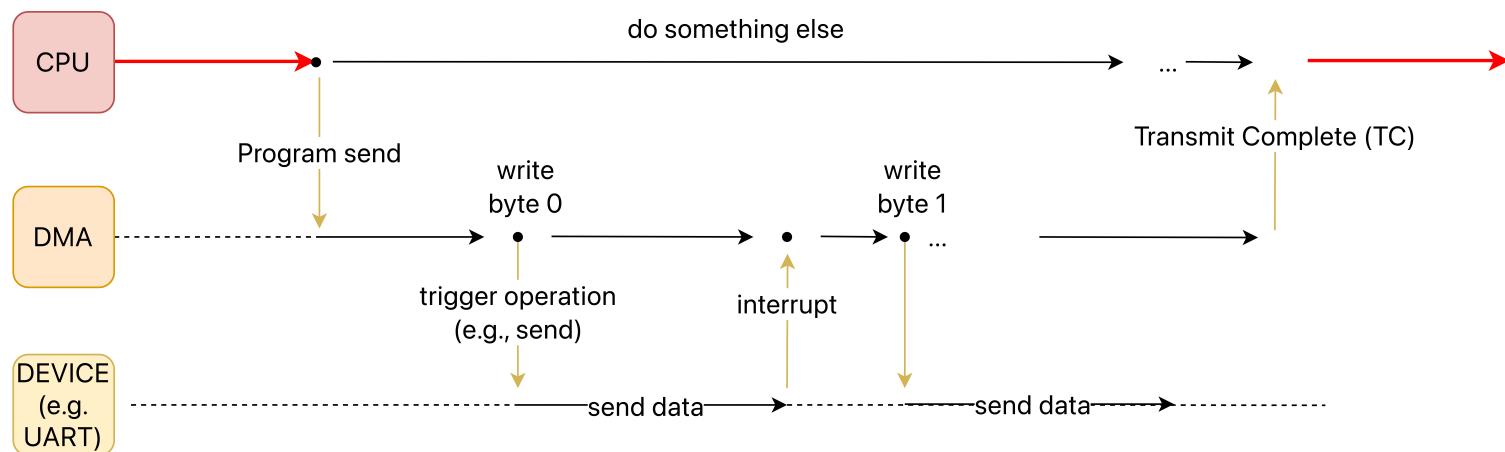
Interrupts

- Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task.
- When the device is finally finished with the operation, it will raise a **hardware interrupt**, causing the CPU to jump into the OS at a predetermined **interrupt service routine (ISR)** or more simply an **interrupt handler**.



DMA

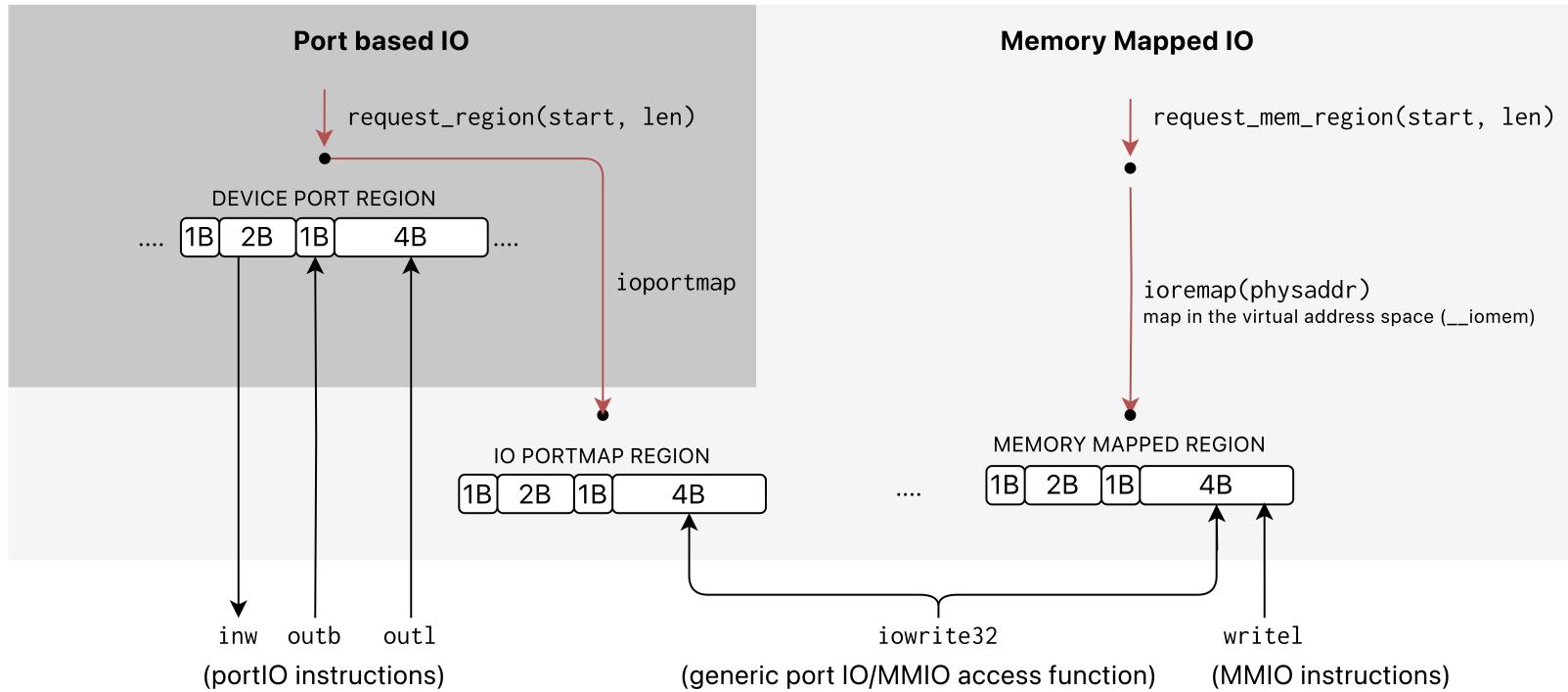
The fundamental concept behind DMA is the introduction of an additional device (DMA controller) that can manage transfers between different devices and memory autonomously, without requiring the CPU's involvement.



Linux low level programming

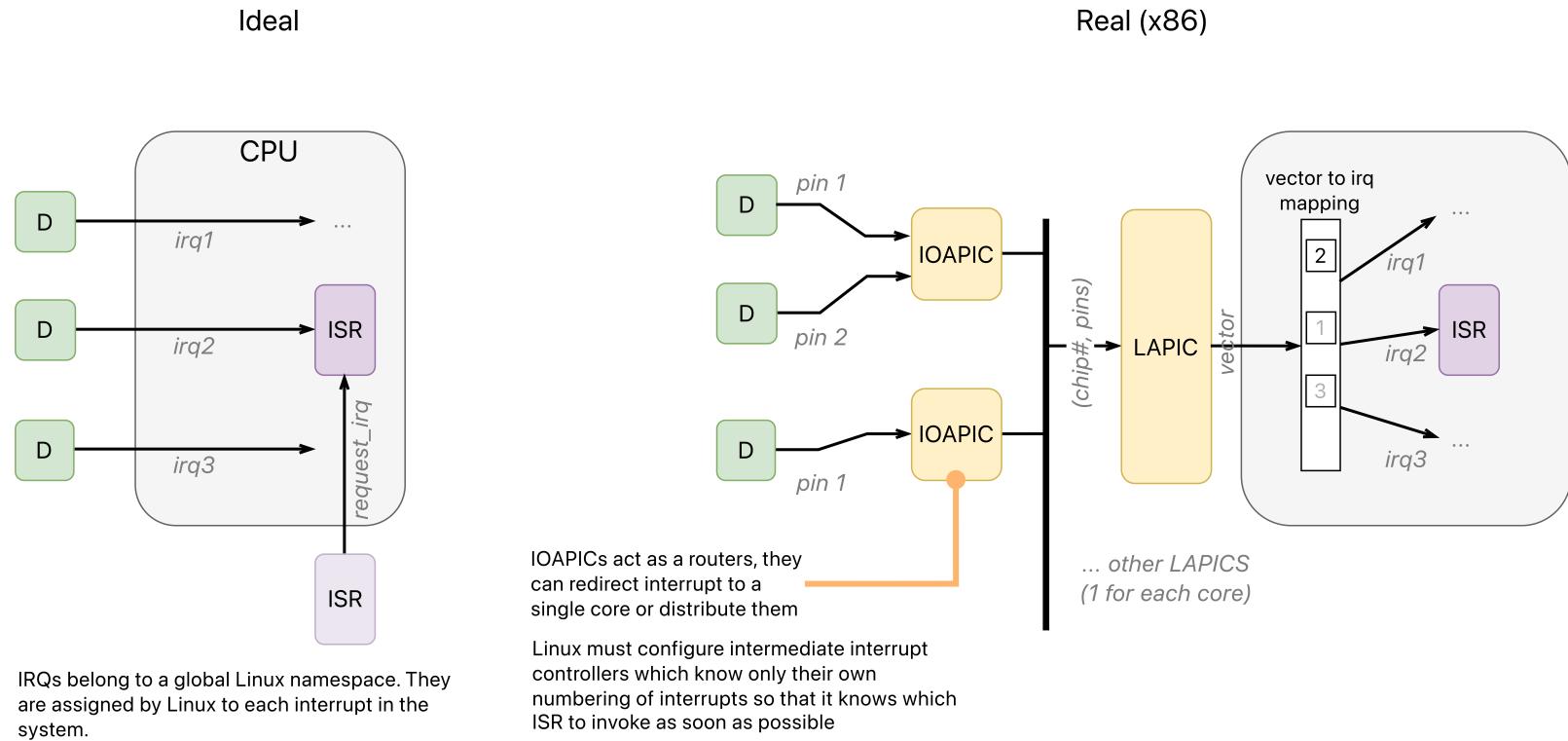
Low level IO

Requesting access to devices



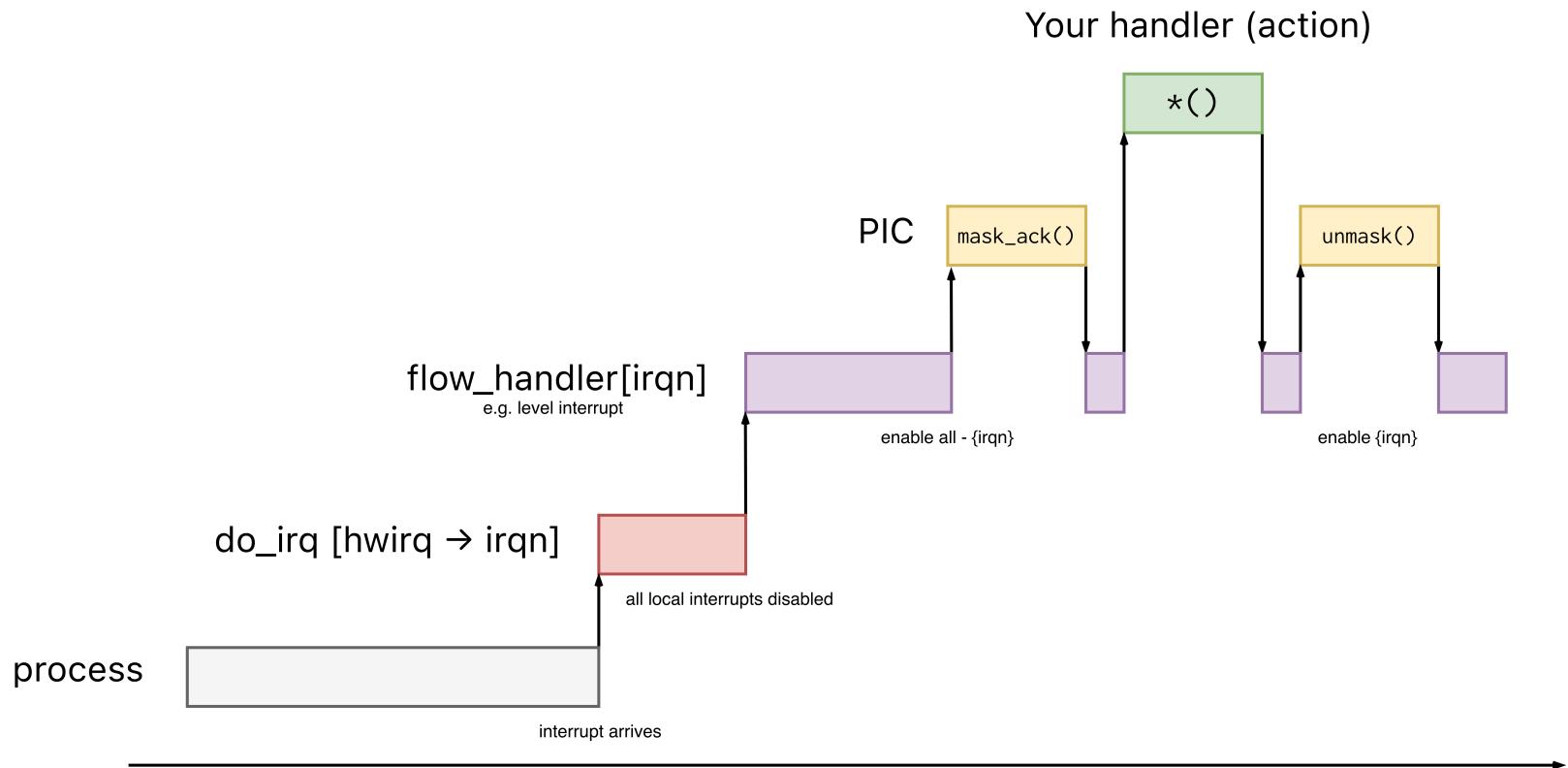
Interrupt management

Interrupt flow



IRQs belong to a global Linux namespace. They are assigned by Linux to each interrupt in the system.

Interrupt flow



Interrupt flow » Registering an action

```
static irqreturn_t handler(int irq, void *mydata) {
    ...
    // acquire locks on shared data
    // read/write from peripherals through MMIO
    // defer work
    // release lock
    return IRQ_HANDLED;
}

static int __init mydriver_init_module(void) {
    ret = request_irq(MY_IRQN, handler, 0, NULL);
    ...
}
```

Interrupt management » Deferring work

Idea

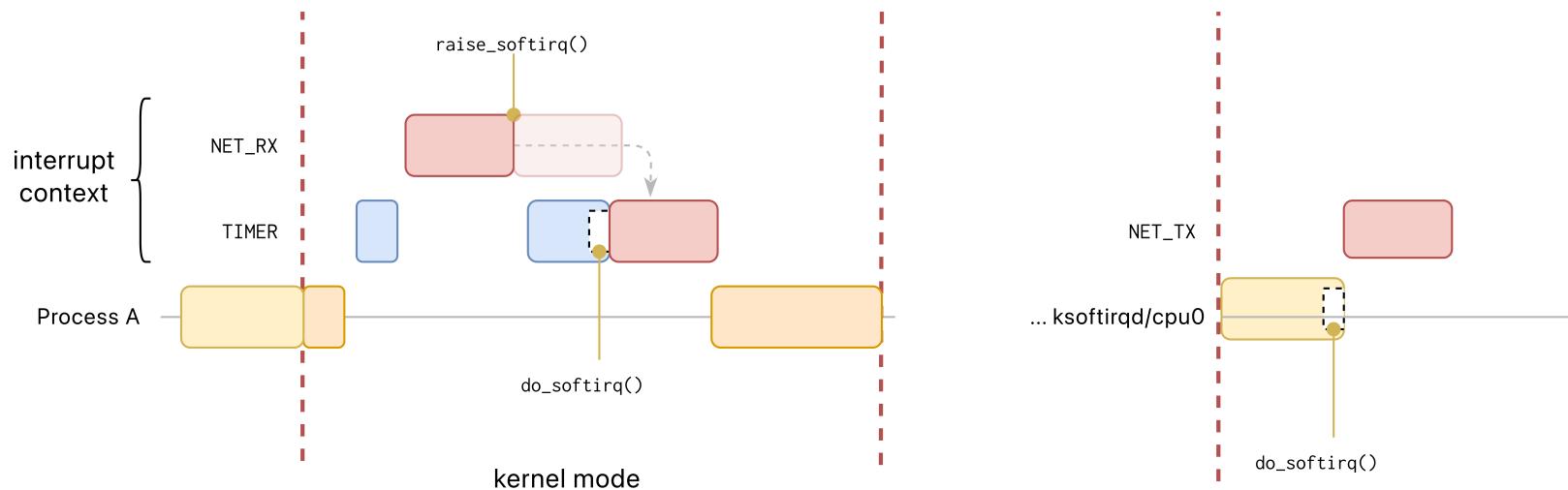
Move the non-critical management of interrupts to a later time to

- Improve responsiveness
- Benefit from aggregation (many small operations merged into a single one)

Implemented by splitting interrupts into two parts:

- **Top half:** executes a minimal amount of work which is mandatory to later finalize the whole interrupt management.
 - Works in a **non-interruptible** context.
 - Schedules some deferred work (**deferred functions**) through the SoftIRQs infrastructure.
- **Bottom half:** Finalizes the work specified by the SoftIRQs at particular **reconciliation points** in time.

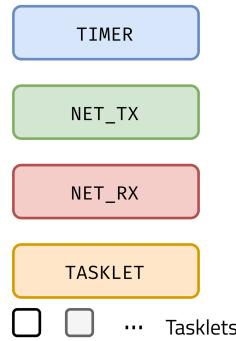
SoftIRQs



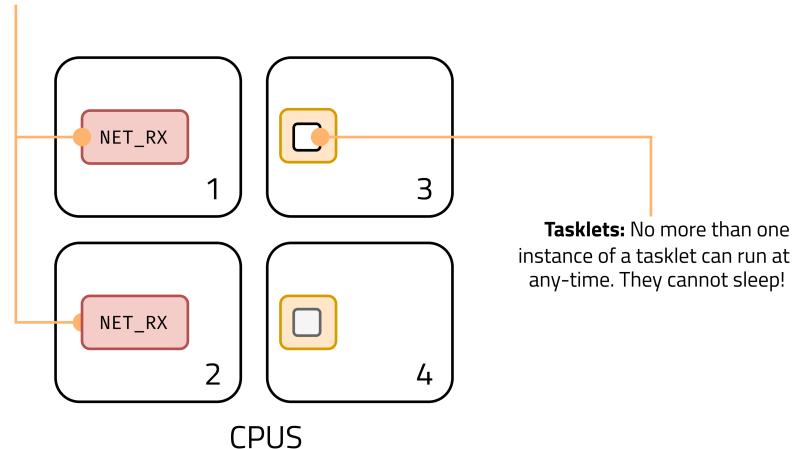
Tasklets

Tasklets are a way to exploit SoftIRQs with ease. They have a simpler interface and kernel ensures no more than one instance of them is running concurrently. It is a one-shot deferral scheme: If you schedule two, only one is ran.

There are very few types of SoftIRQ and they are difficult to program



SoftIRQs never interrupted on the same CPU. However can run at the same time on another



Tasklets

A tasklet is a pointer to a function plus some data; it is represented by the kernel with a list of `tasklet_struct`

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state; /* 0, scheduled or running */
    ...
    void (*func)(unsigned long);
    unsigned long data;
};

...
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, my_data);
...
tasklet_schedule(&my_tasklet); // this is invoked by the interrupt handler to
```

Work queues

- What if your deferred action must block? I.e., allocate a lot of memory, obtain a semaphore, or perform block I/O.
- Use a **work queue** which is a schedulable entity that runs in process context to execute your bottom half.
- General mechanism to submit work to a **worker kernel thread** (`events/n`)

To create work for the `events/n` thread:

```
DECLARE_WORK(work, void (*func)(void *), void *data);
```

To enqueue it:

```
schedule_work(&work);
```

Timers

The `hrtimer` subsystem provides precise, high-resolution software timers for kernel use. It enables sub-millisecond timing, suitable for real-time tasks.

```
struct hrtimer my_timer;
static enum hrtimer_restart my_timer_callback(struct hrtimer *timer) {
    // Optional if you want to repeat it
    hrtimer_forward_now(timer, ms_to_ktime(100));
    return HRTIMER_RESTART;
    // otherwise
    // return HRTIMER_NORESTART;

}

my_timer.function = my_timer_callback;
hrtimer_init(&my_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
hrtimer_start(&my_timer, ms_to_ktime(50), HRTIMER_MODE_REL);
```

Flags:

- `HRTIMER_MODE_REL/ABS` Timer expires after a *relative* interval from the moment it's started or at an absolute time
- `CLOCK_REALTIME / CLOCK_MONOTONIC` Choose between wallclock time or a monotonic time reference.

Linux Device management

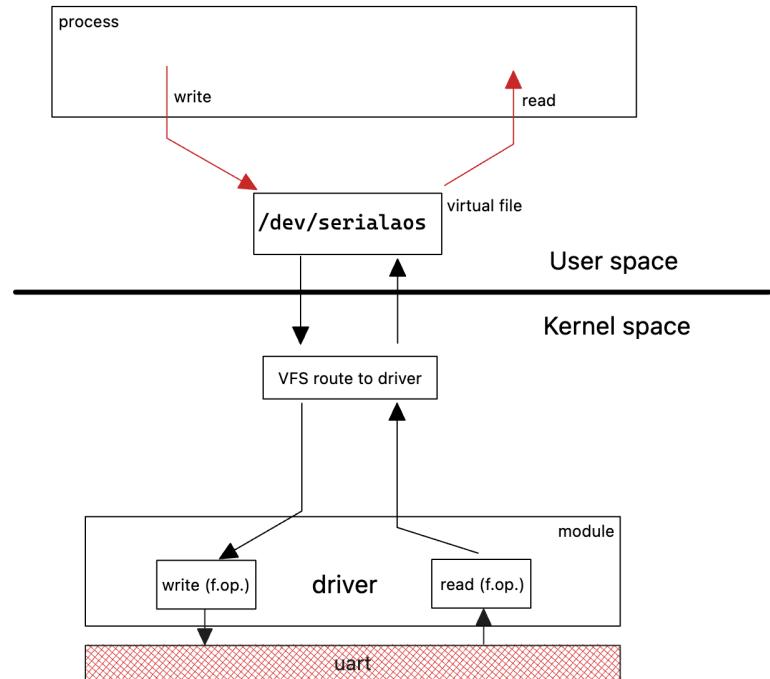
Introduction to devices in Linux

Device categories

- Devices are divided into three categories:
 - A **character device** is characterized by a **character stream**. Writing or reading from the file has **direct impact on the device itself**; no buffering is performed
 - A **block device** is seen as a sequence of numbered blocks. Each block can be individually addressed and accessed through a cache (random access).
 - A **network device**, essentially a sequence of packets.

Device special files

- Linux integrates devices into the file system as **special files**
- Each device is assigned a path name, usually in `/dev`. For example, if you had a uart device named 'serialaos' 😊, this would be assigned the file `/dev/serialaos`.



Device major and minor numbers

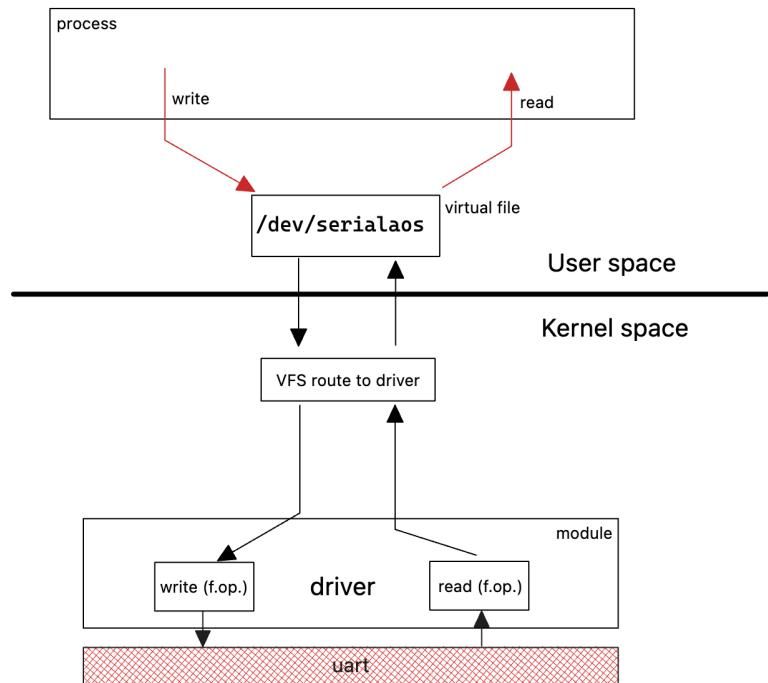
- Each driver has what is called a **major device number** that serves to identify it.
- If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device** number that identifies it.
- Example (same driver, different disks or partitions)

```
ls -l /dev/sd*
#
#           +--- major
#           V   V--- minor
brw-rw---- 1 root disk 8, 1 9 apr 09.31 /dev/sda1
brw-rw---- 1 root disk 8, 2 9 apr 09.31 /dev/sda2
```

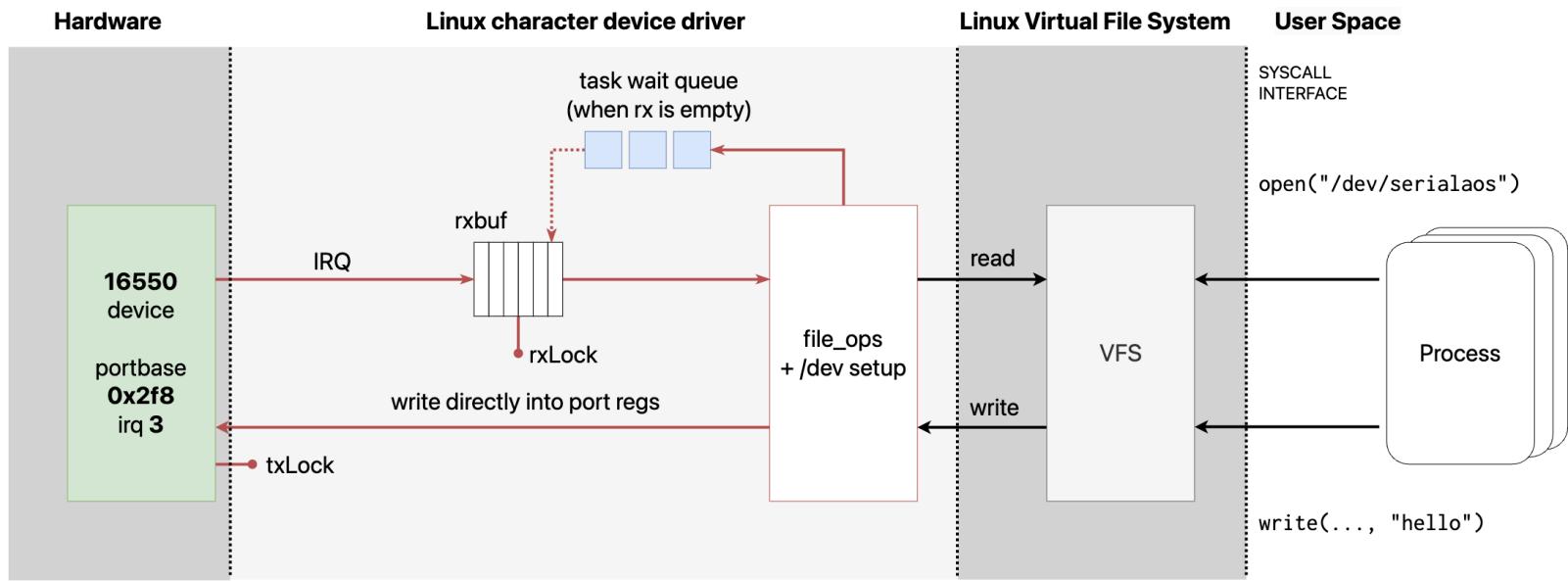
Low level device management » Char devices

Low level driver interface

```
struct file_operations {
    ...
    ssize_t (*read)
        (struct file *, char *, ...);
    ssize_t (*write)
        (struct file *, char *, ...);
    int (*open)
        (struct inode *, struct file *);
    ...
}
```



Low level driver interface » Hands-on lab topics



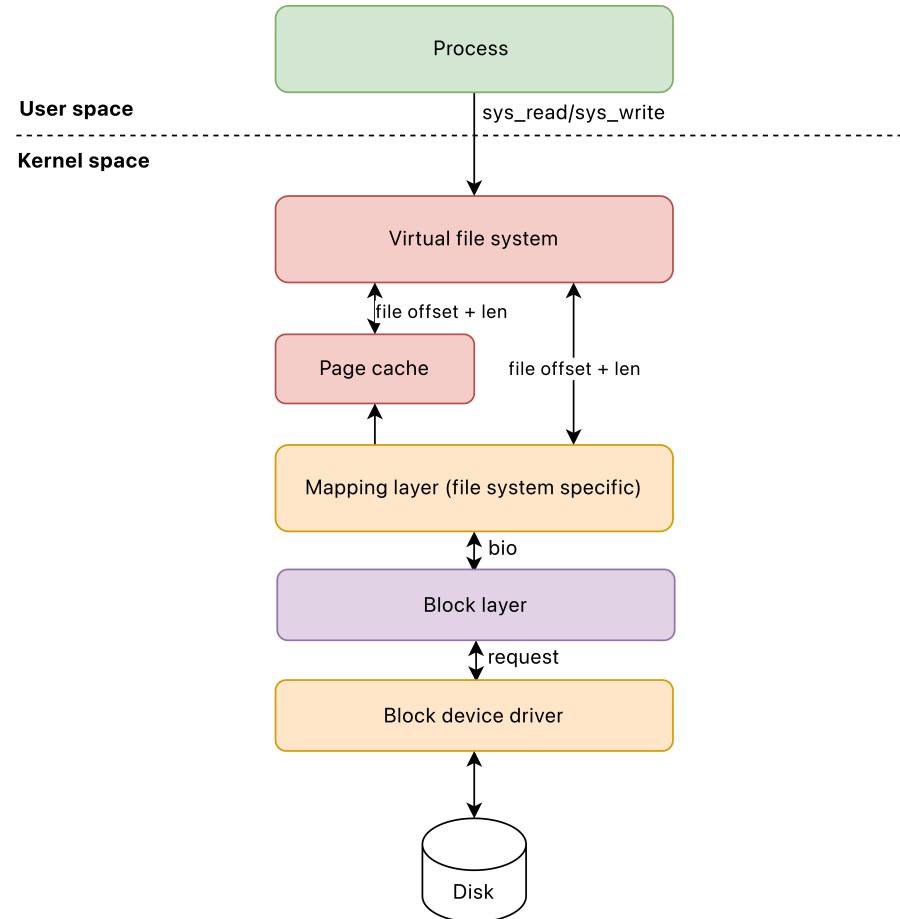
Serialaos driver as implemented during the lab, using only low-level character device primitives

Low level device management » Block devices

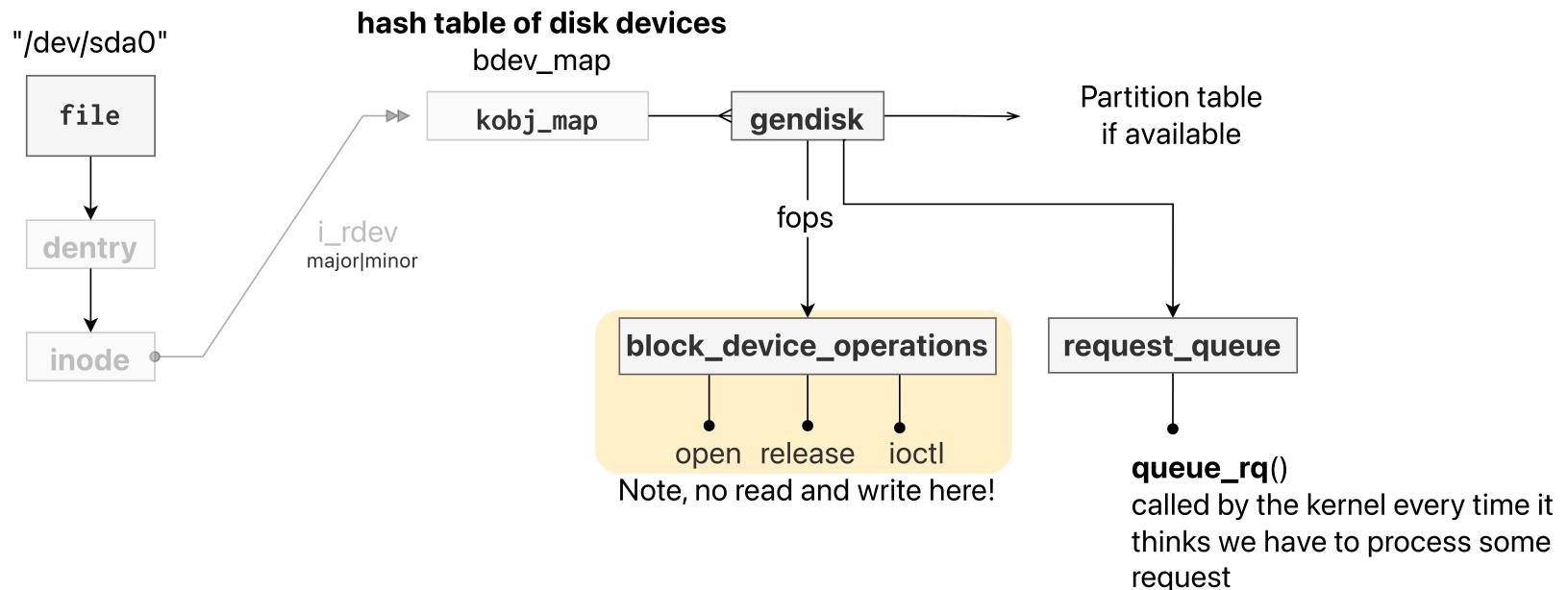
Introduction

- Manage physical storage devices (hard disks, SSDs)
- **Block-Based I/O:** Data processed in indexed chunks (512B - 4KB) instead of byte-by-byte
- **Random Access:** Access any block index, any time.

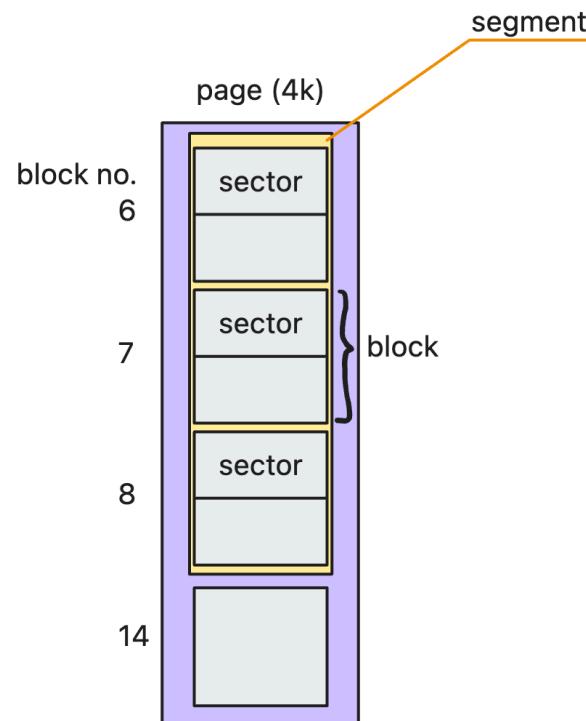
Introduction



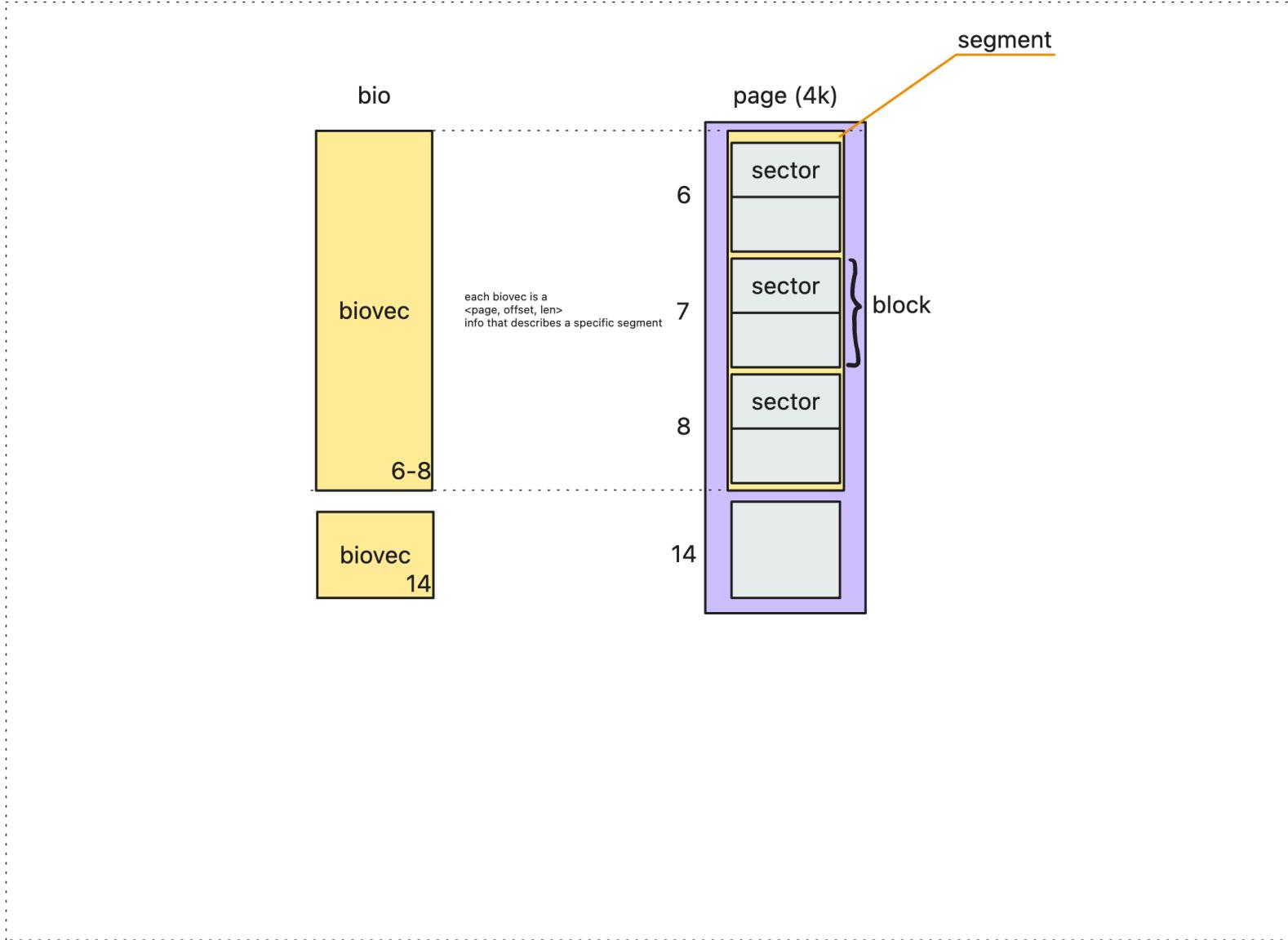
Low level driver interface



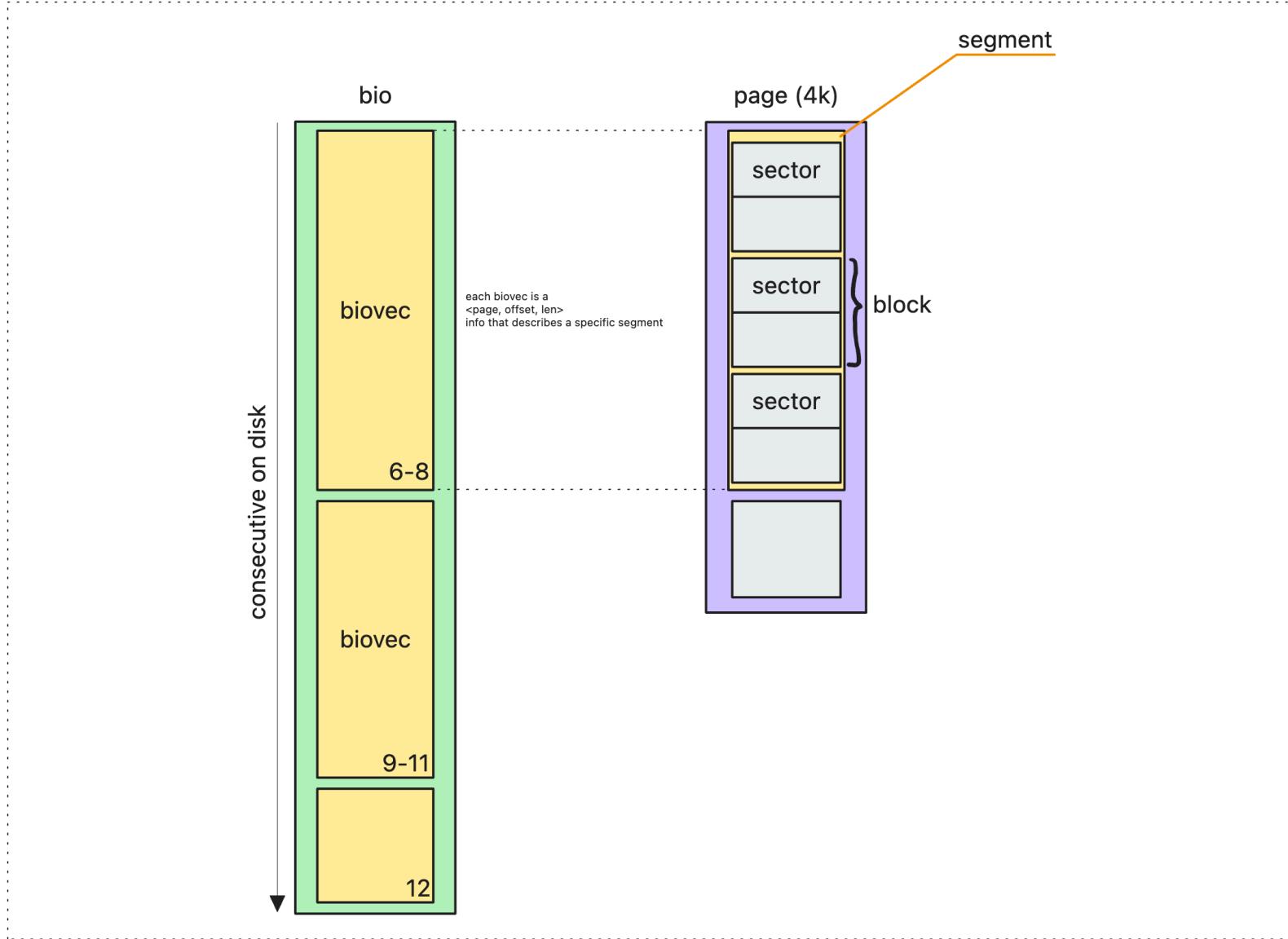
Page cache to Block requests



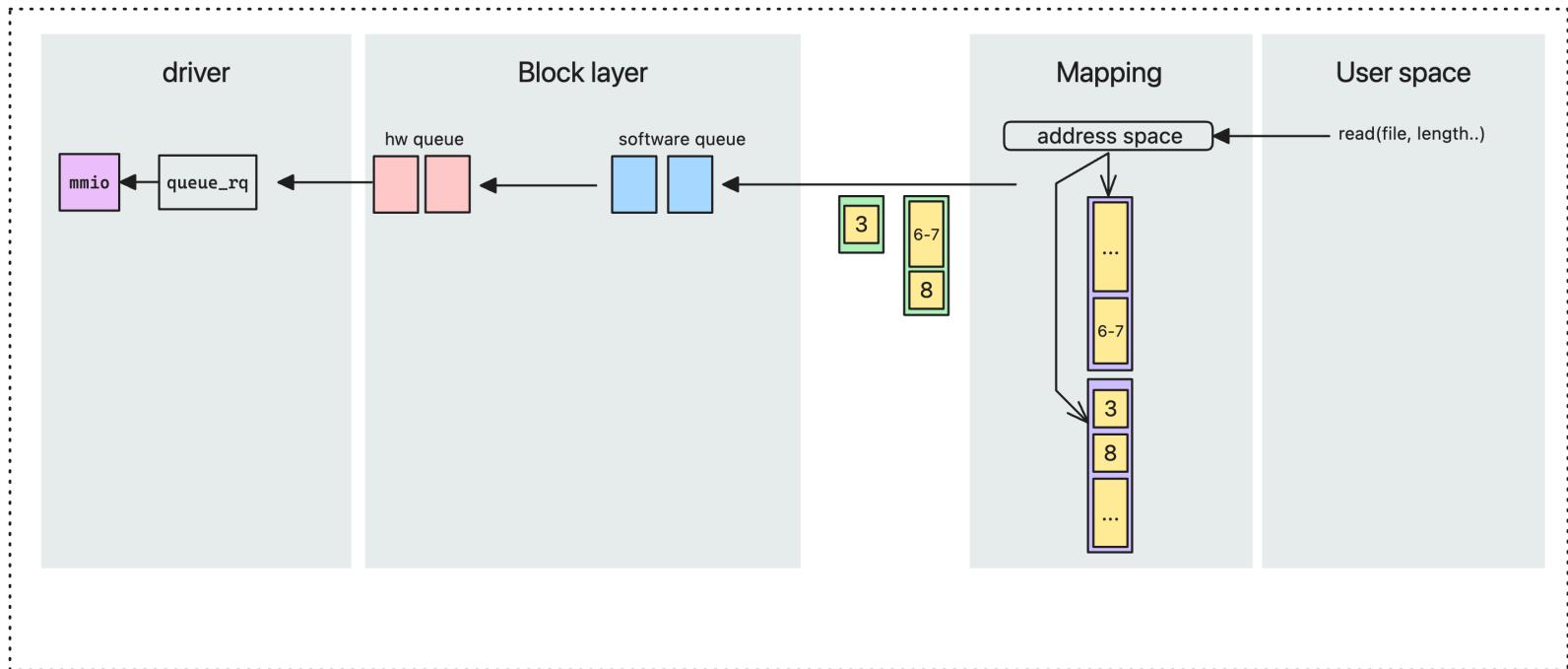
Page cache to Block requests



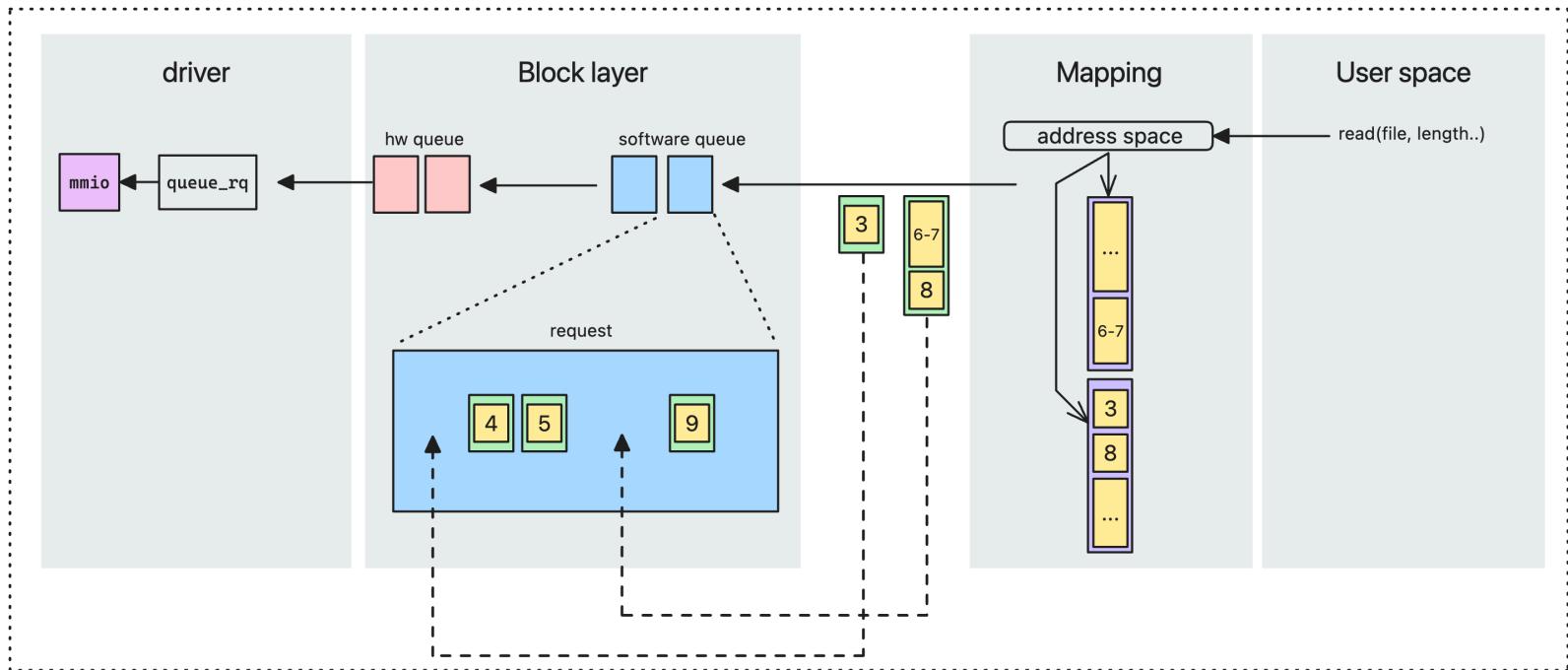
Page cache to Block requests



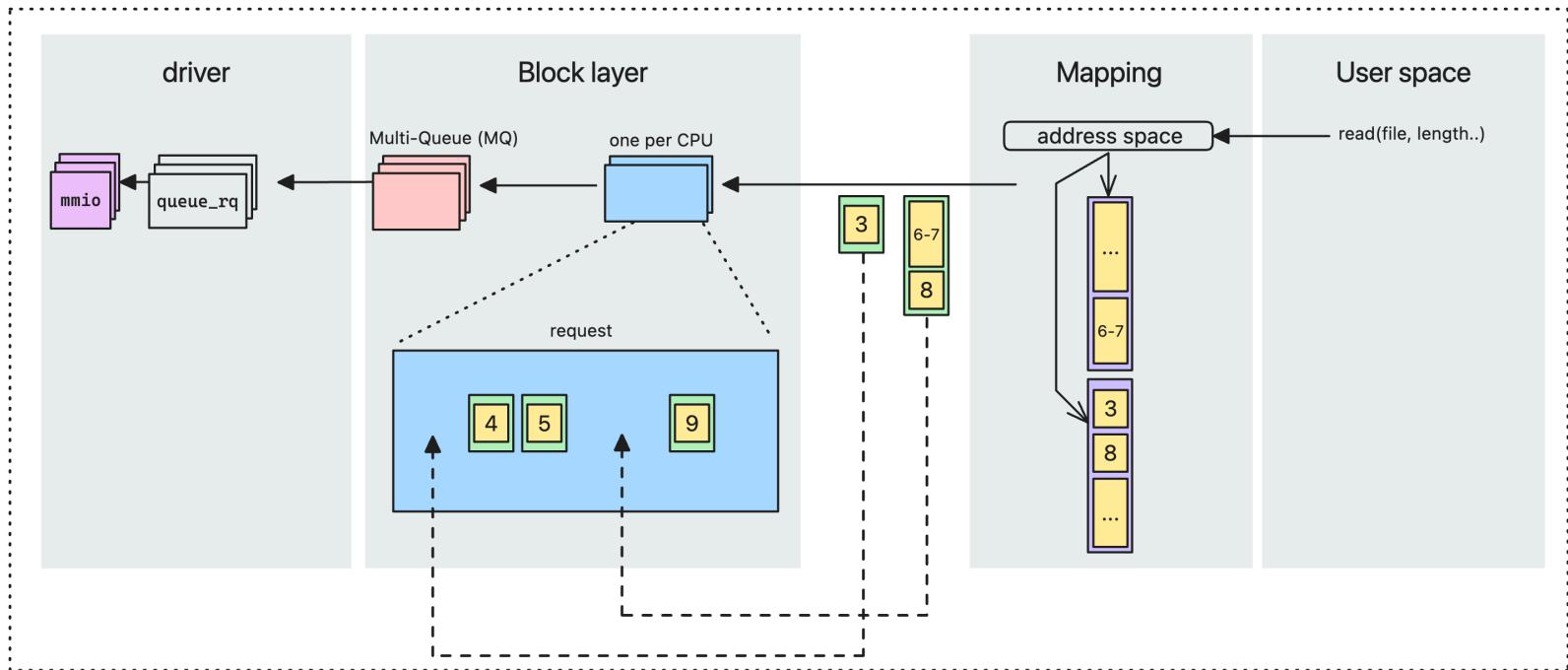
Block requests' management



Block requests' management



Block requests' management



Low level device management » Block devices » IO Schedulers

Introduction

I/O schedulers can have many purposes depending on the goals; common purposes include the following:

- To minimize time wasted by hard disk seeks (still relevant in some cases).
- To prioritize a certain processes' I/O requests.
- To give a share of the disk bandwidth to each running process.
- To guarantee that certain requests will be issued before a particular deadline.

Introduction

History of IO Schedulers

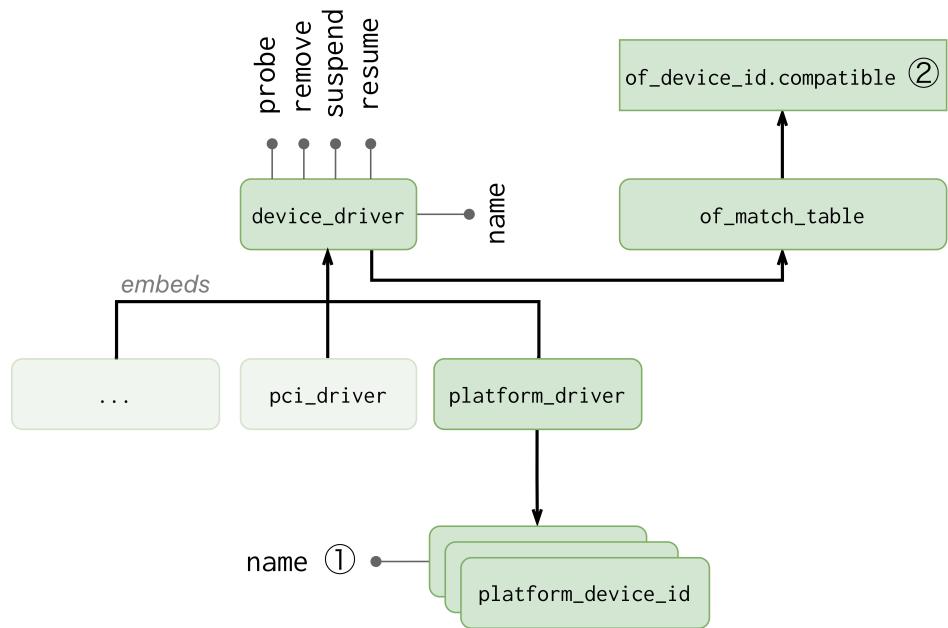
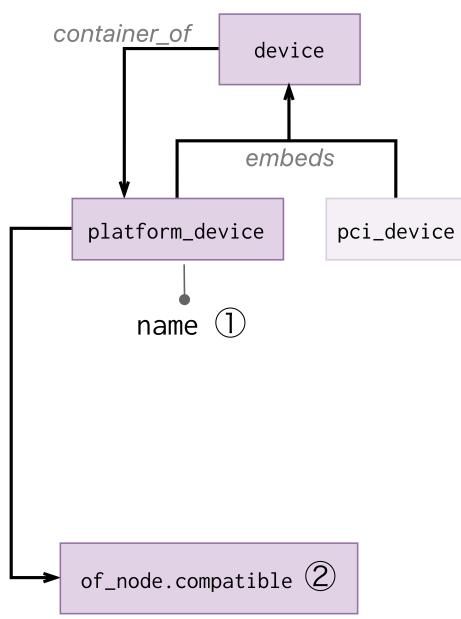
- Linux 2.4: Linus Elevator
- Linux 2.6: Deadline, Anticipatory (removed in 2.6.33), CFQ (Complete Fairness Queueing), Noop (No operation).
- Linux 3.0: Deadline, CFQ, Noop.
- Linux 4.12: MQ-Deadline, CFQ, BFQ (Budget Fair Queueing), Kyber, Noop.
- Linux 4.11: MQ-Deadline, CFQ, Noop (note, first introduction of blk-mq ca 2013)
- Linux 5.0: **MQ-Deadline, BFQ (Budget Fair Queueing), Kyber, Noop.**

High level device management » The device model

What is a device model?

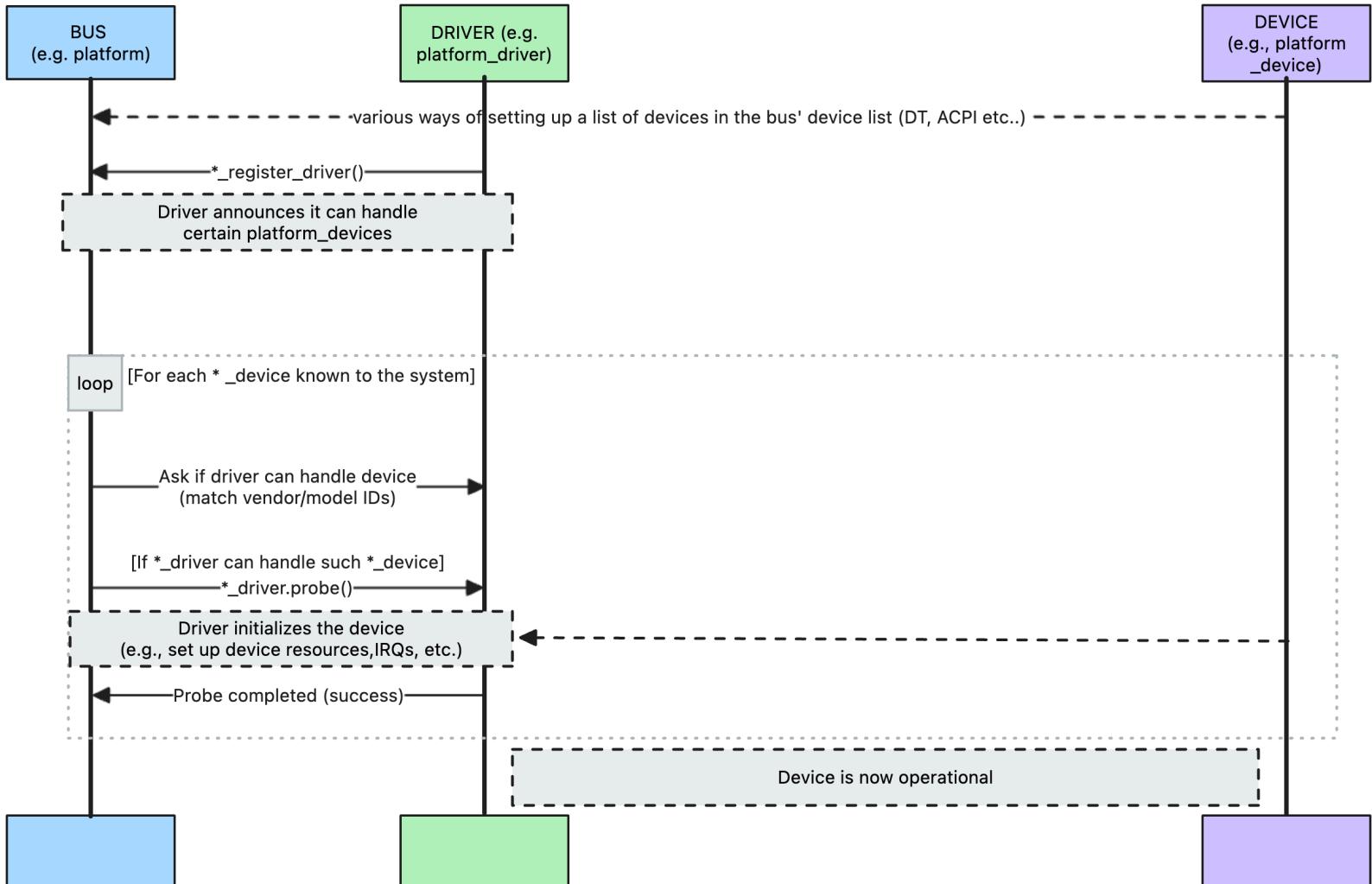
- The Linux kernel runs on several architectures and hardware platforms
- How maximize the reusability of code between platforms?
 - Example: the same USB device driver to be usable on a x86 PC, or an ARM platform, even though the USB controllers used on these platforms are different.
- How to manage device lifetime (new device plugged in for example) without reinventing the wheel?
- This requires a clean organization of the code which is enforced by the kernel through an appropriate API based on a **framework/bus** matrix

The Bus/Driver/Device triplet



- ① Matching done through names between supported device_ids and device names
- ② Matching done through the device tree (`of_node`) compatible string of the device and the `of_device_id.compatible` property of the driver

Driver/Device matching



Runtime representation (Sysfs and Kobjects)

- Kobjects are the **underlying foundation of the device model**
- Represent kernel objects such as **bus**, **devices**, **drivers**, and modules
- kobjects have fields (or attributes) and are exposed to user space via the `sysfs` virtual filesystem (kobject → directory, attributes → file)
- kobjects belong to **ksets**, and the corresponding hierarchy is useful for mimicking real hardware topology
- **kobjects emit uevents** to notify user-space tools like udev of changes
- Example: inserting a usb pen drive, creates entries in `/sys/usb/devices` , `/sys/class/block` and `/dev/sdb` (through udev).

High level device management » The device model » Bus frameworks

The Platform bus framework

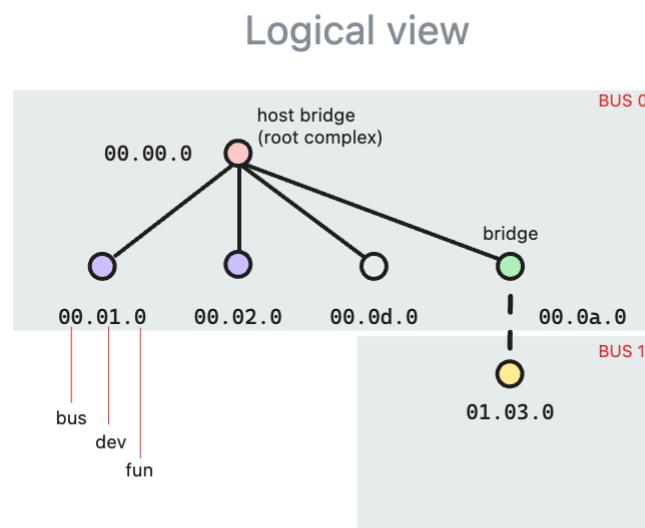
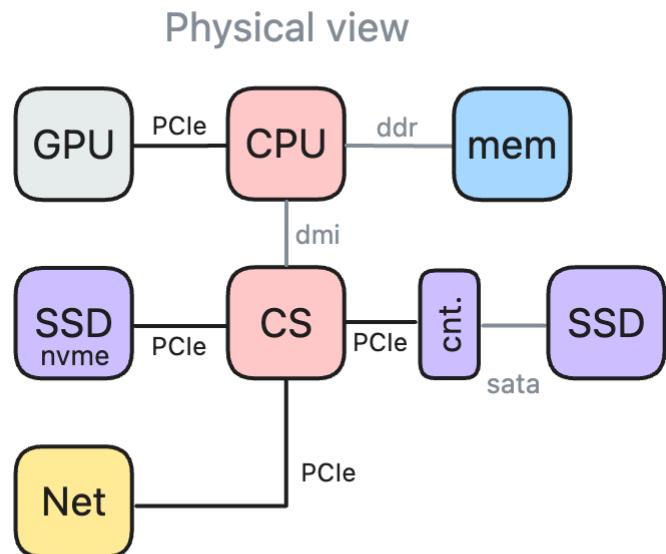
What is it

- Designed for system-on-chip (SoC) and embedded devices
- Manages non-easily-discoverable hardware components
- Represent fixed hardware resources
- Defined in board-specific code or Device Tree

Key Advantages

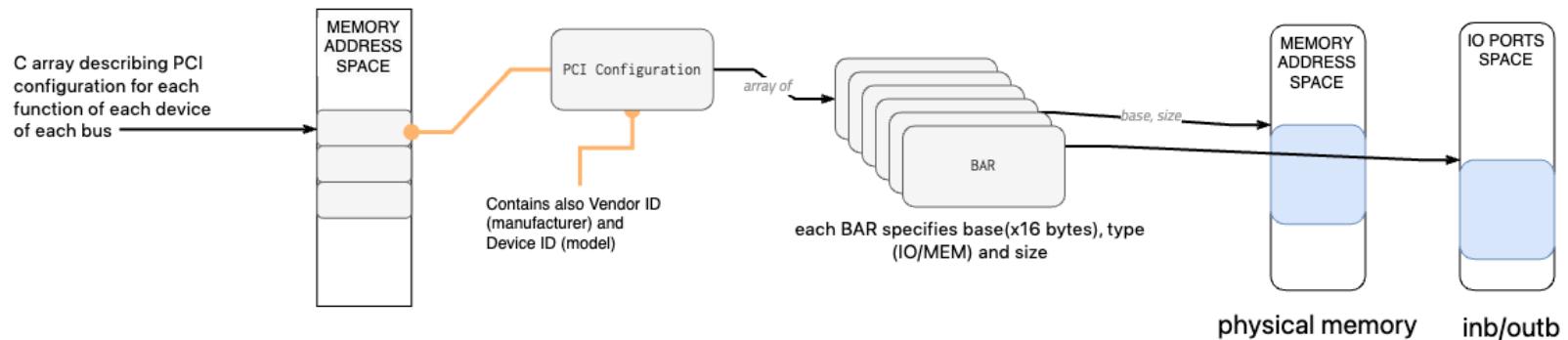
- Simplifies driver development for embedded systems
- Encourages code reusability
- Maintains clean and maintainable codebase for board-specific logic

The PCI bus framework » Logical/Physical view



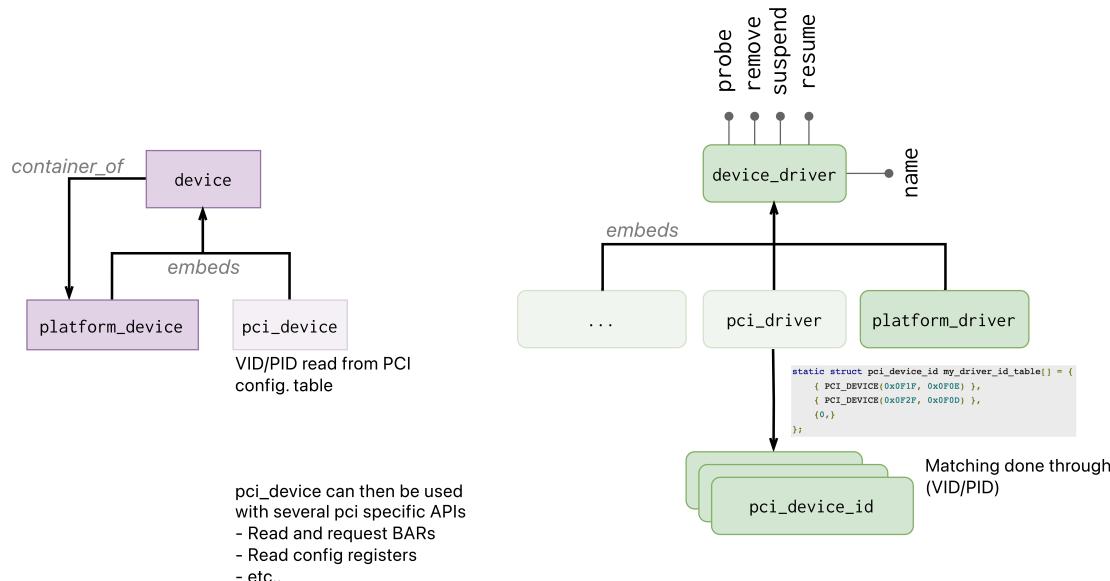
The PCI bus framework » Discovery

Extended configuration access mechanism (PCIe)



The PCI bus framework » Discovery

```
$ dmesg
...
pci 0000:00:01.3: [4431:2212] type 00 class 0x040000
pci 0000:00:01.3: quirk: [io 0x0600-0x063f] claimed by PIIX4 ACPI
pci 0000:00:01.3: quirk: [io 0x0700-0x070f] claimed by PIIX4 SMB
...
...
# [VID : DID]
pci 0000:00:02.0: [1234:1111] type 00 class 0x030000
pci 0000:00:02.0: reg 0x10: [mem 0xfd000000-0xfffffff pref]
pci 0000:00:02.0: reg 0x18: [mem 0xfebb0000-0xfebbffff]
pci 0000:00:02.0: reg 0x30: [mem 0xeba0000-0xebaffff pref]
```

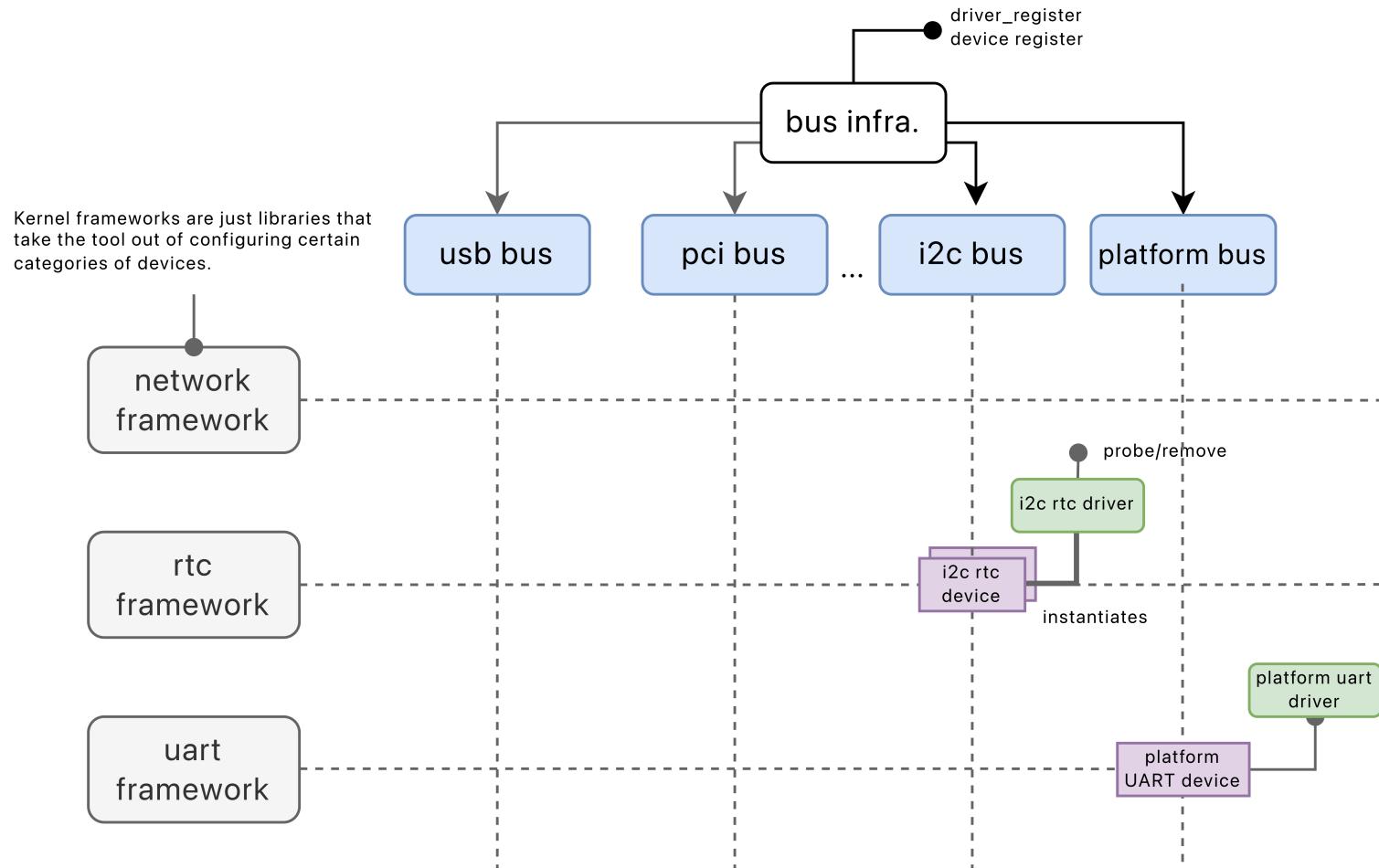


The PCI bus framework » Inspection

```
root@e43fd856ef30:/scripts# lspci
+---- bus number
| +---- device number
V V V---- function
00:00.0 Host bridge: Intel Corporation ...
...
00:0a.0 PCI bridge: Intel Corporation 8
00:0d.0 SATA controller: Intel Corporation ...
...
01:03.0 Ethernet controller: Intel Corporation 82540EM
```

High level device management » The device model » Subsystem-specific frameworks

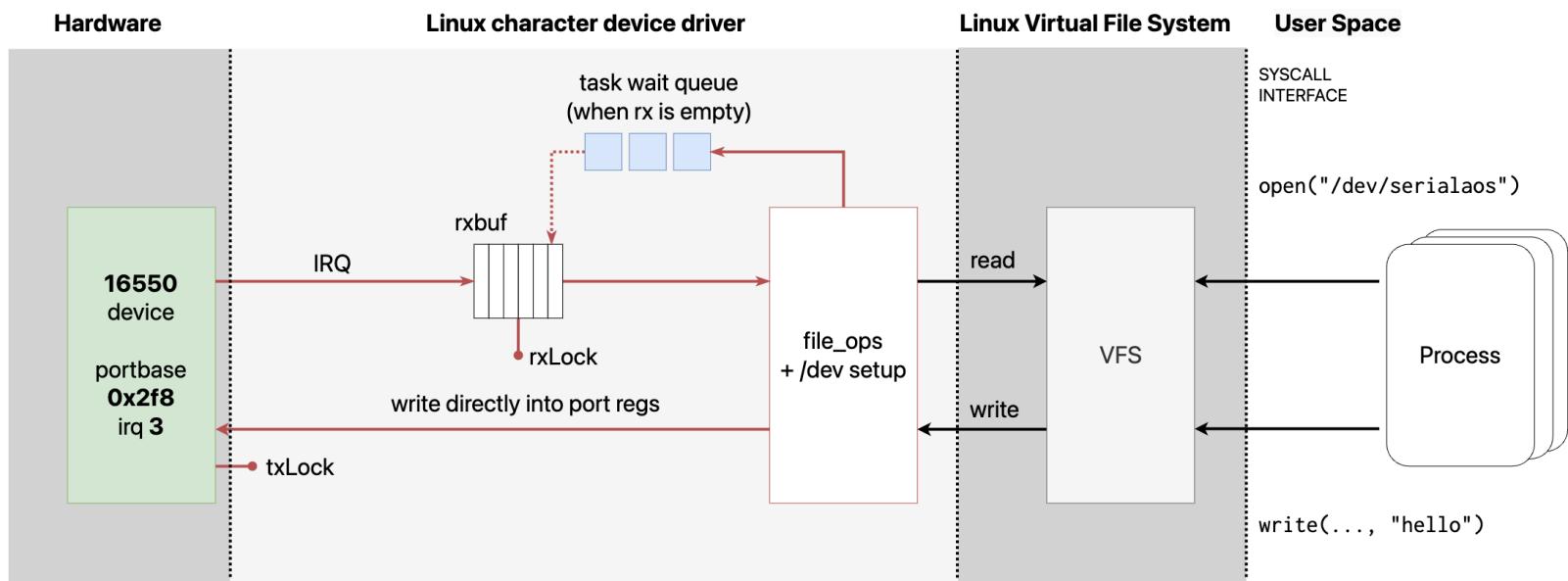
Introduction



List of frameworks

- **RTC (Real Time Clock) framework:** For clocks, provides APIs for time read/set, alarms, etc.
- **UART/serial framework:** For serial interfaces and UARTs
- **GPIO framework:** For generic input/output pin control
- **IIO (Industrial I/O):** For sensors, ADCs, and industrial data acquisition devices
- **DMAEngine framework:** For direct memory access management
- **Media (V4L2/Camera/Audio) frameworks:** For cameras, microphones, and multimedia devices
- **Memory Technology Devices (MTD):** For flash and other memory chips
- **Thermal framework:** For temperature sensors and cooling devices
- **MMC/SD/SDIO framework:** For SD/memory cards
- **SCSI and ATA frameworks:** For disks and storage interfaces
- **Netdevice framework:** For network cards and network interfaces
- **Framebuffer/display framework:** For graphics display devices
- **Input device framework:** For keyboards, mice, touchscreens

Example » The UART framework



Serialaos driver as implemented during the lab, using only low-level character device primitives

Example » The UART framework

