

DEVICE ACCESS

Alex Pajuelo and Juan Jose Costa

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Warning!

- This presentation assumes that you are proficient in the following concepts, previously exposed in the OS course:
 - Device independence
 - Virtual, logical and physical devices
 - I-nodes
 - File Descriptor Table
 - Opened File Table
 - Inode Table
 - I/O system calls
- Review the previous topics of Tema 4: Entrada / Salida of the OS course

Hardware heterogeneity challenge

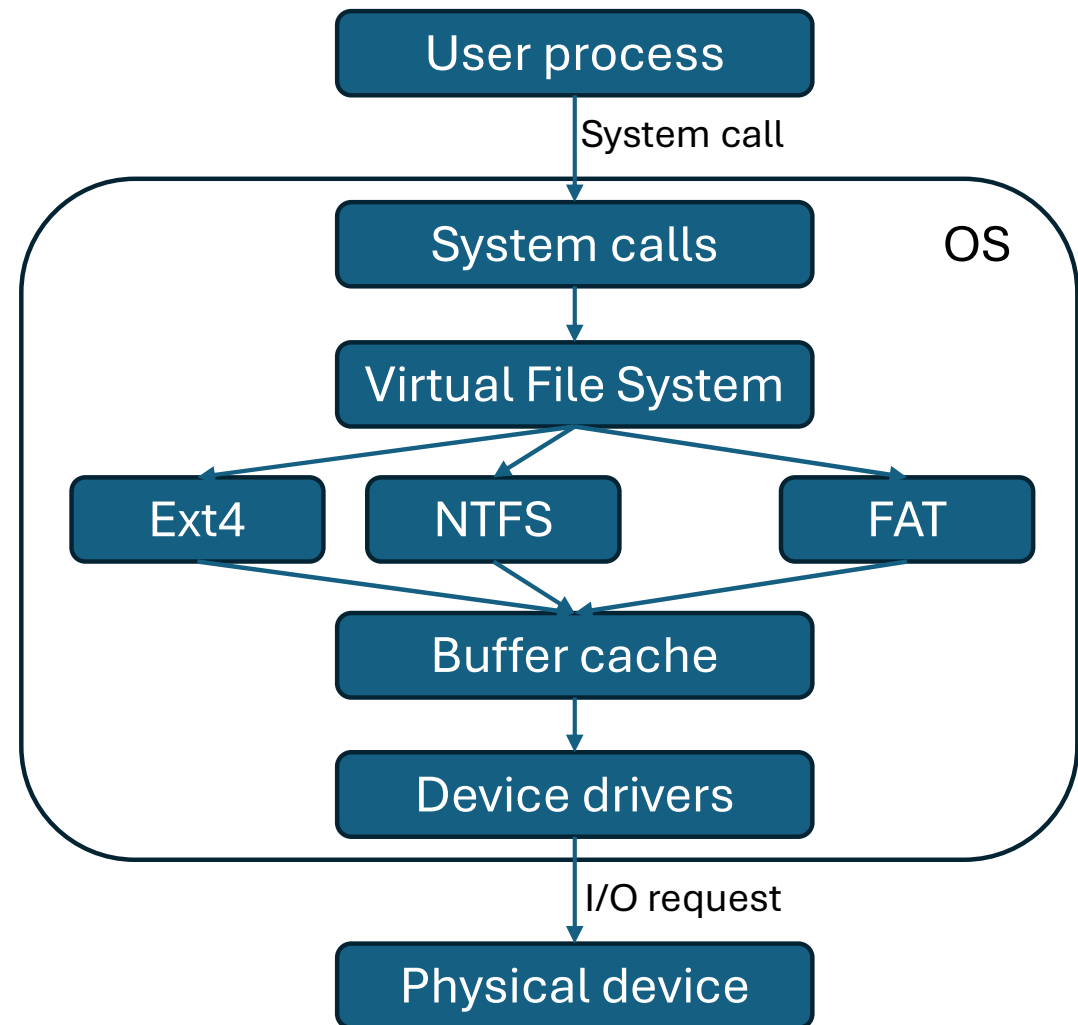
- There are a lot of different I/O interfaces: serial bus (UART, SPI, I2C), parallel bus, PCIe, USB, NVMe, SATA
- There are different transfer models: programmed I/O (PIO), interrupts, direct memory access (DMA/scatter-gather)
- It is unfeasible that a user can implement all the proprietary protocols for every vendor and device
- Solution: OS offers a standard interface to access devices through logical devices (files)

Everything is a file

- Generalization of resources: regular files, folders, sockets, pipes, shared memory and hardware devices are **all represented with the same abstraction**
- That abstraction is a **logical device (file)**
- Offers the **same operations regardless the abstracted resource characteristics**

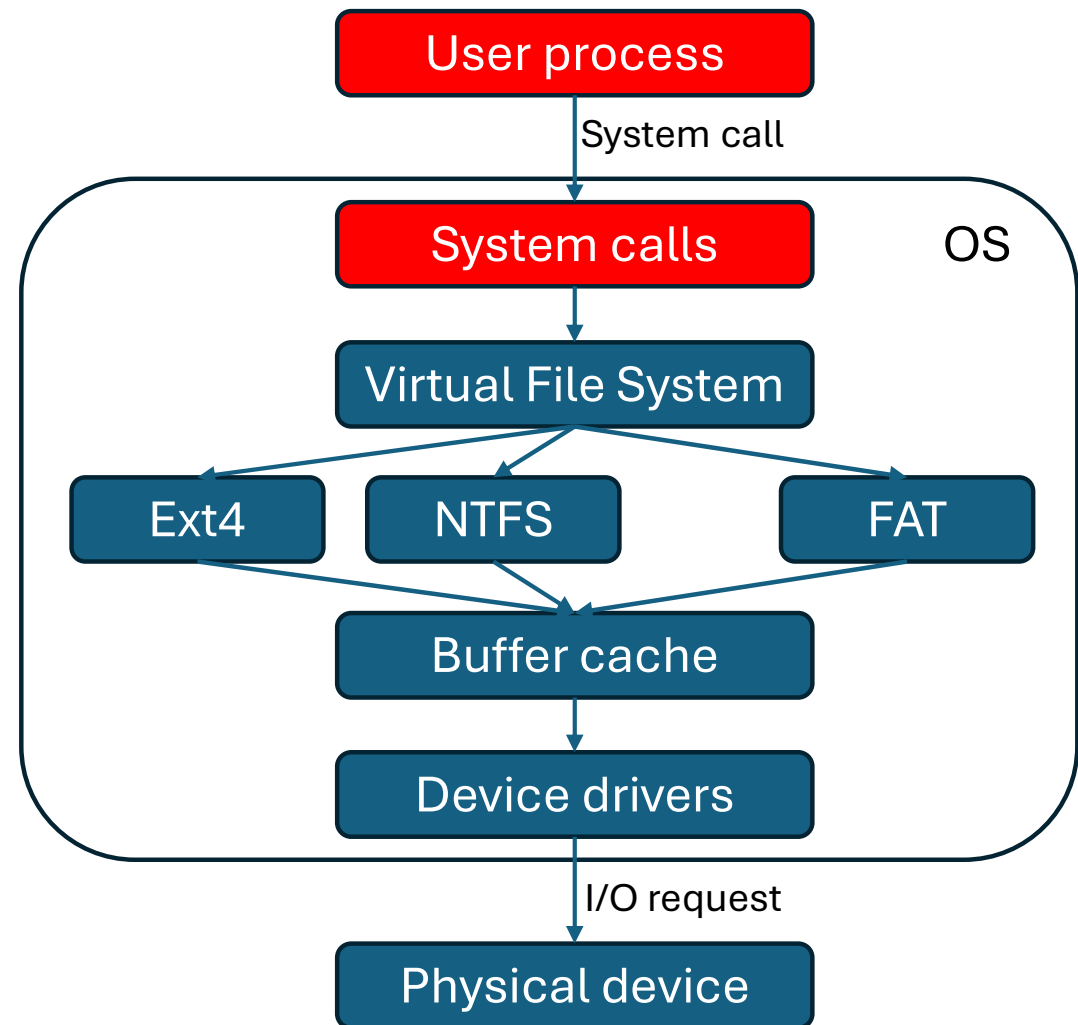
Architecture of I/O

- The I/O architecture is composed by several layers
- Layers are extensible and interchangeable
- Improves modularity, extensibility and compatibility



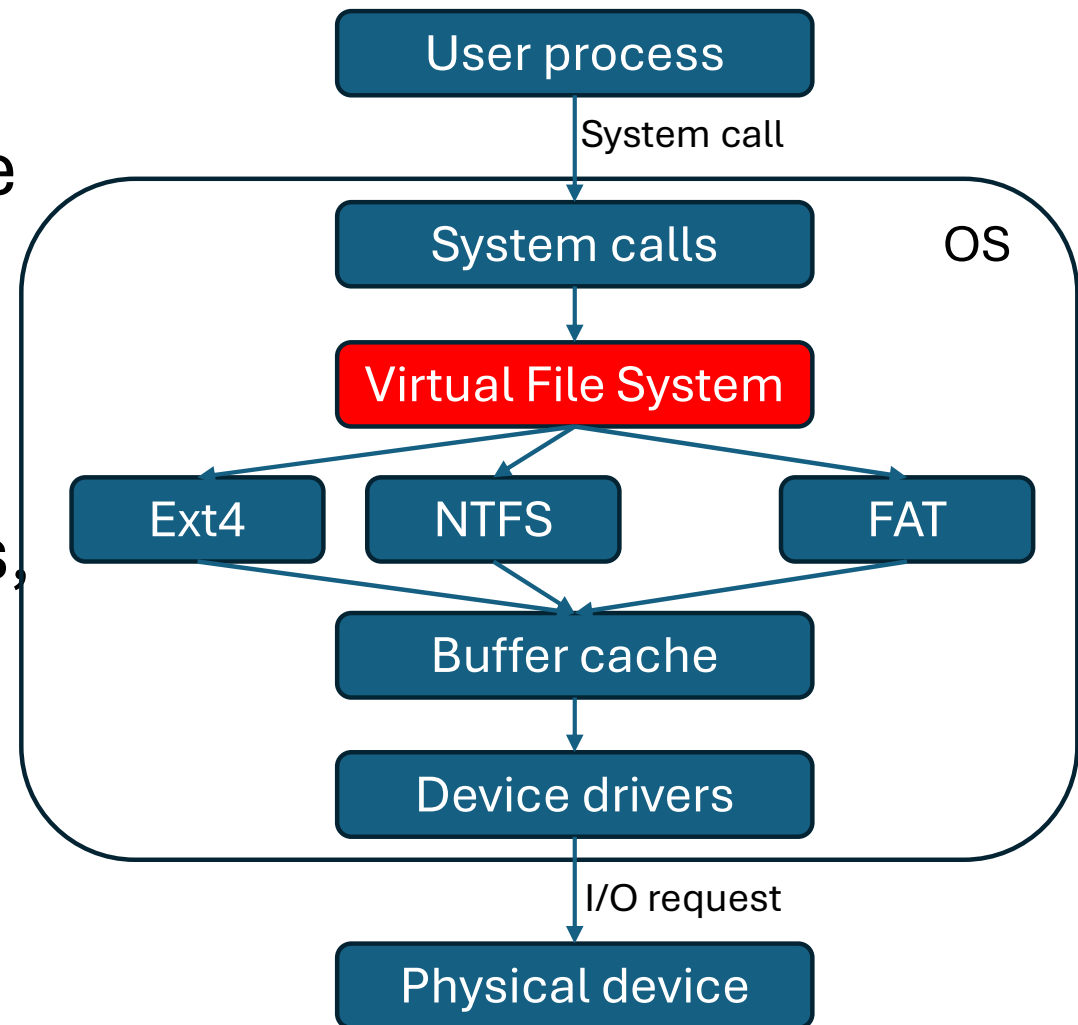
Architecture of I/O

- Already discussed
- The OS must provide system calls to access logical devices



Virtual file system

- The Virtual File System (VFS) is a software indirection layer inside the kernel
- Abstracts physical file systems (ext4, ntfs, ...), virtual file systems (procfs, sysfs, ...) and physical devices
- Provides a unified API to access devices

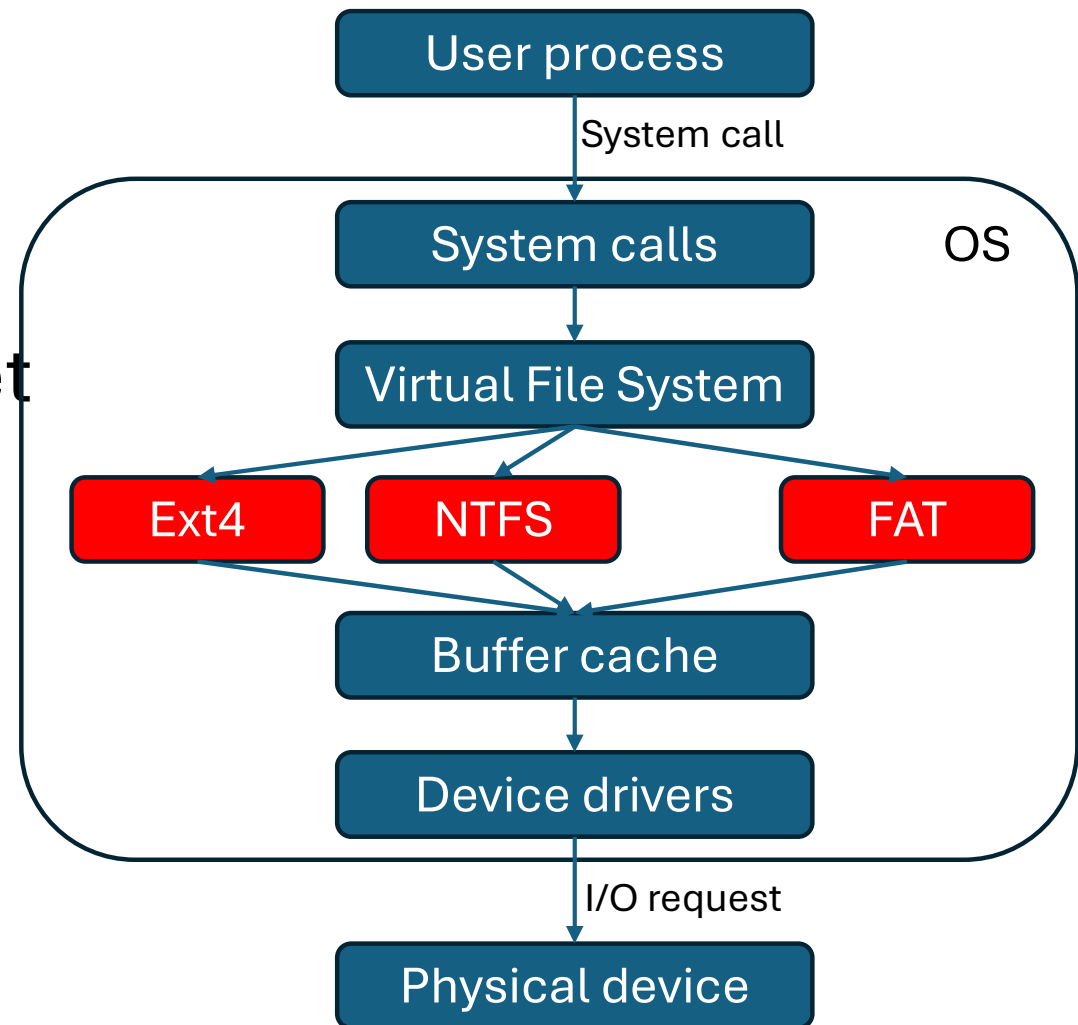


Virtual file system

- VFS is the responsible of translating the generic system calls to logical devices to the specific code for a particular file system
- VFS provides a unified logical view of any file system to the user
- VFS hides all the characteristics and access protocols to the user
- VFS accomplishes this by using **device drivers** and **polymorphism**

Physical file systems

- Physical file systems are drivers that perform the access to a file system in the hard drive
- They know how to interpret the logical file system structure in the hard drive
- They translate file system characteristics to an abstraction the VFS understands



Physical file systems

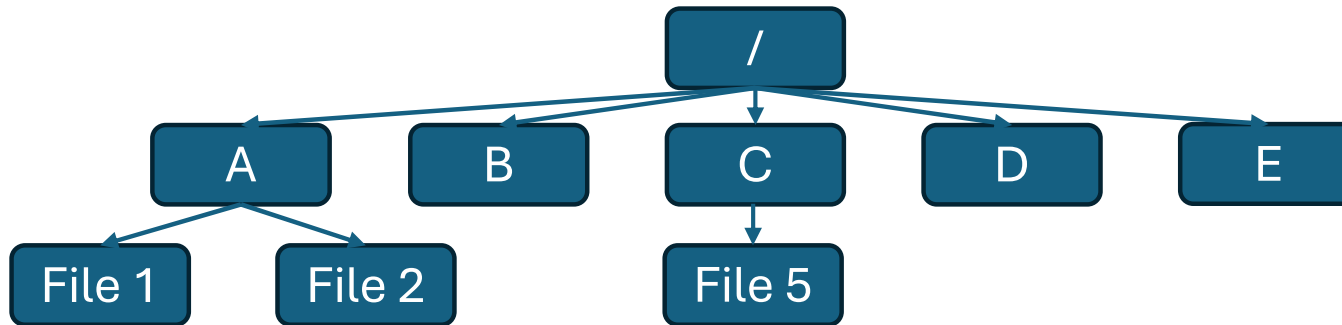
- VFS does not understand the logical structure and characteristics of the file systems stored in a hard driver
- For that, specific file system drivers are provided (like ext4, ntfs, ...)
- Those drivers access a specific file system and translate to the abstraction the VFS provides to the user
- There are several of those drivers depending on the compatibility provided by the OS
- The VFS knows which driver must be called using **mount points**

Mount points

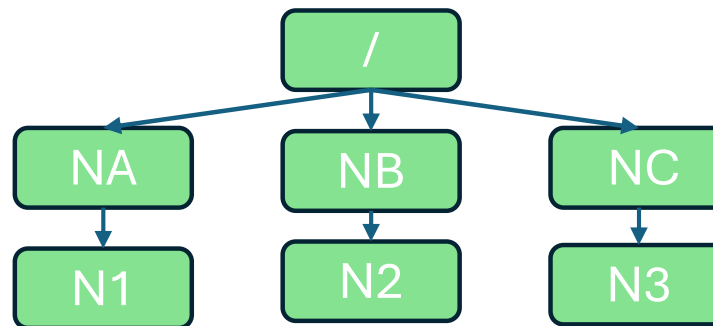
- Before accessing the files and folders of a file system in a hard drive, that file system must be mounted in the global file system tree of Linux
- Mounting means that the file system in a hard drive can be accessed from a given folder in the file system tree
- Mounting is achieved by the command **mount**
- To invoke mount, three parameters are passed: the physical file system to use, the directory in which the partition will be mounted and the logical device associated to the partition

Mount point example

- Current file system tree



- A hard drive with a NTFS file system is accessible from logical device /dev/sdb1



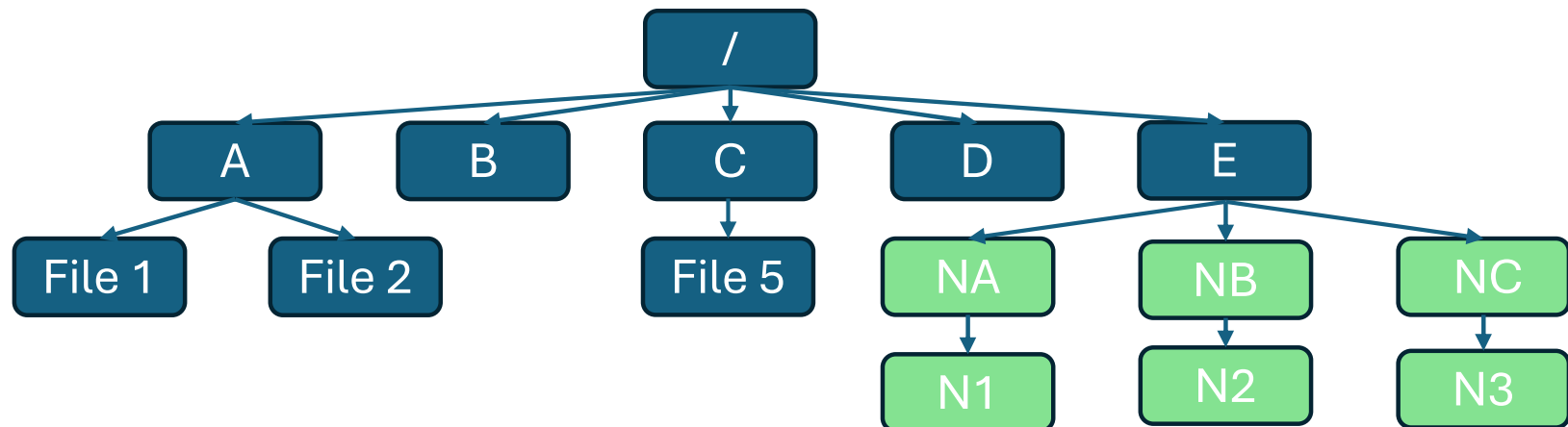
Mount point example

- To make /dev/sdb1 accessible from folder /E

`mount -t ntfs /dev/sdb1 /E`

file system logical device mounting point

- result



How mount works

- The mount command does the following steps:
- Marks the Inode of the folder as mounting point
- Stores in an internal structure the full path of the mounting point, the file system to be used and the mounted logical device
- Notice that the root directory is also a mounting point

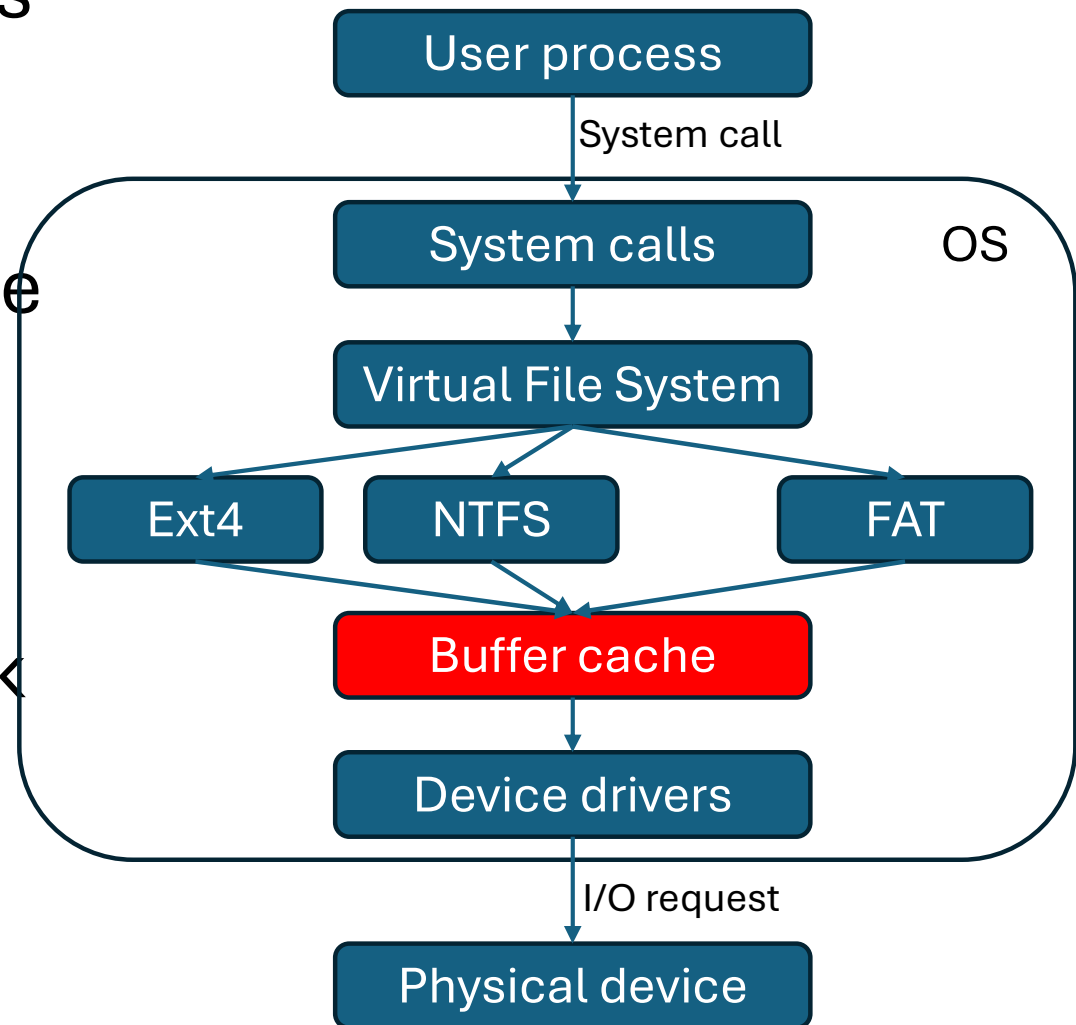
How mount works

- When the VFS traverses a path to access a file, it checks if a given directory in the path is either a mounting point or not
- In case it is, the VFS updates its internal pointer to struct `file_operations` (explained later) to point to that struct of the physical file system driver

```
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1361676k,nr_inodes=340419,mode=755,inode64)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,nodev,noexec,relatime,size=395620k,mode=755,inode64)
/dev/sda2 on / type ext4 (rw,relatime)
```

Buffer cache

- Physical file system drivers transform file accesses to block accesses
- Notice that hard drivers are accessed in block granularity (block = N sectors)
- The buffer cache is a block cache that access device drivers to move blocks from/to hard drives



Buffer cache

- The buffer cache is an optimization that implements the file system to cache blocks of disk
- The physical file system drivers translate file accesses to block accesses
- Those block accesses are passed to the buffer cache to check if they can be served from memory
- If the block is not in memory, the buffer cache uses the device drivers to access physical devices (similar idea of accessing physical file systems from VFS)

Operations of the buffer cache

- The buffer cache provides a standard interface for physical file system drivers
- **Access and Reading**
 - **sb_bread(struct super_block *sb, sector_t block) / __bread(struct block_device *bdev, sector_t block, unsigned size):**

Looks up the block in the buffer cache. If it does not exist, it performs synchronous I/O to read it from the block device. Returns a locked/referenced struct buffer_head*.
 - **sb_getblk(struct super_block *sb, sector_t block) / __getblk(...):**

Fetches the cached block structure or allocates an empty buffer if it does not exist, without initiating a physical read from the disk.
- **Release and Modification**
 - **brelse(struct buffer_head *bh):**

Decrements the buffer's reference count and releases it so it can be recycled by the kernel.
 - **mark_buffer_dirty(struct buffer_head *bh):**

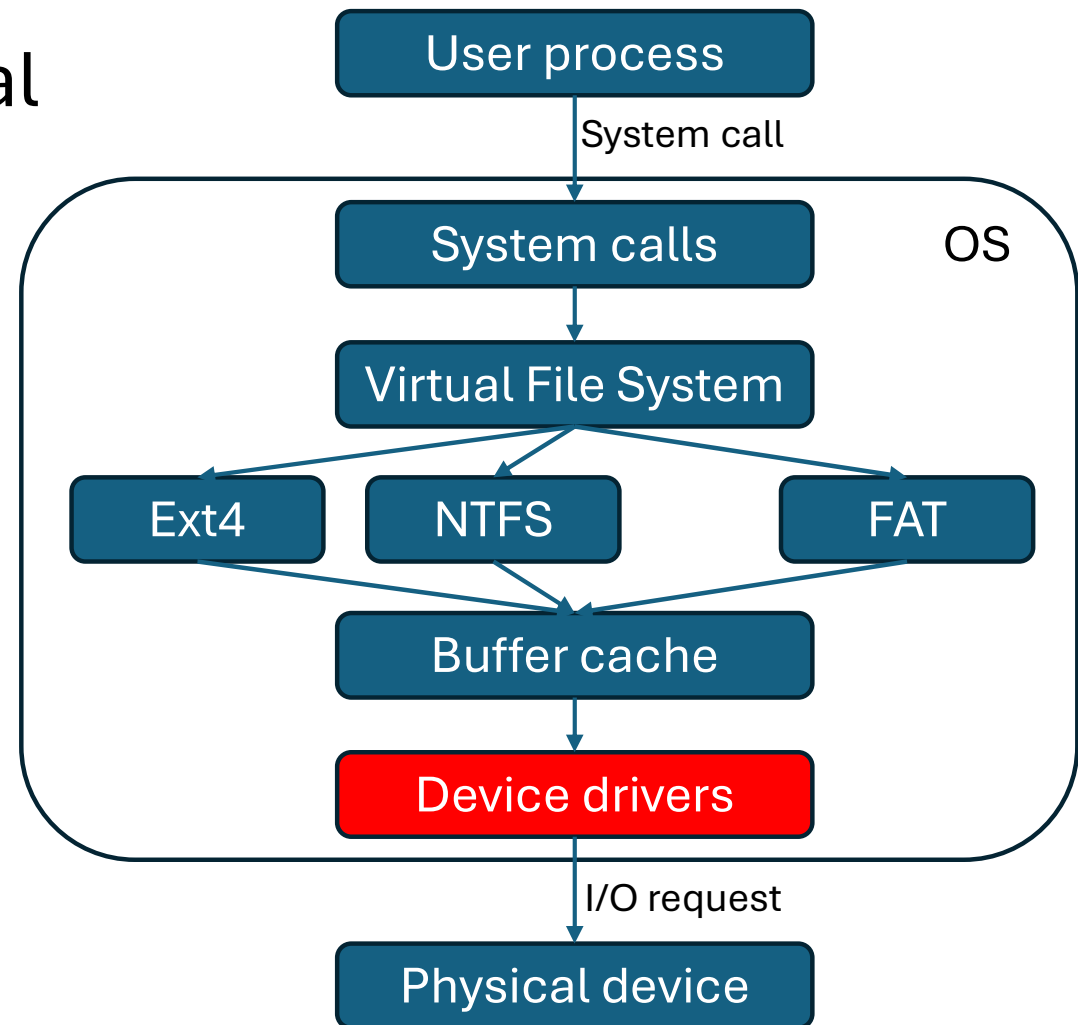
Marks the buffer as "dirty," notifying the cache that the memory content has changed and must eventually be written to disk.
- **Writing and Asynchronous I/O**
 - **sync_dirty_buffer(struct buffer_head *bh):**

Forces a synchronous write to disk for a buffer marked as dirty.
 - **submit_bh(int op, int op_flags, struct buffer_head *bh):**

Converts the buffer_head request into a bio object and submits it to the block device's I/O queue asynchronously.

Device drivers

- Device drivers are drivers that finally access physical devices
- They translate the block transfer operations from the buffer cache to real physical accesses



Device drivers

- A device driver is a specialized piece of software that acts as a translator between an operating system (OS) and a hardware component.
- Hardware manufacturers build their devices using proprietary electronics, registers, and communication protocols. Rather than forcing the OS (like Linux, Windows, or macOS) to know the internal code for thousands of different devices, the OS provides a standard interface. The device driver translates standard OS commands into the specific instructions the physical hardware understands.

Device drivers

- Device drivers always run in privileged level
- But its functionalities must be accessed by the user
- The way to access a device driver by the user is accessing the associated logical device

Structure of a device driver

- A valid device driver must follow the structure defined by the OS (function definitions, macros, steps)
- A typical structure is:
 1. Macro definitions
 2. Operational code
 3. Definition of struct file_operations
 4. Driver registration
 5. Driver initialization
- This steps will be explained later

How a device driver is loaded

- The OS needs a device driver to interact with a hardware device
- For this, the OS, in boot time, scans all the buses in the motherboard to recognize the plugged, if any, hardware devices
- For every hardware device, the OS obtains two numbers
 - Vendor ID (VID): identifier of the vendor of the device
 - Product ID (PID): identifier of the product of the vendor

VID and PID

- The Vendor ID (VID) is a world-wide unique identifier a hardware manufacturer has
 - For example: Microsoft's VID is 0x1414 and Sony's is 0x104D
- Those VIDs are assigned by independent standards organizations, trade associations or governing bodies
- The Product ID (PID) is a unique number that identifies a product among all the products a vendor manufactures
- The vendor is responsible to assign the PID

How a device driver is loaded (cont.)

- The OS, with the type of bus, the VID and the PID creates a hardware string known as MODALIAS:
 - A MODALIAS string for a USB mouse could be:

`usb:v045Ep0040d0000dc00dsc00dp00ic03isc01ip02in00`

- This MODALIAS provides all the information needed to the OS to look for a device driver able to work with that hardware device
- So, as next step, the OS looks for the corresponding device driver

Looking for a device driver

- The OS maintains a list of pairs <MODALIAS, device driver> in a file called /lib/modules/\$(uname -r)/modules.alias
- So, this file tells the OS which device driver must be loaded to work with a given hardware device
- Fragment of a modules.alias file:

```
alias mcb:16z127 gpio_menz127
alias pci:v000010DBd0000802Esv*sd*bc*sc*i* gpio_ml_ioh
alias usb:v0403p6988d*dc*dsc*dp*ic*isc*ip*in* gpio_mpsse
alias usb:v0C52pA064d*dc*dsc*dp*ic*isc*ip*in* gpio_mpsse
```

bus VID PID

device driver

Loading the device driver

- Once the device driver is found, the OS loads it in memory
- After that, the OS calls the initialization routine of the device driver
- This initialization routine:
 - First, registers the driver
 - Second, initializes the internal structures of the driver and the associated hardware (if any)

Registering the device driver

- The device driver registers itself in the OS
- This means that:
 - A Major is associated to the driver
 - The driver publishes which operations provides using the struct `file_operations`
- The kernel, then, stores the pair `<Major, struct file_operations*>` in a structure called `struct cdev`
- So, the kernel maintains a list of `struct cdev`
- There are as many `struct cdev` as loaded device drivers

Major

- The Major is an integer identifier (12 bits) assigned to a device driver that uniquely identifies it in the system
- Normally, the Major identifies a kind of physical or virtual device
- The Major can be assigned:
 - Statically: the device driver uses a particular Major
 - Dynamically: the device driver asks the OS for a free Major
- Some static Majors belong to a well-known type of devices.
- For example, SCSI disks use 8 as Major. So, a device driver able to work with SCSI disks will use 8 as Major.

struct file_operations

- The struct file_operations is a structure defined by the OS that stores the pointers to all possible operations a driver can implement

```
struct file_operations {
    struct module *owner;

    /* Position and IO */
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);

    ...
};
```

- The driver must fill this structure before registering

Filling the struct file_operations

- If a driver implements a particular operation, it must set the corresponding pointer of the struct file_operations to point to the code of that operation
- If the driver does not implement a given operation, it must set the corresponding pointer to NULL

```
#include <linux/fs.h>
#include <linux/module.h>

static int my_open(struct inode *inode, struct file *file) {
    // Initialization code
    return 0;
}

static ssize_t my_read(struct file *file, char __user *buf, size_t len, loff_t *off) {
    // Read logic
    return 0; // Number of bytes read
}

static const struct file_operations my_fops = {
    .owner    = THIS_MODULE,
    .open     = my_open,
    .read     = my_read,
    // .write, .release etc. set to NULL implicitly
};
```

Data transfer bandwidth

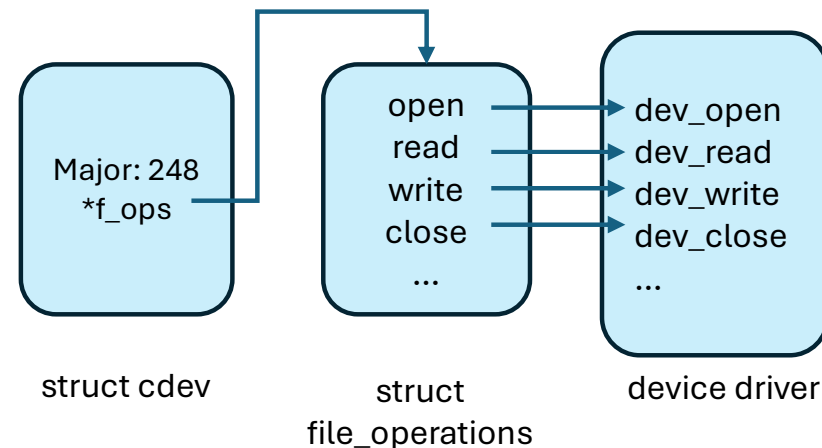
- The last thing to decide before registering the driver is the data bandwidth
- A device driver is a character device if it is only able to transfer one byte at a time.
 - For example, the device driver of the keyboard is a character device because it can only transfer one key (byte) at a time
- A device driver is a block device if it can transfer more than one byte at a time
 - For example, the device driver of the graphical card is a block device because it can transfer a whole texture or a vertex buffer with one operation

Registering the device

- To register the device driver, the function `register_chrdev` (for character devices) or `register_blkdev` (for block devices) are used:
- This function associates a Major number with a struct `file_operations` and stores this association in a struct `cdev`

```
static const struct file_operations fops =  
{ .owner = THIS_MODULE, .open = dev_open, ...};  
  
major_number = register_chrdev(0, DEVICE_NAME, &fops);
```

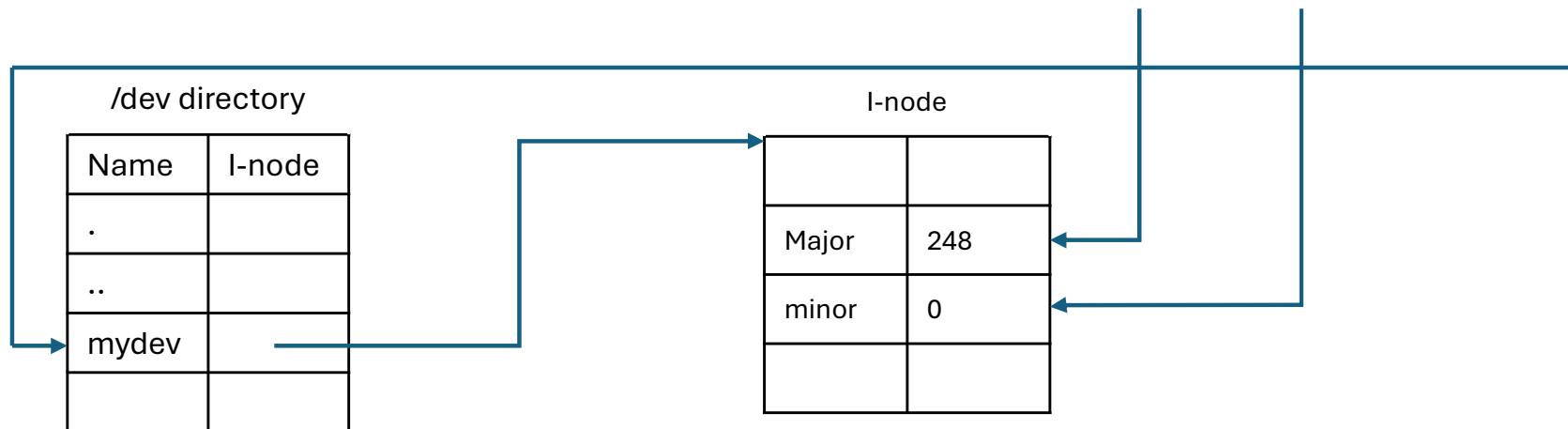
0 to ask for a free Major



Associating a logical device

- After loading and registering the device driver, an associated logical device must be created
- The logical device is a file with an I-node which Major is the same as that of the device driver
- For that, either the device driver or the OS calls `device_create`

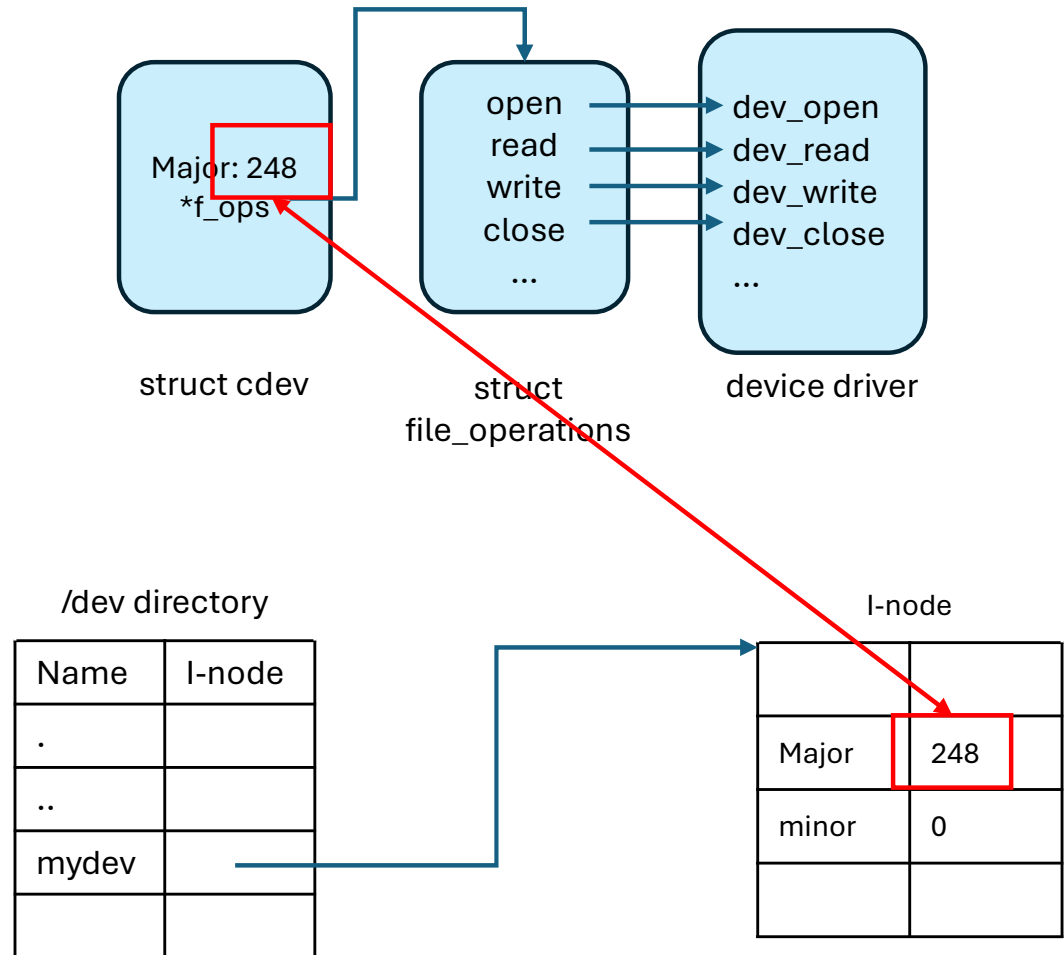
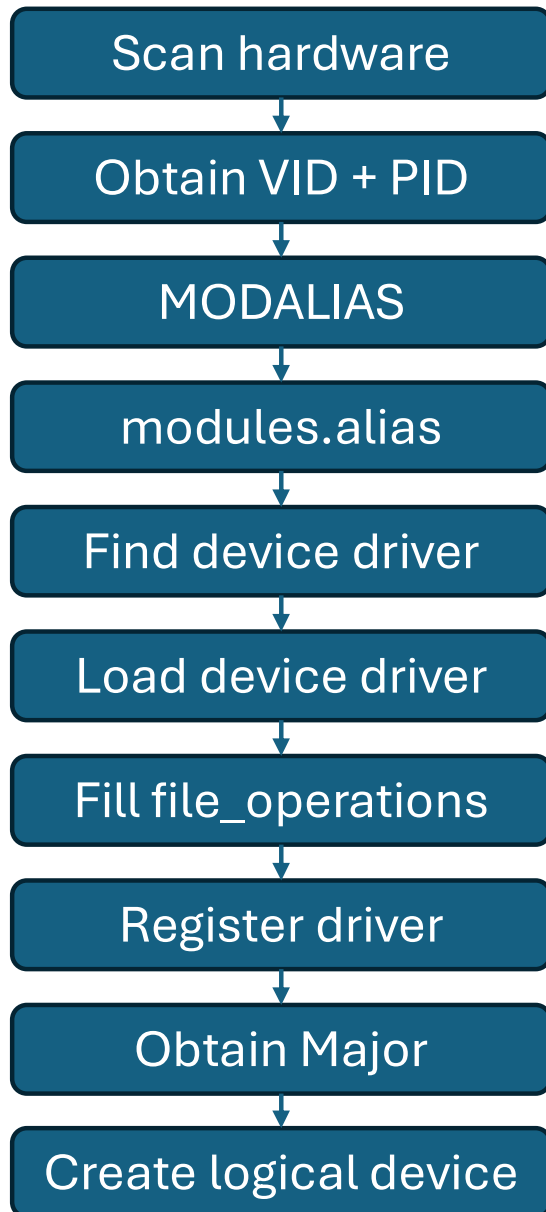
```
device_create(NULL, NULL, MKDEV(major_number, 0), NULL, "mydev");
```



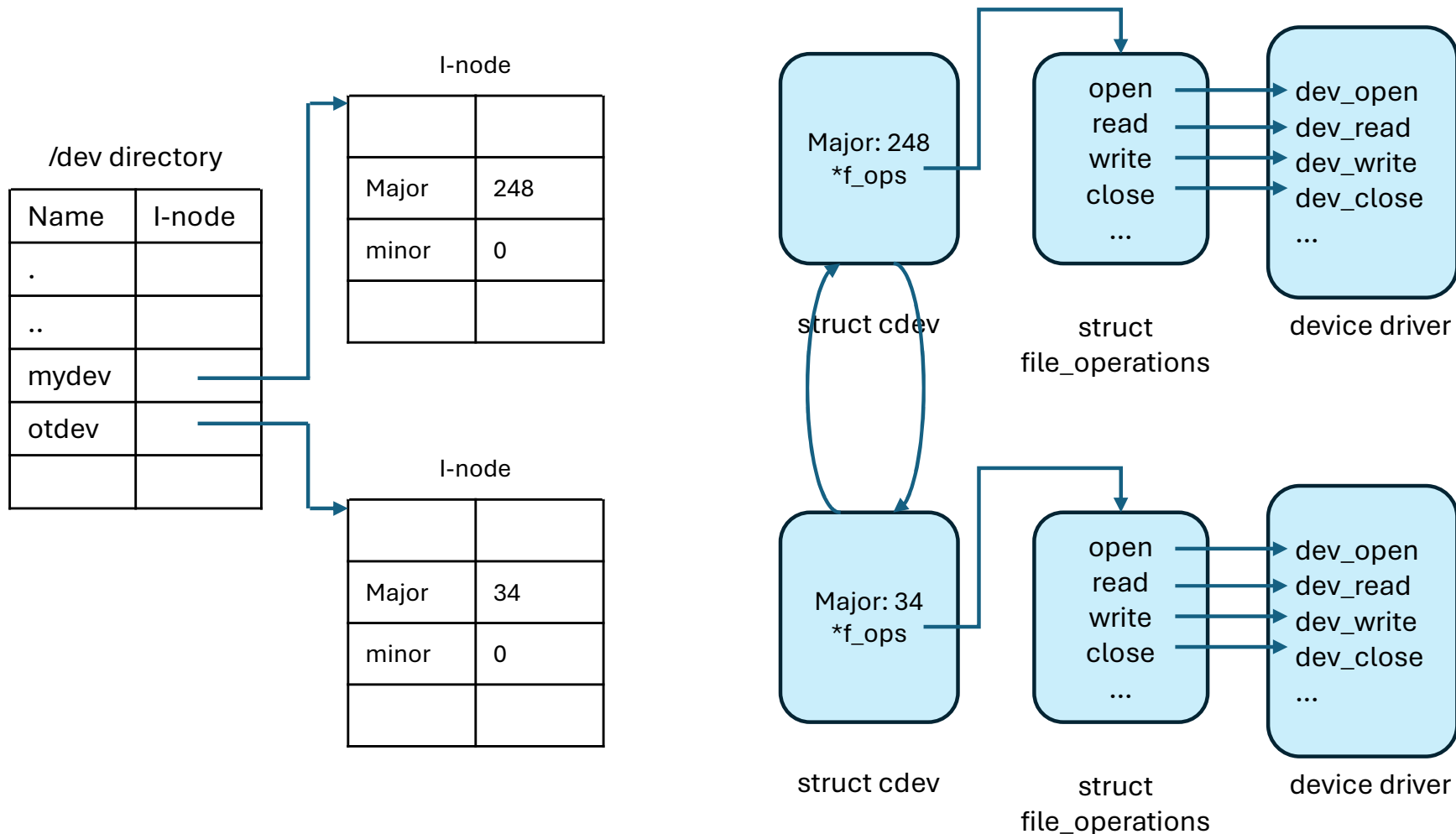
Minor

- The minor is an integer identifier (20 bits) which value is only meaningful for the device driver
- Sometimes it is used to identify several buses of the same type. For example, when there are more than one SATA hard drive plugged
- Some device drivers do not use it at all

Device driver registration summary



Scheme with several device drivers



Dynamically loading a device driver

- A device driver can be loaded after booting the OS
- The procedure is the same but without the hardware scanning and the access to the modules.alias file
- The user can either use insmod or modprobe to dynamically load a device driver
- The user must create manually the associated logical device with mknod or ensure that the device driver creates its own logical device (with device_create)

Opening a logical device

- Remember that to open a logical device, the system call is:

```
int open(char *name, int mode [, int flags]);
```

- Where *name* is the name of the logical device (plus full path) to open
- Flags and mode will be disregarded by now
- Open returns a file descriptor associated to a dynamic instance of the logical device
- If you are not familiar with open, review the OS course

Opening a logical device

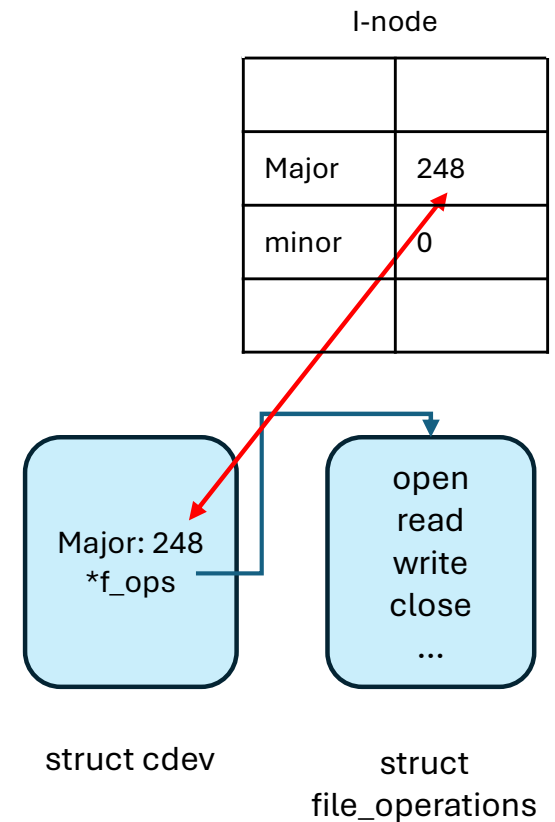
- First, open traverses the path until reaching the Inode of the logical device and stores one copy of that Inode in memory

I-node

Major	248
minor	0

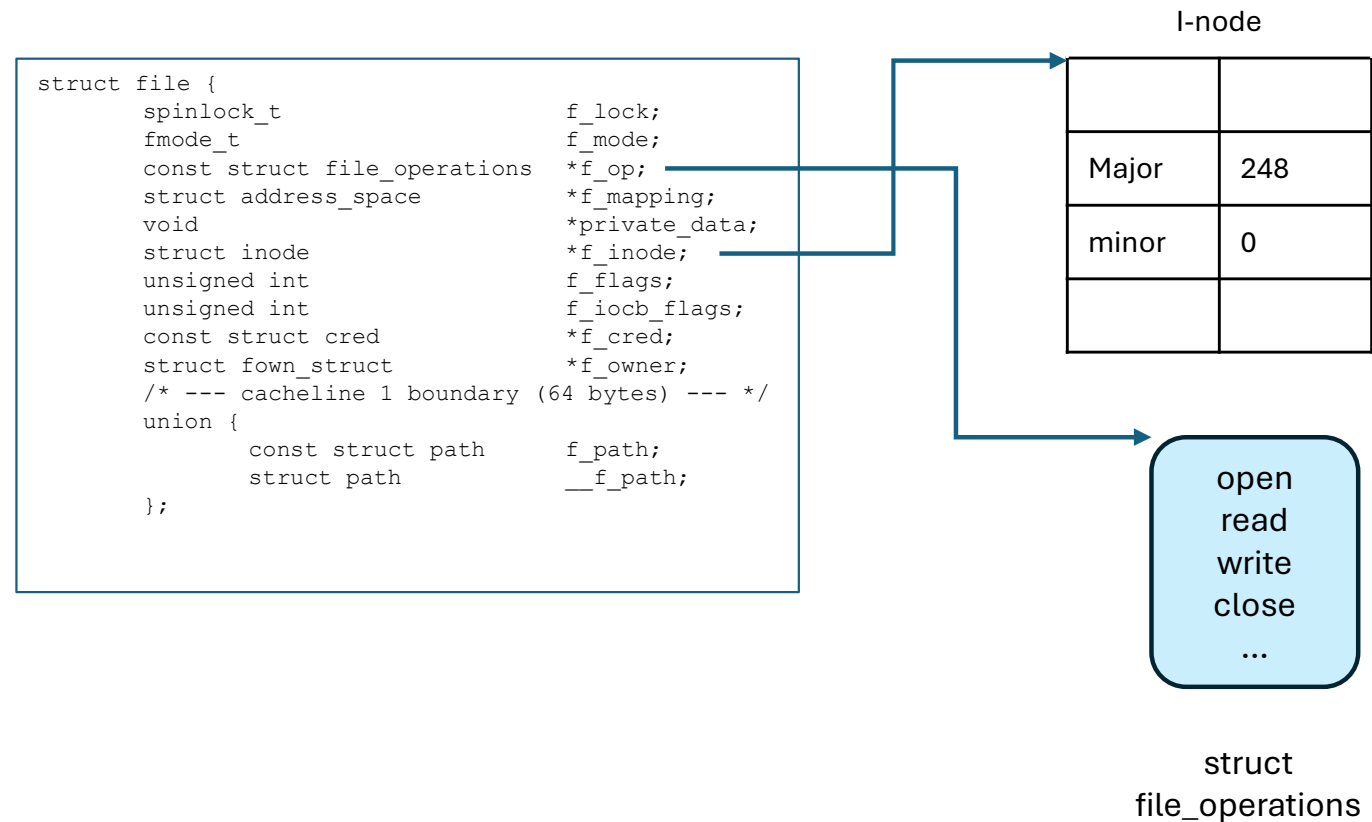
Opening a logical device

- After that, open looks for the Major of the Inode in the struct cdev list and takes the pointer to the struct file_operations.



Opening a logical device

- Now, open allocates a struct file in which it stores, among others, the pointer to the Inode and the pointer to the struct file_operations

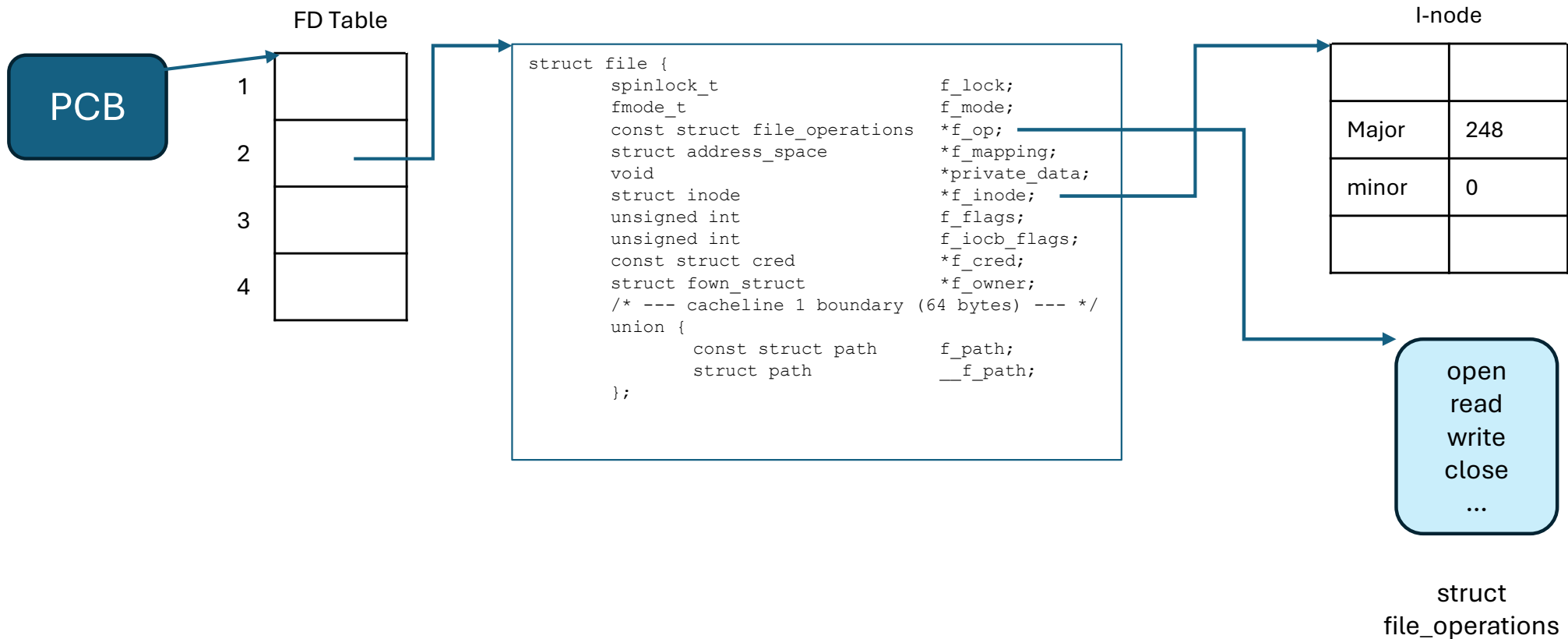


struct file

- The struct file stores all the information about an opened file
- It's a dynamically allocated structure that substitutes the Opened File Table and the Inode Cache
- It is defined in linux/fs.h in the Linux Kernel
- This structure stores the relationship between a logical device and the associated device driver

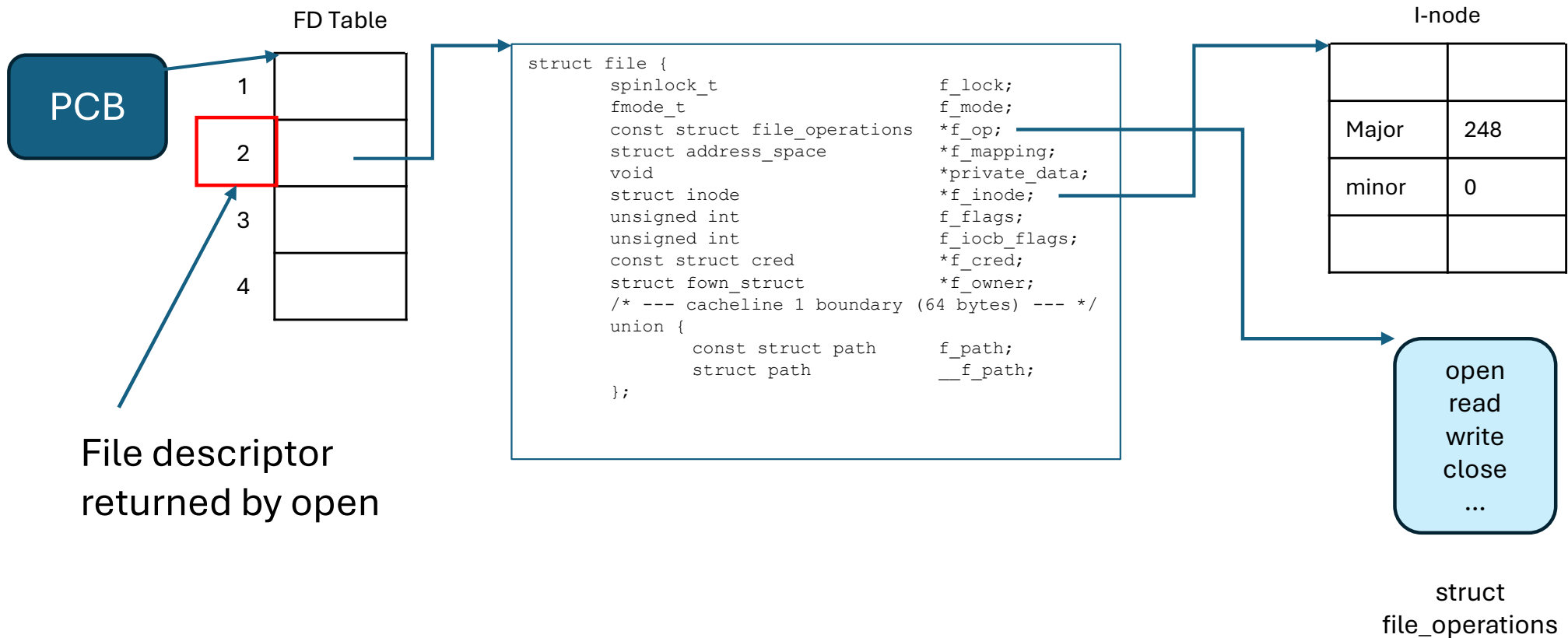
Opening a logical device

- Finally, a pointer to the new struct file is stored in one free entry of the File Descriptor Table



Opening a logical device

- Finally, a pointer to the new struct file is stored in one free entry of the File Descriptor Table



File Descriptor Table

- The File Descriptor Table is local to a process. This means that every process has its own FD Table.
- It stores a pointer to the allocated struct file
- It must be copied when a process is forked
- The file descriptor returned in open is the number of newly allocated entry

Reading from a file descriptor

- Remember that to read data from a file descriptor, the system call is:

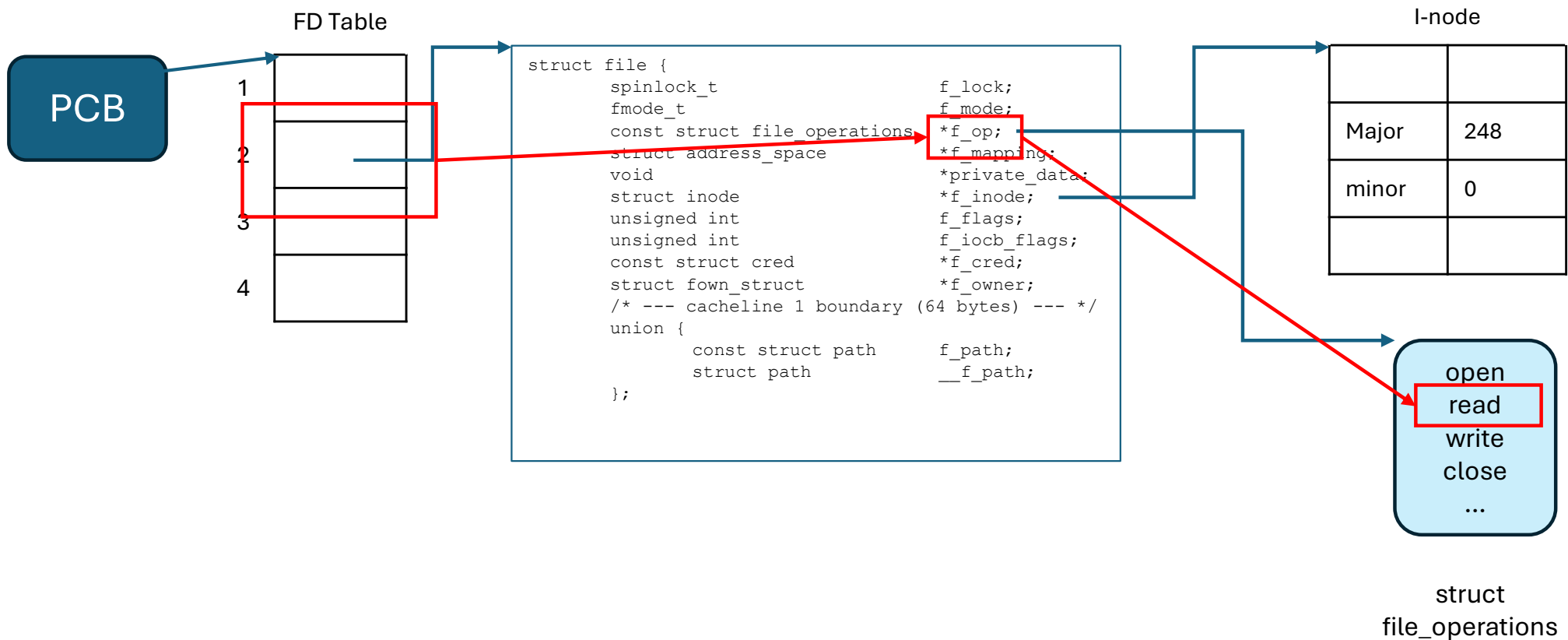
```
int read(int fd, void *buffer, size_t size);
```

- It reads a maximum of size bytes in buffer from fd
- Returns the number of effectively read bytes

Reading from a file descriptor

- Reading from a file descriptor is always the same:

`FDTable[fd]->f_ops->write`

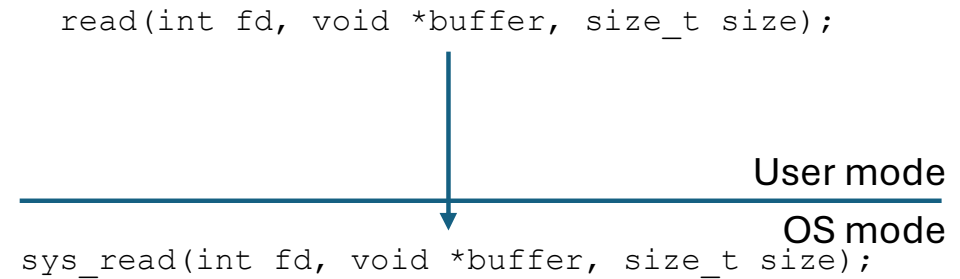


Reading from a file descriptor

- `FDTable[fd]->f_ops->read` is always the same to read from any physical device
- Remember that struct `file_operations` stores pointers to the specific operations in a device driver
- So, since in the struct `file` the pointer of the struct `file_operations` is stored for every logical device, the associated operation of the device driver is executed
- Struct `file_operations` implements polymorphism and ensures that the OS will be always independent of the physical device
- In fact, the relationship is between logical device and device driver

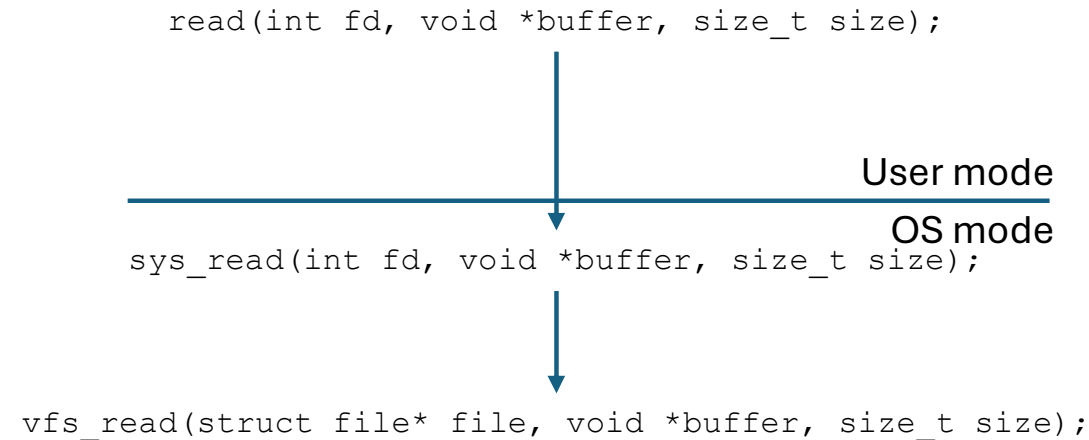
The whole picture

- The user calls the read system call to read data from a file descriptor
- Sys_read is invoked from the system call handler



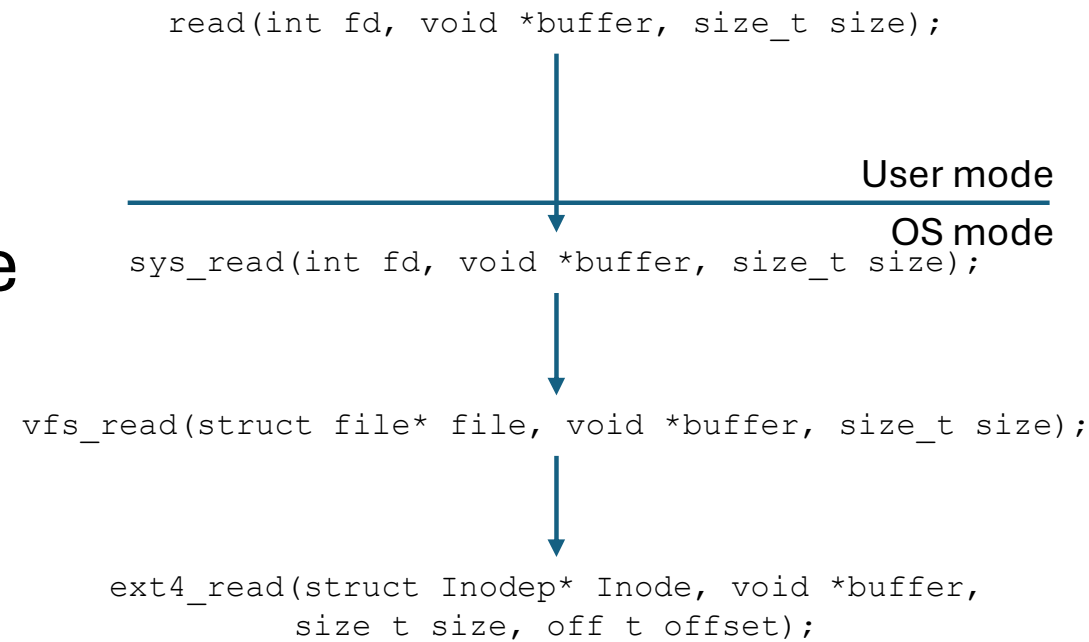
The whole picture

- The VFS is invoked from `sys_read` once the associated struct file is retrieved from the file descriptor



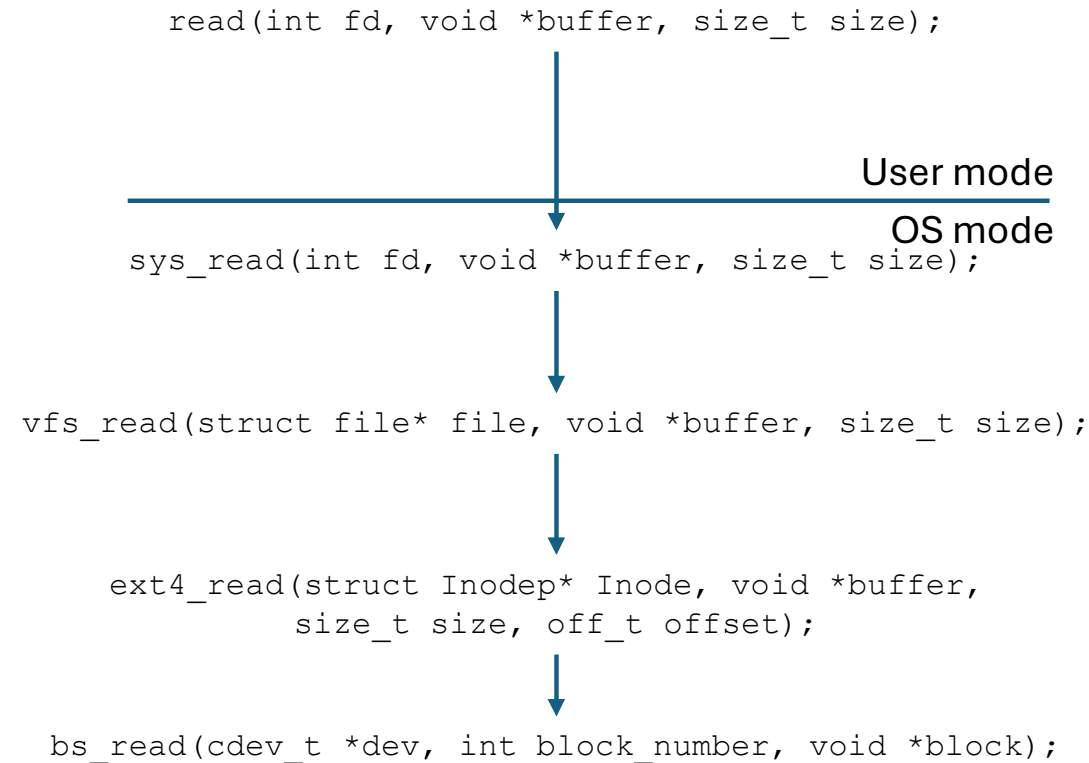
The whole picture

- The virtual file system accesses the struct `file_operations` in the struct `file` and executes the associated operation of the physical file system driver
- Also, the VFS obtains the offset from the struct `file`



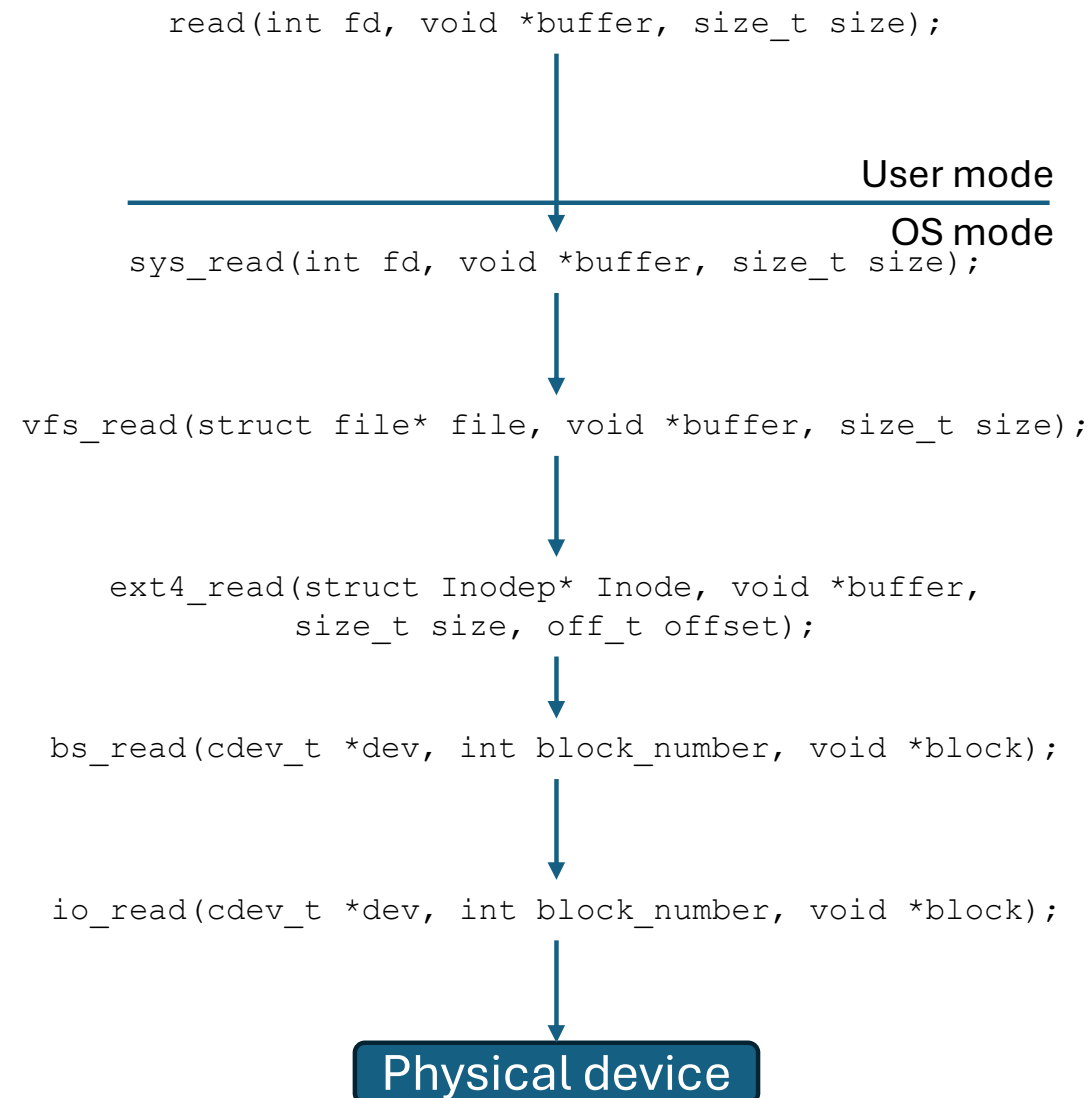
The whole picture

- The driver of the physical file system translates the offset to a given block of disks and calls the buffer cache to obtain the information of that block



The whole picture

- If the desired block is not found in the Buffer Cache, an I/O operation is started from the device driver
- That device driver is the one associated to the physical device



Extending the VFS to non-Linux file systems

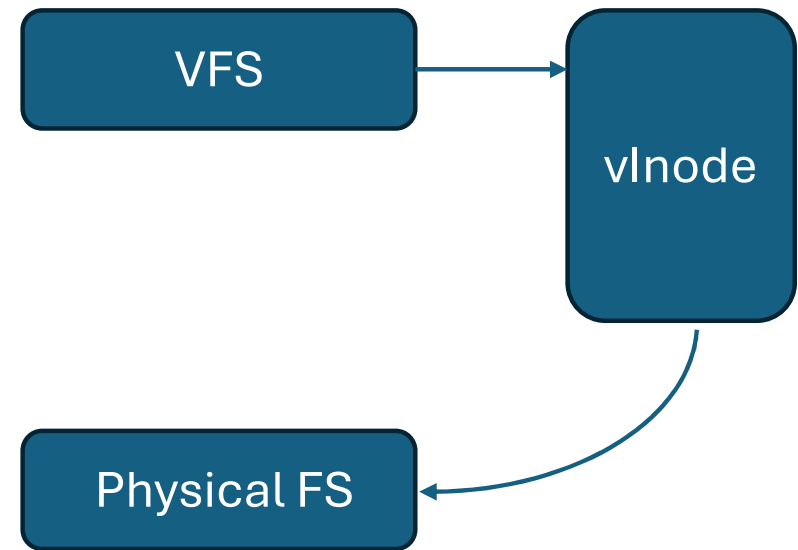
- All these flows work when the underlying physical file systems are Linux compatible (ext2, ext3, ext4, raiser, ...)
- **But not all physical file systems use Inodes**
- For example, NTFS has a similar structure called Record, but FAT32 does not have any Inode-like structure
- Solution: **Inode virtualization**

Virtual Inode

- A virtual Inode (vlnode) is a structure that stores the information required by the VFS to work with a logical device plus an opaque pointer to the physical file system driver private data about that file
- This private data can be an Inode or not depending on the physical file system characteristics

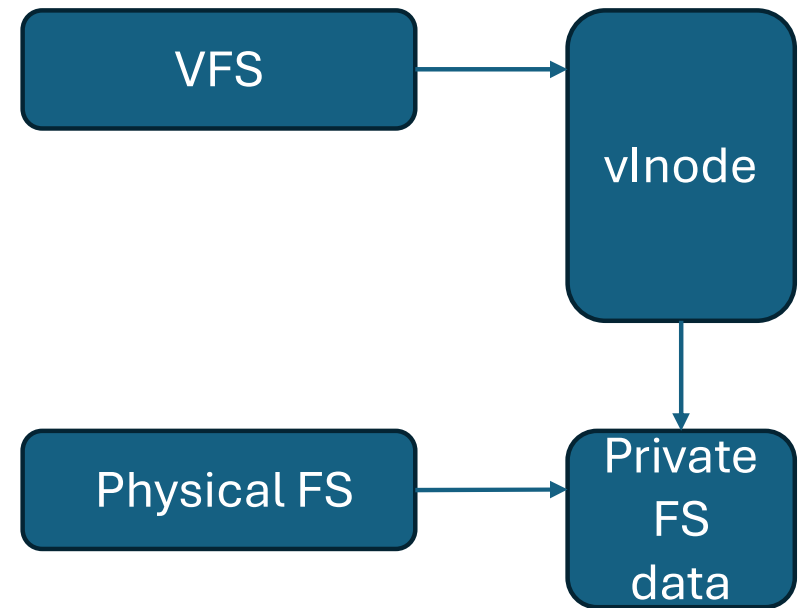
Virtual Inode creation

- When the user wants to open a file, the VFS allocates a vnode
- This vnode is passed to the physical FS driver



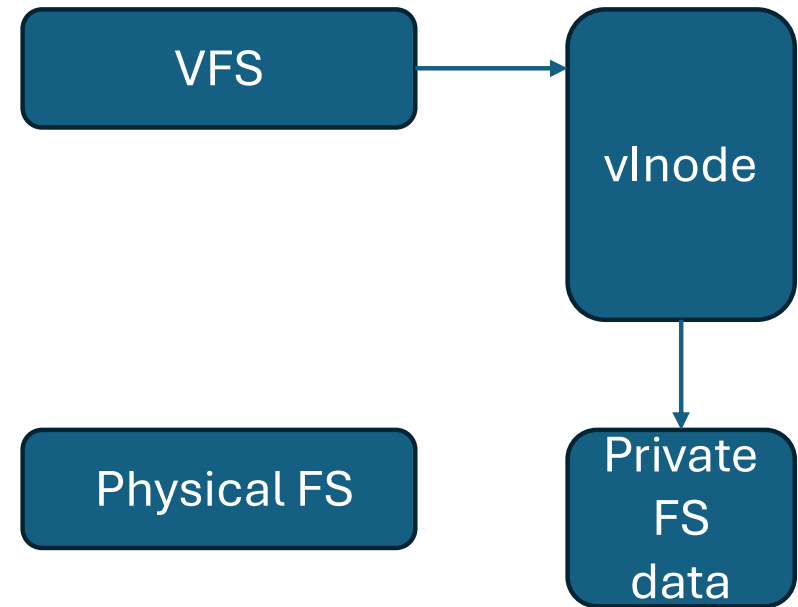
Virtual Inode creation

- The physical FS driver, after retrieving the information of that file, allocates a private structure to store that information
- A reference to that private data is stored in the vnode
- In addition, the physical FS driver fills all the information of the vnode



Virtual Inode creation

- Finally, the VFS updates the remaining file system structures as before
- After this, the VFS will pass the vnode to the physical FS to perform operations
- The physical FS driver will retrieve its private data from the structure associated to the vnode.



Virtualized File Systems

Virtualized File Systems

- Not all file system drivers must be associated to a physical disk
- In fact, VFS allows the use of virtualized file systems
- A virtualized file system is a file system whose data is not stored in a physical disk but dynamically created by a file system driver
- So, the associated device driver never accesses a physical device
- Virtualized file systems are also mounted under a folder
- Examples are: procfs, unionFS, pipes, ...

Procfs

- **procfs** (Process Filesystem) is a virtual pseudo-filesystem—typically mounted at /proc on Linux and Unix-like systems—that acts as an interface to internal kernel data structures.
- Instead of occupying space on a disk or storage device, files in /proc are generated dynamically in RAM by the kernel when you attempt to read or access them.

```
alex@Hotel:~$ ls /proc
1      170    195    234    257    2834   3303   54     83
10     171    1950   235    2570   2835   3372   55     84
100    172    1966   236    258    2836   3380   56     845
101    173    1978   237    259    2838   3397   564    846
102    174    198    2370   2594   2840   34     565    847
103    175    2      2378   2595   2843   3415   57     85
104    1750   20     238    26     2844   3422   572    86
105    1757   200    2382   260    2845   3423   573    87
106    176    201    239    2609   2848   3433   575    88
108    177    203    24     261    2875   3439   58     89
109    1770   204    240    262    2877   3493   59     9
11     1772   205    241    2621   29     35     6     90
110    178    207    242    2629   2995   3500   60     91
112    1781   2075   243    263    3      3510   600    92
1156   179    209    2438   264    30     36     608    93
1157   1791   2092   244    265    3043   366    61     94
1164   1798   21     245    2655   3082   37     62     96
```

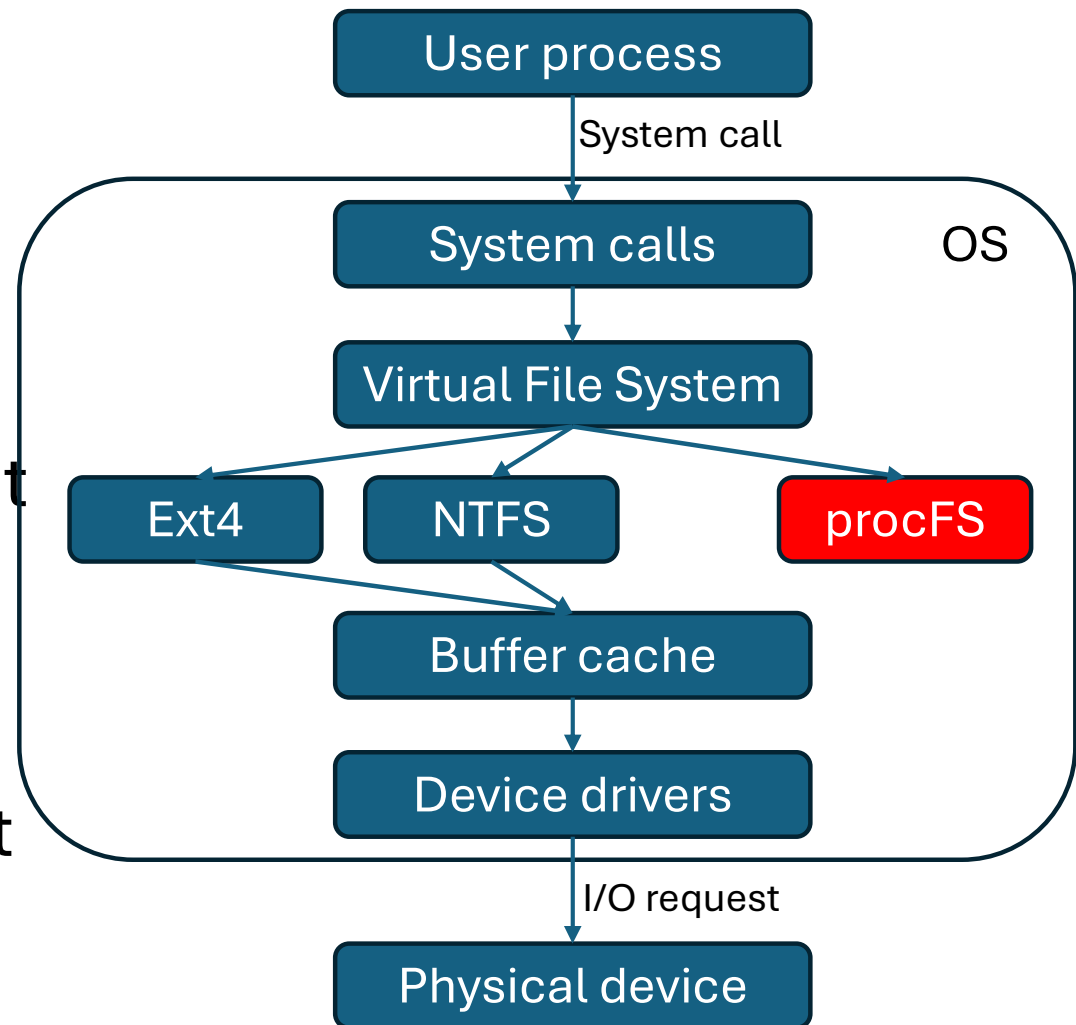
```
irq
kallsyms
kcore
keys
key-users
kmsg
kpagecgroup
kpagecount
kpageflags
latency_stats
loadavg
locks
mdstat
meminfo
misc
modules
mounts
```

Key features of ProcFS

- **Dynamic Virtual Files:** Almost all files in /proc show up as 0 bytes in size on disk. When a process reads a file in /proc, the kernel dynamically formats and returns the requested data instantly.
- **System & Hardware Status:** Exposes system-wide runtime statistics, hardware data, and kernel parameters.
- **Process Tracking:** Provides a dedicated sub-directory for every active process running on the system, named after its Process ID (/proc/[PID]).
- **Kernel Configuration:** While mostly read-only, certain files (such as those under /proc/sys/) allow administrators to modify runtime kernel behavior on the fly by writing directly to them.

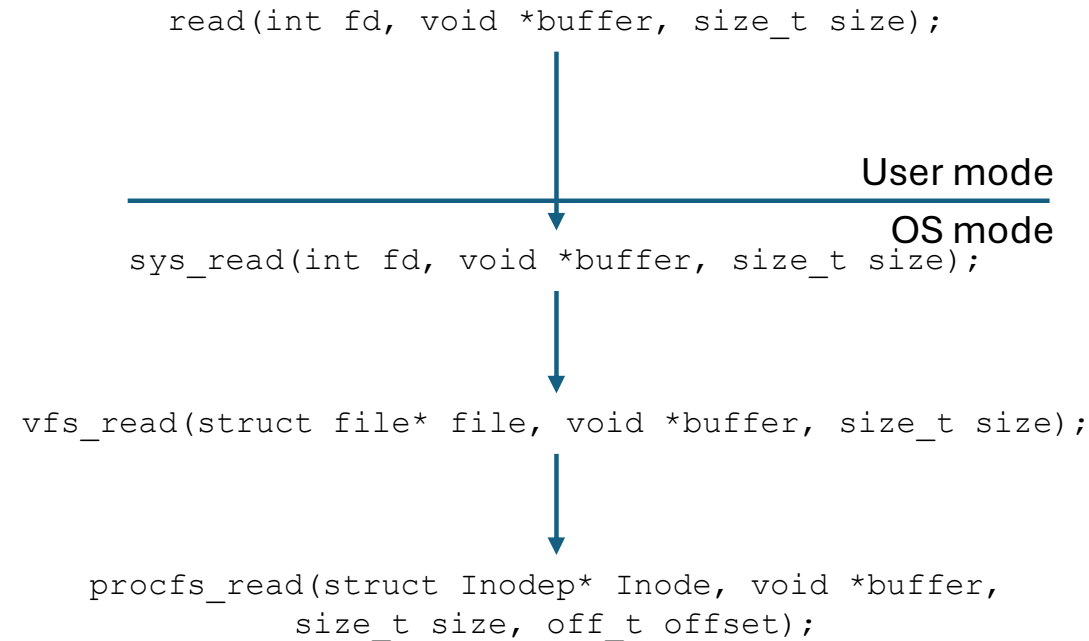
ProcFS device driver

- The ProcFS is implemented as a device driver
- To access it, a mount point /proc is created
- When an operation is performed under the path /proc, the `procfs_file_operations` struct is used to call the device driver
- Notice that procFS does not access the Buffer Cache since the device driver is not associated with any disk



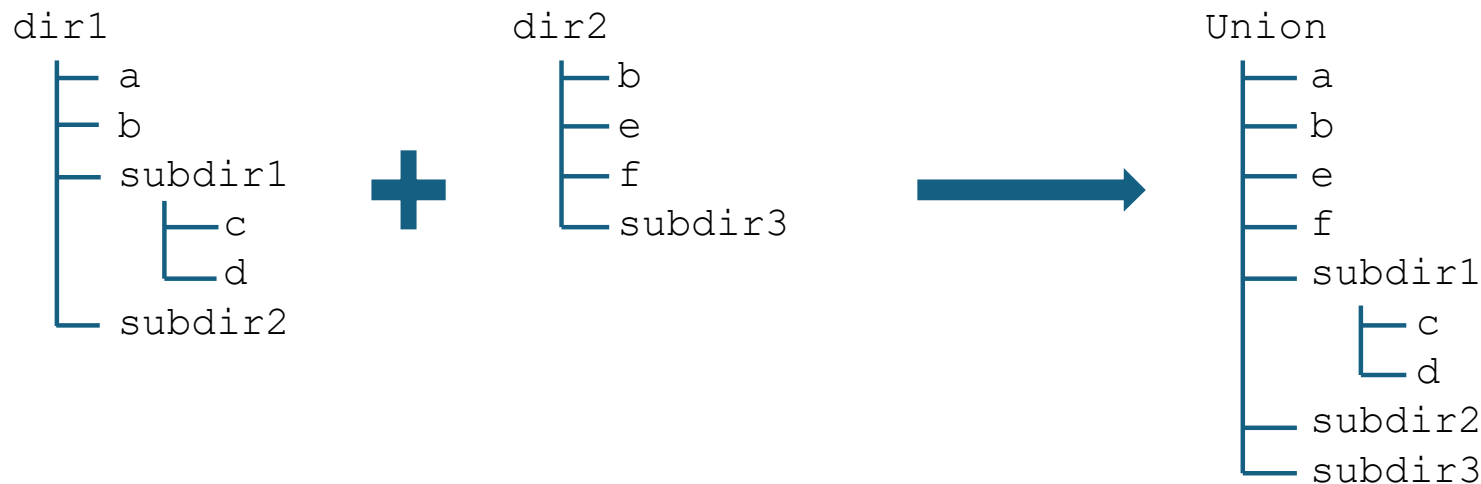
ProcFS read operation

- VFS accesses procFS device driver using the `procfs_file_operations` struct found when the `/proc` mount point is detected
- `Procfs_read` does not access the disk but reads the kernel structures to return data



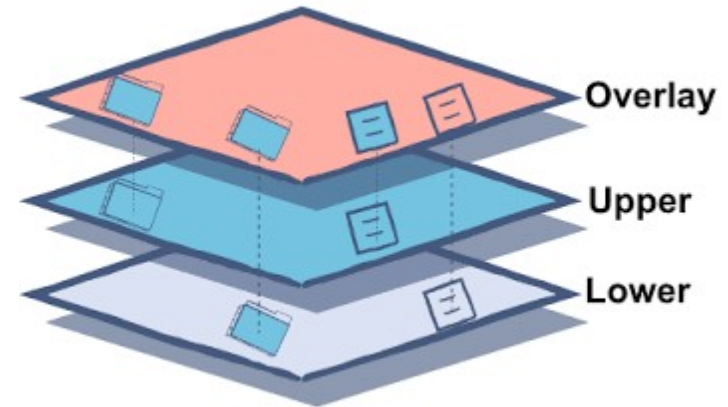
OverlayFS

- **OverlayFS** is a Linux union filesystem service that combines two or more directories into a single merged view. It presents multiple underlying filesystem trees as a unified hierarchy, allowing files from one layer to visually override files from another without modifying the source layers.



Core components & directories in OverlayFS

- When mounting an OverlayFS volume, four distinct directories are involved:
- **lowerdir**: The base layer(s). This is read-only. Multiple lower directories can be stacked together separated by colons.
- **upperdir**: The writable layer. Any new files created, modified, or deleted in the merged view are written here.
- **workdir**: An empty internal directory required on the same filesystem as upperdir used by the kernel to prepare file state transitions before they are committed to the upper layer.
- **merged**: The mount point where the unified view of all layers is presented to users and processes.



Key File Operations in OverlayFS

- **Read:** If a file exists in both the upperdir and lowerdir, the version in upperdir takes precedence. If it exists only in lowerdir, it is read directly from there.
- **Write / Modify (Copy-Up):** When a read-only file from lowerdir is modified, OverlayFS performs a **copy-up** operation: it copies the file into upperdir first and then applies the edits. The original file in lowerdir remains unchanged.
- **Delete (Whiteouts):** If a file from lowerdir is deleted in the merged view, OverlayFS creates a special **whiteout character device file** in upperdir to hide the file from the merged directory listing.

Example of use of OverlayFS

```
sudo mount -t overlay overlay \  
-o lowerdir=/tmp/overlay_demo/lower,upperdir=/tmp/overlay_demo/upper,workdir=/tmp/overlay_demo/work \  
/tmp/overlay_demo/merged
```

- Creates a mount point called *overlay* with the Major and minor of the overlayFS logical device.
- The `-o` means that parameters for the device driver follow
- In this case: the lower, upper and working directories
- The last parameter `/tmp/overlay_demo/merged` is the mount point. The directory used as merged view.

Common use cases of OverlayFS

- **Container Runtimes (Docker, Podman):** Container engines use OverlayFS (overlay2) to stack read-only image layers together and attach a lightweight, writable layer on top for the running container.
- **Live USB Distributions:** Allows a read-only OS image (like a Live Ubuntu USB) to function as a fully writable system by storing changes in RAM or a separate persistent partition.
- **Embedded Systems:** Protects core system files by keeping the root system in a read-only lowerdir while directing runtime writes (logs, temporary configs) to a volatile upperdir.

OverlayFS device driver

- The overlayFS is implemented as a device driver
- To access it, a mount point, the merged view, must be created
- It can be more than one mounting point of overlayFS
- overlayFS performs calls to VFS to access the lower, upper and working directories

