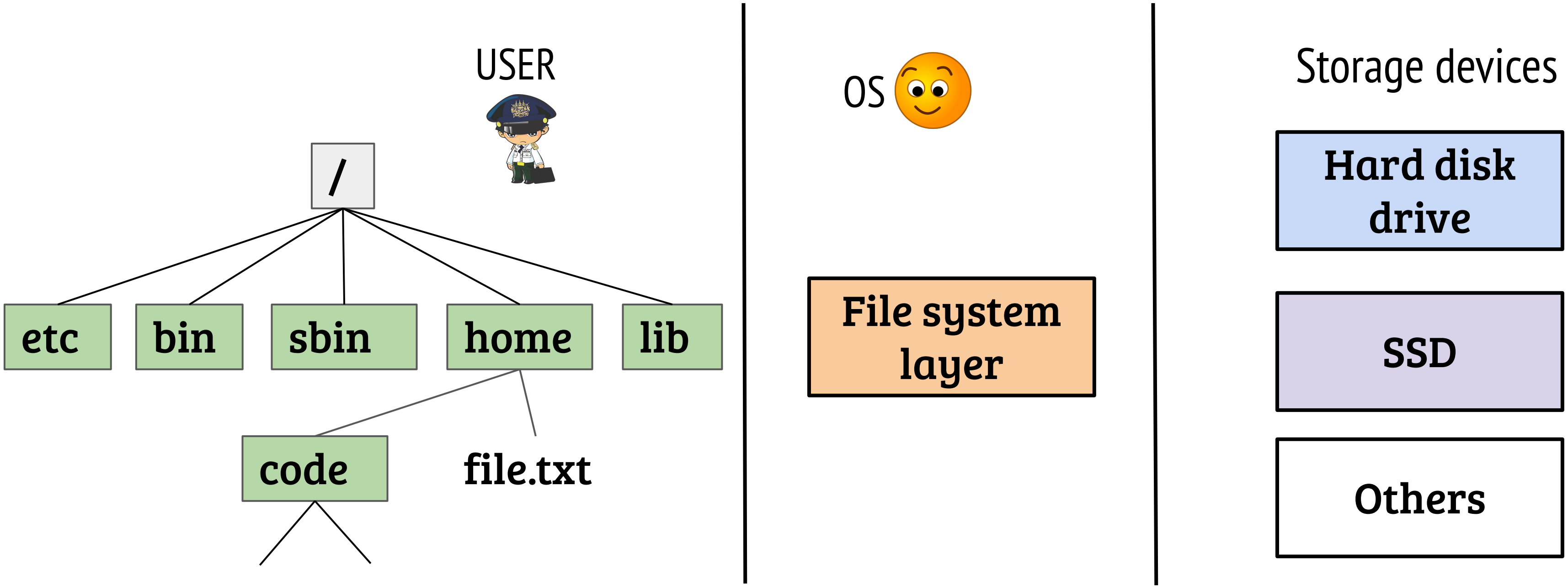
CS614: Linux Kernel Programming

File System Overview

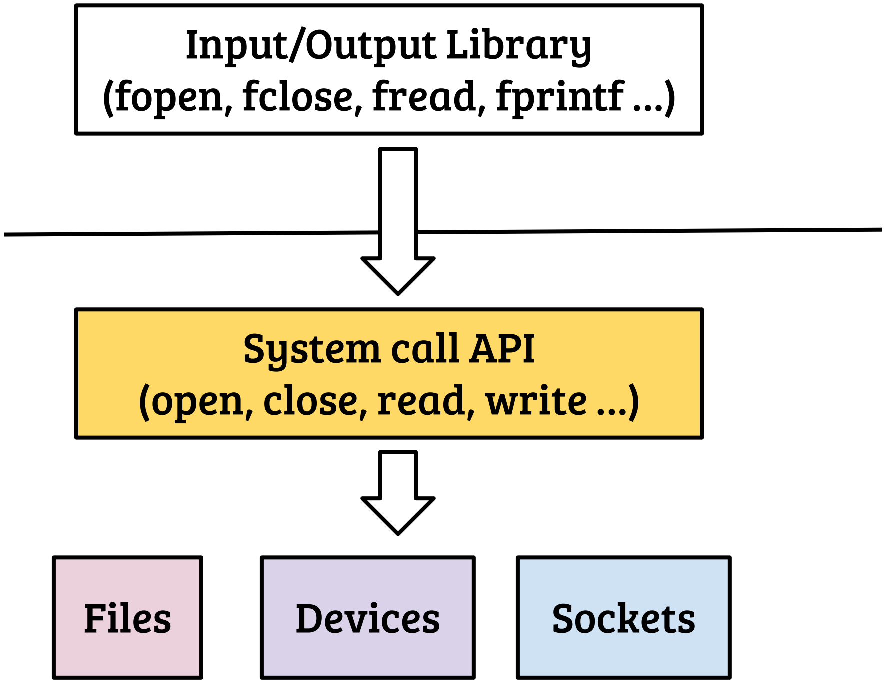
Debadatta Mishra, CSE, IIT Kanpur

Recap: file system



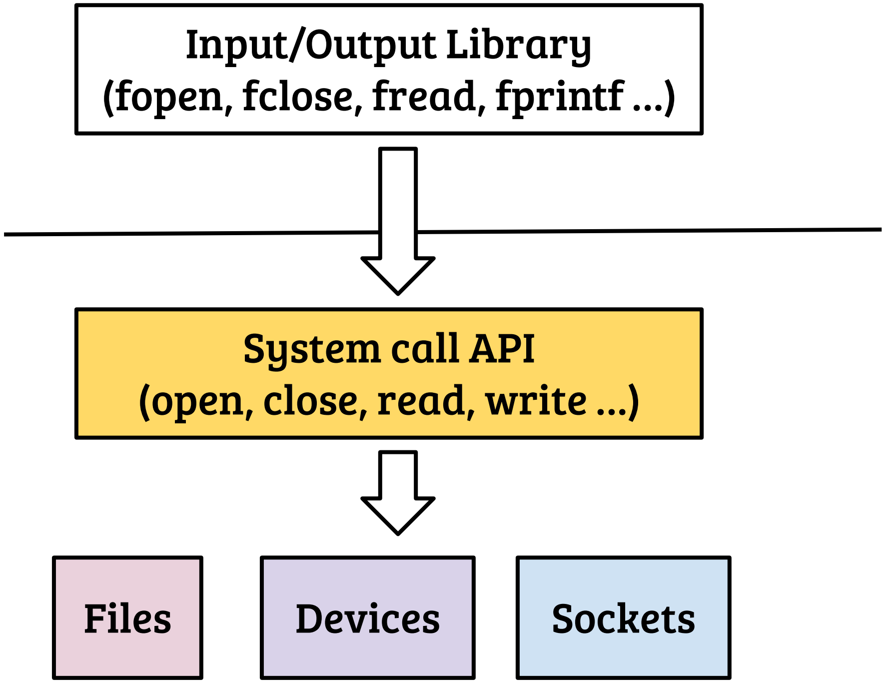
- File system is an important OS subsystem
  - Provides abstractions like files and directories
  - Hides the complexity of underlying storage devices

File system interfacing



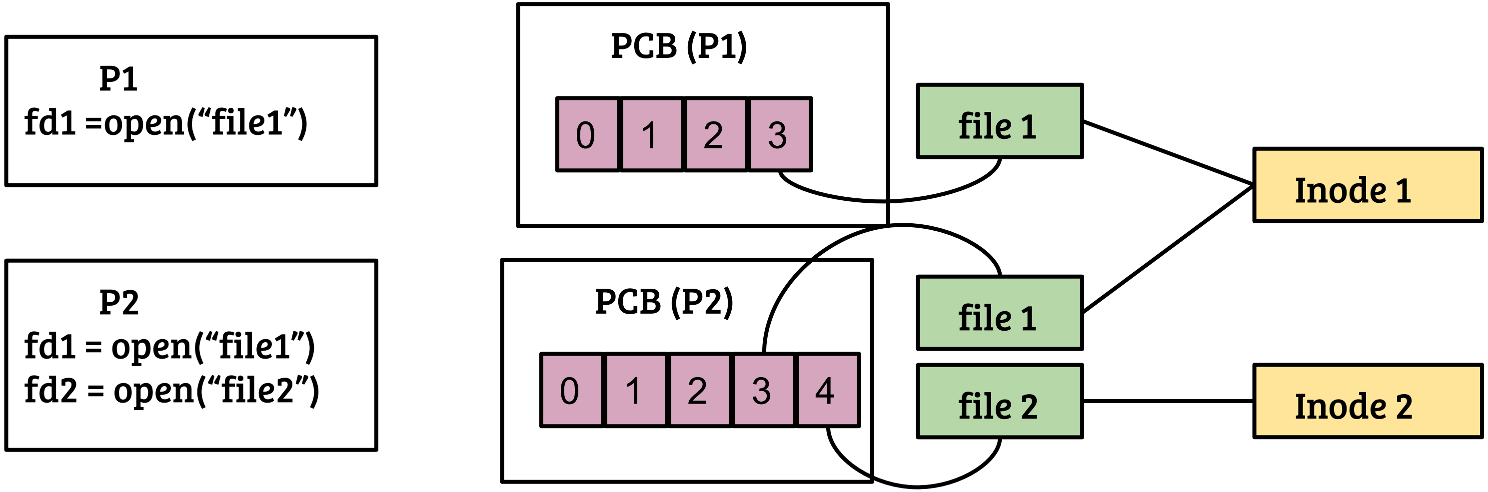
- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.
- Important file related system calls?

File system interfacing



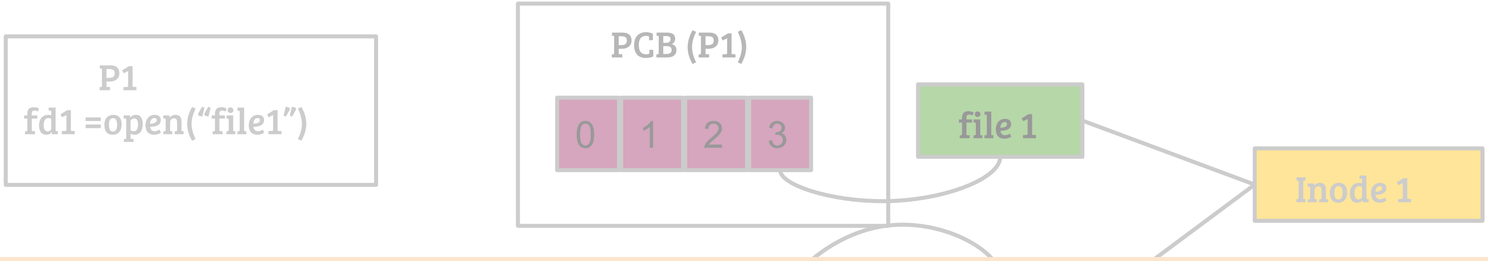
- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.
- Important file related system calls: open, close, read, write, lseek, dup, stat, select, poll ...

Process view of file



- Per-process file descriptor table with pointer to a "file" object
- file object ⟶ inode is many-to-one

Process view of file



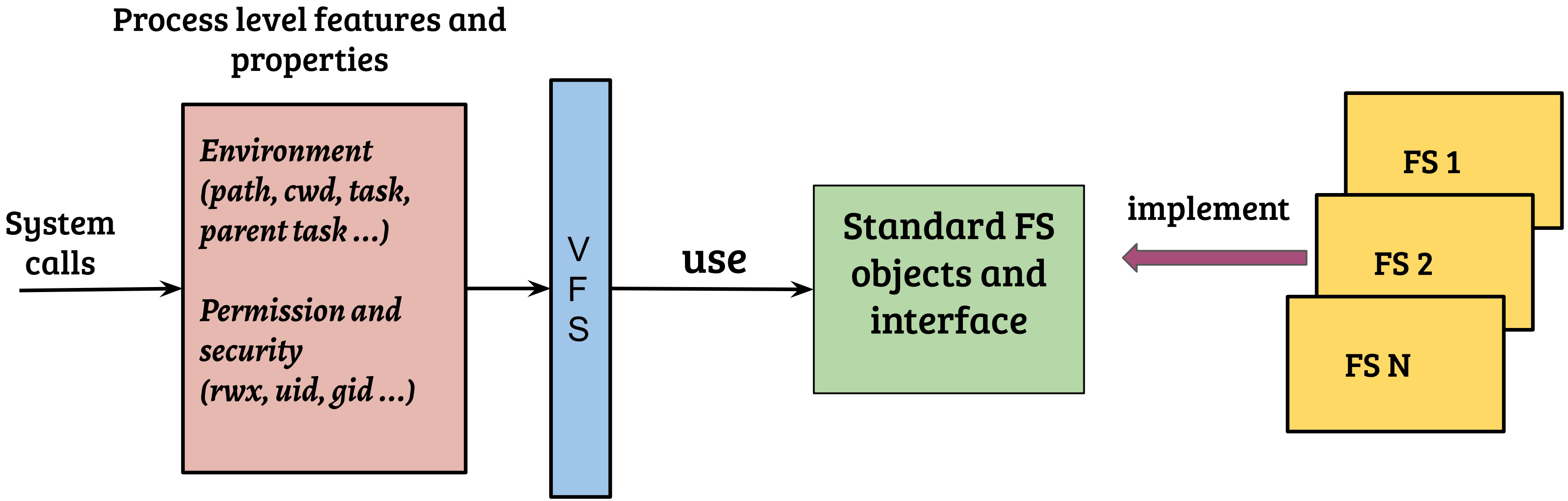
- What happens to the FD table and the file objects across fork( )?
    - What happens in exec( )?
- Can multiple FDs point to the same file object?

Process view of file

PCB (P1)

- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?
- Yes, duped FDs share the same file object (within a process)

Linux virtual file system (VFS)



- Object and interface choices guided by API requirement (mostly)
- Sometimes standards (e.g., POSIX) determines the interfacing
- Implementation can be different for different file systems

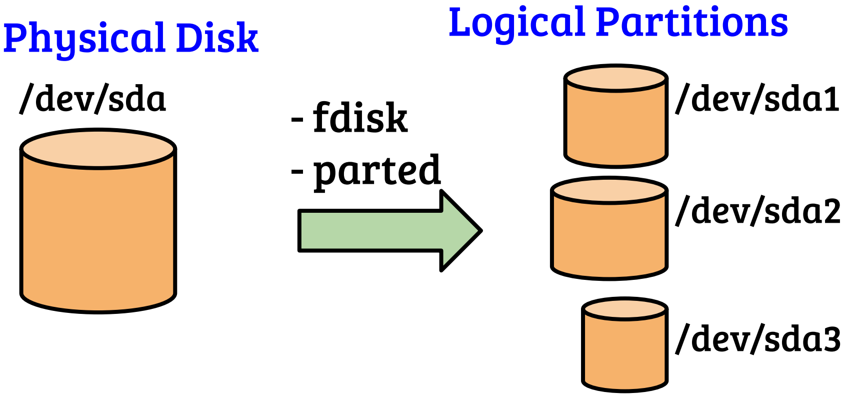
Linux virtual file system (VFS)

Process level features and properties

- VFS to Disk, How the dots are connected?
    - How a FS is created?
    - How system calls are mapped to the file system?

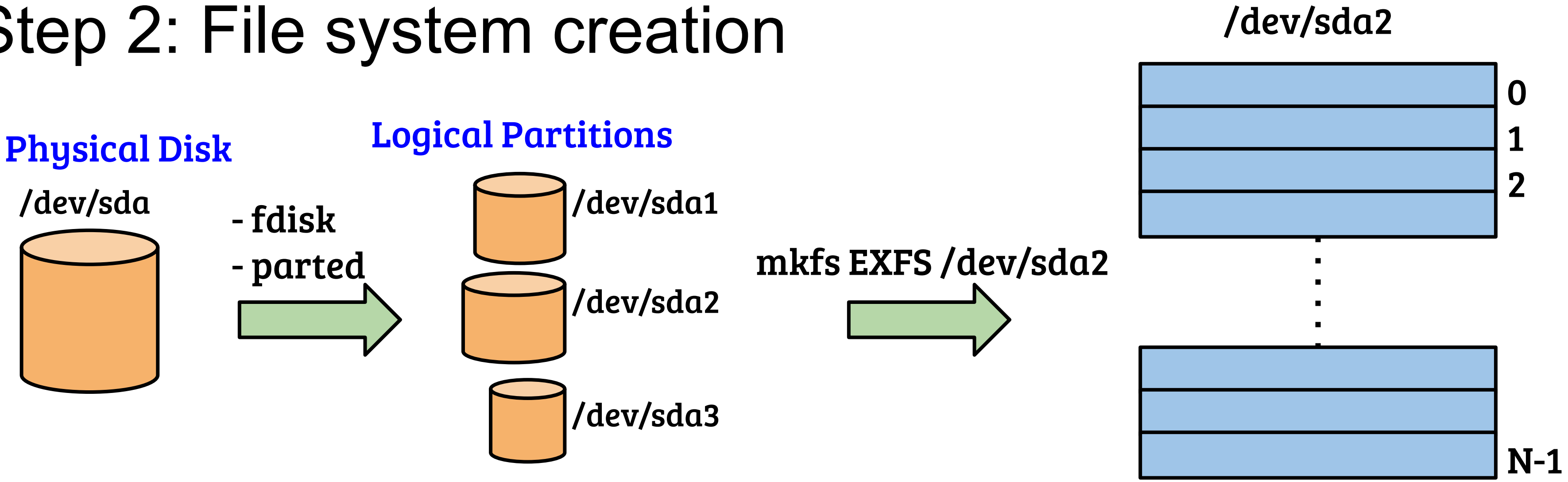
- Object and interface choices guided by API requirement (mostly)
- Sometimes standards (e.g., POSIX) determines the interfacing
- Implementation can be different for different file systems

Step-1: Disk device partitioning



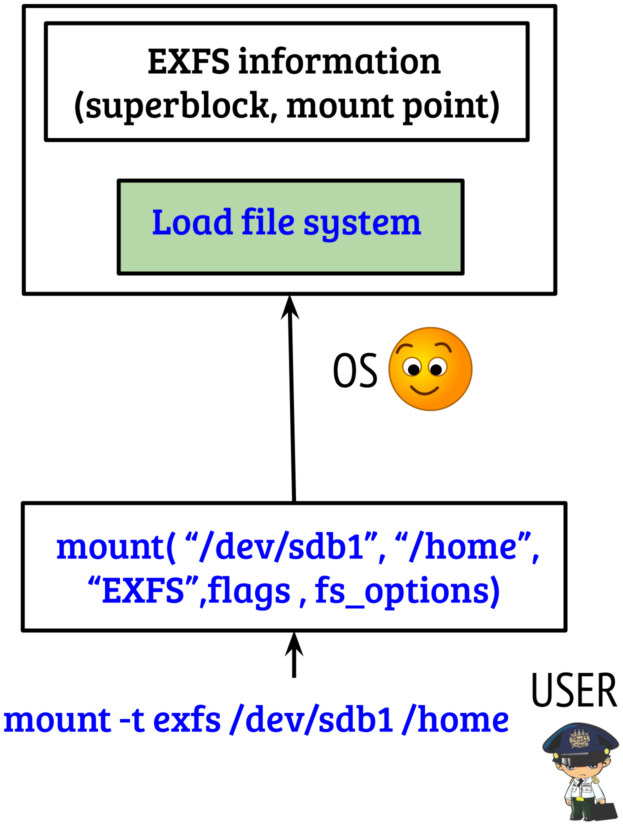
- Partition information is stored in the boot sector of the disk
- Creation of partition is the first step
  - It does not create a file system
- A file system is created on a partition to manage the physical device and present the logical view
- All file systems provide utilities to initialize file system on the partition (e.g., MKFS)

Step 2: File system creation



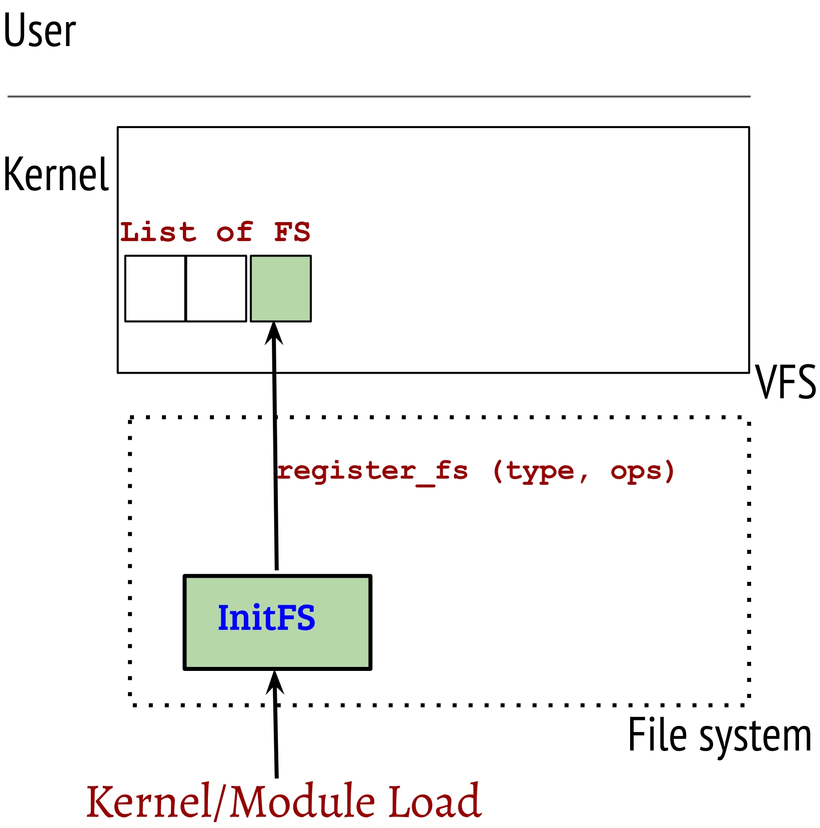
- MKFS creates initial structures in the logical partition
    - Creates the entry point to the filesystem (known as the super block)
    - At this point the file system is ready to be mounted

Step 3: File system mounting



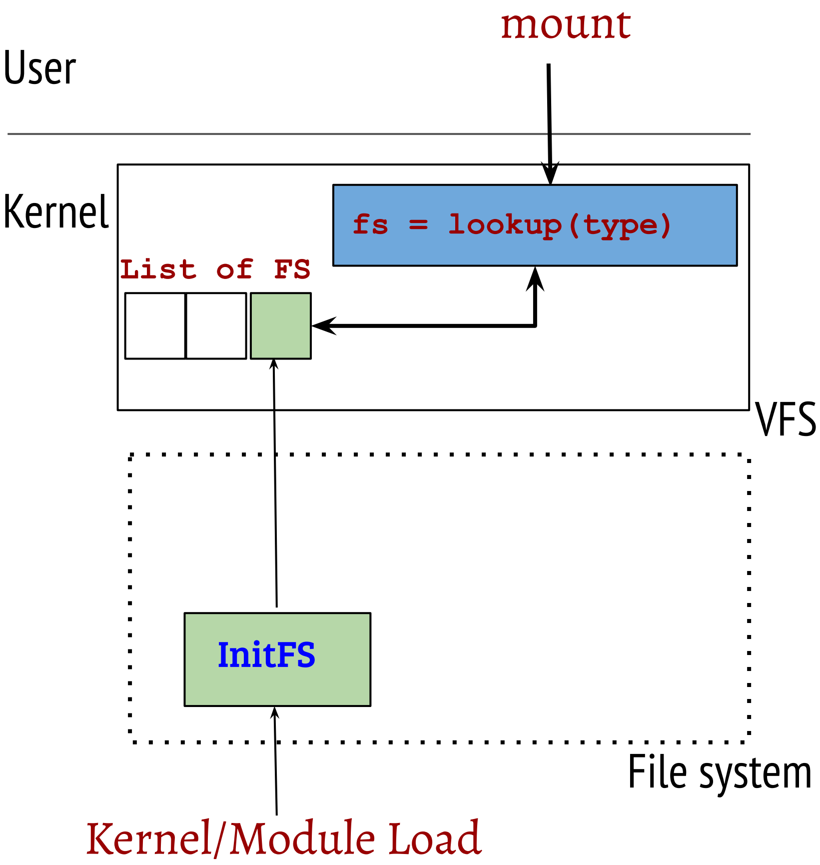
- *mount( )* associates a superblock with the file system mount point
- Example: The OS will use the superblock associated with the mount point "/home" to reach any file/dir under "/home"
- Superblock is a copy of the on-disk superblock along with other information

Details of FS mount in Linux (simplified)



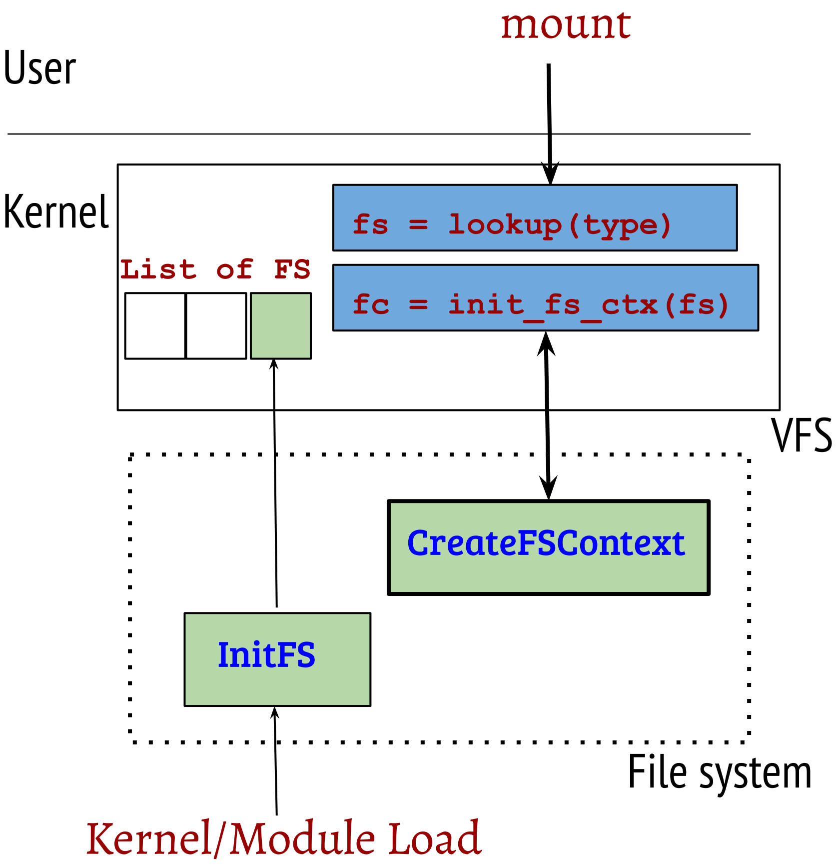
- File system registers itself with the VFS layer during initialization
  - "type" is the identity of the file system (e.g., ext4)
  - "ops" contains the callbacks for different events such as context initialization and mount
- VFS layer maintains a list of registered file system types

Details of FS mount in Linux (simplified)



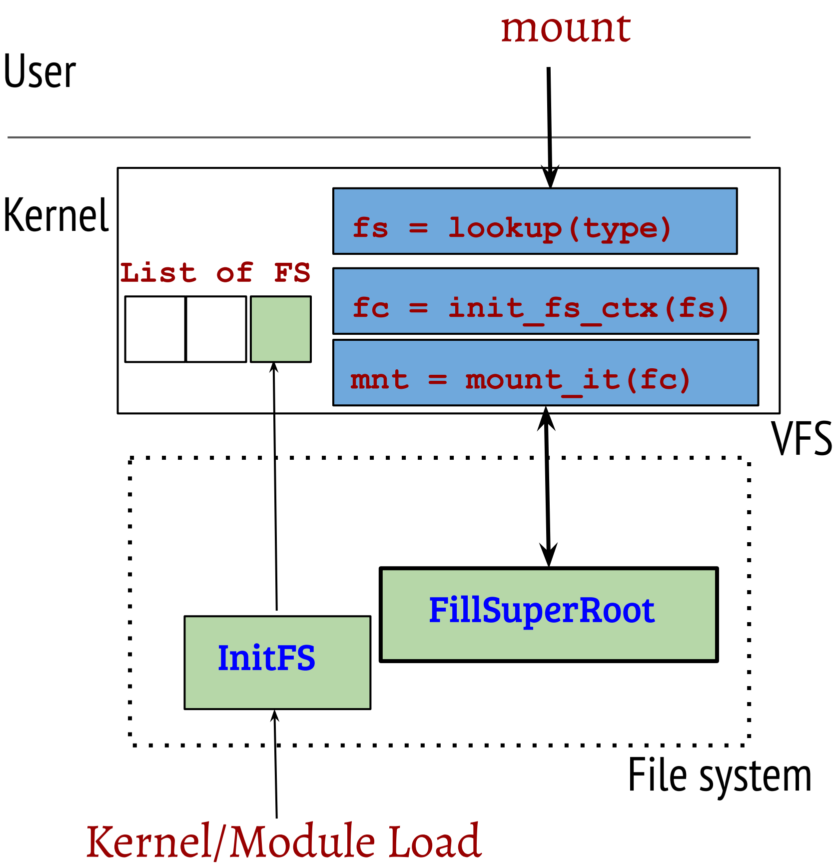
- System call handler for mount looks up the FS type

Details of FS mount in Linux (simplified)



- System call handler for mount looks up the FS type
- Creates a context – an instance of the FS for a given mount point

Details of FS mount in Linux (simplified)



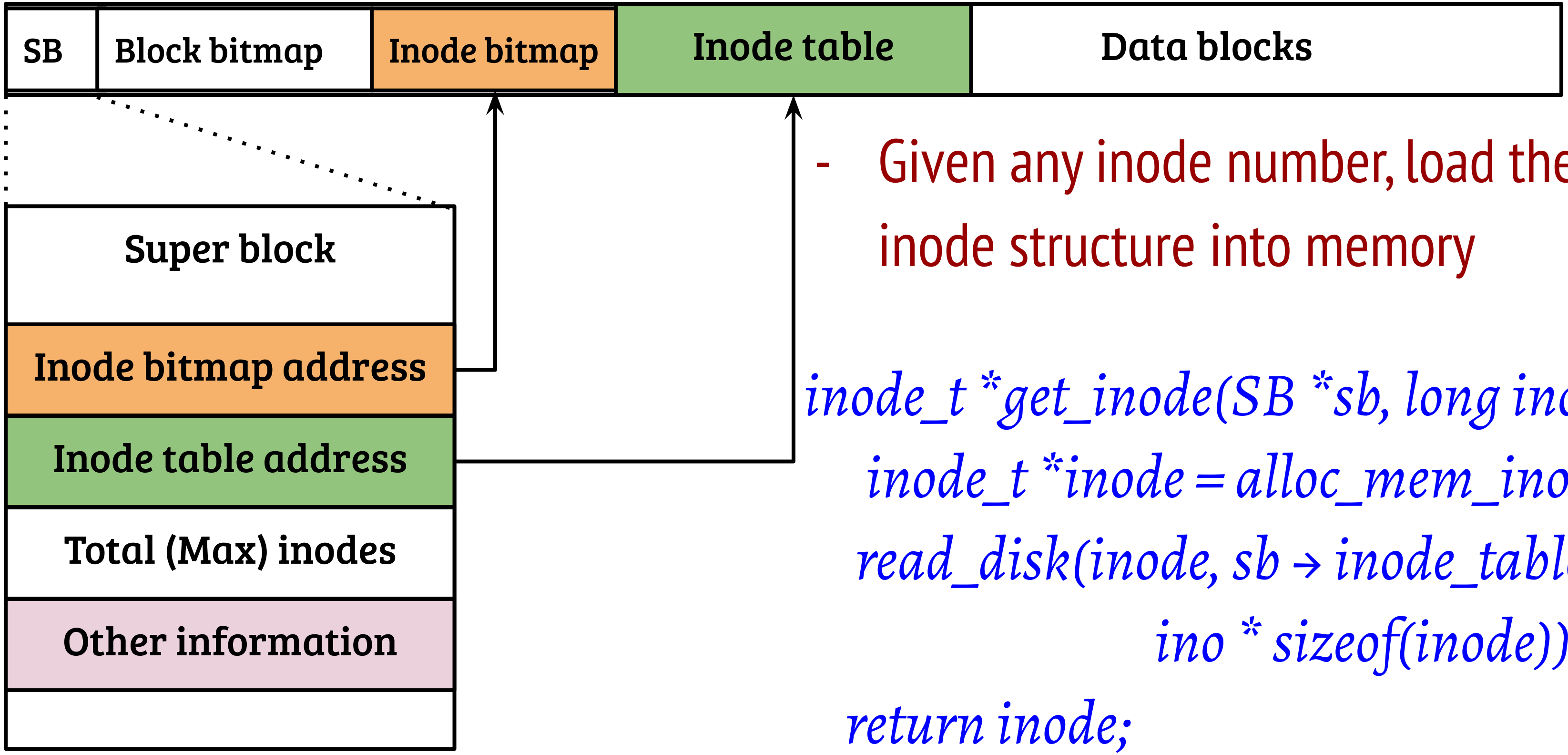
- System call handler for mount looks up the FS type
- Creates a context – an instance of the FS for a given mount point
- The FS fills superblock and root inode information (by performing disk block I/O)
- A new mount point is created at the VFS layer for future use. What kind of use?

Structure of an example superblock

```
struct superblock{
    u16 block_size;
    u64 num_blocks;
    u64 last_mount_time;
    u64 root_inode_num;
    u64 max_inodes;
    disk_off_t inode_table;
    disk_off_t blk_usage_bitmap;
    …
};
```

- Superblock contains information regarding the device and the file system organization in the disk
- Pointers to different metadata related to the file system are also maintained by the superblock
  - Ex: List of free blocks is required before adding data to a new file/directory

Typical file system organization (on-disk)



- Given any inode number, load the inode structure into memory

```
inode_t *get_inode(SB *sb, long ino){
    inode_t *inode = alloc_mem_inode( );
    read_disk(inode, sb → inode_table +
                      ino * sizeof(inode));
    return inode;
}
```

File system organization

SB	Block bitmap	Inode bitmap	Inode table	Data blocks

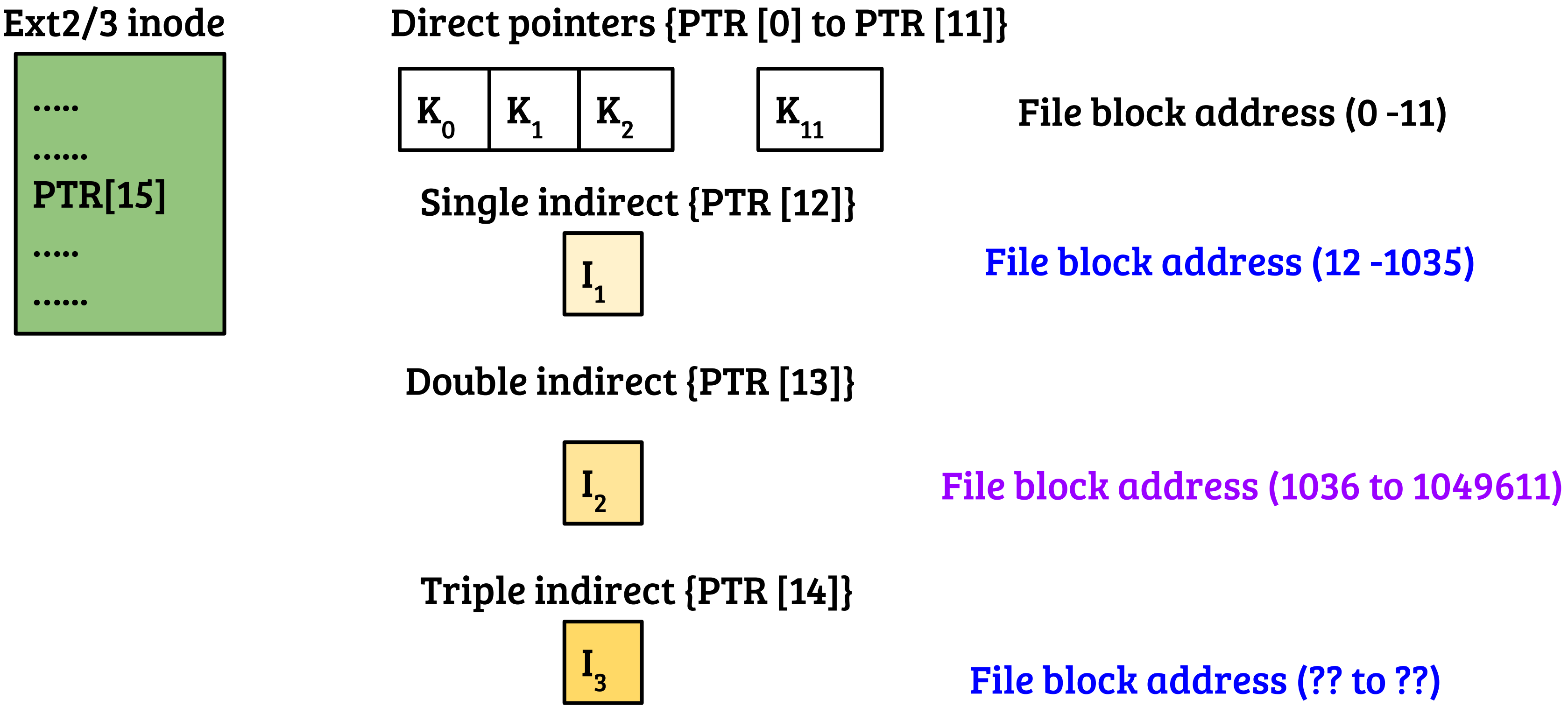
- File system is mounted, the inode number for root of the file system (i.e., the mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Specifically,
  - How to locate the content in disk?
  - How to keep track of size, permissions etc.?

```
return inode;
}
```

Inode

- A on-disk structure containing information regarding files/directories in the unix systems
    - Represented by a unique number in the file system (e.g., in Linux, “ls -i filename” can be used to print the inode)
    - Contains access permissions, access time, file size etc.
    - *Most importantly, inode contains information regarding the file data location on the device*
- Directory inodes also contain information regarding its content, albeit the content is structured (for searching files)

Ext2 file system indexing



File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Specifically,
    - How to locate the content in disk?
    - Index structures in inode are used to map file offset to disk location
    - How to keep track of size, permissions etc.?
    - Inode is used to maintain these information

Organizing the directory content

Fixed size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        char name[FNAME_MAX];
};
```

- Fixed size directory entry is a simple way to organize directory content
- Advantages: avoid fragmentation
- Disadvantages: space wastage

Flat organization of directory entries

Fixed size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        char name[FNAME_MAX];
};
```

Variable size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        u8 entry_len;
        char name[name_len];
};
```

- Variable sized directory entries contain length explicitly
- Advantages: less space wastage (compact)
- Disadvantages: fragmentation issues

File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Read the content of the root inode and search the next level dir/file
- Specifically,
  - How to locate the content in disk?
  - Index structures in inode are used to map file offset to disk location
  - How to keep track of size, permissions etc.?
  - Inode is used to maintain these information

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Examples:
  - Opening a file

fd = open(“/home/user/test.c”, O_RDWR);

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Examples:
  - Opening a file

    fd = open("/home/user/test.c", O_RDWR);

  - Normal shell operations

    /home/user$ ls

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Executables, configuration files, library etc. are accessed frequently
- Many directories containing executables, configuration files are also accessed very frequently. Metadata blocks storing inodes, indirect block pointers are also accessed frequently

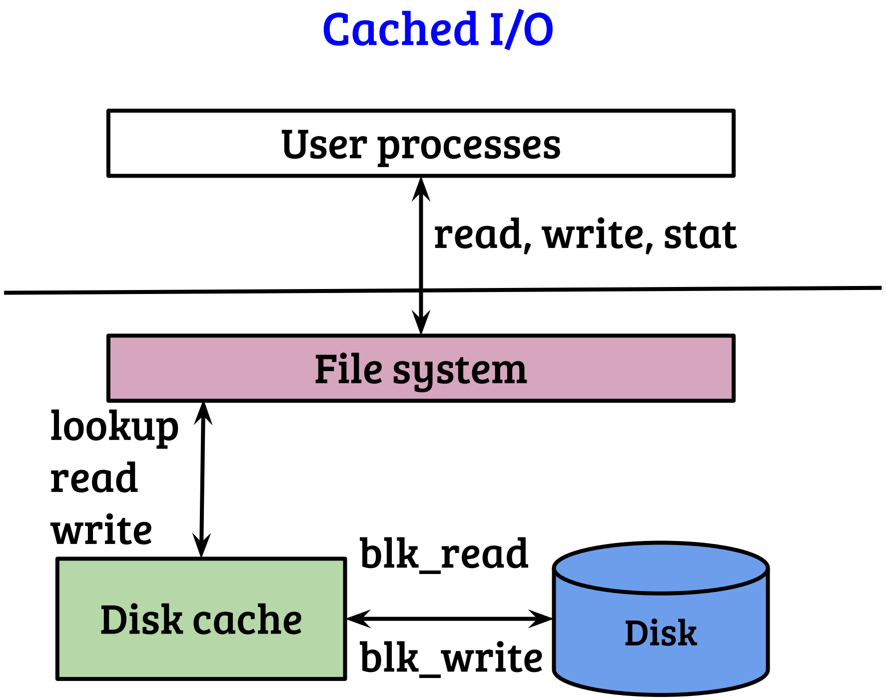
/home/user$ ls

File system and caching

- Accessing data and metadata from disk impacts performance
- Can we store frequently accessed disk data in memory?
  - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
  - Are there any complications because of caching?
  - How the cache managed? What should be the eviction policy?

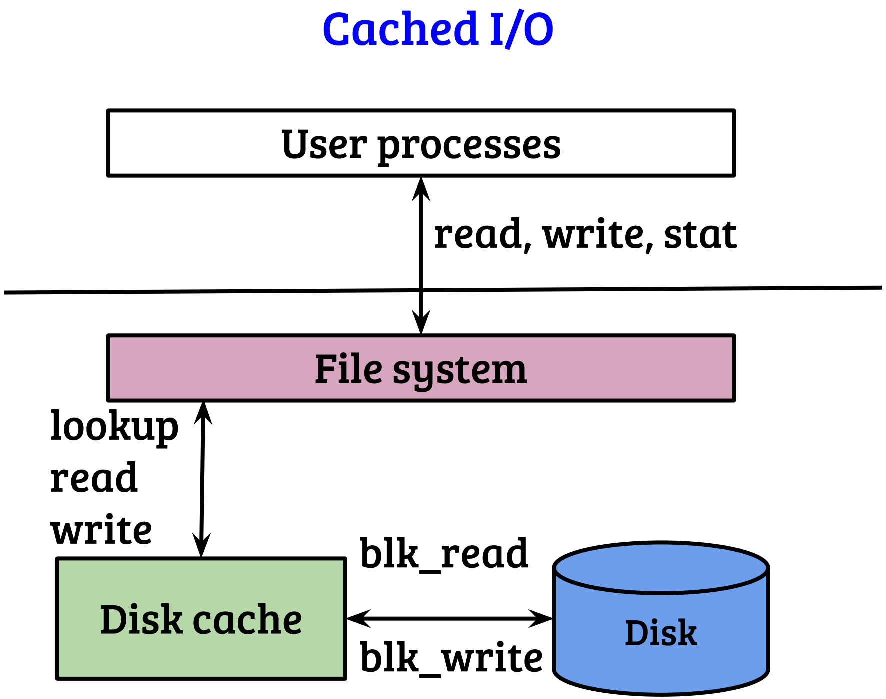
/home/user$ ls

Block layer caching



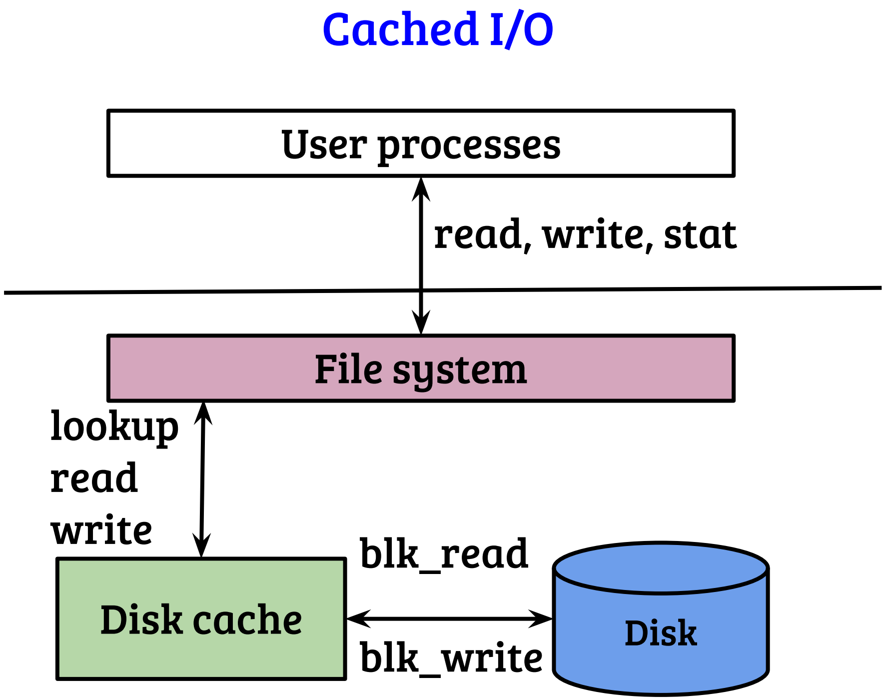
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?

Block layer caching



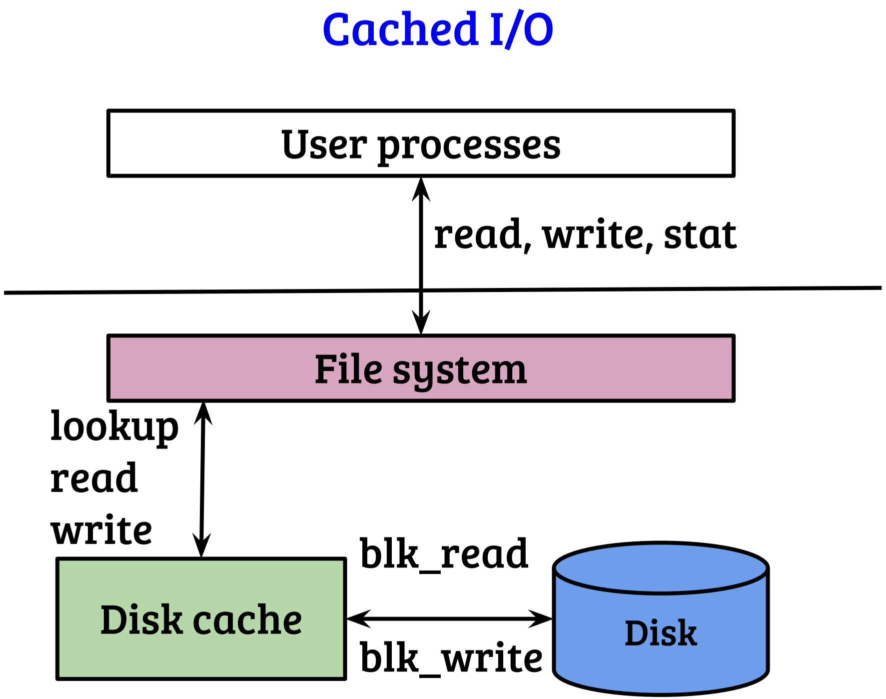
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache

Block layer caching



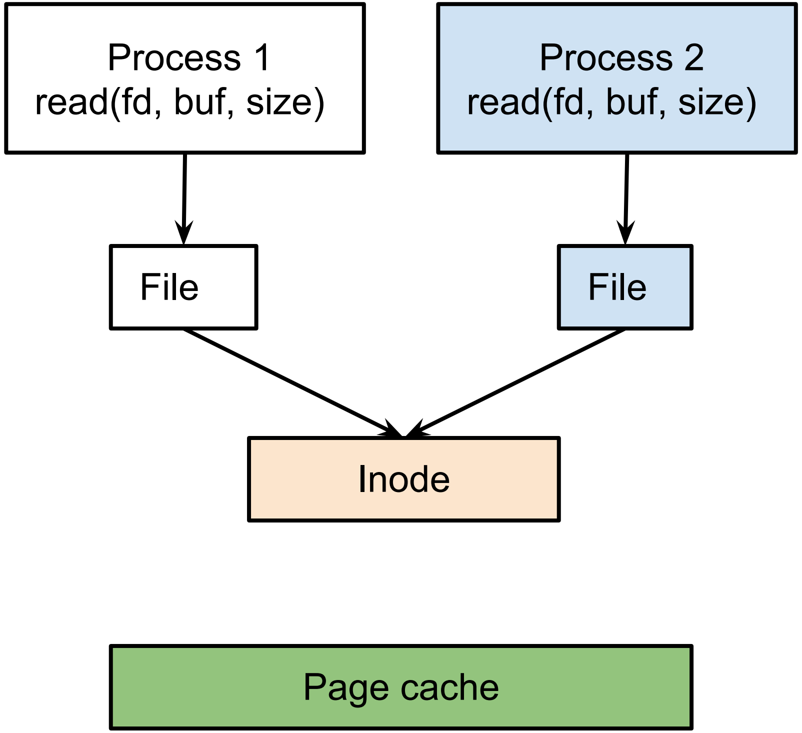
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache
- Works fine for metadata as they are addressed using block numbers

File layer caching (Linux page cache)



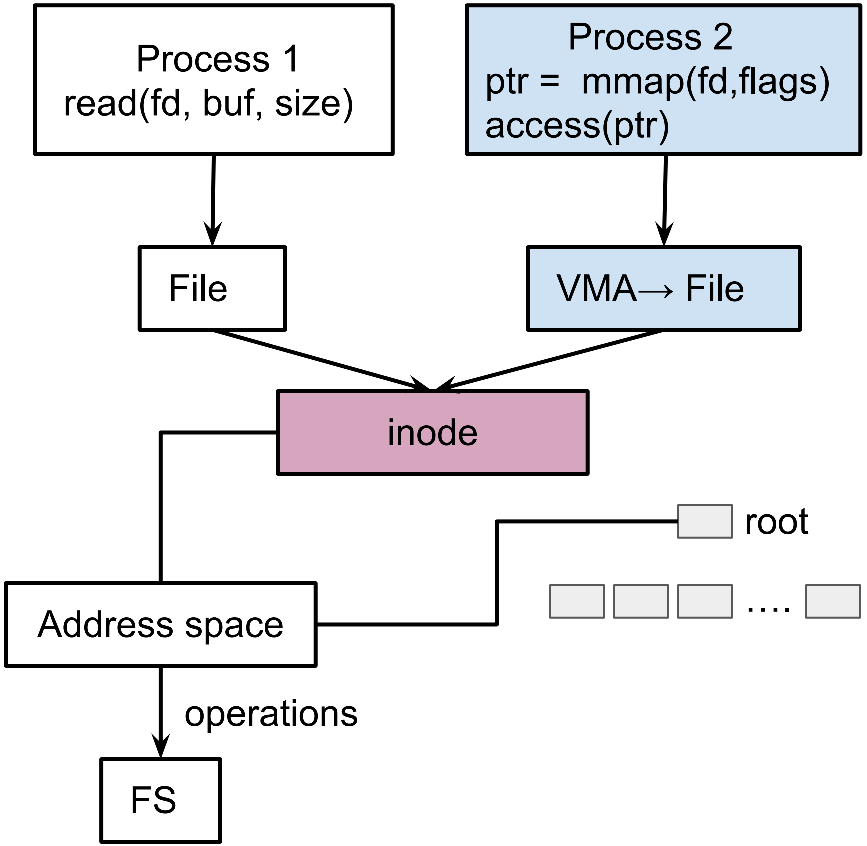
- Store and lookup memory cache using {inode number, file offset} as the key
- For data, index translation is not required for file access
- Metadata may not have a file association, should be handled differently (using a special inode may be!)

Linux page cache: A multi-purpose FS caching layer



- Requirement: File block lookup at different offsets
    - File size can range from very small to huge
- Recall: mmap-ing a file creates a VMA struct
- Should handle both file I/O and page faults

File → Inode → Address spaces → Page Cache



- A per inode cache
  - Lookup, insert, evict, dirty-flush
- Radix tree
  - Root pointed by address space struct
  - Operations at a page size (4K) granularity
- Homework: For a given file, find the number of file blocks cached in PC

File system and caching

- Accessing data and metadata from disk impacts performance
- Can we store frequently accessed disk data in memory?
  - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
  - File layer caching is desirable as it avoids index accesses on hit, special mechanism required for metadata.
  - Are there any complications because of caching?
  - How the cache managed? What should be the eviction policy?

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
    - Example-1: If a write( ) system call is successful, data must be written
    - Example-2: If a file creation is successful then, file is created.
    - Difficult to achieve with asynchronous I/O

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
  - Example-1: If a write( ) system call is successful, data must be written
  - Example-2: If a file creation is successful then, file is created.
  - Difficult to achieve with asynchronous I/O
- Consistency w.r.t. file system invariants
  - Example-1: If a block is pointed to by an inode data pointers then, corresponding block bitmap must be set
  - Example-2: Directory entry contains an inode, inode must be valid
  - Possible, require special techniques