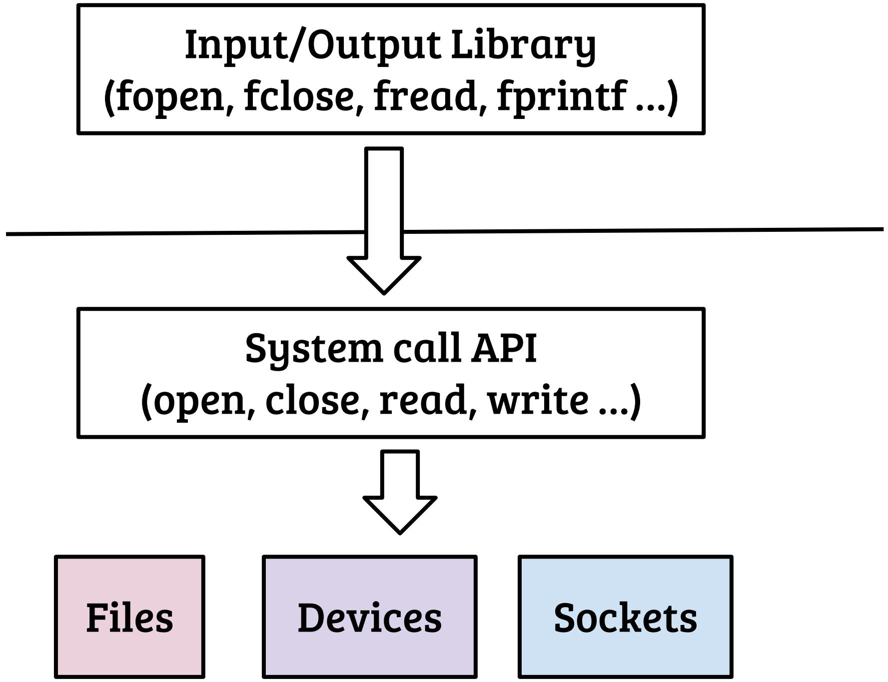
Operating Systems

Filesystem: API and design

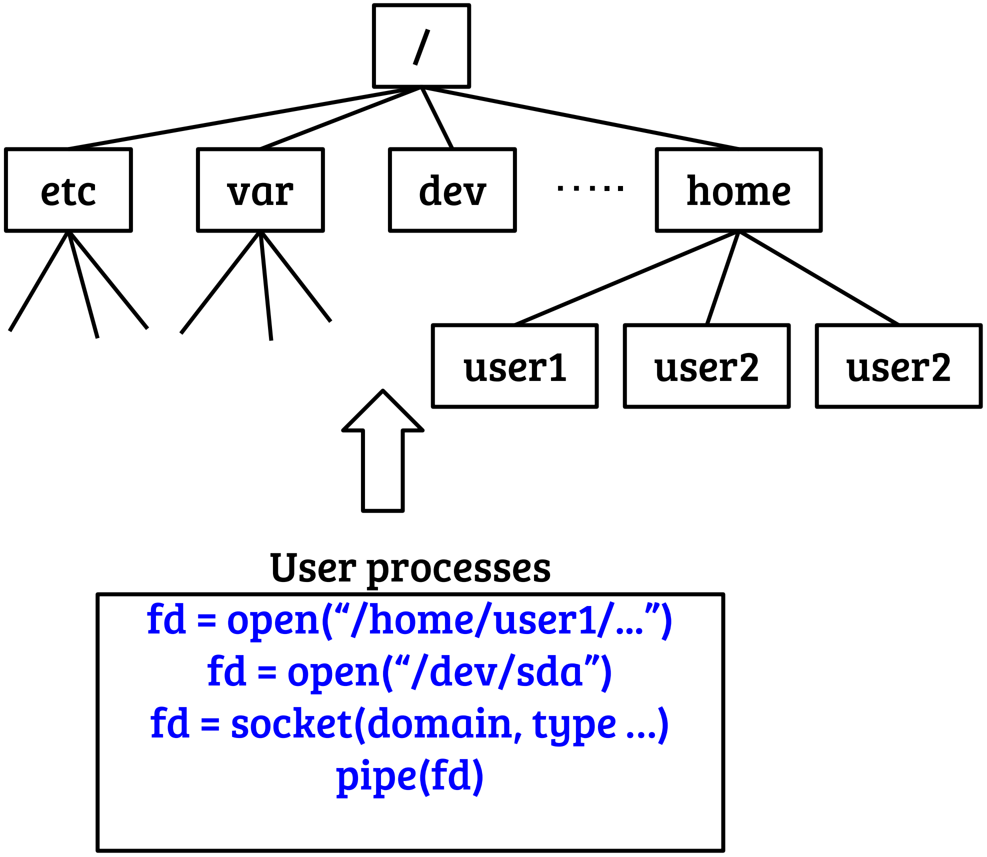
Debadatta Mishra, CSE, IITK

System calls (open, close, read, write)



- 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.

User view of a file system



- Users view filesystem is a big-fat tree structure
- The directory tree can span across many physical devices, or event machines!
- User processes access files through a file handle
- Not all file handles correspond to a path in the directory tree

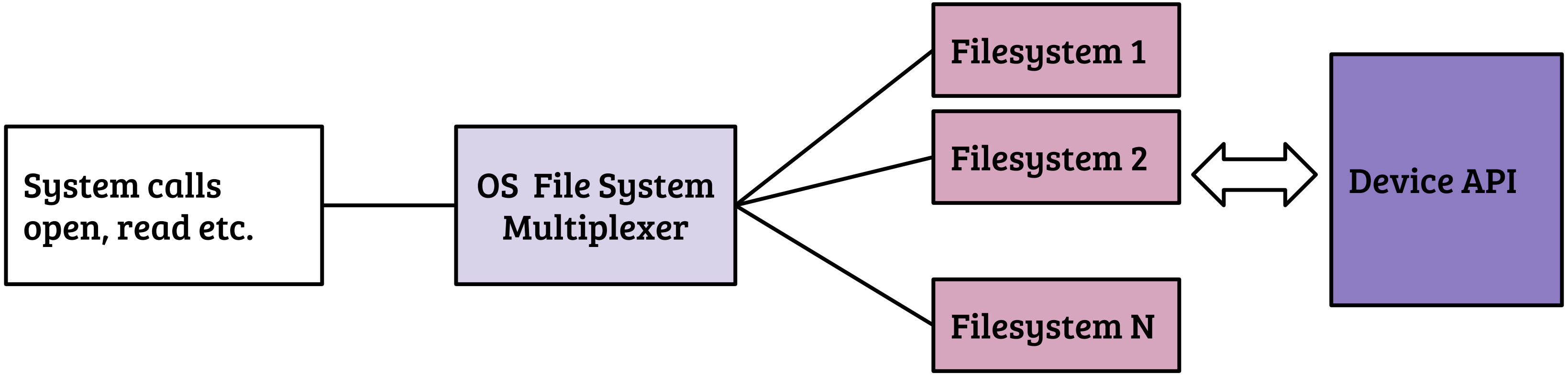
Let us pause, and think!

- What happens when we execute "*ls -ltr /home/user1/code.c*"?
- We know how bash deals with command execution, but Where is *ls* ?
  - Which directory?
  - What device?
  - Which offset?
- How this path is resolved?
  - Can be on any device, any partition
- How access permissions checked?
  - Permissions at every level is required to be checked, why?
- Where is the information regarding the file *code.c*?

Some more FS related system calls

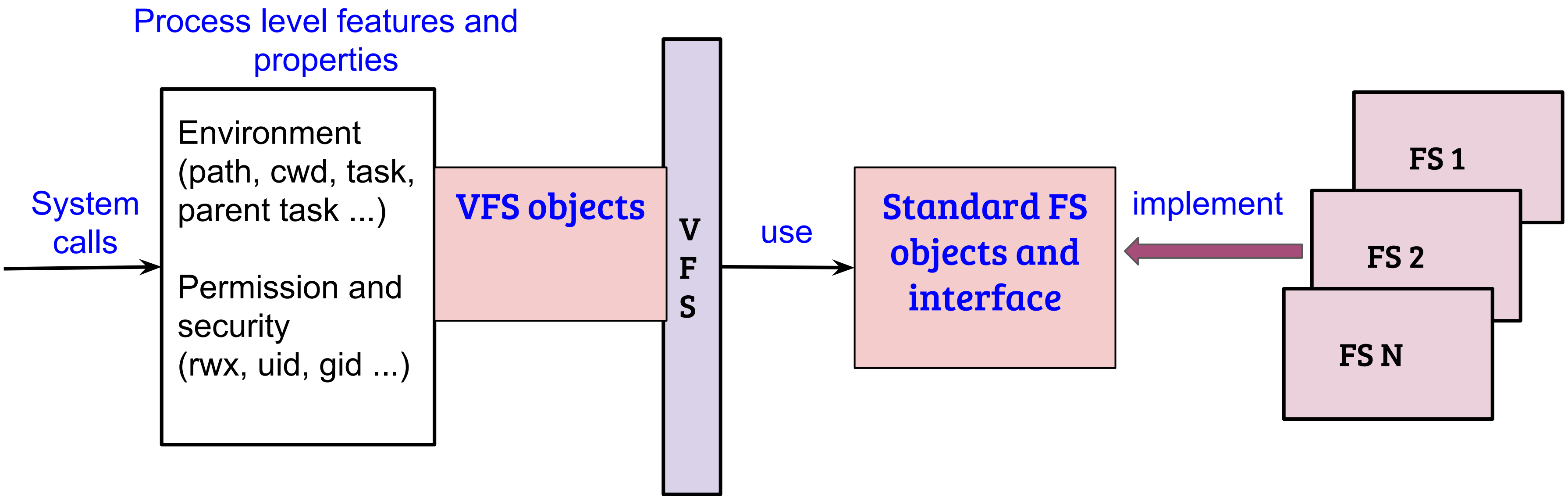
- stat( ) and fstat( )
- lseek( )
- chmod( )
- opendir( )
- chdir ( )
- fsync( )

Multi-FS systems: layers and responsibilities



- Path name translation e.g., open (/home/user1/data/file.c)
- create, delete, chown, chmod …
- Open, read, write, truncate…
- Multiplexer or filesystem?

Layer FS example: Linux virtual file system (VFS)



- Object and interface choices guided by API requirement (mostly)
- Sometimes unix tradition determines the interfacing

VFS - FS interface: superblock

- Every mounted file system must provide a super block to the VFS
  - FS is not a real on-disk FS, does not matter, VFS requires it anyway
- Device information, block size, …
- FS Organization information → Inode information
- Operations: remount, unmount, syncfs, alloc inode, destroy inode,
- First block of a partition is used to store superblock for most file system
- Superblock is the entry point to any file system

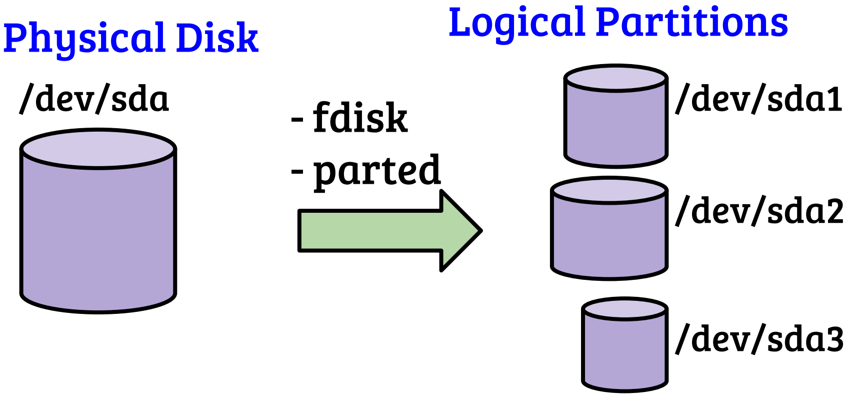
VFS - FS interface: inode

- A traditional representation of a file in unix systems
  - Permissions, access time, file size, file offset → device offset
  - Unique for every regular file in the file system
- Most file systems implement a similar on-disk version
- “Don’t care if you represent a file on disk in a different way, you show me the way I want to see a file”
- Information: permissions, ownership, size, access time etc.
- Operations: child_ inode lookup (parent inode, child name), create ...

VFS - FS interface: dentry

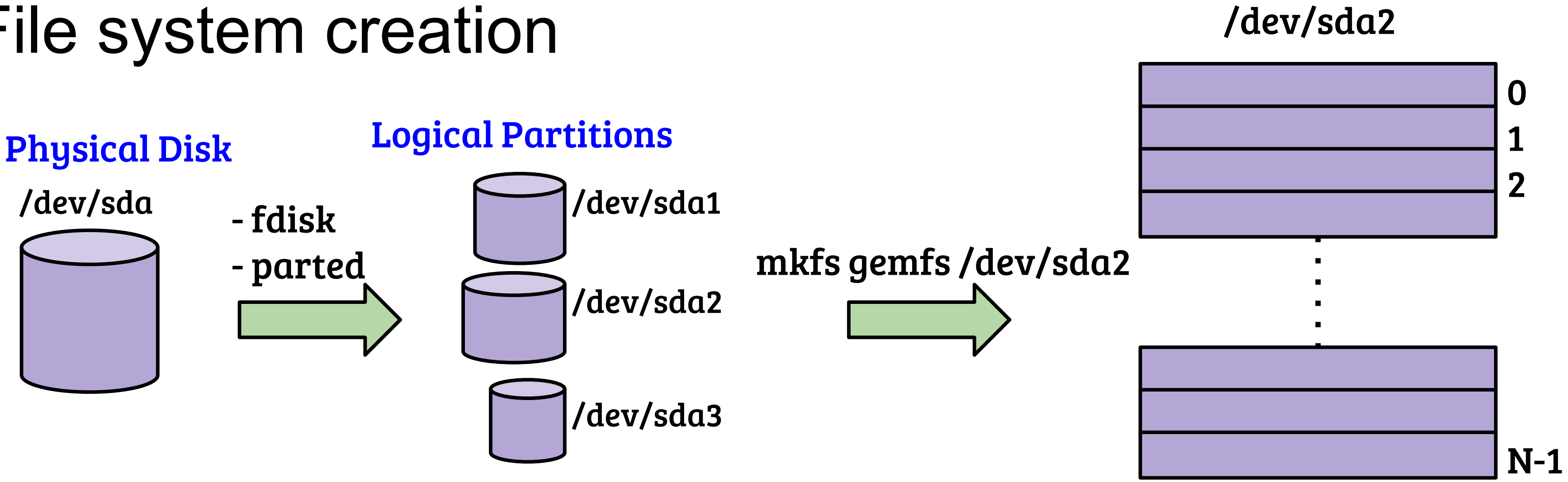
- Dentry represents a specific element in a file path
  - Both for file and directory
- May not have an equivalent on-disk state
- Explicit representation of parent dir - subdirectory relationships
- Path translation relies on dentry objects

Block device: partitioning



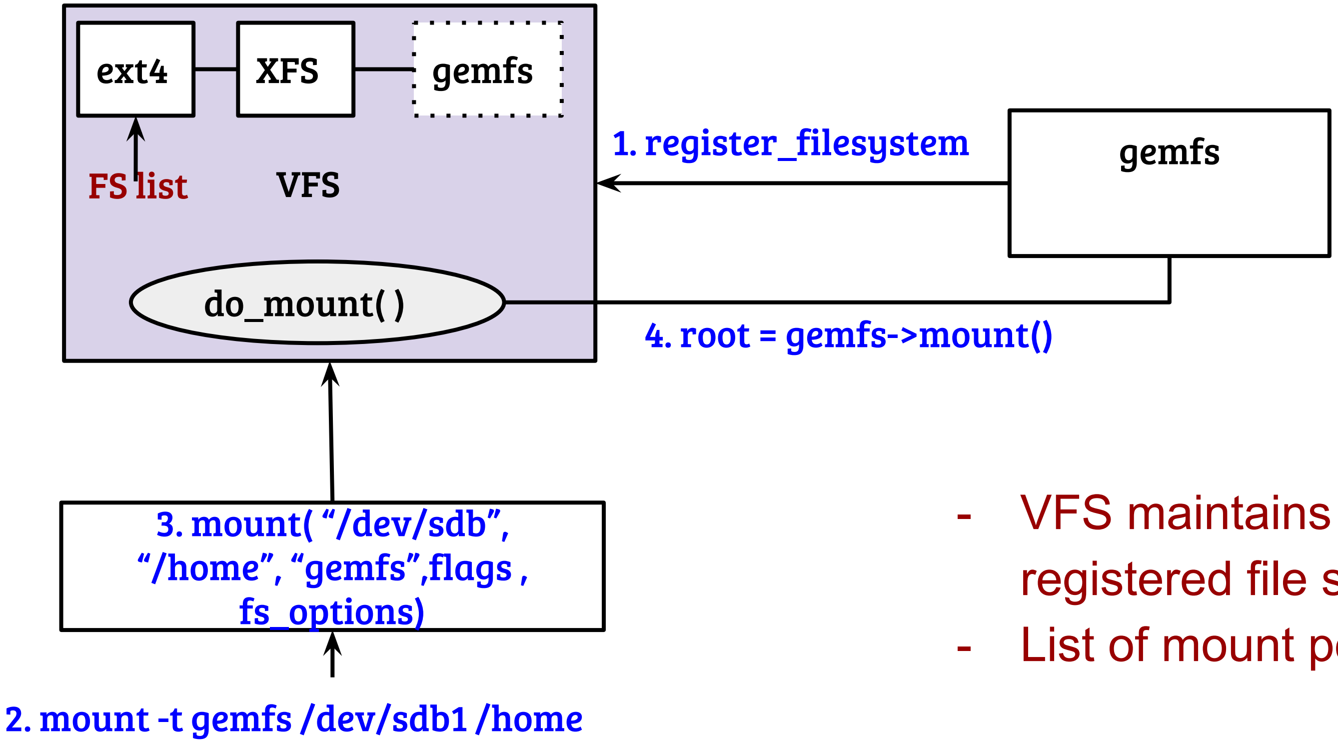
- Where is the partitioning information?
- Most OSs provide a generic block layer to access different logical partitions

File system creation



- Blank partition is useless, VFS requires root dentry
- dentry { inode, superblock}
- Create superblock (in block 0) and other structures to mount the FS
- Once mounted, should be self sufficient for access and expansion

File system registration and mounting



- VFS maintains list of registered file systems
- List of mount points

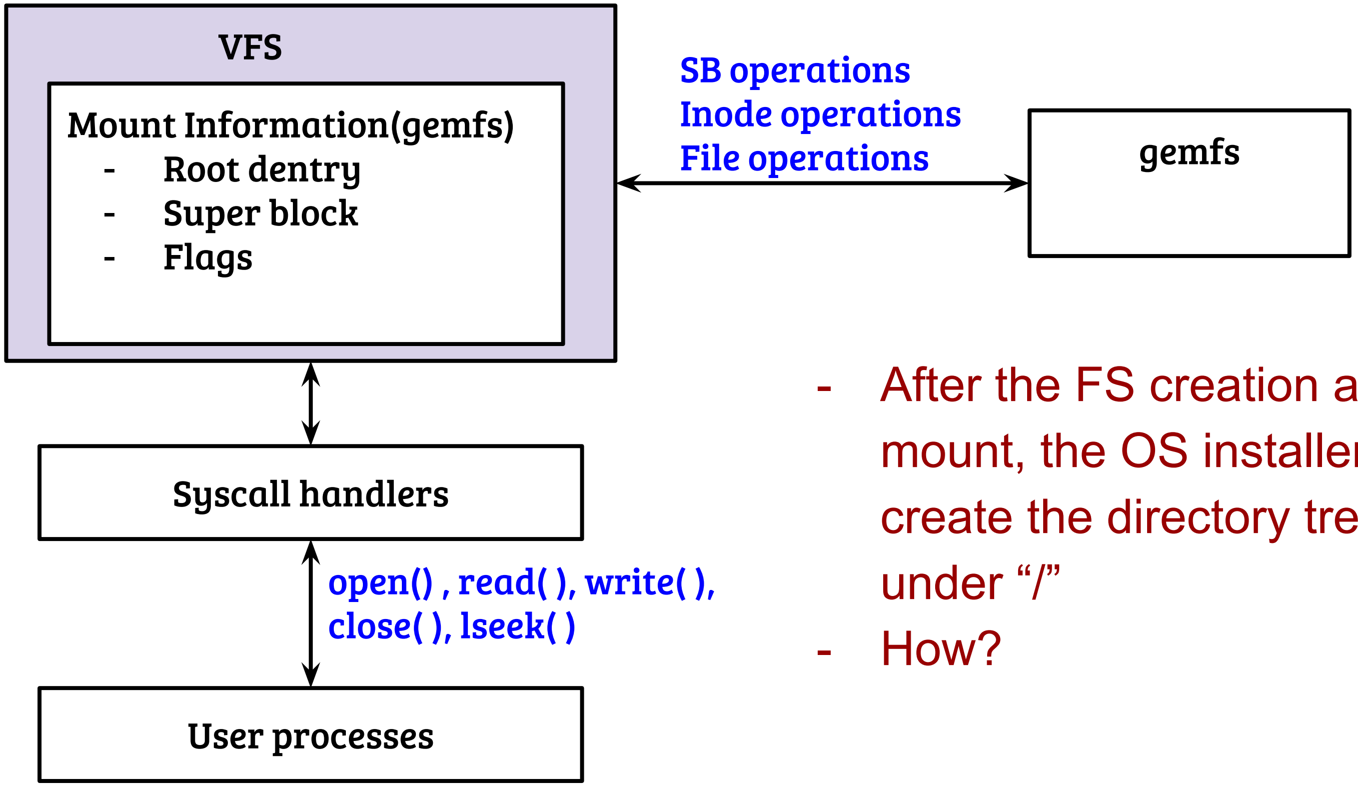
File system registration - VFS Layer

```
struct file_system_type{
                                    const char *type;  /*ext4, btrfs, sysfs ...*/
                                    struct dentry* (*mount) (...);  /*Returns the dentry of mount point*/
                                    ………
};
int register_filesystem( struct file_system_type *fs)
{
   /*Add to the list of file systems*/
}
int do_mount( char *type, char *bdev, char* mount_point)   /*VFS mount*/
{
   …...
   Struct file_system_type *fs = search_type( type);
   root = fs->mount(fs, dev ...);
   ……
   return 0;
}
```

File system registration - FS Layer

```
/*FS Layer.  Template not exact implementation*/
struct file_system_type gemosfs = {
                                        "gemfs" ,    /*Type*/
                                        gemfs_mount,
                                        ………
};
Struct dentry* gemfs_mount(struct file_system_type *fs, char *bdev, …)
{
    struct super_block *sb = create_sb( );
    struct inode *root = create_root_inode( );
    return(new_dentry(sb, root));
}
int init_gemos( )
{
    register_filesystem(&gemosfs);
    …..
}
```

VFS state: post mount

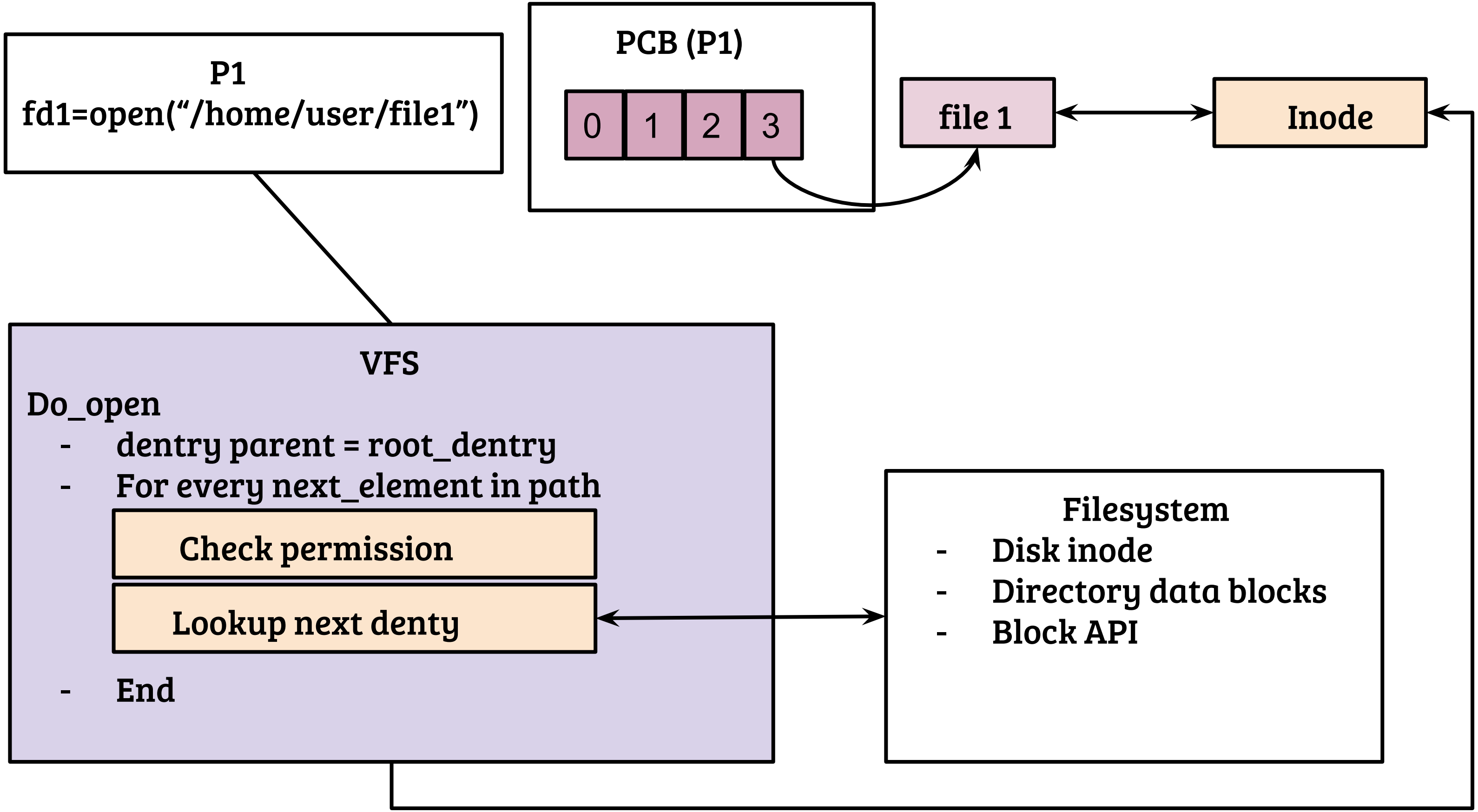


- After the FS creation and mount, the OS installers create the directory tree under "/"
- How?

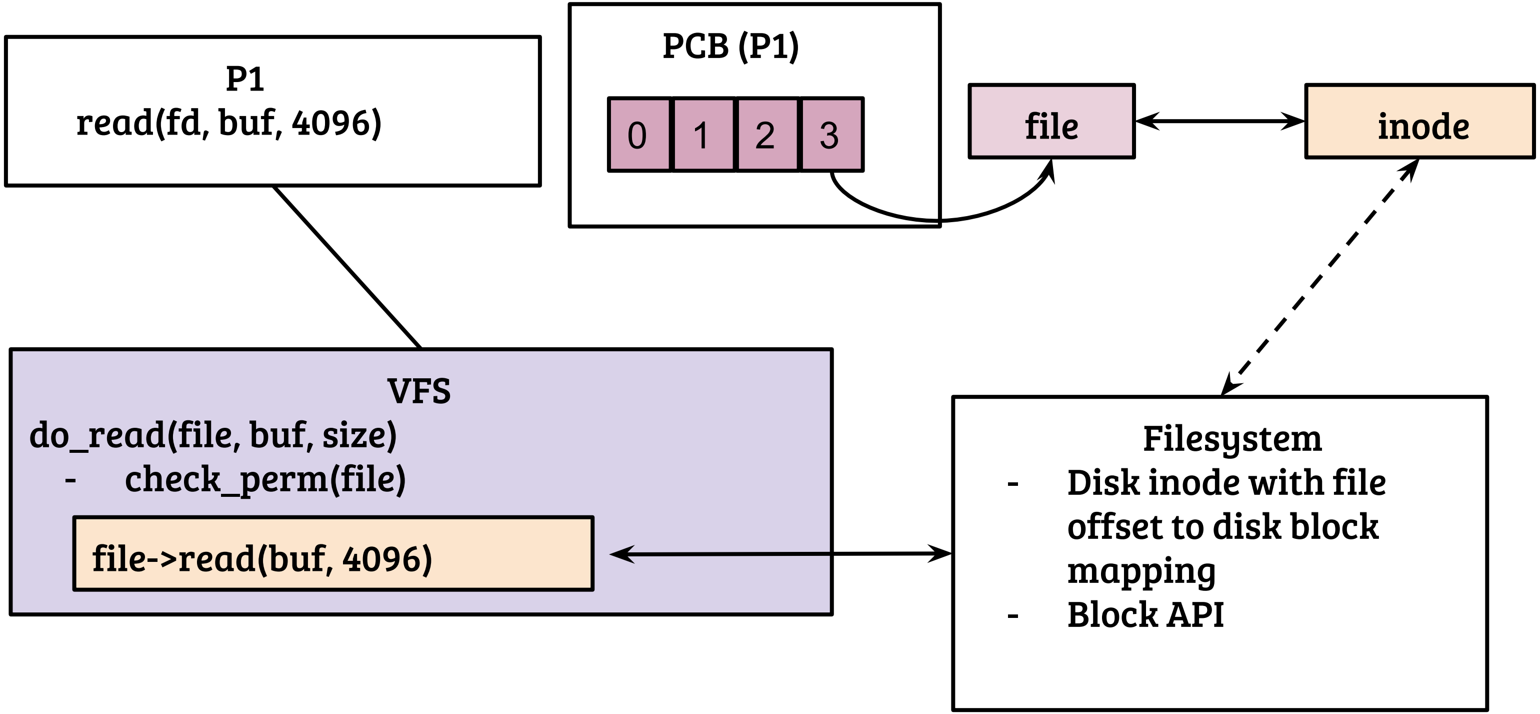
During installation ...

- After booting OS from external disk, mount file system → root_dentry
- Installation process creates the directory tree using system calls like mkdir( ), open(O_CREAT) etc.
- The VFS layer invokes root_dentry->create( ) and so on.

Putting everything together



Putting everything together



- Disk inode can be derived from VFS inode
- How file operations installed?

VFS: optimization

- During path translation multiple invocation of lookup(parent, “next level directory”)
- Possible optimizations?

VFS: optimization

- During path translation multiple invocation of lookup(parent, “next level directory”)
- Possible optimizations?

- Translation cache: A key-value store where key is {parent dentry, child name} which maps to the child dentry
- What about full path caching?

Physical storage management

- File system operate on file-block granularity, mostly same as memory page size (why?)
- The filesystem must have a mechanism to identify free blocks
- Disk size can be TBs, size of maintenance structures can not be overwhelming
- How to manage?

Physical storage management

Alternate 1: A linked list of free blocks

- Performance of allocation and free
- Ease of usage, how to store the next pointer?
- Scalability
- Fault tolerance and consistency

Physical storage management

Alternate 1: A linked list of free blocks

- Performance of allocation and free
- Ease of usage, how to store the next pointer?
- Scalability
- Fault tolerance and consistency

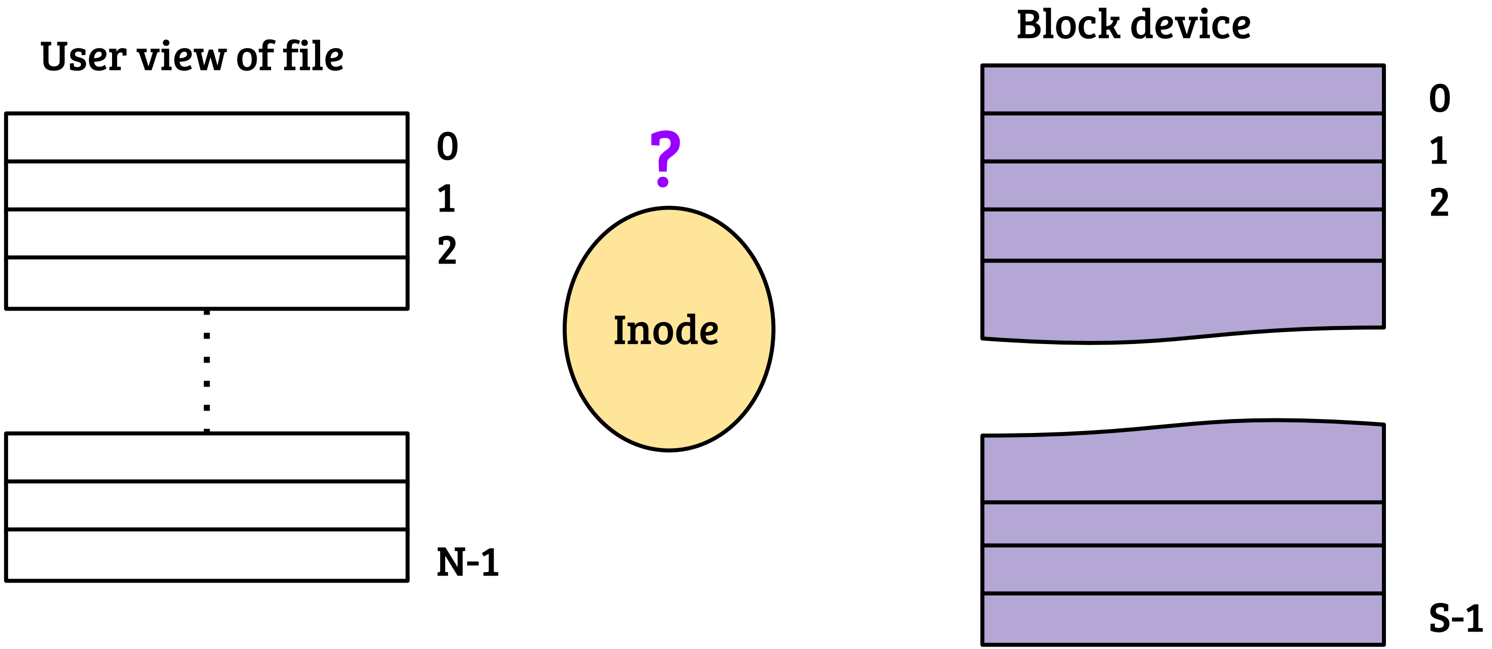
Alternate 2: A bitmap of all disk blocks, where i^{th} bit position → i^{th} block, zero → free, 1→ used

- Performance of allocation and free
- Scalability
- Fault tolerance and consistency

Physical storage management

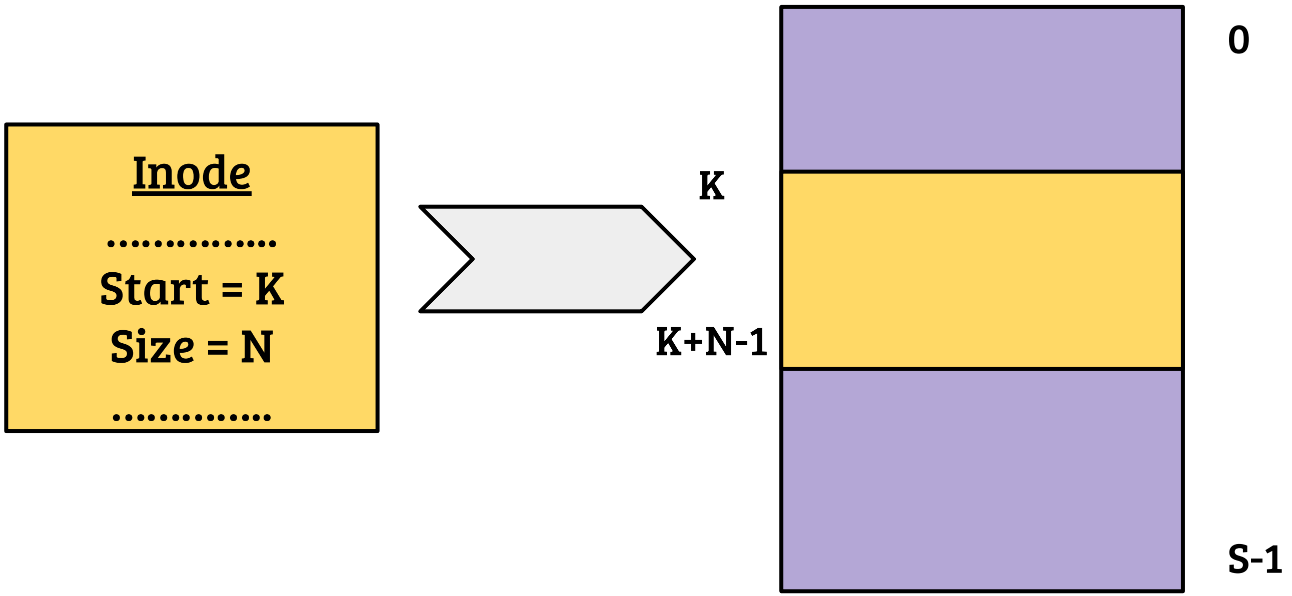
- Block bitmap is used in most of the file systems
- Fault tolerance can be achieved using replication --- possible as it is space efficient

File data storage and retrieval



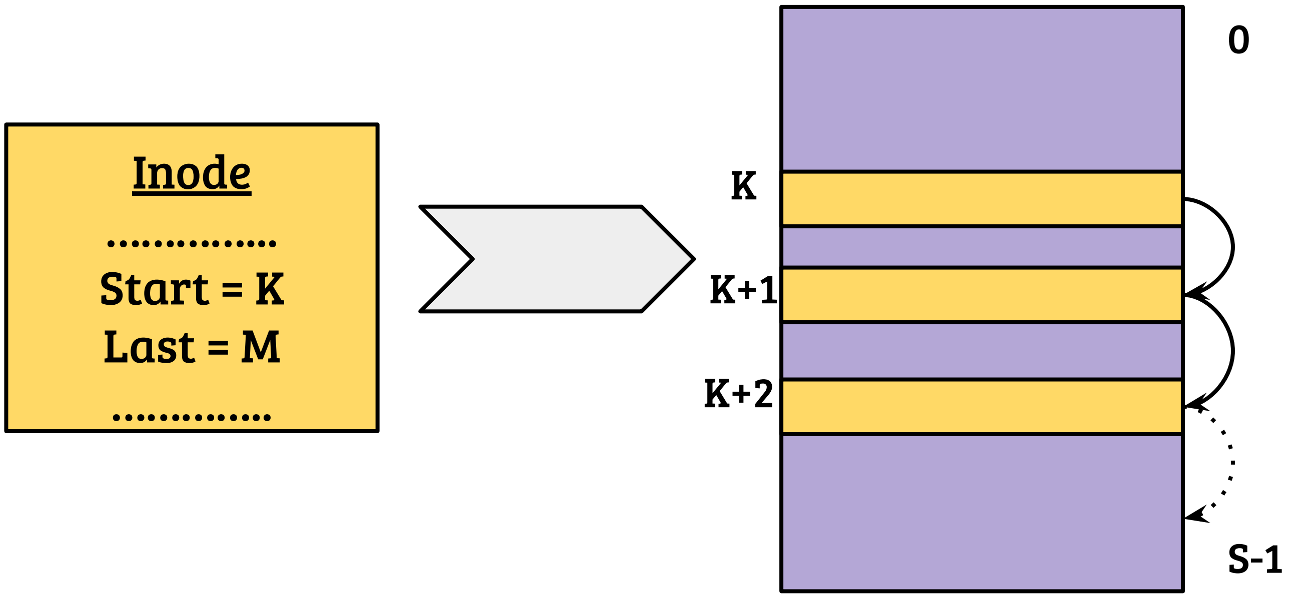
- File size can range from few bytes to gigabytes
- Can be accessed in a sequential or random manner
- How to design the mapping structure?

Contiguous allocation



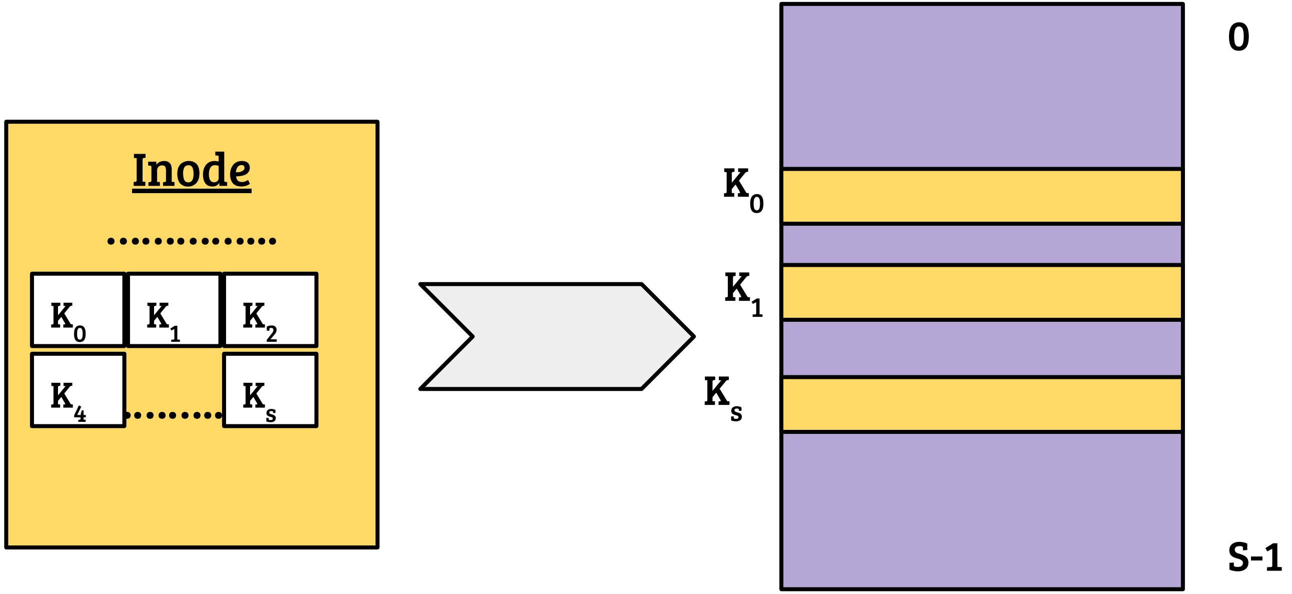
- Allocation strategy: Best fit, first fit etc.
- Works nicely for both sequential and random access
- Append operation → expand file, how?
- Fragmentation issues

Linked allocation



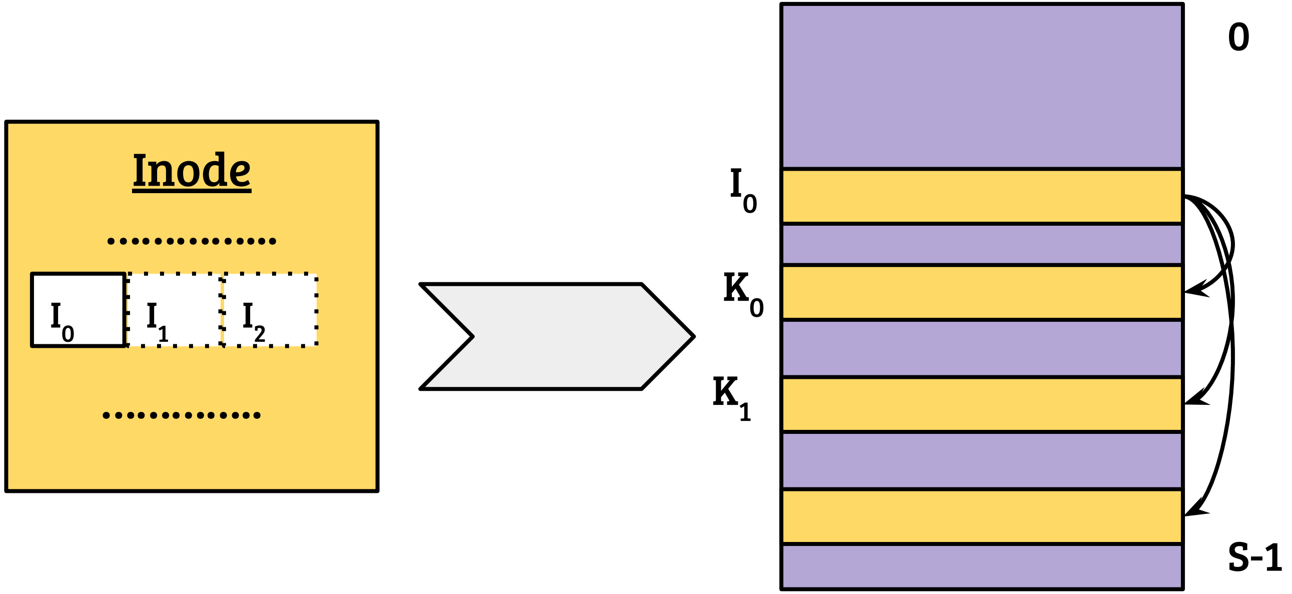
- Inode contains the starting block and size
- Flexible: growth, shrink
- Random access is bad
- Why maintain last block not size?

Direct block pointers



- Inode contains the pointers to the block
- Flexible: growth, shrink, random access is good
- What about large files?
- What about maintaining page table like indirections?

Indirect block pointers



- Inode contains the pointers to a block containing pointers to data blocks
- Flexible: growth, shrink, random access is good
- Maintain metadata block for small files, still limited maximum size

Hybrid block pointers: Unix and Linux

Ext2/3 inode

.....

......

PTR[15]

.....

......

Direct pointers {PTR [0] to PTR [11]}

K_0 K_1 K_2 K_{11}

File block address (0 -11)

Single indirect {PTR [12]}

I_1

File block address (12 -1035)

Double indirect {PTR [13]}

I_2

File block address (1036 to 1049611)

Triple indirect {PTR [14]}

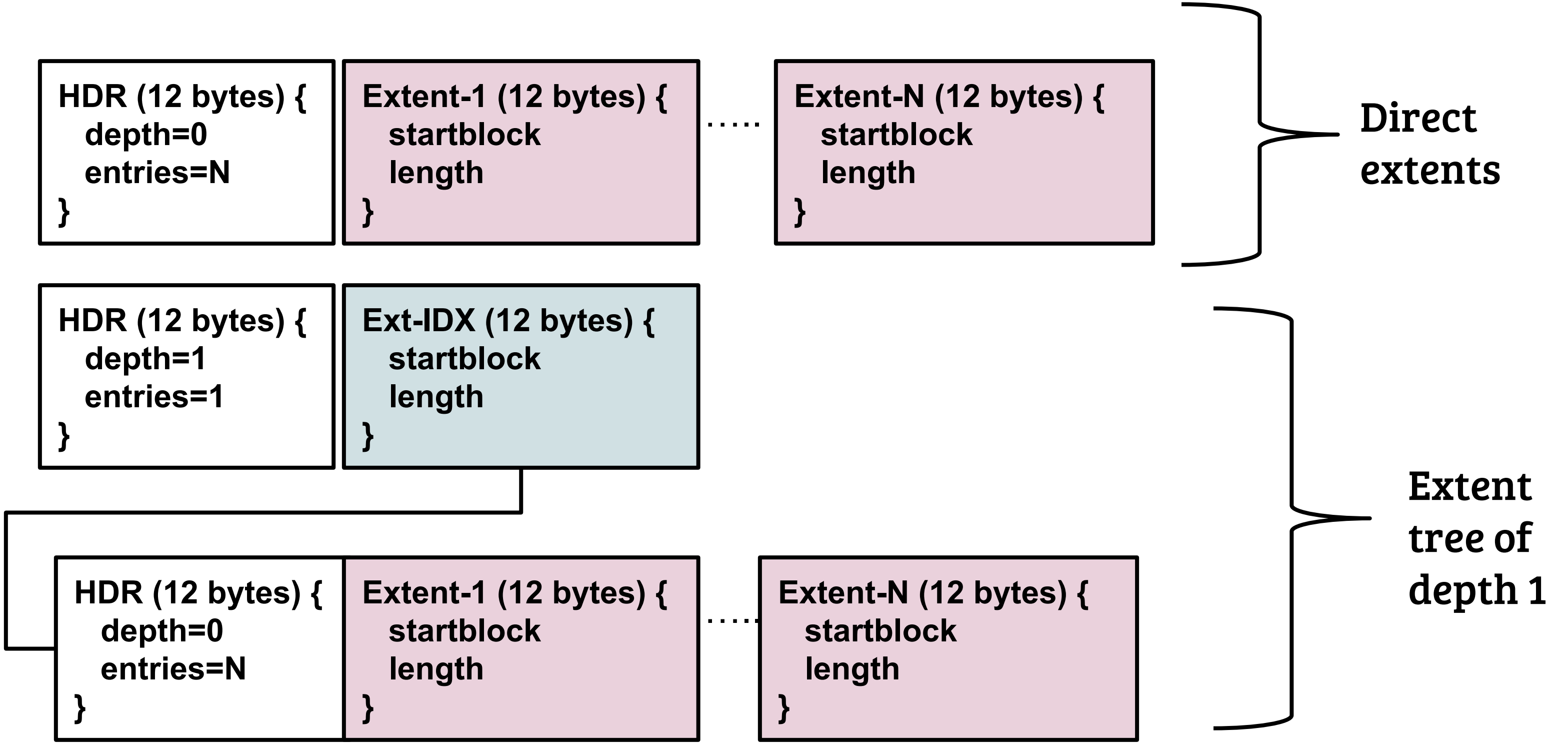
I_3

File block address (?? to ??)

Hybrid organization: pros and cons

- Fast access for small sized files
- Flexible
- Require indirect block lookups for large files
- Issue of fixed blocks
- Example: for a file size of 200 KB, One single indirect index is needed
- Alternate: Why not use {block#, length}?
- Idea: Extent tree in ext4

Ext4 extents and extent tree



Extent organization: pros and cons

- Fast access, both sequential and random
- Flexible across variety of file sizes
- Sequential read of huge files can be magnetic disk friendly
- Indirectly implements variable block size
- Example: For a file size of 200 KB, a direct extent is sufficient
- Can be equivalent to indirect indexing in the worst case

Quiz

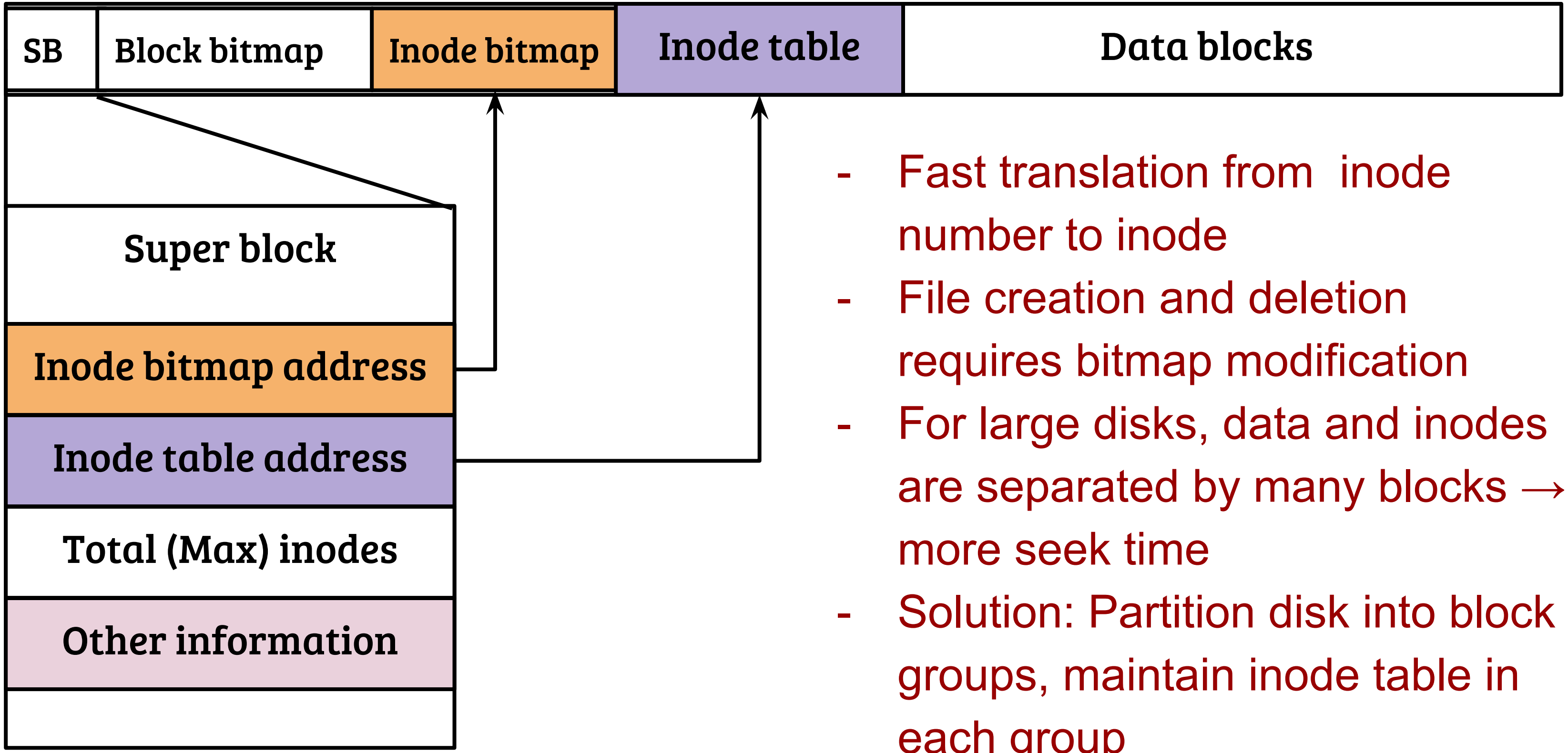
Qn. Which of the following statement(s) are true?

(a) Multiple file descriptors can correspond to the same file object
(b) Two files in your laptop can have same inode
(c) When all file objects corresponding to an inode are released, the inode must be written back to the disk
(d) In a Linux system, if a process P deletes a file while Q is using it, the file system should not allow deletion.
(e) Creation of a new file requires modification of the super block

Inode organization

- Inode maintains persistent information regarding files and directories
- Where to store the inodes?
- Most file systems identify inodes through a unique number
- Inode number → Device offset containing the inode. How?
- Fixed size vs. variable sized inodes
- Assuming fixed size inodes, any ideas?

Inode organization: static reservation



- Fast translation from inode number to inode
- File creation and deletion requires bitmap modification
- For large disks, data and inodes are separated by many blocks → more seek time
- Solution: Partition disk into block groups, maintain inode table in each group

Directory organization

- Directory is also treated as a file, represented by inode, contains pointer to data blocks, data blocks are *structured*
- Directory contains information regarding the children
- Children can be files, directories, links (soft and hard)

Operations on directory

- Search child using file/dir name
- Create new file/directory
- Delete files/directory
- Rename files/directory

Ideas on how to organize?

Directory organization: design choices

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;
        u8 name_len;
        char name[name_len];
};
```

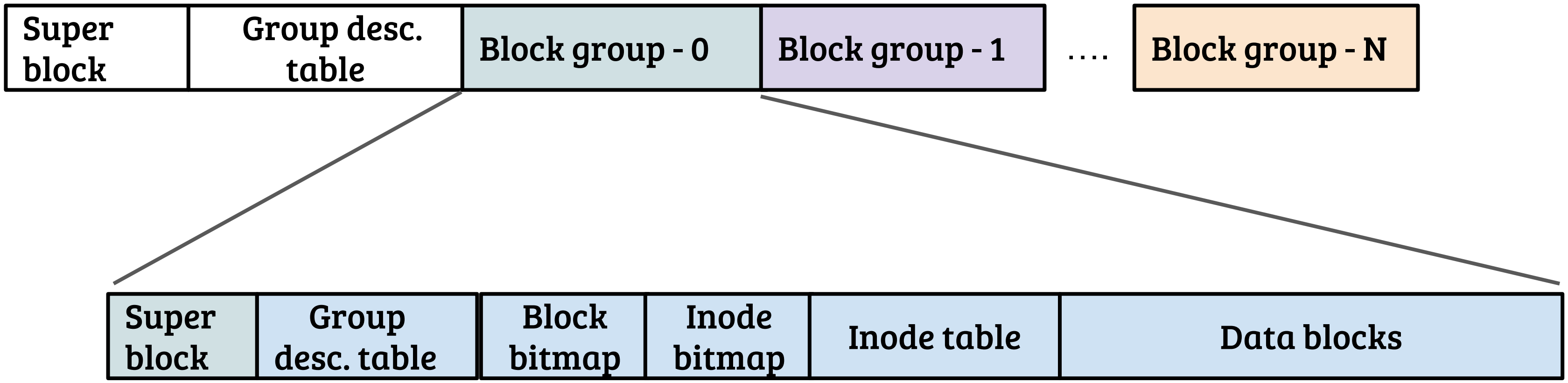
Design 1: Unsorted variable sized directory entry

- Efficiency of search, create ...

Design 2: Sorted using name as the key

- Efficiency of search, create ...

Example: Linux Ext2/3/4 file system



- Advantages?
- How inode is unique?
- Should file data blocks span across groups?
- Why superblock and block desc repeated?

Example: Read “/etc/hosts”