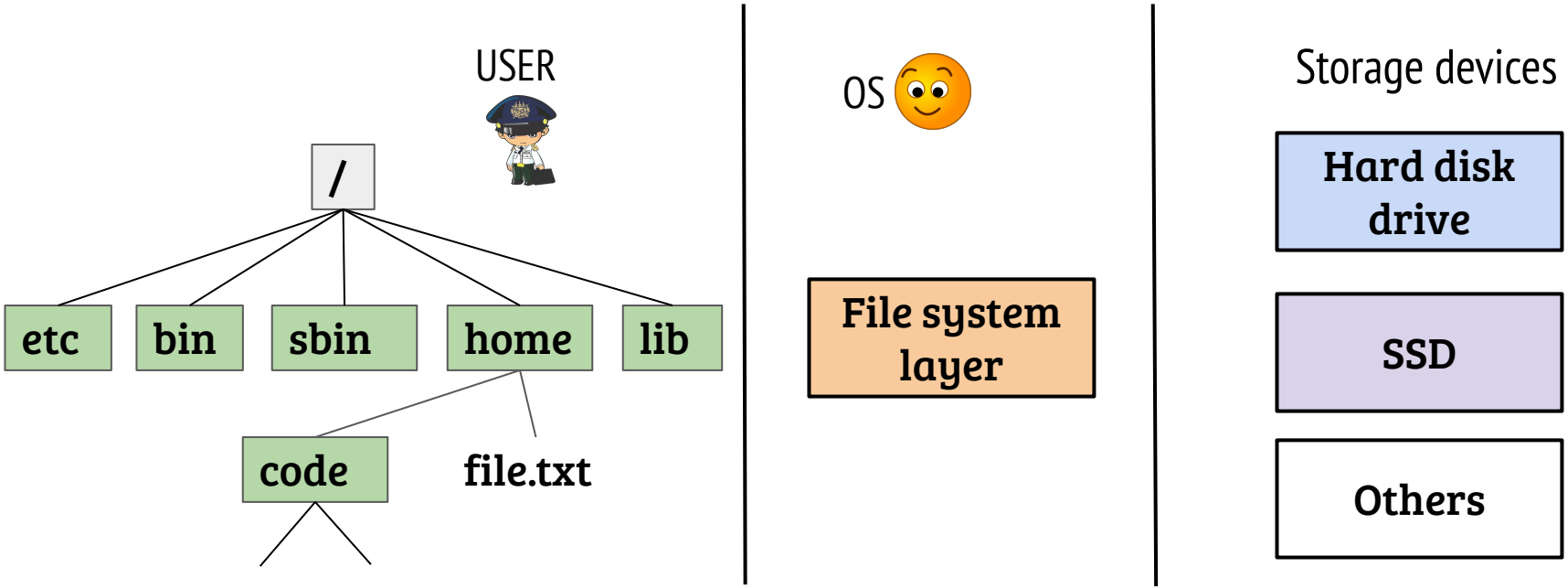
CS330: Operating Systems

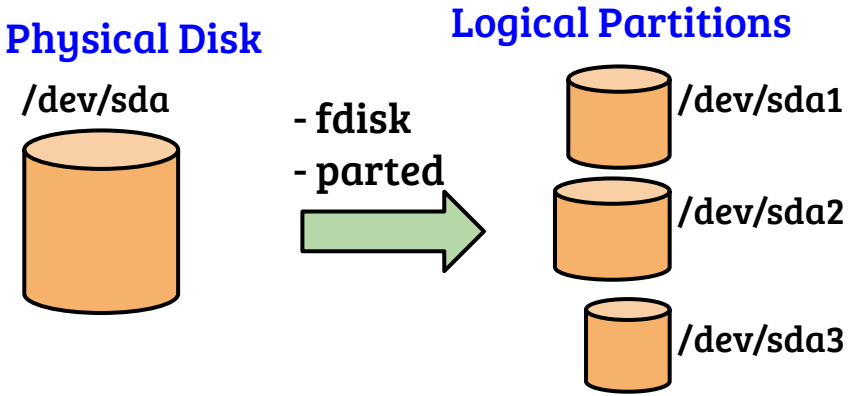
Filesystem

Recap: file system



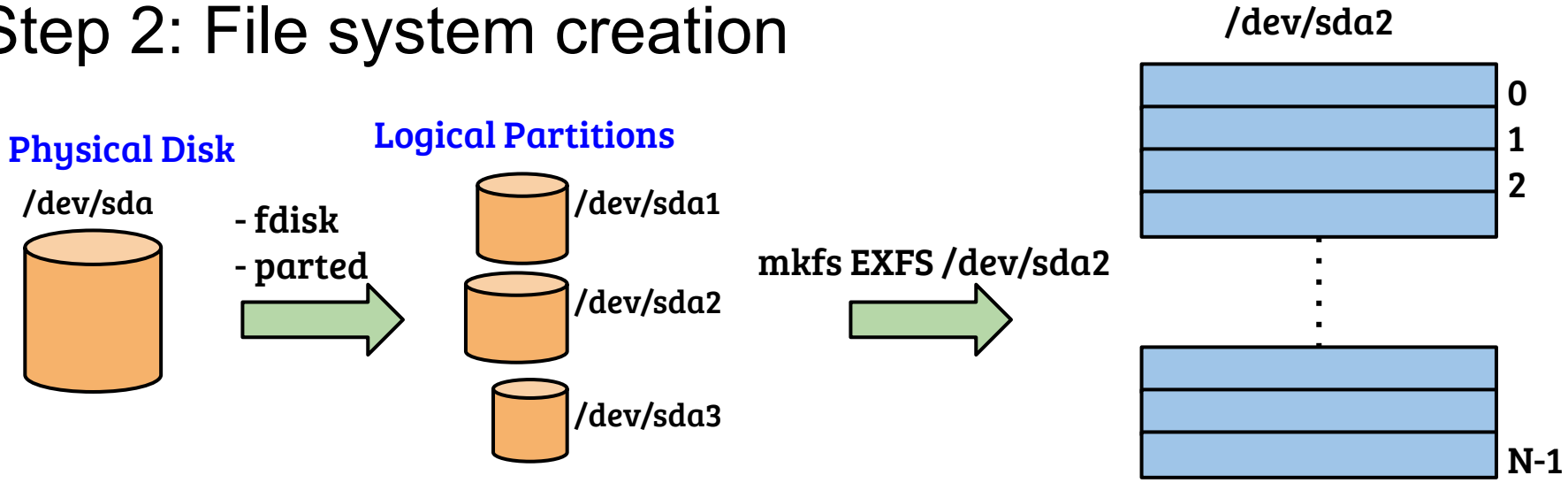
- File system is an important OS subsystem
  - Provides abstractions like files and directories
  - Hides the complexity of underlying storage devices

Step-1: Disk device partitioning



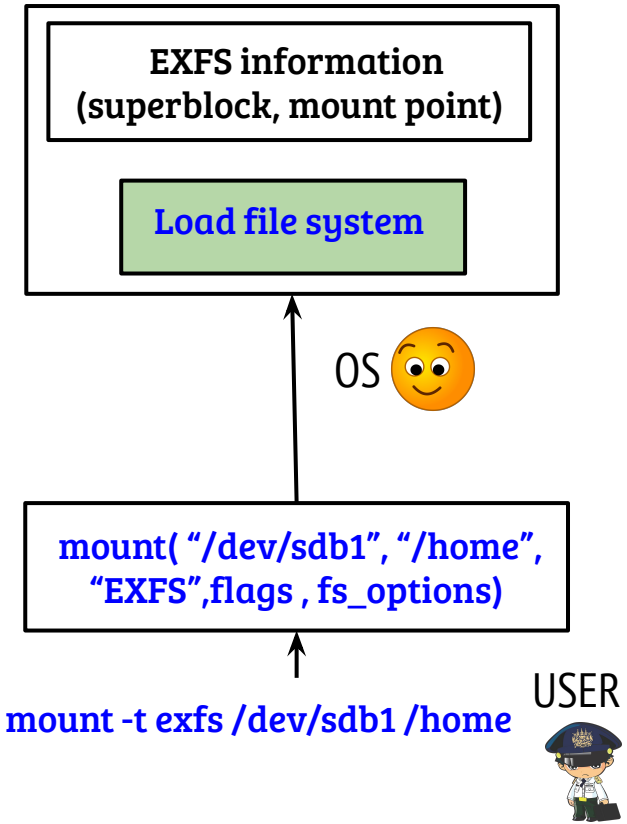
- Partition information is stored in the boot sector of the disk
- Creation of partition is the first step
  - It does not create a file system
- A file system is created on a partition to manage the physical device and present the logical view
- All file systems provide utilities to initialize file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



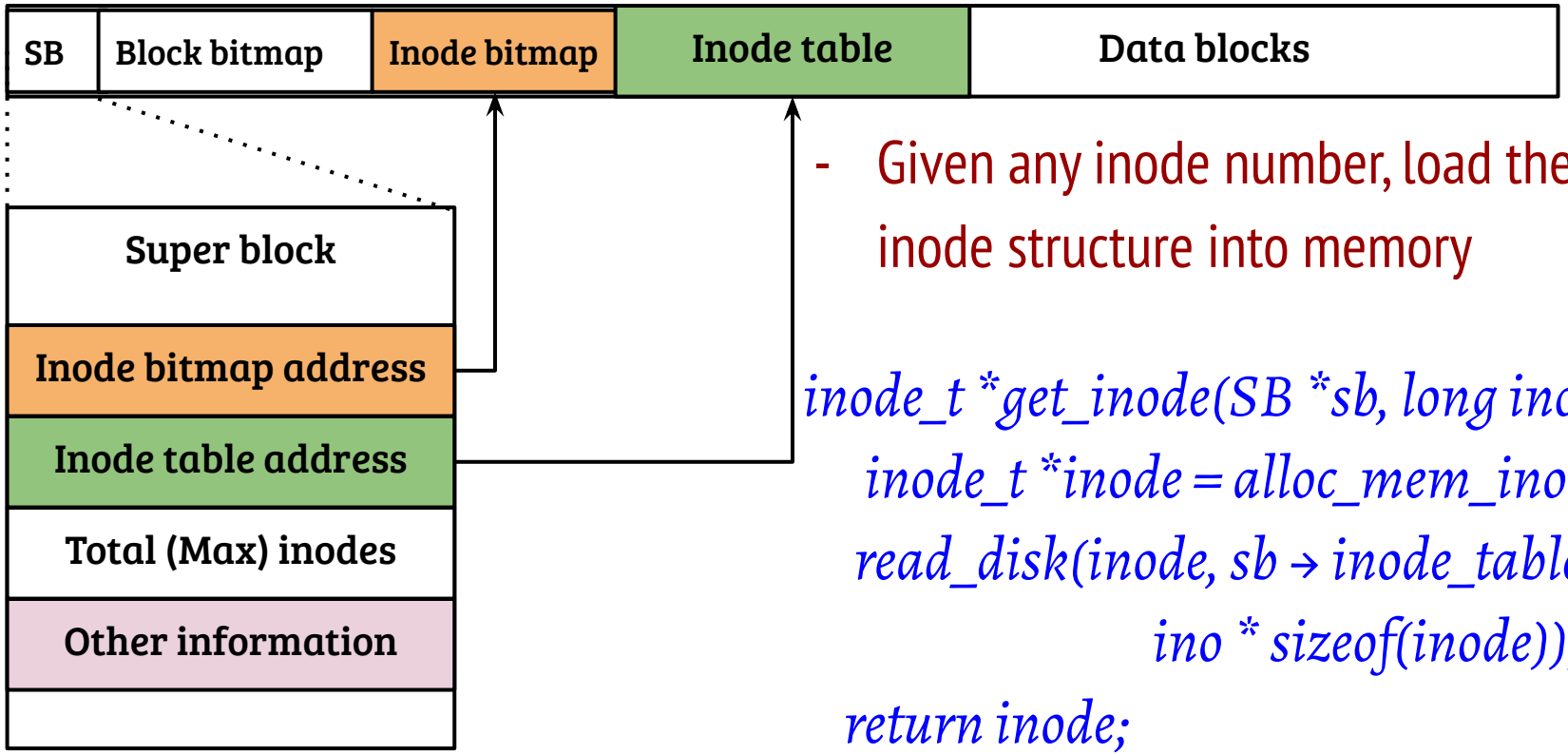
- *mount( )* associates a superblock with the file system mount point
- Example: The OS will use the superblock associated with the mount point "/home" to reach any file/dir under "/home"
- Superblock is a copy of the on-disk superblock along with other information

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

File system organization



- 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

- 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.?

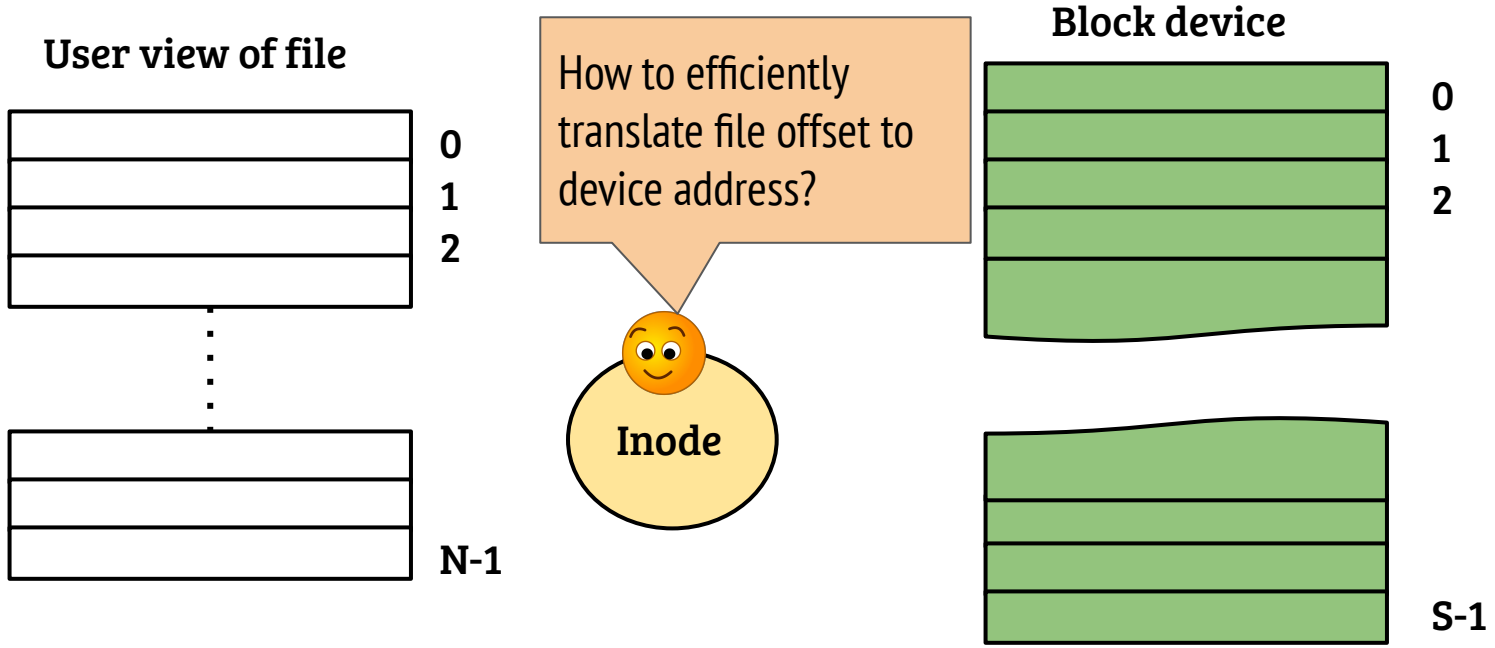
return inode;

}

Inode

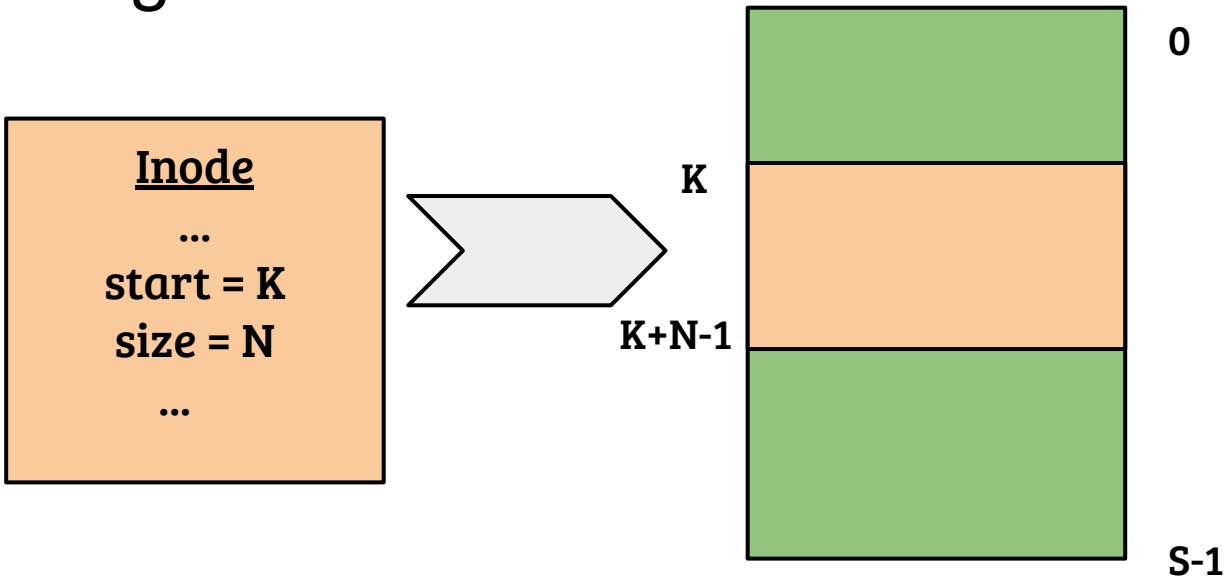
- A on-disk structure containing information regarding files/directories in the unix systems
  - Represented by a unique number in the file system (e.g., in Linux, “ls -i filename” can be used to print the inode)
  - Contains access permissions, access time, file size etc.
  - *Most importantly, inode 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)

Problem: file offset to disk address mapping



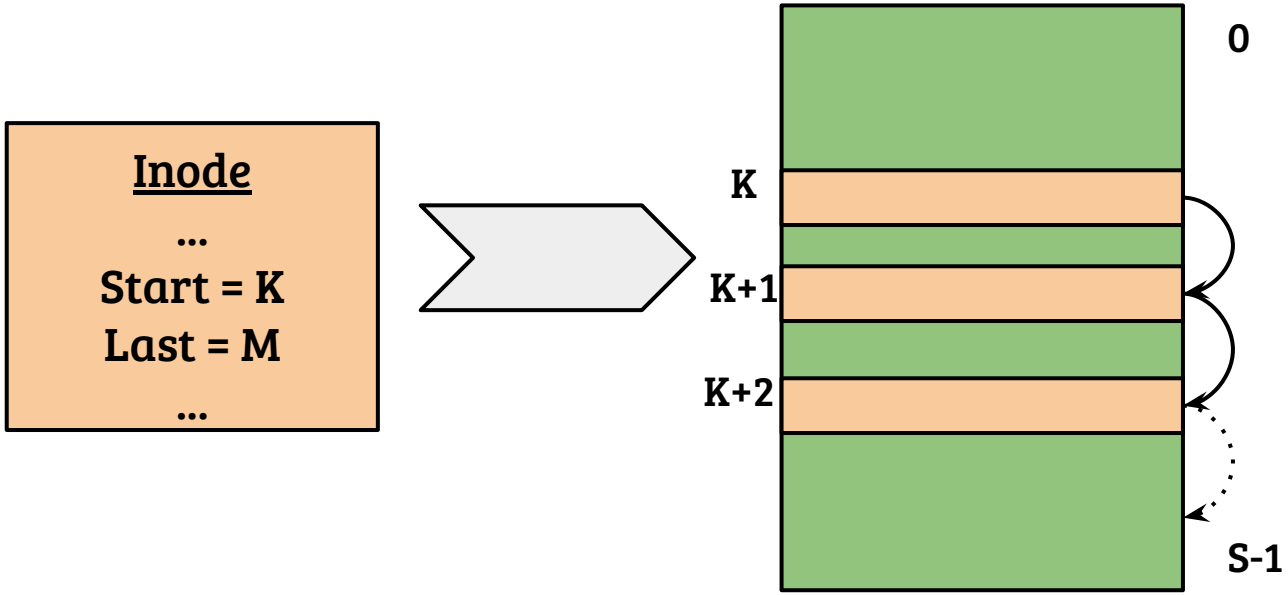
- File size can range from few bytes to gigabytes
- Can be accessed in a sequential or random manner
- How to design the mapping structure?

Contiguous allocation



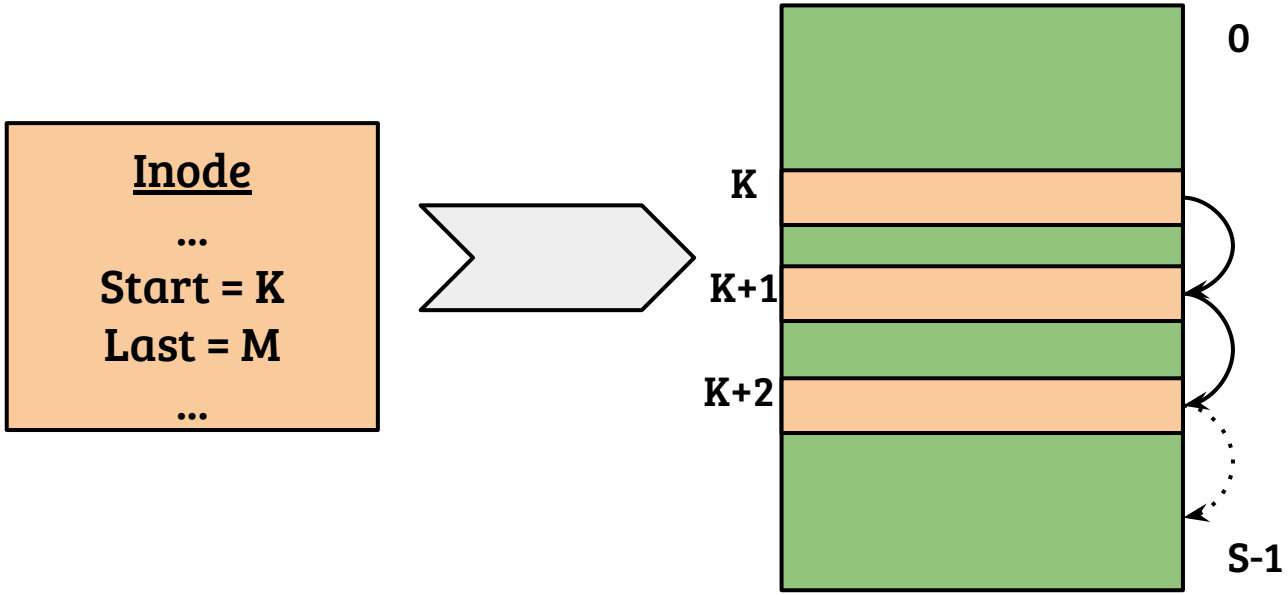
- Works nicely for both sequential and random access
- Append operation is difficult, How to expand files? Require relocation!
- External fragmentation is a concern

Linked allocation



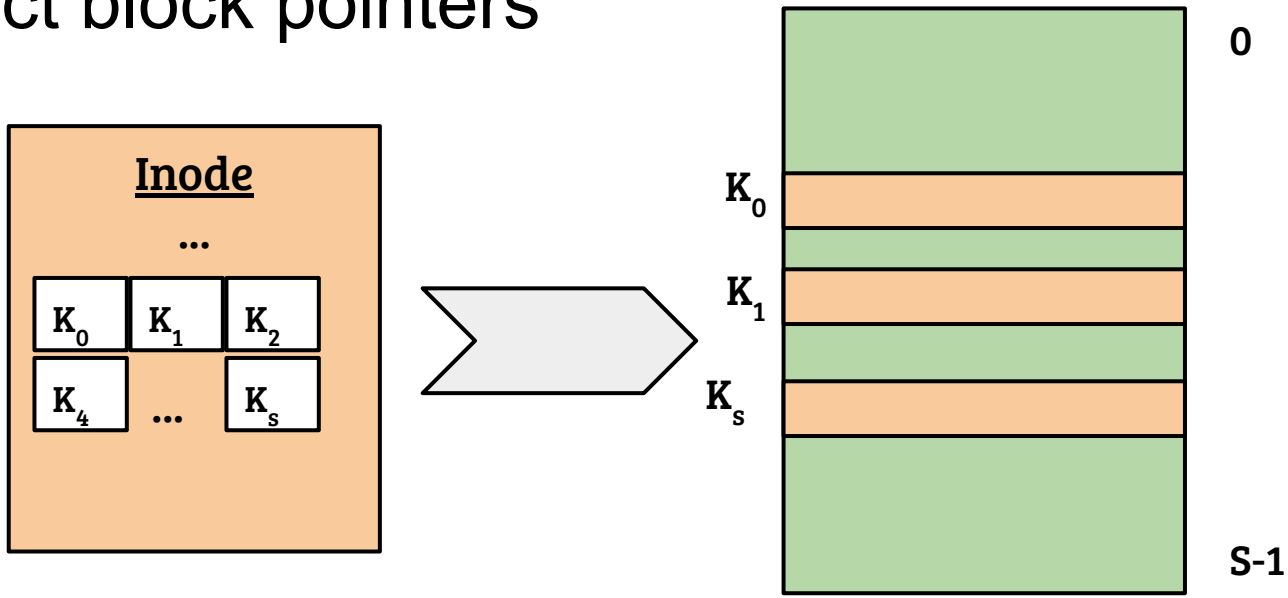
- Every block contains pointer to next block
- Advantage: flexible, easy to grow and shrink, Disadvantage: random access
- Why maintain last block not size?

Linked allocation



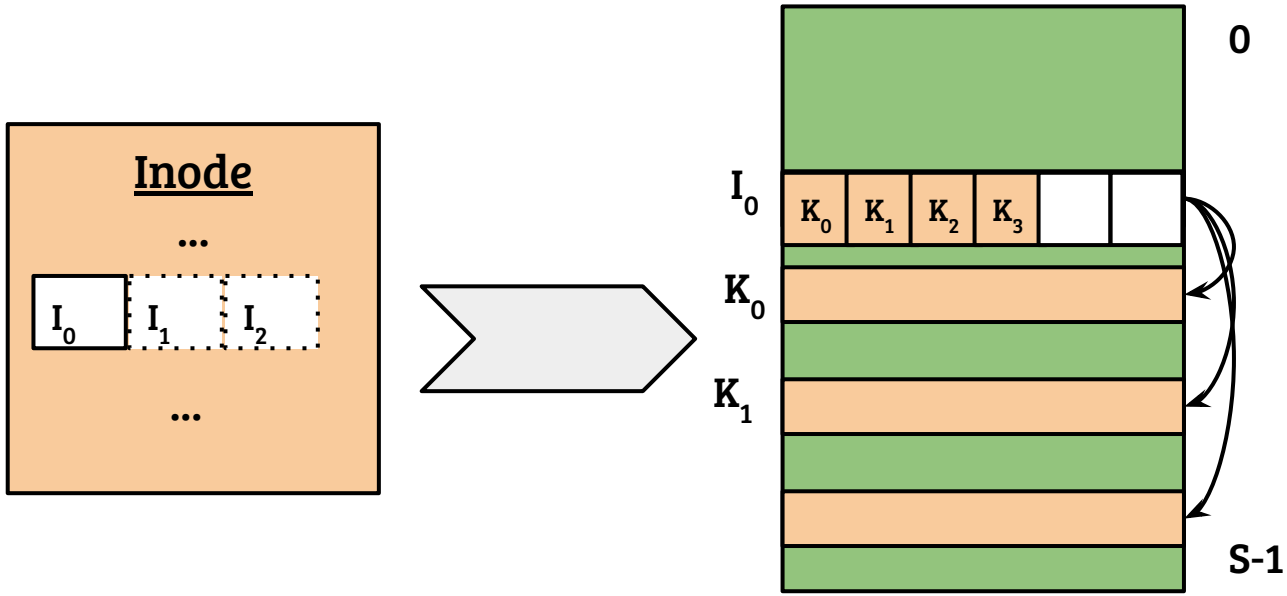
- Every block contains pointer to next block
- Advantage: flexible, easy to grow and shrink, Disadvantage: random access
- Why maintain last block not size? Efficient append operation!

Direct block pointers



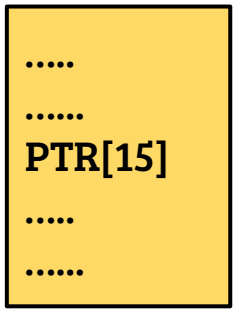
- Inode contains direct pointers to the block
- Flexible: growth, shrink, random access is good
- Can not support files of larger size!

Indirect block pointers



- Inode contains the pointers to a block containing pointers to data blocks
- Advantages: flexible, random access is good
- Disadvantages: Indirect block access overheads (even for small files)

Hybrid block pointers: Ext2 file system



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

File block address (0 -11)

Single indirect {PTR [12]}

File block address (12 -1035)

Double indirect {PTR [13]}

File block address (1036 to 1049611)

Triple indirect {PTR [14]}

File block address (?? to ??)

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: efficient new entry creation, rename
- Disadvantages: space wastage

Organizing the directory content

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: inefficient create, rename

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

CS330: Operating Systems

Filesystem: caching and consistency

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

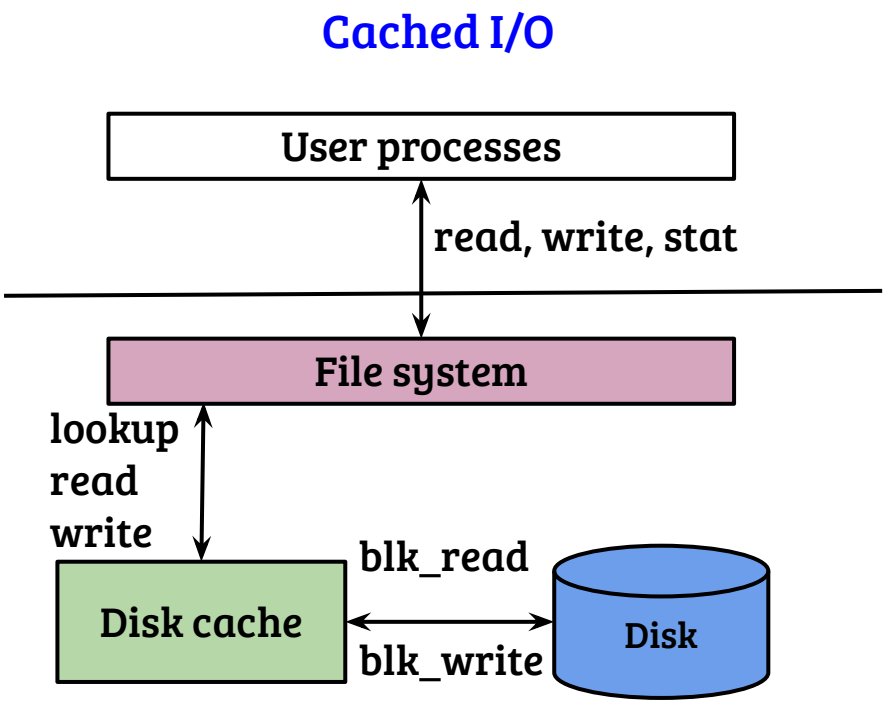
/home/user$ ls

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?

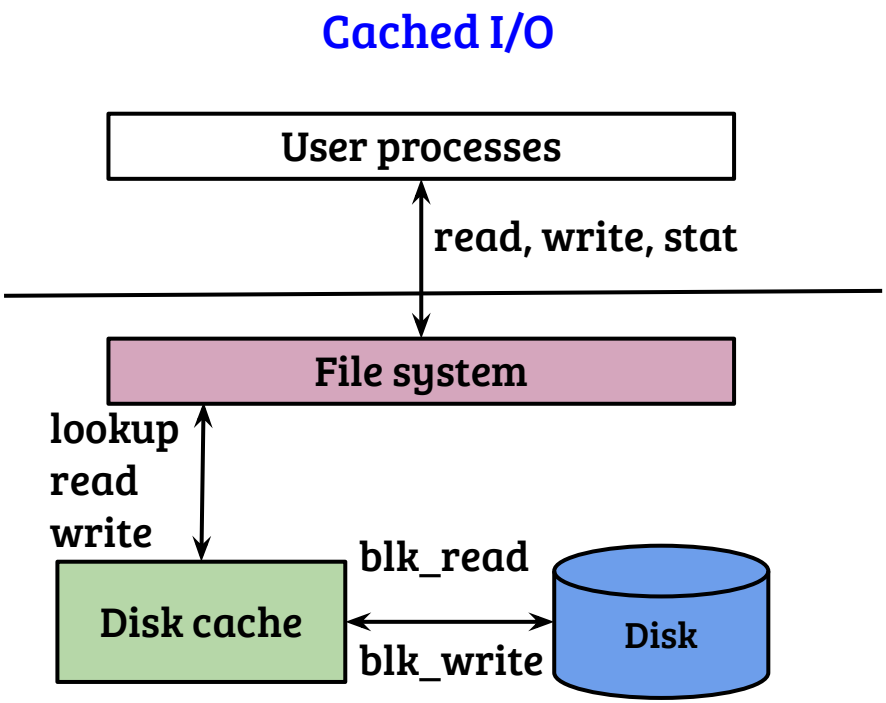
/home/user$ ls

Block layer caching



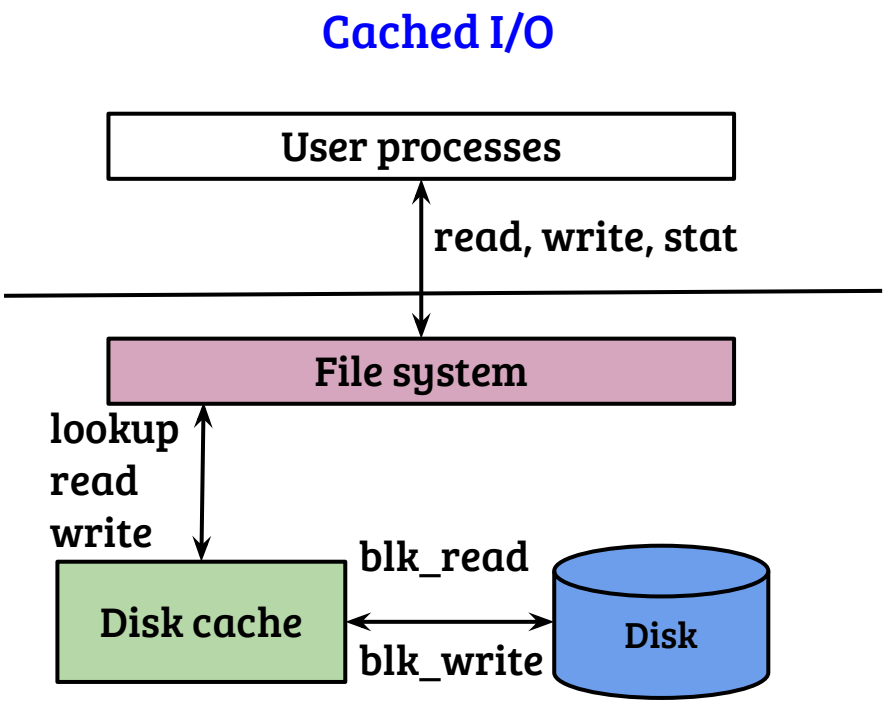
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?

Block layer caching



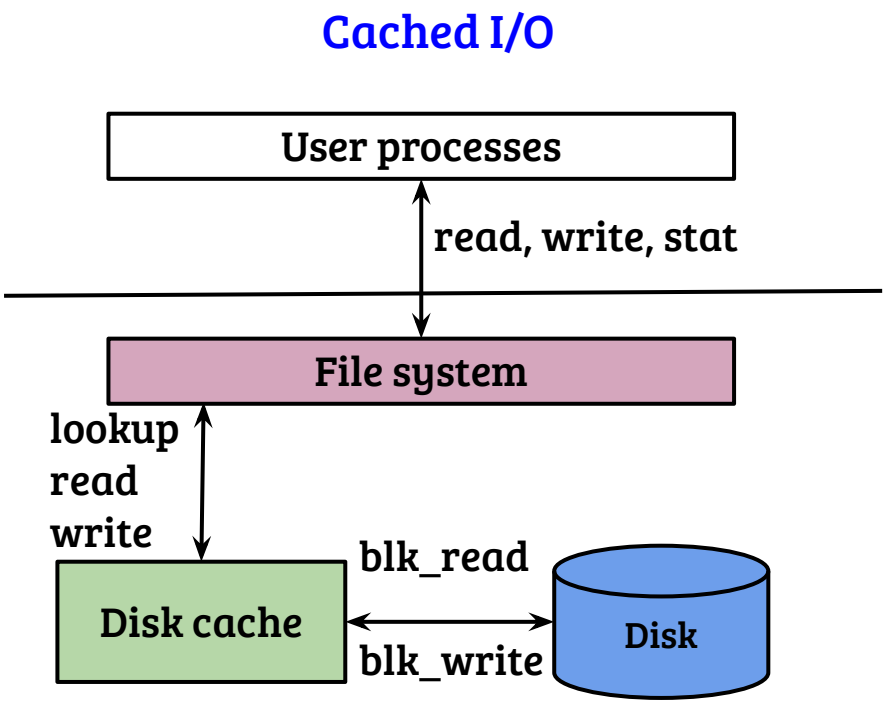
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache

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!)

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?

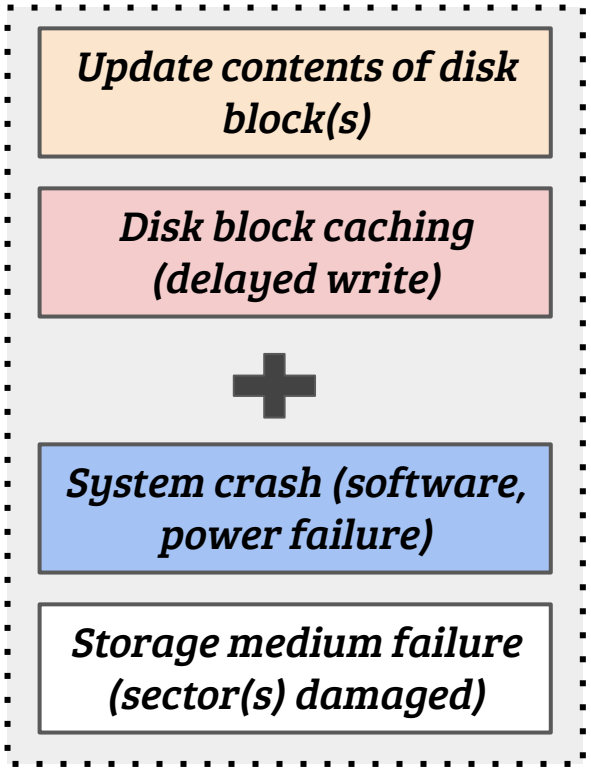
Caching and consistency

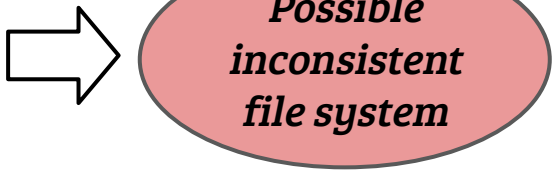
- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
  - Example-1: If a write( ) system call is successful, data must be written
  - Example-2: If a file creation is successful then, file is created.
  - Difficult to achieve with asynchronous I/O

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

File system inconsistency: root causes





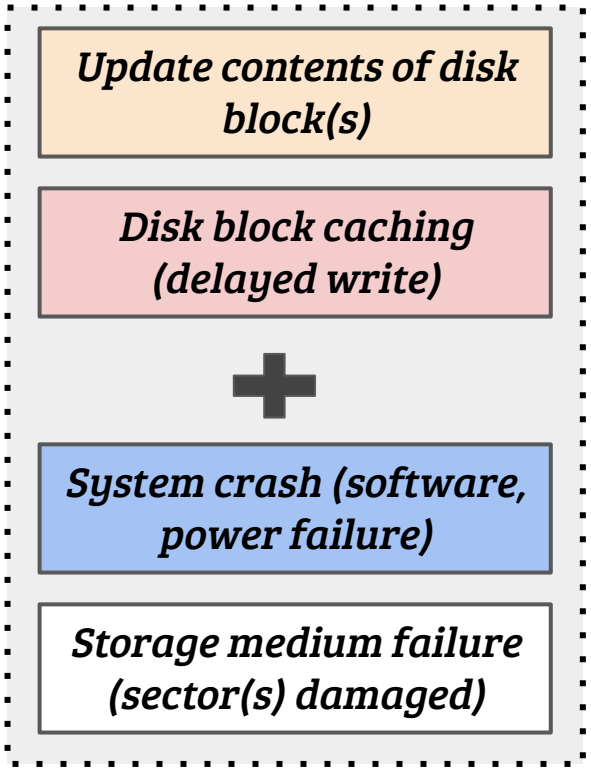
- No consistency issues if user operation translates to read-only operations on the disk blocks

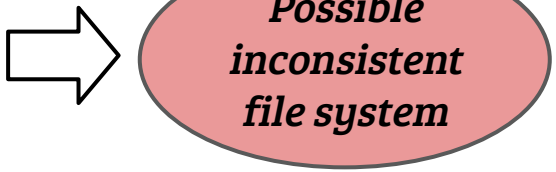
- Always keep in mind: device level atomicity guarantees

CS330: Operating Systems

Filesystem: consistency

Recap: File system inconsistency

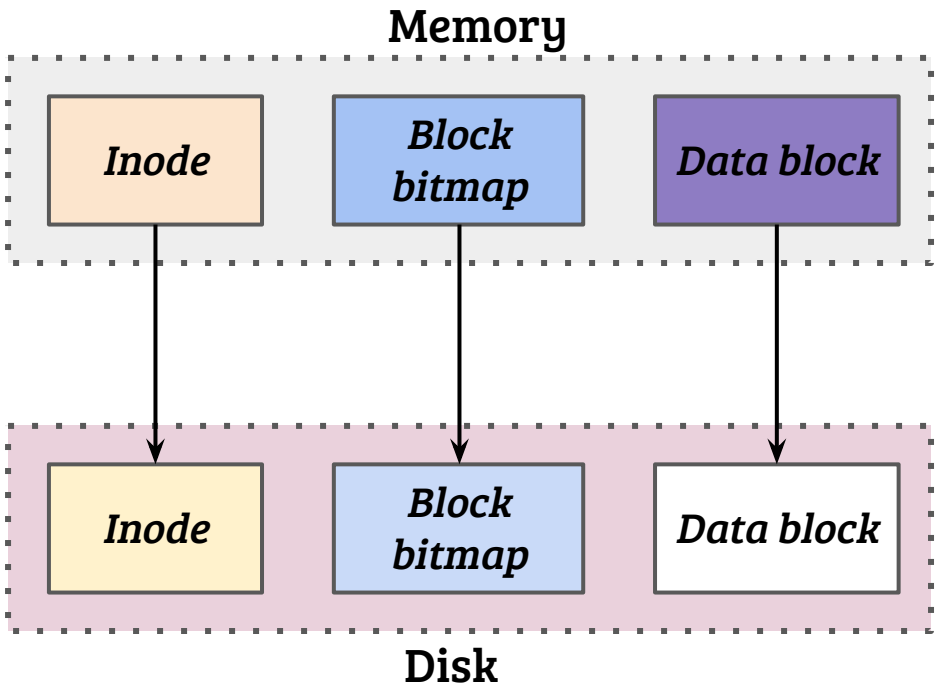




- No consistency issues if user operation translates to read-only operations on the disk blocks
- Device level atomicity may impact file system consistency

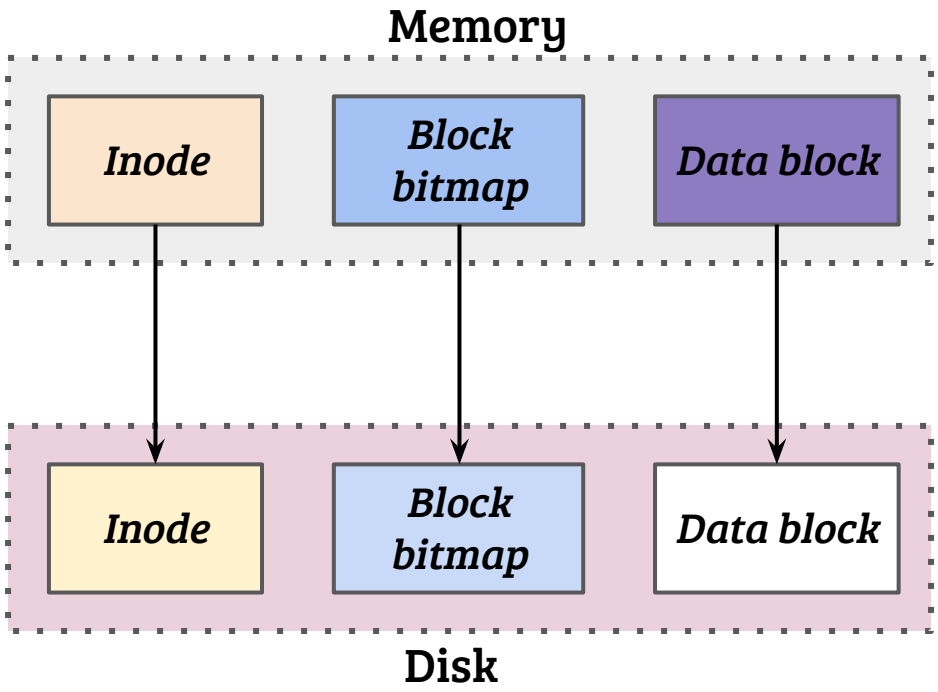
Example: Append to a file

- Steps: (i) seek to the end of file, (ii) allocate a new block, (iii) write user data
- Inode modifications: size and block pointers
- Block bitmap update: set used block bit for the newly allocated block(s)
- Data update: data block content is updated



Example: Append to a file

- Steps: (i) seek to the end of file, (ii) allocate a new block, (iii) write user data
- Inode modifications: size and block pointers
- Block bitmap update: set used block bit for the newly allocated block(s)
- Data update: data block content is updated



Three write operations reqd. to complete the operation, what if some of them are incomplete?

Failure scenarios and implications

Written	Yet to be written	Implications
Data block	Inode, Block bitmap	File system is consistent (Lost data)
Inode	Block bitmap, Data block	File system is inconsistent (correctness issues)
Block bitmap	Inode, Data block	File system is inconsistent (space leakage)

- All failure scenarios may not result in consistency issues!

Failure scenarios and implications

Written	Yet to be written	Implications
Data block, Block bitmap	Inode	File system is inconsistent (space leakage)
Inode, Data block	Block bitmap	File system is inconsistent (correctness issues)
Inode, Block bitmap	Data block	File system is consistent (Incorrect data)

- Careful ordering of operations may reduce the risk of inconsistency
- But, how to ensure correctness?

File system consistency with *fsck*

- Strategy: Do not worry about consistency, recover after abrupt failures
- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?

File system consistency with *fsck*

- Strategy: Do not worry about consistency, recover after abrupt failures
- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?
  - Maintain the last unmount information on superblock

File system consistency with *fsck*

- Strategy: Do not worry about consistency, recover after abrupt failures
- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?
  - Maintain the last unmount information on superblock
- If the FS was not cleanly unmounted, perform sanity checks at different levels: *superblock*, *block bitmap*, *inode*, *directory content*

File system consistency with *fsck*

- Strategy: Do not worry about consistency, recover after abrupt failures
- During FS mount, check if it had been cleanly unmounted when it was last used, How to know?
  - Maintain the last unmount information on superblock
- If the FS was not cleanly unmounted, perform sanity checks at different levels: *superblock*, *block bitmap*, *inode*, *directory content*
- Sanity checks and verifying invariants across metadata. Examples,
  - Block bitmap vs. Inode block pointers
  - Used inodes vs. directory content

File system consistency with *journaling*

- Idea: Before the actual operation, note down the operations in some special blocks (known as journal)
- Journal entry for append operation: *[Start] [Inode block] [Block bitmap] [Data block] [End]*
- When the FS is updated (in a delayed manner) and mark the journal entry as completed (also known as *checkpoint)*

File system consistency with *journaling*

- Idea: Before the actual operation, note down the operations in some special blocks (known as journal)
- Journal entry for append operation: *[Start] [Inode block] [Block bitmap] [Data block] [End]*
- When the FS is updated (in a delayed manner) and mark the journal entry as completed (also known as *checkpoint)*
- Recovery mechanism: incomplete journal entries inspected during the next mount and operations are re-performed

File system consistency with *journaling*

- Idea: Before the actual operation, note down the operations in some special

  - Journal write should not only be synchronous but also performed in the specified order
  - Failure after updating some blocks and rewritten during recovery is not an issue as the data is consistent at the end
  - Failures during journal write is not a problem w.r.t. file system consistency

  mount and operations are re-performed

Metadata journaling: performance-reliability tradeoff

- Journaling comes with a performance penalty, especially for maintaining the data in the journal
- Metadata journaling: data block is not part of the journal entry
- Practical with tolerable performance overheads
- Example journal entry for append: *[Start] [Inode block] [Block bitmap] [End]*
- Strategy: First write the data block (to disk) followed by the journal write and metadata commit afterwards, Why?

Metadata journaling: performance-reliability tradeoff

- Journaling comes with a performance penalty, especially for maintaining the data in the journal
- Metadata journaling: data block is not part of the journal entry
- Practical with tolerable performance overheads
- Example journal entry for append: *[Start] [Inode block] [Block bitmap] [End]*
- Strategy: First write the data block (to disk) followed by the journal write and metadata commit afterwards, Why?
  - If the metadata blocks are not written, FS can be recovered
  - If journal write fails, a write is lost (syscall semantic broken)