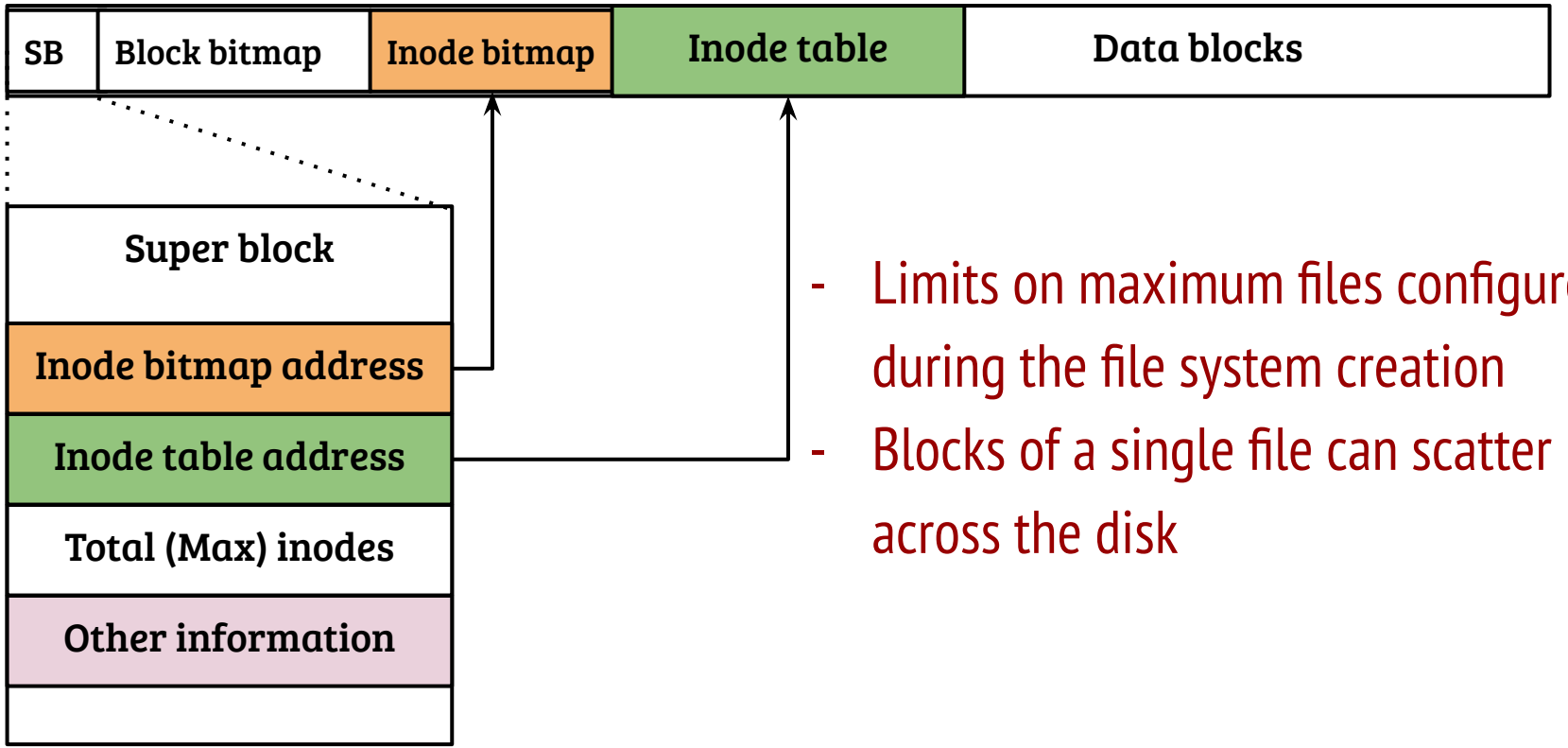
# CS614: Linux Kernel Programming

## Ext4 File System

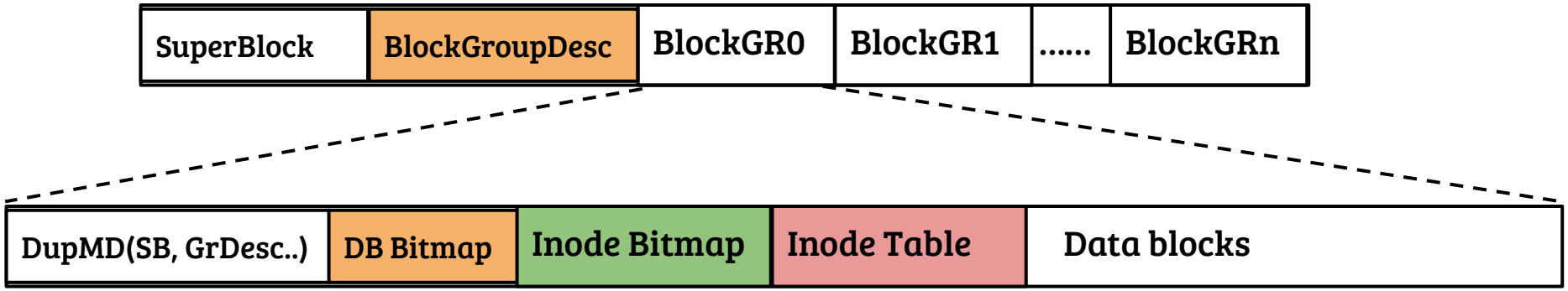
Debadatta Mishra, CSE, IIT Kanpur

# Recap: Basic file system organization (on-disk)



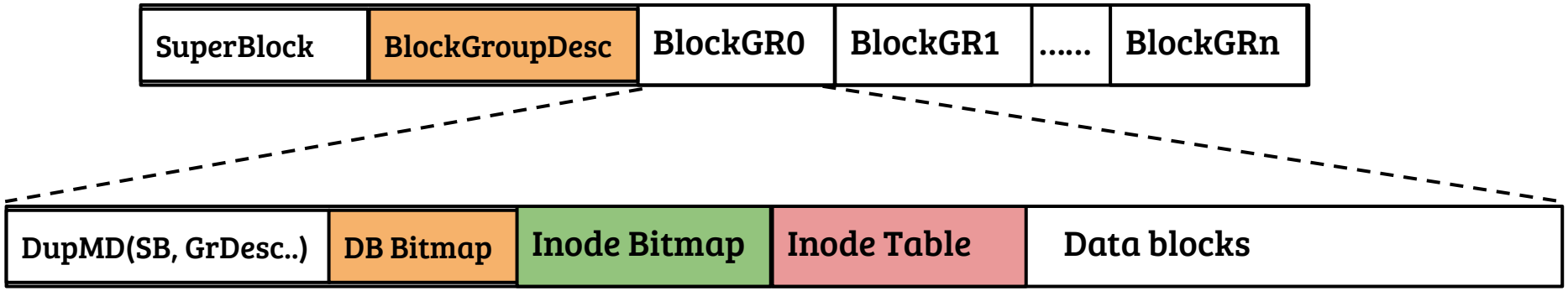

- Limits on maximum files configured during the file system creation
- Blocks of a single file can scatter across the disk

# Ext4 block groups



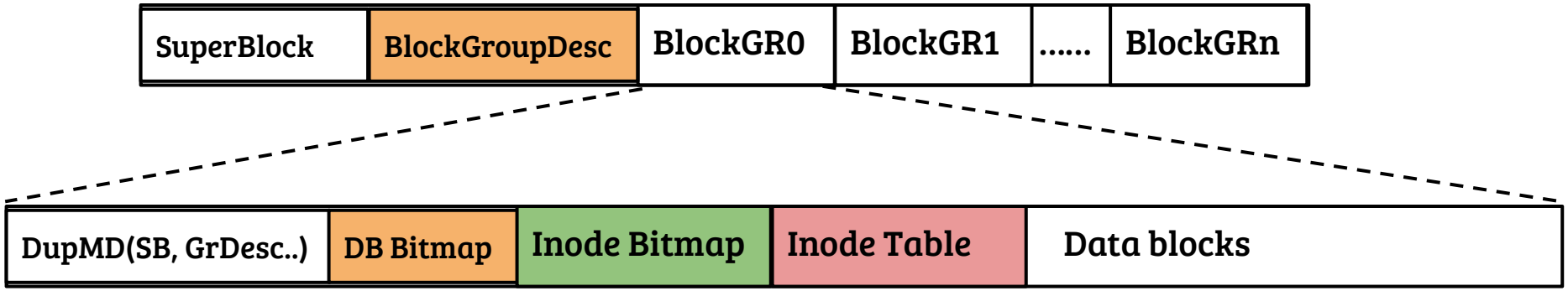

- Ext4 organizes the logical partition into a series of block groups
- Each block group has its block bitmap and inode bitmap
- Superblock contains information regarding the location of block groups
- To reduce seek time, the FS tries to store the blocks of a given file in a single block group

# Ext4 block groups



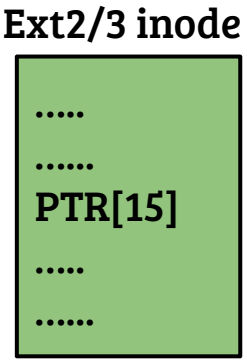

- How to locate the on-disk inode given an inode number?

# Ext4 block groups




- How to locate the on-disk inode given an inode number?
  - SB maintains 'inodes per block group'
  - Calculate the group descriptor ⇒ Check the inode bitmap
  - If present, read it from the table

# Recap: Ext2 file system indexing




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 ??)

# Hybrid organization: pros and cons

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

# Ext4 extents [1]

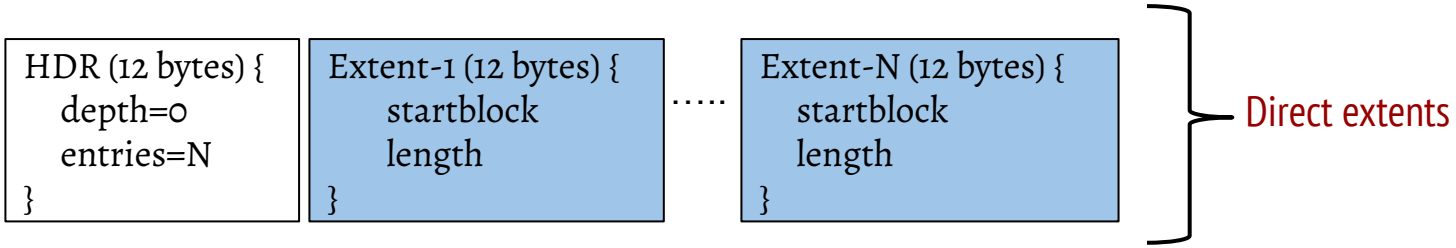
```
struct ext4_extent_header {
    u16  eh_magic;      // Fixed magic:0xF30A.
    u16  eh_entries;    // Number of valid entries
    u16  eh_max;        // Max entries that can follow header
    u16  eh_depth;      // 0 ⇒ direct,  >=1 ⇒ More levels
    u32  eh_generation; // unused for ext4
};
```

```
struct ext4_extent {
    u32  ee_block;      // First logical (file) block extent covers
    u16  ee_len;        // Number of blocks covered by extent
    u16  ee_start_hi;   // High 16 bits of physical block
    u32  ee_start_lo;   // Low 32 bits of physical block
};
```

- Every node of the extent tree starts with the header
- The header can be followed by 'extents' (at leaf-level) or by indirections to the next-level (extent_idx)
- Depth refers to the depth of the extent tree

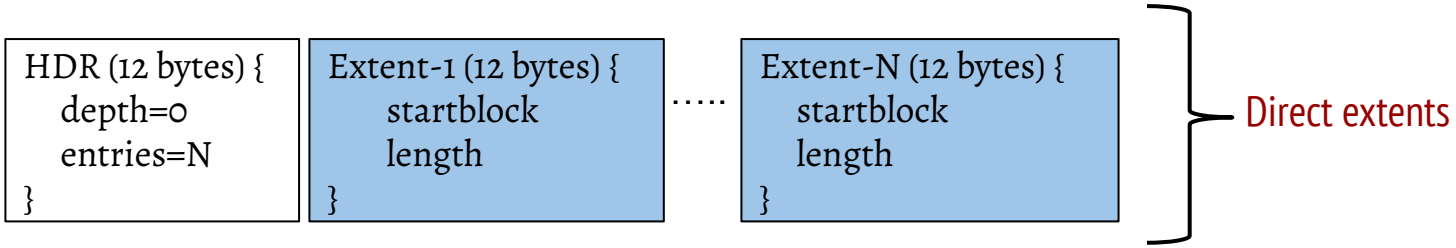
1. Ext4 Disk Layout: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

# Direct extents




- Root node of the extent is also the leaf node
- Sixty bytes of block index (in the inode) : One header and Four extents
- How to map a logical file offset to a block address?
- What is the maximum file size supported?
- What is the minimum file size supported?

# Direct extents




- Root node of the extent is also the leaf node
- Sixty bytes of block index (in the inode) : One header and Four extents
- How to map a logical file offset to a block address? Compare offset with 'startblock' (+use length) to locate the extent entry containing the mapping
- What is the maximum file size supported? 4 * 4KB * 32KB
- What is the minimum file size supported? 4 * 4KB

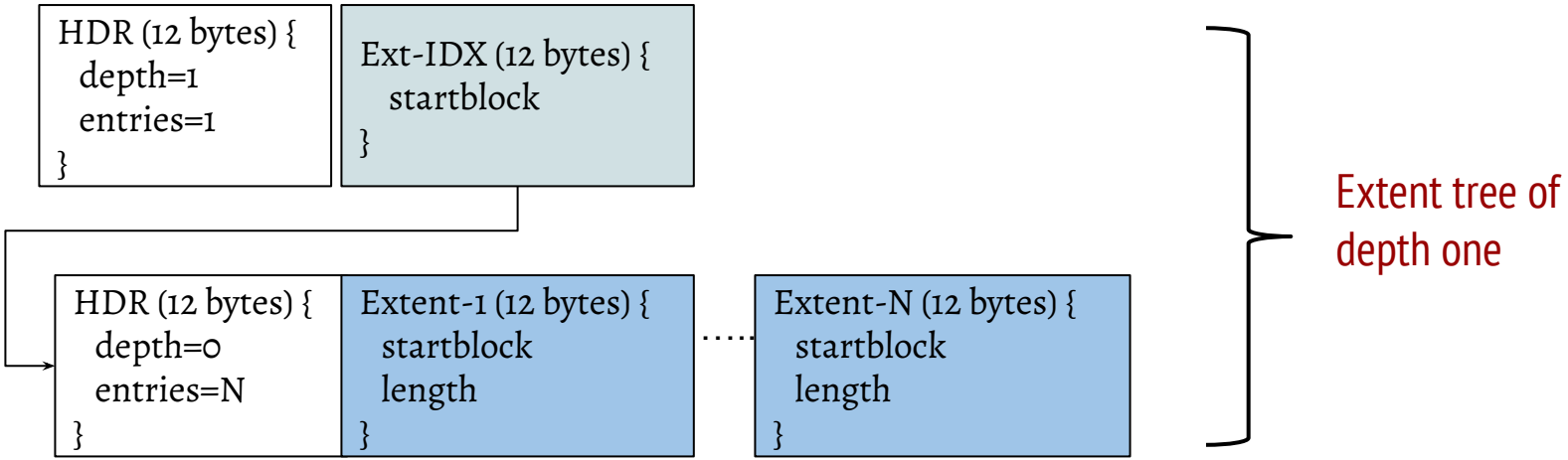
# Ext4 indirect extents and extent tree

```
struct ext4_extent_header {
      u16  eh_magic;      // Fixed magic:0xF30A.
      u16  eh_entries;    // Number of valid entries
      u16  eh_max;        // Max entries that can follow header
      u16  eh_depth;      // 0 ⇒ direct,  >=1 ⇒ More levels
      u32  eh_generation; // unused for ext4
};
```

```
struct ext4_extent_idx {
      u32  ei_block;      //First logical (file) block the subtree covers
      u32  ei_leaf_lo;    //Low 32 bits of physical block
      u16  ei_leaf_hi;    // High 16 bits of physical block
      u16   ei_unused;
};
```

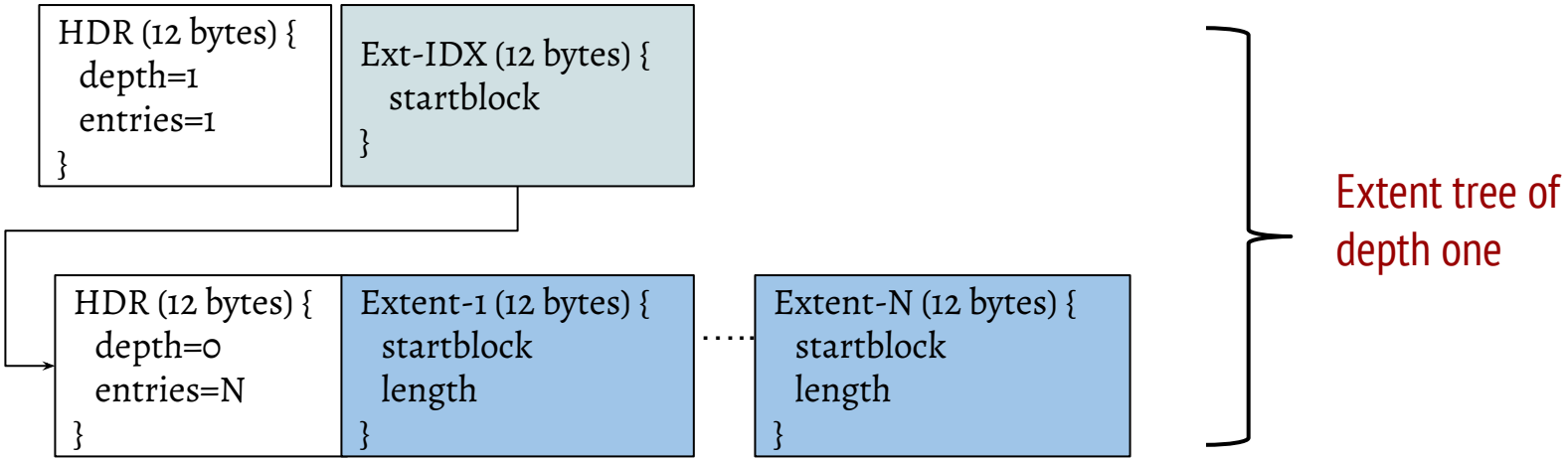
- The IDX structure specifies a block containing the information regarding the next-level node of the tree
- An extent header at the beginning of the specified block determines how to navigate next
- Maximum depth is five

# Ext4 extent tree




- How to map a file offset to a block?
- What is the maximum and minimum size supported by an extent tree of height one?

# Ext4 extent tree

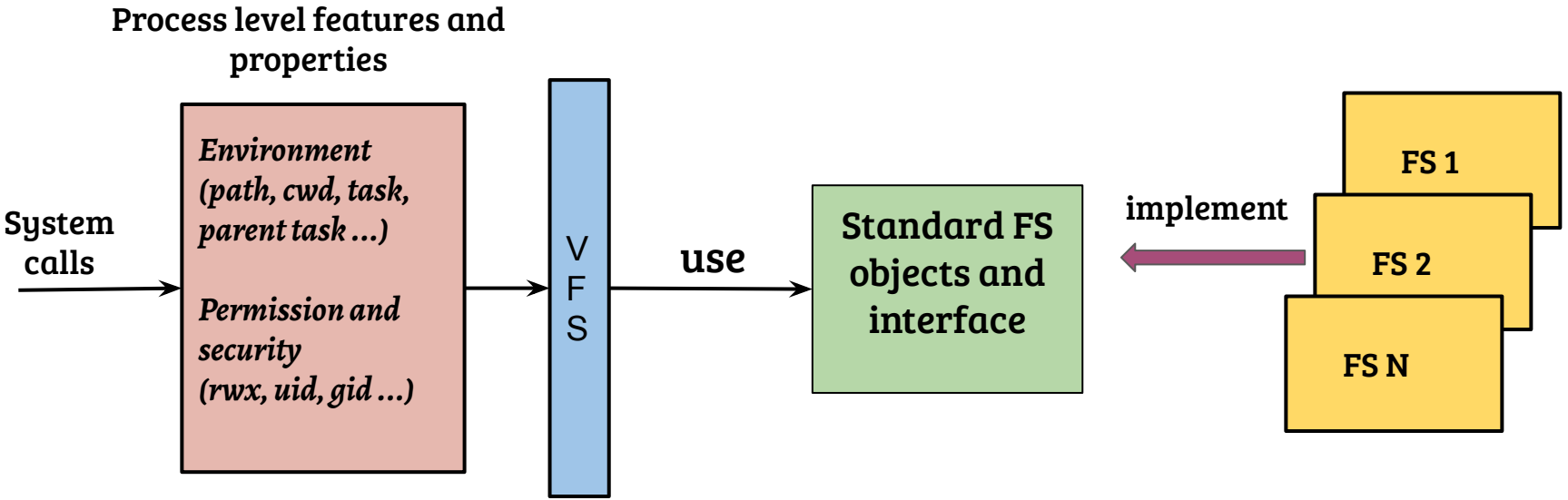



- How to map a file offset to a block? At any non-leaf level navigate to the next level comparing the offset with the IDX entries
- What is the maximum and minimum size supported by an extent tree of height one? (Homework)

# Extent organization: Pros and Cons

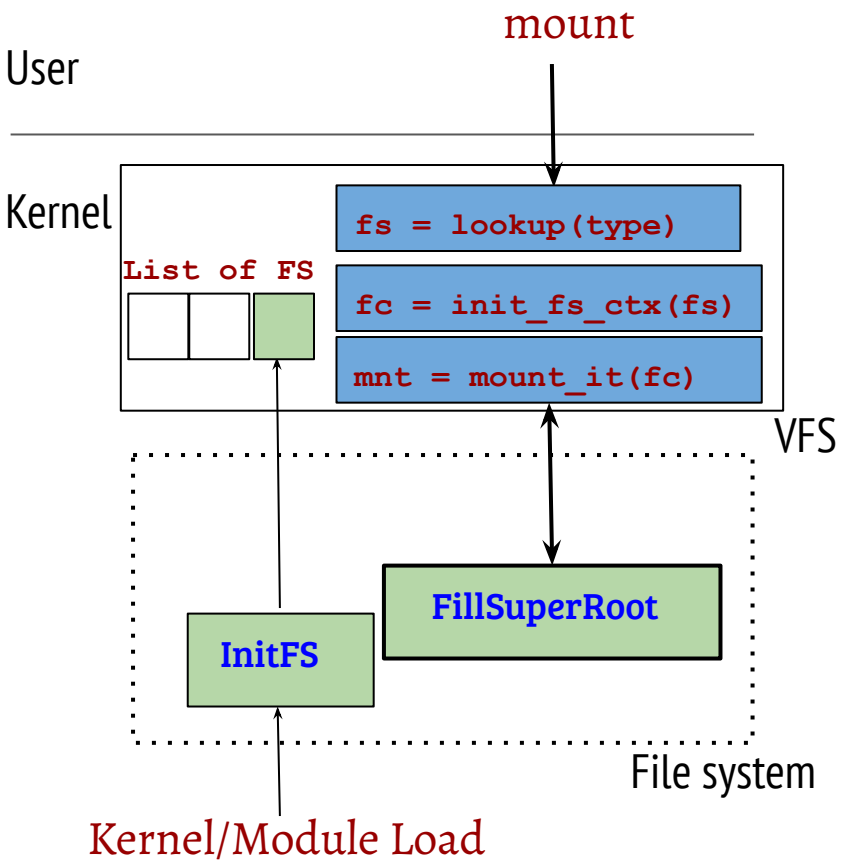
- Fast access because of reduced meta-data, both sequential and random
- Flexible across variety of file sizes
- Sequential read of huge files can be disk friendly
- Indirectly implements variable block size
- Example: For a file size of 200 KB, a direct extent is sufficient
- Can be equivalent to indirect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

# Recap: 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

# Recap: 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?

# Objects and Interfaces: Superblock

- VFS layer super block “struct super_block”, reference to Ext4 superblock “struct ext4_sb_info”, cross reference  using “sb⟶s_fs_info”
- Super block operations filled using “ext4_sops” stucture
  - Important operations: allocation and free of inodes

# Objects and Interfaces: Superblock

- VFS layer super block “struct super_block”, reference to Ext4 superblock “struct ext4_sb_info”, cross reference  using “sb⟶s_fs_info”
- Super block operations filled using “ext4_sops” stucture
  - Important operations: allocation and free of inodes
- Mount: Crucial function “ext4_fill_super”, some important operations
  - Allocate Ext4 superblock, load it using blkdev interfaces (buffer head)
  - Initialize group descriptors
  - Load root inode (ext4_iget, EXT4_ROOT_INO = 2), make VFS dentry
  - Setup operations and cross references

# Objects and Interfaces: Inode

- VFS layer inode “struct inode”, reference to Ext4 in-memory inode “struct ext4_inode_info” ← “struct ext4_inode”, cross reference between VFS inode and ext4_inode_info (VFS inode is contained in ext4_inode_info)
  - Multiple in-memory caches: VFS inode cache,  raw ext4 inode cache

# Objects and Interfaces: Inode

- VFS layer inode “struct inode”, reference to Ext4 in-memory inode “struct ext4_inode_info” ← “struct ext4_inode”, cross reference between VFS inode and ext4_inode_info (VFS inode is contained in ext4_inode_info)
  - Multiple in-memory caches: VFS inode cache,  raw ext4 inode cache
- Operations at inode level are set using VFS inode “i_op” and “i_fop” (using ext4_{file|dir}_operations and ext4_{file|dir}_inode_operations)
  - Important file operations: read_iter, write_iter, mmap etc.
  - Important DIR operations: readdir, sync etc.
  - Inode operations for files are attributed to file system specific ops (extended attributed in ext4, see man ‘xattr’)

# Objects and Interfaces: dentry

- Represents an element in a file system path, Ext4 does not have anything similar. On-disk directory entry is the basis to create dentry
- Important members of “struct dentry”
  - d_parent (parent dentry), d_name, d_inode (ptr to inode, can be NULL)
  - d_flags specify FS specific behavior e.g., DCACHE_OP_REVALIDATE
- Ext4 can be configured to maintain directory entries in linear or hashed structures

# Objects and Interfaces: dentry

- Represents an element in a file system path, Ext4 does not have anything similar. On-disk directory entry is the basis to create dentry
- Important members of "struct dentry"
  - d_parent (parent dentry), d_name, d_inode (ptr to inode, can be NULL)
  - d_flags specify FS specific behavior e.g., DCACHE_OP_REVALIDATE
- Ext4 can be configured to maintain directory entries in linear or hashed structures

```
struct ext2_dir_entry_2 {
        __le32  inode;                  // Inode number, 0 ⇒ unused
        __le16  rec_len;                // length of the entry
        __u8    name_len;               // Name length
        __u8    file_type;              // Regular file, directory, symlink..
        char    name[EXT2_NAME_LEN];    // File name
};
```

# Objects and Interfaces: dentry

- Represents an element in a file system path, Ext4 does not have anything similar. On-disk directory entry is the basis to create dentry
- Important members of “struct dentry”
  - d_parent (parent dentry), d_name, d_inode (ptr to inode, can be NULL)
  - d_flags specify FS specific behavior e.g., DCACHE_OP_REVALIDATE
- Ext4 can be configured to maintain directory entries in linear or hashed structures
- Ext4 directory inode operations (ext4_dir_inode_operations) provide crucial handlers for “lookup”, “create” , “mkdir” etc.

# Path translation

- Important structure: “struct nameidata”
  - Important members: path (mount and current walk state info), last (next element in path), last_type (double dot, simple etc.)
  - path⟶ dentry of the parent: current state of translation
- Every path lookup starts with a valid nameidata
- The function link_path_walk is a high-level driver for translating individual path elements
  - Performs checks for permission on the parent directory
  - Advances path translation by updating the last, last_type and path

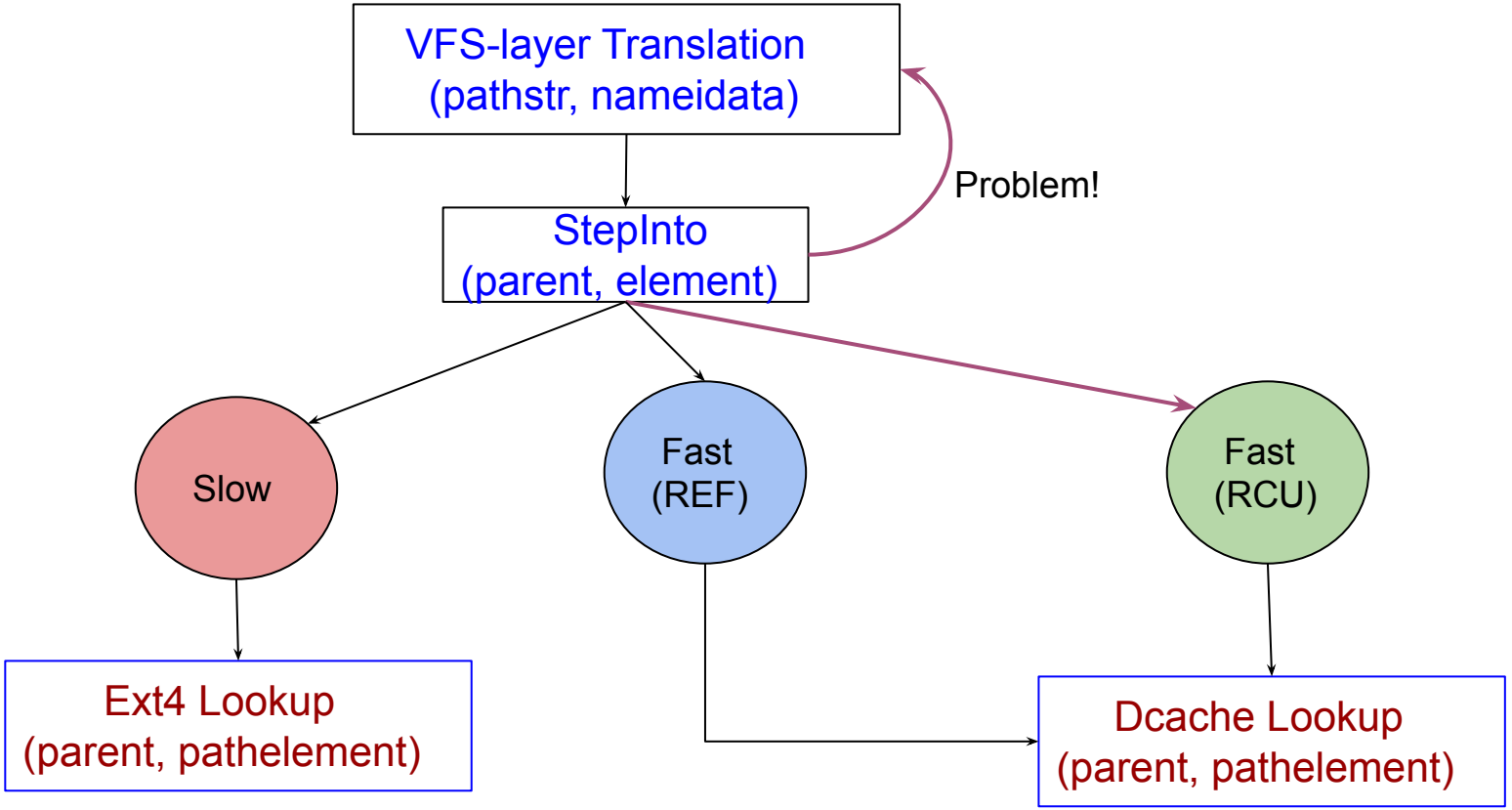
# Operation: Path translation using slow path

- link_path_walk ⟶ walk_component ⟶ lookup_slow ⟶ ext4_lookup
- If found in the directory (passed as the first argument), ext4_lookup links the dentry with the parent and fills its VFS inode
  - Uses ext4_lookup_entry to get the on-disk entry
  - Performs validity checks for the on-disk entry
  - Fills up the inode structure (ext4_iget)
  - Links up the inode with the dentry
- Path lookup continues at link_path_walk (depending on if its is the last component or not)

# Operation: Path translation using fast path

- link_path_walk ⟶ walk_component ⟶ lookup_fast ⟶ {__d_lookup_rcu or __d_lookup}
- Both lookup methods use a hash lookup using “name” and “dentry” as the lookup key followed by a byte-by-byte comparison
- __d_lookup_rcu (RCU walk) and __d_lookup (REF walk) differ in the way lock is used
- RCU-based walk falls back to REF walk if RCU walk fails (because of some issues while walking)
  - Example: do_filp_open (tries RCU, REF walk and revalidation in the order)

# Path translation (Summary view)



# Objects and Interfaces: address spaces

- Address space object is used to manages memory pages belonging to file in the page cache
  - Example usage: lookup by address, dirty writeback
- In Ext4, address space operations is a pointer in inode which is set in the function ext4_iget when preparing the VFS inode
- The address space object is used extensively during read and write operations from both the VFS (page cache related) and Ext4 (syncing etc.)

# Ext4: Read through the page cache

- Implementation of “read” in the Ext4 file system goes through many ping-pongs between the VFS layer (including page cache) and Ext4 FS
  - vfs_read⟶ ext4_file_read_iter ⟶ generic_file_read_iter ⟶ filemap_read ⟶ filemap_get_pages ⟶ … ⟶ read_pages ⟶ ext4_mpage_readpages ⟶ submit_bio (with an endio)
- The file system is involved only to read (or readahead) file blocks from the block device (mostly on a page cache miss)

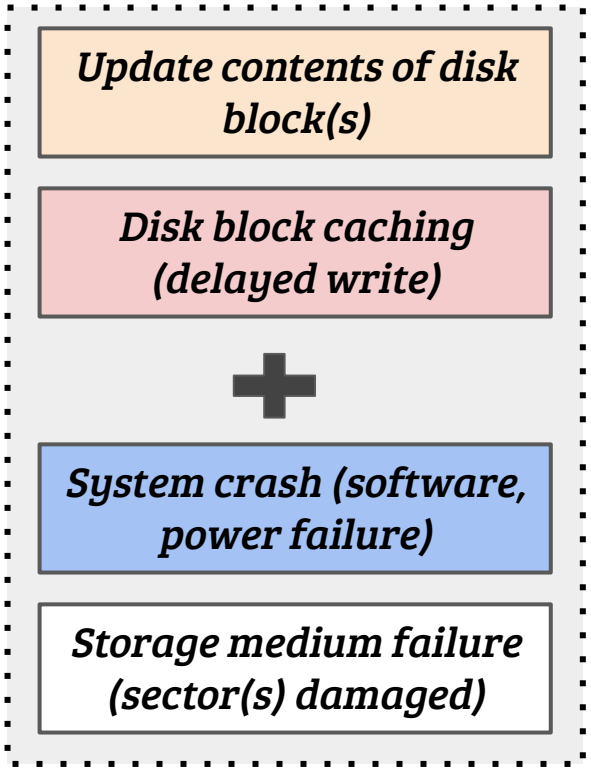
# Ext4: Read through the page cache

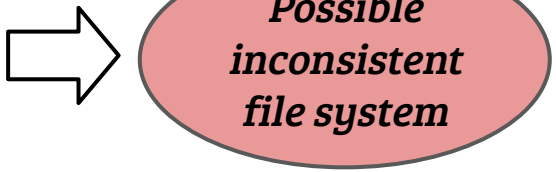
- Implementation of “read” in the Ext4 file system goes through many ping-pongs between the VFS layer (including page cache) and Ext4 FS
  - vfs_read⟶ ext4_file_read_iter ⟶ generic_file_read_iter ⟶ filemap_read ⟶ filemap_get_pages ⟶ … ⟶ read_pages ⟶ ext4_mpage_readpages ⟶ submit_bio (with an endio)
- The file system is involved only to read (or readahead) file blocks from the block device (mostly on a page cache miss)
- Page cache implementation is no more with a radix tree, replaced using an extensible array (xarray), accessed through the file address space

# Caching and consistency (Recap)

- 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

# Recap: File system inconsistency

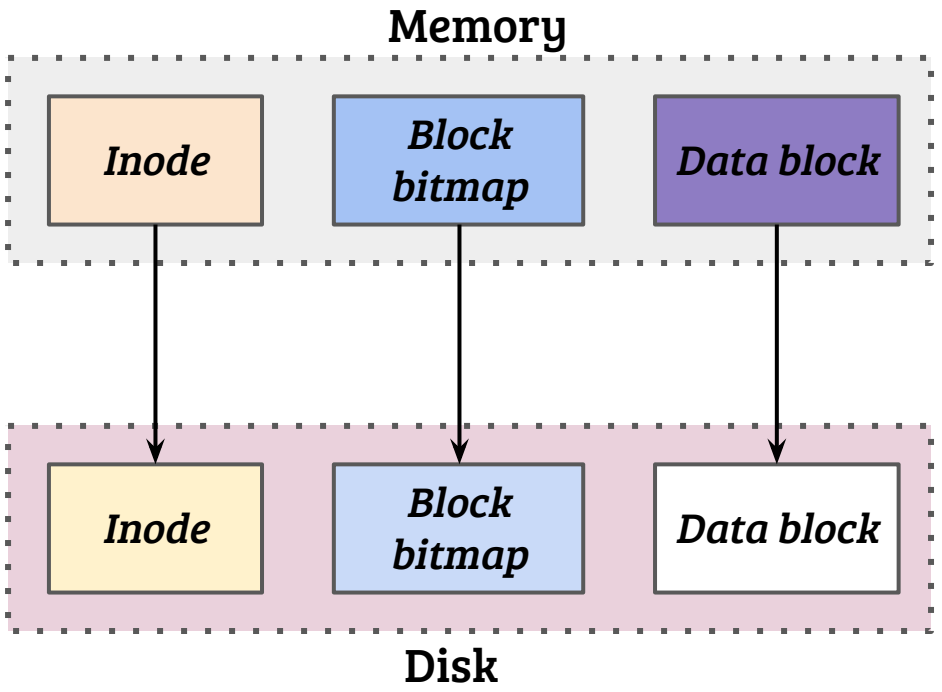



- No consistency issues if user operation translates to read-only operations on the disk blocks
- Device level atomicity may impact file system consistency

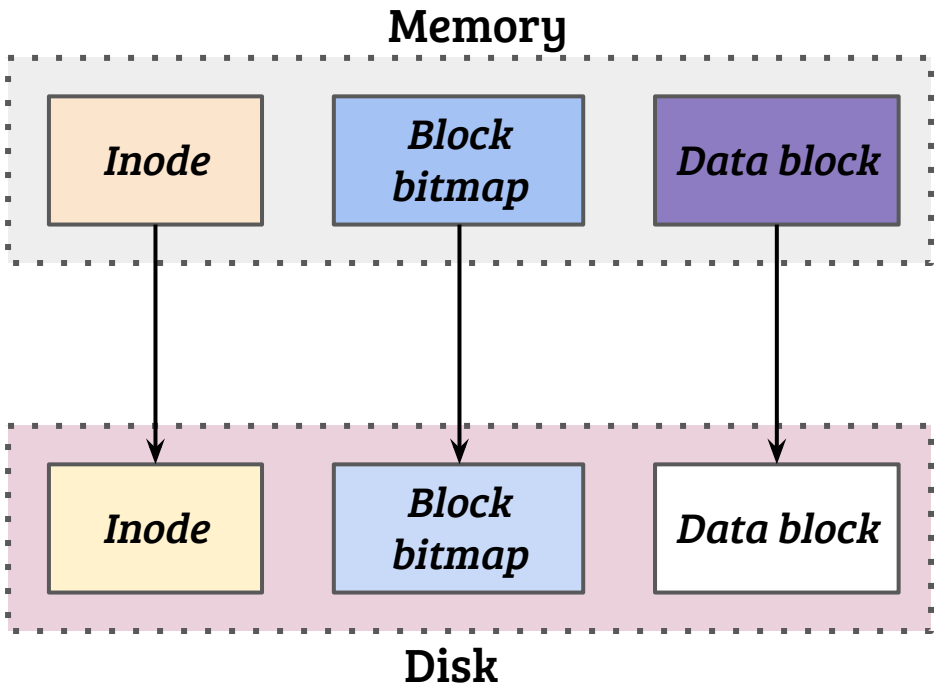
# Example: Append to a file

- 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 journal inode or journal device (a.k.a. Write-ahead logging)
- Journal entry for append operation

# File system consistency with *journaling*

- Idea: Before the actual operation, note down the operations in some special journal inode or journal device (a.k.a. Write-ahead logging)
- Journal entry for append operation


- (1) Write the Journal entry (journal write) (2) Update the file system (checkpoint) (3) Release journal entry

# File system consistency with *journaling*

- Idea: Before the actual operation, note down the operations in some special journal inode or journal device (a.k.a. Write-ahead logging)
- Journal entry for append operation


- (1) Write the Journal entry (journal write) (2) Update the file system (checkpoint) (3) Release journal entry
- Recovery: Journal entries inspected during the next mount and operations are re-performed

# File system consistency with *journaling*

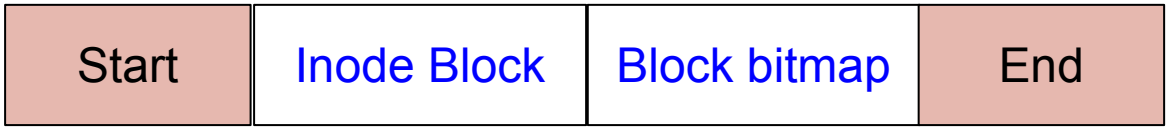

- (1) Write the Journal entry (journal write) (2) Update the file system (checkpoint) (3) Release journal entry
- Implications of a failure during checkpoint?
- Implications of a failure during journal write?
- Are there any special requirements for journal write?
- Can the same inode block be updated after journal write?
- Overheads and optimizations?

# File system consistency with *journaling*


- Implications of a failure during checkpoint? File system state can be recovered during recovery by replaying the journal entries
- Implications of a failure during journal write? No problem if a partial entry can be detected at the time of recovery (may incur data loss in cached I/O)
- Are there any special requirements for journal write? Detection of partial entries ("End" written at the end) or include a checksum in "Start" and "End"
- Can the same inode block be updated after journal write? No issues
- Overheads and optimizations? Batch transactions by holding the blocks in the buffers, every updation applied to the buffers before a periodic journal commit

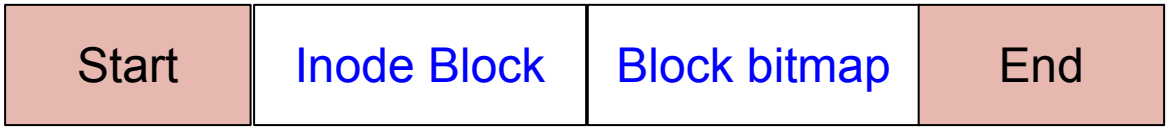
# Metadata journaling: performance-reliability tradeoff

- 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




- Strategy: (1) Write the data block (to disk) (2) Journal write (3) Journal Commit (4) Checkpoint (5) Release
- Why data block should be written first?
- Should the journal write wait for data write to be completed?
- Are there any issues with block reuse?

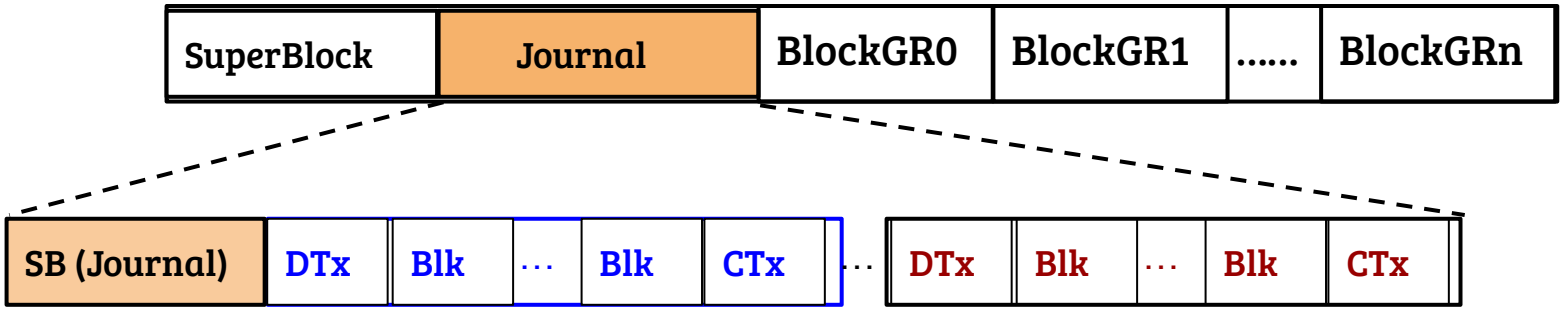
# Metadata journaling: performance-reliability tradeoff




- Strategy: (1) Write the data block (to disk) (2) Journal write (3) Journal Commit (4) Checkpoint (5) Release
- Why data block should be written first?
  - If the metadata blocks are not written, FS can be recovered
  - If journal write fails, a write is lost (syscall semantic broken)
- Should the journal write wait for data write to be completed?
  - Journal write and write to the data block can happen in parallel
  - Journal commit (writing "End") should take place afterwards
- Are there any issues with block reuse? If the nature of block usage change, special handling is required (e.g., revocation records)

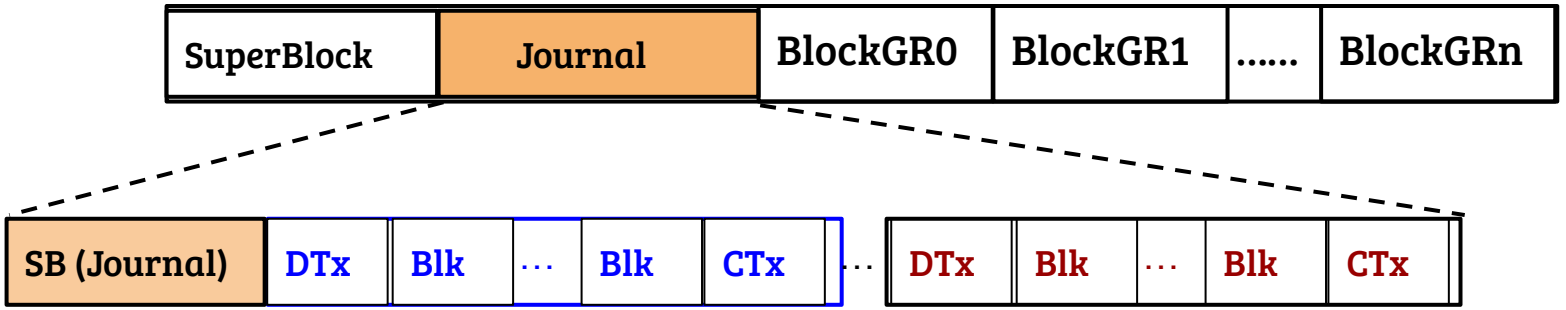
# Journaling in Ext4

- In Ext4, journal can be stored in two ways
  - In the same FS with a special inode (inode num 8)
  - On an external logical block device



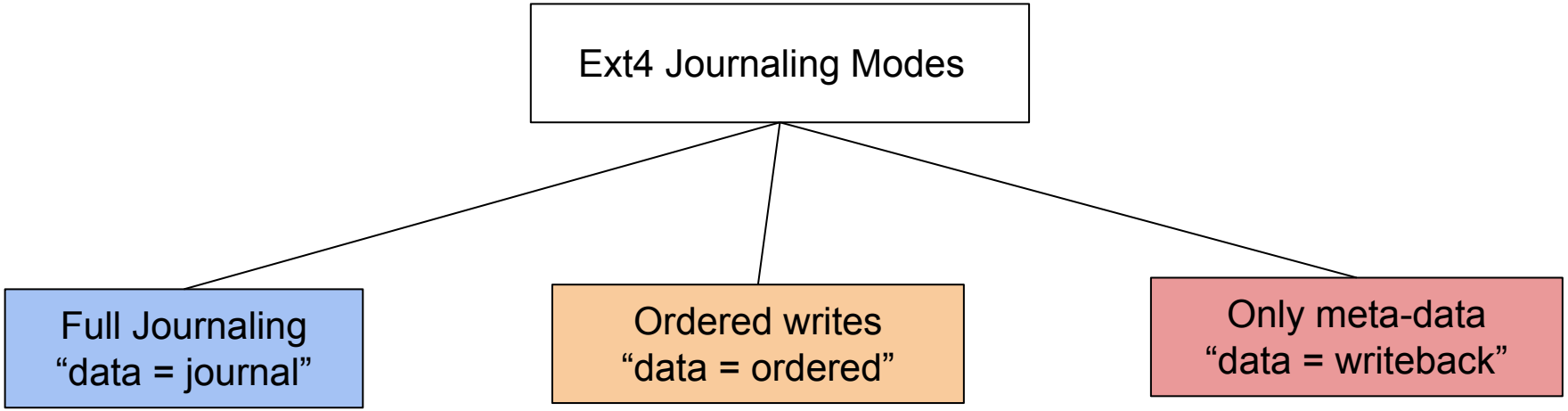
# Journaling in Ext4

- In Ext4, journal can be stored in two ways
  - In the same FS with a special inode (inode num 8)
  - On an external logical block device

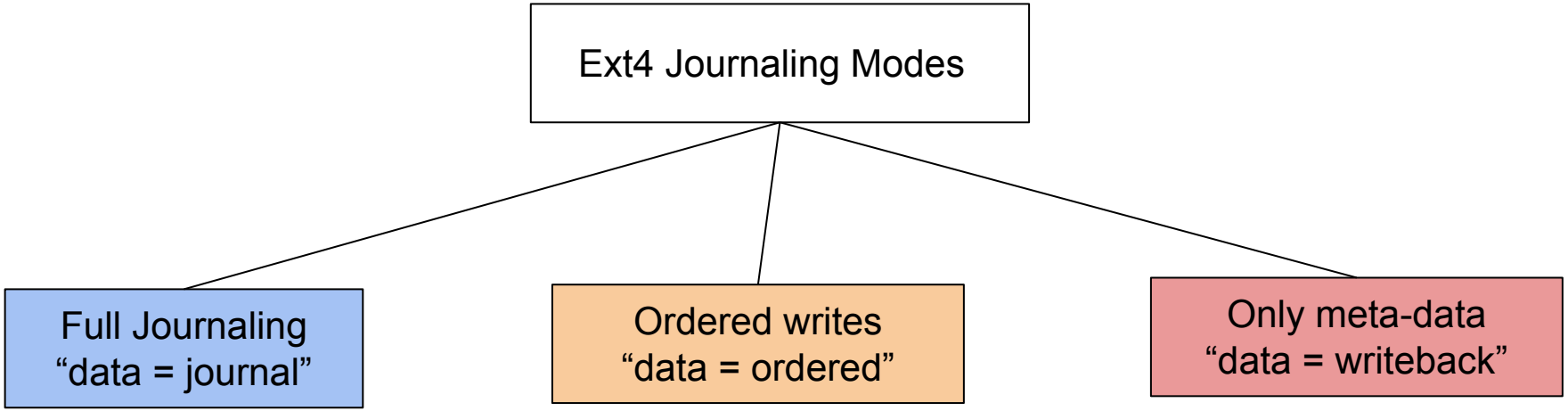



- A contiguous area (default: 128MB) is demarcated as the journal
- The journal super block contains some static information (e.g., size related) and dynamic information (e.g., related to location of valid entries)

# Journaling modes in Ext4

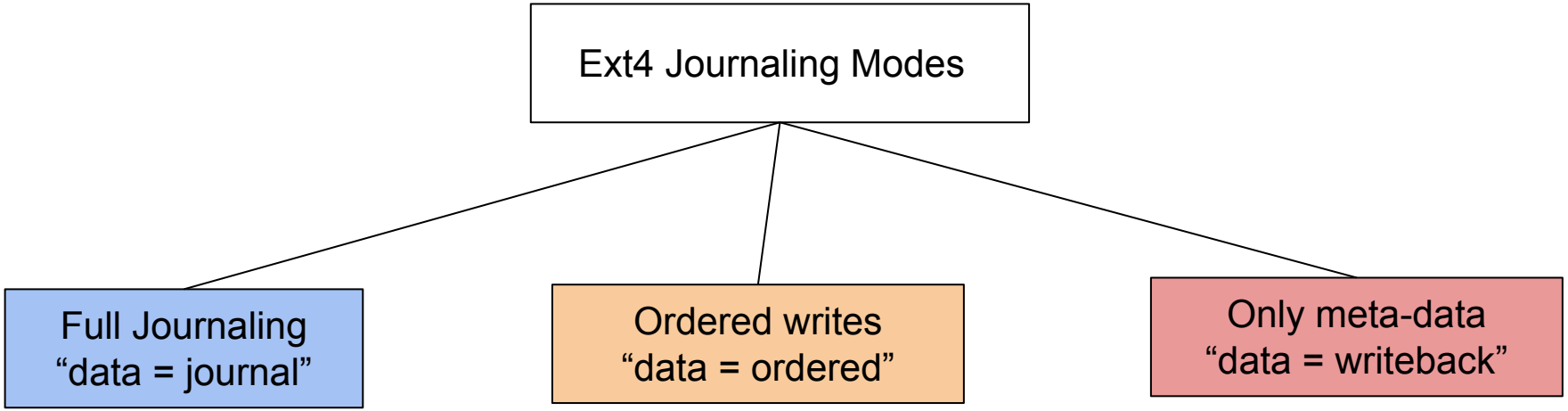


# Journaling modes in Ext4




Both data and MD are journaled, guaranteed consistency, heavy performance penalties

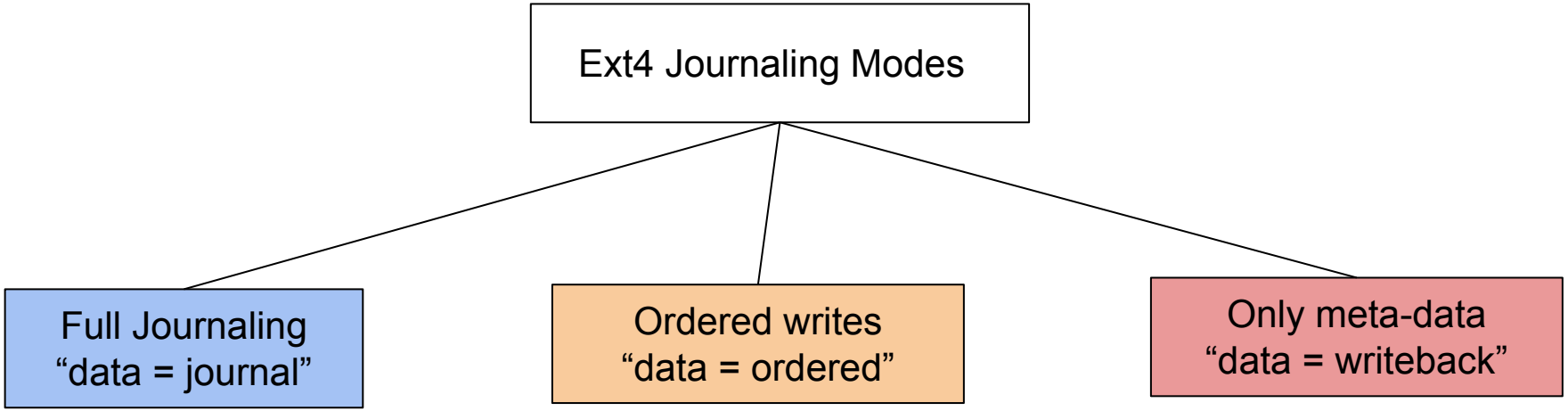
# Journaling modes in Ext4




Both data and MD are journaled, guaranteed consistency, heavy performance penalties

(Default) Data writes before MD is journaled, faster than full journaling, suitable for cached I/O

# Journaling modes in Ext4




Both data and MD are journaled, guaranteed consistency, heavy performance penalties

(Default) Data writes before MD is journaled, faster than full journaling, suitable for cached I/O

No ordering constraints between data and MD. Fast but lead to many possible issues

# Journaling support in Linux

- Linux provides a generic journaling API for the file systems (JBD2)
- Some of important features of JBD2
  - Initializing and loading the journal
  - Marking start of the transaction
  - Committing a transaction
  - A kernel thread for periodic commit

# Journaling support in Linux

- Linux provides a generic journaling API for the file systems (JBD2)
- Some of important features of JBD2
  - Initializing and loading the journal
  - Marking start of the transaction
  - Committing a transaction
  - A kernel thread for periodic commit
- Refer to the code ext4_load_journal, __ext4_journal_stop and __ext4_journal_start for more details