Operating Systems

File Systems

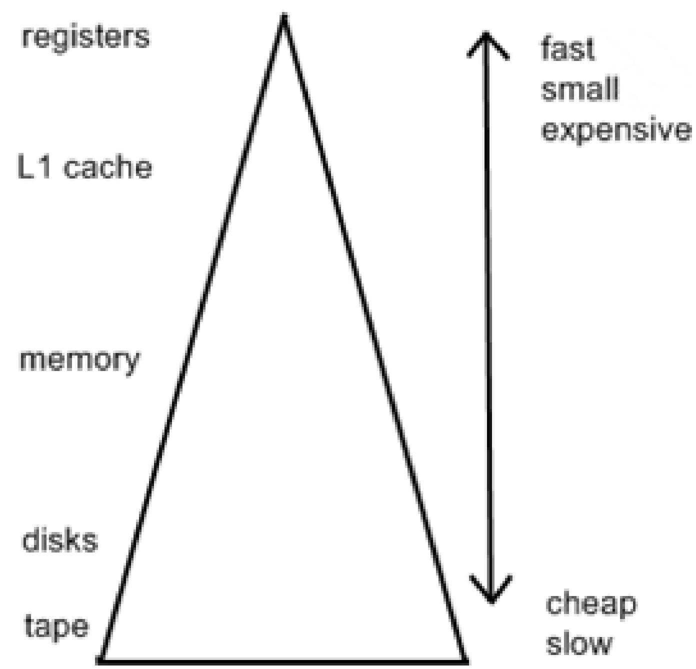
ENCE 360

Motivation – Top Down: Process Need

- Processes store, retrieve information
- When process terminates, memory lost
- How to make it persist?
- What if multiple processes want to share?

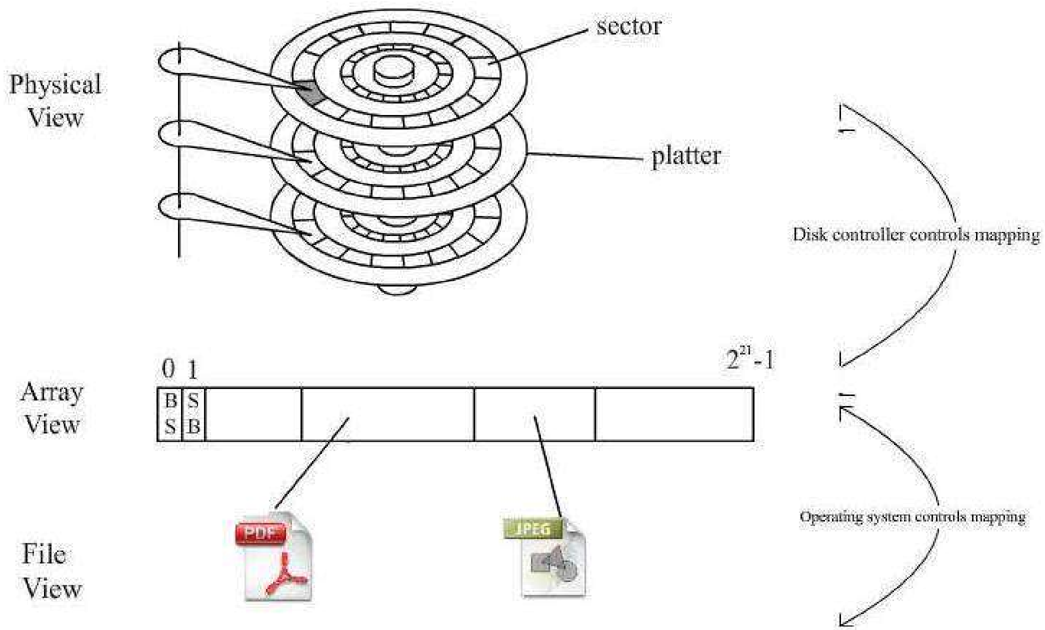
- Requirements:
  - large
  - persistent
  - concurrent access

Solution?
Hard disks are large, persistent!



Motivation – Bottom Up: Hard Disks

bs – boot sector
sb – super block



Disks come formatted with blocks (typically 512 bytes)

- Requirements
  - Differentiation of data blocks
  - Reading and writing of blocks
  - Efficient access

Solution? File Systems

CRUX: HOW TO IMPLEMENT A FILE SYSTEM ON A HARD DISK
How to find information?
How to map blocks to files of all sizes?
How to know which blocks are free?

Outline

- Introduction (done)
- Implementation (next)
- Directories
- Journaling

Chapter 4
MODERN OPERATING SYSTEMS (MOS)
By Andrew Tanenbaum

Chapter 39, 40
OPERATING SYSTEMS: THREE EASY PIECES
By Arpaci-Dusseau and Arpaci-Dusseau

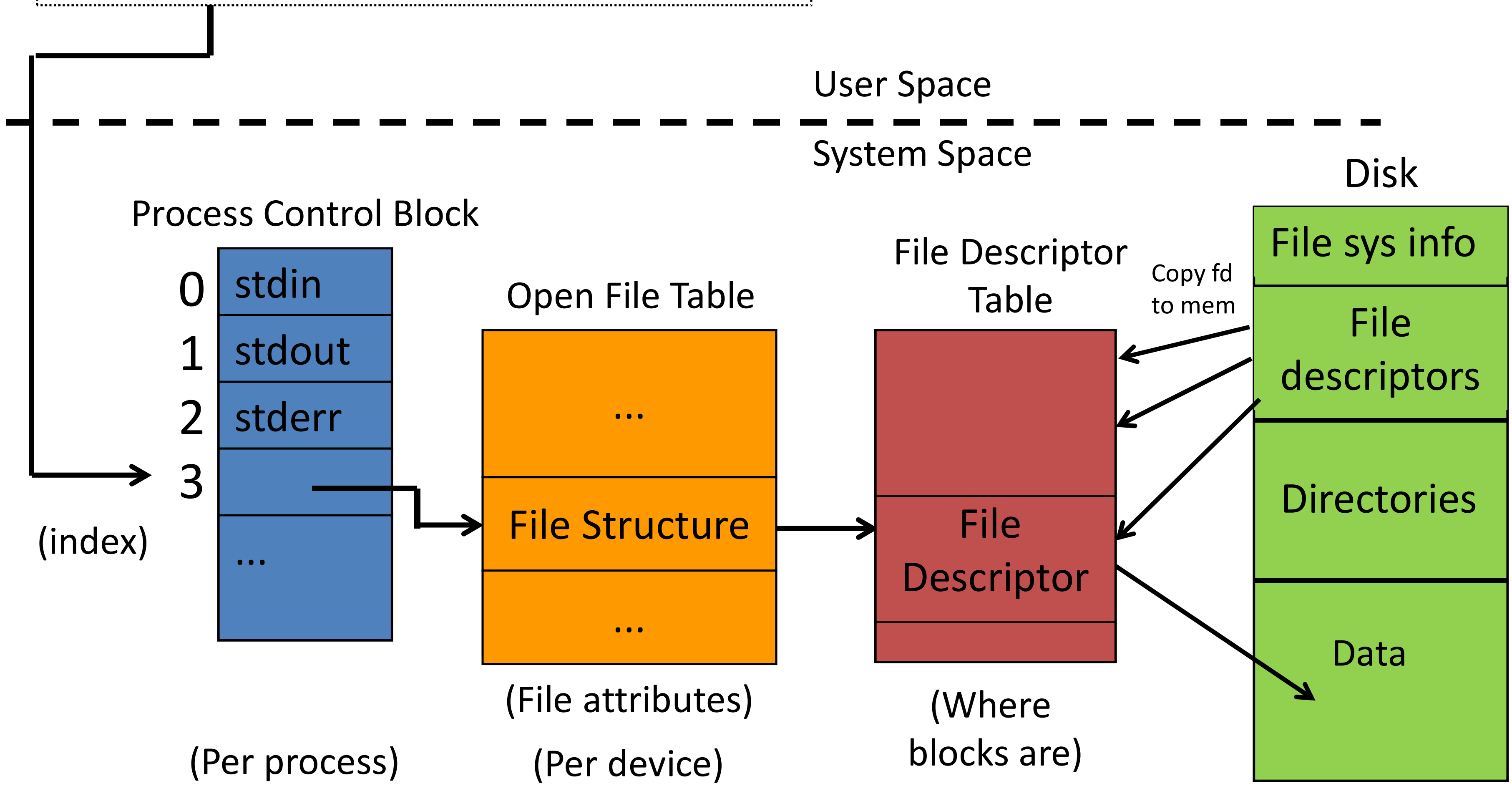
Example: Unix `open()`

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

- `path` is name of file (NULL terminated string)
- `flags` is bitmap to set switch
  - O_RDONLY, O_WRONLY, O_TRUNC …
  - O_CREATE then use `mode` for permissions
- success returns `index`
  - On error, `-1` and set `errno`

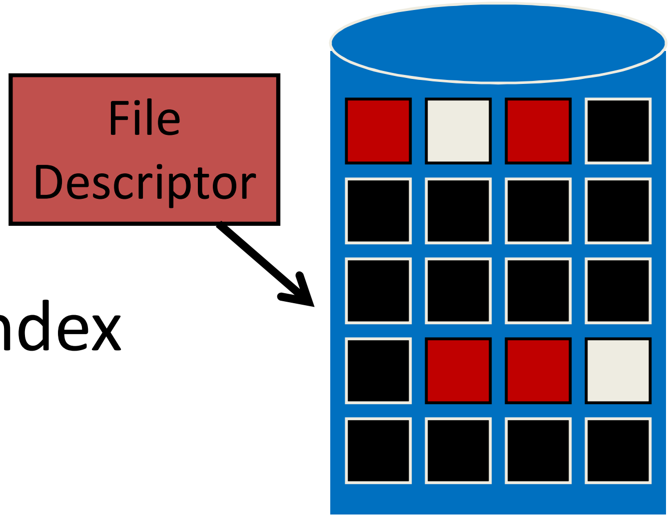
Unix open() – Under the Hood

```
int fid = open("blah", flags);
read(fid, …);
```



File System Implementation

- Core data to track: which blocks with which file?
  - Job of the file descriptor
- Different implementations:
  a) Contiguous allocation
  b) Linked list allocation
  c) Linked list allocation with index
  d) Inode

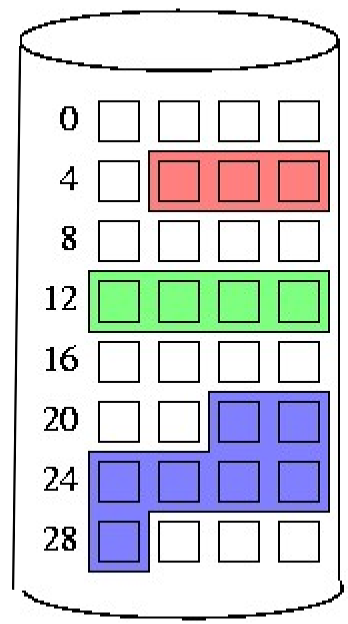


Contiguous Allocation (1 of 2)

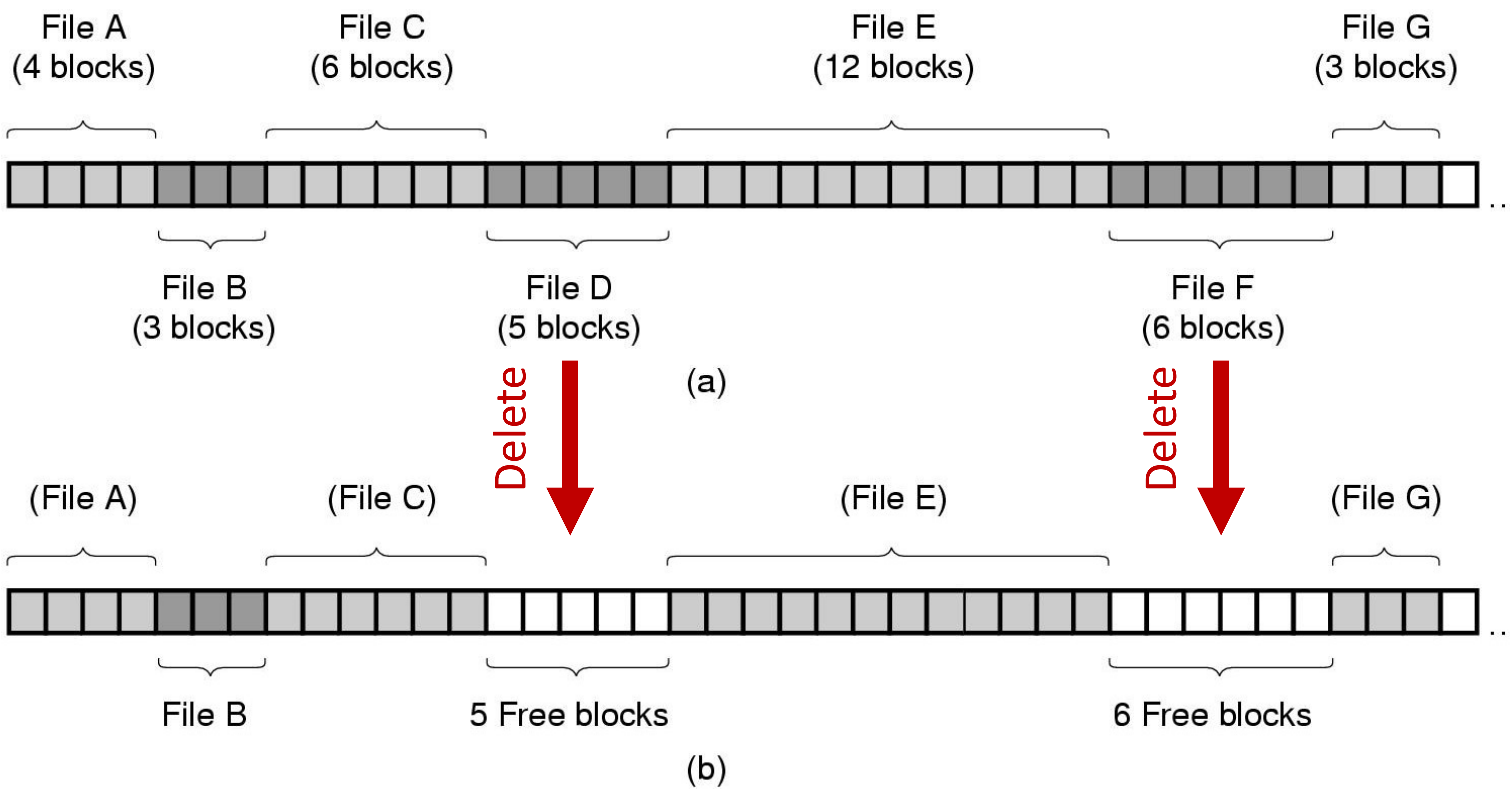
- Store file as contiguous blocks on disk
- Good:
  - Easy: file descriptor knows file location in 1 number (start block)
  - Efficient: read entire file in 1 operation (start & length)
- Bad:
  - Static: need to know file size at creation
    - Or tough to grow!
  - Fragmentation: chunks of disk "free" but can't be used

(Example next slide)

file	start	length
moo	5	3
snow	22	7
fall	12	4



Contiguous Allocation (2 of 2)

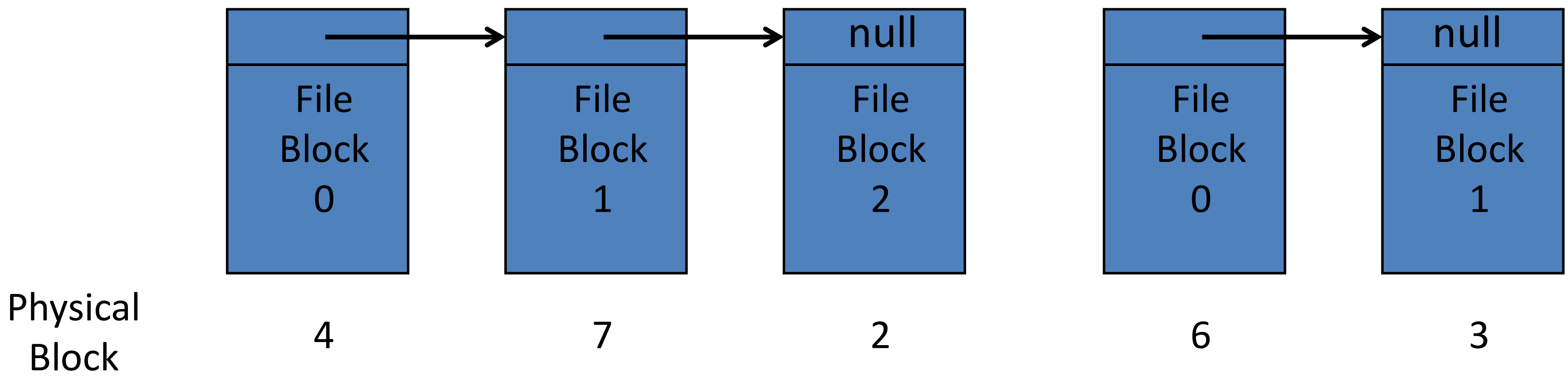


What if want new file, size 8 blocks?

→ Fragmentation ("free" but can't be used)

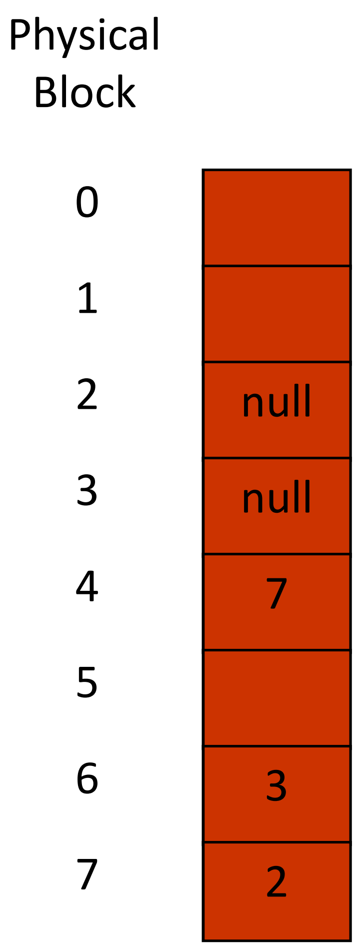
Linked List Allocation

- Keep linked list with disk blocks



- Good:
  - Easy: remember 1 number (location)
  - Efficient: no space lost in fragmentation
- Bad:
  - Slow: random access bad (e.g., process want’s middle block)

Linked List Allocation with Index

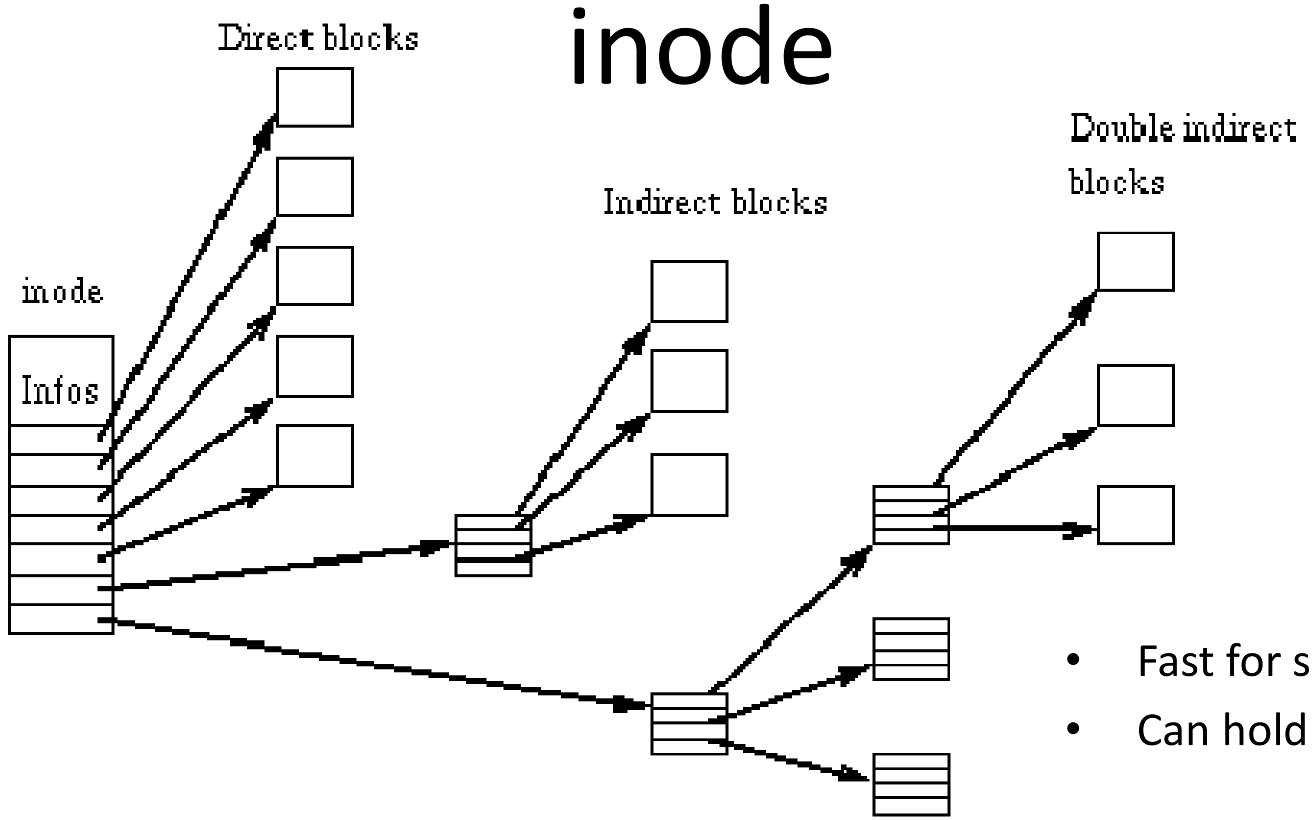


- Table in memory
  - MS-DOS FAT, Win98 VFAT
- Good: faster random access
- Bad: can be large! e.g., 1 TB disk, 1 KB blocks
  - Table needs 1 billion entries
  - Each entry 3 bytes (say 4 typical)
    - → 4 GB memory!

"File Allocation Table"

Common format still (e.g., USB drives) since supported by many OSes & additional features not needed

inode



- Fast for small files
- Can hold large files

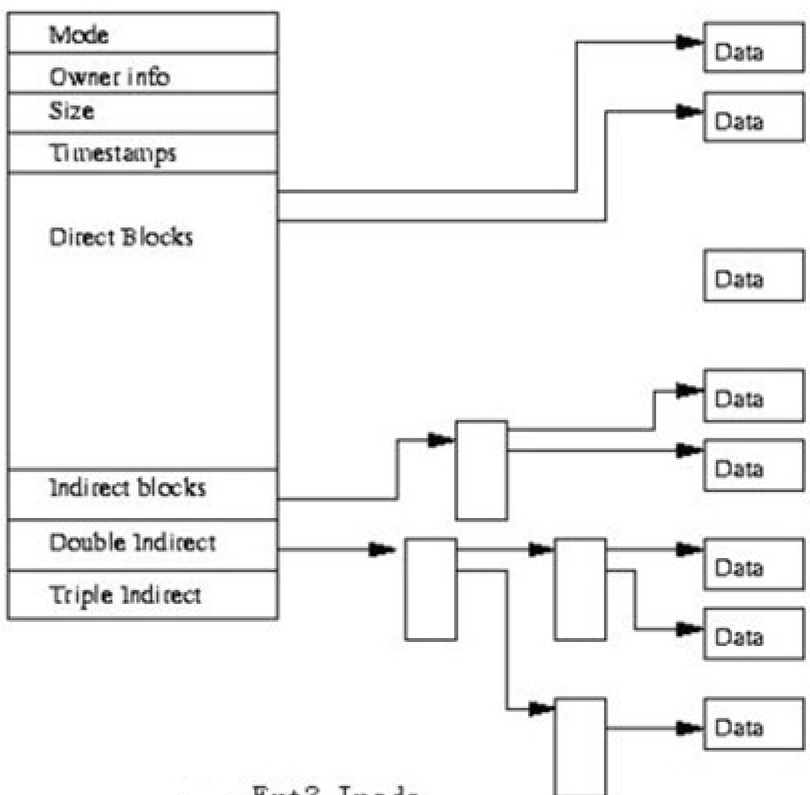
- Typically 15 pointers
  - 12 to direct blocks
  - 1 single indirect
  - 1 doubly indirect
  - 1 triply indirect

- Number of pointers per block? Depends on block size and pointer size
  - e.g., 1k byte block, 4 byte pointer → each indirect has 256 pointers
- Max size of file? Same – depends on block size and pointer size
  - e.g., 4KB block, 4 byte pointer → max size 2 TB

Linux File System: ext3 inode

```
// linux/include/linux/ext3_fs.h
#define EXT3_NDIR_BLOCKS 12                      // Direct blocks
#define EXT3_IND_BLOCK   EXT3_NDIR_BLOCKS + 1    // Indirect block index
#define EXT3_DIND_BLOCK  EXT3_IND_BLOCK + 1      // Double-ind. block index
#define EXT3_TIND_BLOCK  EXT3_DIND_BLOCK + 1     // Triple-ind. block index
#define EXT3_N_BLOCKS    EXT3_TIND_BLOCK + 1     // (Last index & total)

struct ext3_inode {
  __u16     i_mode;        // File mode
  __u16     i_uid;         // Low 16 bits of owner Uid
  __u32     i_size;        // Size in bytes
  __u32     i_atime;       // Access time
  __u32     i_ctime;       // Creation time
  __u32     i_mtime;       // Modification time
  __u32     i_dtime;       // Deletion time
  __u16     i_gid;         // Low 16 bits of group Id
  __u16     i_links_count; // Links count
  __u32     i_blocks;      // Blocks count
  ...
  __u32    i_block[EXT3_N_BLOCKS];  // Block pointers
  ...
}
```



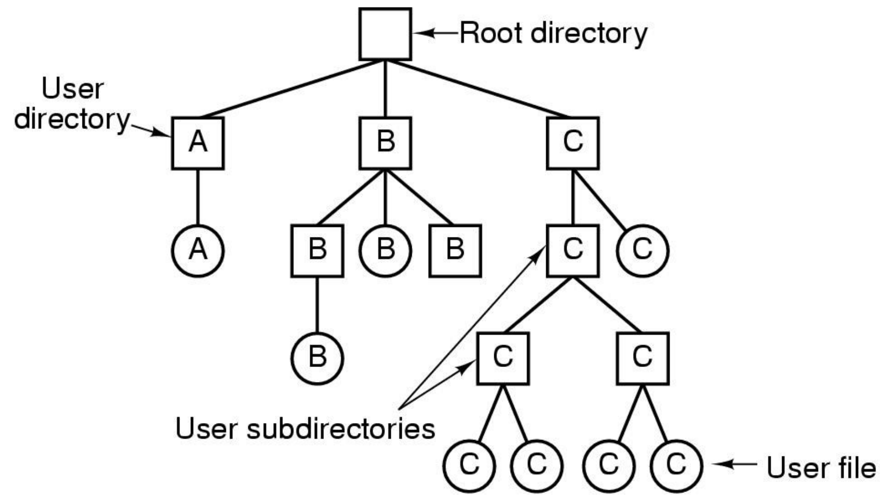
Ext3 Inode

Outline

- Introduction (done)
- Implementation (done)
- Directories (next)
- Journaling

Directory Implementation

- Just like files ("wait, what?")
  - Have data blocks
  - File descriptor to map which blocks to directory
- But have special bit set so user process cannot modify contents
  - Data in directory is information / links to files
  - Modify only through system call (right)
- Tree structure, directory most common



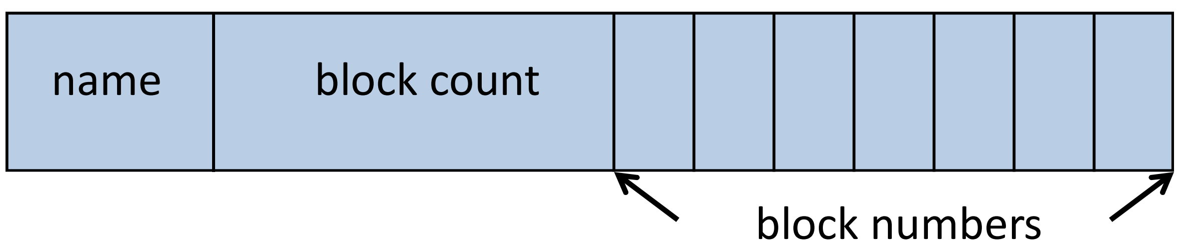
Directory System Calls

- Create
- Delete
- Opendir
- Closedir
- Readdir
- Rename
- Link
- Unlink

See: "`ls.c`"

Directories

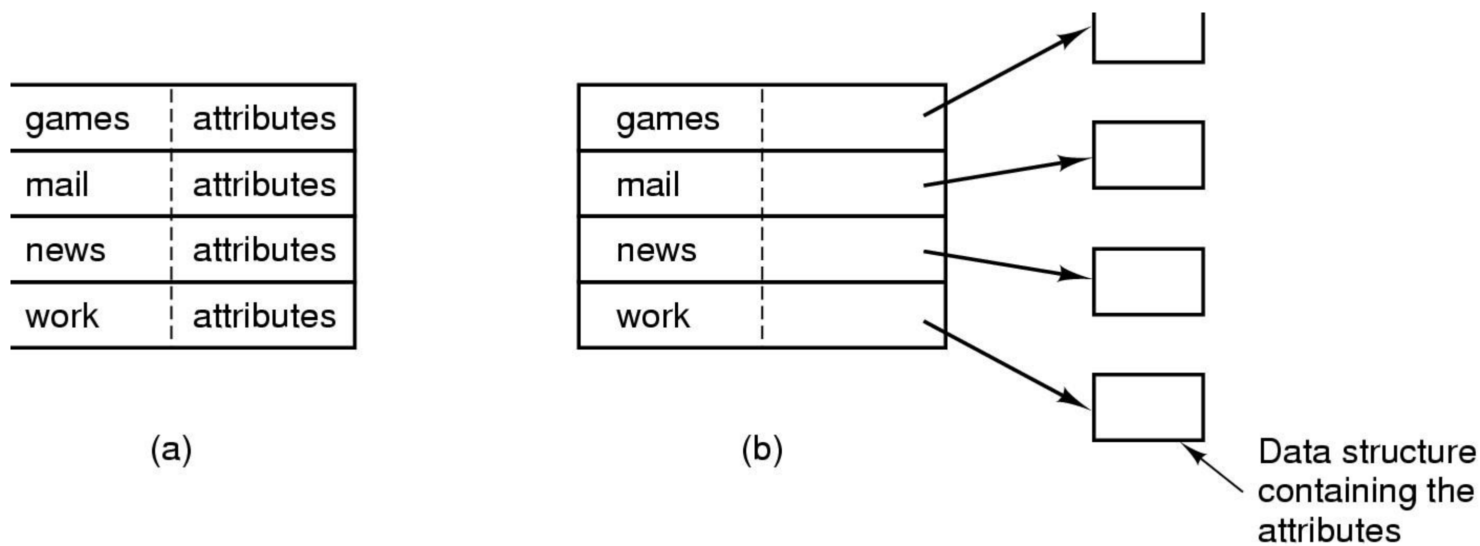
- Before reading file, must be opened
- Directory entry provides information to get blocks
  - Disk location (blocks, address)
- Map ASCII name to *file descriptor*



Where are file attributes (e.g., owner, permissions) stored?

Options for Storing Attributes

a) Directory entry has attributes (Windows)
b) Directory entry refers to file descriptor (e.g., inode), and descriptor has attributes (Linux)



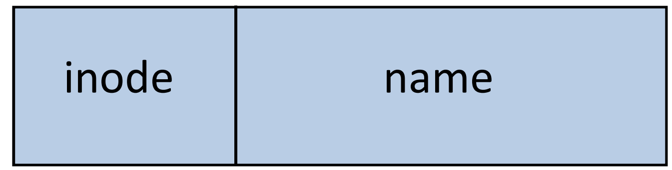
Windows (FAT) Directory

- Hierarchical directories
- Entry:
  – name
  – type (extension)
  – time
  - date
  - block number (w/FAT)

name	type	attrib	time	date	block	size

Unix Directory

- Hierarchical directories
- Entry:



  - name
  - inode number (try "`ls -i`" or "`ls -iad .`")
- Example, say want to read data from below file

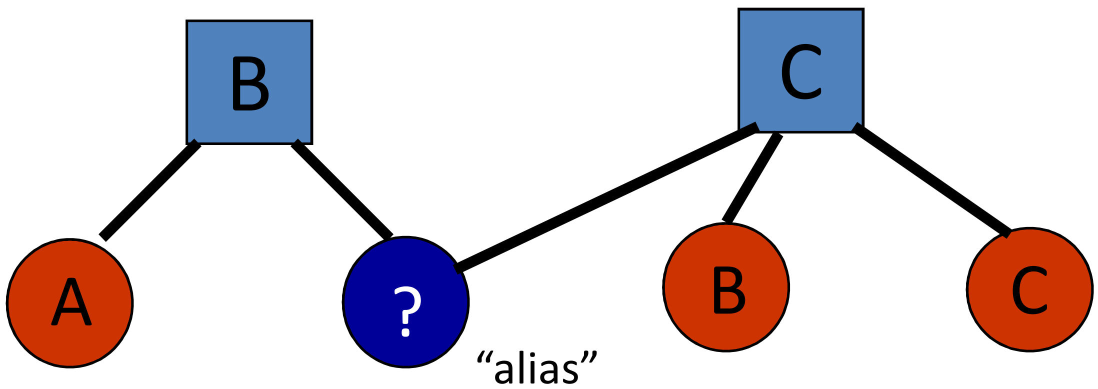
  `/usr/bob/mbox`

  Want contents of file, which is in blocks

  Need file descriptor (inode) to get blocks

  How to find the file descriptor (inode)?

User Access to Same File in More than One Directory



(Instead of tree, really have directed acyclic graph)

Possibilities for "alias":

A. Refer to file descriptor in two locations – "hard link"

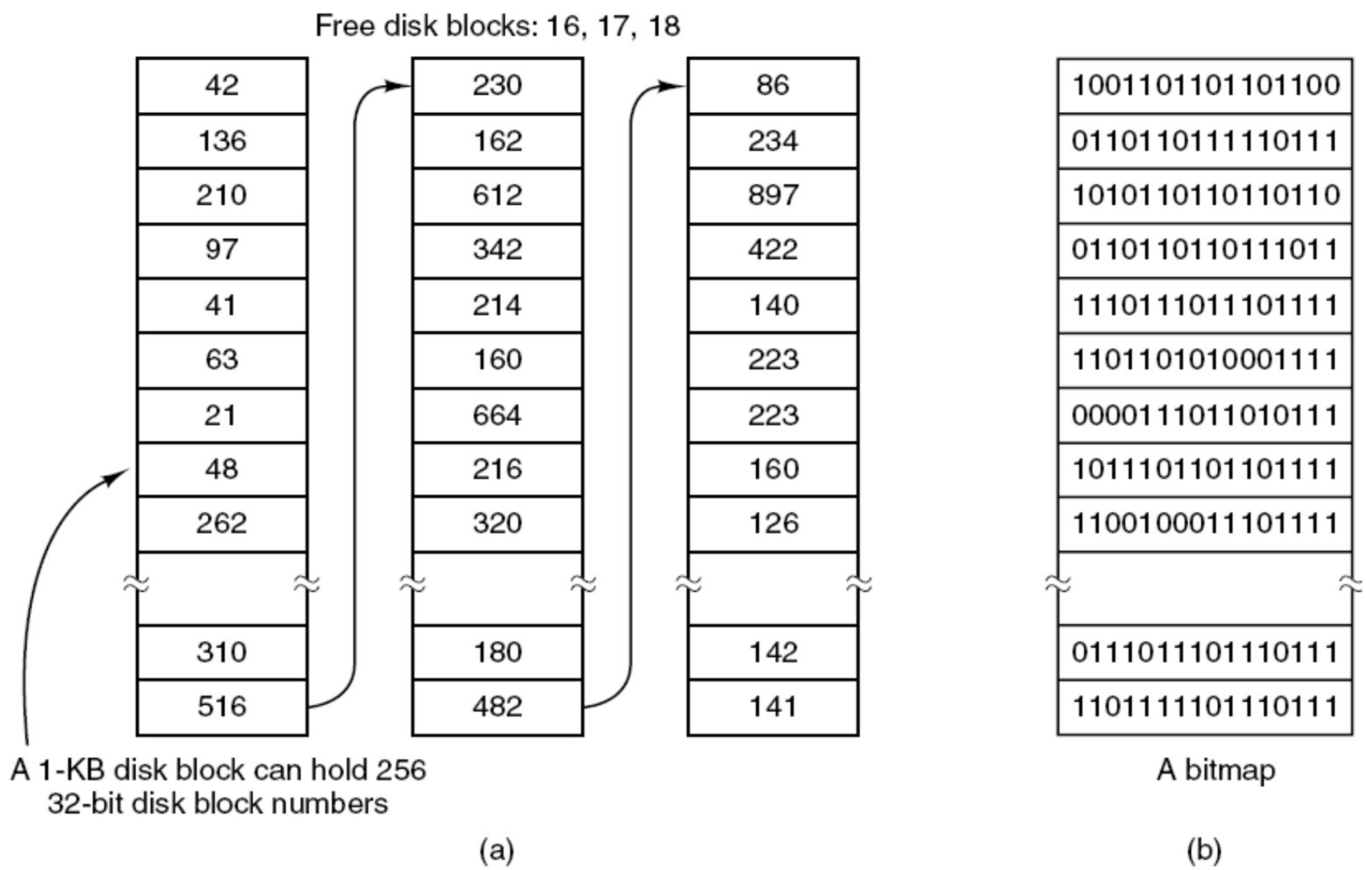
B. Special directory entry points to real directory entry – "soft link"

Examples: try "`ln`", "`ln -s`" and "`ls -i`"

Windows "shortcut" – but only viewable by graphic browser, absolute paths, with metadata, can track even if move

Keeping Track of Free Blocks

Keep one large "file" of free blocks (use normal file descriptor)



Contents are linked-list of free blocks (can be small when full, but no locality)

Contents are bitmap of free blocks (preserves locality, but 1-bit/block)

Outline

- Introduction (done)
- Implementation (done)
- Directories (done)
- Journaling (next)

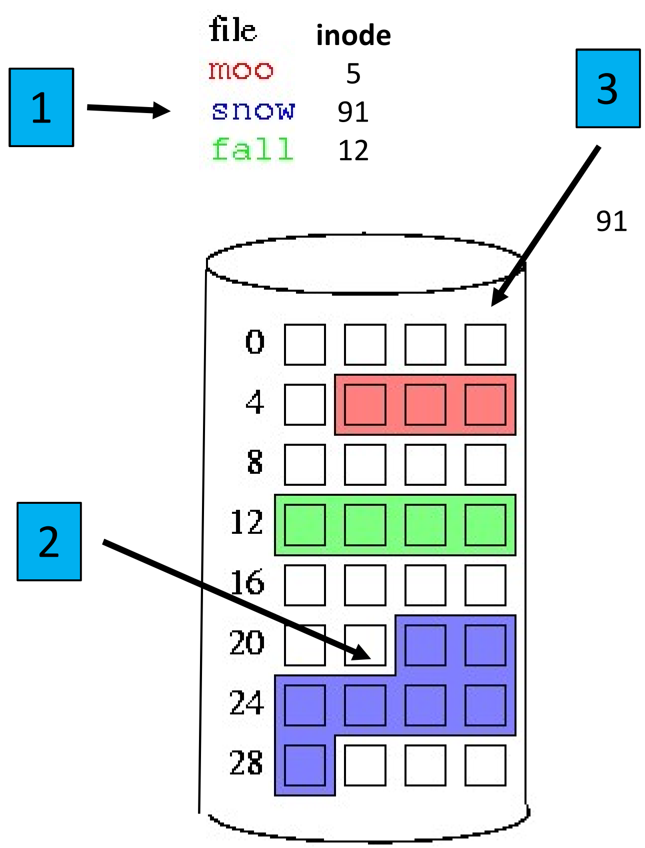
Need for Robust File Systems

- Consider upkeep for removing file
  1. Remove file from directory entry
  2. Return all disk blocks to pool of free disk blocks
  3. Release file descriptor (e.g., inode) to pool of free descriptors
- What if system crashes in middle?

  a) inode becomes orphaned (`lost+found`, 1 per partition)

  b) Same blocks free *and* allocated

  If flip steps, blocks/descriptor free but directory entry exists!
- Crash consistency problem



Crash Consistency Problem

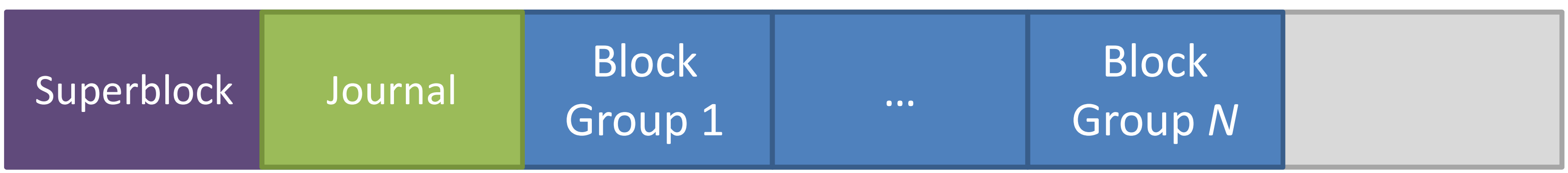
- Disk guarantees that single sector writes are atomic
  - But no way to make multi-sector writes atomic
- How to ensure consistency after crash?
  1. Don't bother to ensure consistency
     - Accept that the file system may be inconsistent after crash
     - Run program that fixes file system during bootup
     - File system checker (e.g., *fsck*)
  2. Use transaction log to make multi-writes atomic
     - Log stores history of all writes to disk
     - After crash log "replayed" to finish updates
     - Journaling file system

File System Checker – the Good and the Bad

- Advantages of File System Checker
  - Doesn't require file system to do any work to ensure consistency
  - Makes file system implementation simpler
- Disadvantages of File System Checker
  - Complicated to implement *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - Usually **super slooooooow…**
    - Scans entire file system multiple times
    - Consider really large disks, like 400 TB RAID array!

Journaling File Systems

1. Write intent to do actions (a-c) to log (aka “journal”) *before* starting
   – Option - read back to verify integrity before continue
2. Perform operations
3. Erase log



- If system crashes, when restart read log and apply operations
- Logged operations must be *idempotent* (can be repeated without harm)

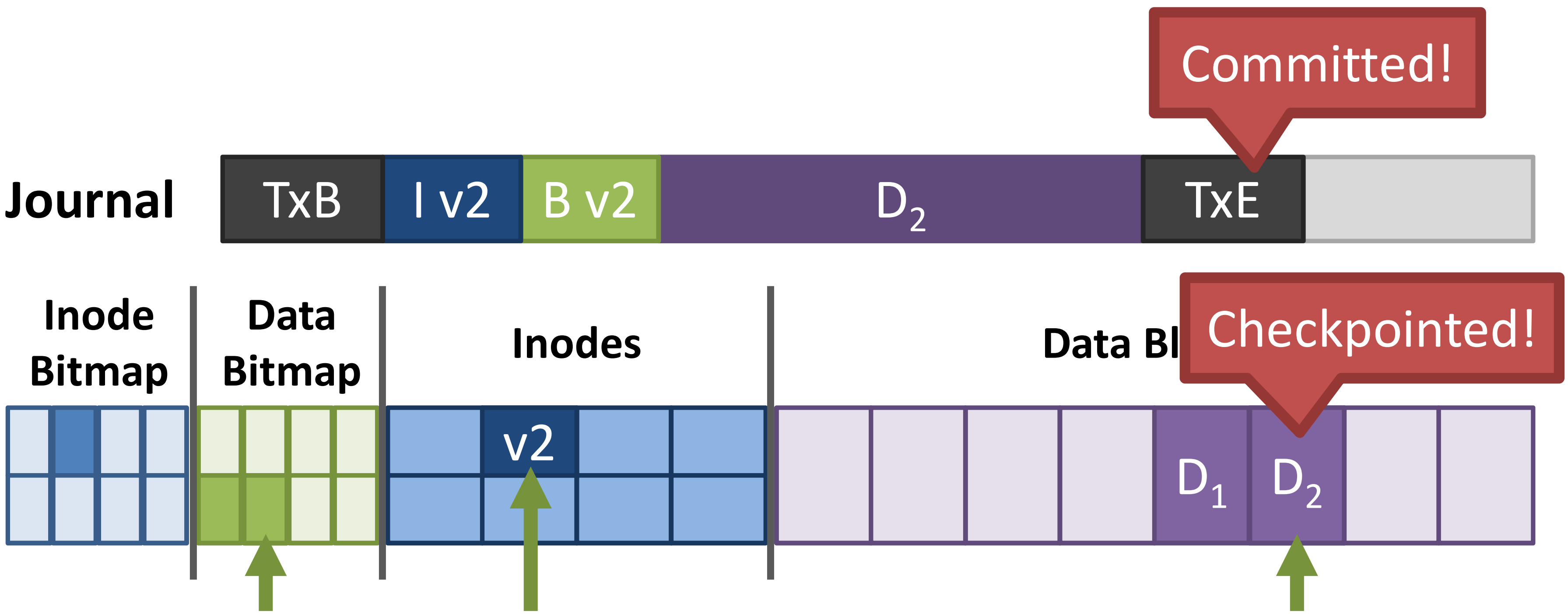
Journaling Example

- Assume appending new data block (D_2) to file
  - 3 writes: inode v2, data bitmap v2, data D_2
- Before executing writes, first log them

1. TxB: Begin new transaction with unique ID=1
2. Write updated meta-data block (inode, data bitmap)
3. Write file data block
4. TxE: Write end-of-transaction with ID=*1*

Commits and Checkpoints

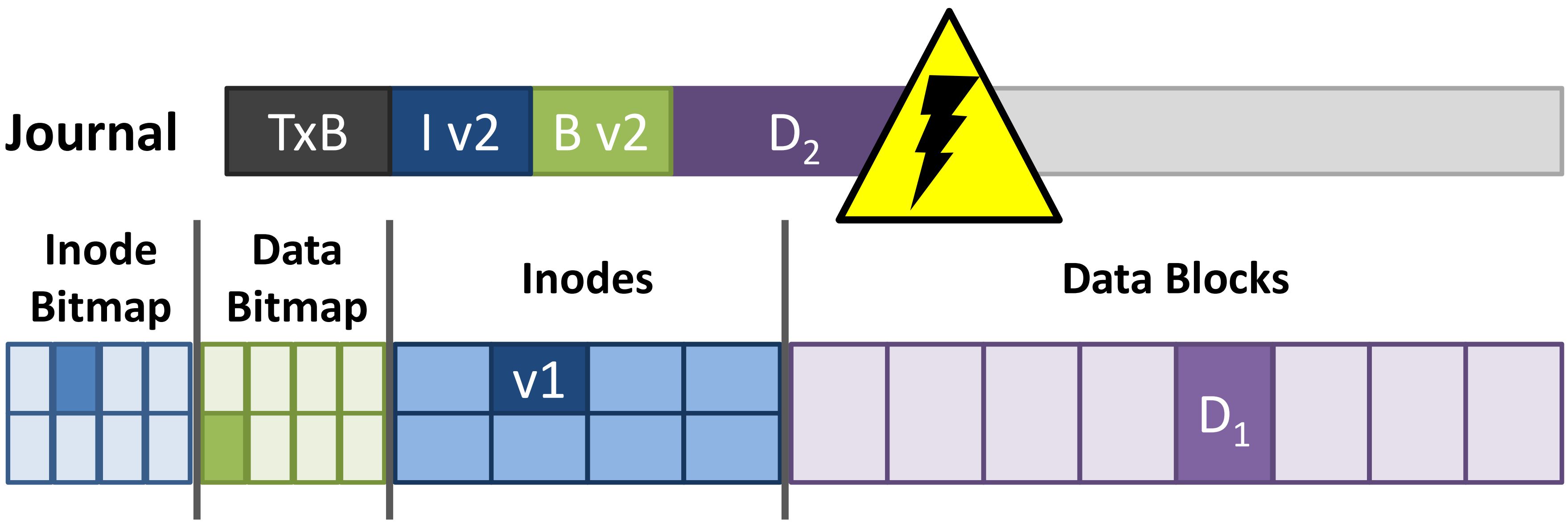
- Transaction committed after all writes to log complete
- After transaction is completed, OS checkpoints update



- Final step: free checkpointed transaction

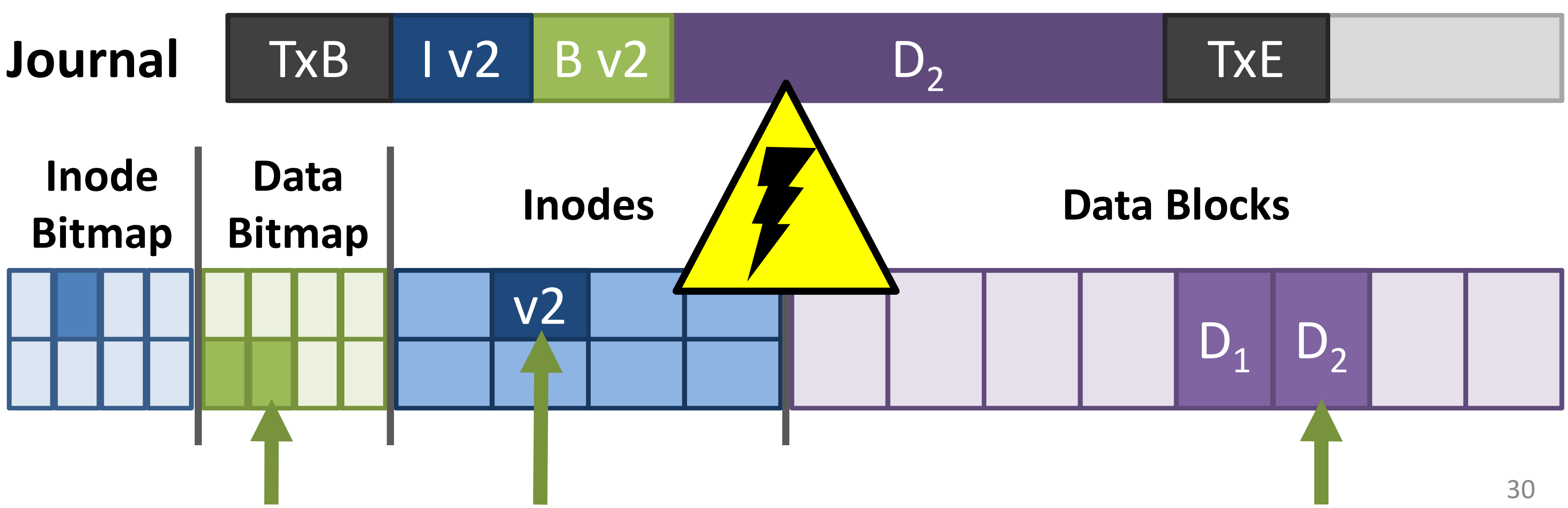
Crash Recovery (1 of 2)

- What if system crashes during logging?
  - If transaction not committed, data lost
  - But, file system remains consistent!



Crash Recovery (2 of 2)

- What if system crashes during checkpoint?
  - File system may be inconsistent
  - During reboot, transactions committed but not completed are replayed in order
  - Thus, no data is lost and consistency restored!



Journaling Summary

- Advantages of journaling
  - Robust, fast file system recovery
    - No need to scan entire journal or file system
  - Relatively straight forward to implement
- Disadvantages of journaling
  - Write traffic to disk doubled
    - Especially file data, which is probably large
  - Can fix! Only journal meta-data!

  (Left for student exploration)

- Today, most OSes use journaling file systems
  - ext3/ext4 on Linux
  - NTFS on Windows
- Provides crash recovery with relatively low space and performance overhead
- Next-gen OSes likely move to file systems with copy-on-write semantics
  - btrfs and zfs on Linux

Outline

- Introduction (done)
- Implementation (done)
- Directories (done)
- Journaling (done)