§ Fast File System (FFS)

“Old FS”

- Very similar to the design we just covered
- Disk throughput starts out slow, and quickly deteriorates
  - Compared to max throughput, deteriorates from ~18% to **~2%**
  - Why?
    - Metadata & data are not close to each other
    - Files become **fragmented** over time

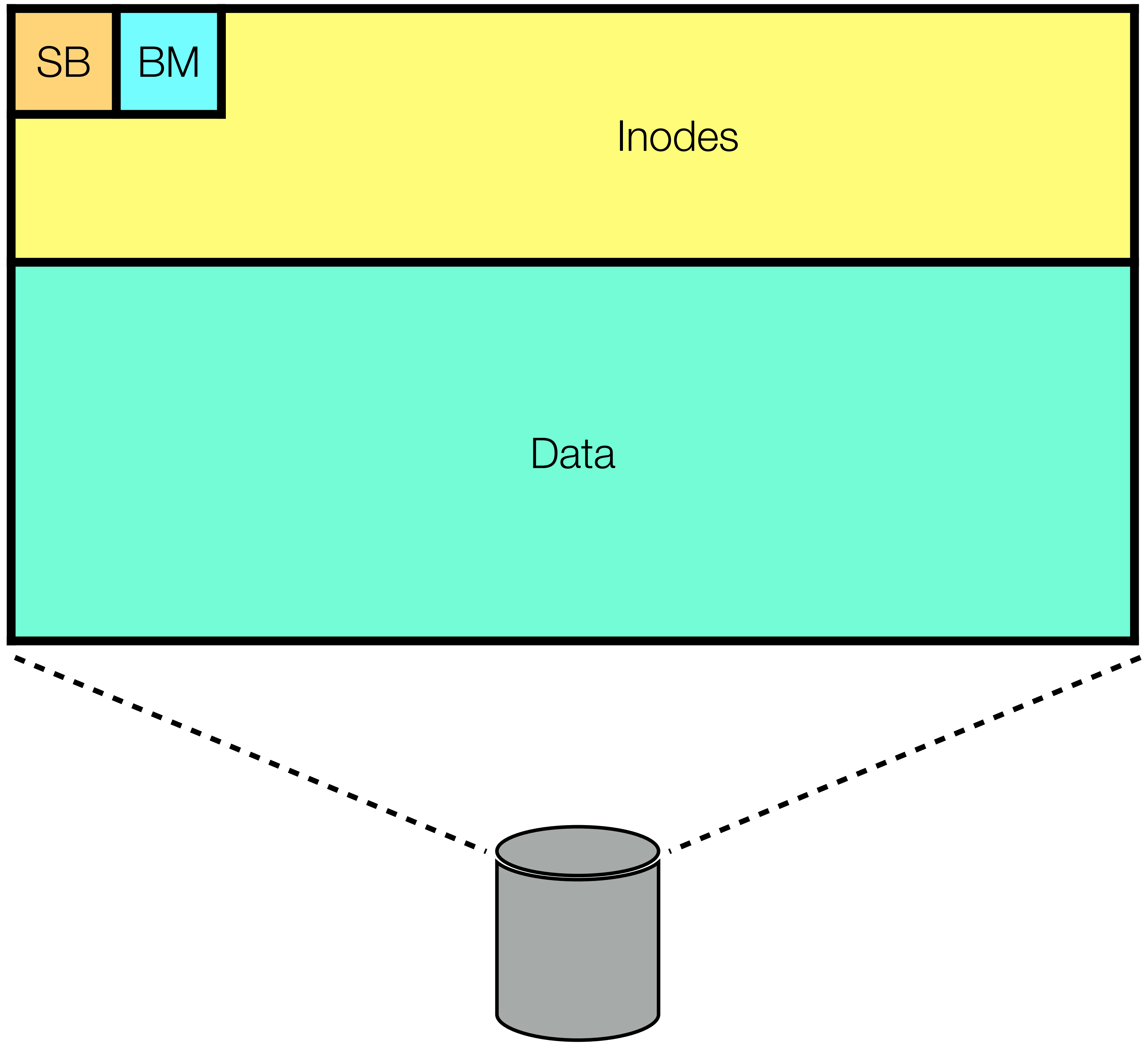
Some “simple” fixes

- Increasing block size
  - Doubling block size more than doubles performance
  - If scaled up, risks wasting space for small files (internal fragmentation)
- Can periodically defragment files (i.e., move all blocks for files next to each other on the disk)
  - Time-consuming and impractical

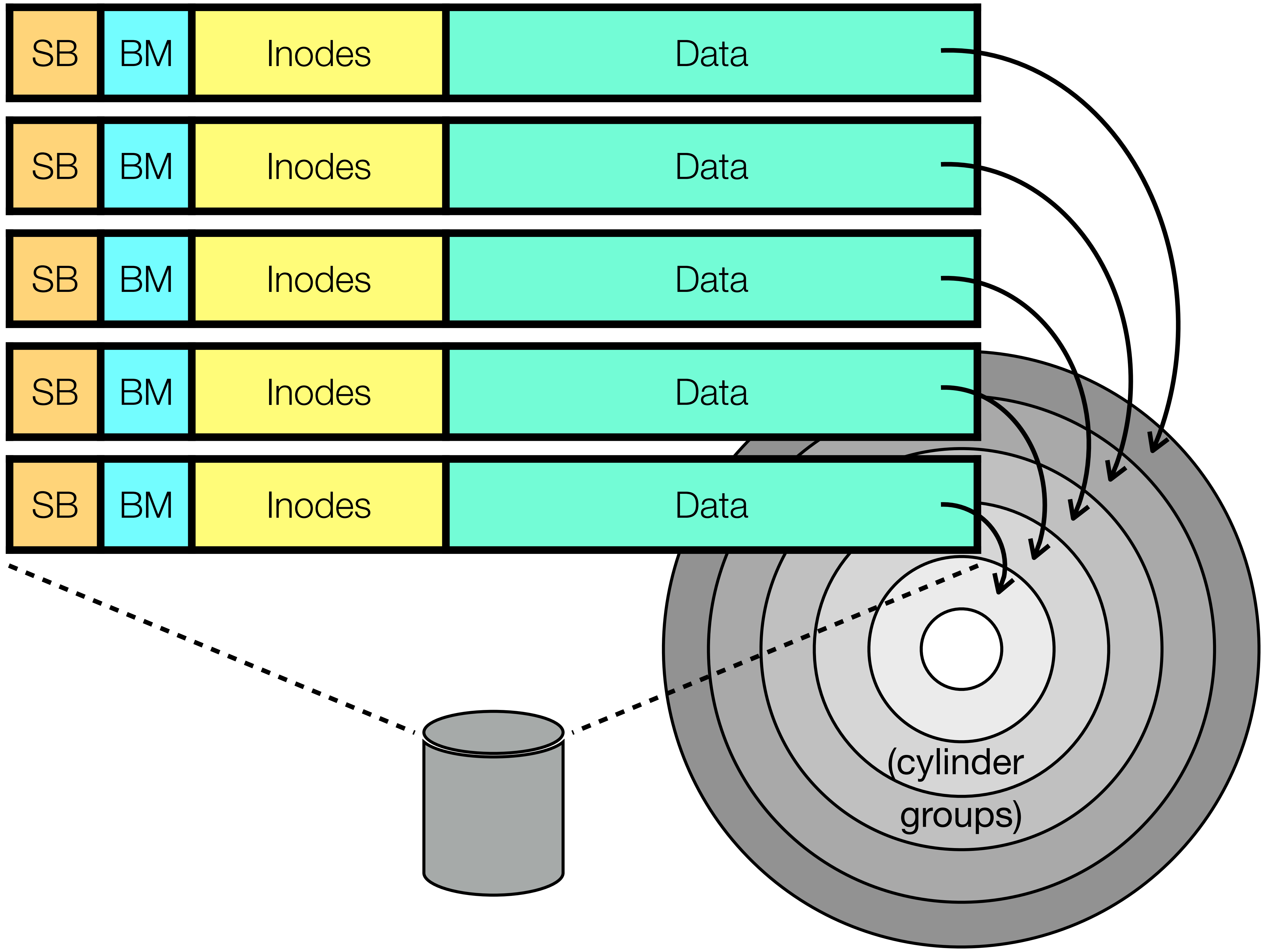
Solution: HDD-aware FS

- Fast File System (FFS) was among the first to heavily optimize for HDDs
  - Inspiration for many modern FSes, including ext2/ext3
- Also introduced important quality-of-life improvements:
  - Long file names
  - Atomic rename
  - Symbolic links

Locality groups



vs.



Smart structure allocation

- Goal: reduce seeks by improving spatial locality of common accesses
  - Create logical block “groups” for related structures
    - Keep file inodes close to the directory inodes that contain them
    - Allocate the first data block of a file close to its inode
      - Keep file inodes, indirect blocks, and data blocks together

Large blocks

- Increased block size up to 8 KB
  - If average file size = 2 KB, how much disk space is wasted?
    - Average file uses 25% of a block; 75% of reserved space is wasted!
  - To mitigate internal fragmentation, the last block of a file may reside in a **block fragment** (512 byte sized)
    - Complicates things when "growing" a file — must copy and coalesce

Takeaways

- At a low level, we should recognize device idiosyncrasies
  - E.g., treat HDD as rotating magnetic platters, not random access memory!
    - (Even RAM is not truly random access memory!)
- But also take care to revisit and update these assumptions
  - SSDs should not be treated like HDDs!

§ FS Consistency and Journaling

What can go wrong?

- Crux of the problem: a FS update may require writing to many disk blocks
  - No way to guarantee they all succeed!
    - Operations may only be partially carried out … to what end?

E.g., growing a file

1. Update in-memory free space bitmap
2. Update in-memory inode (“vnode”)

crash!

3. Write updated inode to disk
4. Write updated free space bitmap to disk

- No persistent structures updated — no FS issues
  - But user may be confused on reboot to find data not saved
    - Compromise: FS can guarantee persistence on explicit flush operation

E.g., growing a file

1. Update in-memory free space bitmap
2. Update in-memory inode ("vnode")
3. Write updated inode to disk

crash!

4. Write updated free space bitmap to disk

- Inode indicates new block is reserved … but block is still marked as free!
  - Dangerous FS inconsistency: block may be reused for another file
    - May manifest as unpredictable data corruption/sharing

E.g., growing a file

1. Update in-memory free space bitmap
2. Update in-memory inode (“vnode”)
3. Write updated free space bitmap to disk

crash!

4. Write updated inode to disk

(swapped from before)

- Block is marked as allocated … but not actually in use by any file
  - “Lost” space, but no real danger (compared to previous scenario)

E.g., deleting a file (last link)

1. Update directory structure (detecting that # links to inode = 0)
2. Mark inode block and all data blocks as free in bitmap
3. Write updated free space bitmap to disk

crash! ···

4. Write updated directory to disk

- All file-related blocks are now marked as free … but the inode that ties them together is still linked from a directory
   - Dangerous free-space-in-use situation again!

E.g., deleting a file (last link)

1. Update directory structure (detecting that # links to inode = 0)
2. Mark inode block and all data blocks as free in bitmap
3. Write updated directory to disk (swapped from before)

crash!

4. Write updated free space bitmap to disk

- "Orphaned inode" situation — inode is still allocated and refers to data blocks, but it has no links
  - Preferable to potential data corruption

Soft updates

- Imminent data corruption vs. storage "leak"
  - Latter is the lesser of two evils
- **Soft updates** is a system of ordering on-disk structure updates such that FS inconsistencies are limited to lost space
  - But we don't want to lose space forever!

FSCK

- Manually walk through all FS metadata (superblock, inodes, directories)
  - Allocated inodes with 0 links can be freed
  - Allocated blocks with no referencing inodes can be “garbage collected”
- Unix fsck utility can report:
  - Orphaned inodes, incorrect link counts, “lost” data blocks, incorrect superblock counts, etc.
  - Also recovers “lost and found” data

We can do better!

- Soft updates is not trivial to implement, and requires frequent flushing
    - Certain structures *must* be written before others
        - May interfere with caching policies
- FSCK is time-consuming, and there is no way to restore system to a known prior state
    - I.e., fixes restores consistency, but the end result may not reflect a logical “snapshot” of the FS at any particular time

Journaling

- Simple idea:
  A. Write down what you're about to do
  B. Go and do it
- If system crashes during A, no harm done
- If system crashes after A but before B finishes, we can "replay" A
  - If necessary, finish up

E.g., journaling

① journal log →

<transaction>

- update inode X with data B_x
- update free-space bitmap M with data B_m
- update data blocks I, J, K, … with data B_i, B_j, B_k, …

② journal commit → </transaction>

③ checkpoint → Perform transaction contents (update blocks X, M, I, J, K, …)

④ free → Mark journal entry as completed and free for reuse

E.g., journaling

① journal log →

<transaction>

update inode X with data B_x

update free-space bitmap M with data B_m

crash!

update data blocks I, J, K, … with data B_i, B_j, B_k, …

- Transaction not committed and not started
  - Nothing to do but delete the partial transaction record — no FS inconsistencies to worry about

E.g., journaling

① journal log → <transaction>

update inode X with data B_x

update free-space bitmap M with data B_m

update data blocks I, J, K, … with data B_i, B_j, B_k, …

② journal commit → </transaction>

crash! ③

- Journal entry committed but checkpoint not complete
- Simply replay the journal entry!

Managing overhead

- Journal is treated as a "circular log" — entries can be reused when done
- But still a huge **write-twice penalty**!
  - Every block is written twice: once to journal, once to final destination
- Can drastically reduce overhead with a **semantic** / **metadata** journal
  - Data block contents are not written to journal, but rather update data blocks at final destinations before creating journal entry
    - Avoids FS consistency issues, but partial data updates are possible

Eliminating write-twice?

- Clever idea: the filesystem *is the journal*
  - Just keep appending new entries to the journal instead of overwriting existing metadata/data
  - To get the current state of any file, replay the journal
    - Periodically save checkpoints to limit replay, and garbage collect unreachable blocks
- Inspiration for **log-structured filesystems**
  - Not very practical for HDDs (high fragmentation), but work well in SSDs!


That's all Folks!