Crash Consistency: FSCK and Journaling

Chapter 42

Previously in CS212…

- We looked at a new take on the file system abstraction
- Details of the FFS
- How we optimize the file system using caches
- But…caches tend to be fast but volatile…what do we do?

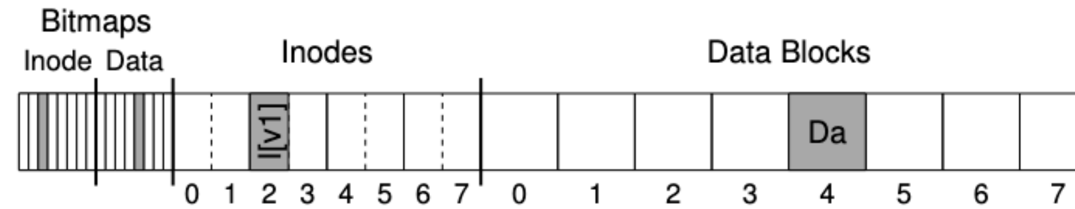
Accidents Happen

- Suppose you lose power at your residence
- Suppose your computer encounters an error it cannot recover from, and it crashes the system
  - Windows - ”Blue Screen” :’(
  - MacOS Linux - Kernel Panic
- How do we update all the necessary data structures to keep track of our filesystem?

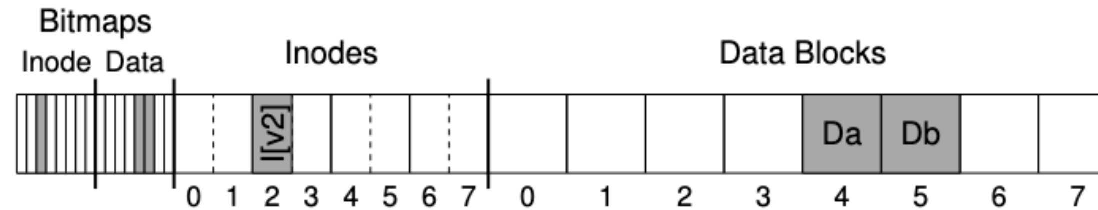
Crash-Consistency Problem

- Disks can only perform one request at a time
- Most reads and writes touch multiple data structures on disk
  - Like bitmaps, inodes, and data blocks
- If we were to be unable to complete all the necessary writes the system is now in an **inconsistent** state (partially updated)
- How can we ensure that the disk is resilient to these issues?

Example




```
owner        : remzi
permissions  : read-write
size         : 1
pointer      : 4
pointer      : null
pointer      : null
pointer      : null
```




```
owner        : remzi
permissions  : read-write
size         : 2
pointer      : 4
pointer      : 5
pointer      : null
pointer      : null
```

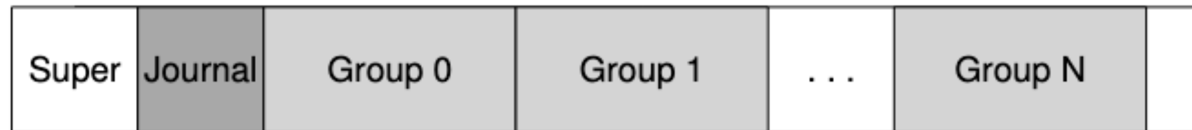
- If only the data is written it’s not a **file-system crash consistency** issue as the metadata is ”correct”, but does mean we have lost data
  - Similar if we successfully wrote to the bitmap and the indoes but not the data block
- Any other combination of one metadata block and/or the data block leave the metadata in an inconsistent state.
- I get it already, what do we do!

The File System Checker

- Allow the inconsistencies to happen and fix them later during a reboot
    - Unix tool fsck
- Systematically scans the disk and checks the
    - Superblock
    - Free blocks (bitmaps)
    - Inode state
    - Inode links
    - Duplicate pointers in Inodes
    - Bad block pointers in Inodes
    - Directory contents
- Problem…
    - Super slow! Imagine a very large filesystem.

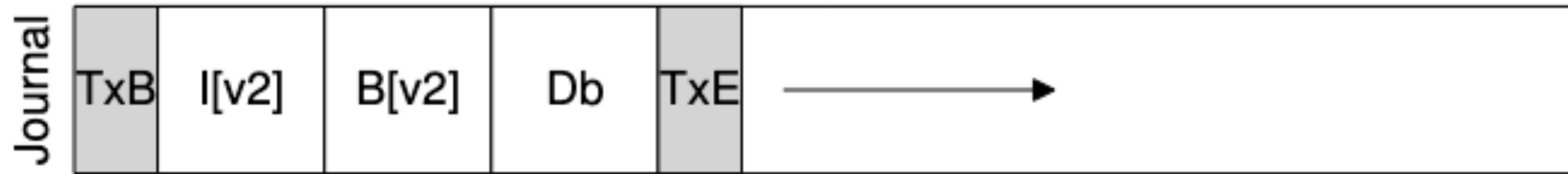
Journaling (Write-Ahead Logging)

- A concept lifted from database management systems
- Add a bit of record keeping before each write



- Before you change stuff on disk, write an entry that outlines what you'll be doing to the **journal**
- But…that's more data written to the disk (overhead)
  - Yes, with the hope of saving time during recovery

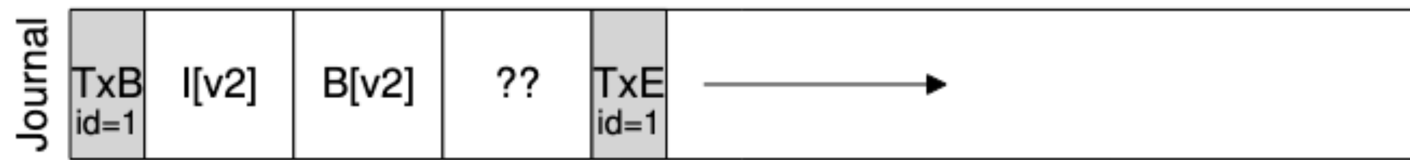
How data journaling works



- TxB indicate the beginning of a transaction that provides
  - A transaction identifier
  - Information about the update like block addresses
- I[v2], B[v2], Db
  - The actual content of the data blocks of data that need updating
- TxE indicates the end of a transaction that provides
  - A matching transaction identifier
- With the transaction recorded, we then **checkpoint** the filesystem with the transaction information (I[v2], B[v2], Db)

What if the crash happens during journaling?

- Well, we could write each journal data unit one at a time
  - Very slow…
- We could write all of it at once (sequential)
  - Fast, but the order is not guaranteed



- Better approach
  - Journal Write – Write transaction begin and block data to the log (wait for complete)
  - Journal Commit – Write the TxE to end the transaction (needs to be a 512-byte block)
  - Checkpoint – write the contents of the transaction to disk

- To limit write traffic overhead, some file systems opt to buffer the creation of journal transactions

Recovery with Journaling

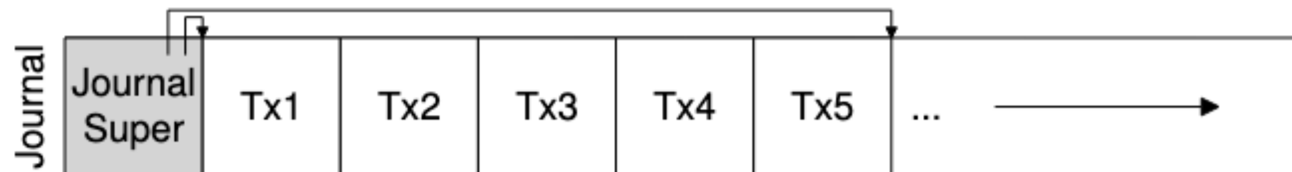
- If the OS crashes before the journal entry is complete it is skipped
  - Journal commit operation was not completed

- If the journal commit operation is complete the OS can crash anytime after, and the transactions can be recovered

- Journal is replayed in order updating the on-disk structures
  - Worst case, some updates are partially performed before the crash, and the operation is duplicated

The Journaling Log

- Journaling space is finite
  - We don't need a list of ALL transactions especially if they have completed successfully
  - The longer the log, the longer the recovery
- If the log fills, we can't write to disk (safely)
- Takes the form of a reusable **circular data structure**
- We can add some metadata for the log and mark transactions as free when we are complete
  - New step that happens after a successful checkpoint



Data Journaling vs Ordered Journaling

- We’ve been talking about Data Journaling (because the transaction includes the data)
  - This means EVERYTHING is written twice!
  - Halves the peek read and write bandwidth of the drive
- What if we didn’t write EVERYTHING to the journal
  - Ordered journaling (aka: Metadata Journaling)
- If we write the user data to disk first, we can use the journal only to hold the necessary pointers (bitmaps and inodes) and update that later
  - NOTE: Directories ARE included in the journal as they are considered metadata
- This approach is at the core of most modern journaled file systems

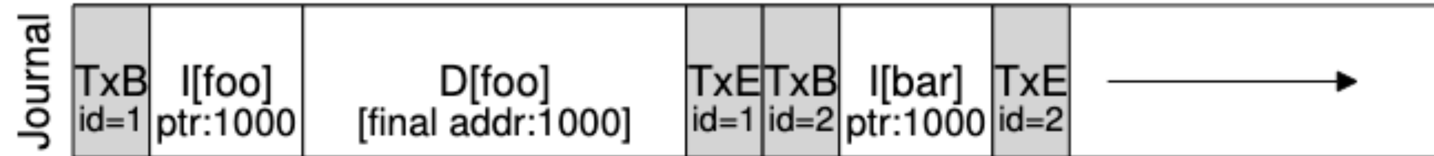
Ordered Journaling Steps

1. Data write – Write user data to disk
2. Journal metadata write – Write begin and metadata to log and WAIT FOR THE WRITES TO COMPLETE
3. Journal commit – Write the transaction end block WAIT FOR THE WRITE TO COMPLETE
4. Checkpoint metadata – write metadata to disk
5. Free – mark the journal entry as available for overwrite

- Note: You can do issues operations for 1 and 2 concurrently but they must be done before operation 3

Edge Case

- Assume we have directory foo, and we add a file to it
- We then delete that directory and create a file named bar
- The file bar is stored in the same block address where the directory foo existed



- With ordered journaling, during a replay after a crash, the directory would overwrite the file bar stored on disk!
- In Linux ext3, a new record is added to the journal called **revoke**
  - Any entry identified by a revoke record is not replayed

Next Time

- EXAM REVIEW!