Crash Consistency

Operating Systems

Moshe Sulamy

Tel-Aviv Academic College

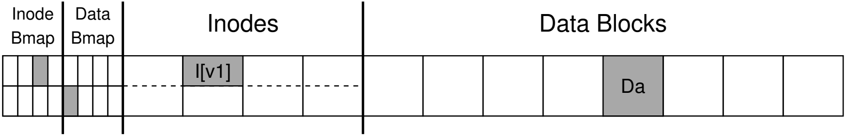
Crash Consistency

- File system data structures must **persist**
  - Major challenge: update structures despite **power loss** or **system crash**
- **Crash-consistency problem**
  - On-disk structures left in an **inconsistent** state

Crashes can occur at arbitrary points in time.
How do we ensure the file system remains valid?

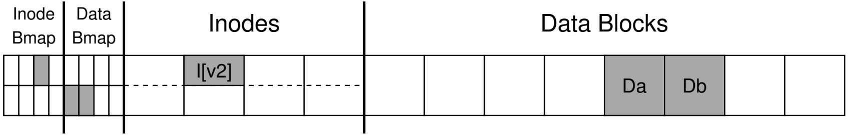
Detailed Example

- Workload: append 4KB data block to existing file
  - Open file → `lseek()` → issue 4KB write → close file
- Simple file system
  - **inode bitmap**, **data bitmap**, inodes, and data blocks



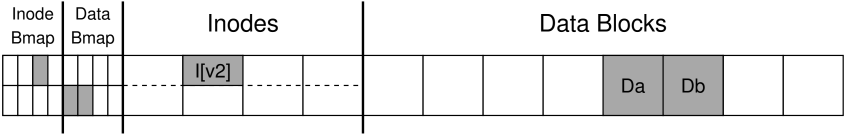
Detailed Example

- Write three blocks to disk:
  - Updated inode, updated data bitmap, data block
  - Final on-disk image:



Detailed Example

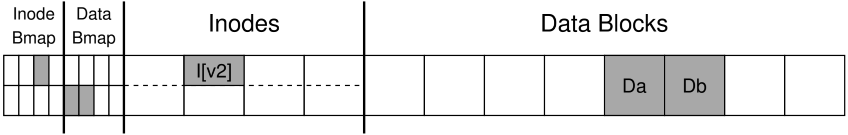
- Write three blocks to disk:
  - Updated inode, updated data bitmap, data block
  - Final on-disk image:



- Usually not immediate
  - Data in main memory (**page cache** or **buffer cache**)
  - File system issues write requests later

Detailed Example

- Write three blocks to disk:
  - Updated inode, updated data bitmap, data block
  - Final on-disk image:



- Usually not immediate
  - Data in main memory (**page cache** or **buffer cache**)
  - File system issues write requests later
- What if a crash happens after one or two writes?

Crash Scenarios

- Only a single write succeeds:
  - **Data block**
    - Data is on disk
    - No inode points to it
    - No bitmap says it is allocated
    - Not a problem! (except lost data for the user)
  - **Updated inode**
    - The inode points to Db, but no data
    - Will read **garbage** data from disk
    - **File-system inconsistency**: on-disk bitmap says block is free
  - **Updated bitmap**
    - Bitmap indicates block is allocated
    - No inode points to it
    - Inconsistent: **space leak**

Crash Scenarios

- Two writes succeed and the last one fails:
  - **Updated inode and bitmap**
    - File system metadata is consistent
    - Data block has garbage data
  - **Updated inode and data block**
    - The inode pointing to correct data
    - Inconsistency: bitmap shows block as free
  - **Bitmap and data block**
    - Inconsistency between inode and data bitmap
    - Block was written but no idea which file it belongs to

Crash Scenarios

- Ideally: move file system from one consistent state to another **atomically**
  - But disk only commits one write at a time
  - Crashes or power loss may occur
- Called: **crash-consistency problem**

Solution #1

- Let inconsistencies happen, fix them later
- The File System Checker (`fsck`)
  - UNIX tool for finding and repairing inconsistencies
  - Can't fix all problems
    - e.g., file system looks consistent but inode points to garbage data
  - Only goal: keep file system metadata consistent

Solution #1

- Summary of what `fsck` does:
  - **Superblock**: make sanity checks
    - e.g., file system size greater than allocated blocks
    - Suspect superblock? may use alternate copy
  - **Free blocks**: scan inodes to produce correct version of bitmap
    - Inconsistency $\rightarrow$ trust information within inodes
    - Same check is performed for all the inodes (inode bitmaps)
  - **Inode state**: each inode is checked for corruption
    - Also indirect blocks, double indirect blocks
    - e.g., valid type field (regular file, directory, symbolic link, etc.)
    - Problems not easily fixed? clear inode

Solution #1

- Summary of what `fsck` does:
  - **Inode links**: verify link count of each allocated inode
    - Scans through entire directory tree
    - Mismatch: fix count within the inode
    - No directory refers to it: moved to `lost+found` directory
  - **Duplicates**: check for duplicate pointers
    - Two different inodes refer to same block
    - Clear bad inode, or copy pointed-to block
  - **Bad blocks**: check bad block pointers outside valid range
    - While scanning through list of pointers
    - e.g., block address greater than partition size
  - **Directory checks**: additional integrity checks on contents
    - Directories contain specifically formatted information
    - `"."` and `".."` are the first entries
    - Each referred inode is allocated
    - No directory is linked to more than once

File System Checker

- Building `fsck` requires intricate knowledge of file system
- Fundamental problem: too slow
  - Scan entire disk to find allocated blocks
  - Read entire directory tree
  - Can take many minutes or hourse
- Consider previous example:
  - Just three blocks are written to disk
  - Incredibly expensive to scan entire disk for problem during an update of three blocks

Solution #2

- **Journaling (write-ahead logging)**
  - Add a bit of work during updates $\rightarrow$ reduce work during recovery
- How journaling works:
  - Before overwriting structures: write a note describing operation
  - Note is "write ahead" part, structure organized as "log"
  - If a crash takes place, go back and look at the note to try again
  - Know exactly what/how to fix instead of scanning entire disk

Journaling

- **Linux ext3**
  - Most on-disk structures identical to **Linux ext2**, with journaling
  - e.g., block groups, each contains inode and data bitmaps, inodes, data blocks

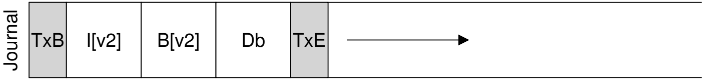
Journaling

- **Linux ext3**
  - Most on-disk structures identical to **Linux ext2**, with journaling
  - e.g., block groups, each contains inode and data bitmaps, inodes, data blocks

- New key structure: journal itself
  - Occupies small amount of space within partition or another device

Data Journaling

- Example: our canonical update
  - Wish to write inode (I[v2]), bitmap (B[v2]) and data block (Db)
- Before writing to final disk locations, first write to log (journal)



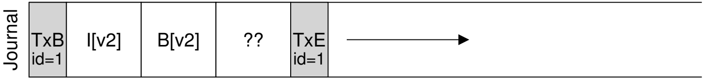
- TxB: transaction begin block
- Middle three blocks: exact content of blocks themselves
  - **Physical logging**
  - Alternate idea: **logical logging**
- TxE: transaction end block

Data Journaling

- **Checkpoint**
  - Transaction safely on disk $\rightarrow$ ready to overwrite structures in file system
- Initial sequence of operations:
  - **Journal write**: write the transaction
    - Including transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log
    - Wait for writes to complete
  - **Checkpoint**
    - Write pending metadata and data updates to final locations

Data Journaling

- Crash occurs during write to the journal:
  - Issue one write at a time
    - Wait for each to complete, then issue next
    - Slow
  - Write all five blocks at once
    - Five writes into one sequential write → faster
    - Unsafe: may crash between writes
    - Looks like a valid transaction

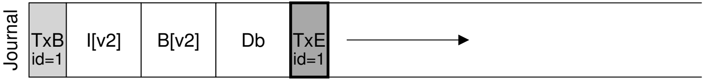


- Bad for user data, worse for critical piece, e.g., superblock

Data Journaling

- Issue transactional write in two steps
  1. Write all blocks except TxE

  2. When writes complete, issue write of TxE



Data Journaling

- Our current protocol:
  - **Journal write**
    - Write transaction contents (TxB, metadata, and data)
    - Wait for these writes to complete
  - **Journal commit**
    - Write transaction commit block (TxE)
    - Wait for write to complete $\rightarrow$ transaction is **committed**
  - **Checkpoint**
    - Write pending metadata and data updates to final locations

Recovery

- Crash before transaction written to log?
  - Our job is easy: pending update is skipped
- After transaction committed, before checkpoint is complete?
  - Can **recover** the update
  - On boot, scan log for committed transactions to **replay**
  - Called **redo logging**

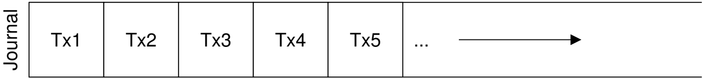
Batching Log Updates

- Create two files in the same directory
  - Commit to journal twice
  - Write same blocks over and over
- Solution: buffer updates to a **global transaction**
  - Mark as dirty: in-memory inode bitmap, inodes, directory data, directory inode
  - Add to list of blocks that form current transaction

  - Commit a single global transaction $\rightarrow$ avoid excessive writes

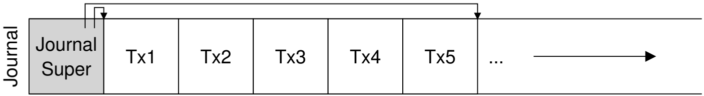
Finite Log

- Two problems with log of finite size
  1. The larger the log, the longer the recovery
  2. Log full → no more transactions can be committed



Finite Log

- Solution? **circular log**
  1. Take action some time after a checkpoint
  2. Free the log space used by the transaction
- **Journal superblock**
  - Marks oldest and newest non-checkpointed trnsactions

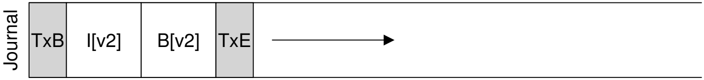


Finite Log

- Another step in our protocol:
  - **Journal write**
    - Write transaction contents (TxB, metadata, and data)
    - Wait for these writes to complete
  - **Journal commit**
    - Write transaction commit block (TxE)
    - Wait for write to complete $\rightarrow$ transaction is **committed**
  - **Checkpoint**
    - Write pending metadata and data updates to final locations
  - **Free**
    - Some time later, mark transaction as free in journal
    - By updating the journal superblock

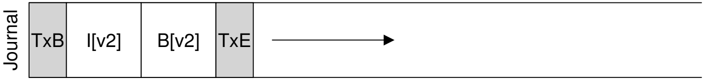
Metadata Journaling

- Still a problem: Writing every data block twice
  1. Commit to log
  2. Checkpoint to disk
- **Ordered journaling**: user data is not written to the journal
  - Write data block to file system proper



Metadata Journaling

- Still a problem: Writing every data block twice
  1. Commit to log
  2. Checkpoint to disk
- **Ordered journaling**: user data is not written to the journal
  - Write data block to file system proper



- Write data after transaction?
  - Problem → may point to garbage data
- Write data before transaction → avoids problem

Metadata Journaling

- The protocol:
  - **Data write**
    - Write data to final location
    - Wait for completion
  - **Journal metadata write**
    - Write transaction contents (TxB and metadata)
    - Wait for writes to complete
  - **Journal commit**
    - Write transaction commit block (TxE)
    - Wait for write to complete $\rightarrow$ transaction is **committed**
  - **Checkpoint**
    - Write pending metadata and data updates to final locations
  - **Free**
    - Later, mark the transaction free in journal superblock

Summary

- **Crash consistency problem**
  - On-disk structures left in an inconsistent state
- File System Checker (`fsck`)
  - Let inconsistencies happen, fix them later
  - Scan superblock, free blocks, inode state, inode links, duplicates, bad blocks, directories
  - Keep file system metadata consistent
  - Fundamental problem: too slow
- **Journaling** (**write-ahead logging**)
  - Add work during updates $\rightarrow$ reduce work during recovery
  - Before writing structures: write note describing operation
  - **Checkpoint**: transaction safely on disk
  - Batch log updates, **circular log**, **metadata journaling**