Crash Consistency

How to recover from failure?

By Matthew Tancreti
For COM S 352
Iowa State University

Crash Consistency

Problem of **crash consistency**

Power loss or system crash can happen between any two writes, thus on-disk state may only be partially updated

File system has inconsistent state when system reboots

How to recover from failure?

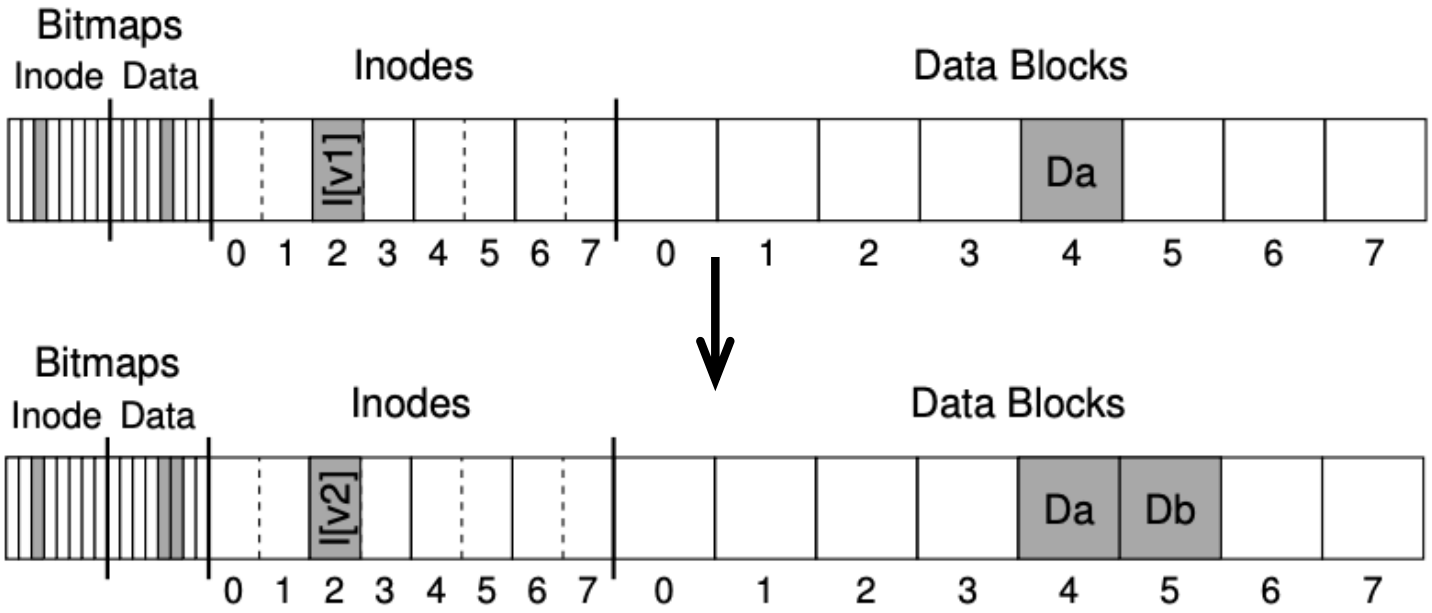
Hard drive internal [source]

Example Operation

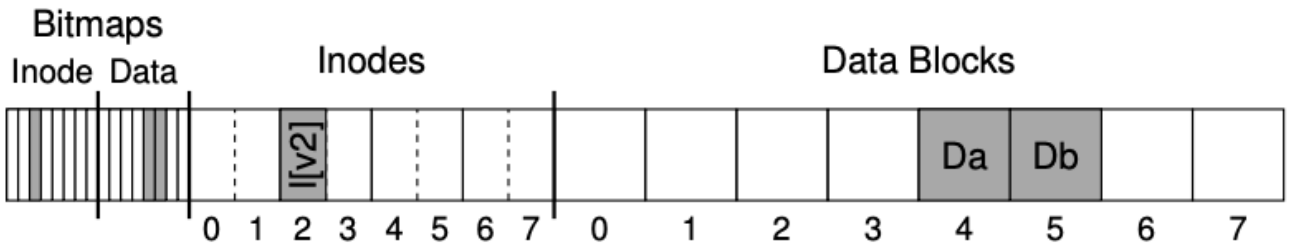
Example: append a single block of data to an existing file

Requires writing new data block, updating inode and updating bitmap

For performance, blocks may be written physically to the disk in any order (e.g., caching and write buffering)



Crash Scenarios



Scenario 1: Just data block *Db* is written – data is on disk but no inode points to it, as if write never happened, from perspective of crash consistency this is not a problem

Scenario 2: Just inode *I[v2]* is written – inode points to data block that has not been written, result is garbage data in file

Scenario 3: Just updated bitmap *B[v2]* is written – bitmap indicates data block is allocated but no inode points to it, the data block will never be freed from memory – **space leak**

But wait, there's more…

Scenario 4: Only inode and bitmap are written to disk – inode and bitmap are consistent, but data has garbage

Scenario 5: Only inode and data block are written to disk – the bitmap has not been updated to allocate the datablock, in future data block could be overwritten

Scenario 6: Only bitmap and data block are written to disk – space is allocated but not used, will never be freed

Crash Recovery with File System Checker

First strategy, let inconsistencies happen and them fix them later (when rebooting)

Unix tool **fsck** (file system check) is used to check and fix file system on boot

Steps to check and recover file system:

- Superblock – check superblock, basic sanity check that number of allocated block is not greater that the file system size
- Free blocks – scan inodes, indirect blocks and double indirect blocks to construct a correct version of bitmaps
- Inode state – check each inode for corruption
- Inode links – scan directories to make correct count of links to inodes
- Duplicates – check for duplicate pointers, where two inodes point to the same block
- Bad blocks – check for clearly wrong pointers, for example, they point out of range
- Directory checks – check for directory corruption, for example directory hierarchy should be a tree

Disadvantage - need to check entire file system, slow for big file systems

Journaling (Write-ahead logging)

Second strategy, before making any change to the file system, first store a note to the disk about what operations are about to be performed

Write ahead of the operation into a log, called **write-ahead logging** or **journaling**

If crash occurs you know from log exactly what needs to be checked and fixed, no need to scan entire file system for errors

File System Organization with Journal

Assume the use of groups (like in Fast File System), an extra region is set aside for a journal

Physical Logging

In **physical logging** the complete contents of the update are first written to a log

Protocol (version 1):

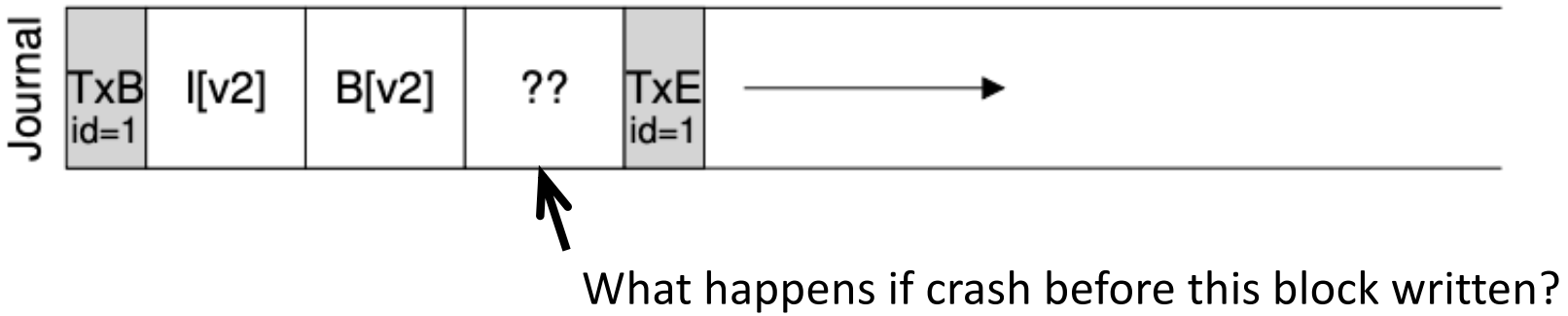
1. **Journal write** – write the full metadata and data to be updated (inode, bitmap and data block) to log
2. **Checkpoint** – write the metadata and data to their final locations in the file system

What happens when crash during write to journal?

The journal entry is written in a transaction (e.g., TxB, I[v2], B[v2], Db, TxE) where TxB is transaction begin and TxE is transaction end

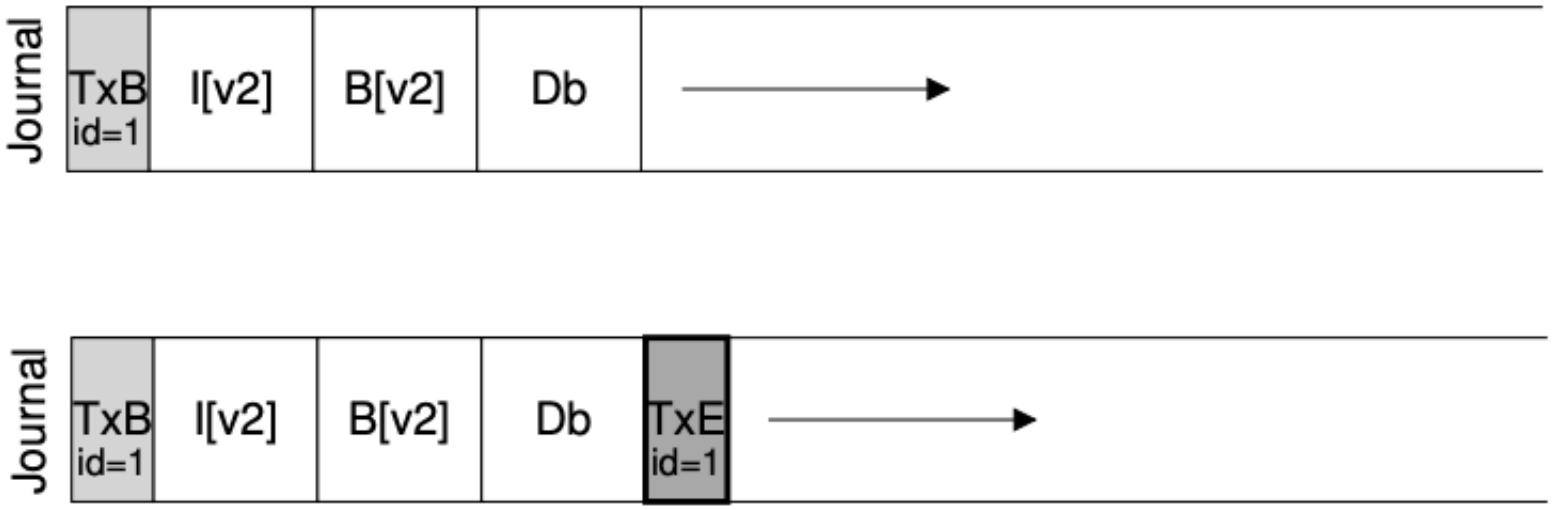
If blocks are written in order one after the other, can simply check for TxE

To speed up file system, blocks can be written out of order



Two step solution

Two steps: first write everything expect for TxE, then write TxE when done

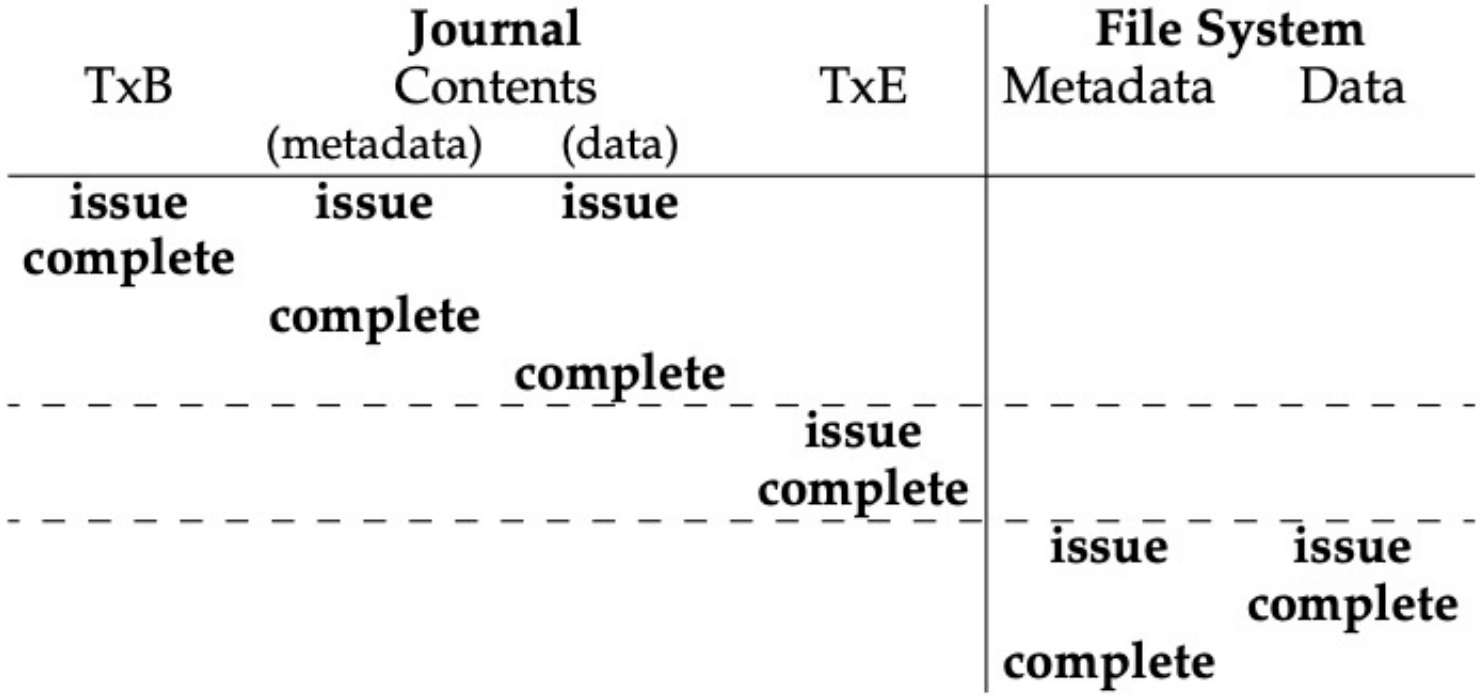


Journal Protocol

Protocol (version 2):

1. **Journal write –** write the contents of the transaction (TxB, metadata and data)
2. **Journal commit –** write the transaction commit block (TxE)
3. **Checkpoint –** write the metadata and data to their final locations in the file system

Data Journaling Timeline

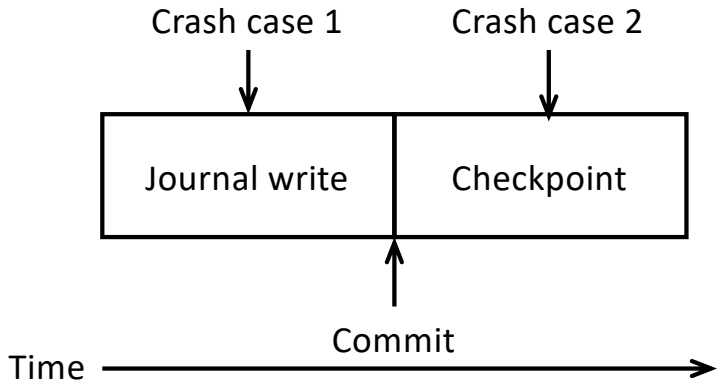


Recovery

Two cases

1. If crash happens before transaction written to log, simply ignore the pending update, the file system is in a consistent state
2. If crash happens after commit to log, but before checkpoint, during reboot replay every committed transaction in order

Doesn't matter when during checkpointing crash occurs, some of the replayed transactions will just repeat what is already on the disk, end result is a consistent file system



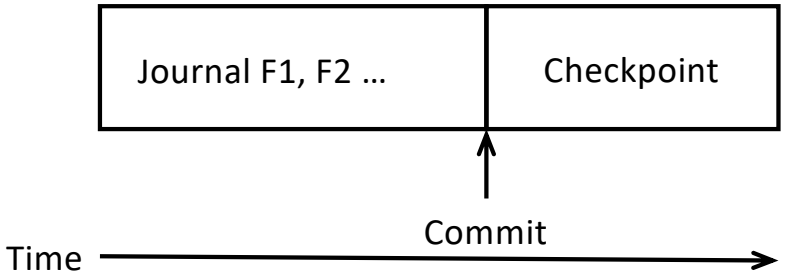
Batching Log Updates

Logging can create a lot of disk traffic

Suppose a directory gets modified several times to write to multiple files, every modification requires logging the directories inode, writing the same block over and over

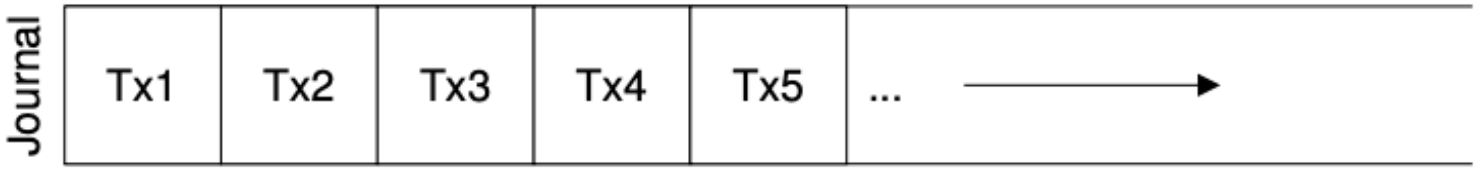
Some file systems use buffering, multiple updates go into one global transaction

All writes within a time period (e.g., 5 seconds) are combined into a single transaction



Finite Log

Log continues to grow with every transaction, can fill disk



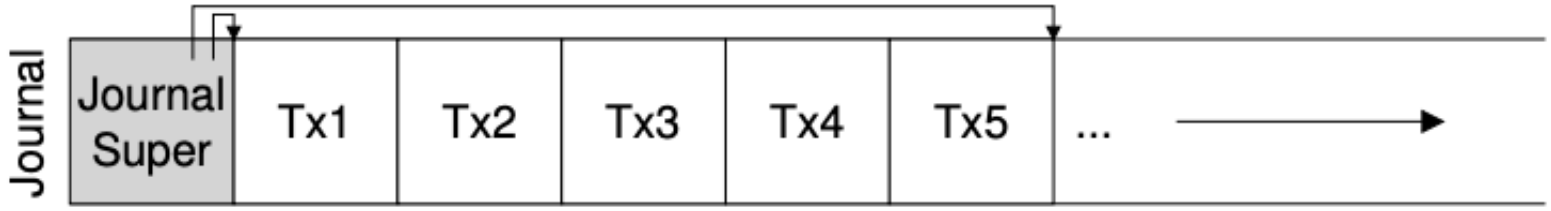
Idea: transactions are only need to replay when crash happens during checkpointing, after a checkpoint is complete the transaction can be removed

Journal Superblock

Write transaction into a circular log (new transactions written over old freed transactions)

Journal superblock points to the first and last valid transaction in the circular log

When checkpoint complete first transaction is freed by changing pointer to the next transaction



Adding Free Step

Protocol (version 3):

1. **Journal write –** write the contents of the transaction (TxB, metadata and data)
2. **Journal commit –** write the transaction commit block (TxE)
3. **Checkpoint –** write the metadata and data to their final locations in the file system
4. **Free –** mark the transaction free in the journal by updating the journal superblock

Metadata Journaling

We have now reduced time to recover (replay) and size of log (free), but normal operations of the file system are still slow

Need to write everything to the journal first, doubling write traffic

Idea: reduce writes to log by only logging metadata (bitmaps and inodes) and not data blocks, the data blocks are written directly to the file system

Question: when to write the data blocks to make the file system recoverable?

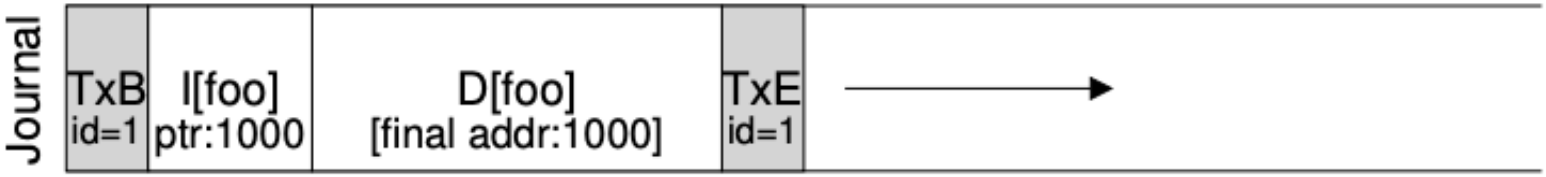
Metadata Journaling Protocol

Protocol (version 4):

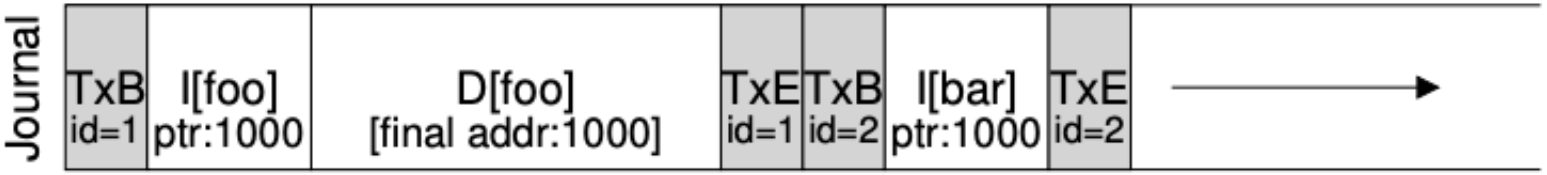
1. **Data write –** write the data to final location and wait for completion
2. **Journal metadata write –** write the begin block and metadata to the log, wait for writes to complete
3. **Journal commit –** write the transaction commit block (TxE) to the log, wait for write to complete
4. **Checkpoint metadata –** write the contents of the metadata update to their final locations within the file system
5. **Free –** mark the transaction free in the journal by updating the journal superblock

Corner cases when deleting files

User creates directory



User deletes directory and creates new file that reuses inode block



Metadata Journaling