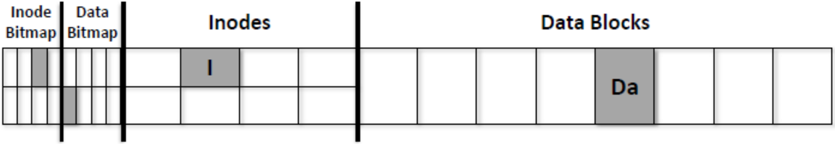
IE411: Operating Systems

Crash consistency

Lets append to a file

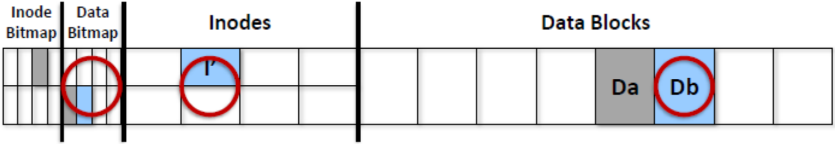
- fs: 8-bit inode bitmap, 8-bit data bitmap, 8 inodes, 8 data blocks



- Before the append
  - Single inode is allocated
  - Single allocated data block
  - The inode is denoted I

Lets append to a file

- After append



  - data bitmap is updated – we have allocated one more data block
  - inode is updated (I') – a new data block, size, and access time
  - new data block is allocated (Db)

Lets append to a file

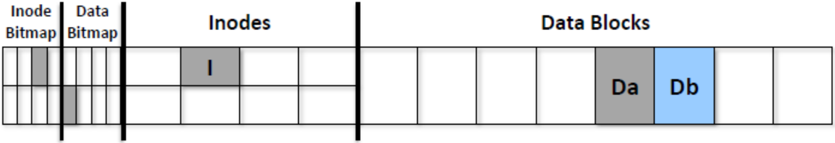
- To do the append, fs performs 3 separate writes to the disk
- Unexpected power loss or system crash → some writes may be completed while others are not
- The file system could be left in an inconsistent state

Crash scenarios

- Assume a crash occurs after only one of the writes has taken place
- 3 possible scenarios
    - Db is only written to disk
    - Data bitmap is only written to disk
    - I' is only written to disk
- Lets see what problems can occur

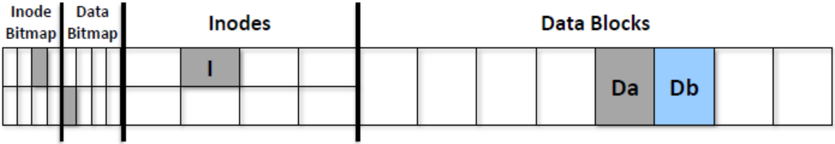
Example: Crash scenarios (1)

- Only data block (Db) is written to disk



Example: Crash scenarios (1)

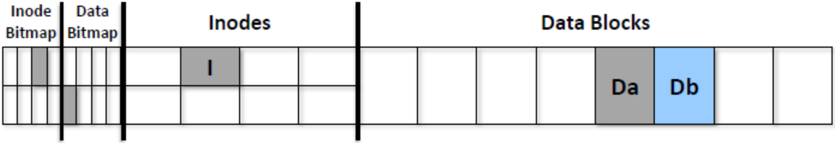
- Only data block (Db) is written to disk



  - no inode points to data block 5 (Db)
  - data bitmap says data block 5 is free
  - it is as if the write never occurred

Example: Crash scenarios (1)

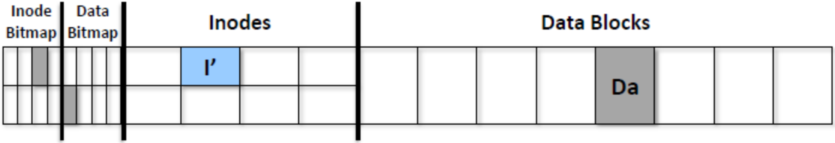
- Only data block (Db) is written to disk



  - no inode points to data block 5 (Db)
  - data bitmap says data block 5 is free
  - it is as if the write never occurred
- file system metadata – completely consistent

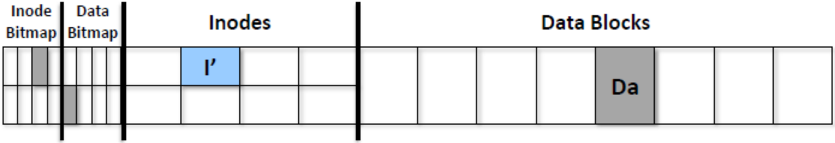
Example: Crash scenarios (2)

- Only updated inode (I') is written to disk



Example: Crash scenarios (2)

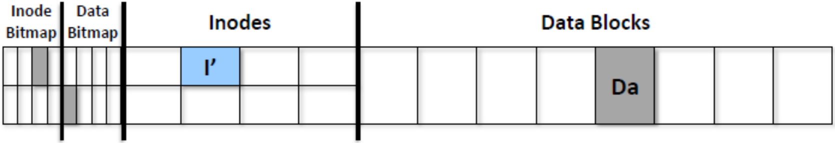
- Only updated inode (I’) is written to disk



  - inode I’ points to data block 5, but data bitmap says it’s free
  - read will get garbage data (old contents of data block 5)
  - if data block 5 is allocated to another file later, the same block will be used by two inodes

Example: Crash scenarios (2)

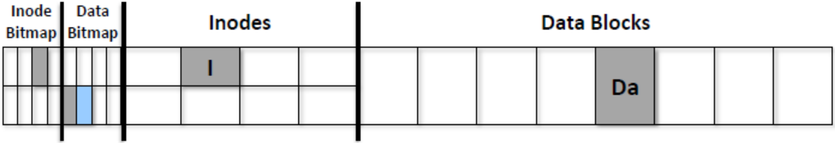
- Only updated inode (I') is written to disk



  - inode I' points to data block 5, but data bitmap says it's free
  - read will get garbage data (old contents of data block 5)
  - if data block 5 is allocated to another file later, the same block will be used by two inodes
- file-system inconsistency

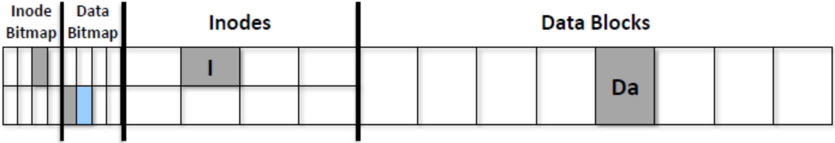
Example: Crash scenarios (3)

- Only updated data bitmap is written to disk



Example: Crash scenarios (3)

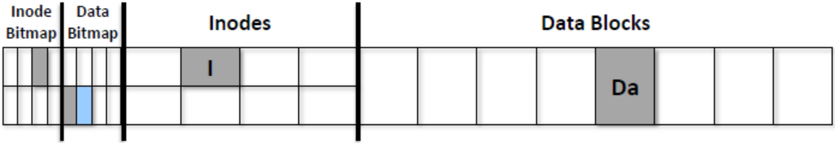
- Only updated data bitmap is written to disk



  - data bitmap says data block 5 is allocated, but no inode points to it
  - data block 5 will never be used by the file system
  - lost data block (space leak)

Example: Crash scenarios (3)

- Only updated data bitmap is written to disk



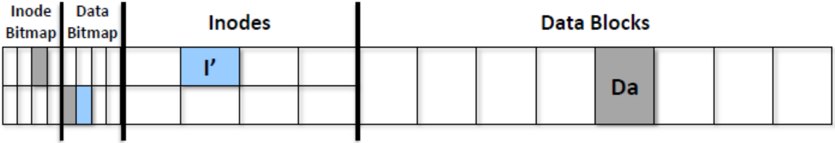
  - data bitmap says data block 5 is allocated, but no inode points to it
  - data block 5 will never be used by the file system
  - lost data block (space leak)
- file-system inconsistency

Crash scenarios

- Assume a crash occurs after only two of the writes have taken place
- 3 possible scenarios
  - Only inode and data bitmap are written to disk
  - Only inode and Db are written to disk
  - Only data bitmap and Db are written to disk

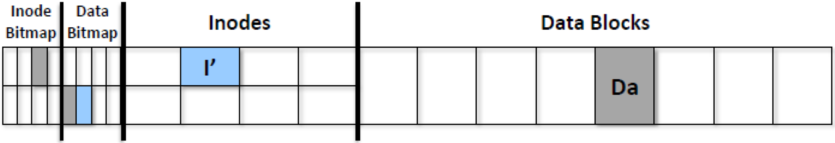
Example: Crash scenarios (4)

- Only inode and data bitmap are written



Example: Crash scenarios (4)

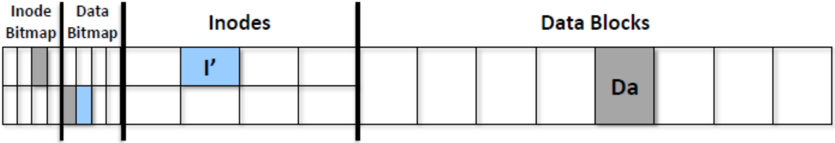
- Only inode and data bitmap are written



  - inode I' has a pointer to data block 5 and data bitmap says it is in use
  - read will get garbage data (old contents of data block 5)

Example: Crash scenarios (4)

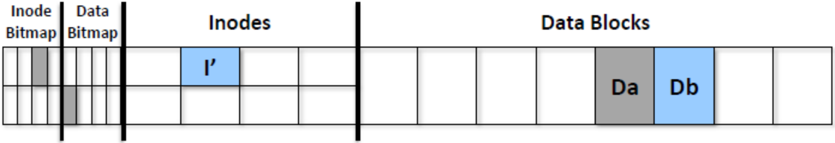
- Only inode and data bitmap are written



  - inode I' has a pointer to data block 5 and data bitmap says it is in use
  - read will get garbage data (old contents of data block 5)
- file system metadata – completely consistent

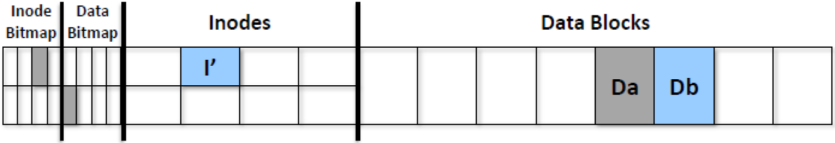
Example: Crash scenarios (5)

- Only inode and data block are written



Example: Crash scenarios (5)

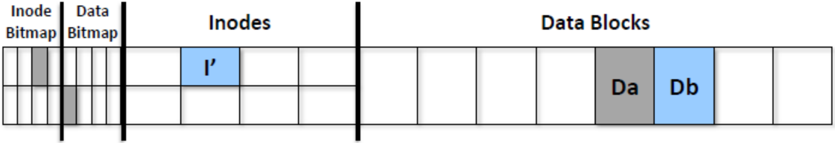
- Only inode and data block are written



  - inode I' has a pointer to data block 5, but data bitmap says it is free
  - data block 5 can be reallocated to another inode

Example: Crash scenarios (5)

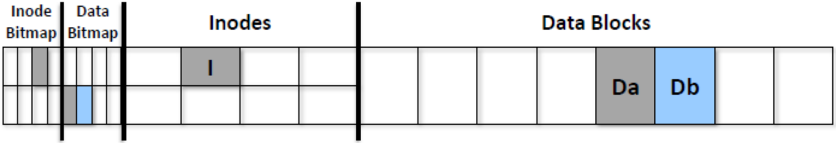
- Only inode and data block are written



  - inode I' has a pointer to data block 5, but data bitmap says it is free
  - data block 5 can be reallocated to another inode
- file-system inconsistency

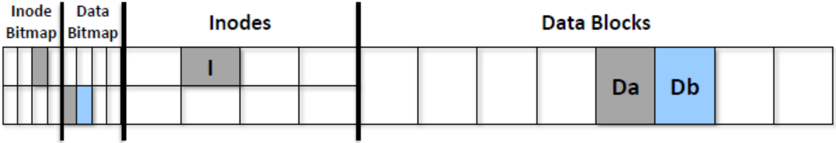
Example: Crash scenarios (6)

- Only data bitmap and data block are written



Example: Crash scenarios (6)

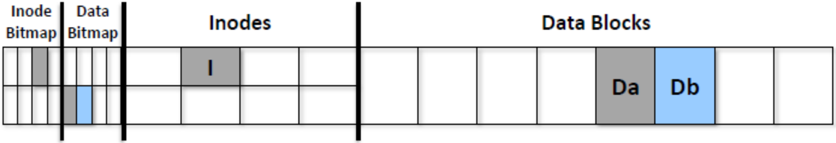
- Only data bitmap and data block are written



  - data bitmap indicates data block 5 is in use, but no inode points to it
  - data block 5 will never be used by the file system
  - lost data block (space leak)

Example: Crash scenarios (6)

- Only data bitmap and data block are written



  - data bitmap indicates data block 5 is in use, but no inode points to it
  - data block 5 will never be used by the file system
  - lost data block (space leak)
- file-system inconsistency

Summary

- 1 block written
  - Db   // ok
  - I[v2]   // inconsistency
  - B[v2]   // inconsistency

Summary

- 1 block written
  - Db // ok
  - I[v2] // inconsistency
  - B[v2] // inconsistency
- 2 blocks written
  - I[v2], B[v2] // fs has garbage
  - I[v2], Db // inconsistency
  - B[v2], Db // inconsistency

Crash consistency problem (consistent update)

- The system may crash or lose power between any two disk writes, and thus the on-disk state may only partially get updated → inconsistent state
- How do we ensure that the file system keeps the on-disk image in a consistent state?

Two approaches

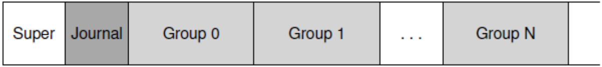
- Approach 1: Fix inconsistent file system during bootup
  - Unix utility called fsck (chkdsk on Windows)
  - fsck is slow because it checks the entire file system after crash

Two approaches

- Approach 1: Fix inconsistent file system during bootup
  - Unix utility called fsck (chkdsk on Windows)
  - fsck is slow because it checks the entire file system after crash
- Approach 2: Use a transaction log to make multi-writes atomic
  - After a crash the log can be replayed to finish updates
  - Journaling file system, e.g. Ext3 and NTFS

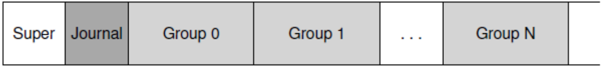
Journaling (Write-Ahead Logging)

- When updating the disk, before overwriting the structures in place, first write them into a log (elsewhere on disk)



Journaling (Write-Ahead Logging)

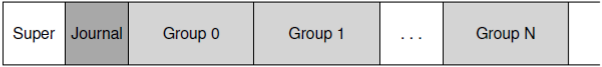
- When updating the disk, before overwriting the structures in place, first write them into a log (elsewhere on disk)



- After the log is written, update the final disk locations

Journaling (Write-Ahead Logging)

- When updating the disk, before overwriting the structures in place, first write them into a log (elsewhere on disk)



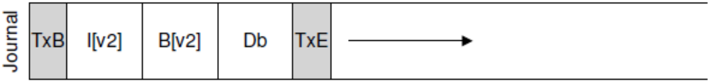
- After the log is written, update the final disk locations
- If a crash takes place during the update, the log has exactly the right information to fix the problem

Data Journaling Example (1)

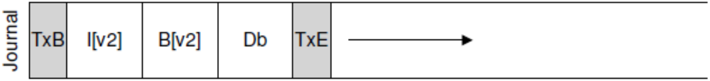
- Assume we are appending to a file
    - Three block writes: inode (I[v2]), bitmap (B[v2]), and data block (Db)

Data Journaling Example (1)

- Assume we are appending to a file
  - Three block writes: inode (I[v2]), bitmap (B[v2]), and data block (Db)
- Before writing them to their final disk locations, first write them to the log

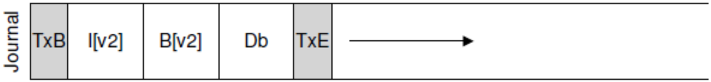


Data Journaling Example (2)



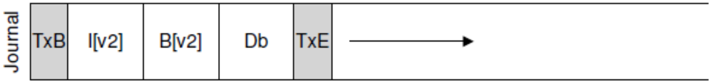
- TxB: Transaction begin block
  - it contains some kind of transaction identifier (TID)
  - it also contains the final disk addresses of the blocks

Data Journaling Example (2)



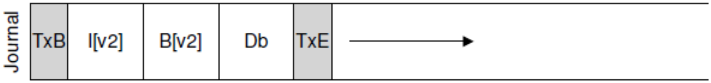
- TxB: Transaction begin block
  - it contains some kind of transaction identifier (TID)
  - it also contains the final disk addresses of the blocks
- The middle three blocks just contain the exact content of the blocks themselves; called physical logging

Data Journaling Example (2)



- TxB: Transaction begin block
    - it contains some kind of transaction identifier (TID)
    - it also contains the final disk addresses of the blocks
- The middle three blocks just contain the exact content of the blocks themselves; called physical logging
- TxE: Transaction end block

Data Journaling Example (2)



- TxB: Transaction begin block
    - it contains some kind of transaction identifier (TID)
    - it also contains the final disk addresses of the blocks
- The middle three blocks just contain the exact content of the blocks themselves; called physical logging
- TxE: Transaction end block
    - a marker of the end of this transaction
    - it also contains the TID

Checkpointing

- Once the transaction is safely written to the journal, we proceed to overwrite the old structures in the file system
- This is called checkpointing

Checkpointing

- Once the transaction is safely written to the journal, we proceed to overwrite the old structures in the file system
- This is called checkpointing
- To checkpoint our examples, issue the writes I[v2], B[v2], and Db to their disk locations

Crash Recovery Using Journal (1)

- If a crash happens during journal write
    - Ignore the half-written transaction during recovery

Crash Recovery Using Journal (1)

- If a crash happens during journal write
    - Ignore the half-written transaction during recovery
    - No checkpointing took place → FS blocks are not changed

Crash Recovery Using Journal (2)

- Suppose a crash happens after journal write but before (or during) checkpointing
- During recovery, replay transaction by writing the recorded changes to FS blocks

Crash Recovery Using Journal (2)

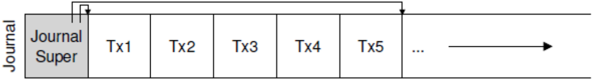
- Suppose a crash happens after journal write but before (or during) checkpointing
- During recovery, replay transaction by writing the recorded changes to FS blocks
- This is correct even if crash happened during checkpointing
  - i.e., even if some FS blocks were written before crash
  - Why?

Crash Recovery Using Journal (2)

- Suppose a crash happens after journal write but before (or during) checkpointing
- During recovery, replay transaction by writing the recorded changes to FS blocks
- This is correct even if crash happened during checkpointing
  - i.e., even if some FS blocks were written before crash
  - Why?
  - Because we will just overwrite them with the same data

Journal Implementation

- Implement the log as a circular buffer
- Deallocate journal transactions that have been checkpointed, allowing the space to be reused
- Journal has its own superblock
  - includes pointers to oldest and newest non-checkpointed transactions



Journal protocol (v1)

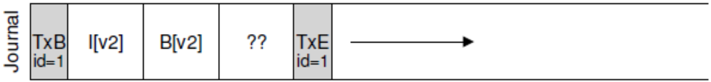
1. Journal write
   - Write the transaction to the log
     - TxB, all pending data, metadata updates, and TxE
   - Wait for these write to complete

Journal protocol (v1)

1. Journal write
   - Write the transaction to the log
     - TxB, all pending data, metadata updates, and TxE
   - Wait for these write to complete
2. Checkpoint
   - Write the pending metadata and data updates to their final locations

Unsafe writing strategy

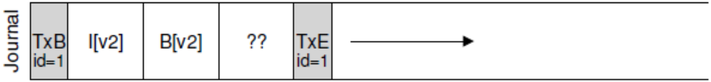
- Remember we need to write all of the following blocks to the journal
    - TxB, I[v2], B[v2], Db, TxE
- Disk internally may perform scheduling and complete the writes in any order
- E.g., the disk may write (1) TxB, I[v2], B[v2], and TxE and only later (2) write Db
- If a crash occurs between (1) and (2), the journal entry looks like this



- Problem?

Unsafe writing strategy

- If a crash occurs between (1) and (2), the journal entry looks like this



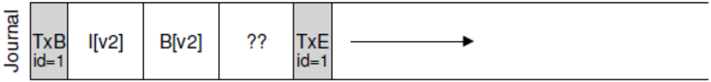
Journal

TxB id=1	I[v2]	B[v2]	??	TxE id=1

- Problem?

Unsafe writing strategy

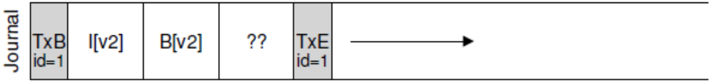
- If a crash occurs between (1) and (2), the journal entry looks like this



- Problem?
    - Transaction looks valid, but the data is missing

Unsafe writing strategy

- If a crash occurs between (1) and (2), the journal entry looks like this



Journal

TxB id=1 | I[v2] | B[v2] | ?? | TxE id=1

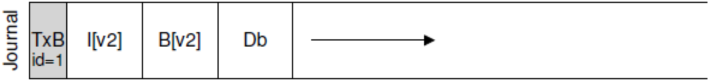
- Problem?
  - Transaction looks valid, but the data is missing
  - During replay, garbage data is written to the file system

Safer writing strategy

- Issue TxB and everything up to (but not including) TxE

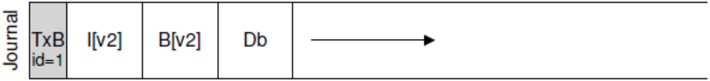
Safer writing strategy

- Issue TxB and everything up to (but not including) TxE

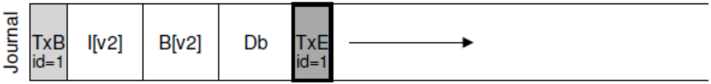


Safer writing strategy

- Issue TxB and everything up to (but not including) TxE



- Once those writes have all completed, issue TxE

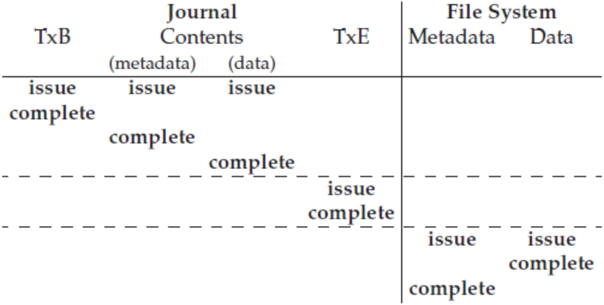


- Once the TxE is persistent, the checkpoint can be issued at some future moment

Revised journal protocol (v2)

1. Journal write:
    - write the contents of the transaction excluding TxE to the log
2. Journal commit (new):
    - write the transaction commit block (TxE)
3. Checkpoint:
    - write the contents of the update to their locations

Data Journaling Timeline



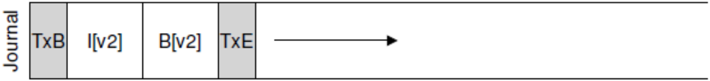
TxB	Journal Contents (metadata)	(data)	TxE	File System Metadata	Data
issue	issue	issue			
complete					
	complete				
		complete			
			issue		
			complete		
				issue	issue
					complete
				complete	

Metadata journaling (1)

- File data are written twice! Expensive

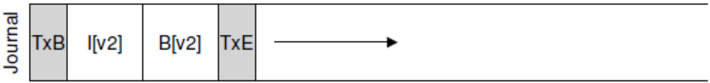
Metadata journaling (1)

- File data are written twice! Expensive
- In metadata journaling, the data block Db, previously written to the log, is not written to the journal



Metadata journaling (1)

- File data are written twice! Expensive
- In metadata journaling, the data block Db, previously written to the log, is not written to the journal



- The data block Db is written directly to its on-disk location

Metadata journaling (2)

- When should we write the data block Db?
- Suppose we write Db to disk when checkpointing the metadata

Metadata journaling (2)

- When should we write the data block Db?
- Suppose we write Db to disk when checkpointing the metadata
- Not safe (why?)

Metadata journaling (2)

- When should we write the data block Db?
- Suppose we write Db to disk when checkpointing the metadata
- Not safe (why?)
    - if a crash occurs while writing Db, the inode will point to garbage data

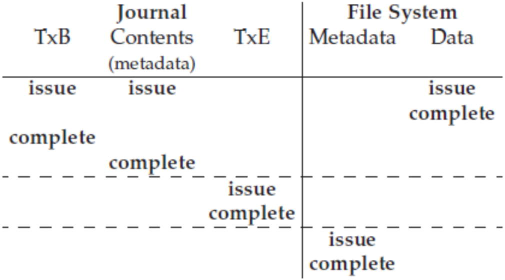
Metadata journaling (3)

- To avoid having inode pointing to garbage data
    - First, write the data block to disk and the metadata to the journal
    - Wait for the metadata and data block writes to complete before writing TxE block to journal

Revised journal protocol (v3)

1. Data write:
    - write the data block to final location
2. Journal write:
    - write the begin block (TxB) and metadata to the log
3. Journal commit (new):
    - write the transaction commit block (TxE)
4. Checkpoint:
    - write the contents of the update to their locations

Metadata journaling timeline



	Journal		File System	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	issue			issue
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Next time

- We investigate ways to improve file system performance