42. Crash Consistency: FSCK and Journaling

Operating System: Three Easy Pieces

Crash Consistency

Crash Consistency

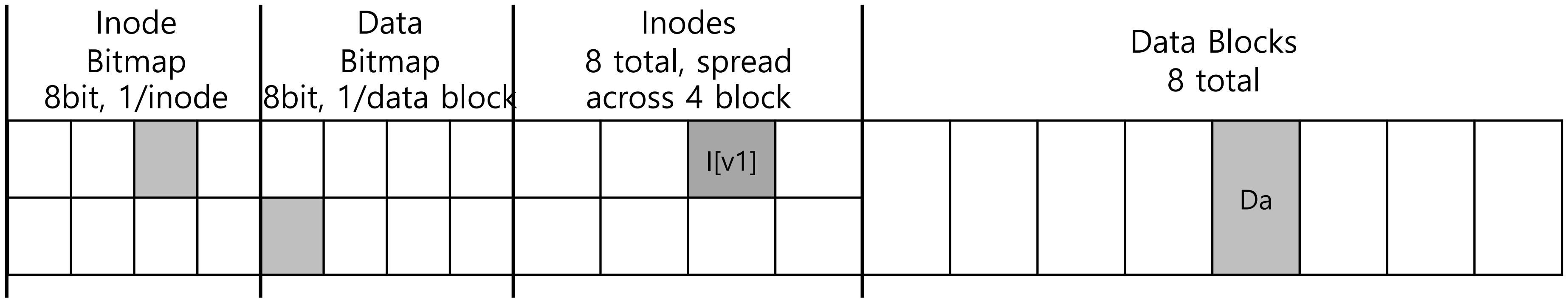
- Unlike most data structure, file system data structures must **persist**
  - They must survive over the long haul, stored on devices that retain data despite **power loss.**

- One major challenge faced by a file system is how to update persistent data structure despite the presence of a **power loss** or **system crash**.

- We'll begin by examining the approach taken by older and current file systems.
  - **fsck**(file system checker)
  - **journaling**(write-ahead logging)

A Detailed Example (1)

- Workload
  - Append of a single data block(4KB) to an existing file
  - open() → lseek() → write() → close()

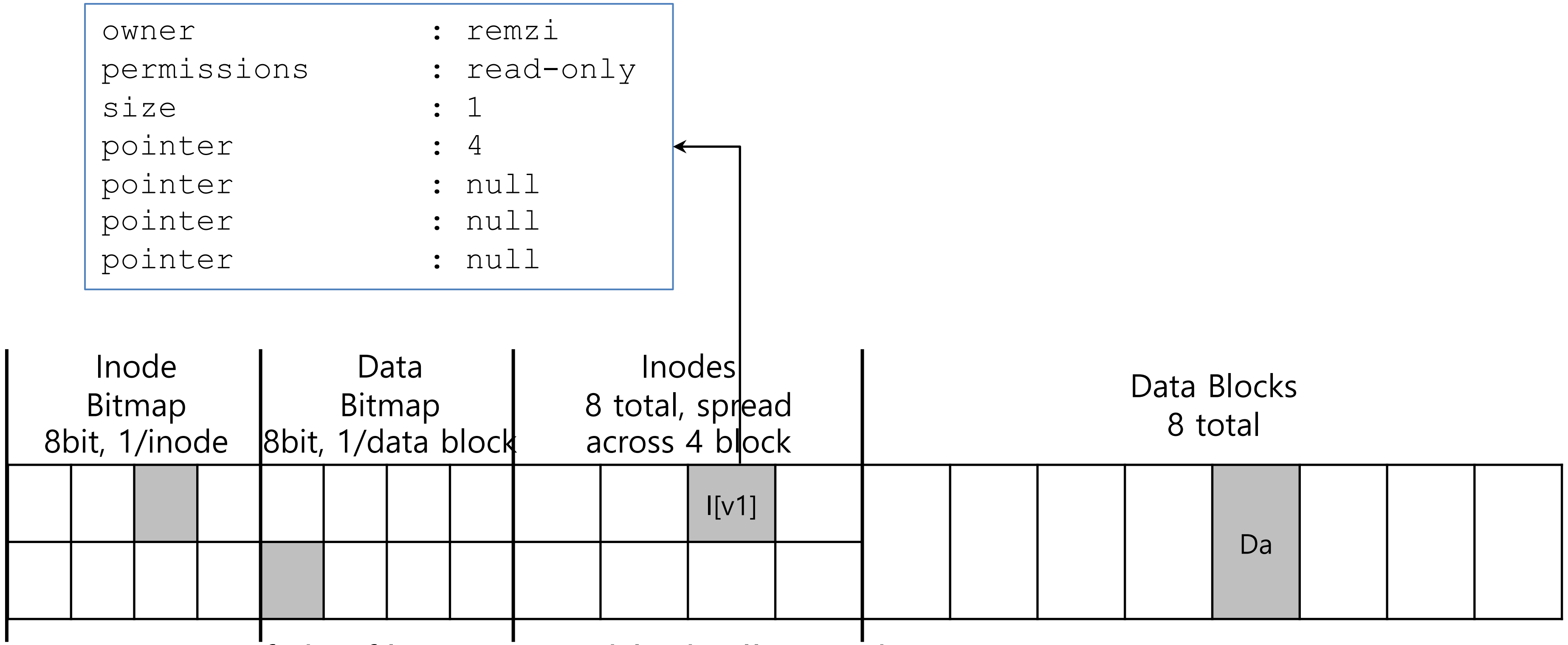

Inode Bitmap 8bit, 1/inode
Data Bitmap 8bit, 1/data block
Inodes 8 total, spread across 4 block
Data Blocks 8 total
I[v1]
Da

- Before append a single data block
  - single inode is allocated (inode number 2)
  - single allocated data block (data block 4)
  - The inode is denoted I[v1]

A Detailed Example (2)

- Inside of I[v1] (inode, before update)

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

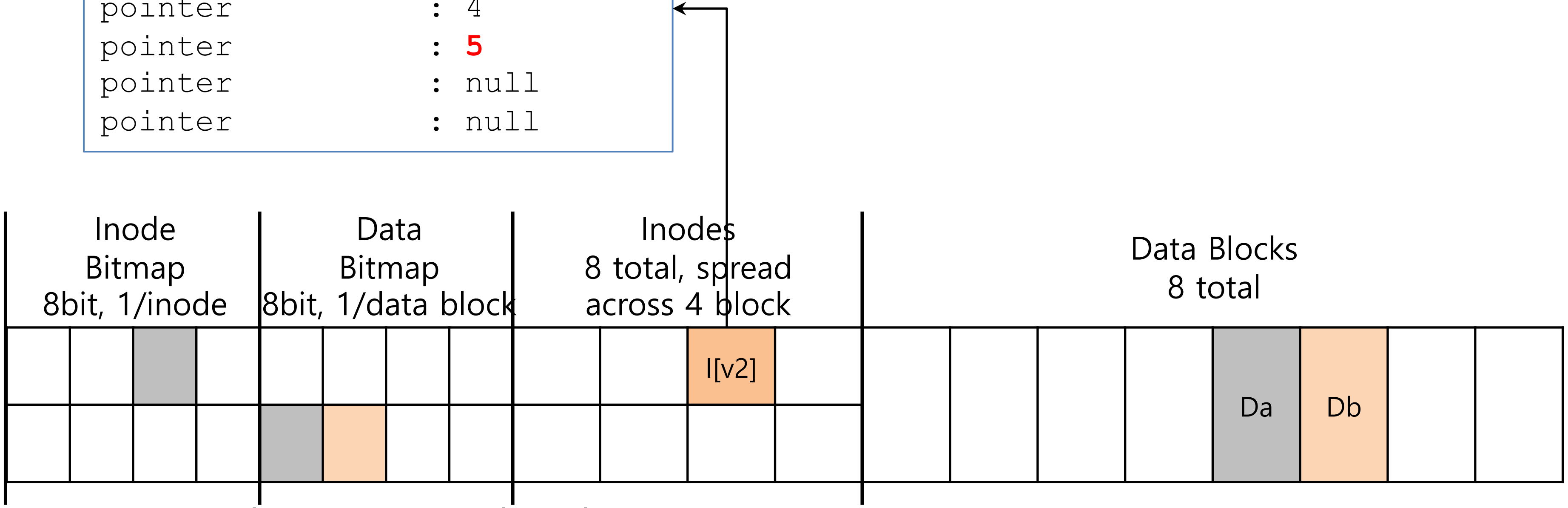

Inode Bitmap 8bit, 1/inode
Data Bitmap 8bit, 1/data block
Inodes 8 total, spread across 4 block
Data Blocks 8 total
I[v1]
Da

- Size of the file is 1 (one block allocated)
- First direct pointer points to block4 (Da)
- All 3 other direct pointers are set to `null`(unused)

A Detailed Example (3)

- After update

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



Inode Bitmap 8bit, 1/inode
Data Bitmap 8bit, 1/data block
Inodes 8 total, spread across 4 block
Data Blocks 8 total
I[v2]
Da
Db

  - Data bitmap is updated
  - Inode is updated (I[v2])
  - New data block is allocated (Db)

A Detailed Example (end)

- To achieve the transition, the system perform three separate writes to the disk.
  - One each of inode I[v2]
  - Data bitmap B[v2]
  - Data block (Db)
- These writes usually don't happen immediately
  - dirty inode, bitmap, and new data will sit in main memory
  - **page cache** or **buffer cache**
- If a crash happens after one or two of these write have taken place, but not all three, the file system could be left in a **funny state**

Crash Scenario (1)

- Imagine only a single write succeeds; there are thus three possible outcomes
  1. Just the data block(Db) is written to disk
     - The data is on disk, but there is no inode
     - Thus, it is as if the write never occurred
     - This case is not a problem at all (for the filesystem)
  2. Just the updated inode(I[v2]) is written to disk
     - The inode points to the disk address (5, Db)
     - But the Db has not yet been written there
     - We will read **garbage** data(old contents of address 5) from the disk
     - The bitmap says block isn’t allocated by inode otherwise
       - **Problem : file-system inconsistency**

Crash Scenario (2)

3. Just the updated bitmap (B[v2]) is written to disk
   - The bitmap indicates that block 5 is allocated
   - But there is no inode that points to it
   - Thus, the file system is inconsistent again
   - **Problem : space leak,** as block 5 would never be used by the file system

Crash Scenario (3)

- There are also three more crash scenarios. In these cases, two writes succeed and the last one fails
  1. The inode(I[v2]) and bitmap(B[v2]) are written to disk, but not data(Db)
     - The file system metadata is completely consistent
     - **Problem : Block 5 has garbage in it**
  2. The inode(I[v2]) and the data block(Db) are written, but not the bitmap(B[v2])
     - We have the inode pointing to the correct data on disk
     - **Problem : inconsistency between the inode and the old version of the bitmap(B1)**

Crash Scenario (end)

3. The bitmap(B[v2]) and data block(Db) are written, but not the inode(I[v2])
   - **Problem : inconsistency between the inode and the data bitmap**
   - We have no idea which file it belongs to

The Crash Consistency Problem

- What we'd like to do ideally is move the file system from one consistent state to another **atomically**

- Unfortunately, we can't do this easily
  - The disk only commits one write at a time
  - Crashes or power loss may occur between these updates

- We call this general problem the **crash-consistency problem**

Solution #1: The File System Checker

The File System Checker (1)

- The File System Checker (**`fsck`**)
  - `fsck` is a Unix tool for finding inconsistencies and repairing them.
  - `fsck` check super block, Free block, Inode state, Inode links, etc.
  - Such an approach can't fix all problems
    - example : The file system looks consistent but the inode points to garbage data.
  - The only real goal is to make sure the file system metadata is internally consistent (i.e., bitmaps and inodes agree).

The File System Checker (2)

- Basic summary of what `fsck` does:
  - **Superblock**
    - `fsck` first checks if the superblock looks reasonable
      - Sanity checks : file system size > number of blocks allocated
    - Goal : to find suspect superblock
    - In this case, the system may decide to use an **alternate** copy of the superblock
  - **Free blocks**
    - `fsck` scans the inodes, indirect blocks, double indirect blocks,
    - The only real goal is to make sure the file system metadata is internally consistent.

The File System Checker (3)

- Basic summary of what `fsck` does: (Cont.)
  - **Inode state**
    - Each inode is checked for corruption or other problem
      - Example : valid type file, directory, symbolic link, etc)
    - If there are problems with the inode fields that are not easily fixed.
      - The inode is considered suspect and cleared by `fsck`
  - **Inode Links**
    - `fsck` also verifies the link count of each allocated inode
      - To verify the link count, `fsck` scans through the entire directory tree
    - If there is a mismatch between the newly–calculated count and that found within an inode, corrective action must be taken
      - Usually by fixing the count with in the inode

The File System Checker (4)

- Basic summary of what `fsck` does: (Cont.)
  - **Inode Links** (Cont.)
    - If an allocated inode is discovered (refcnt!=0) but no directory refers to it, it is moved to the lost+found directory
  - **Duplicates**
    - fsck also checks for duplicated pointers
    - Example : Two different inodes refer to the same block
      - If on inode is obviously bad, it may be cleared
      - Alternately, the pointed-to block could be copied

The File System Checker (5)

- Basic summary of what `fsck` does: (Cont.)
  - **Bad blocks**
    - A check for bad block pointers is also performed while scanning through the list of all pointers
    - A pointer is considered "bad" if it obviously points to something outside a valid range
    - Example : It has an address that refers to a block greater than the partition size
      - In this case, `fsck` can't do anything too intelligent; it just removes the pointer

The File System Checker (6)

- Basic summary of what `fsck` does: (Cont.)
  - **Directory checks**
    - `fsck` does not understand the contents of user files
      - However, directories hold specifically formatted information created by the file system itself
      - Thus, `fsck` performs additional integrity checks on the contents of each directory
    - Example
      - making sure that "." and ".." are the first entries
      - each inode referred to in a directory entry is allocated?
      - ensuring that no directory is linked to more than once in the entire hierarchy

The File System Checker (end)

- Building a working `fsck` requires intricate knowledge of the filesystem
- `fsck` have a bigger and fundamental problem: **too slow**
  - scanning the entire disk may take many minutes or hours
  - Performance of `fsck` became prohibitive.
    - as disk grew in capacity and RAIDs grew in popularity
- At a higher level, the basic premise of `fsck` seems just a tad irrational: *search-the-entire-house-for-keys* recovery algorithm,
  - It is incredibly expensive to scan the entire disk
  - It works but is wasteful
  - Thus, as disk(and RAIDs) grew, researchers started to look for other solutions

Solution #2: Journaling

Journaling (1)

- **Journaling** (**Write-Ahead Logging**)
  - When updating the disk, before over writing the structures in place, first write down a little note describing what you are about to do
  - Writing this note is the "write ahead" part, and we write it to a structure that we organize as a "log"
  - By writing the note to disk, you are guaranteeing that if a crash takes places during the update of the structures you are updating, you can go back and look at the note you made and try again
  - Thus, you will know exactly what to fix after a crash, instead of having to scan the entire disk

Journaling (Cont.)

- We'll describe how Linux ext3 incorporates journaling into the file system.
  - Most of the on-disk structures are identical to Linux ext2
  - The new key structure is the journal itself
  - It occupies some small amount of space within the partition or on another device

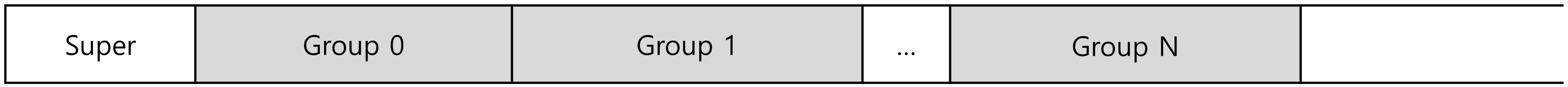


Fig.1 Ext2 File system structure

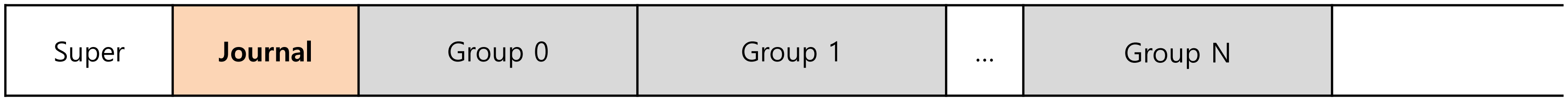
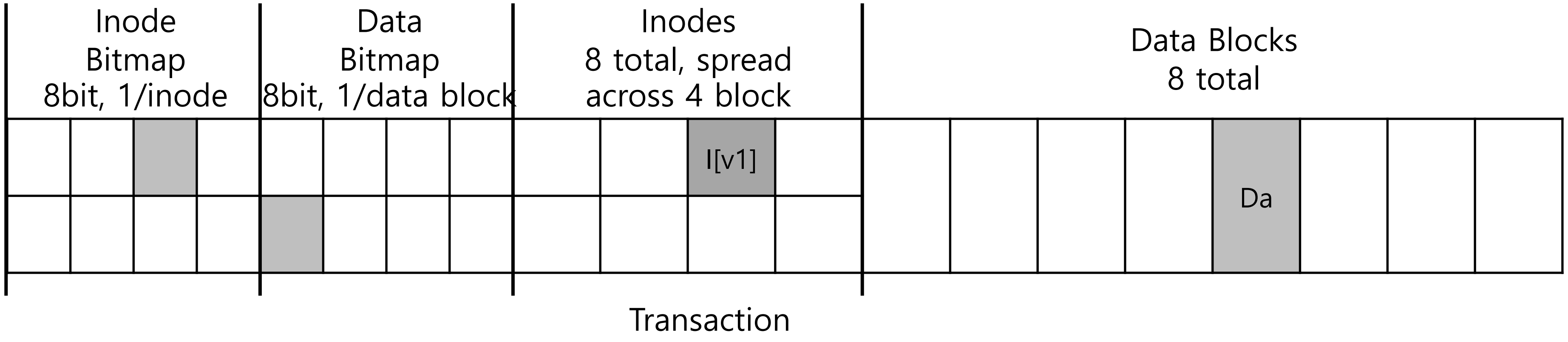


Fig.2 Ext3 File system structure

Data Journaling (1)

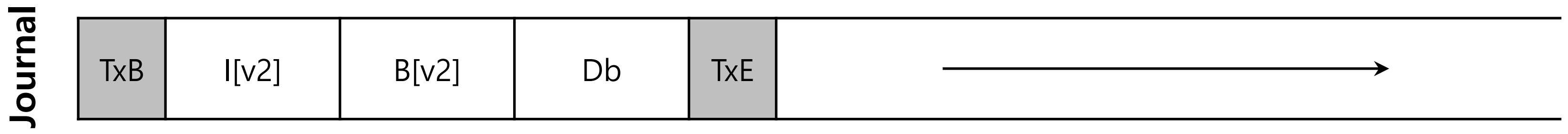
- Data journaling is available as a mode with the ext3 file system

- Example : our canonical update again
  - We wish to update inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk
  - Before writing them to their final disk locations, we are now first going to write them to the log(a.k.a. journal)


Inode Bitmap 8bit, 1/inode
Data Bitmap 8bit, 1/data block
Inodes 8 total, spread across 4 block
Data Blocks 8 total
I[v1]
Da
Transaction

Data Journaling (2)

- Example : our canonical update again (Cont.)



- **TxB**: Transaction begin block
  - It contains some kind of **transaction identifier(TID)**
- The middle three blocks just contain the exact content of the blocks themselves
  - This is known as **physical logging**
- **TxE**: Transaction end block
  - Marker of the end of this transaction
  - It also contain the TID

Data Journaling (3)

- **Checkpoint**
  - Once this transaction is safely on the disk, we are ready to overwrite the old structures in the file system
  - This process is called **checkpointing**
  - Thus, to checkpoint the file system, we issue the writes I[v2], B[v2], and Db to their disk locations

Data Journaling (4)

- Our initial sequence of operations:
  1. **Journal write**
     - Write the transaction to the log and wait for these writes to complete
     - TxB, all pending data, metadata updates, TxE
  2. **Checkpoint**
     - Write the pending metadata and data updates to their final locations

Data Journaling (5)

- When a crash occurs during the writes to the journal
  1. **Transaction each one at a time**
     - 5 transactions (TxB, I[v2], B[v2], Dnb, TxE)
     - This is slow because of waiting for each to complete
  2. **Transaction all block writes at once**
     - Five writes -> a single sequential write : Faster way
     - However, this is unsafe
       - Given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order

Aside: Forcing Writes to Disks

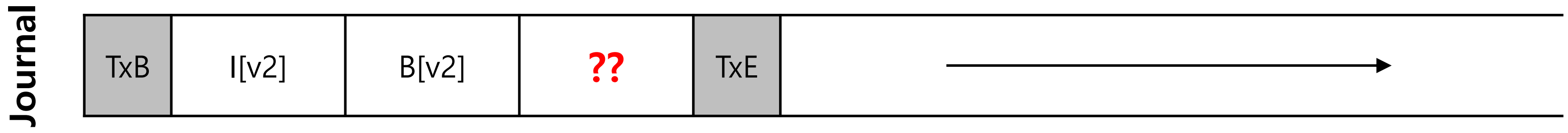
- Old times
  - Wait to interrupt back from controller
- Now
  - Write buffering inside the disk (called **immediate reporting**)
  - Solutions:
    - Disable write buffering
    - Use **Write barriers***: Analogous to fences in relaxed consistency memory models*
      - Some disk might ignore them!

Data Journaling (6)

- When a crash occurs during the writes to the journal (Cont.)

  2. **Transaction all block writes at once (Cont.)**

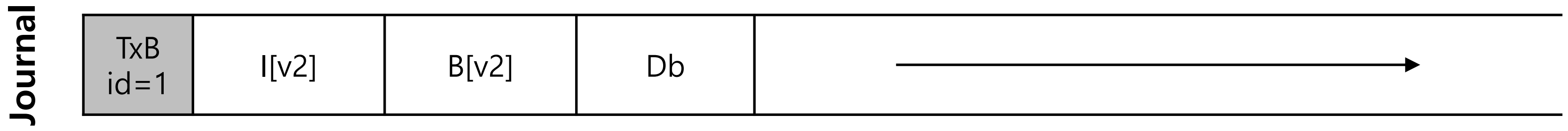
     - Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db
     - Unfortunately, if the disk loses power between (1) and (2)



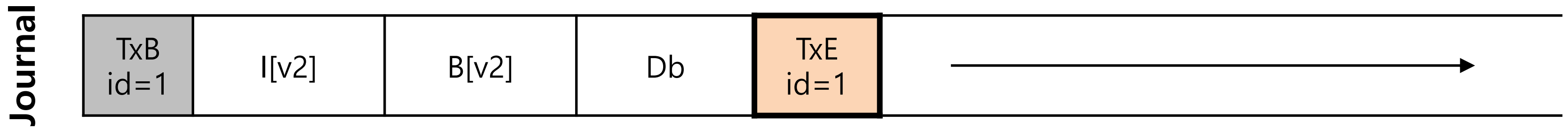
     - Transaction looks like a valid transaction.
       - Further, the file system can't look at that forth block and know it is wrong.
       - It is much worse if it happens to a critical piece of file system, such as superblock.

Data Journaling (7)

- When a crash occurs during the writes to the journal (Cont.)
  2. **Transaction all block writes at once (Cont.)**
     - To avoid this problem, the file system issues the transactional write in two steps
     - First, writes all blocks **except the TxE block** to journal



     - Second, The file system issues the write of the TxE



     - An important aspect of this process is the atomicity guarantee provided by the disk.
       - The disk guarantees that any 512-byte write either happen or not
       - Thus, TxE should be a single 512-byte block

Data Journaling (8)

- When a crash occurs during the writes to the journal (Cont.)

  2. **Transaction all block writes at once (Cont.)**

     - Thus, our current protocol to update the file system, with each of its three phases labeled:

       1. Journal write: write the contents of the transaction to the log
       2. **Journal commit (added) :** write the transaction commit block
       3. Checkpoint **:** write the contents of the update to their locations

Aside: log write optimization

- Wait for content transaction before to sent the **end transaction** sector

- Simple solution to write the whole transactions at once (used by ext4)
  - Include a **checksum** of the contents in **both parts** of the of the transaction
  - If during replay there is a **discrepancy** between the **computed** checksum and the **stored** checksum, the transaction is ignored

- Reads from journal are also protected by the checksum

Data Journaling (end)

- Recovery (at system boot)
  - If the crash happens **before the transactions** is written to the log
    - The pending update is **skipped**
  - If the crash happens **after the transactions** is written to the log, but **before the checkpoint**
    - **Recover** the update as follow:
      - Scan the log and look for transactions that have committed to the disk (and not checkpointed)
      - Transactions are replayed

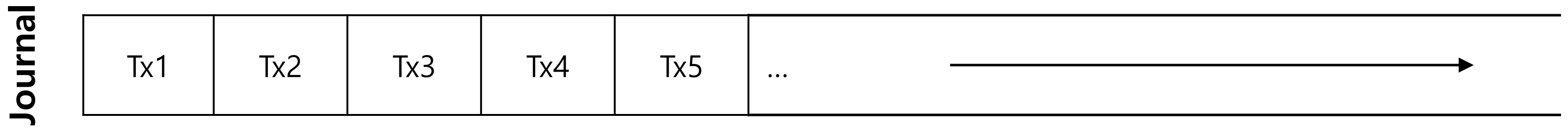
Batching Log Updates

- If we create two files in same directory, same inode, directory entry block is to the log and committed twice.

- To reduce excessive write traffic to disk, journaling manage a **global transaction** (v.gr. ext 3)
    - Write the content of the global transaction forced by synchronous request.
    - Write the content of the global transaction after a timeout (let say of 5 seconds).

Making The log Finite (1)

- The log is of a finite size
  - Two problems arise when the log becomes full



1. The larger the log, the longer recovery will take
   - Simpler but less critical
2. No further transactions can be committed to the disk
   - Thus, making the file system “less than useful”

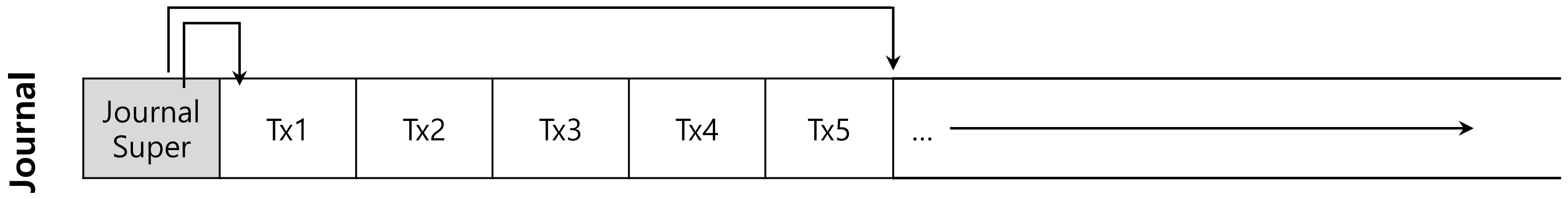
Making The log Finite (2)

- To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over
  - Therefore, the journal is referred to as a **circular log**.

- To do so, the file system must take action some time after a checkpoint
  - Specifically, once a transaction has been checkpointed, the file system should free the space

Making The log Finite (3)

- journal super block
  - Mark the oldest and newest transactions in the log.
  - The journaling system records which transactions have not been check pointed.



Making The log Finite (end)

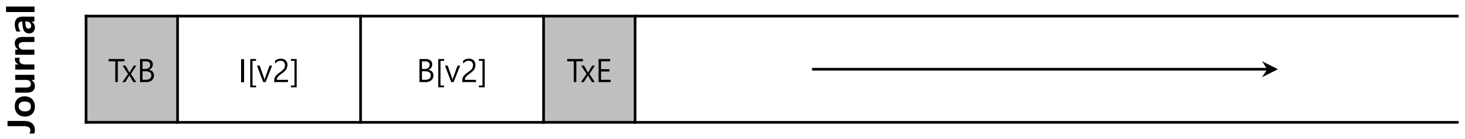
- journal super block (Cont.)
  - Thus, we add another step to our basic protocol
    1. Journal write
    2. Journal commit
    3. checkpoint
    4. **Free**
       - Some time later, mark the transaction free in the journal by updating the journal Superblock

Metadata Journaling (1)

- There is a still problem : writing every data block to disk **twice**
  - Commit to log (journal file)
  - Checkpoint to on-disk location.
- People have tried a few different things in order to speed up performance.
  - Example : A simpler form of journaling is called **ordered journaling (metadata journaling)**
    - **User data is not written to the journal**

Metadata Journaling (2)

- Thus, the following information would be written to the journal



- The data block Db, previously written to the log, would instead be written to the file system proper

Metadata Journaling (3)

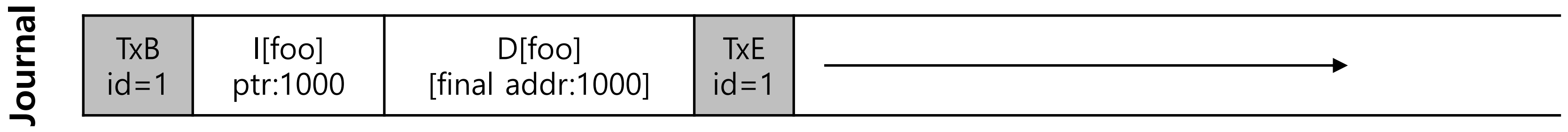
- The modification does raise an interesting question: **when should we write data blocks to disk?**
- Let's consider an example
  1. Write Data to disk after the transaction
     - Unfortunately, this approach has a problem
     - The file system is consistent but I[v2] may end up pointing to garbage data
     - (still ext3 allows it, it's called unordered journaling)
  2. Write Data to disk before the transaction
     - It avoids the problems

Metadata Journaling (end)

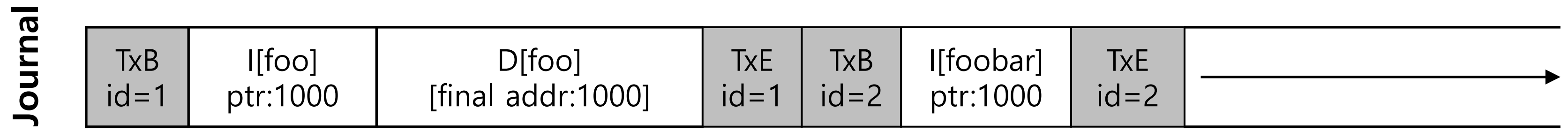
- Specifically, the protocol is as follows:
  1. **Data Write(added)**: Write data to final location
  2. **Journal metadata write(added)**: Write the begin and metadata to the log
  3. Journal commit
  4. Checkpoint metadata
  5. Free

Tricky case: Block Reuse (1)

- Some metadatas should not be replayed.
- Example
  1. Directory "foo" is updated (D[foo] is considered metadata)

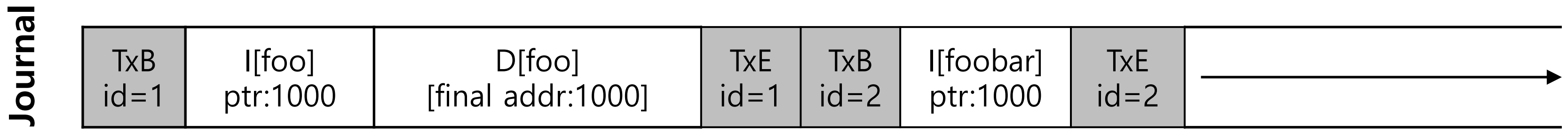


  2. Directory "foo" id deleted. block 1000 is freed up for reuse
  3. User Create a file "foobar", reusing block 1000 for data



Tricky case: Block Reuse (2)

4. Now assume a crash occurs and all of this information is still in the log.



5. During replay, the recovery process replays everything in the log
   - Including the **write of directory data in block 1000**
6. The replay thus overwrites the user data of current file `foobar` with old directory contents

Tricky case: Block Reuse (2)

- Solution
  - What Linux ext3 does instead is to add a **new type** of record to the journal, Known as a **revoke** record
  - When replaying the journal, the system first scans for such revoke records
  - Any such revoked data is never replayed

Data Journaling Timeline

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

Data Journaling Timeline

Metadata Journaling Timeline

TxB	**Journal contents**	TxE	**File System**	
	(metadata)		Metadata	Data
issue	issue			Issue
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Metadata Journaling Timeline

- This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)