Crash Consistency: FSCK and Journaling

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Crash Consistency: Introduction

→ 전원이 꺼지더라도 일관성은 유지해야 함.

- **File system data structures must persist, stored on devices that retain data despite power loss**
  - One major challenge faced by a file system is how to update persistent data structures despite the presence of a power loss or system crash

read(), write(), ...

- **Crash consistency is that changes in file system are guaranteed from a valid state to another valid state despite crash**
  - If file system is crashed among multiple writes, the on-disk structure may be left in an inconsistent state

- **We will see two approaches to handle this issue**
  - The approach taken by older file systems, known as fsck or "file system checker"

    → 실제 write하기 전에 log로 저장함
  - The other is journaling (known as write-ahead logging), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes

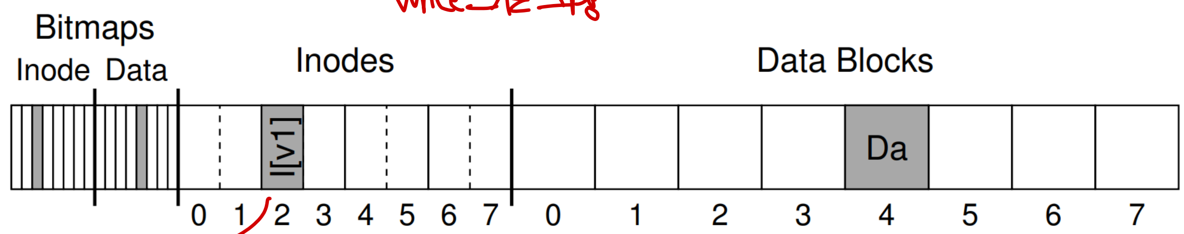
A Detailed Example

- **Consider a simple file system with 8 inodes and 8 data blocks**
  - A single inode is allocated (2nd), and a single allocated data block (4th)
  - The corresponding bitmaps are marked and the inode/data are denoted I[v1]/Da

write 되는 과정

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

→direct pointer



  - When we append to the file, the inode will be updated to have a pointer to a new block, the corresponding data bitmaps are marked and the inode becomes I[v2]

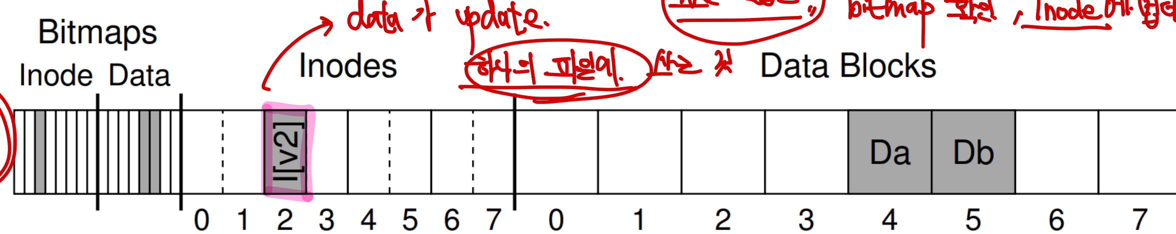
data가 update.

쓰는 과정은, bitmap 확인, Inode에 업데이트.

하나의 파일에 쓰는 것

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

journal



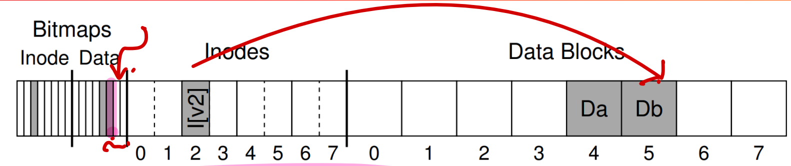
  - This requires multiple separate write to the disk for bitmap, inode, and data and these write will be delayed by page cache or data buffer → write가 바로되지는 않고 cache되어 한방에 함.

memory

- **What if a crash happens after one or two of the writes take place?**

→ 이러면 어떻게 복구할 것인가.

Crash Scenario



- **Three possible outcomes for only a single write success**

Outcome	Meaning
→ 그냥 쓰레기 데이터로 취급해주면 그만! **Only Db is written**	Db is on disk but bitmap/inode say no data on block 5 No problem on crash consistency perspective
→ bitmap이 업데이트가 안됨 **Only I[v2] is written**	I[v2] has a pointer to block 5, but bitmap says it's free (file system inconsistency) and it will read "garbage" data
→ Db를 working하는 inode가 없음 = 공간 낭비. **Only B[v2] is written**	B[v2] indicates the block 5 is allocated but no inode (file system inconsistency) and it will result in space leak

- **Three possible outcomes for two write success and one failure**

Outcome	Meaning
I[v2], B[v2] are written, but not Db → 쓰레기 값이 들어가 있음	The file system metadata is completely consistent The block 5 has garbage in it
I[v2], Db are written, but not B[v2] → 나중에 다른 write가 오면 overwrite	I[v2] has a pointer to block 5 that includes Db but bitmap says it's free (file system inconsistency)
B[v2], Db are written, but not I[v2] → 가리키고 있는게 없음	B[v2] says the block 5 is allocated and the block 5 has Db but no inode (file system inconsistency)

- **Crash-consistency problem (consistent-update problem)**
  - The write (inode/bitmap/data block) to the disk "is not atomic"

Solution #1: The File System Checker

lock T1 T2

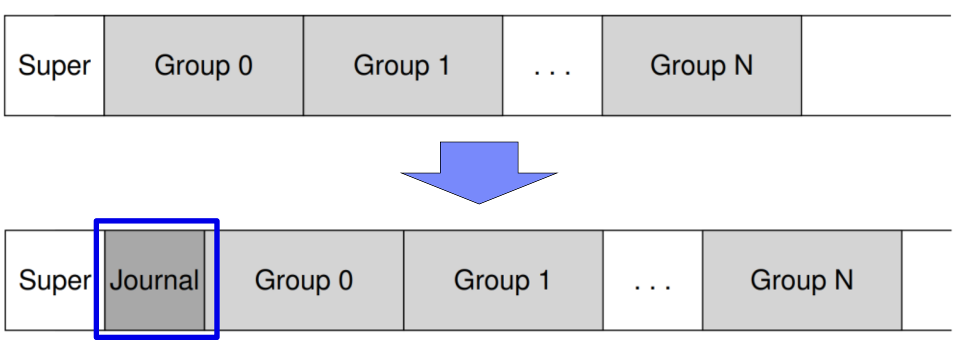
- **Early file systems took a simple approach to crash consistency**
  - Basically, they decided to let inconsistencies happen and then fix them later
  - `fsck` is a UNIX tool for finding such inconsistencies and repairing them
- **The basic summary of what `fsck` does is:**
  - Superblock: examine if the superblocks looks reasonable by sanity check
  - Free blocks: make correct bitmaps by scanning inodes, indirect links, etc
  - Inode state: check the inodes for corruption or other problems (e.g. type)
  - Inode links: verify the link count of each allocated inode by directory scan
  - Duplicates: check for duplicate pointers
  - Bad blocks: check for bad block pointers (e.g. pointer is out of valid range/size)
  - Directory checks: check the integrity on the contents of each directory
- **`fsck` has a bigger and perhaps more fundamental problem**
  - `fsck` and similar approaches are too slow 10TB, ... 하나하나. block들 full scan.
  - With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many hours or days

Solution #2: Journaling (or Write-Ahead Logging)

- **Write-ahead logging (called journaling) was invented to address the consistent update problem**
  - Many modern file systems (e.g. Linux ext3/4, Windows NTFS) uses journaling
- **The basic idea of the journaling is as follows:**
  - When updating the disk, first write down a little note (in a well-known location on disk) describing what you are about to do; write-ahead + log
  - If a crash takes places during the update, you can go back and look at the note you made and try again

① log + write.
중간에 문제.
↓
② log를 보고 똑같이 write.

- **We will see how Linux ext3's journaling works**
  - The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device

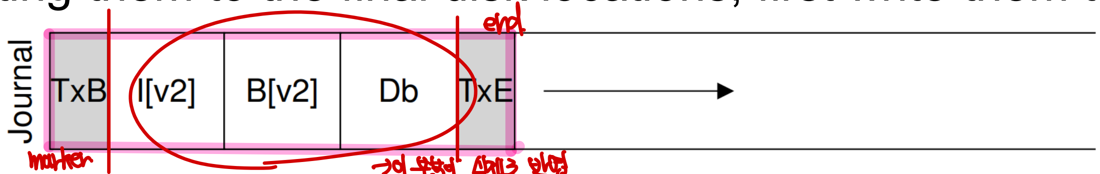


Linux ext2 file system structure (without journaling)

Linux ext3 file system structure (with journaling)

Data Journaling: Basic

- **Consider writes of inode (I[v2]), bitmap (B[v2]), data block (Db)**
  - Before writing them to the final disk locations, first write them to the log (journal)

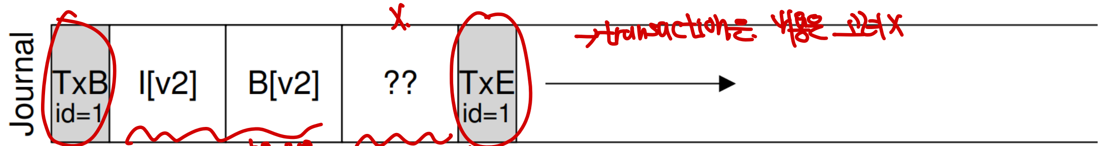


  - TxB (Transaction Begin) includes information about the update (e.g. the final addresses of I[v2], B[v2], and Db) and transaction identifiers (TID)
  - I[v2], B[v2], Db contain the exact contents of blocks; this is known as physical logging as physical contents are updated in journal (logical logging is possible)
    →실제 데이터는 x
  - TxE (Transaction End) is a marker of the end of this transaction with the TID
- **Once the transaction is safely on disk, it is ready for actual write**
  - The checkpointing is the process of writing the physical log to their final disk locations; if we have successfully checkpointed, then we are basically done
- **The initial sequence of operations:**
  - ① Journal write: write the transaction and wait for these writes to complete
  - ② Checkpoint: write the pending metadata and date updates to the final locations

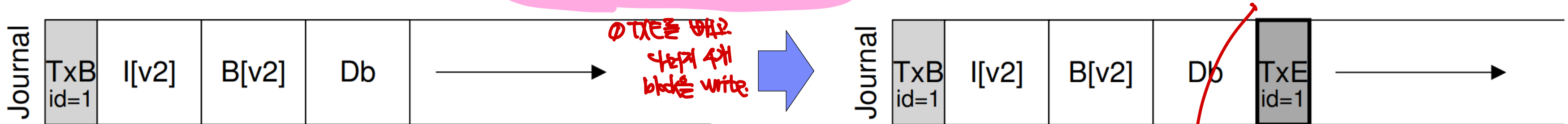
Data Journaling: Crash

→Journaling 자체가 atomic하지 않다.

- **It is problematic when a crash occurs during "transactional write"**
  - To write a set of blocks in the transaction to disk, all five block writes are issued at once due to disk speed, however, this is unsafe
  - What if a crash occurs between writes of 1) TxB, I[v2], B[v2], TxE and 2) Db?



  - The transaction looks valid and it will write "??" to the final location for recovery
- **To avoid this problem, "two step" transactional write is used**
  - Write all blocks except for the TxE block to the journal, issuing all writes at once
  - Issue the write of the TxE block atomically (512-byte for HDD) 한번에 모아서 할 수도 있음



- **Three phase protocol to update the file system**
  - Journal write: write the transaction (no ~~TxB~~ TxE) and wait for the writes to complete
  - Journal commit: write the rest transaction (TxE) and wait ➔ committed
  - Checkpoint: write the metadata and data updates to their final locations

Metadata Journaling

DB는 C(LOG) Journaling 과정에서 다 써야함
→ 비효율적

- **Journaling requires double write traffic and slow down system**
  - The ordered journaling (or metadata journaling) is the nearly the same with data journaling, except that user data is not written to the journal

Journal	TxB	I[v2]	B[v2]	TxE	⟶			

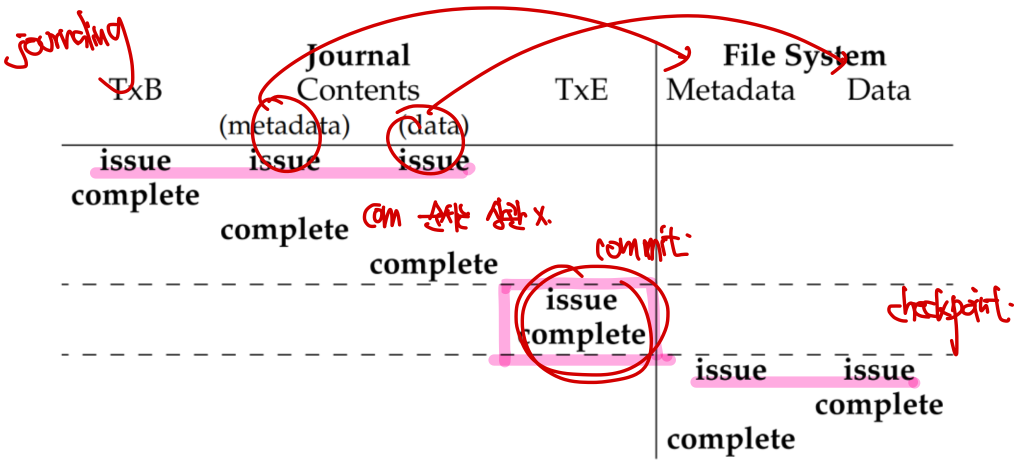
- **Then, when should we write data blocks to disk?**
  - The ordering of the data write matter for metadata journaling
  - What if we write Db to disk after the transaction (I[v2], B[v2]) completes? I[v2] may end up pointing to garbage data (e.g. crash before Db write is done)
  - To avoid this, file systems write data blocks to the disk first.

(meta 데이터가 먼저 location에 써지면 문제 발생)

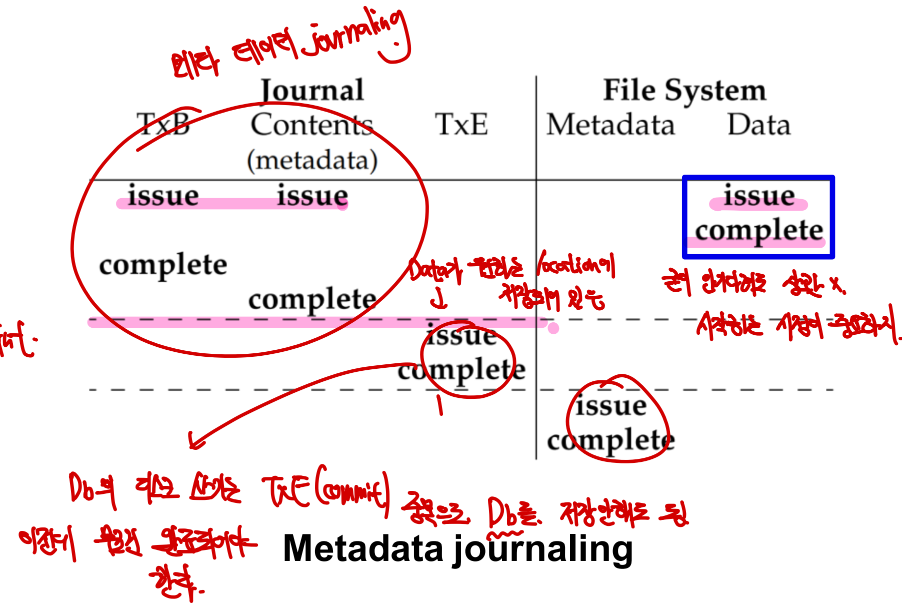
- **The protocol of the metadata journaling is as follows:**
  - ① Data write: write data to final location and wait for completion (wait is optional)
  - ② Journal metadata write: write the transaction (no TxB) and wait for completion
  - ③ Journal commit: write the rest transaction (TxE) and wait ➔ committed
  - ④ Checkpoint: write the metadata updates to the final locations
  - ⑤ Free: later, mark the transaction free in journal superblock

Journaling Timelines & Recovery

- **Journaling protocol timelines**
  - In metadata journaling, data block writes can be performed concurrently with transactional writes (except for TxE); these writes must be done before commit



Data journaling



Metadata journaling

- **A crash may happen at any time during the sequence of updates**
  - If a crash happens before the checkpoint is complete, it can be recovered
  - Since the transactions are "committed", they are replayed (try to write again) 다시 checkpoint replay
  - This form of logging is one of the simplest forms and is called redo logging