File System Journaling

Journaling

- **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

- 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

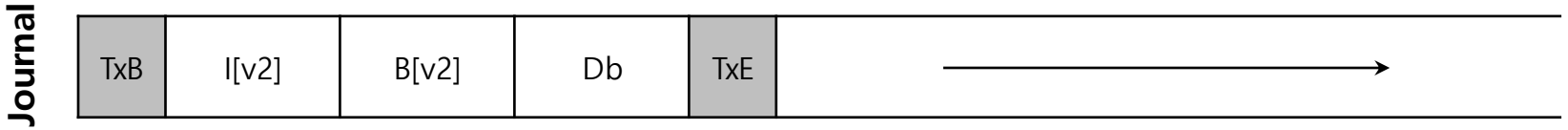
Data Journaling

- 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)

Data Journaling

- 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

- **Checkpoint**
  - Once this transaction is safely on 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

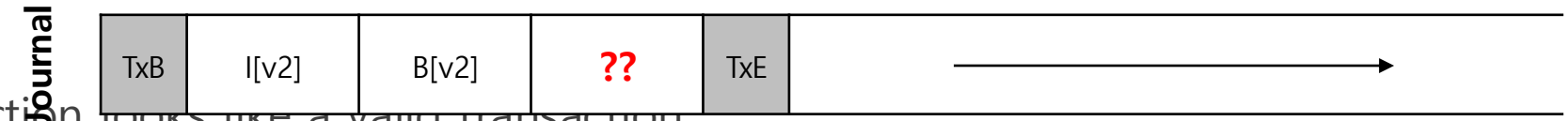
- Our initial sequence of operations:
  1. **Journal write**
     - Write the transaction to the log and wait for these writes to complete
     - TxB, all pending data, metadata updates, TxE
  2. **Checkpoint**
     - Write the pending metadata and data updates to their final locations

Data Journaling

- 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

Data Journaling

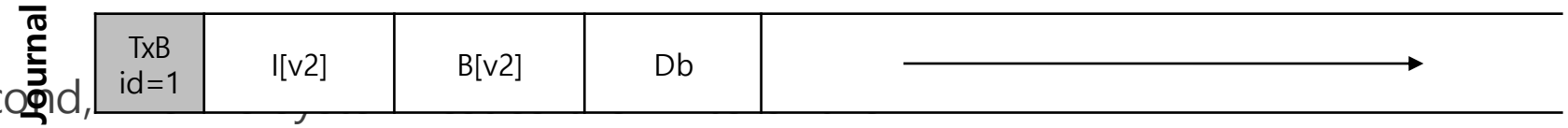
- When a crash occurs during the writes to the journal
  - **2. Transaction all block writes at once (Cont.)**
    - Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db
    - Unfortunately, if the disk loses power between (1) and (2)



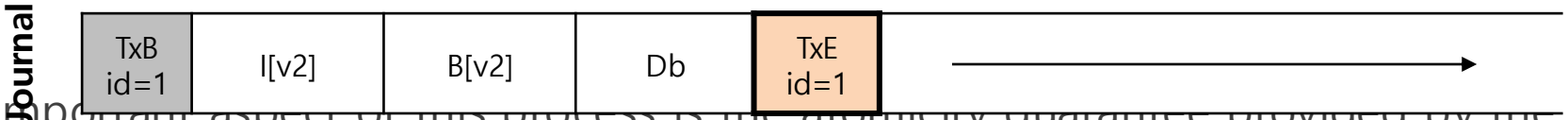
    - Transaction looks like a valid transaction.
      - » Further, the file system can't look at that forth block and know it is wrong
      - » It is much worse if it happens to a critical piece of file system, such as superblock

Data Journaling

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



    - Second, [illegible]



    - 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

- When a crash occurs during the writes to the journal
  - Transaction all block writes at once (Cont.)
    - Thus, our current protocol to update the file system, with each of its three phases labeled:
      - » Journal write : write the contents of the transaction to the log
      - » Journal commit (added) : write the transaction commit block
      - » Checkpoint : write the contents of the update to their locations

Data Journaling

- Recovery
  - 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 lock for transactions that have committed to the disk
      - » Transactions are replayed

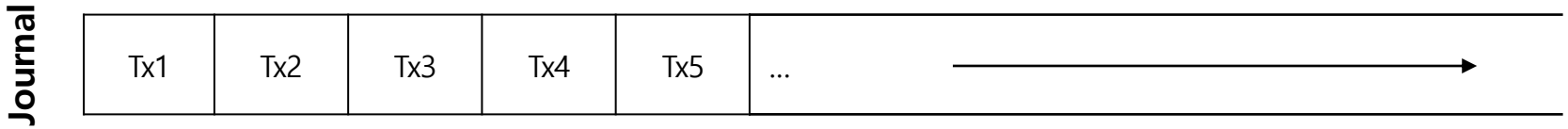
Batching Log Updates

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

- To reduce excessive write traffic to disk, journaling manage the **global transaction**
  - Write the content of the global transaction forced by synchronous request
  - Write the content of the global transaction after timeout of 5 seconds

Making The log Finite

- 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"

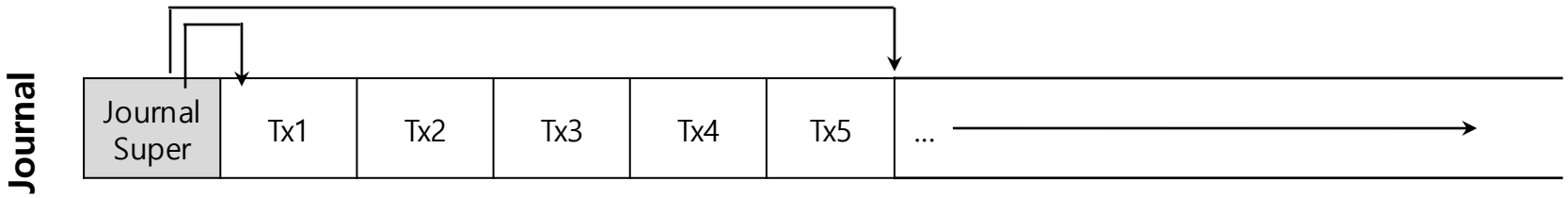
Making The log Finite

- To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over
  - This is why 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

- 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

- 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

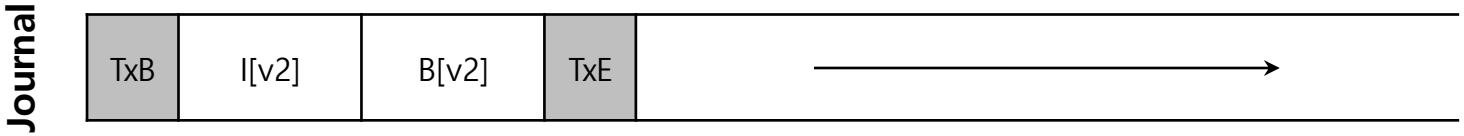
Metadata Journaling

- 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

- 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

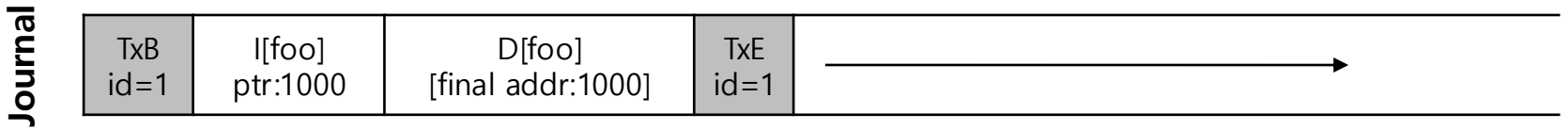
- The modification does raise an interesting question: **when should we write data blocks to disk?**
- Let's consider an example
  1. Write Data to disk after the transaction
     - Unfortunately, this approach has a problem
     - The file system is consistent but I[v2] may end up pointing to garbage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

Metadata Journaling

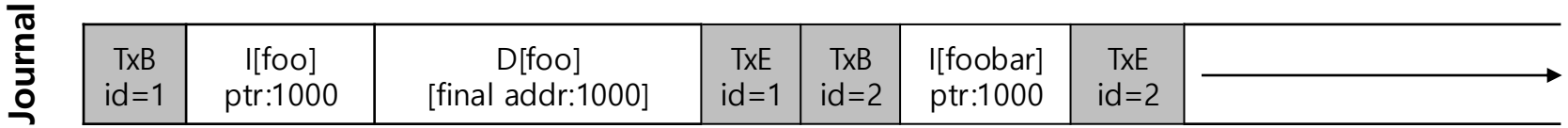
- Specifically, the protocol is as follows:
  1. **Data Write(added)**: Write data to final location
  2. **Journal metadata write(added)**: Write the begin and metadata to the log
  3. Journal commit
  4. Checkpoint metadata
  5. Free

Tricky case: Block Reuse

- Some metadatas should not be replayed
- Example
  1. Directory “foo” is updated.

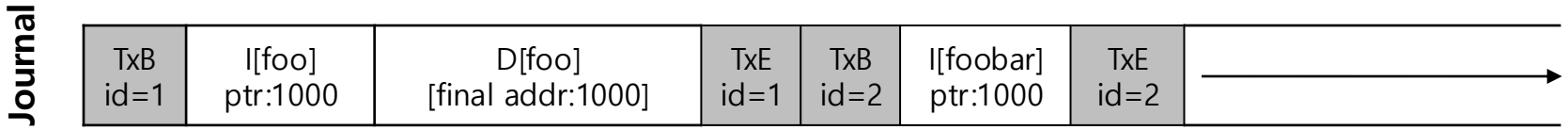


  2. Directory “foo” is deleted. block 1000 is freed up for reuse
  3. User Create a file “foobar”, reusing block 1000 for data



Tricky case: Block Reuse

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

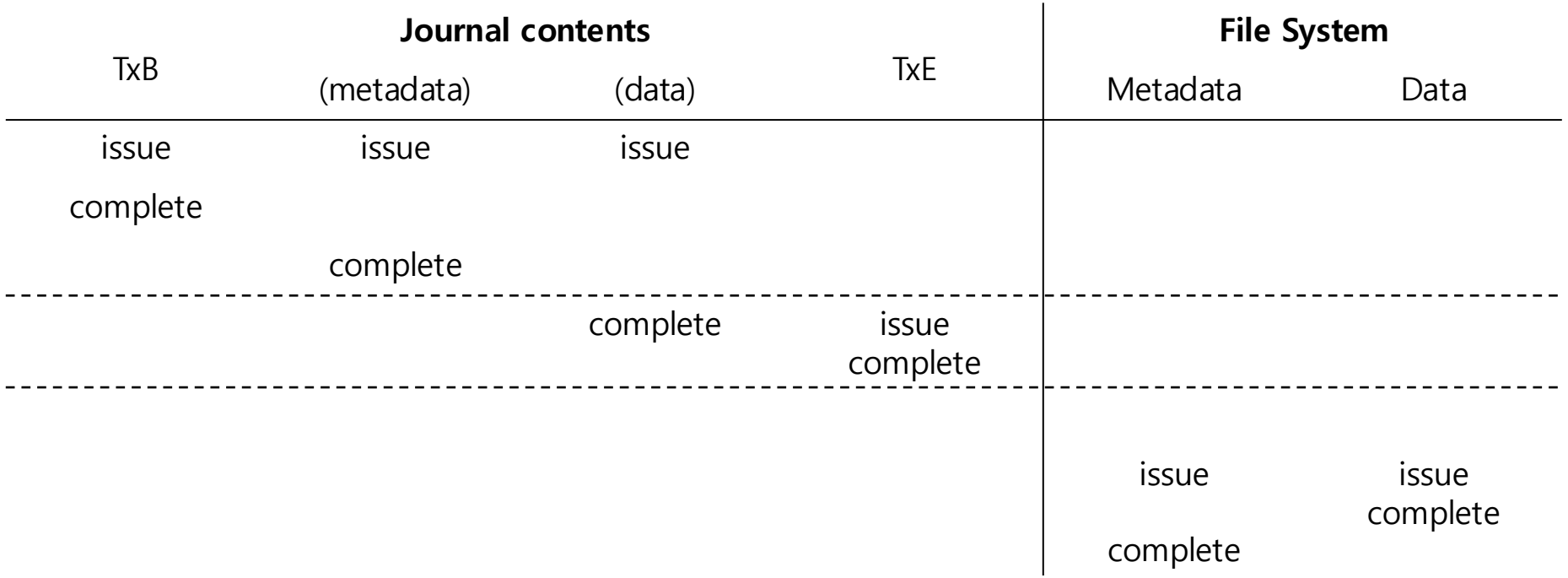


5. During replay, the recovery process replays everything in the log
   - Including the write of directory data in block 1000
6. The replay thus overwrites the user data of current file `foobar` with old directory contents

Tricky case: Block Reuse

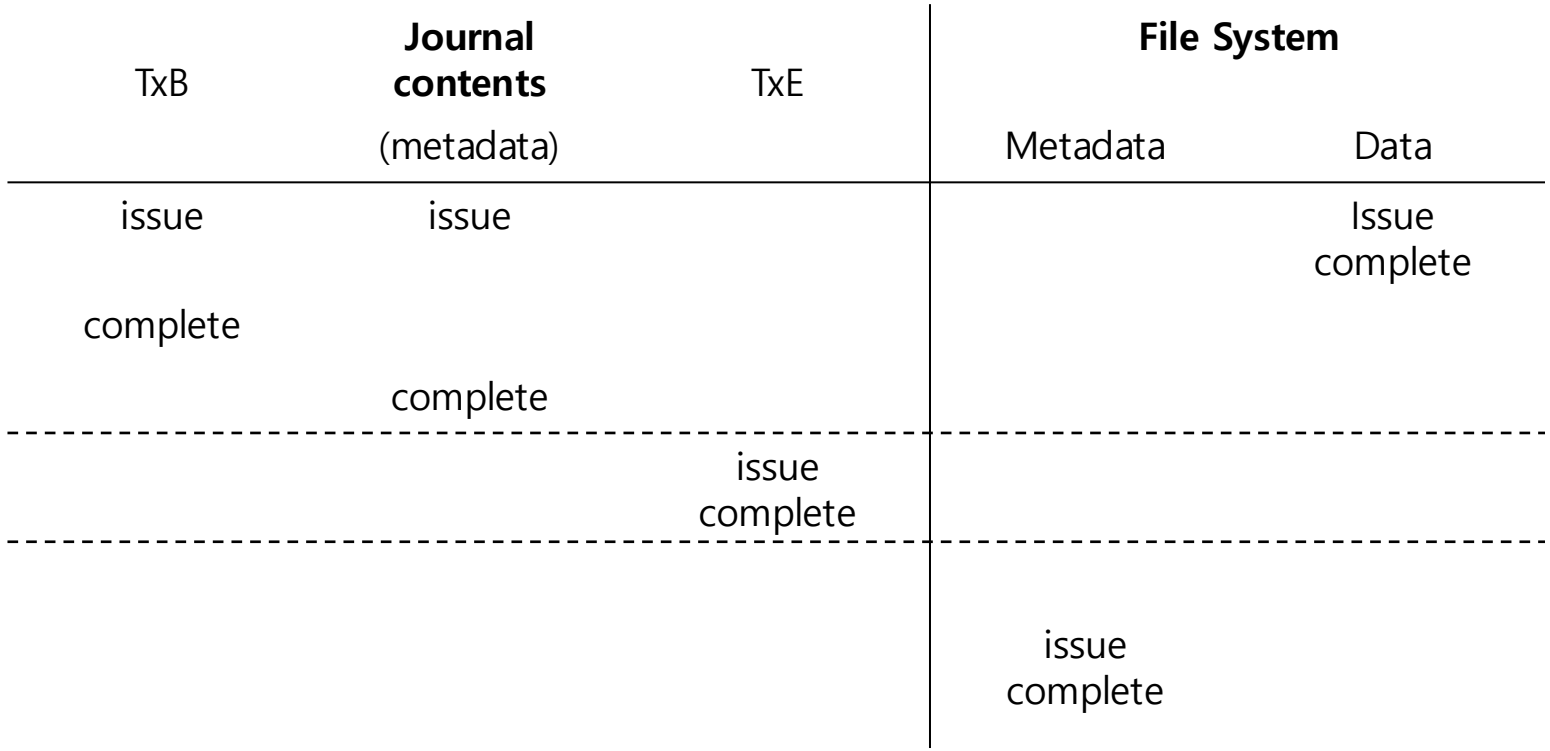
- Solution
  - What Linux ext3 does instead is to add a **new type** of record to the journal, Known as a **revoke** record
  - When replaying the journal, the system first scans for such revoke records
  - Any such revoked data is never replayed

Data Journaling Timeline



Data Journaling Timeline

Metadata Journaling Timeline



Metadata Journaling Timeline