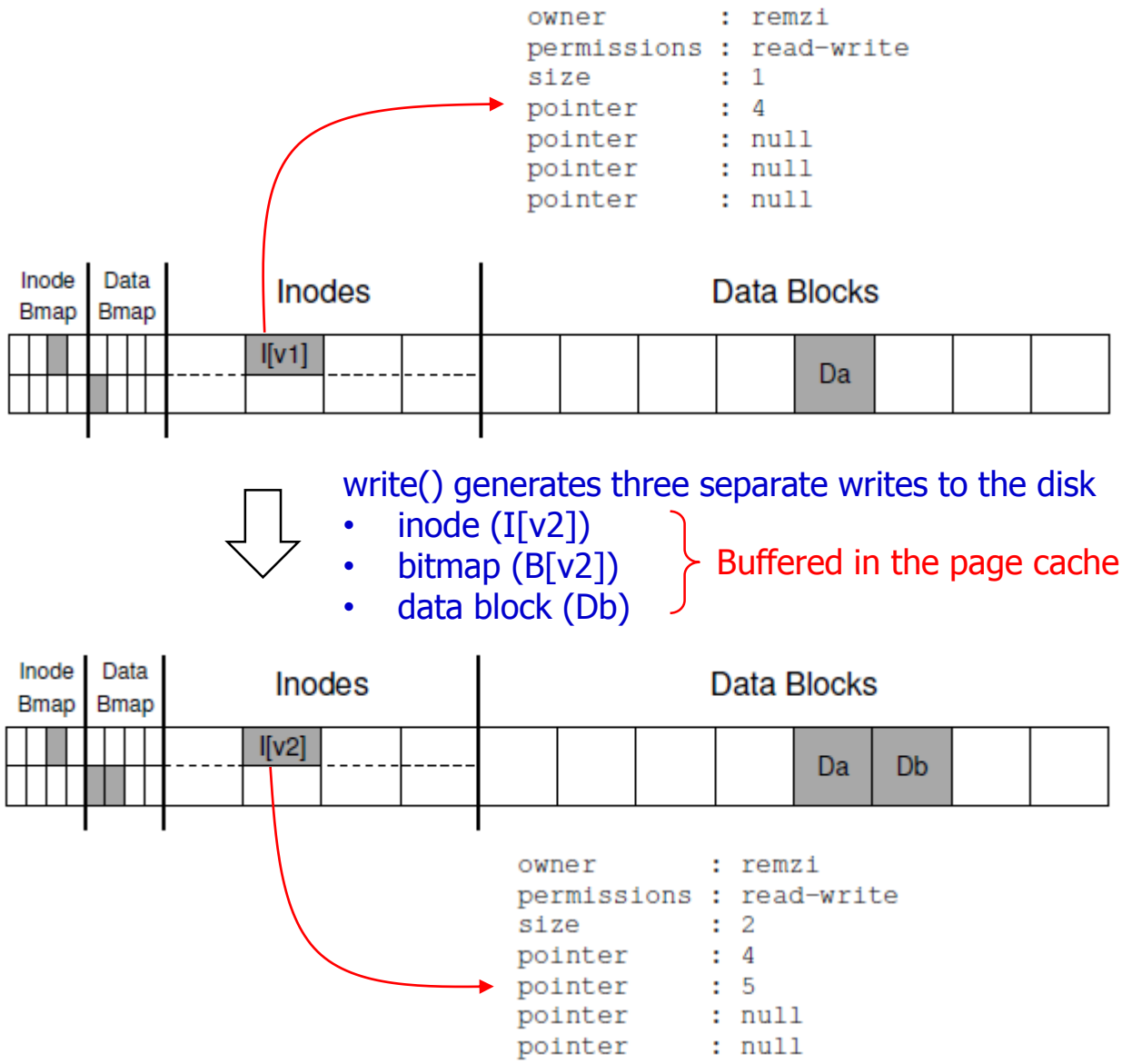
Crash Consistency: FSCK and Journaling

Crash-consistency problem

- File system data structures must persist
  - stored on HDD/SSD despite power loss or system crash
- Crash-consistency problem
  - The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated ➔ inconsistent state.
  - Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a consistent state?
- Two approaches
  - FSCK (file system checker)
  - Journaling (write-ahead logging)

Updates on-disk structures



If a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in an inconsistent state

Crash Scenarios - Only a single write succeeds

- **Just the data block (Db) is written to disk**
  - Data is on disk, but no inode that points to it and no bitmap
  - It is as if the write never occurred.
  - Not a problem at all, from the perspective of file-system crash consistency.
- **Just the updated inode (I[v2]) is written to disk**
  - Inode points to the disk address (5), but Db has not yet been written there.
  - We will read garbage data from the disk (the old contents)
  - Bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. ➔ inconsistent
  - Should be resolved
- **Just the updated bitmap (B[v2]) is written to disk**
  - Bitmap indicates that block 5 is allocated, but there is no inode that points to it.
  - Inconsistent
  - If left unresolved, a space leak (block 5 would never be used)

Crash Scenarios - Two writes succeed

- **I[v2] and B[v2] are written to disk**
  - The file system metadata is completely consistent
  - Everything looks OK from the perspective of the file system's metadata.
  - One problem: 5 has garbage in it.
- **I[v2] and Db are written**
  - Inconsistency between the inode and the old version of the bitmap (B1).
  - Need to resolve the problem before using the file system.
- **B[v2] and Db are written**
  - Inconsistency between the inode and the data bitmap.
  - Even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

Crash Consistency Problem

- We must move the file system from one consistent state to another **atomically**
- Unfortunately, the disk only commits one write at a time, and crashes or power loss may occur between these updates.
  - inconsistency in file system data structures
  - space leaks
  - garbage data

- **crash-consistency problem** (**consistent-update problem**).

Solution #1: File System Checker

- Let inconsistencies happen and then fix them later (when rebooting)
- **fsck** is a UNIX tool for finding inconsistencies and repairing them
- Can't fix all problems, only consider metadata consistency
  - Some inodes point to garbage data
- Run before the file system is mounted
- Multiple phases
  - **Superblock**
    - superblock sanity checks such as making sure the file system size is greater than the number of blocks allocated.
    - For a corrupt superblock; need to use an alternate copy
  - **Free blocks**
    - scans the inodes, indirect blocks, double indirect blocks, etc., to know currently allocated blocks within the file system.
    - produce a correct version of the allocation bitmaps, trusting the information within the inodes.
    - produce a correct version of the inode bitmaps.

Solution #1: File System Checker

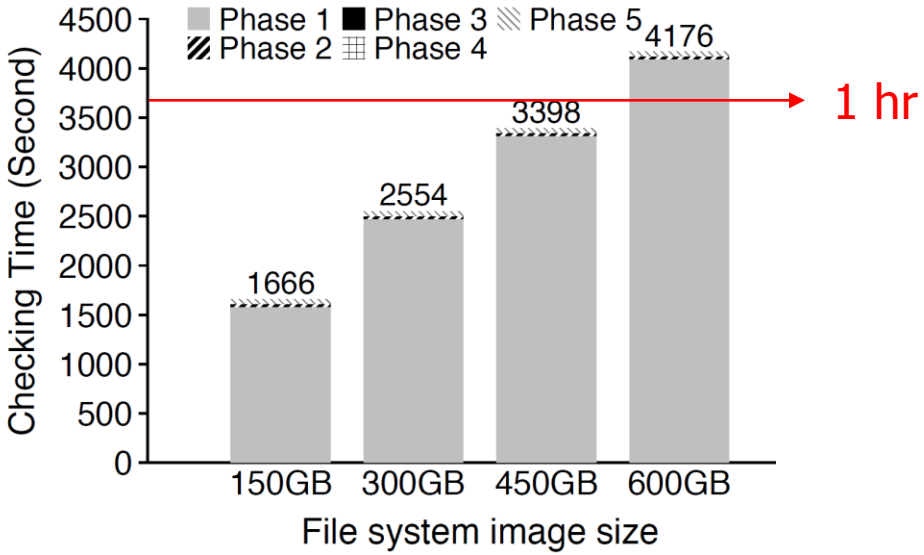
- **Inode state**
  - Each inode is checked for corruption or other problems.
  - E.g., fsck makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.).
  - If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by fsck; the inode bitmap is correspondingly updated.
- **Inode links**
  - verifies the link count of each allocated inode.
  - To verify the link count, fsck scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system.
  - If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory.

Solution #1: File System Checker

- **Duplicates**
  - checks for the cases where two different inodes refer to the same block.
  - If one inode is obviously bad, it may be cleared.
  - Alternately, give each inode its own copy.
- **Bad blocks**
  - Bad block pointers if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size.
  - It just removes the pointer from the inode or indirect block.
- **Directory checks**
  - making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

Solution #1: File System Checker

- **Problems**
  - Requires intricate knowledge of the file system
  - Too slow: take many minutes or hours
  - Wasteful
    - scan the entire disk to fix problems that occurred during an update of just three blocks



[Ma et al. FAST'13]

Solution #2: Journaling (Write-Ahead Logging)

- An idea from the world of database management systems
- Many modern file systems use the journaling idea
  - Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, Windows NTFS.
- Basic idea
  - Before overwriting the file system structures in place, first write down a little log (somewhere else on the disk, journal area) describing what you are about to do.
  - If a crash takes places during the update (overwrite) of the structures, you can redo (replay logs)
  - If a crash takes places during the write of the logs, you can undo (ignore logs)
  - Not scanning the entire disk.

ext3 file system

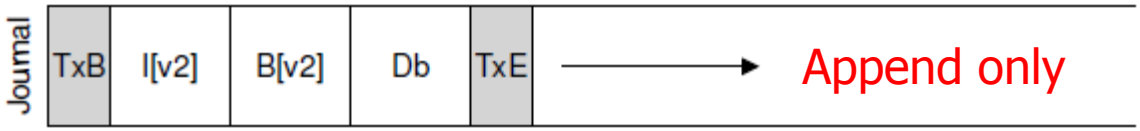
Data Journaling

1. Journal write

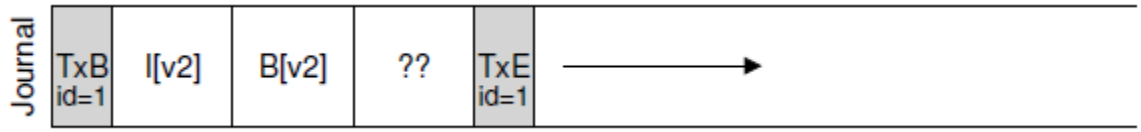
- Write the transaction to the log
  - transaction-begin block (TxB): transaction identifier (TID), information about the pending update (e.g., the final addresses of I[v2], B[v2], Db)
  - all pending data and metadata updates: the exact contents of the blocks themselves (physical logging, cf. logical logging)
  - transaction-end block (TxE): marker of the end of this transaction, and also contain the TID
- Wait for these writes to complete.

2. Checkpoint

- Write the pending metadata/data updates to their final locations.
- writes I[v2], B[v2], and Db to their disk locations

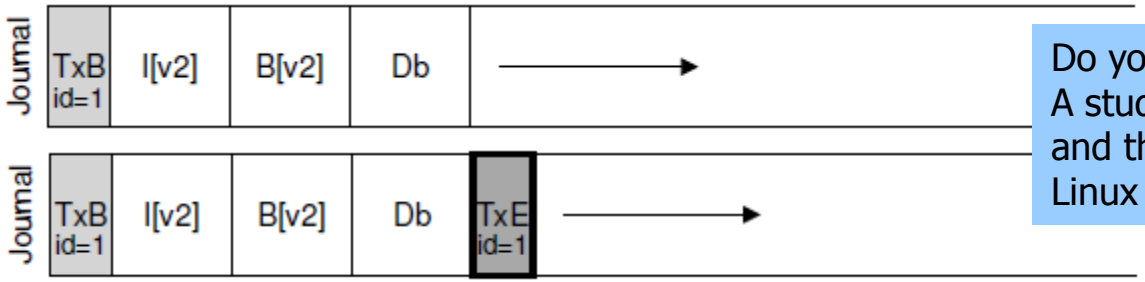


- **Crash** during the writes to the journal.
  - Disk internally may perform scheduling and complete small pieces of the big write in any order



Data Journaling

- Two steps of transactional write
  - writes all blocks except the TxE block to the journal
  - When those writes complete, the file system issues the write of the TxE block



Do you have a more intelligent idea? A student of Remzi proposed one, and the idea is now implemented in Linux ext4.

- Three phases

Due to write buffer in disk, we need write barriers

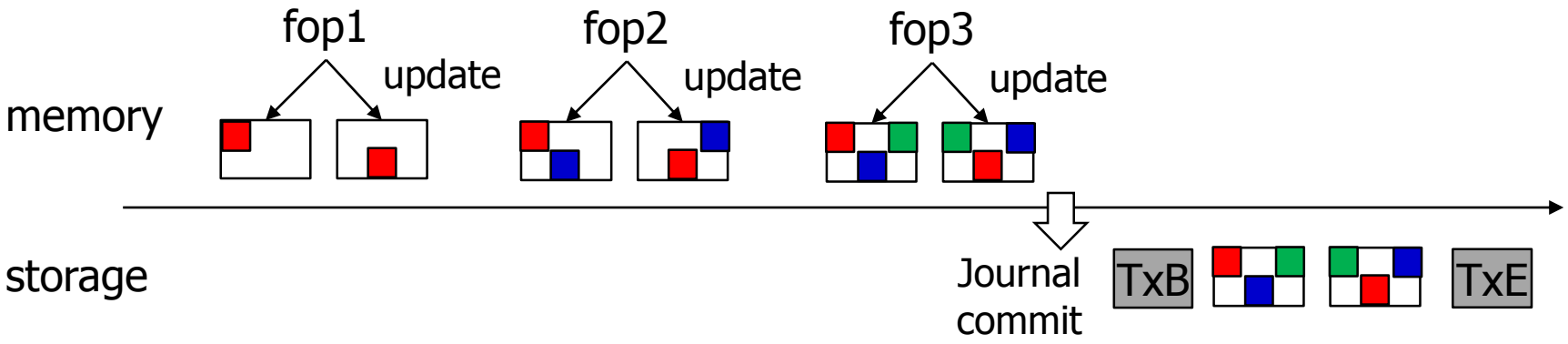
  - 1. Journal write: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete (write barrier, flush)
  - 2. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for write to complete (flush or atomic 512B); transaction is said to be committed.
  - 3. Checkpoint: Write the contents of the update (metadata and data) to their final on-disk locations.

Recovery

- If the crash happens before the transaction is written safely to the log (i.e., before the journal commit completes)
  - The pending update is simply skipped.
- If the crash happens after the transaction has committed to the log, but before the checkpoint is complete
  - The file system can recover the update.
  - Redo logging: when the system boots, the recorded transactions are replayed (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations.
- Crash may happen at any point during checkpointing
  - some of the updates to the final locations of the blocks have completed.
  - These updates are simply performed again during recovery.

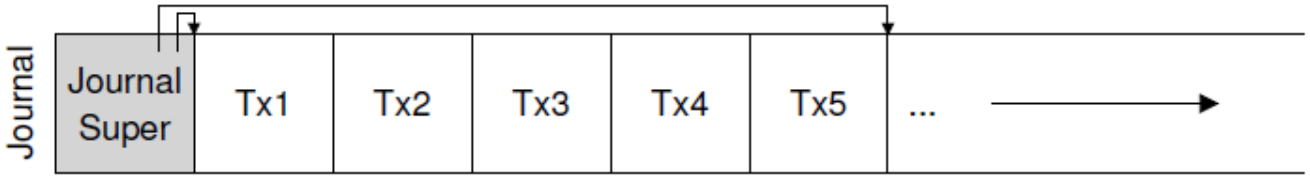
Batching Log Updates

- Rather than commit each update to disk one at a time; all updates can be buffered into a global transaction (e.g., Linux ext3)
  - Compound transaction
- File system just marks the in-memory data structures as dirty, and adds them to the list of blocks that form the current transaction.
- When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates.
- Can avoid excessive write traffic to disk



Making The Log Finite

- The write-ahead log is of a finite size. If we keep adding transactions to it, it will soon fill.
  - The larger the log, the longer recovery will take
  - When the log is full, no further transactions can be committed to the disk.
- Journaling file systems treat the log as a circular data structure, re-using it over and over ➔ circular log
- Once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused.
- Mark the oldest and newest non-checkpointed transactions in the log in a **journal superblock**



Metadata Journaling

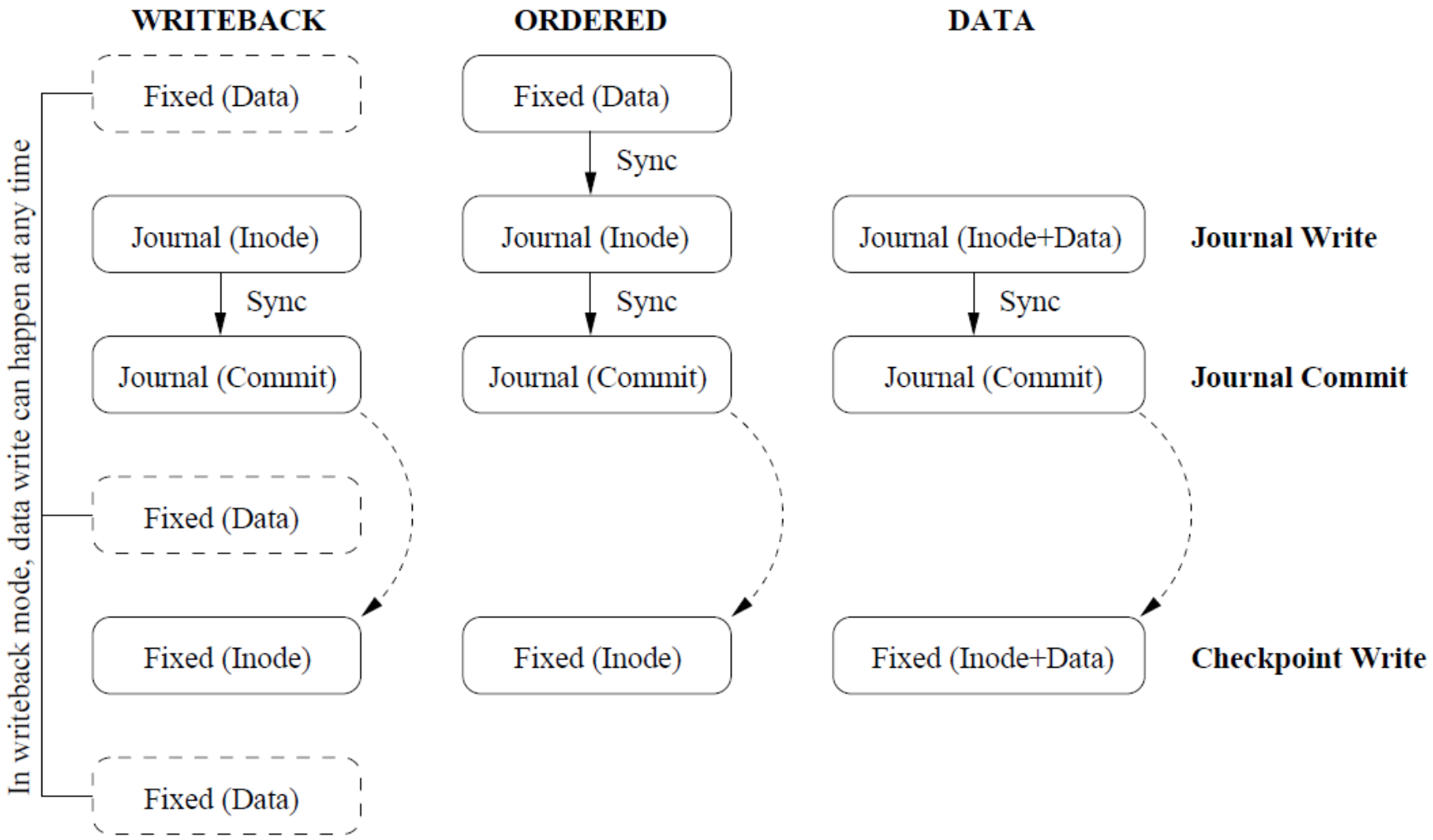
- In data journaling, for each write to disk, we are also writing to the journal first, thus doubling write traffic
  - especially painful during sequential write workloads
  - there is a costly seek between writes to the journal and writes to the main file system
- Ordered journaling (metadata journaling)
  - user data is not written to the journal
  - The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write
  - reduces the I/O load of journaling
  - When should we write data blocks to disk?
  - If we write Db to disk after the transaction (containing I[v2] and B[v2]) completes, the file system is consistent but I[v2] may end up pointing to garbage data.

Metadata Journaling

- Write the pointed-to object before the object that points to it
  - write data blocks (of regular files) to the disk first, before related metadata is written to journal
  - can guarantee that a pointer will never point to garbage

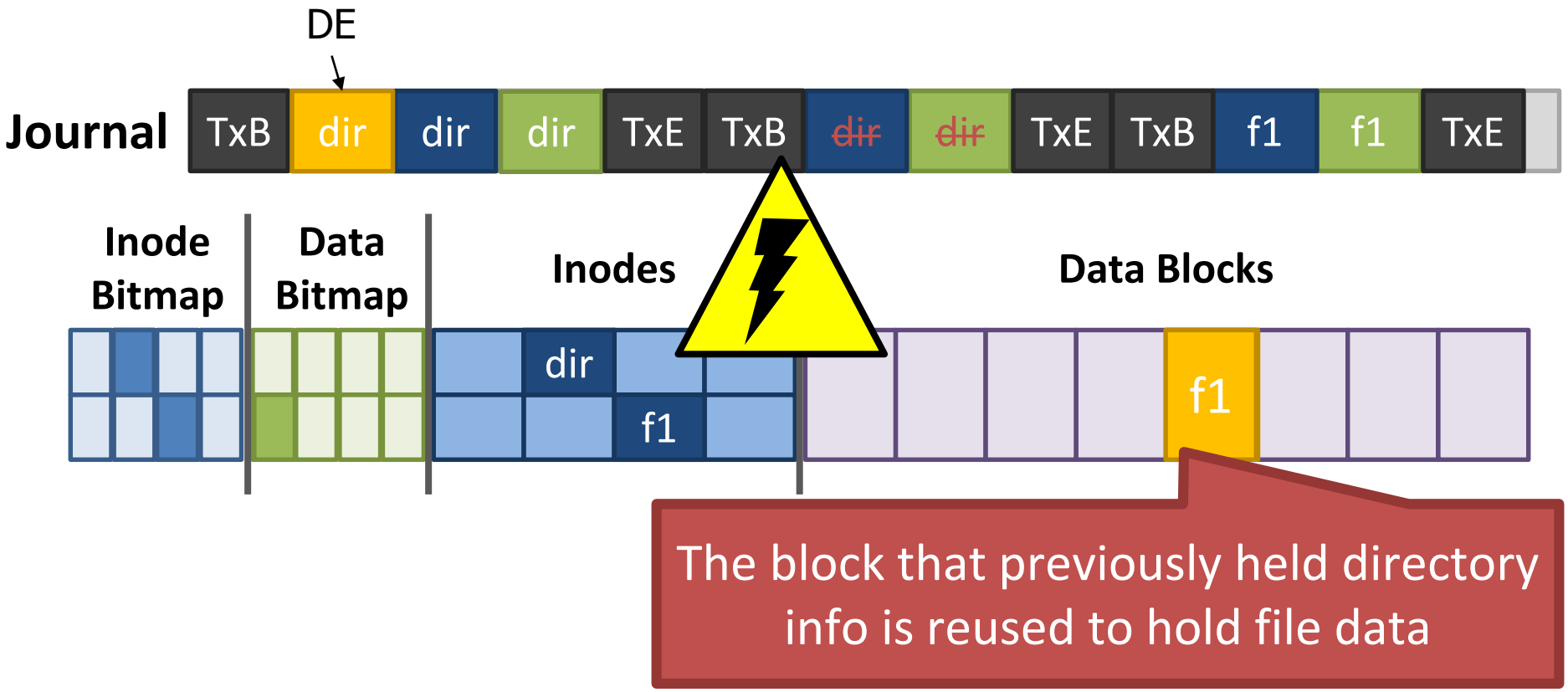
- **Ordered journaling** (Linux ext3/4, Windows NTFS, SGI' XFS)
  1. **Data write**: Write data to final location; wait for completion
  (the wait is optional; Step 1 must complete before Step 3).
  2. **Journal metadata write**: Write the begin block and metadata to the log; wait for writes to complete.
  3. **Journal commit**: Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now committed.
  4. **Checkpoint metadata**: Write the contents of the metadata update to their final locations within the file system.
  5. **Free**: Later, mark the transaction free in journal superblock.
- Ext3 supports also the unordered journaling (writeback journaling)

Ext3 Journaling modes



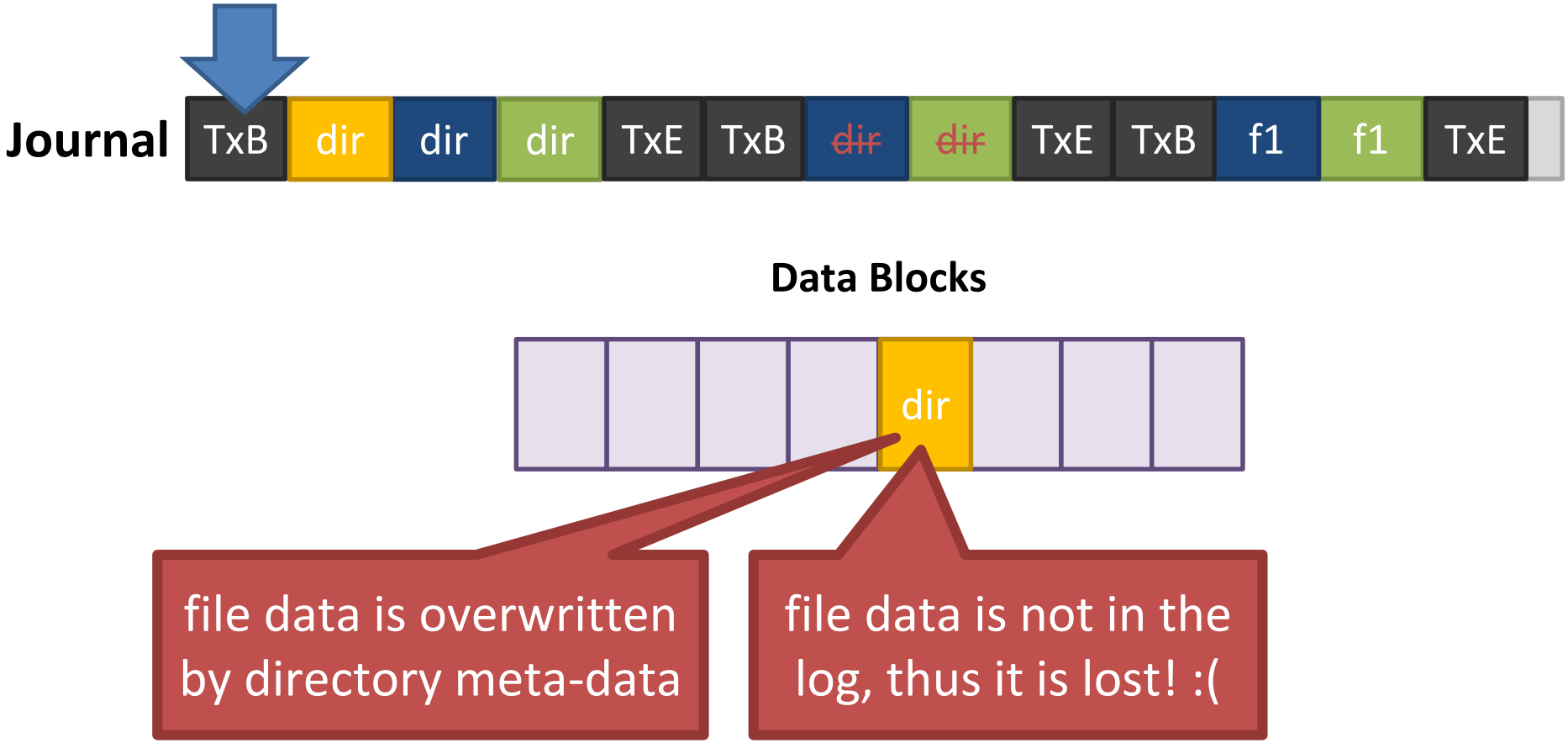
Delete and Block Reuse in metadata journaling

1. **Create a directory: inode and data are written**
2. **Delete the directory: inode is removed**
3. **Create a file: inode and data are written**



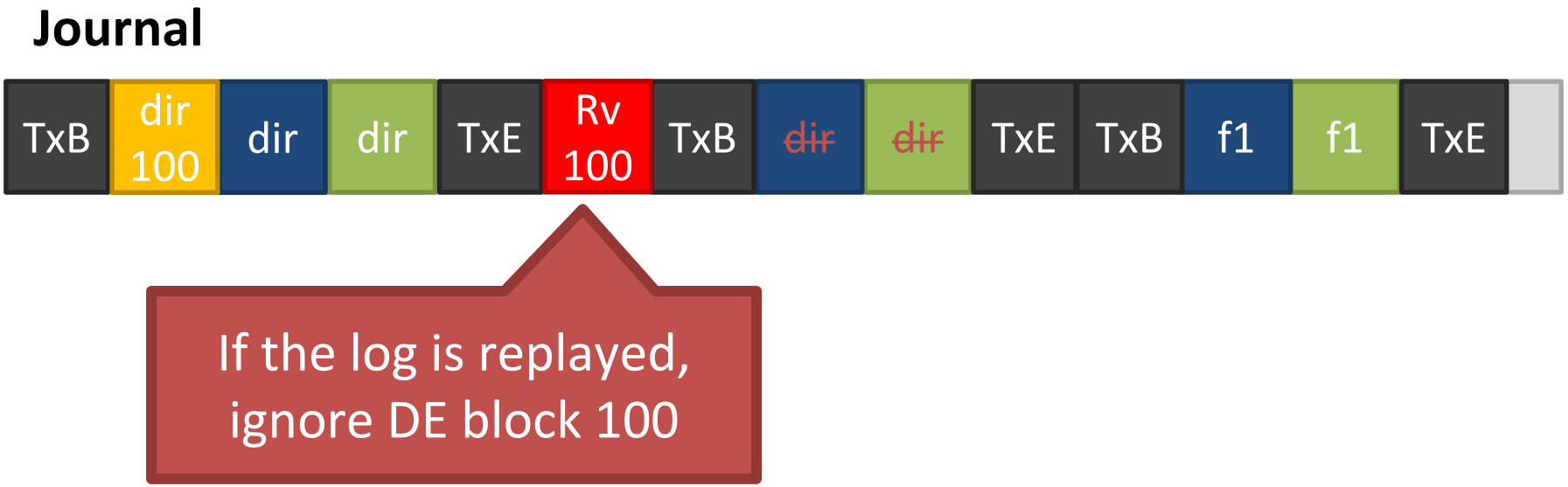
The Trouble With Delete

- **What happens when the log is replayed?**

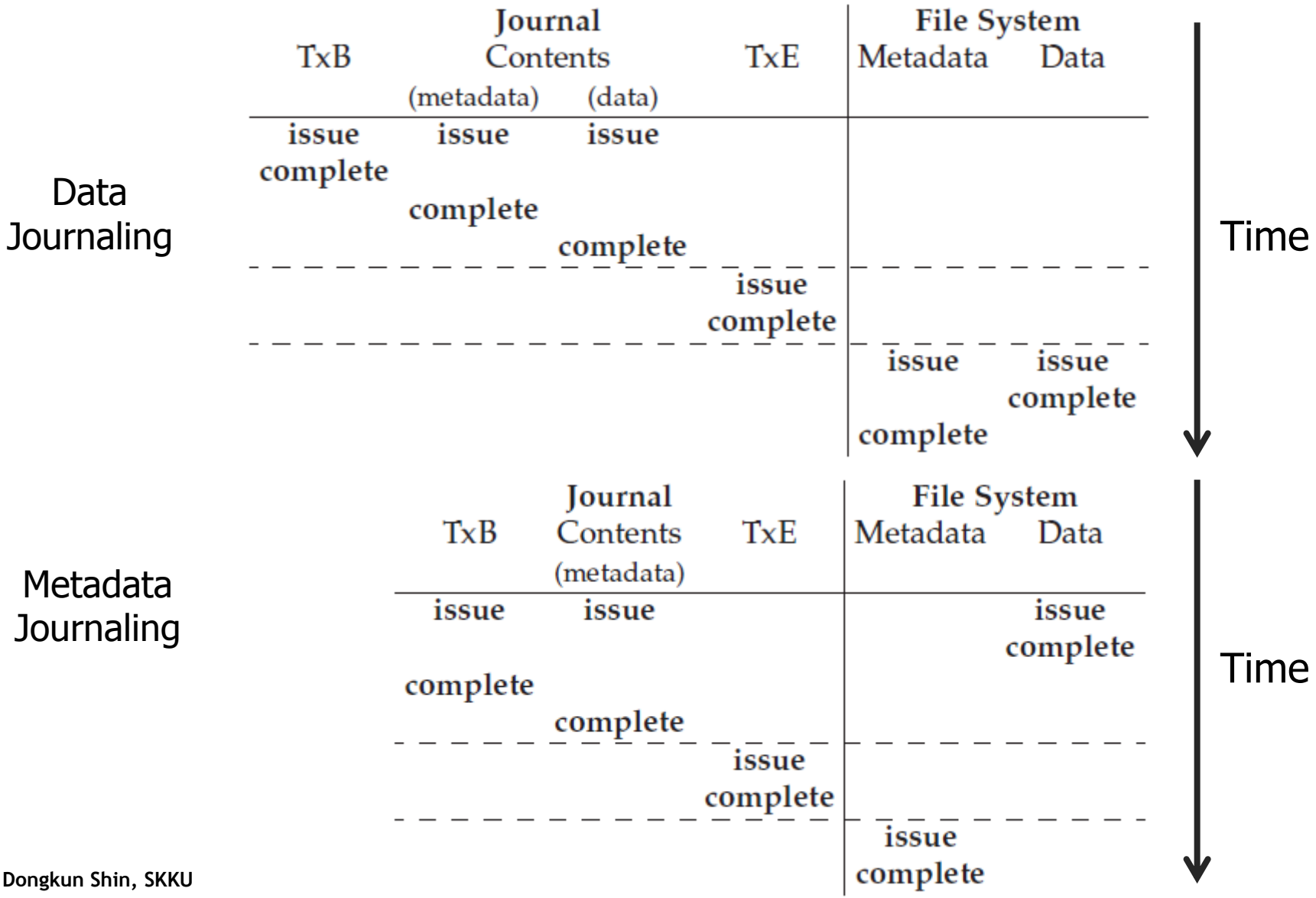


Handling Delete

- **Strategy 1: don't reuse blocks until the delete is checkpointed and freed**
- **Strategy 2: add a revoke record to the log**
  - ext3 used revoke records



Wrapping Up Journaling: A Timeline

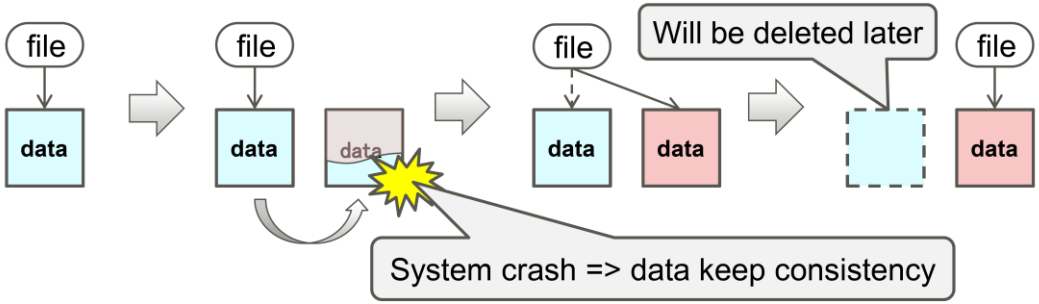


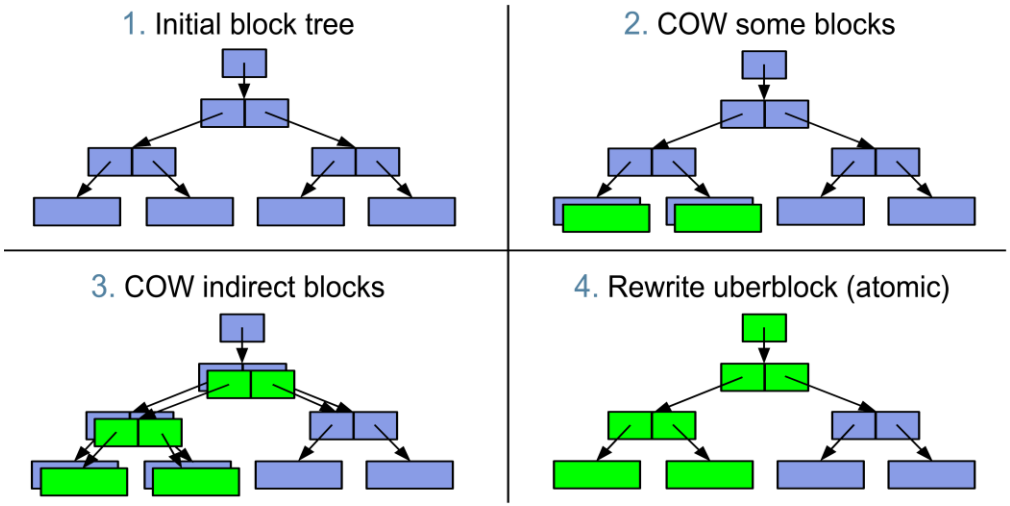
Solution #3: Other Approaches

- Soft Updates
  - carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state
  - e.g., by writing a pointed-to data block to disk before the inode that points to it, we can ensure that the inode never points to garbage
  - Implementing Soft Updates can be a challenge
    - Requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Solution #3: Other Approaches

- **Copy-on-Write (COW)**
  - Sun's ZFS, BTRFS
  - Never overwrites files or directories in place
  - Places new updates to previously unused locations on disk.
  - Log-structured file system (LFS) is an early example of a COW.

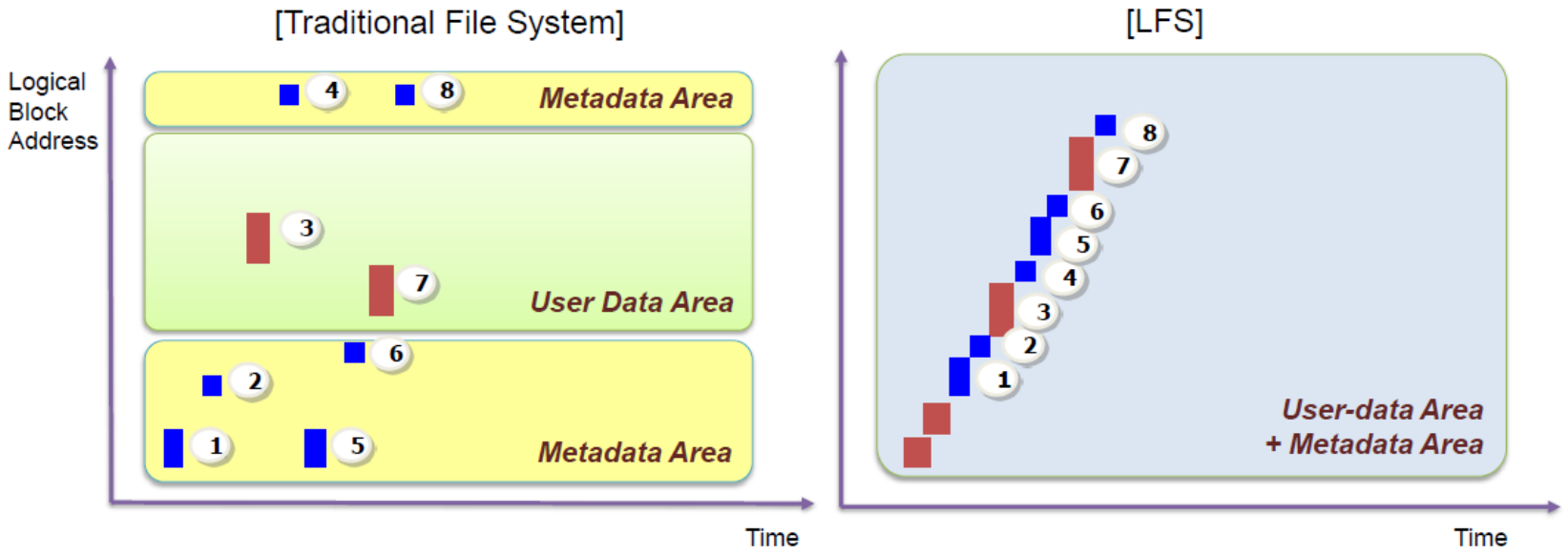




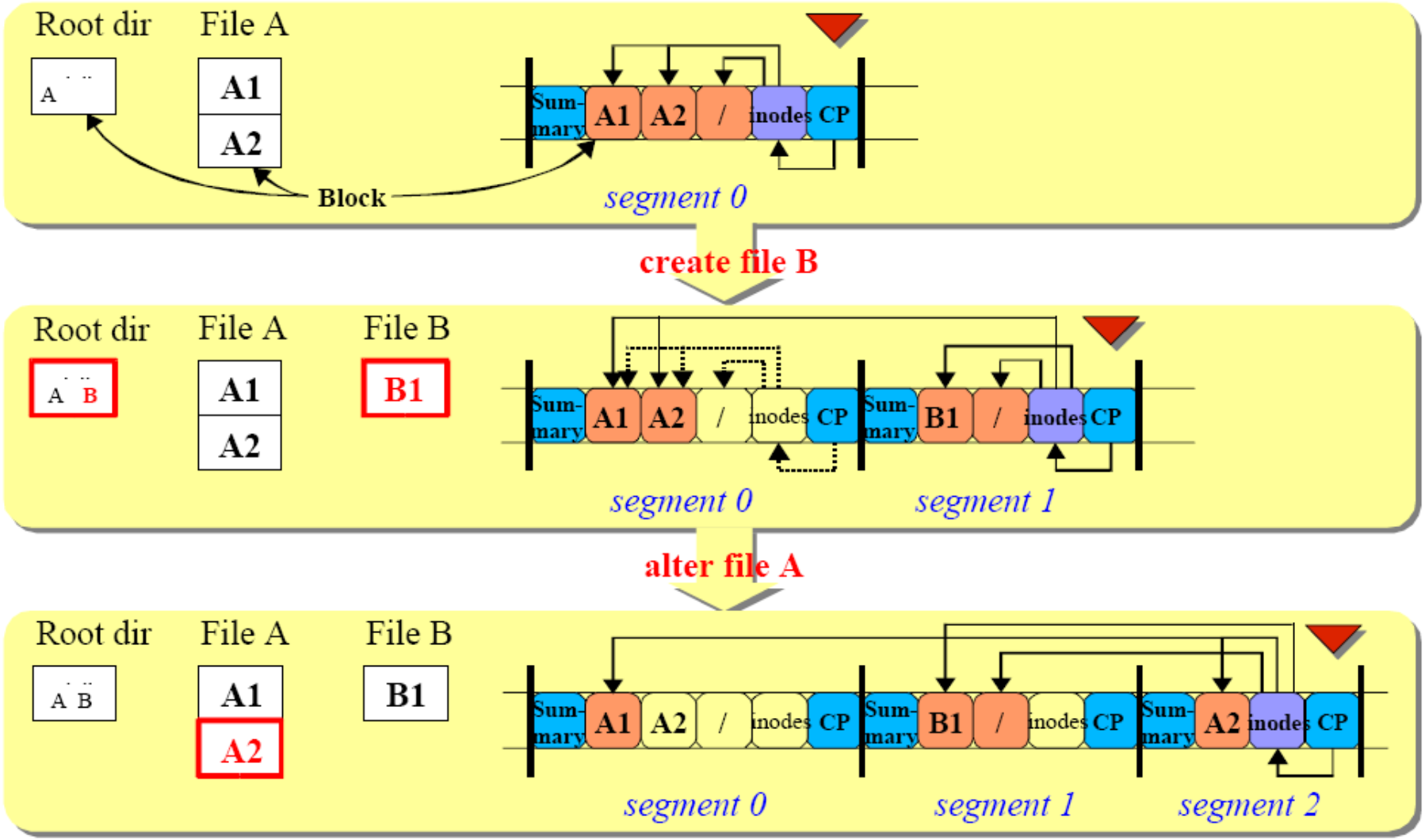
flip the root structure of the file system to include pointers to the newly updated structures

Log-Structured File System

- **Assume the whole disk space as a big contiguous area**
- **Write all data sequentially**
  - Application's random I/O is converted to sequential I/O through LFS
- **"frequent metadata updates" is key challenge in LFS**
- **Recover quickly with "checkpoint"**

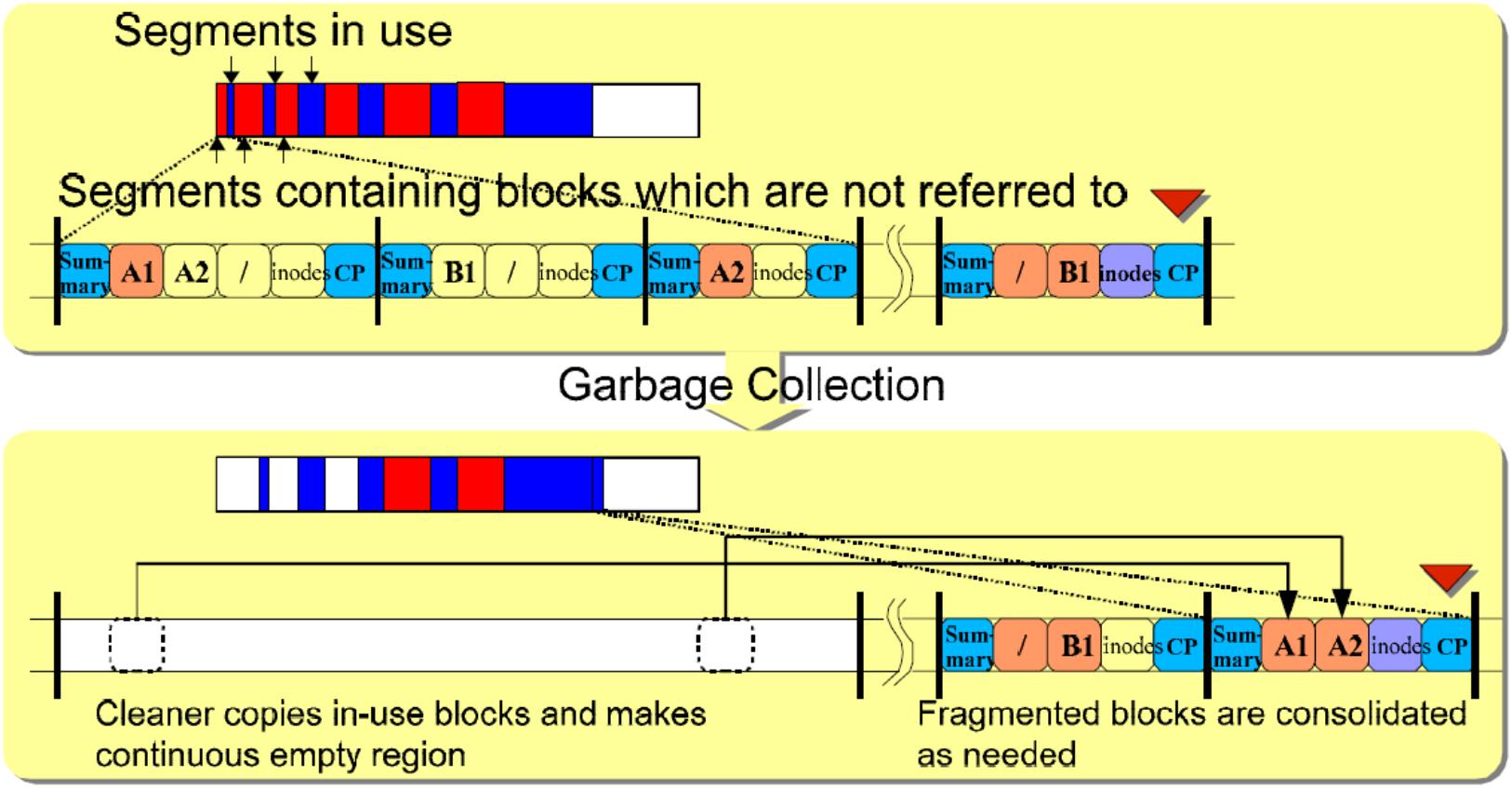


Log-Structured File System



Log-Structured File System

- **Garbage Collection (Cleaner)**
  - Reuse of segments while writing
  - A key challenge in LFS with snapshot



Homework

- Report on Chap 43 (Log-structured File Systems)