Persistence: Journaling

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 7 due Dec 8th @ 11:59pm
  - Tests out later today
  - Cover Log File System on Thursday
  - A `prebuilt_disk` with files and directories has been added to the repository
- Midterm 3 scheduled for Dec 12th in-class
  - Alternate time Dec 18th @ 12:25pm (email me)
- Course Survey starts Nov 29th
- The CS department's annual climate survey to learn about student's experience
  https://cerp.co1.qualtrics.com/jfe/form/SV_9vFybdrfLKUkVcq/?id=wisc_cs

Review Fast File System

- Treat the disk like it's a disk
  - Divide disk into **groups**
  - Each group gets superblock, block bitmap, inode bitmap, inode table, and data blocks
- *Keep related stuff together, keep unrelated stuff far apart*
  - Place directories in group with low number of directories but high number of free inodes
  - Place files (both data and inode) in group with directory
- Large File Exception

Quiz 18 Fast File System

https://tinyurl.com/cs537-fa23-q18

FSCK and Journaling

- Crash Consistency Problem
- Solution 1: FSCK
- Solution 2: Journaling
  - Data Journaling, Recovery, Metadata Journaling
- Solution 3: Other Approaches

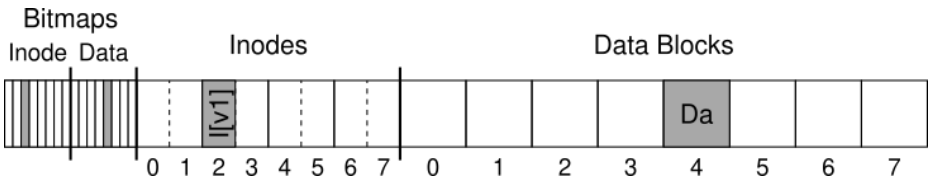
Consistency in File System

The file system consists of several data structures which need to be **consistent with each other**. Consider the case of appending a single data block to an existing file. The following changes must occur:

- Update block bitmap to acquire a free block
- Update inode to point to new data block
- Write data to new data block

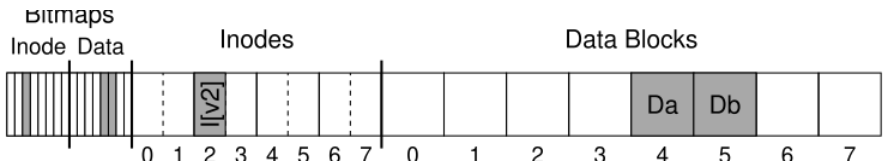
If a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

Before:




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

After:




```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

3 Crash Scenarios

- **Just the data block (Db) is written to disk**
  - Since the inode and data bitmap are not updated, the block is not attached to the file and could still be allocated.
  - FS is consistent but does not contain the new data in Db.
- **Just the updated inode is written to disk**
  - Since inode is updated but data was not written to Db, garbage is attached to the end of the file
  - Since data bitmap was not modified, block could be allocated to a different file (**File System Inconsistency**)
- **Just the updated bitmap is written to disk**
  - data block has not been attached to the file, resulting in a **space leak**

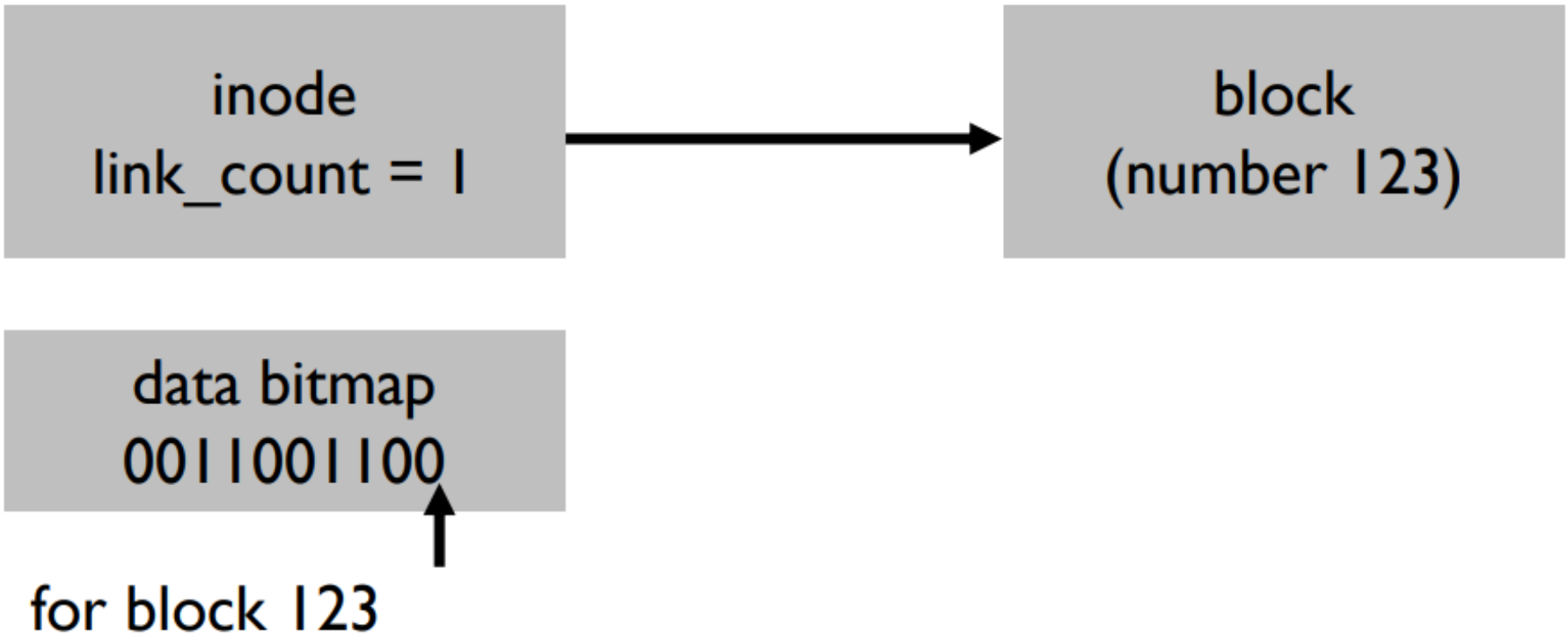
We want file system updates to happen **atomically** from one consistent state to another.

Solution #1: The File System Checker: `fsck`

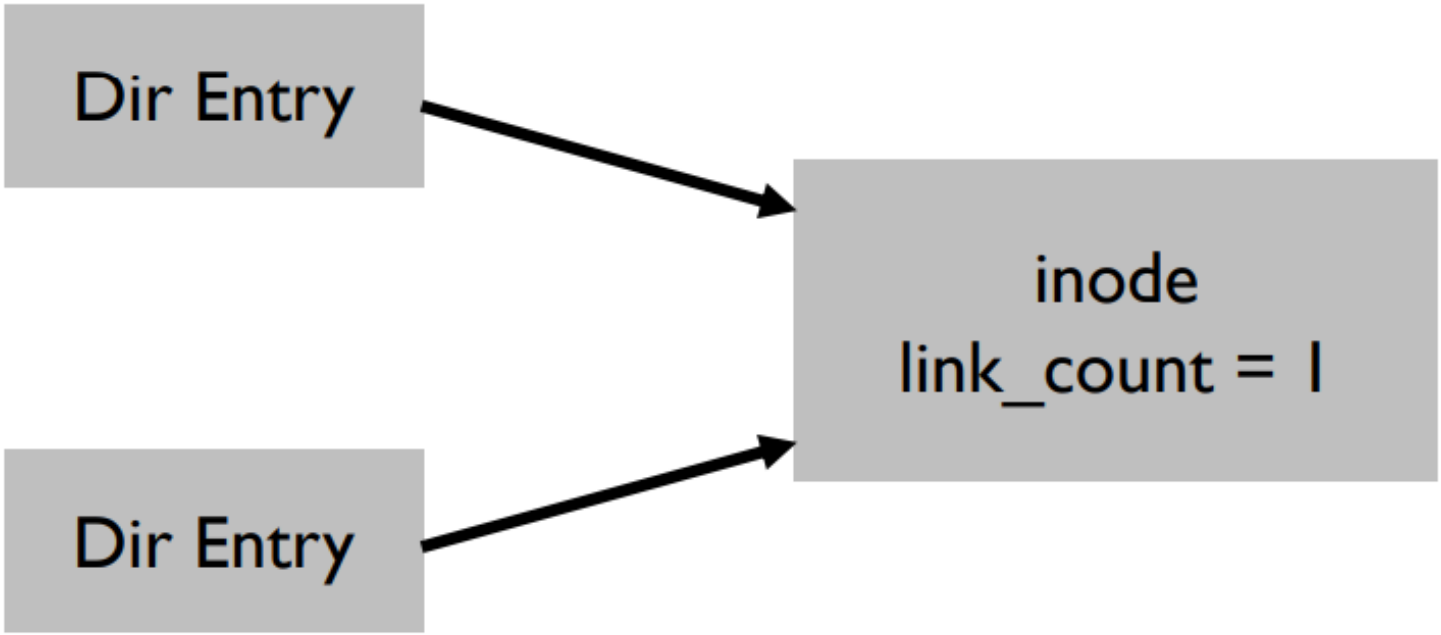
Let inconsistencies happen, then fix by running `fsck`:

- First check **superblock**, that it looks reasonable.
- **Free blocks**: scan the inodes, direct blocks, indirect blocks, etc., to build a list of allocated and free data blocks. Compare this list with the data bitmap. Do the same for the inode bitmap compared to the inode table.
- **Inode state**: check for corruption, e.g. valid type field.
- **Inode links**: verify the link count
- **Duplicates**: Check for duplicate pointers to same data blocks
- **Bad Blocks**: Check for pointers outside range of valid data blocks
- **Directory Checks**: Integrity check (each directory contains a `.` and `..` pointer to proper values, inodes exist, each directory linked to once).

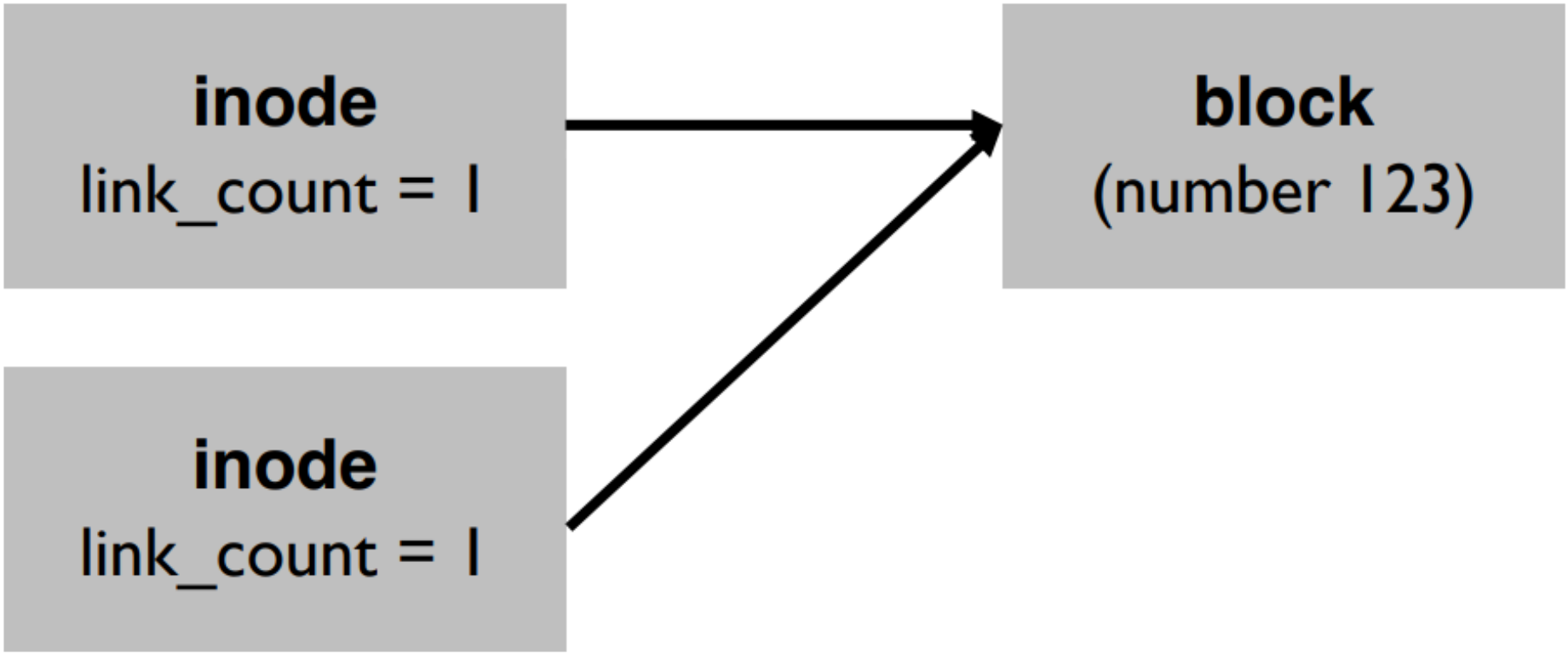
Free Blocks Example



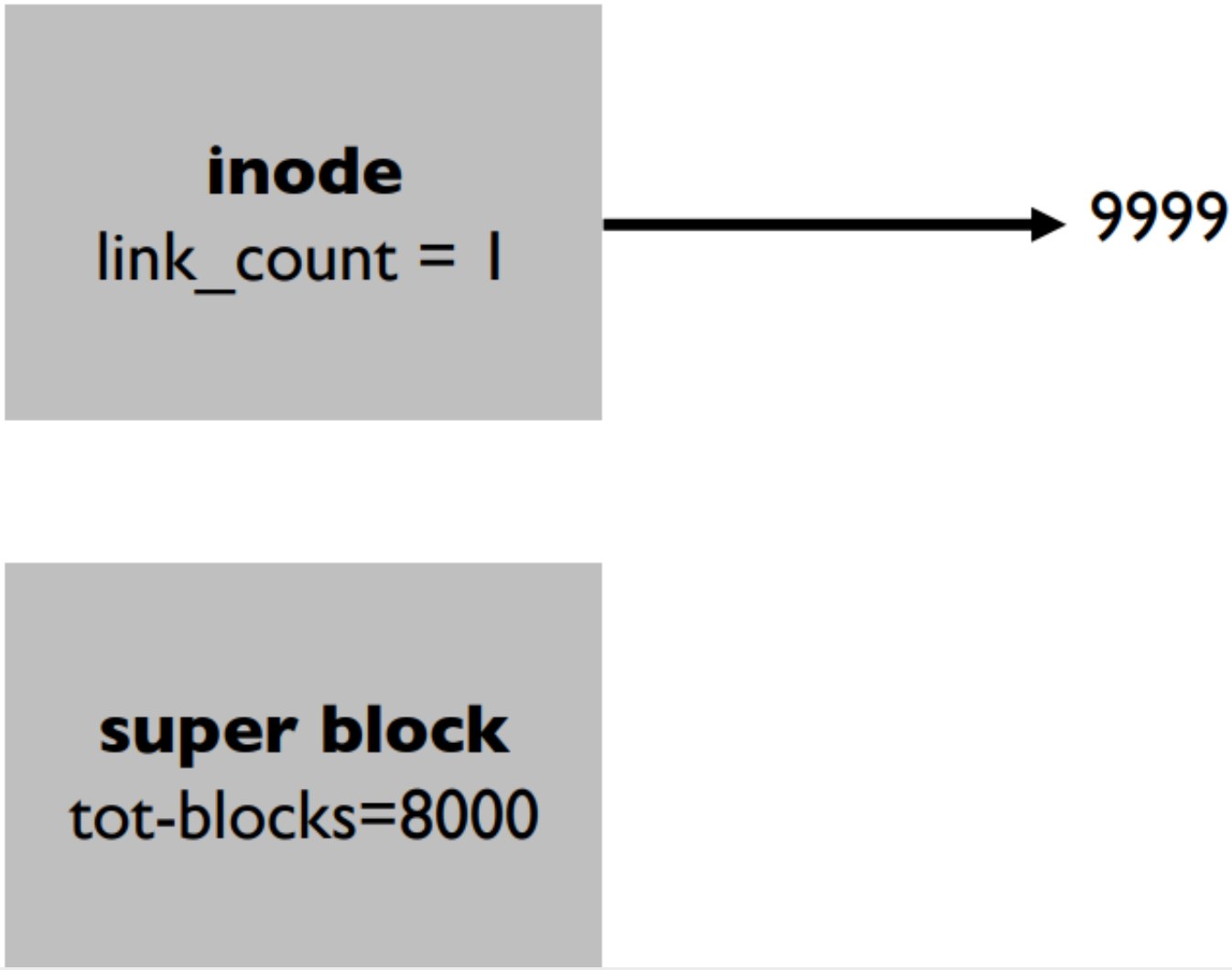
Link Count Example



Duplicate Pointers

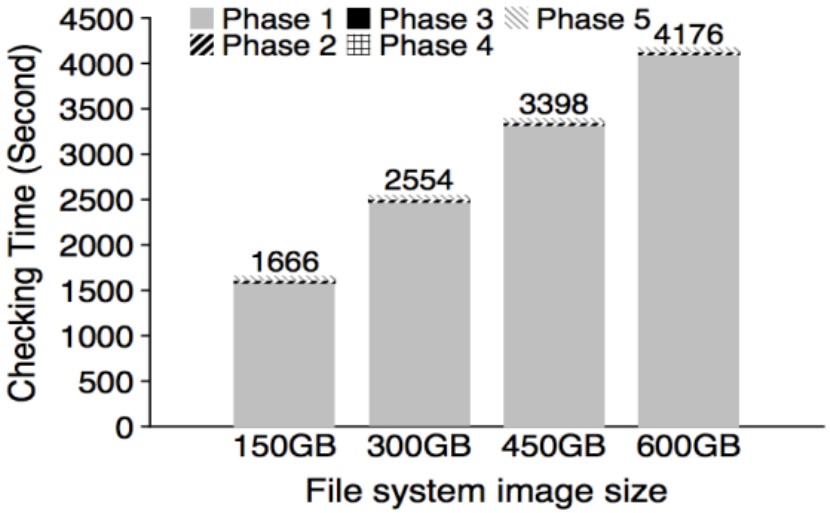


Bad Pointer



Problems with fsck

- Not always obvious how to fix
- Don't know "correct" state, just consistent state
- Slow



Checking a 600GB disk takes ~70 minutes

ffsck: fast file system checker
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

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

Before updating the disk, first write down a little note (in the **journal** on disk) describing what you are about to do.

On a crash during update, go back and look at the journal.

Adds a bit of work during update but greatly reduces work required during recovery

Data Journaling

1. **Journal write** – Write out TxB, I[v2], B[v2], and Db, wait for these to complete
2. **Journal commit** – Write out TxE, wait for this to complete, transaction is **committed**
3. **Checkpoint** – Write the contents of the update to their final on-disk locations

The TxB block contains information about the pending update (e.g. final addresses for blocks and a **transaction identifier**)
The disk guarantees any 512-byte write is atomic – the TxE is a single 512-bytes.

Recovery

Crashes can occur at any time:

- If occurs before the commit, the update is skipped
- If occurs after the commit but before the checkpoint finishes:
  - On boot, scan the log and look for committed transactions and **replay** them in order: called **redo logging**
  - In the worse case, a transaction might be performed again

Batching Log Updates

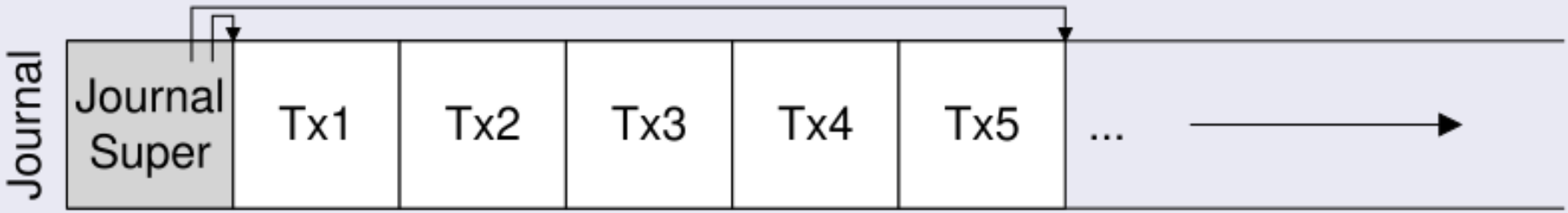
- This journaling protocol adds a lot of extra disk traffic
- Some FS do not commit each update one at a time, rather buffer all updates into a **global transaction**

Making Log Finite

Journaling file systems treat the log as a **circular log** adds a 4th step to protocol:

4. **Free**: some time later, mark the transaction free in the journal by updating the journal superblock

Journal Superblock keeps track of portion of journal with non-checkpointed transactions.



Metadata Journaling

With data journaling protocol, each data block is written to the disk **twice**, which is especially painful during sequential writes.

A simpler form of journaling is called **ordered journaling** or **metadata journaling**.

Its nearly the same except data is **not** written to the journal.

Data is written out before the metadata is placed into the journal transaction.

Metadata Journaling Protocol

By forcing data write first, a file system can guarantee that a pointer will never point to garbage – commonly used technique.

1. **Data write:** Write data to final location; wait for completion (optional)
2. **Journal metadata write:** Write the begin block and metadata to log; wait for completion
3. **Journal commit:** Write the transaction commit block (containing TxE)
4. **Checkpoint metadata:** Write the contents of the metadata to final locations
5. **Free:** Later, mark the transaction free in journal superblock

Tricky Case: Block Reuse

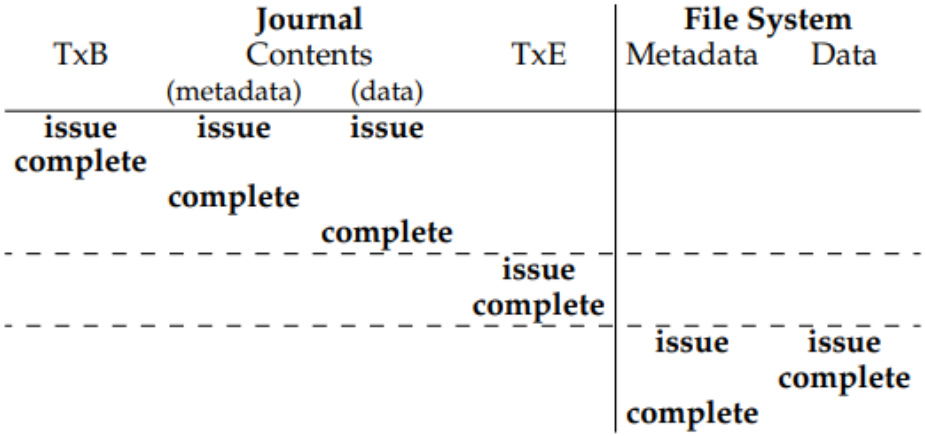
- User adds file `foo` to a directory
  - Causes contents of directory (metadata) to be written to log
- User deletes directory and all its contents (directory data block freed)
- User creates a new file with data, which is allocated the freed data block and writes to it

On a crash, replaying the log would cause the directory data block to be written, wiping out the new file's data

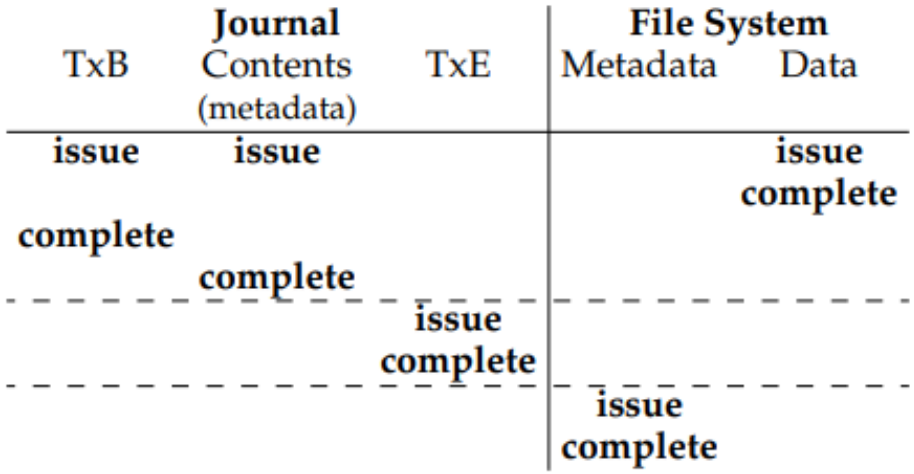
Linux's `ext` FS has **revoke** entries in log. Replaying first checks for these entries and doesn't replay entries that are revoked

Timelines

Data Journaling Timeline



Metadata Journaling Timeline



- Note the **write issue**s which can occur simultaneously
- Completion time is determined by the I/O subsystem, which may reorder writes to improve performance
- Horizontal dashed lines representing barriers waiting for completion of writes are enforced by FS for protocol correctness

Solution #3: Other Approaches / Ideas

- **Soft Updates**
  - Carefully order all writes to the FS
  - Requires intricate knowledge of each FS data structure
- **Copy-On-Write (COW)**
  - Never overwrite files or directories, rather, place new updates at previously unused locations on disk
  - Basis of **Log File System** (lecture next time)
- **Backpointer-based Consistency (BBC)**
  - Developed here at UW
  - Every block contains a back-pointer, so data blocks point to the inode they belong to.
  - Consistency can be checked between the forward pointers in the inode with the backpointers
- **Optimistic Crash Consistency**
  - Uses generalized form of **transaction checksum**, allowing more concurrent writes