ILLINOIS TECH

College of Computing

CS 450 Operating Systems
File System Consistency

Yue Duan

Why Consistency is an Issue?

- File system may perform several disk writes to serve a single request
  - Caching ===> not knowing the exact time at which the writes might happen
- If FS is interrupted between writes, may leave data in inconsistent state
- What can cause the interrupt?
  - power loss
  - hard reboot
  - kernel panic
  - FS bugs
  - etc.

Running Example

- Interrupts are practically impossible to avoid
  - **inconsistencies will happen**
  - need a mechanism to fix inconsistent state
- Consider appending a new block to a file
  - e.g., because of a write() syscall
- What are the blocks that need to be written?
  - FS data bitmap
  - file's inode (inode table block containing the inode)
  - new data block

Possible Inconsistencies

- What happens if crash happens after updating these blocks?
  - in terms of FS consistency

1). bitmap:	leaked space (block not usable anymore)
2). data:	nothing bad
3). inode:	point to garbage + another file may use block
4). bitmap and data:	leaked space (block not usable anymore)
5). bitmap and inode:	point to garbage
6). data and inode:	another file may use block

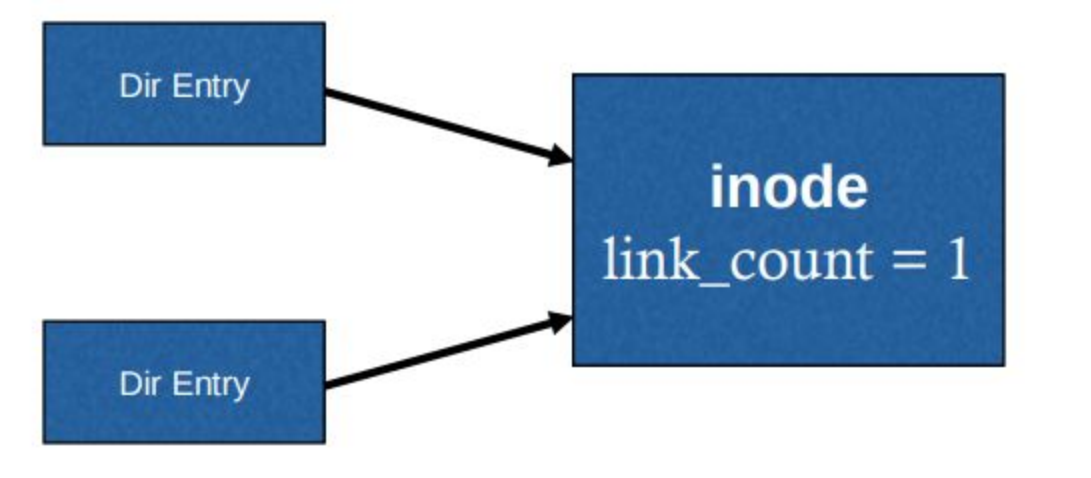
Solution #1: FSCK

- File System Checker
  - often read “FS-check”
- Strategy:
  - after crash, scan the whole disk for contradictions and “fix” if needed
  - keep file system off-line until FSCK completes
- Example, how to tell if data bitmap block is consistent with inodes?
  - read every valid inode + indirect blocks
  - if pointer to data block, corresponding bit should be 1; else bit is 0

FSCK Checks

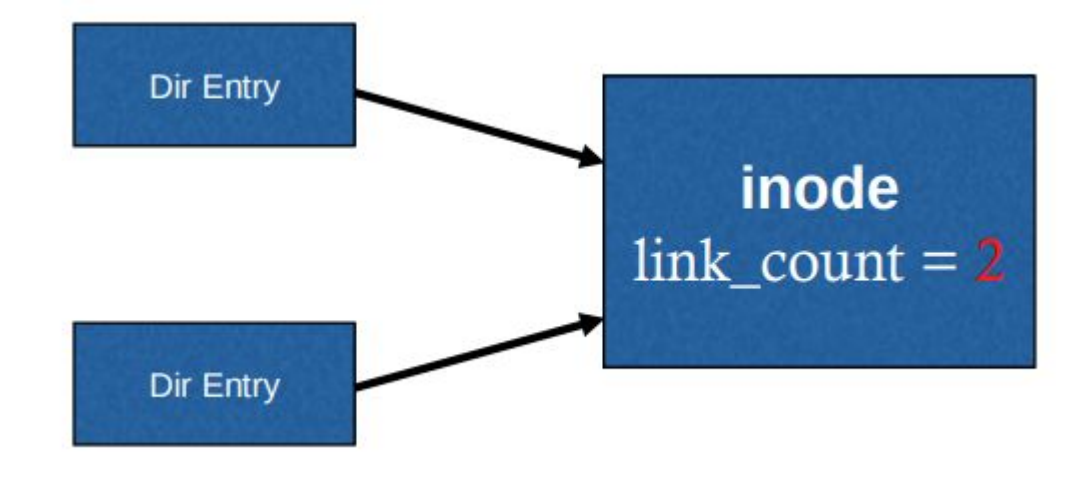
- First question: How to check for consistency?
  - heuristic checks based on what we expect from a ‘consistent’ FS
  - Examples:
    - Do superblocks match?
    - Is the list of free blocks correct?
    - Do number of dir entries equal inode link counts?
    - Do different inodes ever point to same block?
    - Are there any bad block pointers?
    - Do directories contain “.” and “..”?
- Second question: how to solve problems once found?
  - not always easy
  - goal: reconstitute some consistent state

Example 1: Link Count



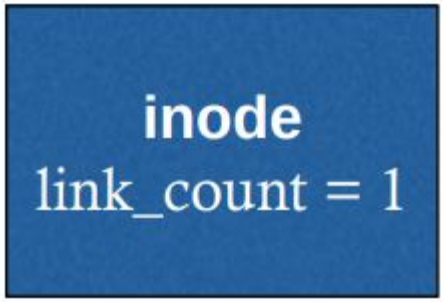
- How to fix to restore consistency?

Example 1: Link Count



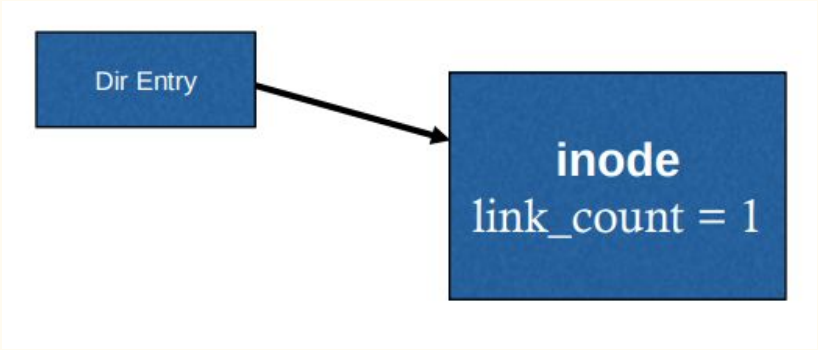
- Simple fix!

Example 2: Link Count



- How to fix to restore consistency?

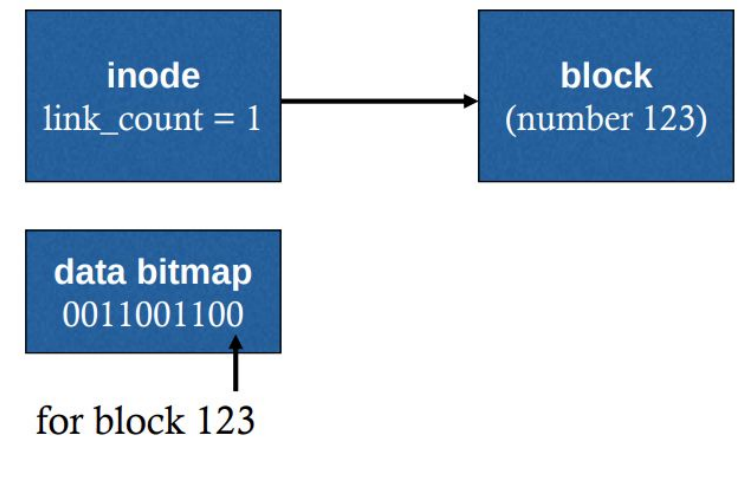
Example 2: Link Count




```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
```

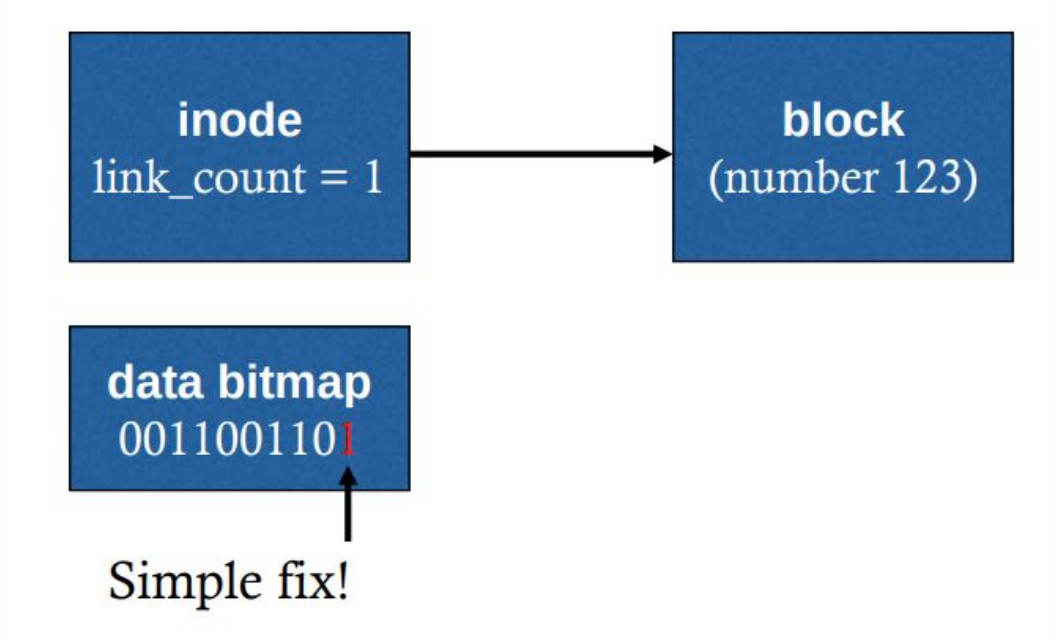
- The lost+found directory
  - a special directory that contains files that have been deleted or lost in a disk operation

Example 3: Data Bitmap

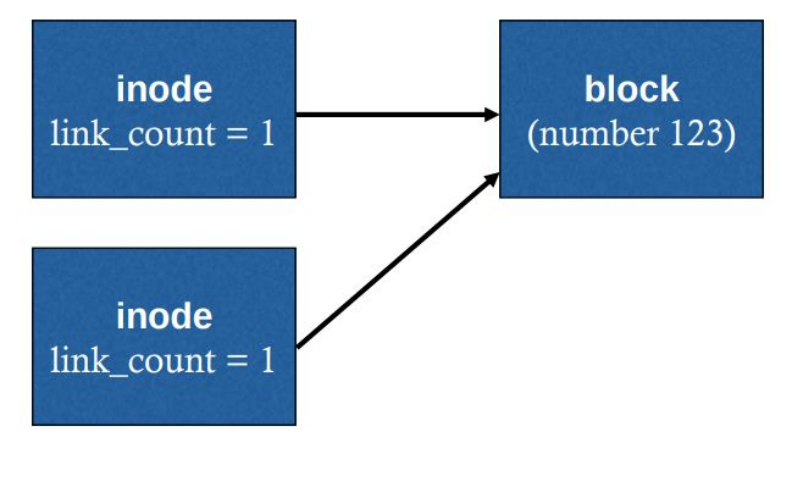


- How to fix to restore consistency?

Example 3: Data Bitmap

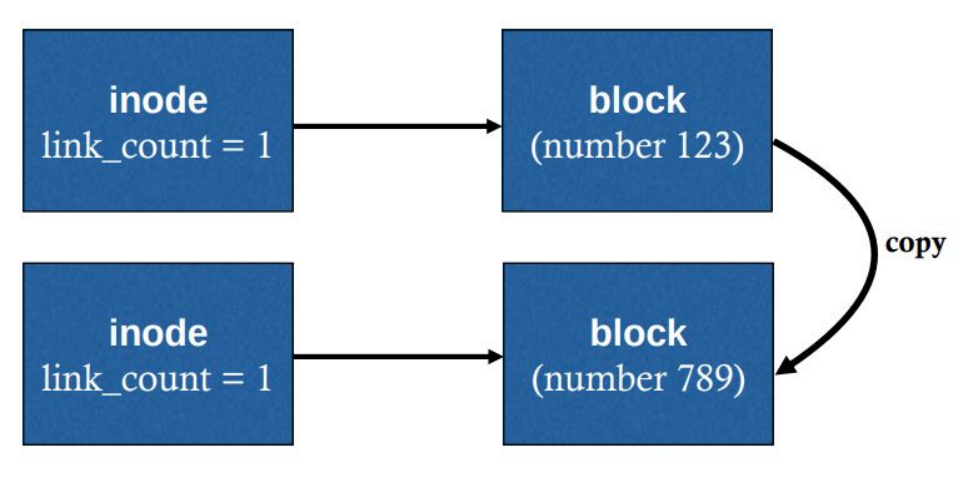


Example 4: Duplicate Pointers



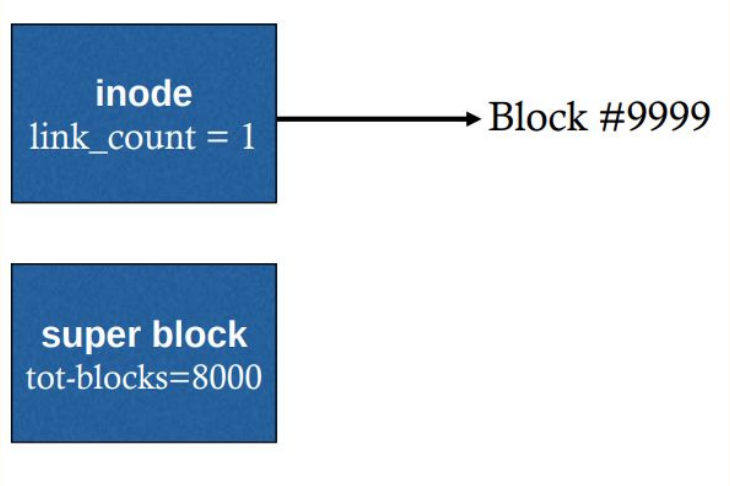
- How to fix to restore consistency?

Example 4: Duplicate Pointers



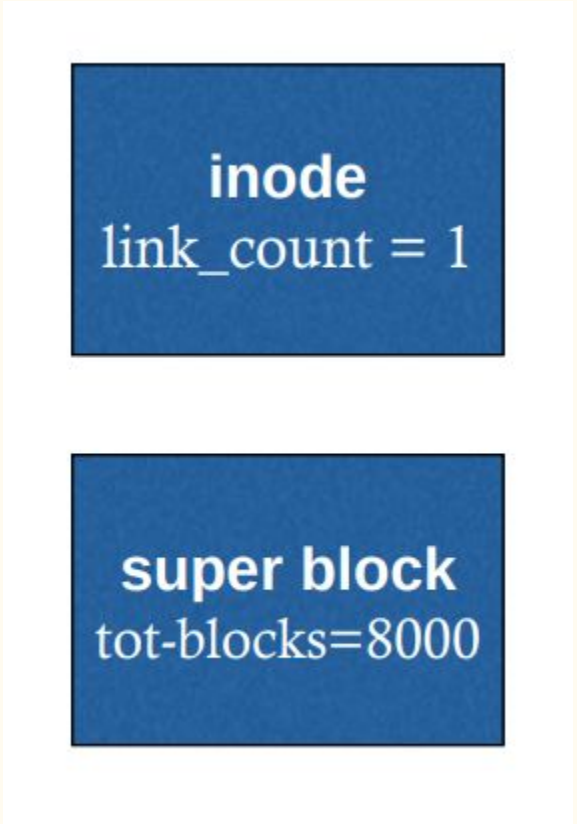
- Simple enough, but is this correct?

Example 5: Bad Pointer



- How to fix to restore consistency?

Example 5: Bad Pointer

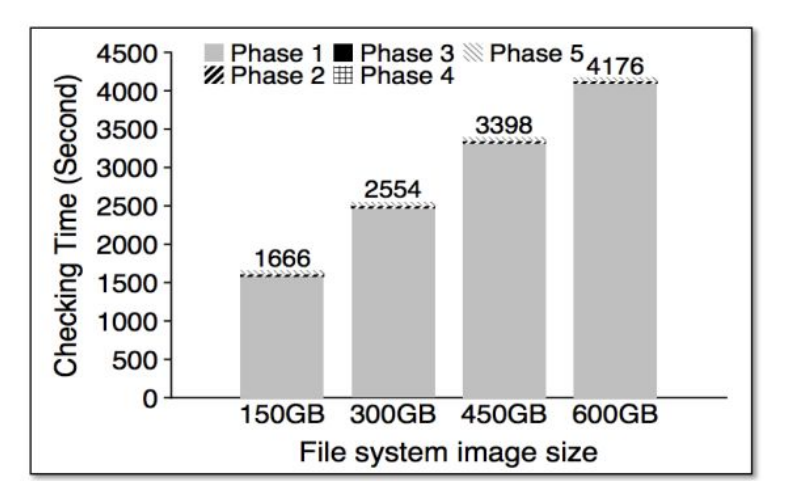


- Simple enough, but is this correct?

Problems with FSCK

- Problem 1: functionality
  - not always obvious how to fix file system image
  - don’t know “correct” state, just consistent one
- Problem 2: performance
  - FSCK is awfully slow!

FSCK is Very Slow



Source: "ffsck: The Fast File System Checker"

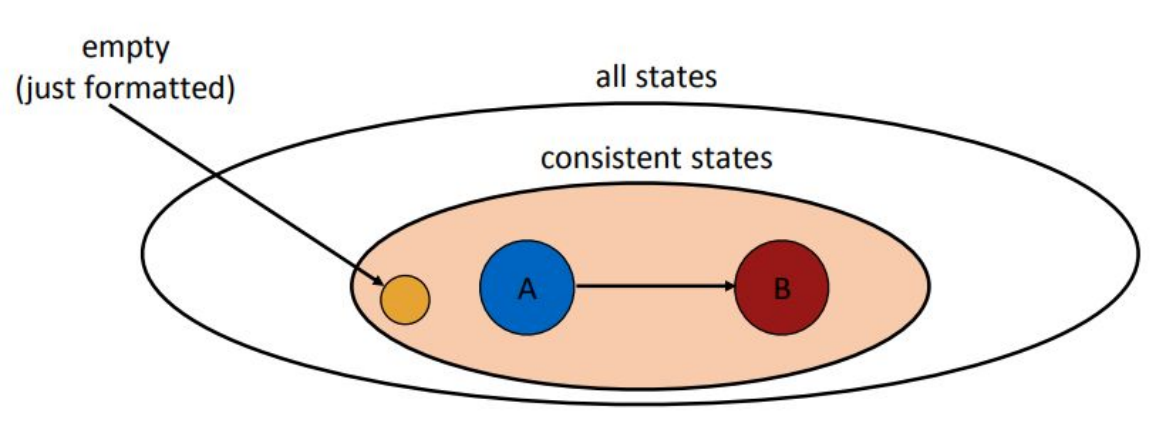
- Checking a 600GB disk takes ~70 minutes

Solution #2: Journaling

- Goals
  - 1) ok to do some **recovery work** after crash, but not to read entire disk
  - 2) don't move file system to just any consistent state, get **correct state**
- Strategy: achieve **atomicity** when there are multiple disk updates
- Definition of atomicity for concurrency
  - operations in critical sections are not interrupted by operations on related critical sections
- Definition of atomicity for persistence
  - collections of writes are not interrupted by crashes
  - either "all new" or "all old" data is visible

Consistency vs. Correctness

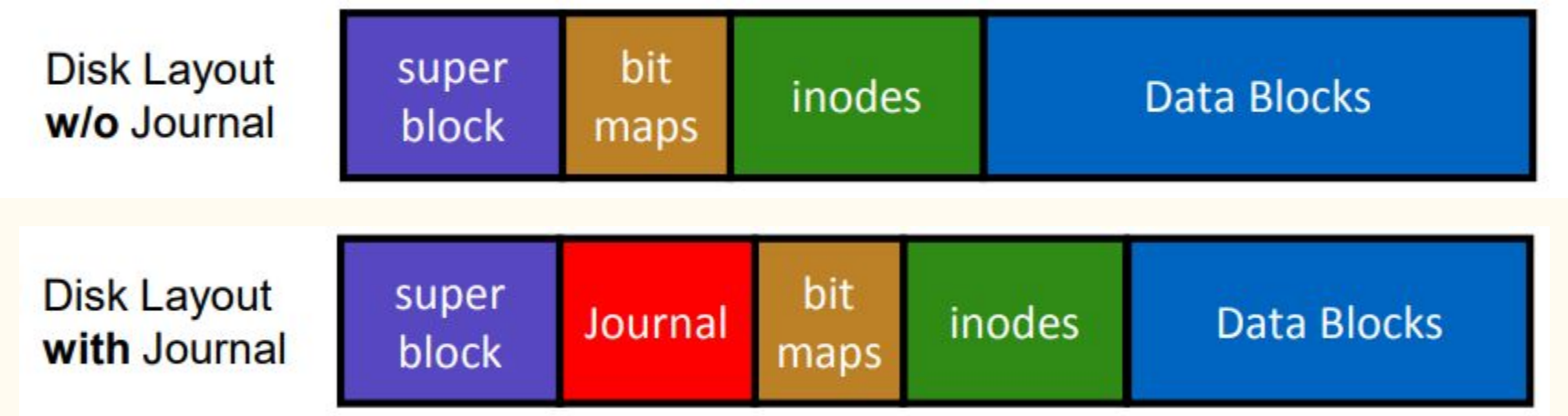
- Say a set of writes moves the disk from state A to B



FSCK gives consistency.
Atomicity gives A or B.

Journaling Strategy

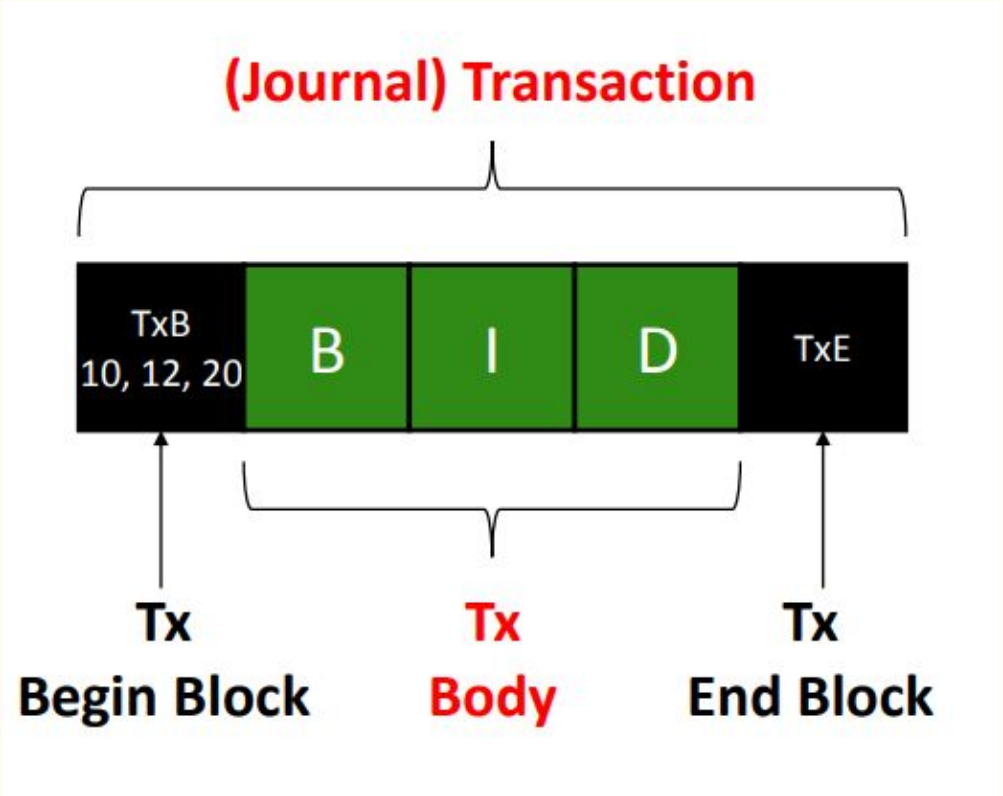
- Log all disk changes in a journal before writing them to file system
- Journal itself is a “temporary” persistent space on disk
  - could be the same disk as FS or a different one (for added reliability)



How Journaling Works

- Consider the running example
  - need to write a data-bitmap block (B), an inode table block (I), and a new data block (D)
  - let's say B is block #10, I is block #12, and D is block #20
- Before writing to those blocks, store intended changes in the journal

How Journaling Works



How Journaling Works

- Order of operations
  - 1) **Journal write**: write the following to the journal
    - a Tx Begin block with disk block numbers of all blocks that will be changed
    - new content of blocks that will be changed (Tx Body)
    - a Tx End block to indicate that all the intended changes are safely in the journal
  - 2) **Checkpoint**: Write the actual FS blocks

Crash Recovery Using Journal

- Journal transaction ensures atomicity
  - all disk writes needed to take FS from “one consistent state” to “next consistent state” are recorded first
  - this ensures atomicity w.r.t. crashes
- If a crash happens during journal write
  - ignore the half-written transaction during recovery
  - crash happened during journal write
    - → no checkpointing took place → FS blocks are not changed

Crash Recovery Using Journal

- If a crash happens after journal write but before (or during) checkpointing
  - during recovery, replay transaction by writing the recorded changes to FS blocks
- This is correct even if crash happened during checkpointing
  - i.e., even if some FS blocks were written before crash
  - why?
  - because we will just overwrite them with the same data

Order of Writes

- Question: in what order should we send the writes to disk?
  - Does the order between journal write and checkpointing matter?
    - of course!
  - What happens if checkpointing begins before journal writes are finished?
    - inconsistent FS state in case of crash
  - → Checkpointing should only begin after the whole transaction is safely on the disk

Order of Writes

- Does the order of journal writes matter?
  - TxB, Tx Data and TxE
  - hint: what is the purpose of TxE block?
- Disk can do TxB and Tx Body in any order
- TxE written last to indicate Tx is fully in the journal
- Revised order of operations:
  - 1) Journal write (TxB and Tx Body)
  - 2) Journal commit (write TxE)
  - 3) Checkpoint

Finite Journal

- Journal size is limited
  - at some point we should free up journal space
  - after a transaction is checkpointed, we can free its space in the journal
- Journal often treated as a circular FIFO
  - with pointers to the first and last not-checkpointed transactions
  - store this information in a journal superblock
- Revised order of operations:
  - 1) Journal write (TxB and Tx Body) – advance the FIFO tail pointer
  - 2) Journal commit (write TxE) – advance the FIFO tail pointer
  - 3) Checkpoint
  - 4) Free – advance the FIFO head pointer

Journaling Modes

- Data journaling
  - both data + metadata in the journal
  - lots of data written twice, safer
- Metadata journaling + ordered data writes
  - only metadata in the journal
  - data writes should happen before metadata is in journal
    - why not after?
    - because inode can point to garbage data if crash
  - faster than full data, but constrain write orderings

Journaling Modes

- Metadata journaling + unordered data writes
  - data write can happen anytime w.r.t. metadata journal
  - fastest, most dangerous
  - still guarantees structural consistency
- Ordered metadata journaling is the most popular
  - NTFS, ext3, XFS, etc.
- In ext3, you can choose any journaling mode

THANK YOU!