FS Consistency & Journaling

Nima Honarmand
(Based on slides by Prof. Andrea Arpaci-Dusseau)

Why Is Consistency Challenging?

- File system may perform several disk writes to serve a single request
  - Caching makes things worse by 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 interrupt write operations?
  - Power loss and hard reboot
  - Kernel panic (could be due to bugs not in FS)
  - FS bugs
- These are practically impossible to avoid → **inconsistencies will happen**
  - Need a mechanism to recover from (or fix) inconsistent state

Running Example

- 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 after only updating some of these blocks?
  - In terms of FS consistency

a) bitmap:	leaked space (block not usable anymore)
b) data:	nothing bad
c) inode:	point to garbage + another file may use block
d) bitmap and data:	leaked space (block not usable anymore)
e) bitmap and inode:	point to garbage
f) data and inode:	another file may use block

How to fix file system inconsistencies?

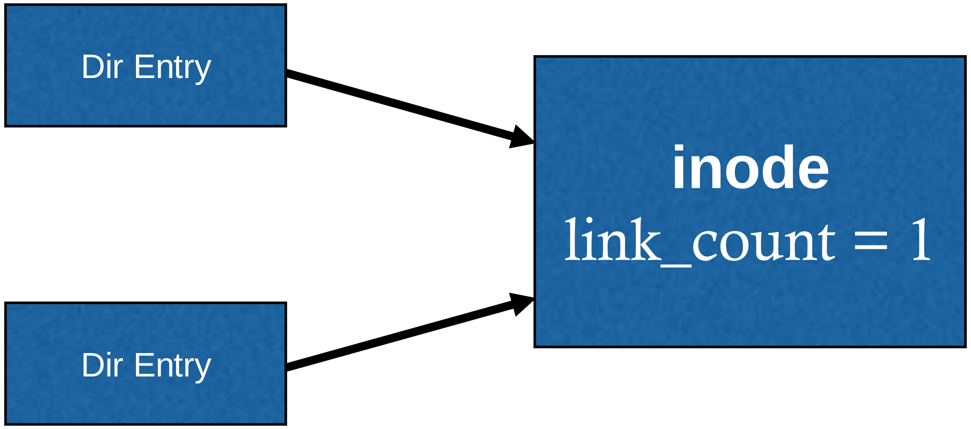
Solution #1: FSCK

- File System Checker
  - Often read “FS-check”
- Strategy:
  - After crash, scan whole disk for contradictions and “fix” if needed
  - Keep file system off-line until FSCK completes
- For 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
- Interlude: how does OS know if an FSCK is needed?
  - Superblock is marked “dirty” when mounted
  - Upon clean shutdown/reboot, kernel removes the “dirty” mark

FSCK Checks

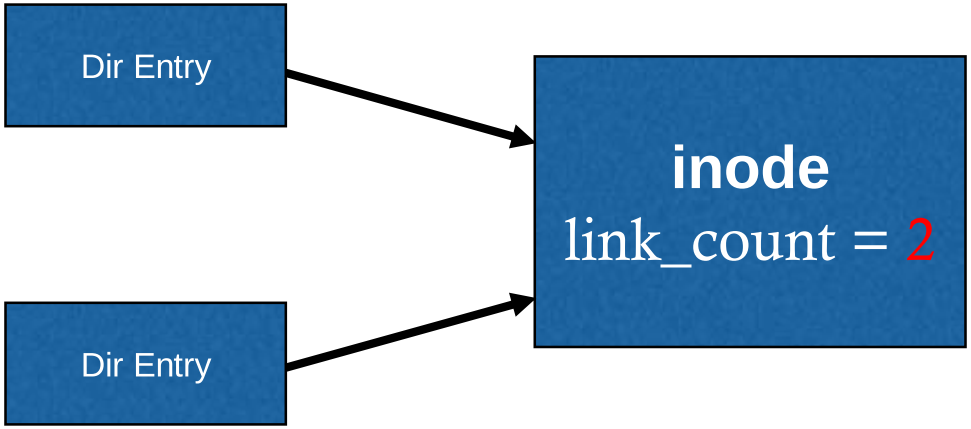
- **First big question: How to check for consistency?**
  - Hundreds of types of checks over different fields…
  - All are heuristic checks based on what we expect from a "consistent" FS state
- Do superblocks match?
  - FS usually keeps multiple superblock copies for reliability reasons
- Do directories contain "." and ".."?
- Do number of dir entries equal inode link counts?
- Do different inodes ever point to same block?
- …
- **Second big question: how to solve problems once found?**
  - Not always easy to know what to do
  - Goal is to reconstitute some consistent state

Example 1: Link Count



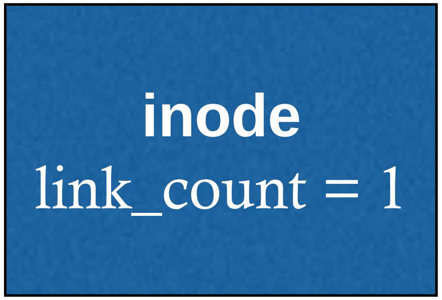
How to fix to restore consistency?

Example 1: Link Count



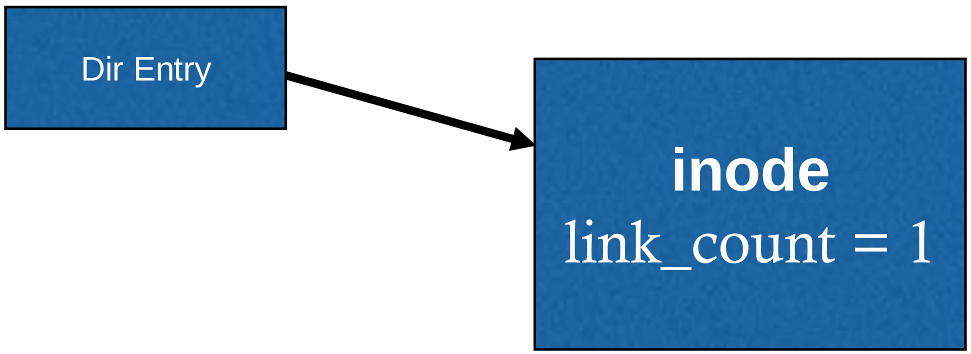
Simple fix!

Example 2: Link Count



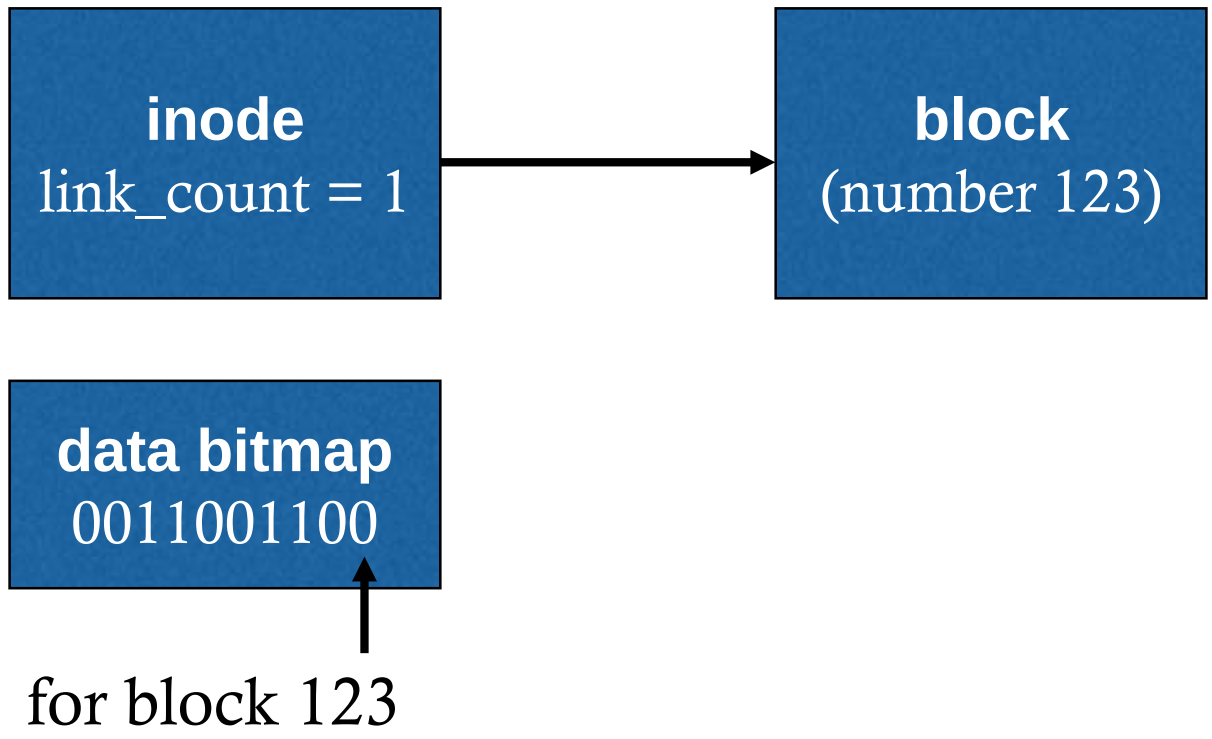
How to fix to restore consistency?

Example 2: Link Count



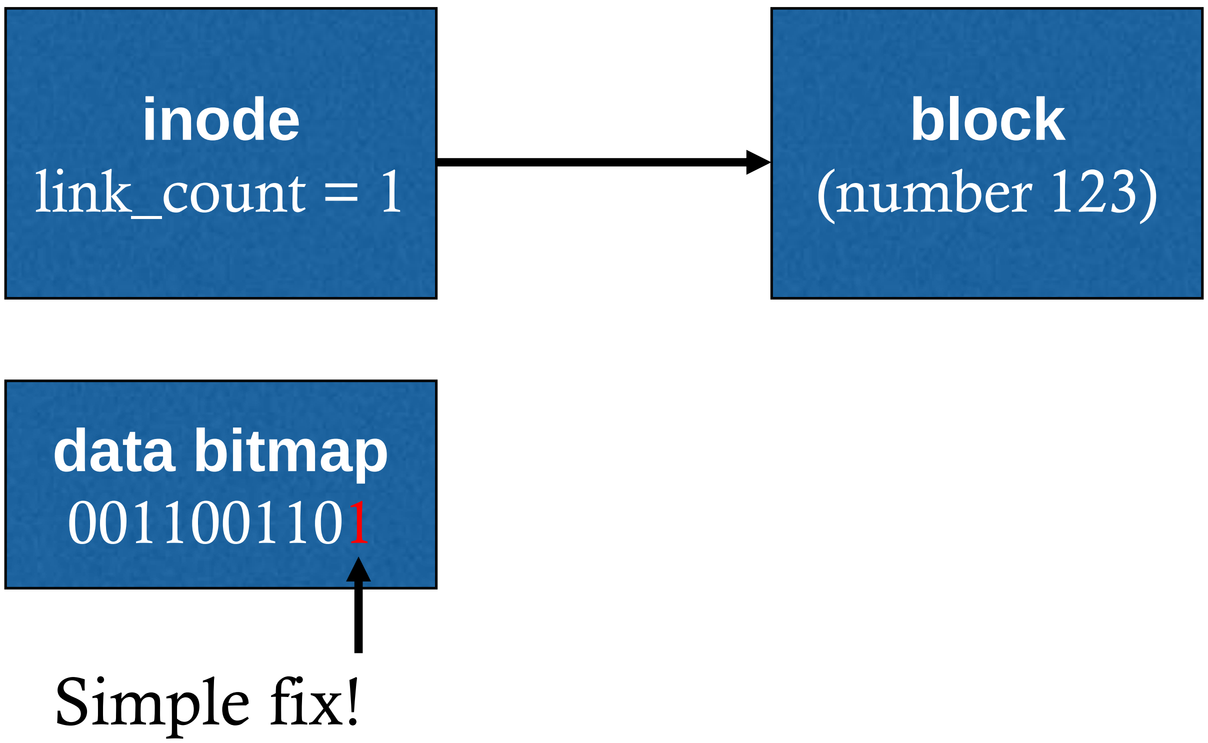

```
ls -l /
total 150
drwxr-xr-x  401 18432 Dec 31  1969 afs/
drwxr-xr-x.   2 4096  Nov  3 09:42 bin/
drwxr-xr-x.   5 4096  Aug  1 14:21 boot/
dr-xr-xr-x.  13 4096  Nov  3 09:41 lib/
dr-xr-xr-x.  10 12288 Nov  3 09:41 lib64/
drwx------.   2 16384 Aug  1 10:57 lost+found/
...
```

Example 3: Data Bitmap

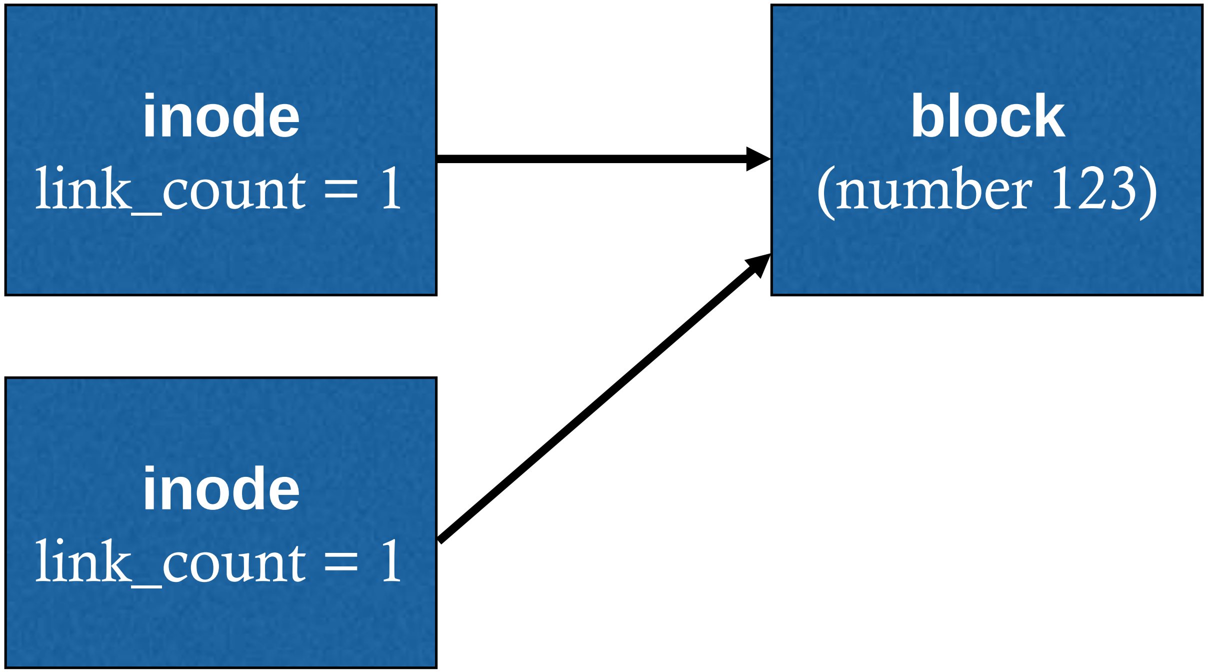


How to fix to restore consistency?

Example 3: Data Bitmap

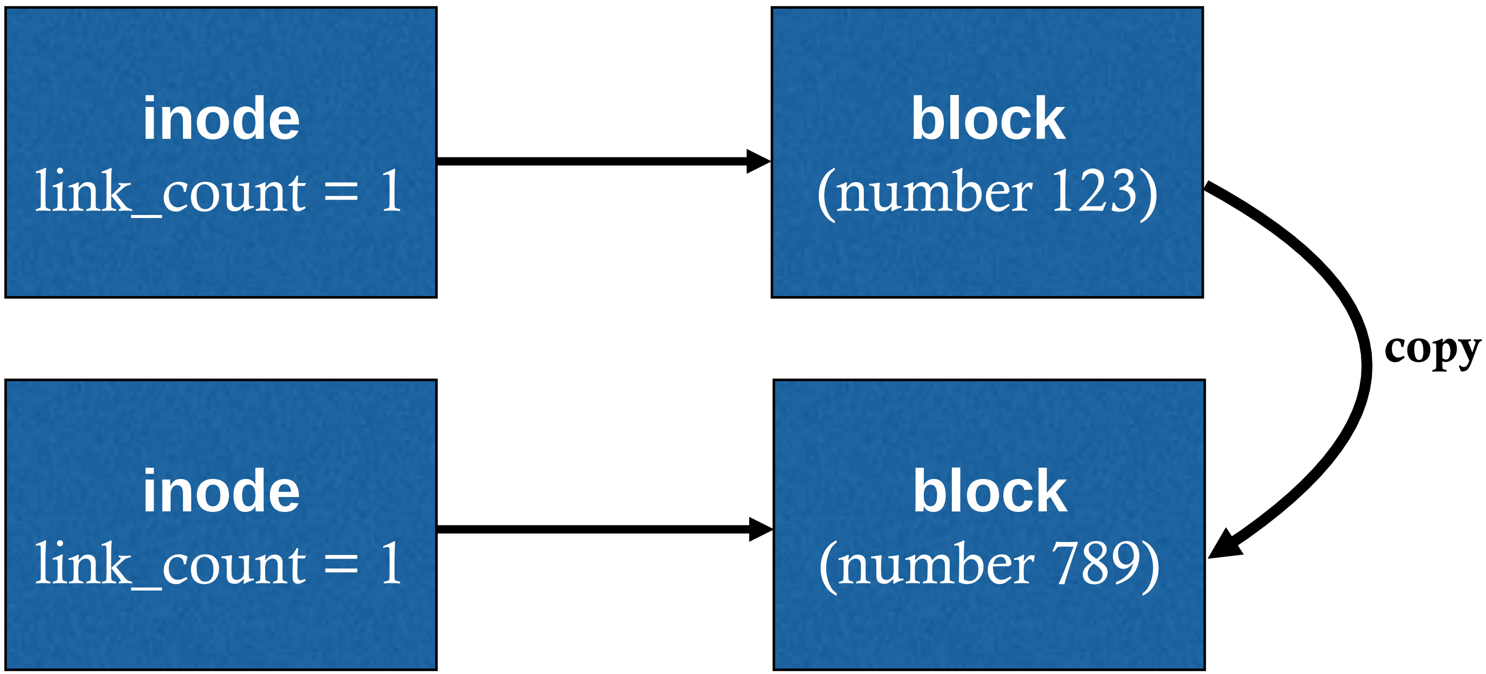


Example 4: Duplicate Pointers



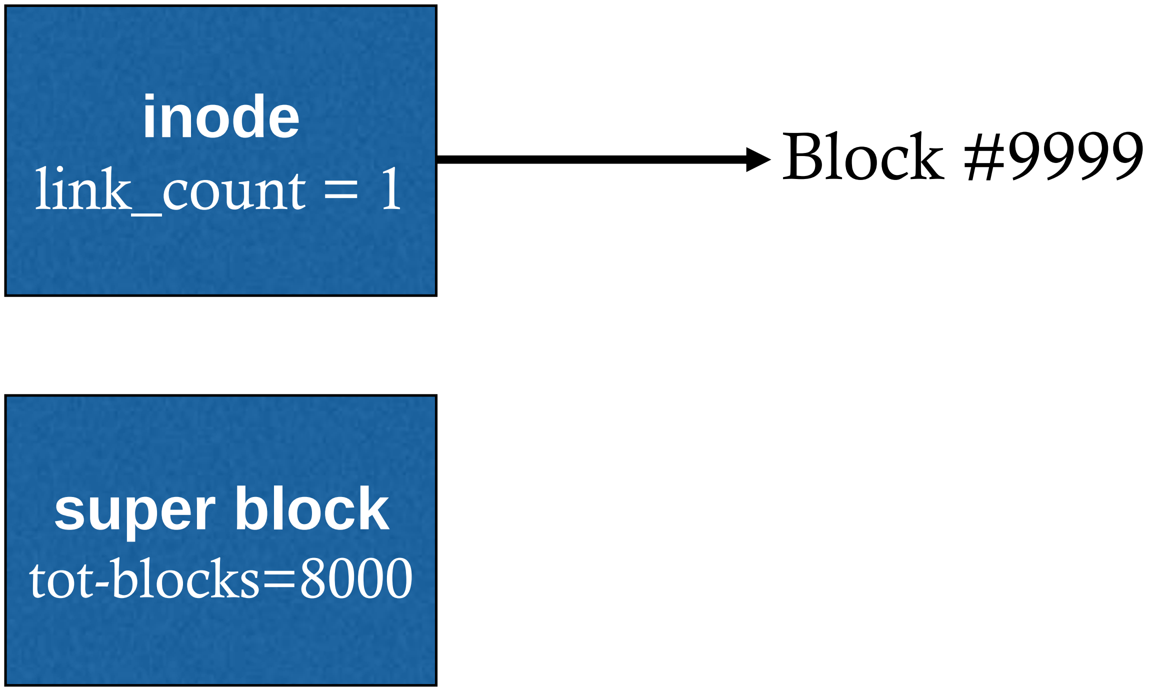
How to fix to restore consistency?

Example 4: Duplicate Pointers



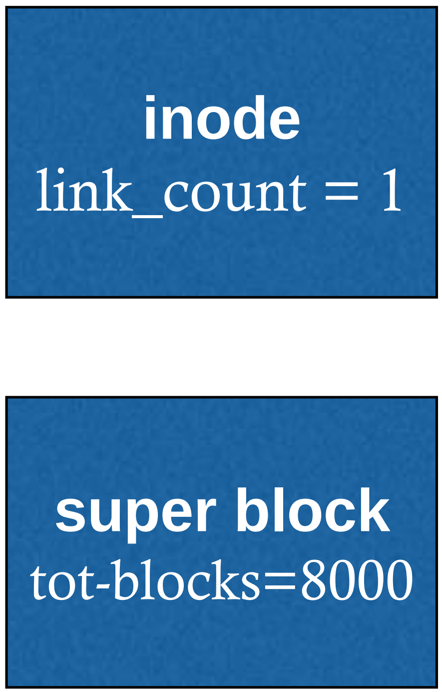
Simple, but is this correct?

Example 5: Bad Pointer



How to fix to restore consistency?

Example 5: Bad Pointer

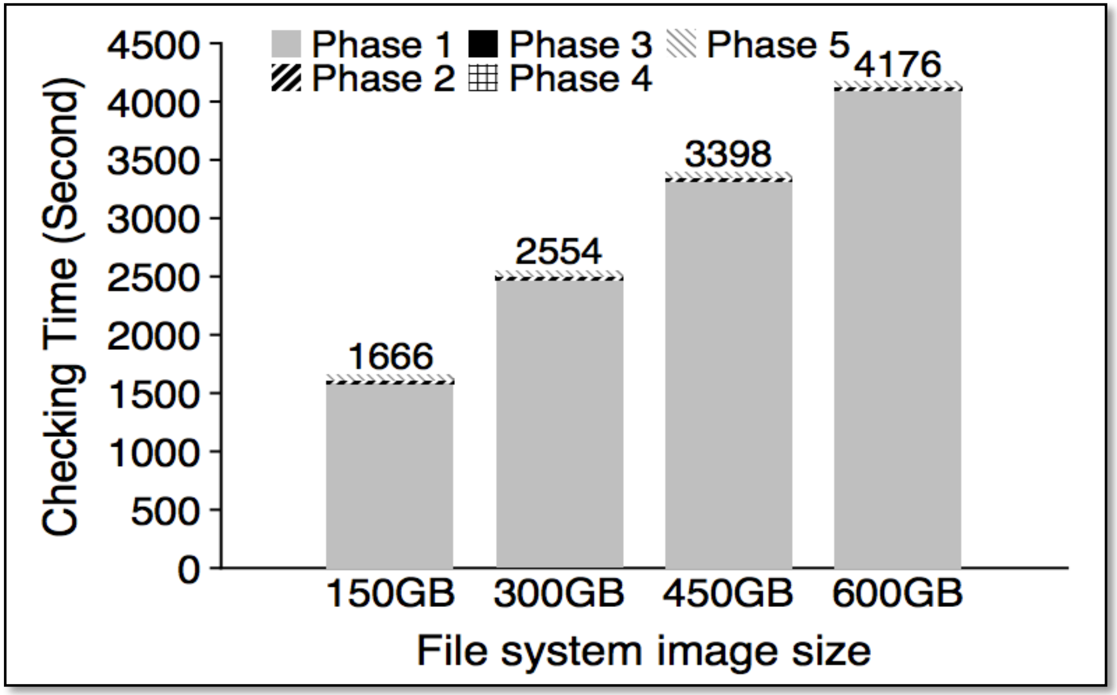


Simple, but is this correct?

Problems with FSCK

- Problem 1: functionality
  - Not always obvious how to fix file system image
  - Don’t know “correct” state, just consistent one
  - Easy way to get consistency: reformat disk!
- 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**

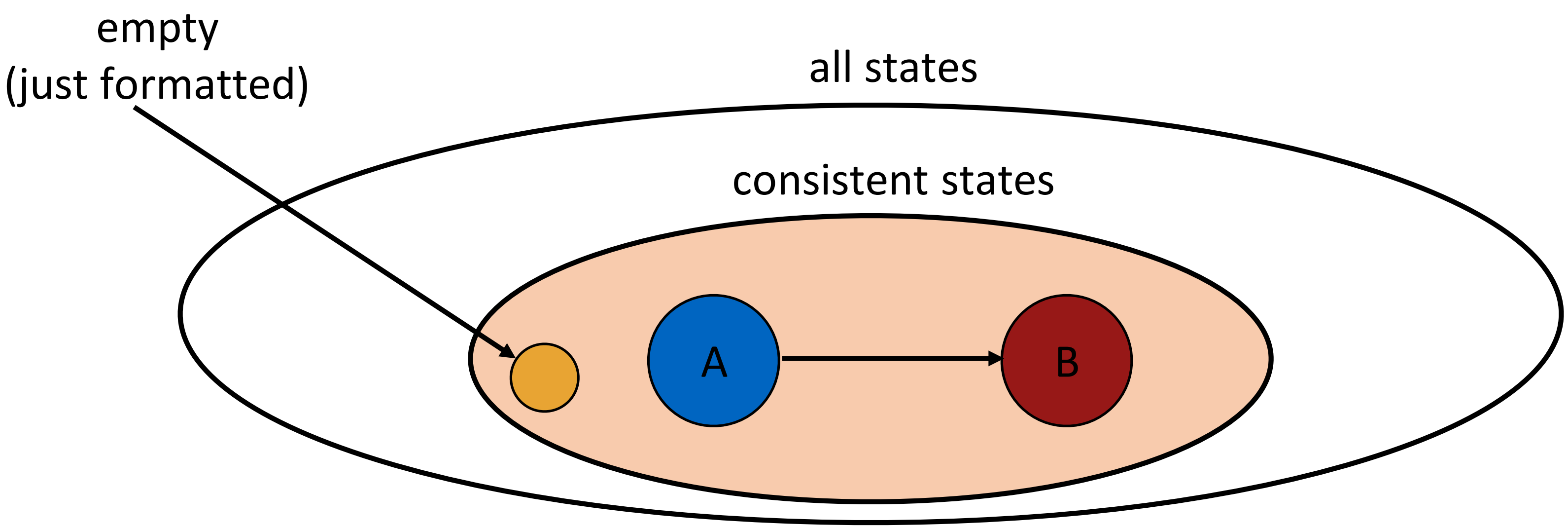
- Strategy: achieve **atomicity** when there are multiple disk updates

- Definition of <u>atomicity for concurrency</u>
  - Operations in critical sections are not interrupted by operations on related critical sections

- Definition of <u>atomicity for persistence</u>
  - 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



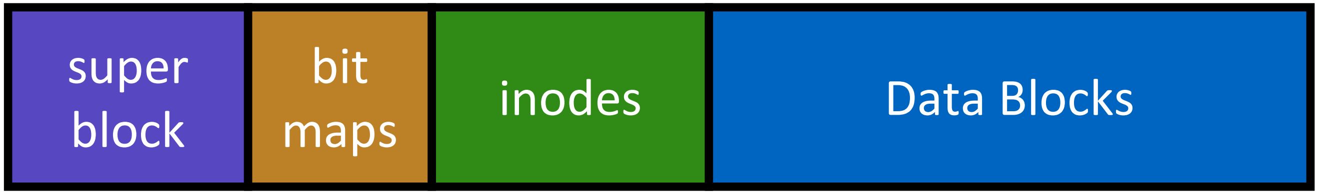
FSCK gives consistency.
Atomicity gives A or B.

Journaling Strategy

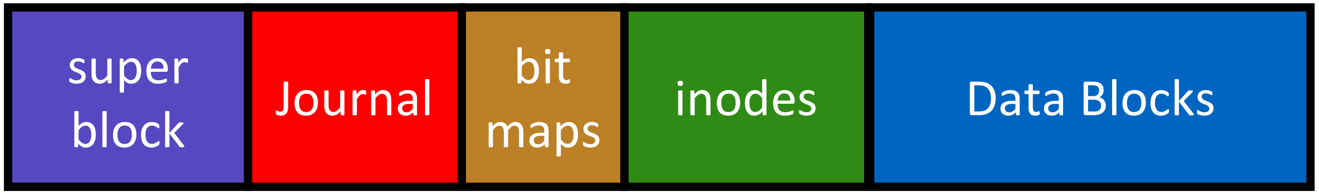
- Log all disk changes in a journal **before** writing them to file system proper

- Journal itself is a “temporary” persistent space on disk
  - Could be the same disk as FS or a different one (for added reliability)

Disk Layout
w/o Journal

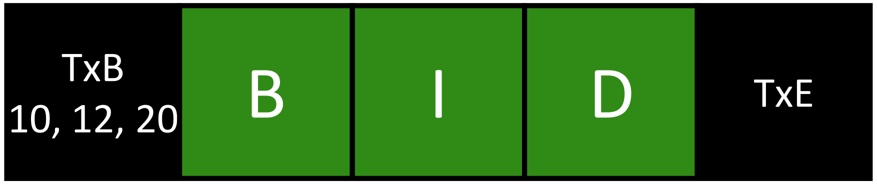


Disk Layout
with Journal

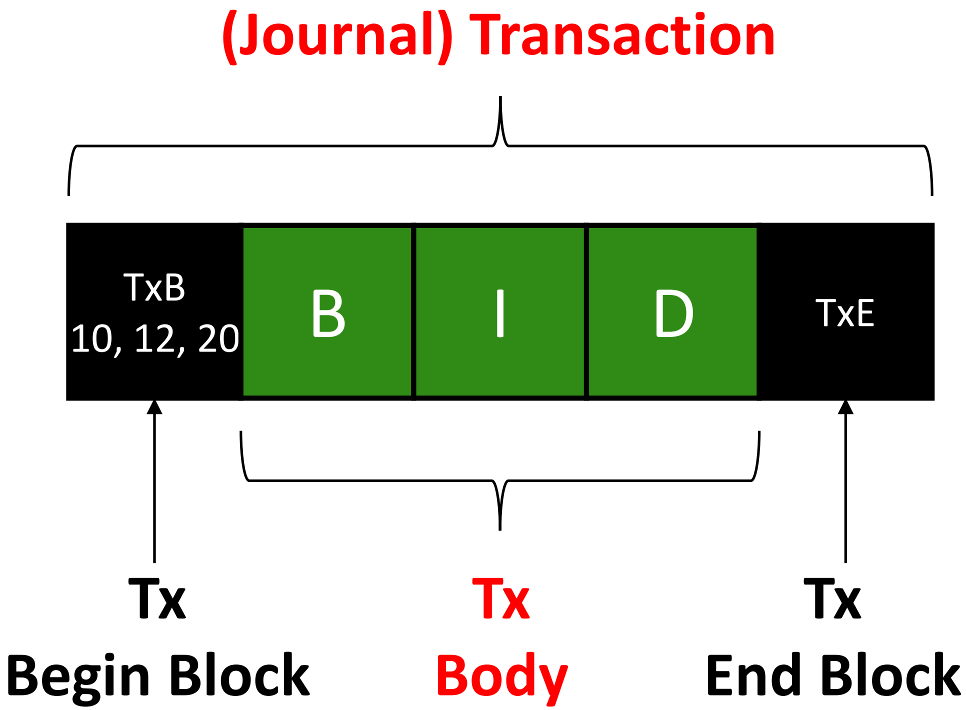


How Journaling Works

- Consider our 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



Journaling Terminology



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 (1)

- 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 <u>during journal write</u>
  - Ignore the half-written transaction during recovery
  - Crash happened during journal write → no checkpointing took place → FS blocks are not changed

Using Journal for Crash Recovery (2)

- If a crash happens <u>after journal write but before (or during) checkpointing</u>
  - 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 (1)

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 (2)

- 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
- When is it safe to do so?
  - 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 <u>not-checkpointed transactions</u>
  - 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 Optimizations

- Journaling has two major sources of overhead
  1) It more than doubles the number of disk writes
     - Every block first written to journal, then to FS
     - Also, there are TxB and TxE to write
  2) It enforces a lot of ordering between disk writes
     - TxB, Tx Body → TxE
     - TxE → Checkpointing
- Interlude: Why is it bad to enforce ordering?
  - It reduces the effectiveness of disk scheduling algorithms
- How can we reduce these overheads?

Optimization 1: Batching Updates

- Instead of logging updates of every system call separately, merge many operations into one big transaction
  - E.g., start a new transaction every 5 seconds; during the current 5-sec interval all disk changes go into the same Tx
  - You'll still have atomicity, so no inconsistency problems
- Benefit?
  - Fewer write orderings
  - Fewer TxB and TxE blocks
- Drawback?
  - On a crash, might lose more operations

Optimization 2: Journal Metadata Only

- So far, we journaled both metadata changes (bitmaps, inodes, etc.) as well as data changes (file data blocks)
- Structural consistency of FS only requires atomicity of metadata operations
- On the other hand, most of Tx Body is file data (typically)
- So, what if we just do metadata journaling?
  - Will reduce Tx size significantly

Journaling Modes (1)

- ***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 <u>before</u> metadata is in journal
    - Why not after?
    - Because inode can point to garbage data if crash
  - Faster than full data, but constrains write orderings

Journaling Modes (2)

- ***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

Conclusion

- Most modern file systems use journals
  - ordered-mode for metadata is popular
- FSCK is still useful for weird cases
  - Bit flips
  - FS bugs
- Some advanced file systems don’t use journals, but only do writes on unused blocks (never overwrite blocks)
  - Copy-on-Write file systems (e.g., ZFS)
  - Log-structure file systems (e.g., LFS)