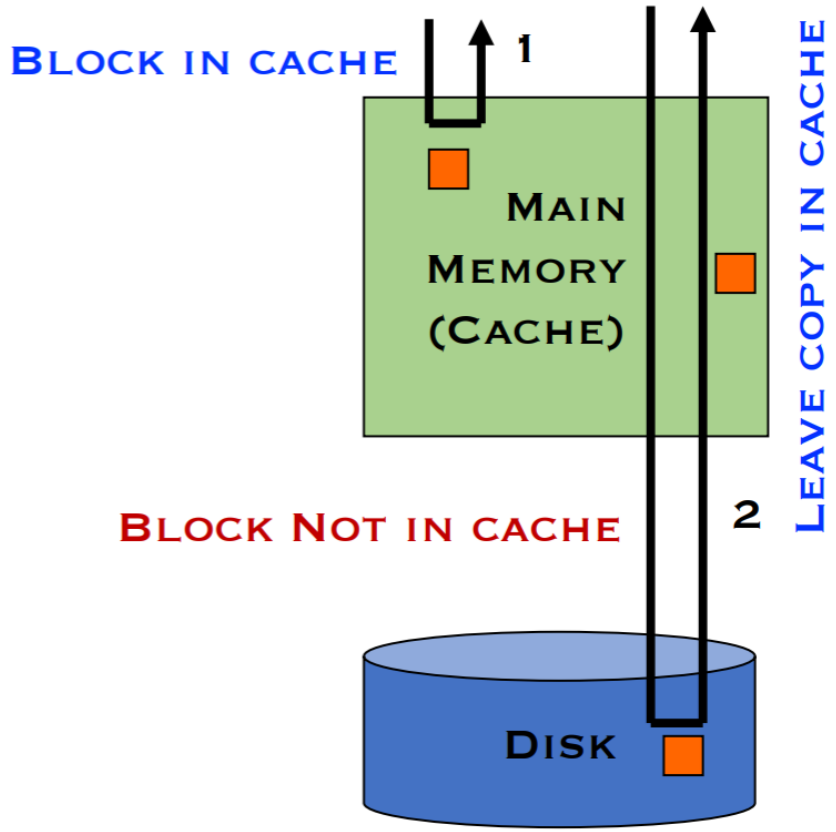
Crash Consistency

Instructor: Youngjin Kwon

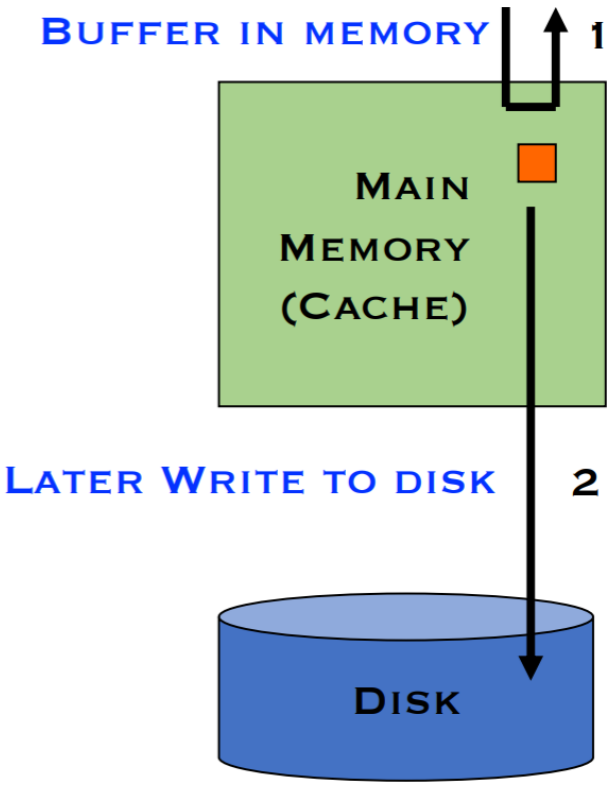
Review: Read I/O path

- read() from file
  - Check if block is in cache
  - If so, return block to user [1 in figure]
  - If not, read from disk, insert into cache, return to user [2]

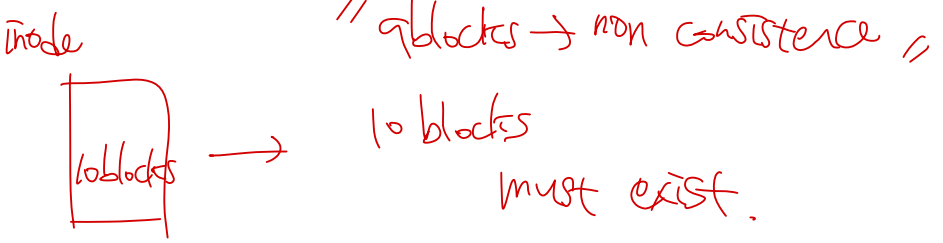


Review: Write I/O path

- write() to file
  - Write is buffered in memory ("write behind") [1]
  - Sometime later, OS decides to write to disk [2]
    - Periodic flush or fsync call
- Why delay writes?
  - Implications for performance
  - Implications for reliability (crash consistency)



New requirement: Crash consistency



- Definition
  - Atomically update file system from one consistent state to another

- What does the consistent state mean?

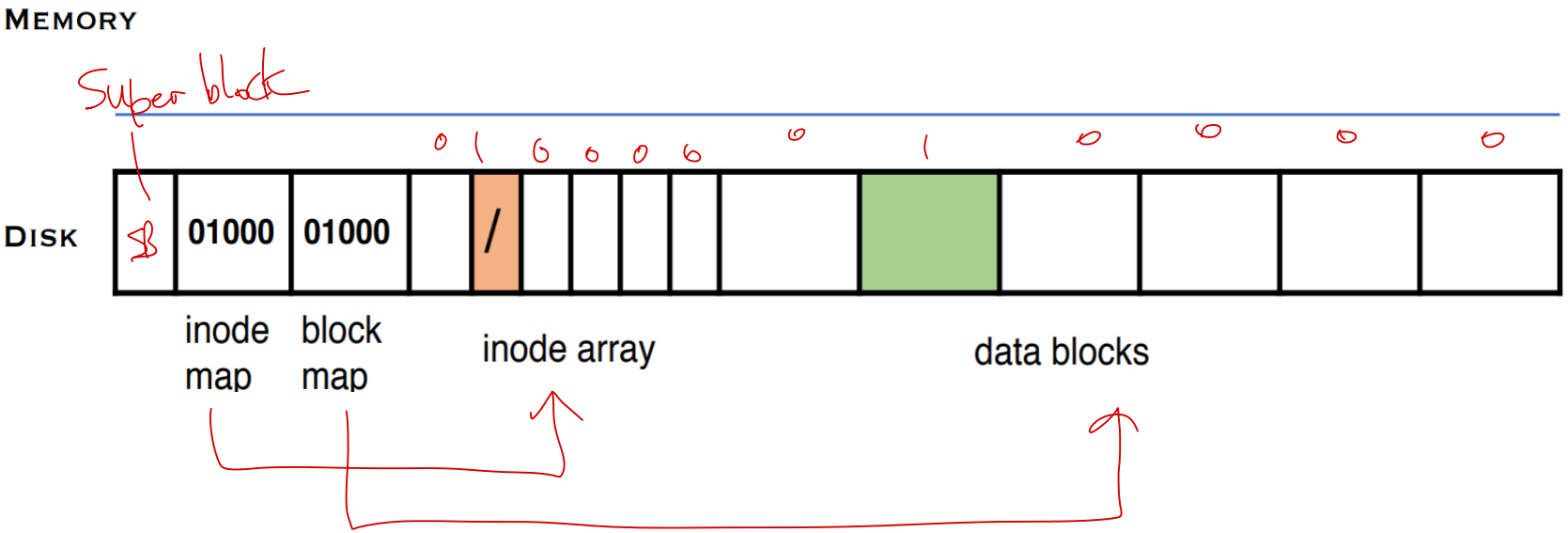
  Metadata (e.g., bitmaps, inode, directory) states and metadata/data states are consistent

Metadata state = Data state

ex. inode, directory, block allocation bitmap, inode table

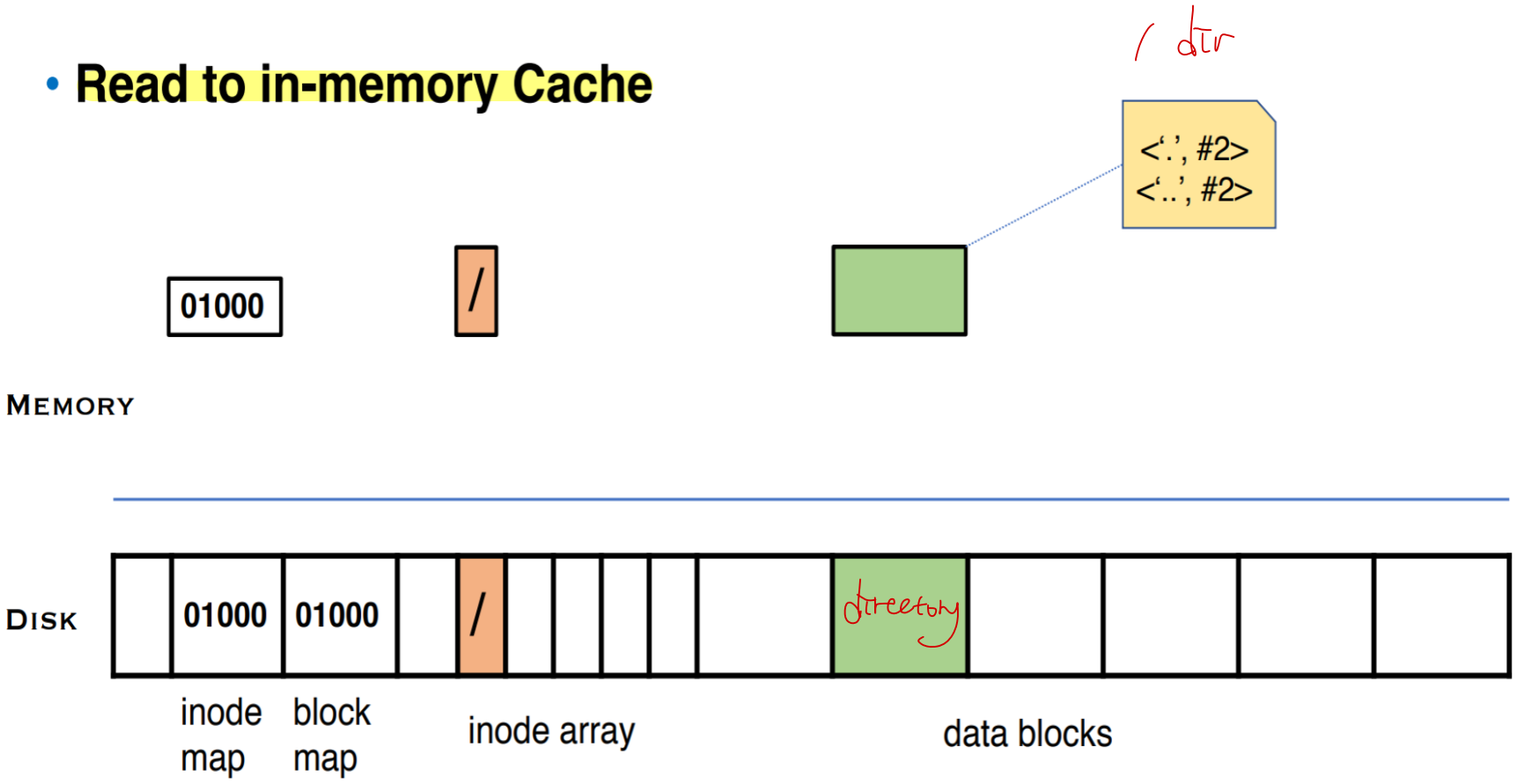
Example: File Creation (step 1)

- **Initial state**

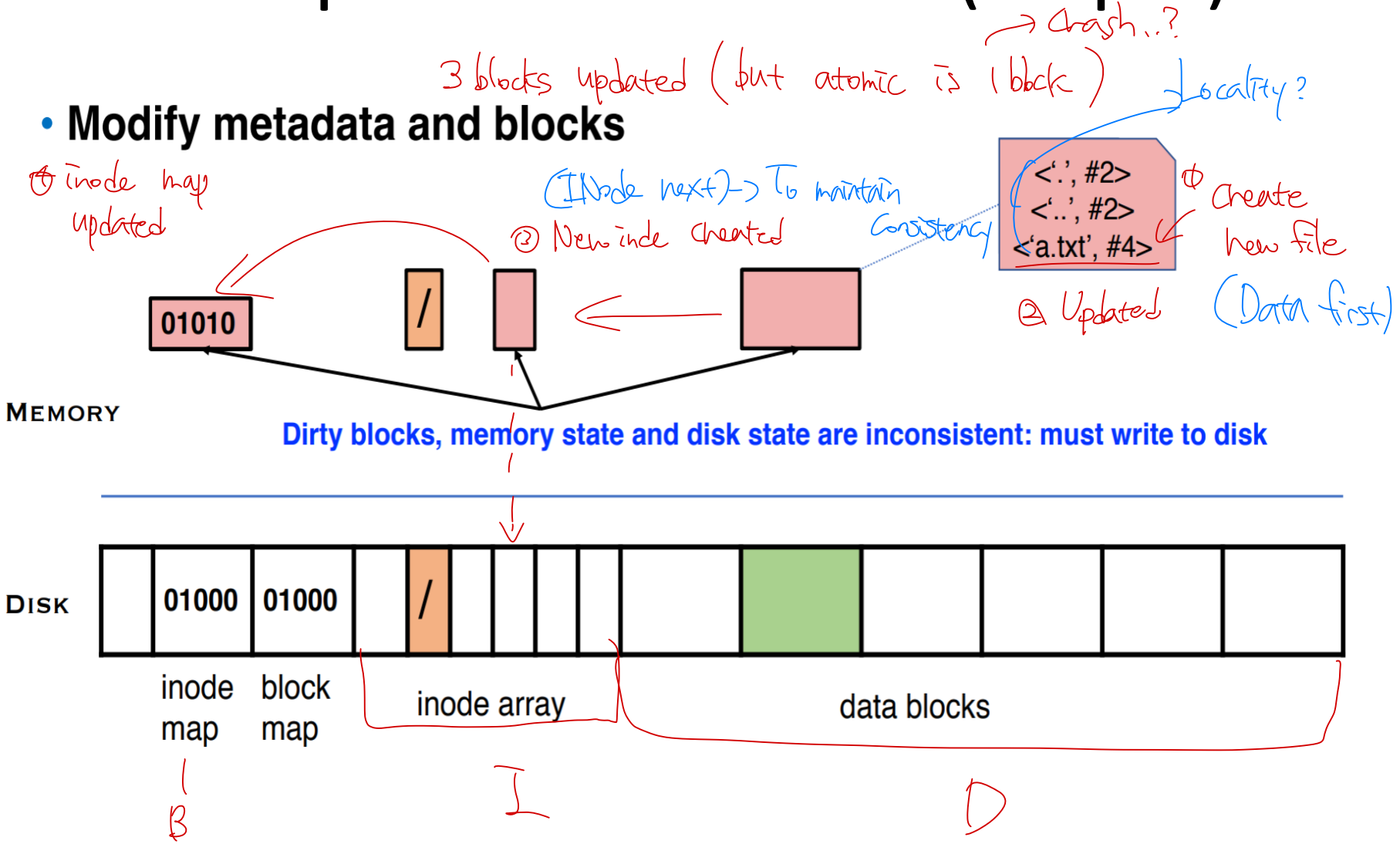


Example: File Creation (step 2)

- **Read to in-memory Cache**



Example: File Creation (step 3)



What if crash happens?

- Disk & SSD: atomically write one sector (=block)
  - Atomic: if crash, a sector is either completely written, or none of this sector is written

- An FS operation (e.g., file creation) may modify multiple sectors!

- **A Crash → FS partially update file system states → Lead to inconsistent state!**

→ Atomically write multiple sectors

Possible crash cases

- File creation requires updating three blocks
  - Inode bitmap (B)
  - Inode for new file (I)
  - Parent directory data block (D)

on disk

DRAM

- Old and new contents of the blocks
  - B = 01000 B' = 01010
  - I = free I' = allocated, initialized
  - D = {<".",2> <"..",2>} D' = D + {<"a.txt",4>} + new data blocks for a.txt

Possible crash cases

- Crash scenarios?

- B I D
- B' I D
- B I' D
- B I D'
- B' I' D
- B' I D'
- B I' D'
- B' I' D'

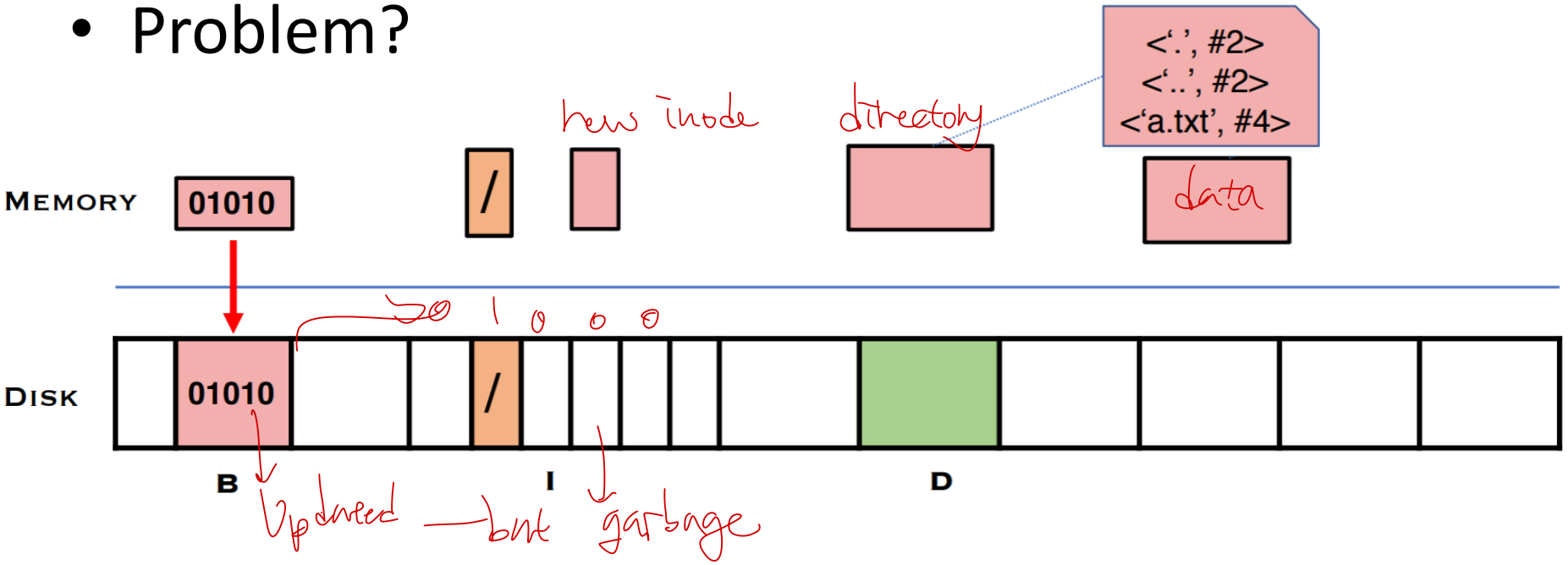
Any subset can be written because **OS can reorder operations!**

Analysis 1: Write bitmap first

- Write ordering: B → (I or D)
  - But crash right after writing B

B' I D

- Problem?



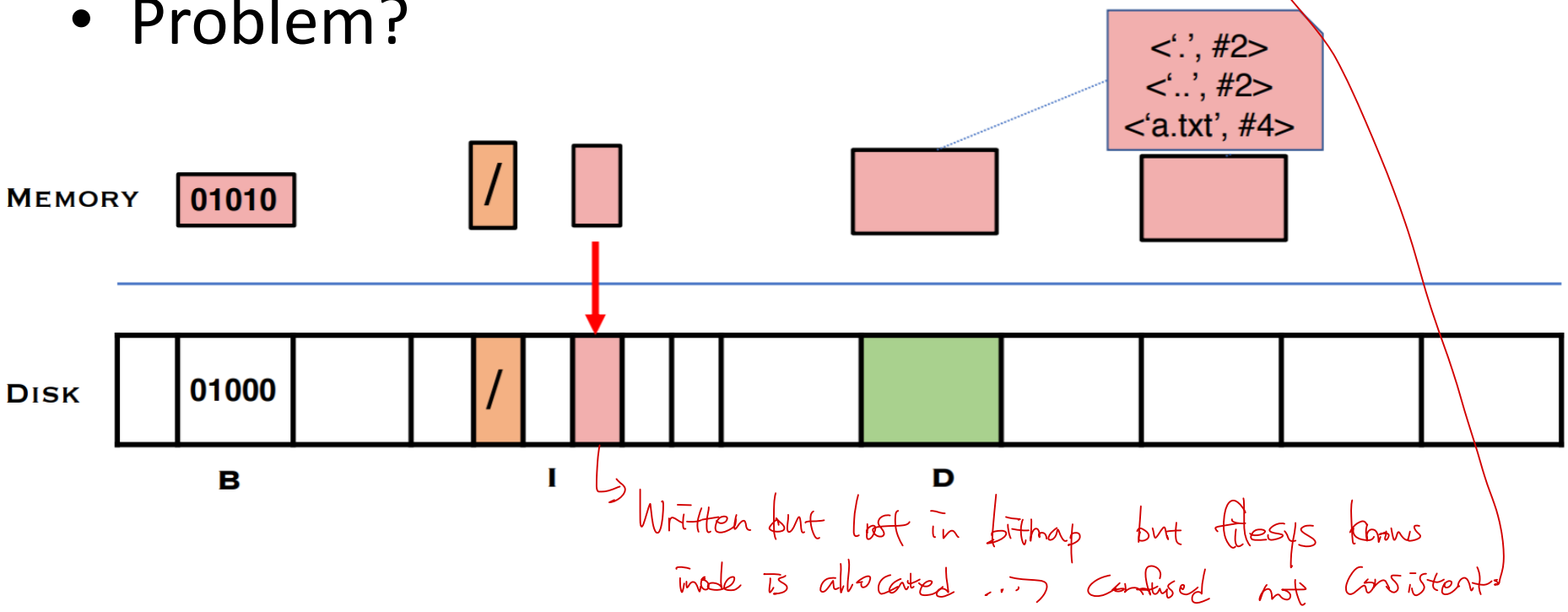
Analysis 2: Write inode first

- Write ordering: I → (B or D)
  - But crash after writing I

B I' D

Need extra action to check

- Problem?



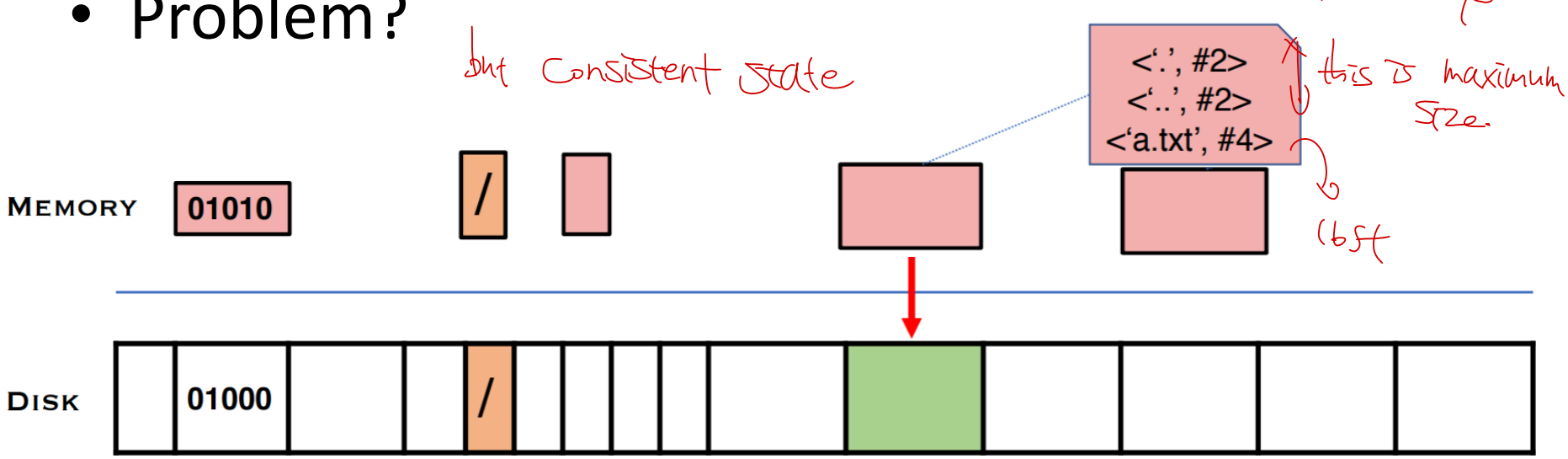
Analysis 3: Write data first

- Write ordering: D → (I or B)
  - But crash after writing D

B I D'

- Problem?

Data is inaccessible but consistent state

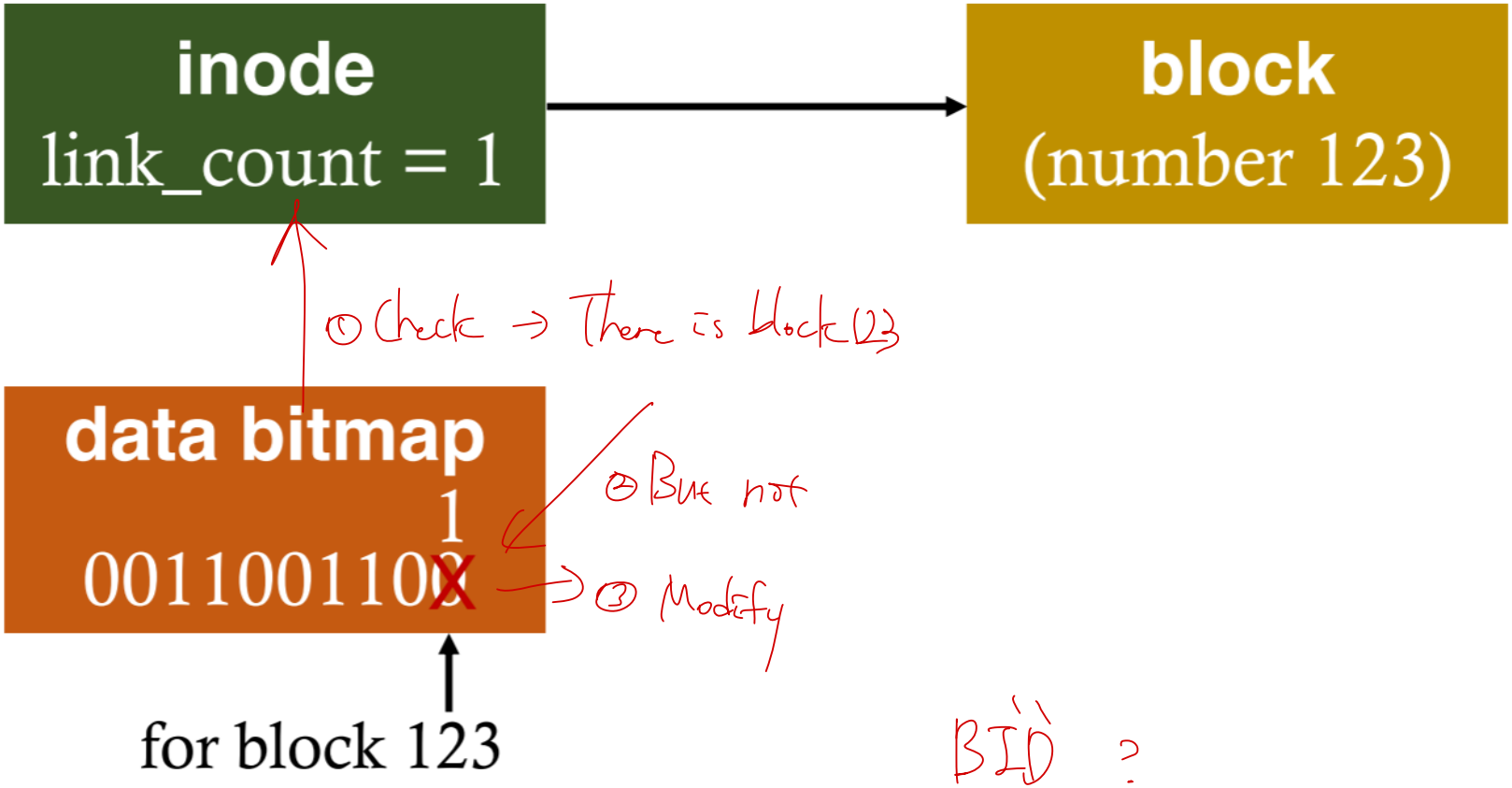


Traditional solution: FSCK

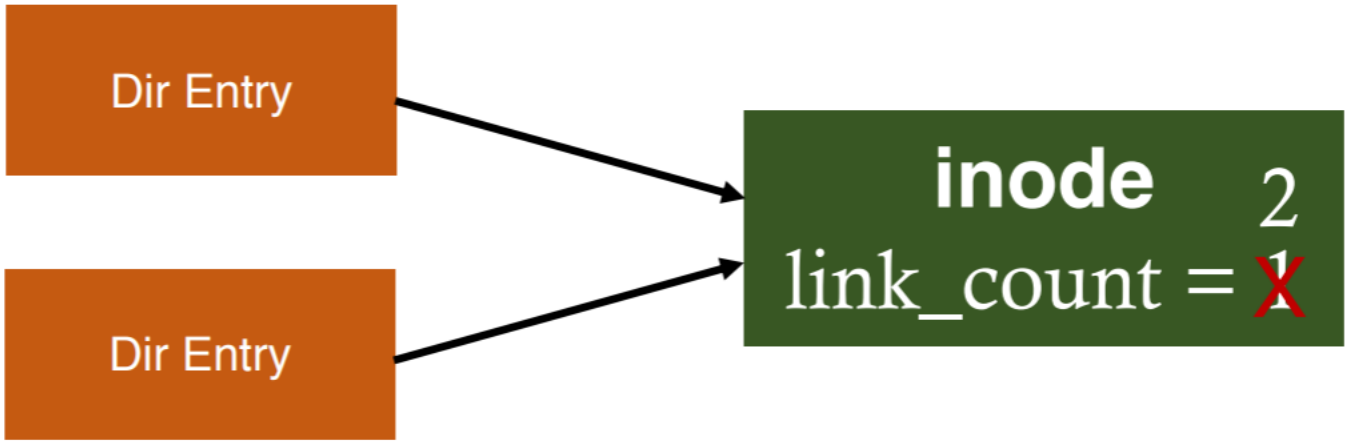
- FSCK: "file system checker"

- When system boots:
  - Make multiple passes over file system, looking for inconsistencies
    - e.g., inode pointers and bitmaps, directory entries and inode reference counts
  - "Try to fix" automatically

FSCK example 1



FSCK Example 2



Traditional Solution: FSCK

- FSCK: “file system checker”
- When system boots:
  - Make multiple passes over file system, looking for inconsistencies
  - Try to fix automatically or punt to admin
    - Example: B’ I D, B I’ D → revert I' to I
- Problem:
  - Cannot fix all crash scenarios
    - Can B’ I D’ be fixed? → No fix possible
  - Performance
    - Sometimes takes hours to run on large disk volumes
    - Does fsck have to run upon every reboot?

↳ revert B' to B

B I D' → No fix possible but consistent

B' I D → No fix possible

B' I D' → Update to B'?

→ only after crash (Crash flag)

Better Solution: Journaling

- Idea: Write "intent" down to disk before updating file system
  - Intent: Describes what you are about to do
  - Called the "Write Ahead Logging" or "journal"
  - Originated from database community

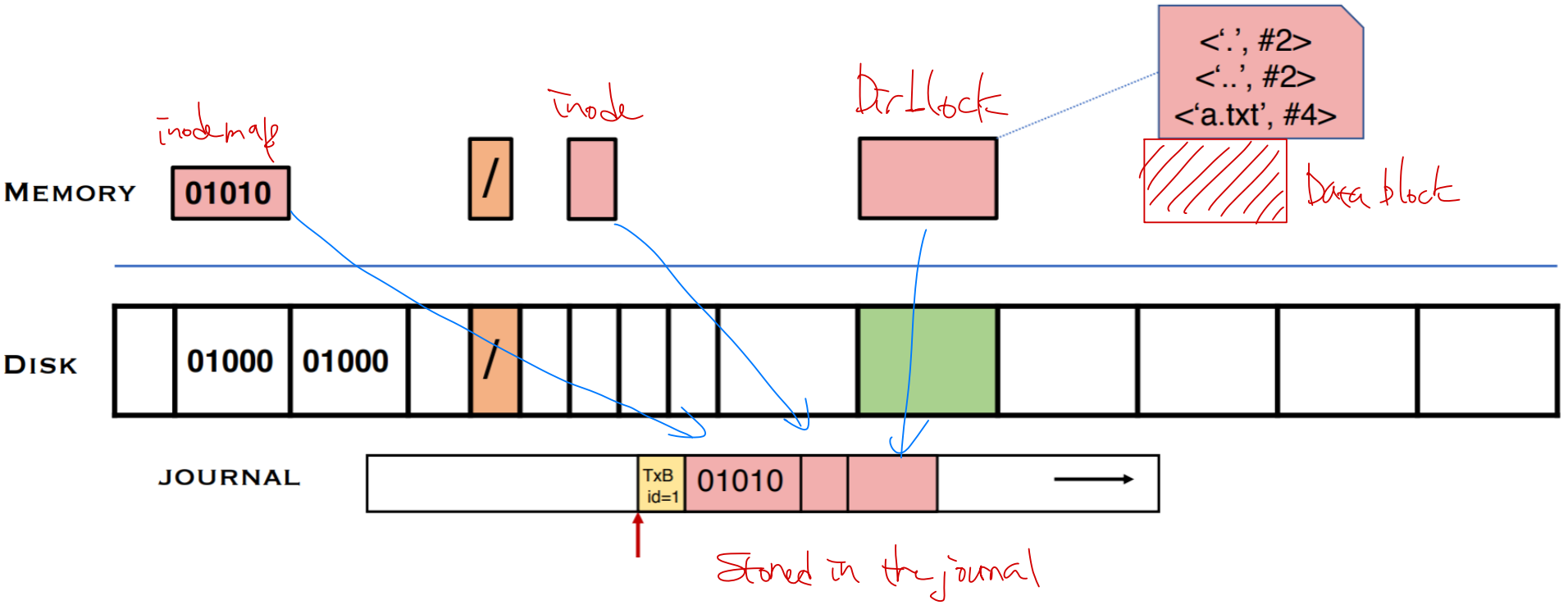
- When crash occurs, look through log to see what was going on
  - Use contents of log to fix file system structures
    - Crash before "intent" is written (not committed) → no-op
    - Crash after "intent" is written (committed) → redo op
    - The process is called "recovery"

How to represent the “intent”?

- Physical journaling: write real block contents of the update to log
  - Four totally ordered steps
    - Commit dirty blocks to journal as one transaction (TxBegin, I, B, D blocks) → Atomic
    - Write commit record (TxEnd)
    - Copy dirty blocks to real file system (checkpointing)
    - Reclaim the journal space for the transaction
- Logical journaling: write logical record of the operation to log
  - “Add entry F to directory data block D”
  - Complex to implement
  - May be faster and save disk space

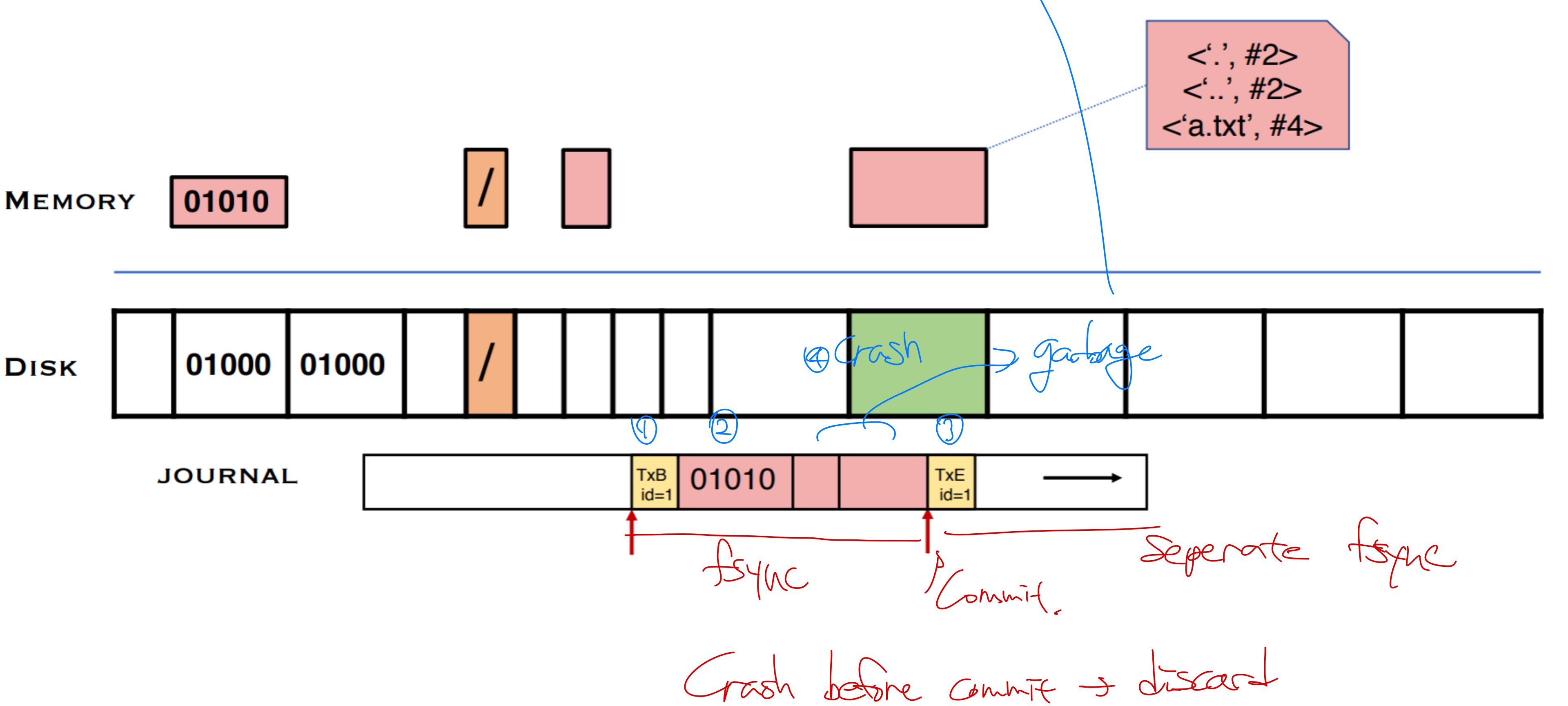
Physical journaling (Step 1)

Write metadata and data blocks to journal



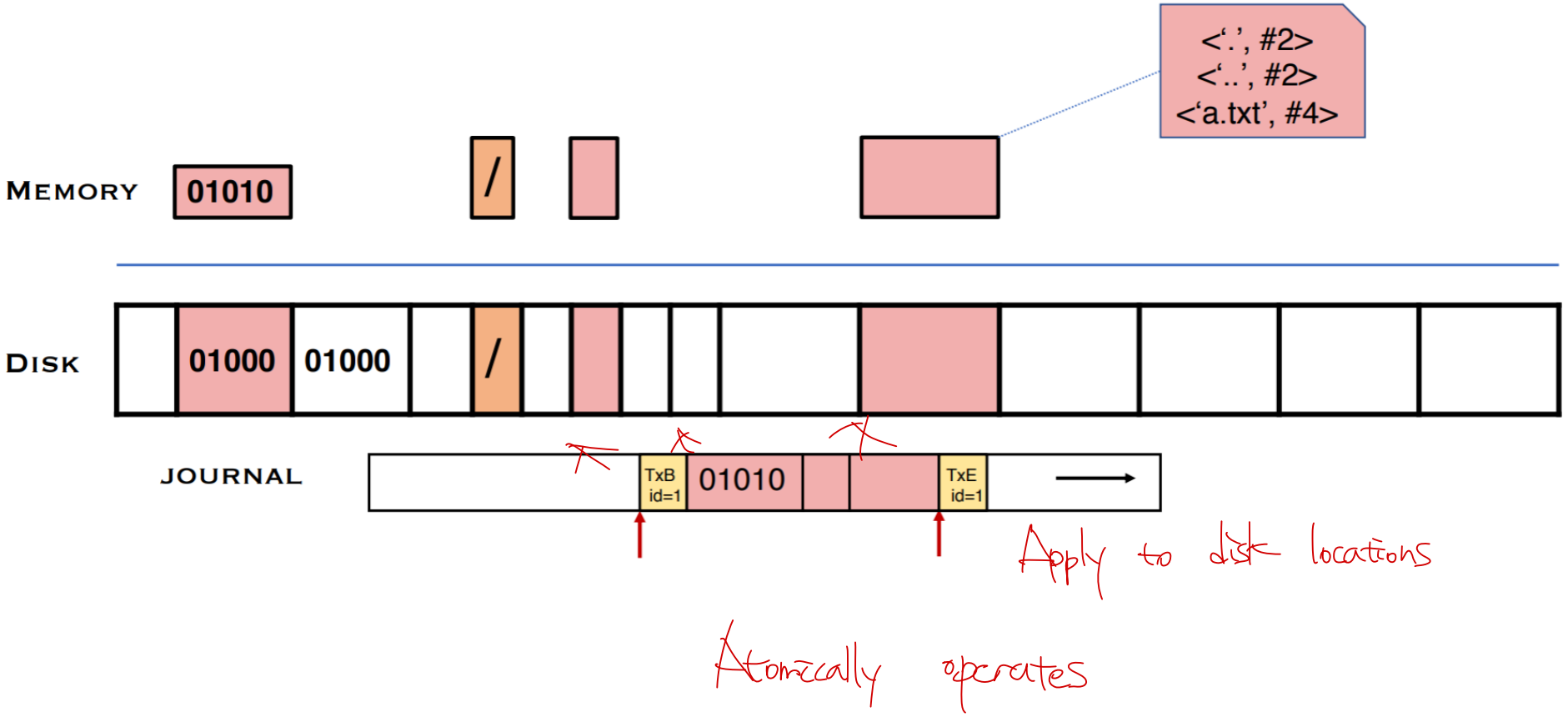
Physical journaling (Step 2)

Write commit record



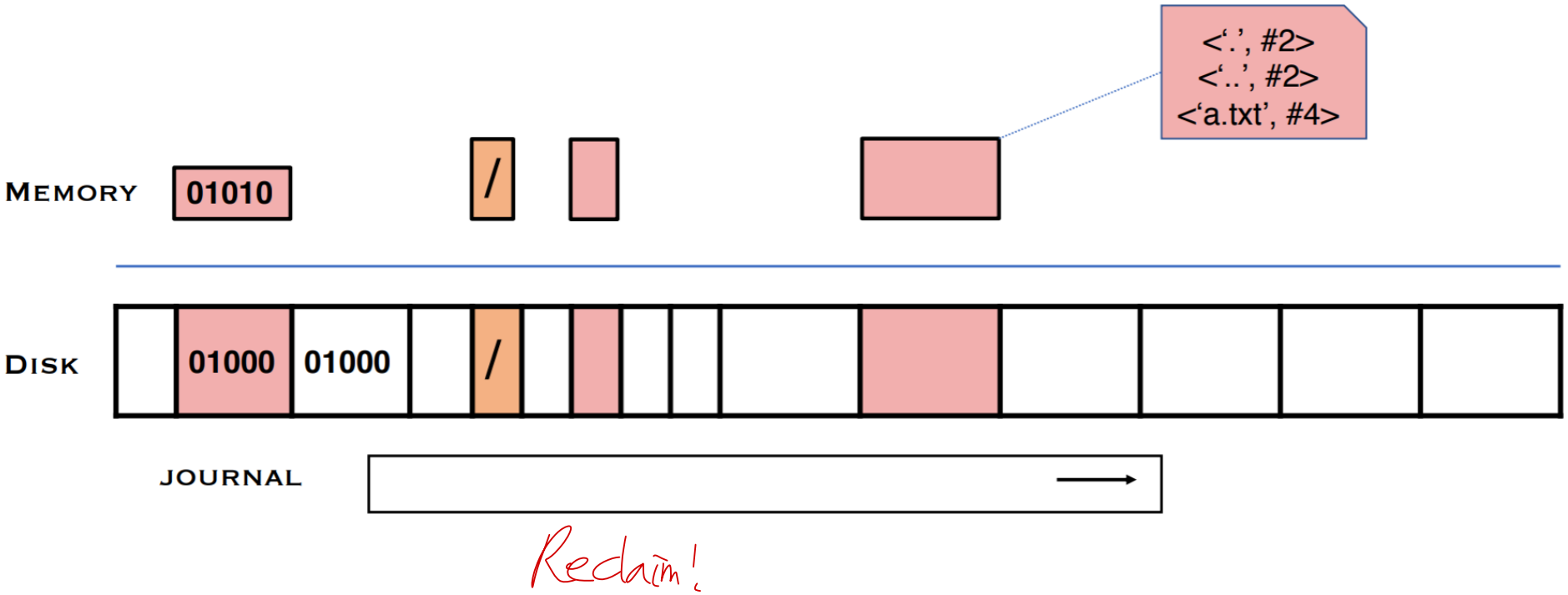
Physical journaling (Step 3)

Copy dirty blocks to in-place location (checkpointing)



Physical journaling (Step 4)

Reclaim journal space



What if there is a crash?

- Recovery: Go through log and "redo" operations that have been successfully committed to log

- What if …
  - TxBegin but not TxEnd in log? → Discard
  - TxBegin through TxEnd are in log, but I, B, and D have not yet been checkpointed?
    - How could this happen? → ?
    - How about merging step 2 and step 1?

Write orders in Journaling

- **<** (left happened first) or **>** (right happened first)
  - Journal write (→) FS write (checkpointing)
  - Journal commit (←) Journal write
  - Journal clean (←) FS write

happen first

Ext3 Journaling modes

- Journaling has cost
  - Extra writes (twice?) and two seeks
    - ↳ Separate step 1 & 2 write + FS write (2 more writes probably...)
- Several journaling modes to balance consistency and performance

- Journaling mode:
  - Data journal: journal all writes including file data
    - Problem: expensive (IO bandwidth & latency) to journal data
  - Ordered mode: write file data to in-place location, then journal metadata
    - Problem: no guarantee of atomic updating multiple data blocks