CS 318 Principles of Operating Systems

Fall 2021

Lecture 19: File System Crash Consistency

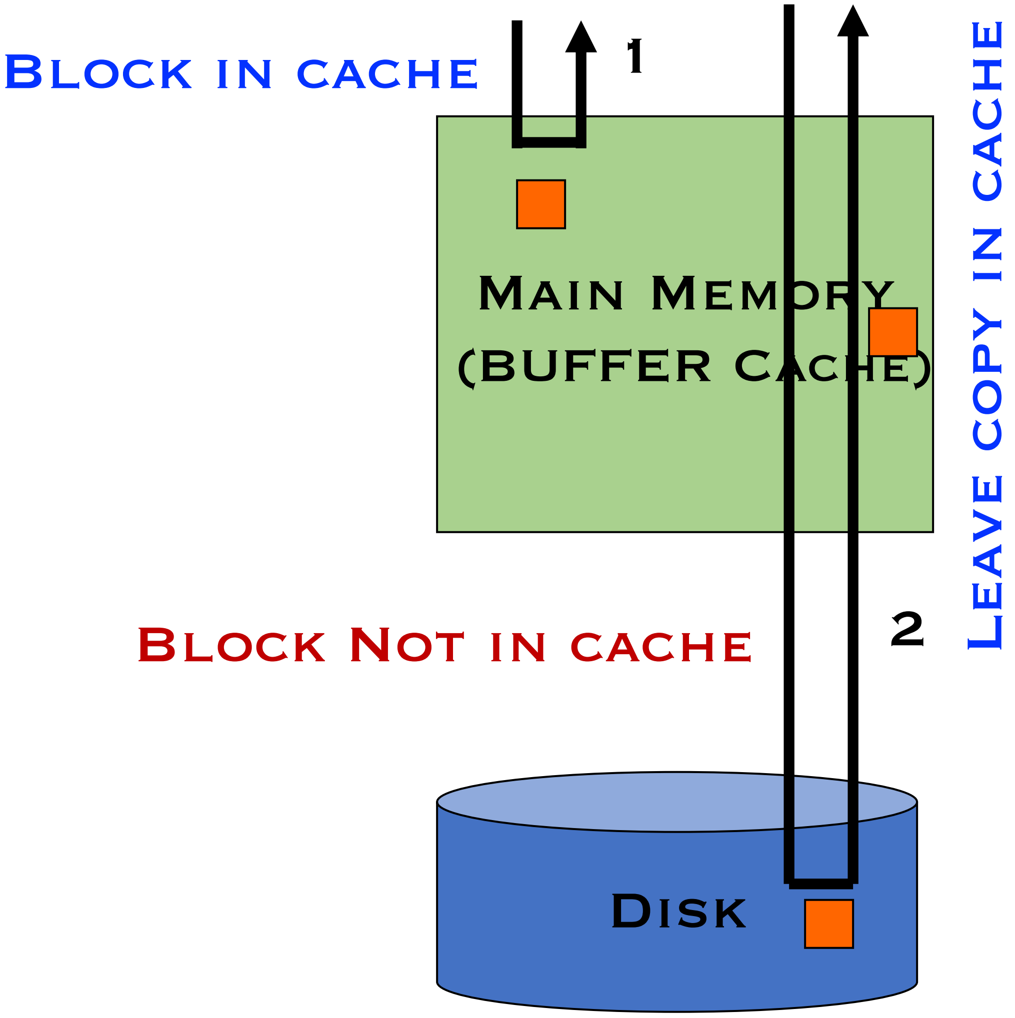
Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Review: File I/O Path (Reads)

`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]



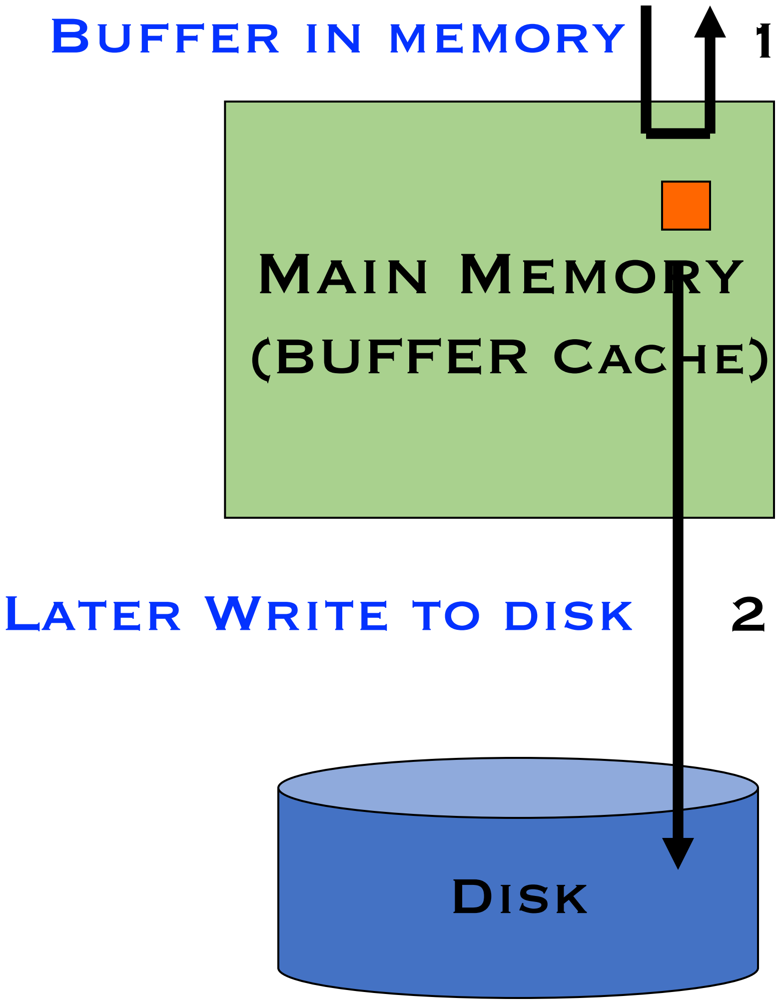
Review: File I/O Path (Writes)

`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



The Consistent Update Problem

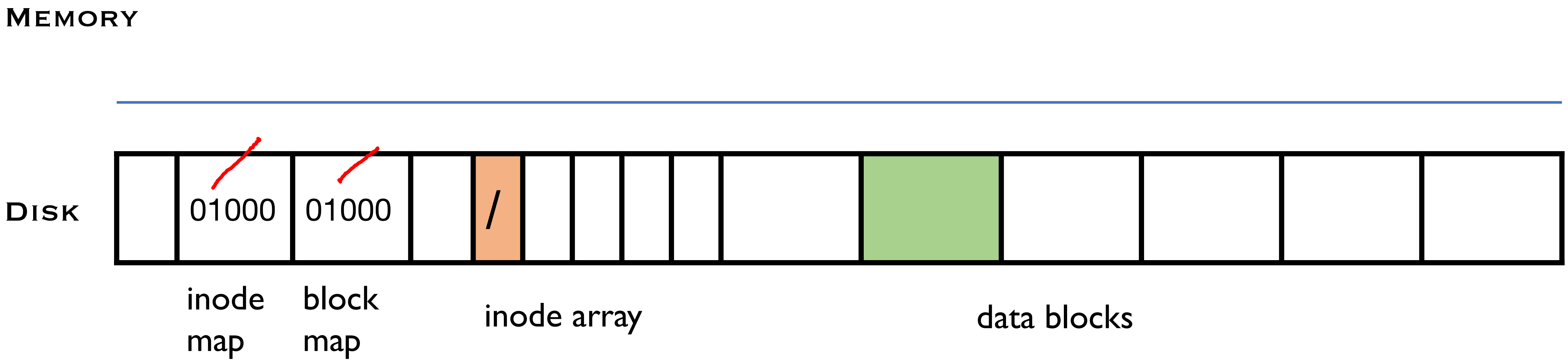
Goal: atomically update file system from one consistent state to another,

- What do we mean by consistent state?

Challenge: an update may require modifying several sectors, despite that the disk only provides atomic write of one sector at a time

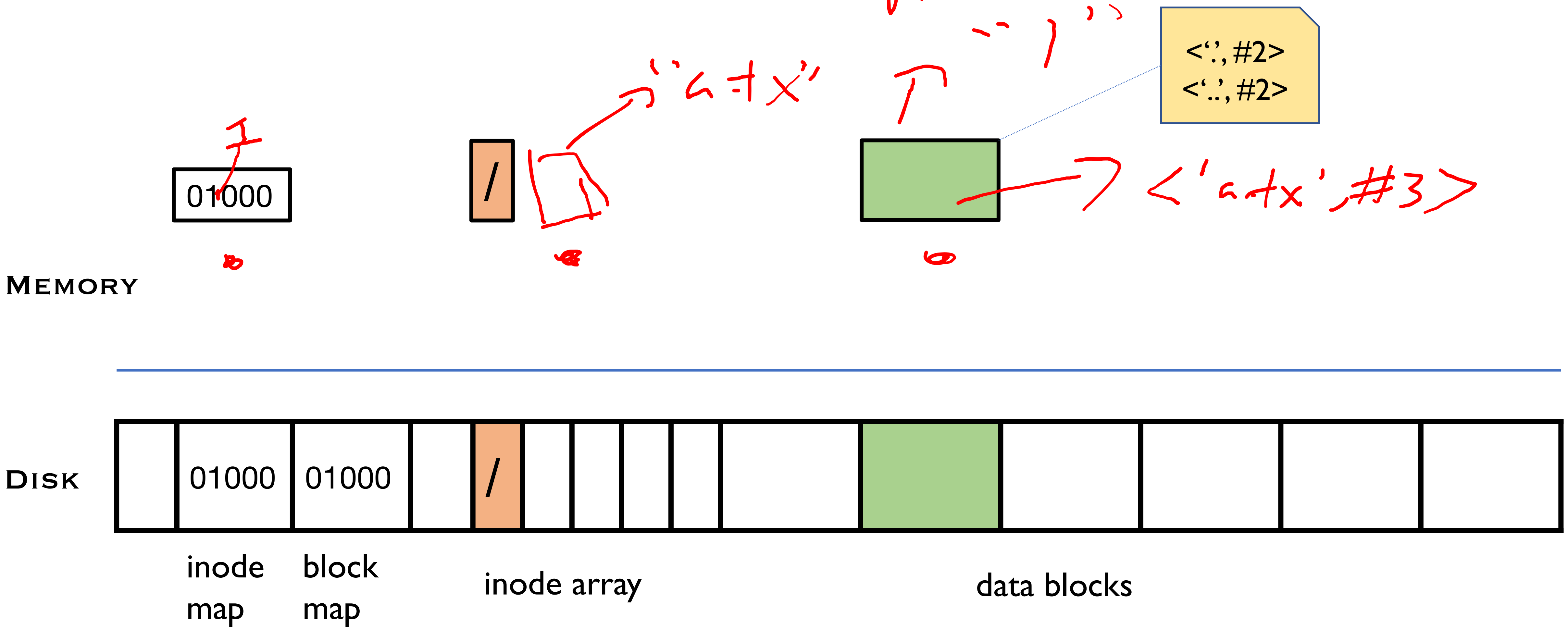
Example: File Creation of /a.txt

Initial state



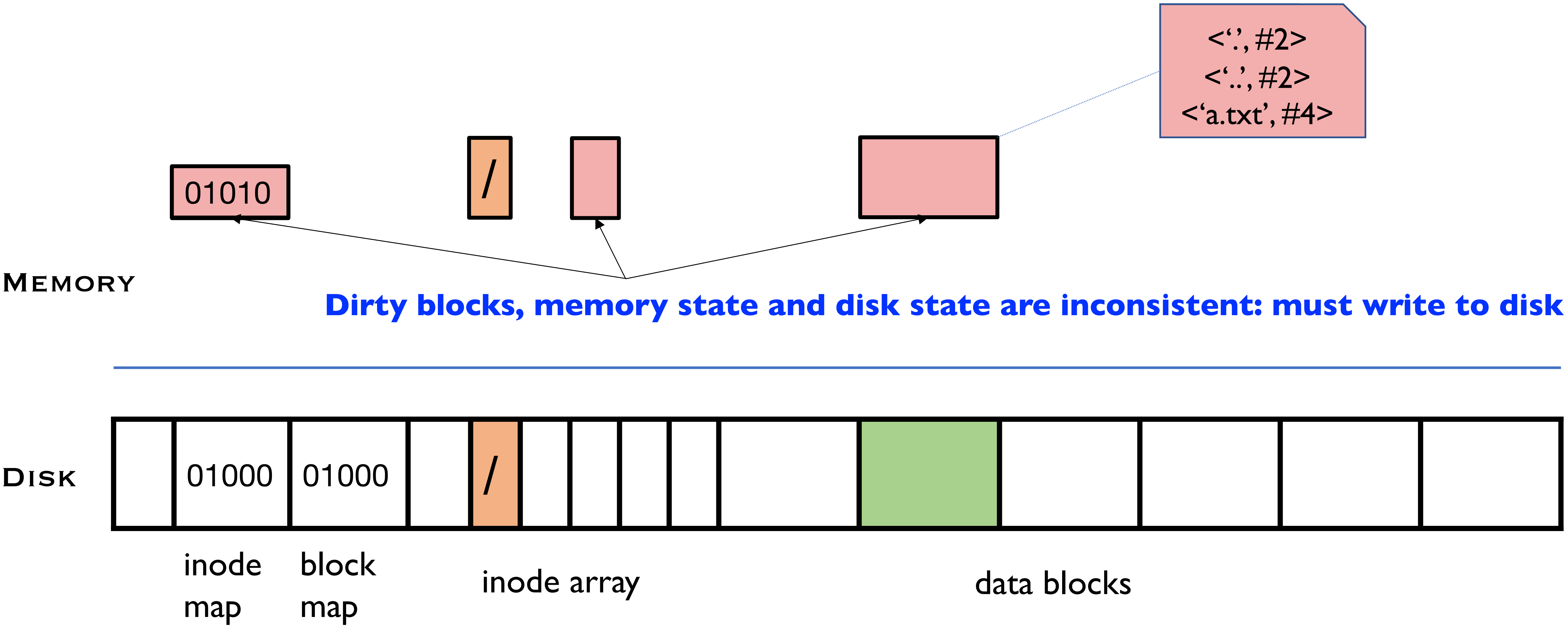
Example: File Creation of /a.txt

Read to in-memory Cache



Example: File Creation of /a.txt

Modify metadata and blocks



Crash?

Disk: atomically write one sector

- Atomic: if crash, a sector is either completely written, or none of this sector is written

An FS operation may modify multiple sectors

Crash → FS partially updated

Possible Crash Scenarios

File creation dirties three blocks

- inode bitmap (B)
- inode for new file (I)
- parent directory data block (D)

Old and new contents of the blocks

- B = 01000 B’ = 01010
- I = free I’ = allocated, initialized
- D = {} D’ = {<‘a.txt’, 4>}

Possible Crash Scenarios

Crash scenarios: any subset can be written

- B I D
- B’ I D
- B I’ D
- B I D’
- B’ I’ D
- B’ I D’
- B I’ D’
- B’ I’ D’

The General Problem

Writes: Have to update disk with N writes

- Disk does only a single write atomically

Crashes: System may crash at arbitrary point

- Bad case: In the middle of an update sequence

Desire: To update on-disk structures atomically

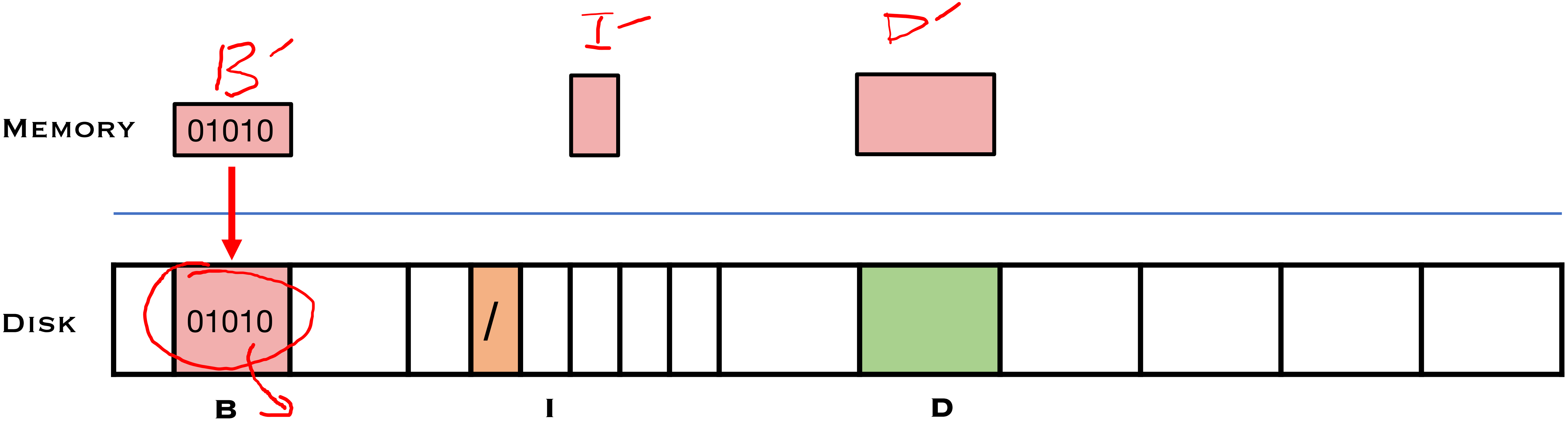
- Either all should happen or none

Example: Bitmap First

Write Ordering: Bitmap (B), Inode (I), Data (D)

- But CRASH after B has reached disk, before I or D

Result?

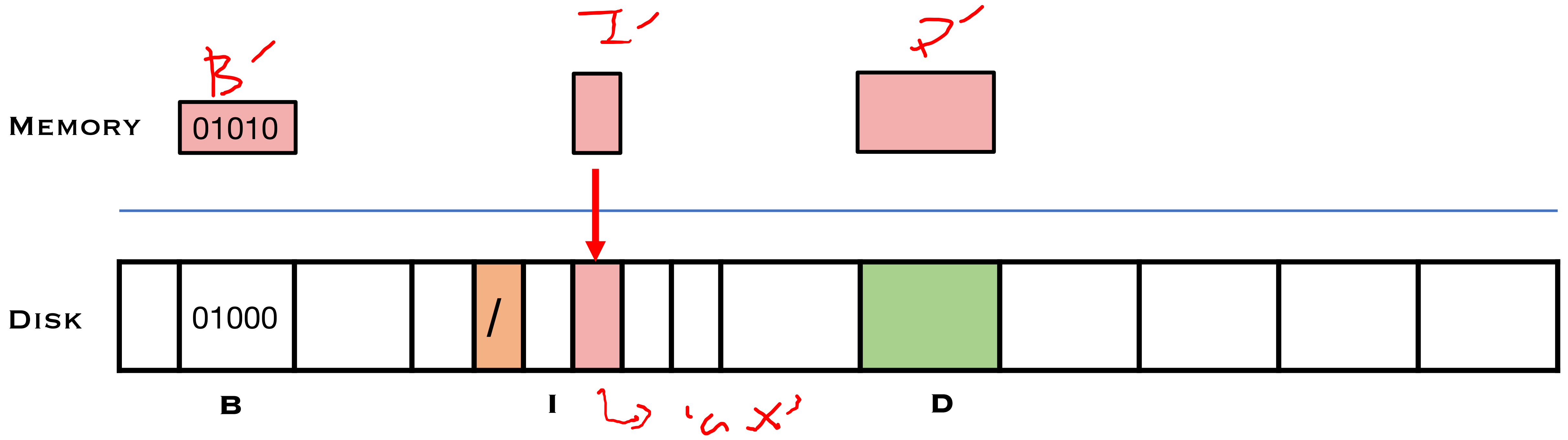


Example: Inode First

Write Ordering: Inode (I), Bitmap (B), Data (D)

- But CRASH after I has reached disk, before B or D

Result?

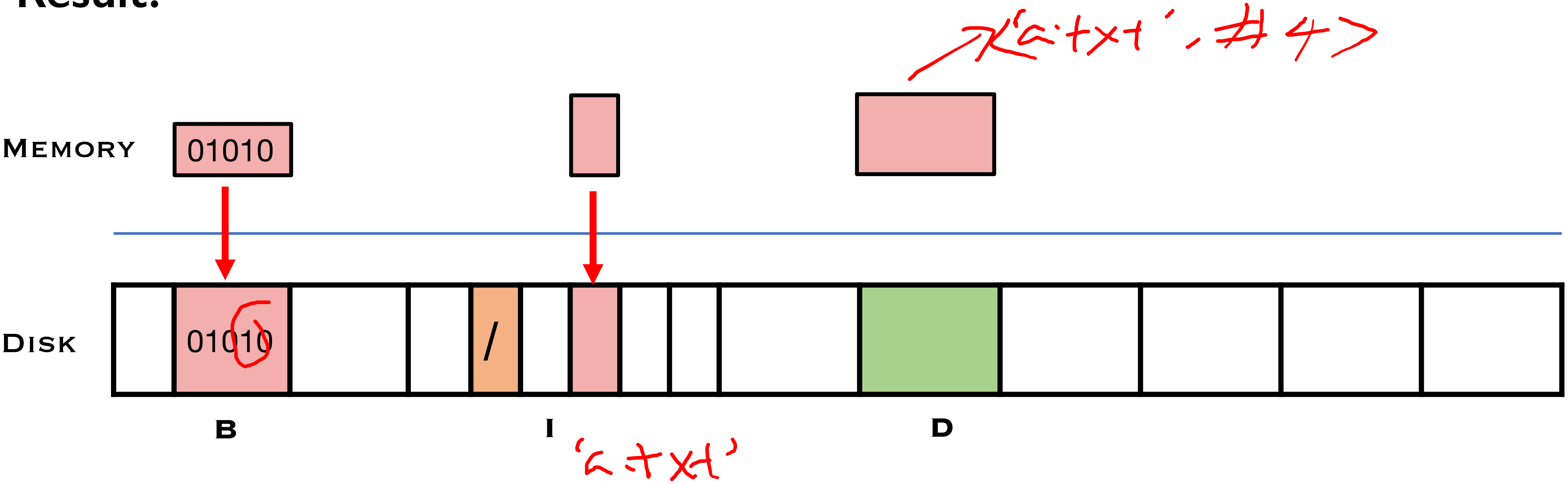


Example: Inode First

Write Ordering: Inode (I), Bitmap (B), Data (D)

- But CRASH after I AND B have reached disk, before D

Result?



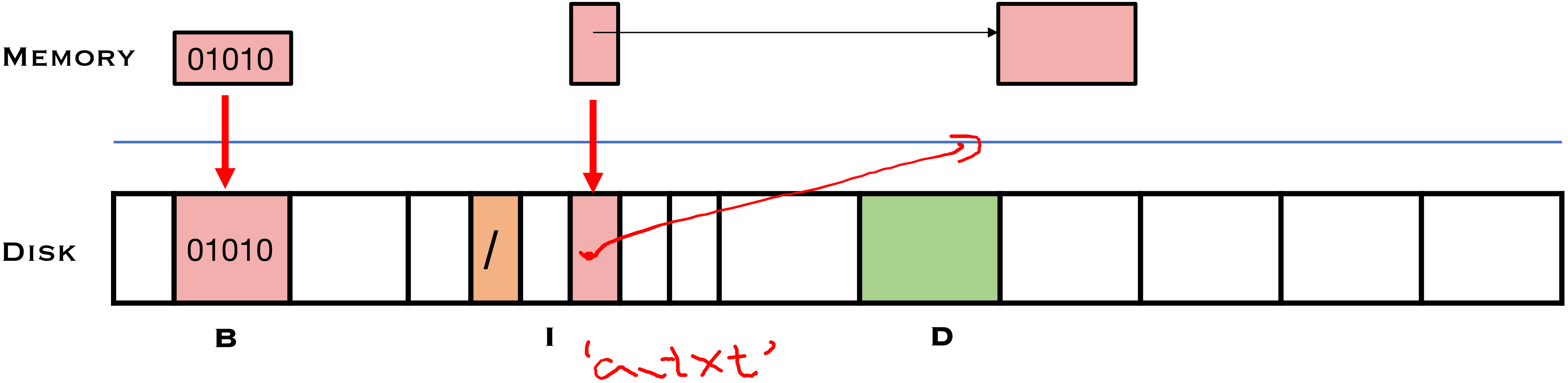
Example: Inode First

Write Ordering: Inode (I), Bitmap (B), Data (D)

- But CRASH after I AND B have reached disk, before D

Result?

- What if data block is a new block for the new file (i.e., create file with data)

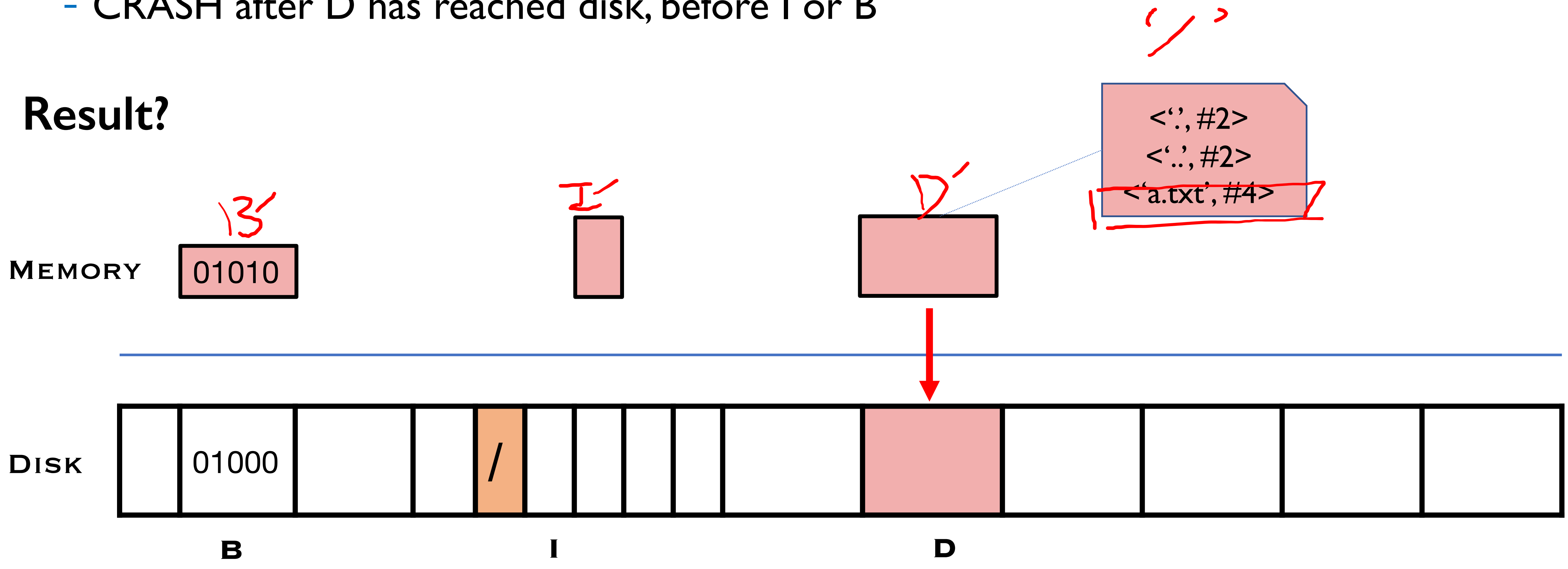


Example: Data First

Write Ordering: Data (D) , Bitmap (B), Inode (I)

- CRASH after D has reached disk, before I or B

Result?



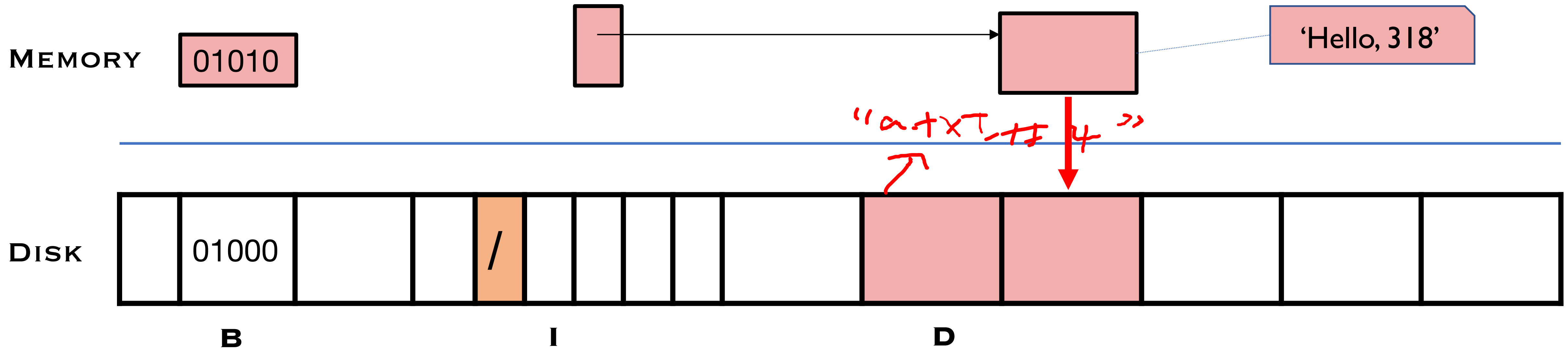
Example: Data First

Write Ordering: Data (D) , Bitmap (B), Inode (I)

- CRASH after D has reached disk, before I or B

Result?

- What if data block is a new block for the new file (i.e., create file with data)



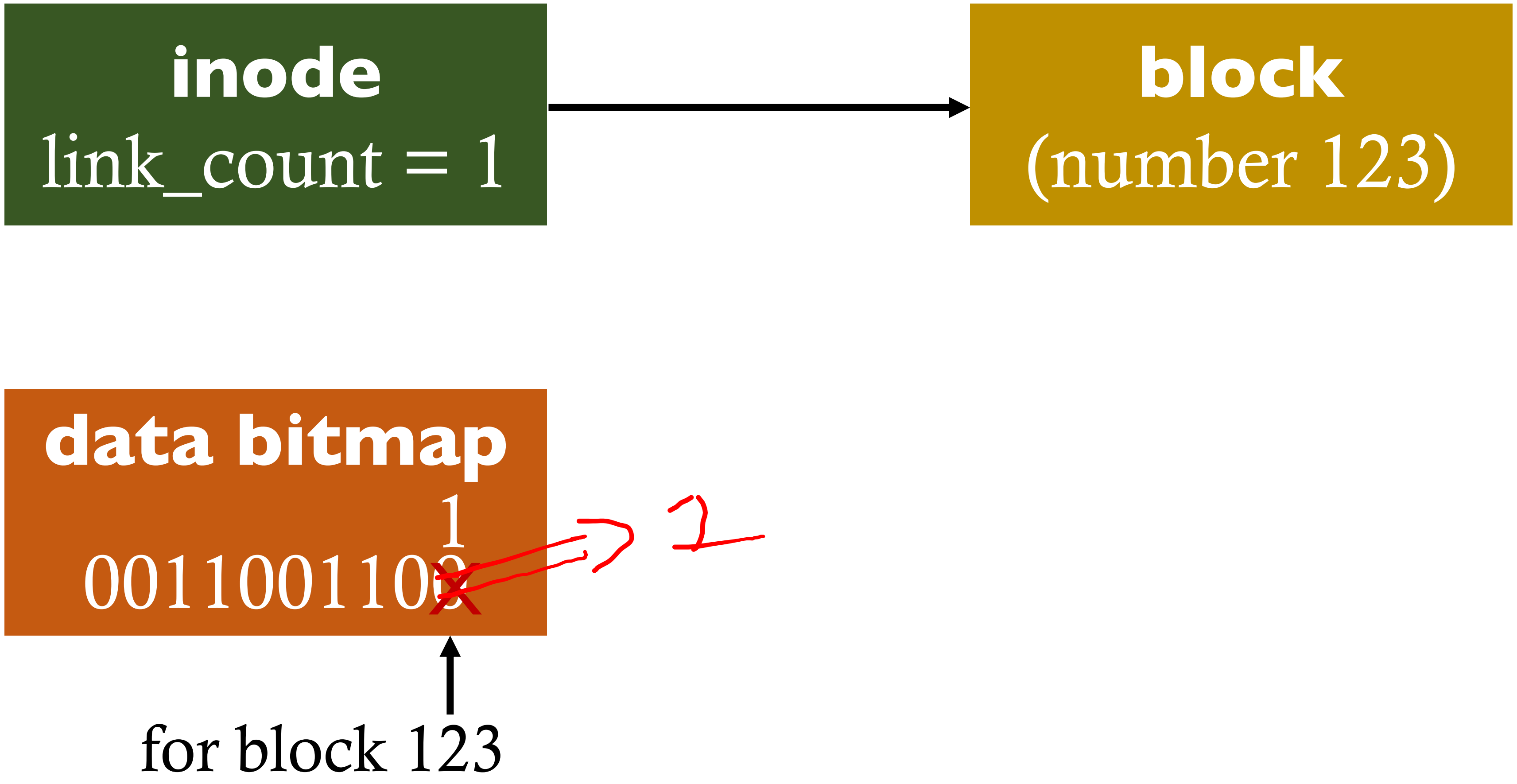
Traditional Solution: FSCK

FSCK: “file system checker”

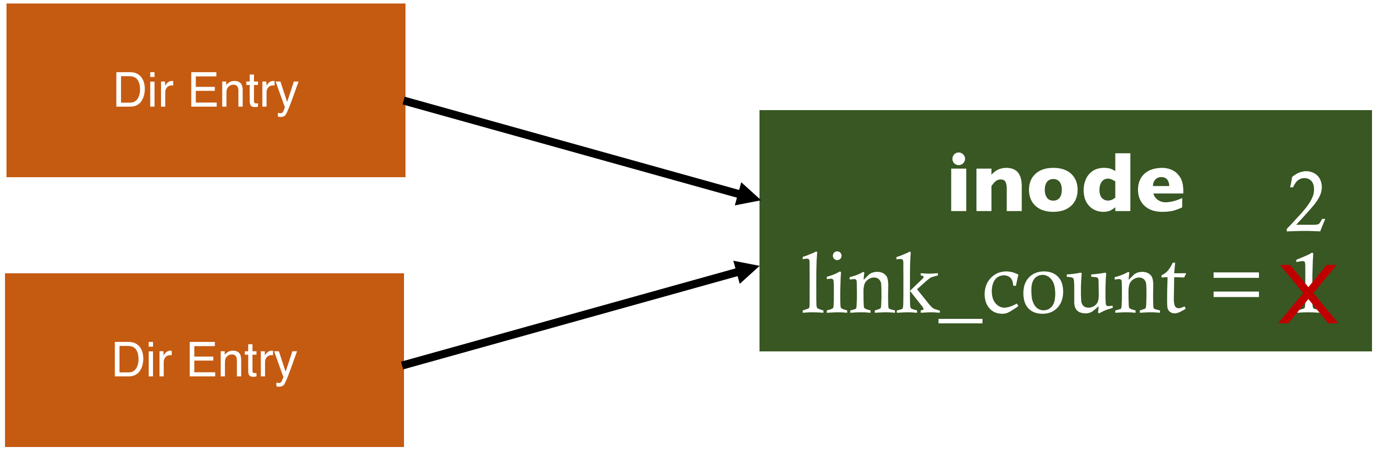
When system boots:

- 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

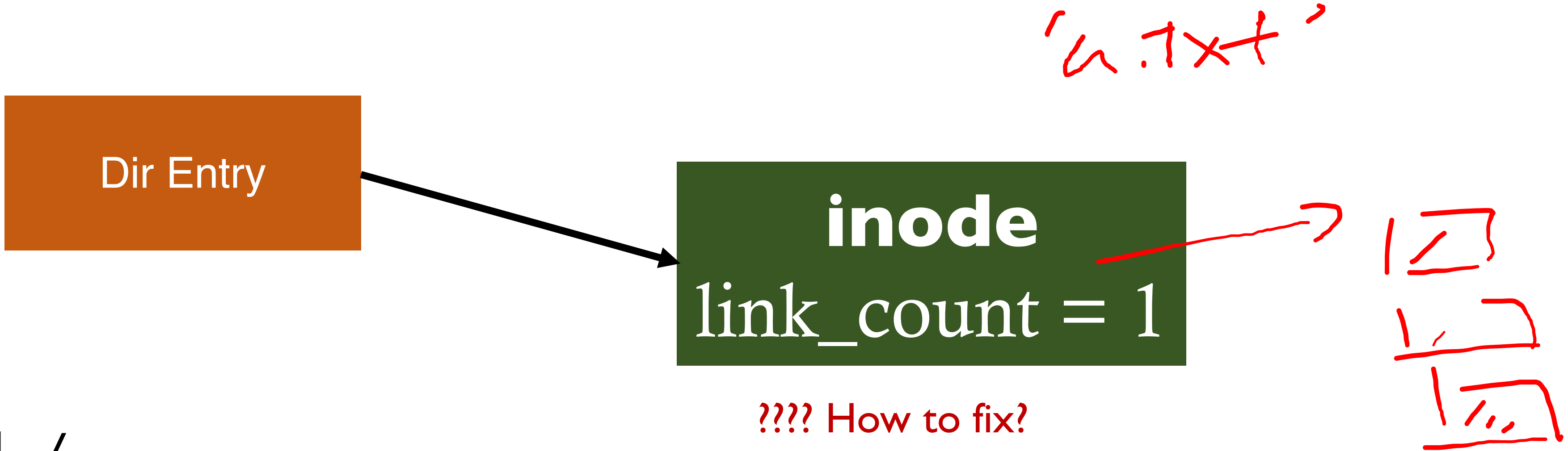
FSCK Example 1



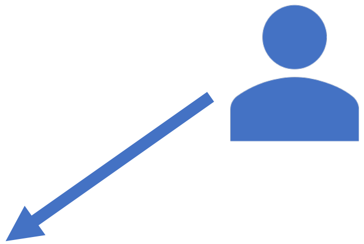
FSCK Example 2



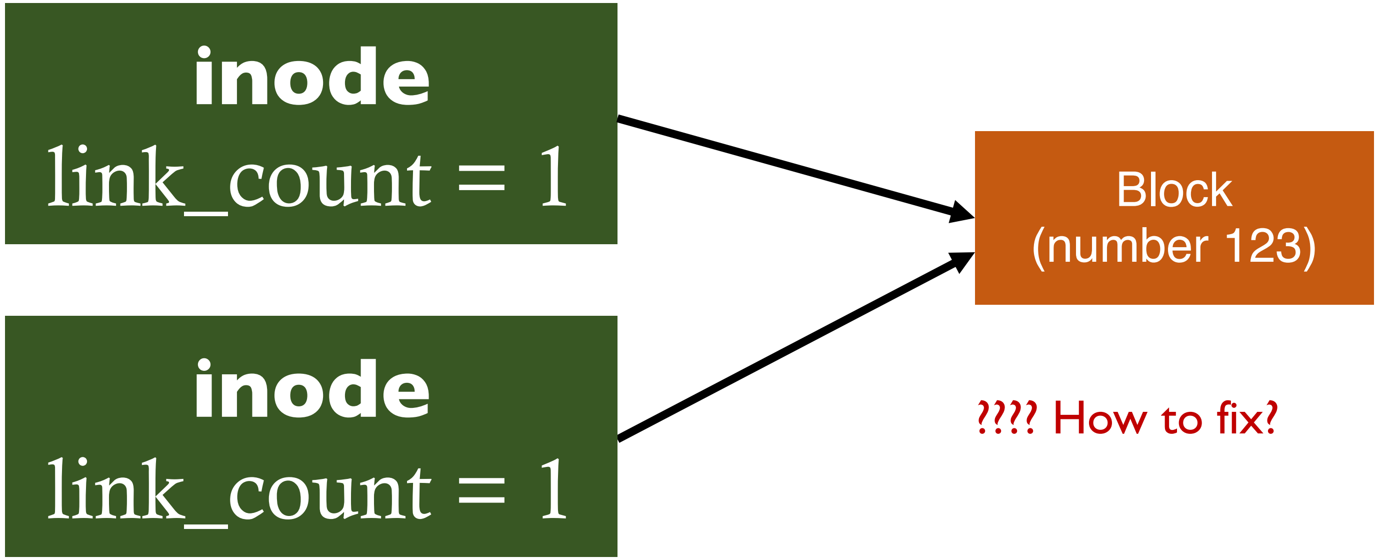
FSCK Example 3




```
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/
...
```

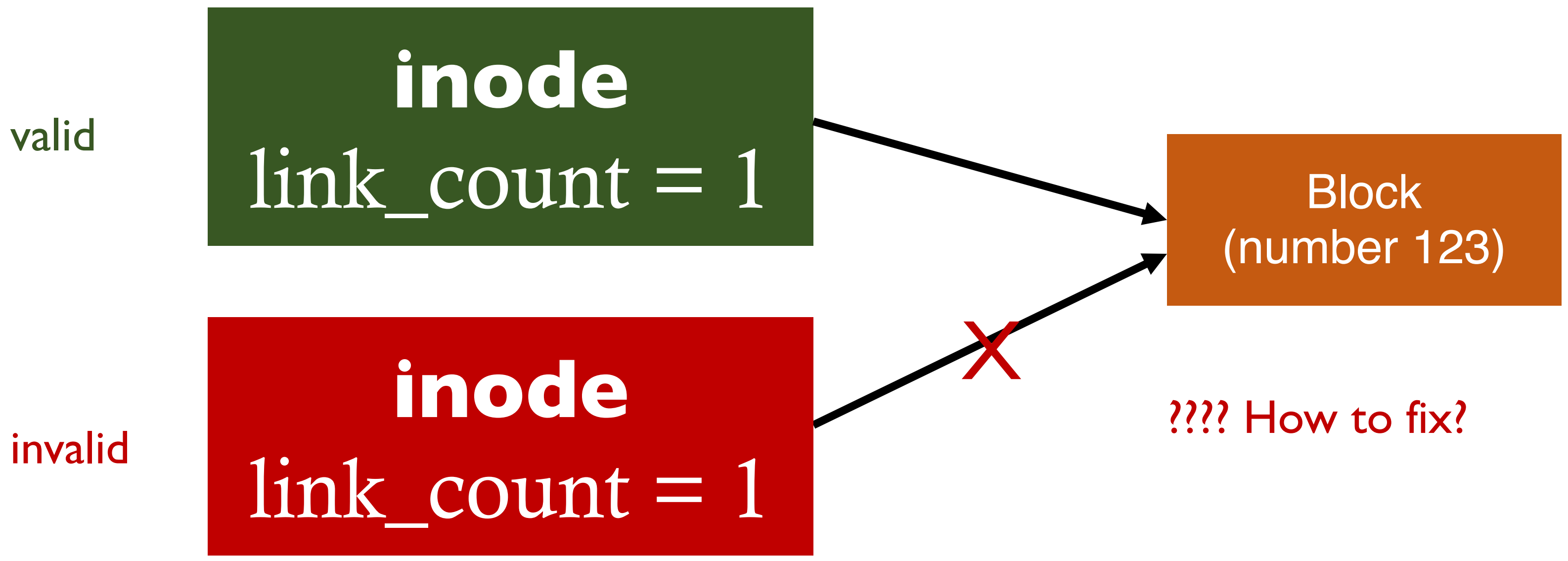


FSCK Example 4

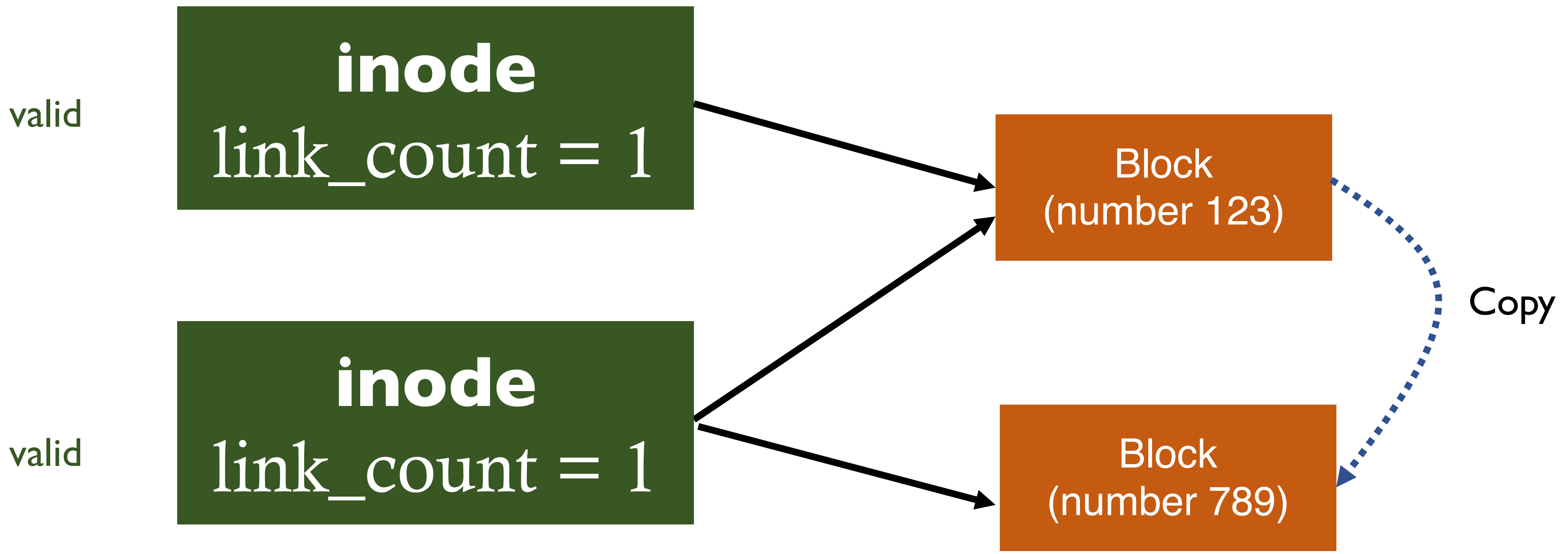


???? How to fix?

FSCK Example 4.a



FSCK Example 4.b



Traditional Solution: FSCK

FSCK: “file system checker”

When system boots:

- Make multiple passes over file system, looking for inconsistencies
- Try to fix automatically
  - Example: B’ I D, B I’ D
- Or punt to admin
  - Check `lost+found`, manually put the missing-link files to the correct place

Traditional Solution: FSCK

Problem:

- Cannot fix all crash scenarios
  - Can B’ I D’ be fixed?
- Performance
  - Sometimes takes hours to run
    - Checking a 600GB disk takes ~70 minutes
  - Does fsck have to run upon every reboot?
- Not well-defined consistency

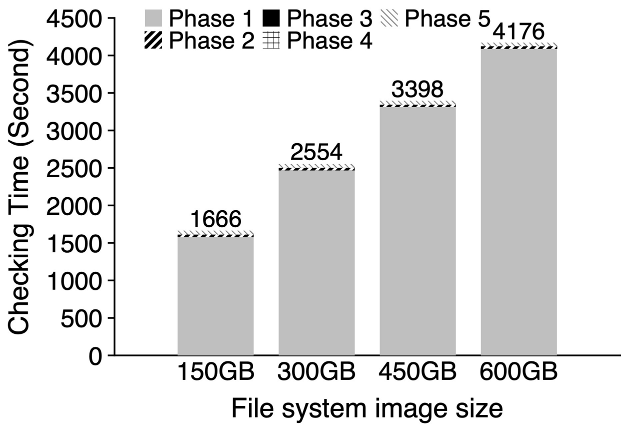


Figure 1: **e2fsck Execution Time By Size.** *This graph shows e2fsck’s execution time for differently sized file-system images, broken down by time spent in each phase.*

“ffsck: The Fast File System Checker”, Ao Ma et al. FAST ‘13

Another Solution: Journaling

Idea: Write "intent" down to disk before updating file system

- 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 → no-op
  - Crash after "intent" is written → redo op
- The process is called "recovery"

Case Study: Linux Ext3

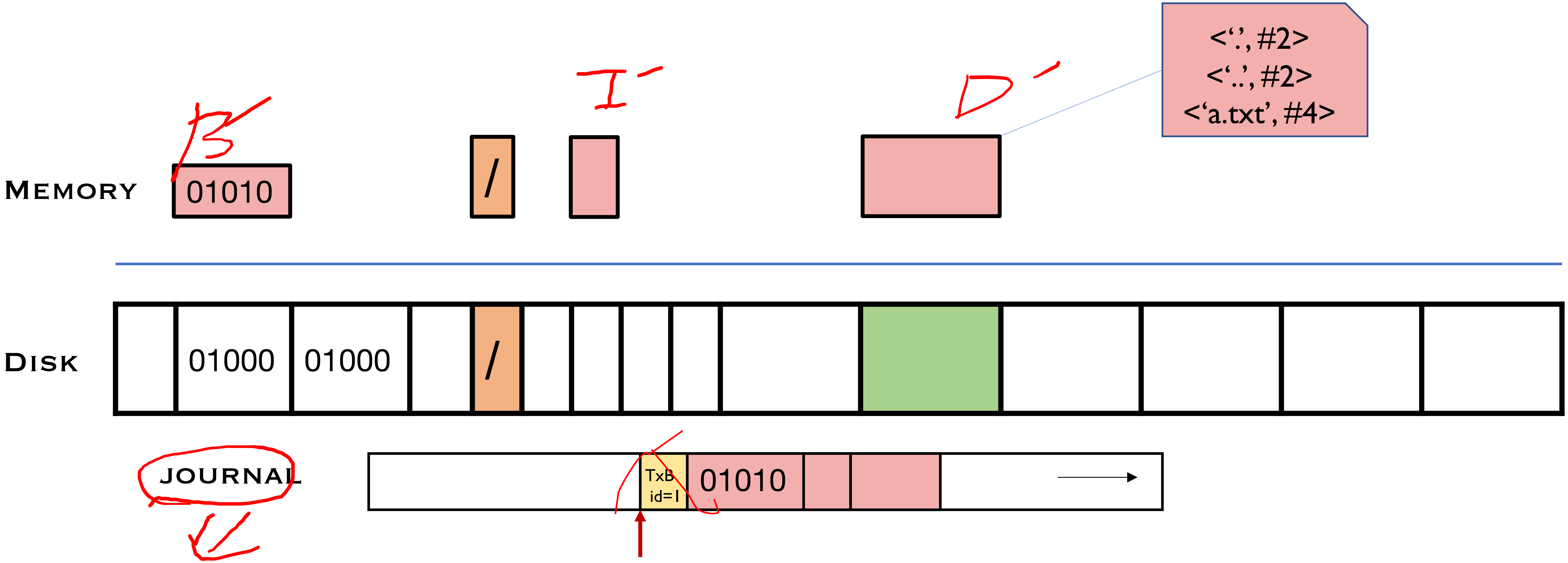
Physical journaling: write real block contents of the update to log

- Four totally ordered steps
  - Commit dirty blocks to journal as one transaction (`TxBegin`, I, B, D blocks)
  - Write commit record (`TxEnd`)
  - Copy dirty blocks to real file system (checkpointing)
  - Reclaim the journal space for the transaction

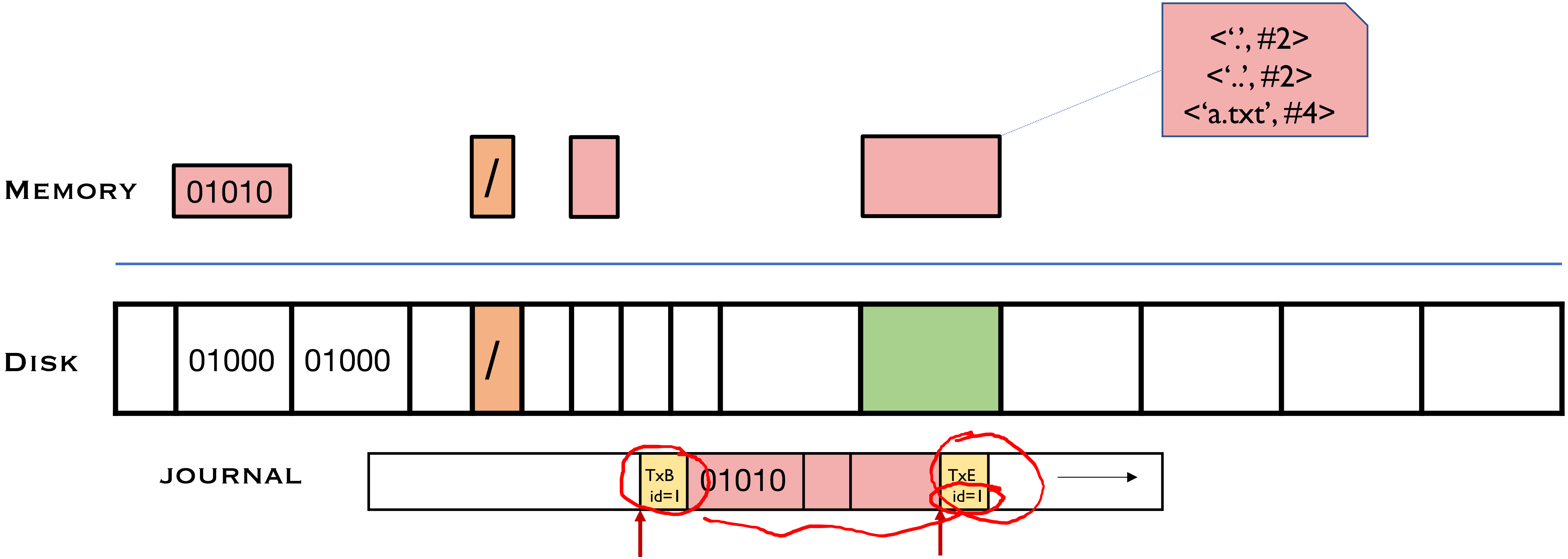
Logical journaling: write logical record of the operation to log

- "Add entry F to directory data block D"
- Complex to implement
- May be faster and save disk space

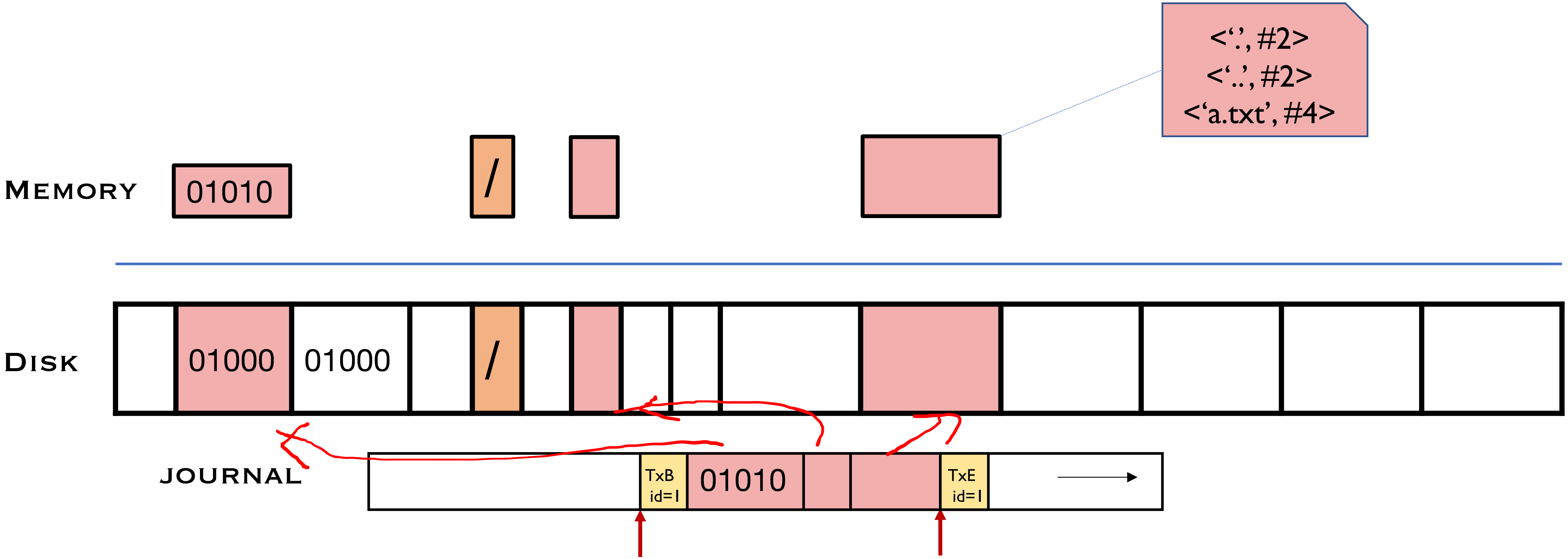
Step 1: Write Blocks to Journal



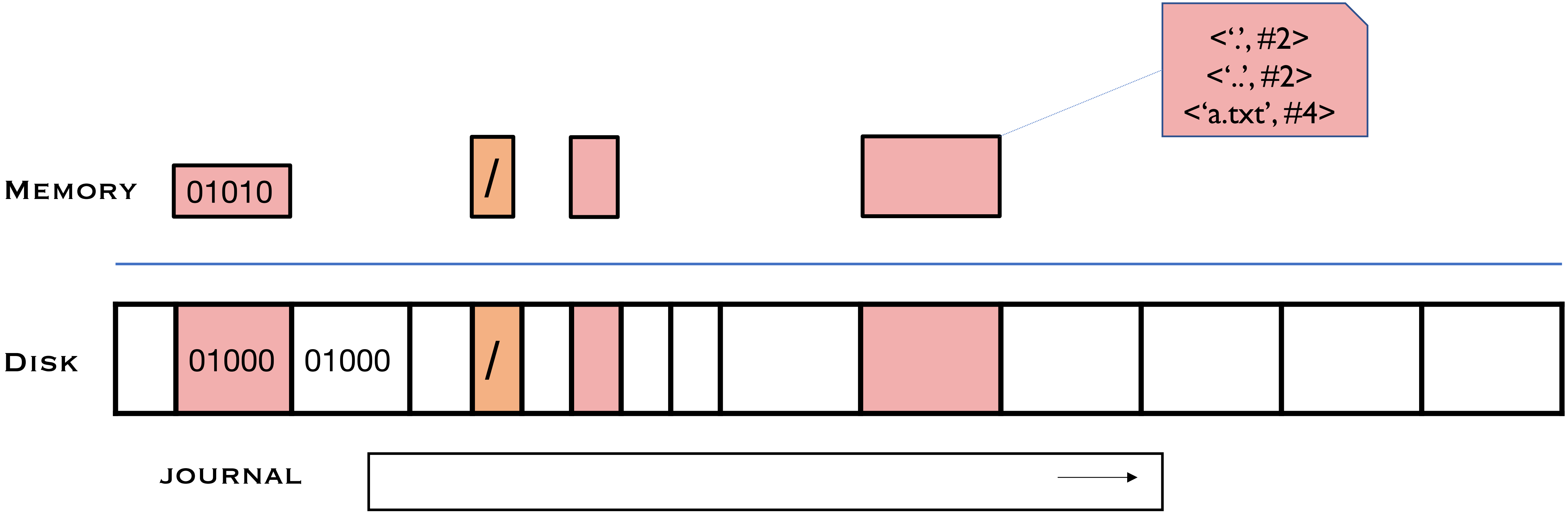
Step 2: Write Commit Record



Step 3: Copy Dirty Blocks to Real FS



Step 4: Reclaim Journal Space

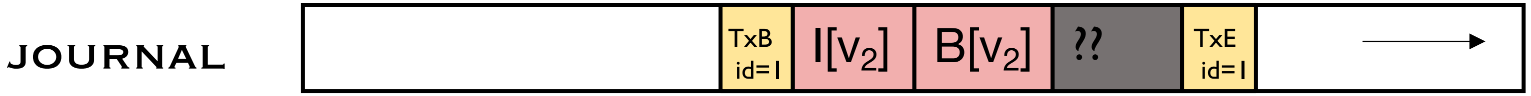


What If There Is A Crash?

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

What if …

- `TxBegin` but not `TxEnd` in log? → no-op why?
- `TxBegin` through `TxEnd` are in log, but I, B, and D have not yet been checkpointed?



  - How could this happen?
  - Why don't we merge step 2 and step 1? → |

- What if Tx is in log, I, B, D have been checkpointed, but Tx has not been freed from log?

→ re-do

Summary of Journaling Write Orders

Journal writes < FS writes

- Otherwise, crash ➔ FS broken, but no record in journal to patch it up

FS writes < Journal clear

- Otherwise, crash ➔ FS broken, but record in journal is already cleared

Journal writes < commit record write < FS writes

- Otherwise, crash ➔ record appears committed, but contains garbage

Ext3 Journaling Modes

Journaling has cost

- one write = two disk writes, two seeks

Several journaling modes balance consistency and performance

Data journaling: journal all writes, including file data

- Problem: expensive to journal data

Metadata journaling: journal only metadata

- Used by most FS (IBM JFS, SGI XFS, NTFS)
- Problem: file may contain garbage data

Ordered mode: write file data to real FS first, then journal metadata

- Default mode for ext3
- Problem: old file may contain new data

Summary

The consistent update problem

- Example of file creation and different crash scenarios

Two approaches to crash consistency

- FSCK: slow, not well-defined consistency
- Journaling: well-defined consistency, different modes

Other approach

- Soft updates (advanced OS topics)

Next Time…

Read Appendix B