Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 20: Crash consistency

Prof. Ryan Huang

Administration

Project 4 is released

- Due on April 22nd

Multi-threaded network file server

- Network programming, file systems, client-server, threads/concurrency
- Experience writing a substantial concurrent program

Project 2: hardest concepts

Project 3: hardest data structures

Project 4: largest program

The consistent update problem

Guarantee from disk: atomically write one sector

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

A file system operation may modify multiple sectors

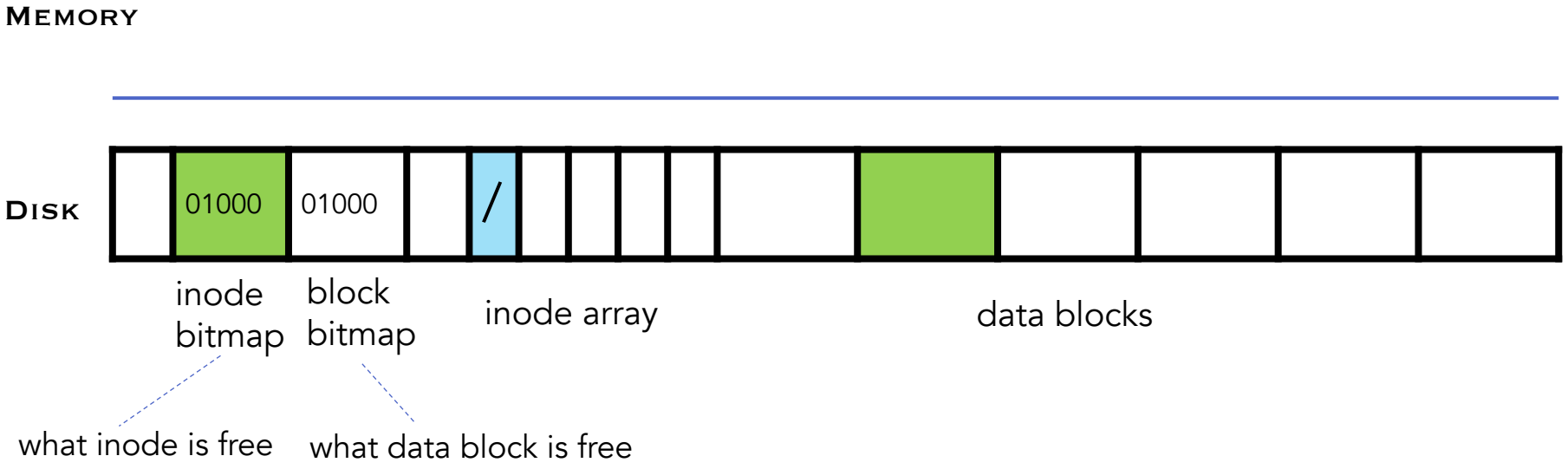
- E.g., creating a file requires writing the file header (e.g., an inode) and updating directory data to point to new file header

Crash ➔ File system is partially updated

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

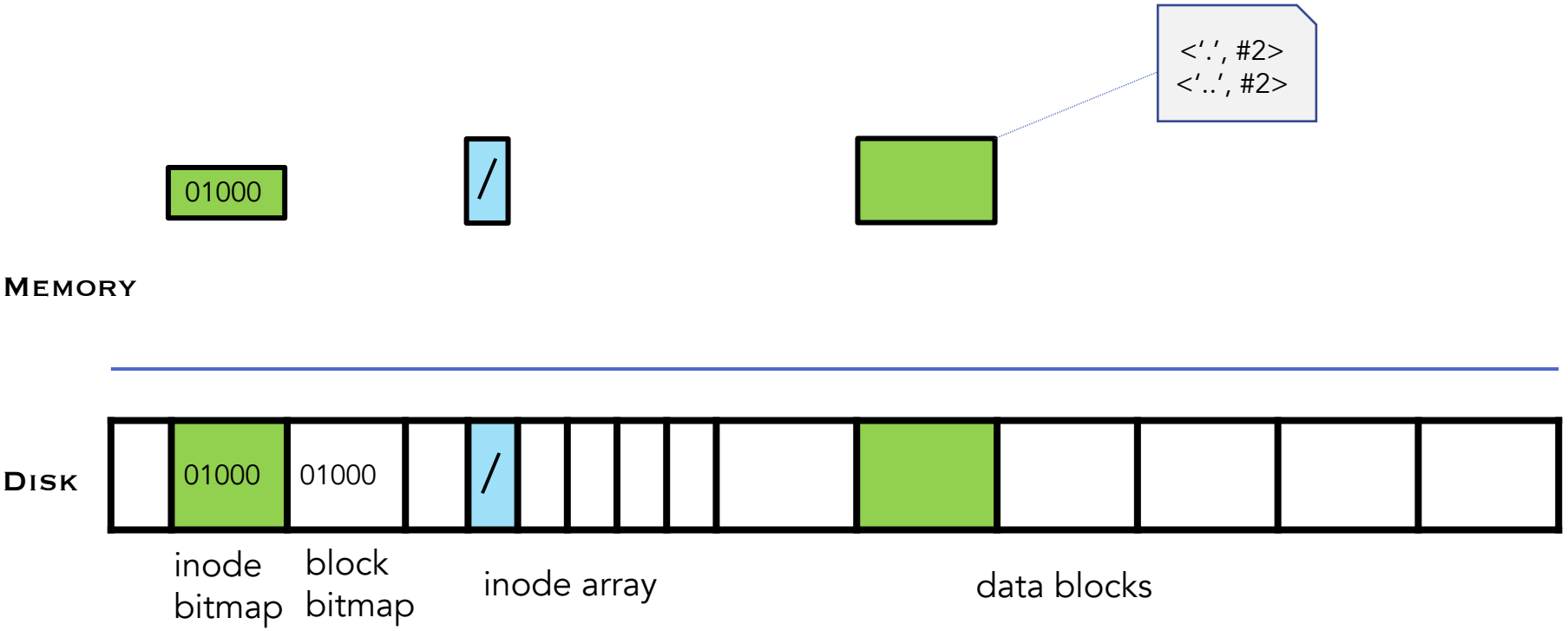
Example: create file /os.txt

Initial state



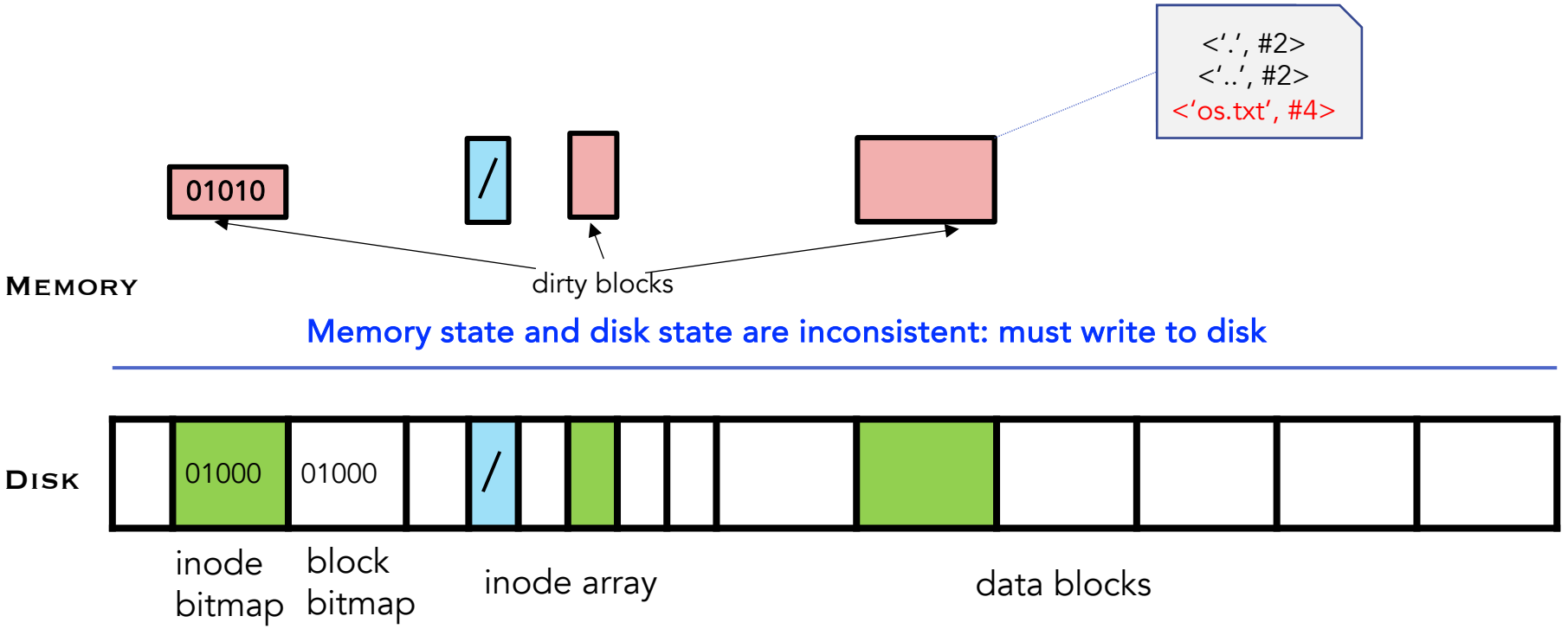
Example: create file /os.txt

Read to in-memory cache



Example: create file /os.txt

Modify metadata and blocks



Crash during updates

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

Old	New
B = 01000	B′ = 01010
I = free	I′ = allocated, initialized
D = {…}	D′ = {…, <‘os.txt’, 4>}

Crash scenarios: any subset can be written

- B′ I D, B I′ D, B′ I′ D, …, B′ I′ D′

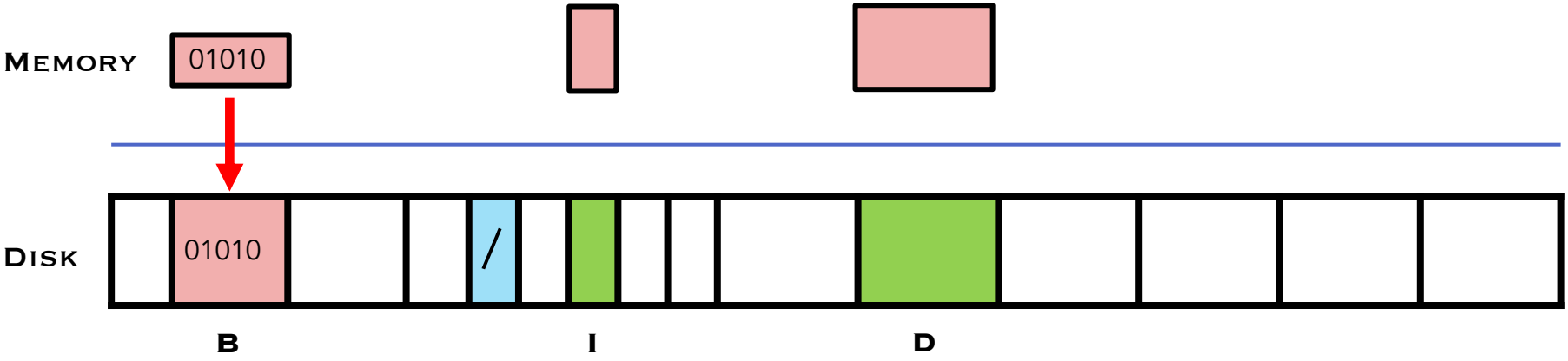
Is careful ordering enough?

Option 1: bitmap first

Write ordering: bitmap (B), inode (I), data (D)

CRASH *after* B reaches disk, *before* I or D

Result?

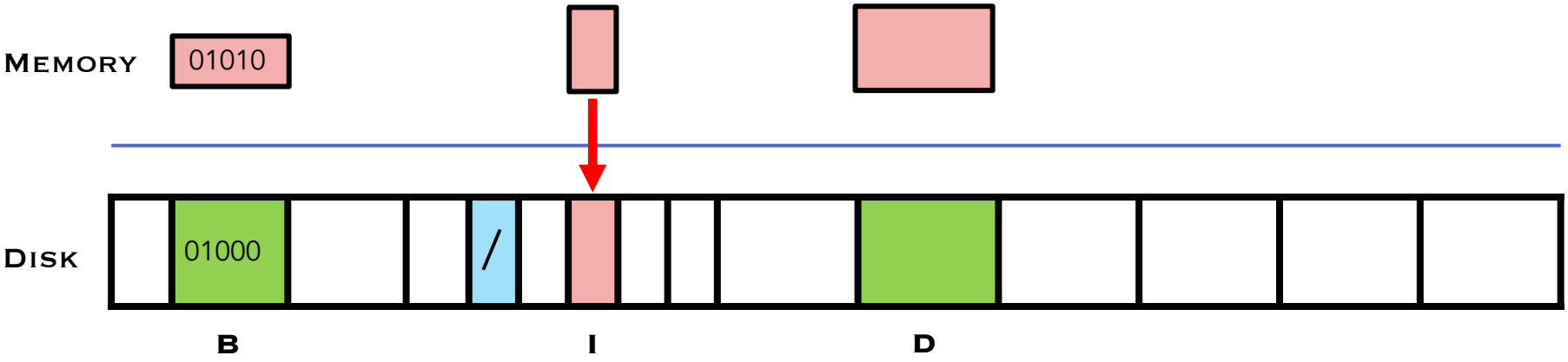


Option 2: inode first

Write ordering: inode (I), bitmap (B), data (D)

CRASH *after* I reaches disk, *before* B or D

Result?

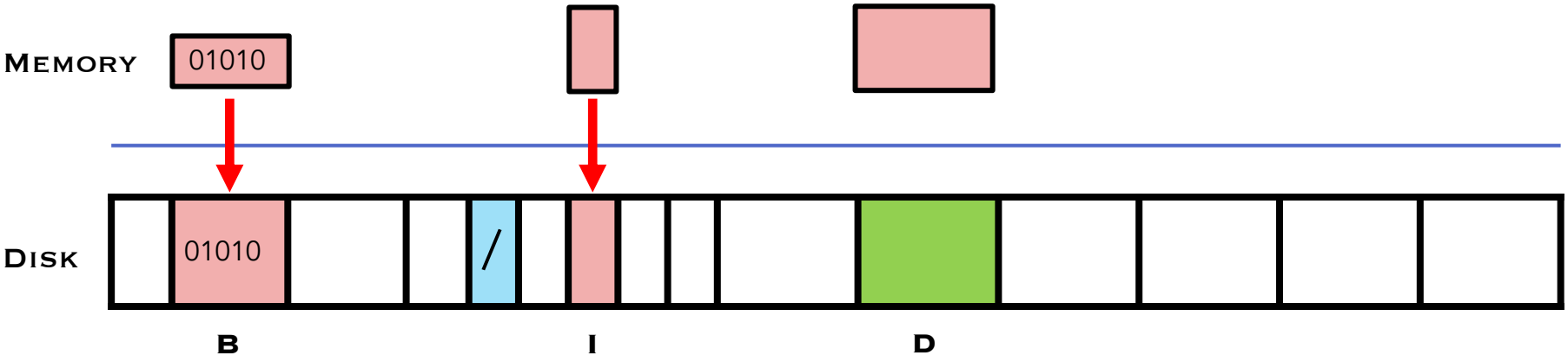


Option 2: inode first

Write ordering: inode (I), bitmap (B), data (D)

CRASH *after* I AND B reach disk, *before* D

Result?

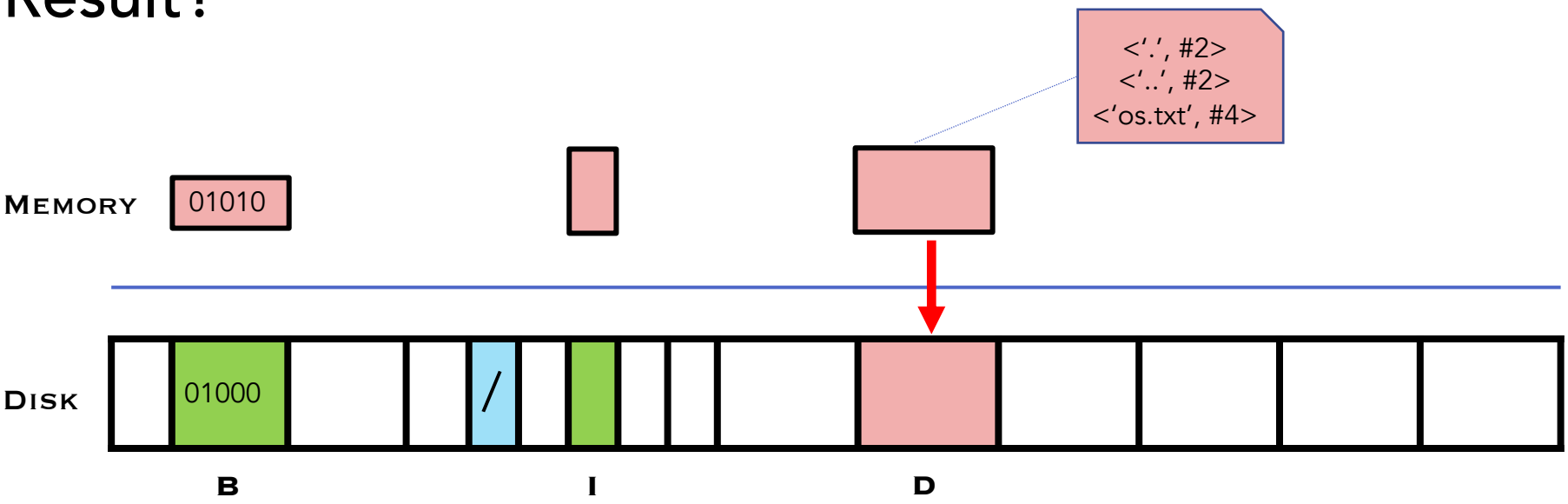


Option 3: data first

Write ordering: data (D) , bitmap (B), inode (I)

CRASH *after* D reaches disk, *before* I or B

Result?



What does this remind you of?

Transactions

A set of persistent updates that appear to be atomic and durable, even if the system crashes

- ACID: Atomic, Consistent, Isolated, Durable

```
begin transaction
  write disk
  write disk
  write disk
end (commit) transaction
```

Only two acceptable outcomes if crash occurs:

- "All": all disk writes in the transaction complete
- "Nothing": no disk writes in the transaction occur

Transaction via logging

Divide storage into:

- Data store: Persistent copy of data
- Log (journal): Append-only record of changes

Write "intent" to log before updating data store

- Called the "Write Ahead Logging" or "journaling"
- Originated from database community

When crash occurs, inspect log

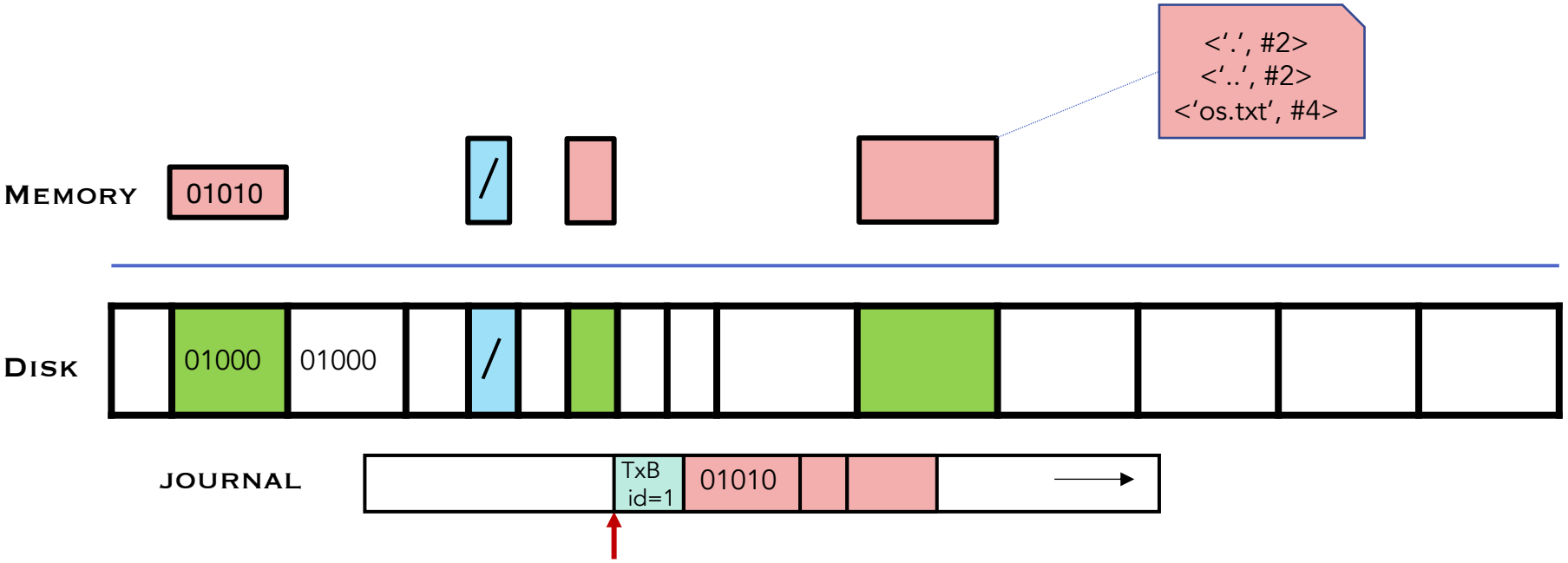
- Use contents of log to fix file system structures
  - Crash before "intent" is fully written ➔ no-op
  - Crash after "intent" is fully written ➔ redo op
- The process is called "recovery"

Logging steps

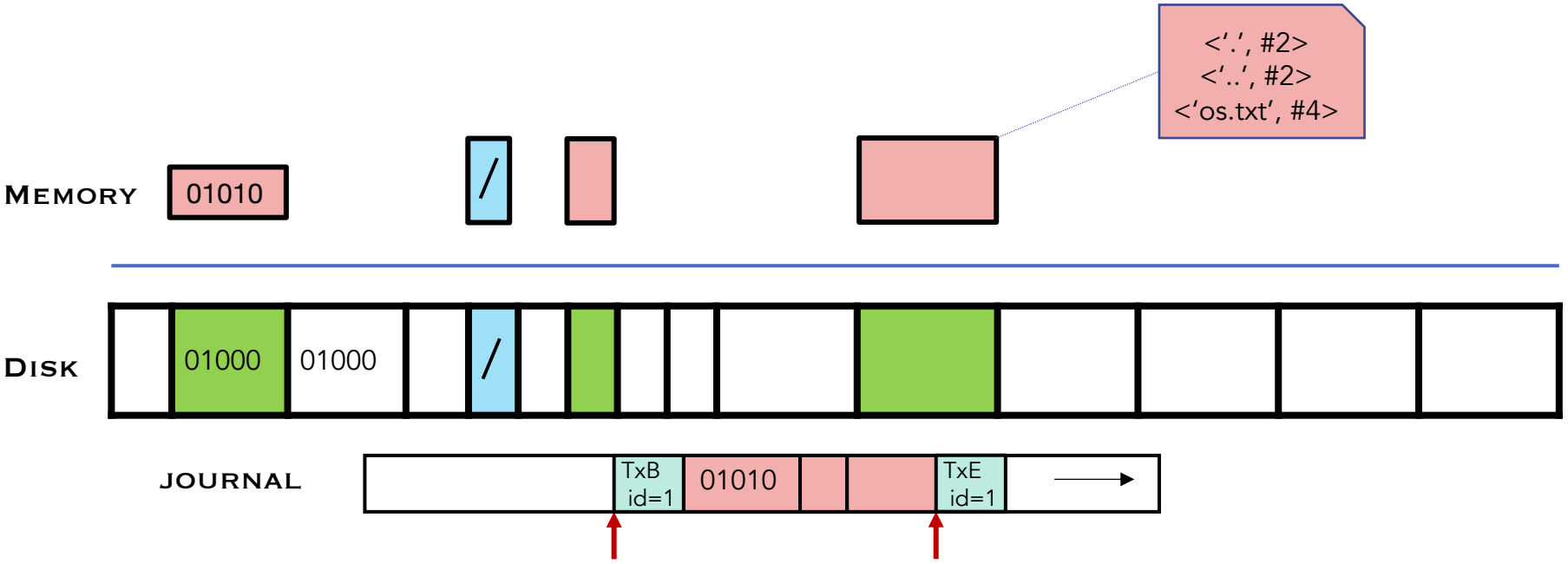
Four totally ordered steps:

1. Write updates (dirty blocks) to journal
   - `TxBegin`, I, B, D blocks
2. Write commit record
   - `TxEnd`
3. Copy dirty blocks to real file system
4. Reclaim the journal space for the transaction

Step 1: Write updates to journal

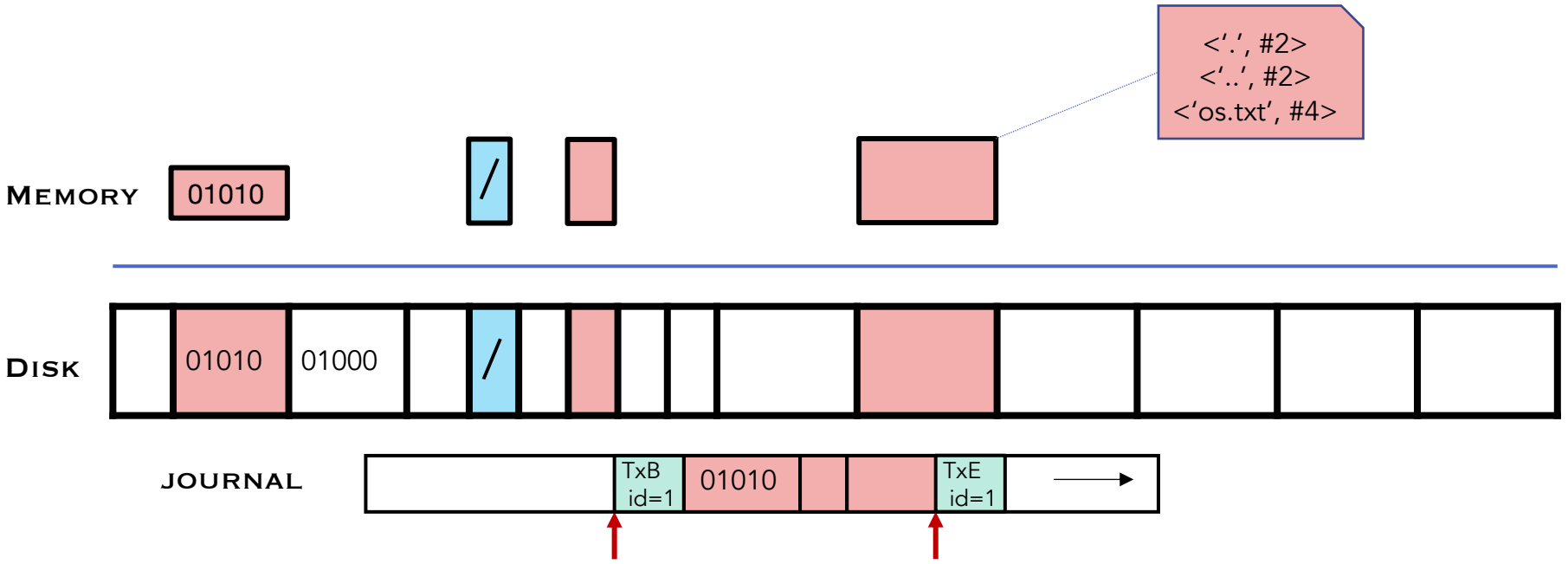


Step 2: Write commit record

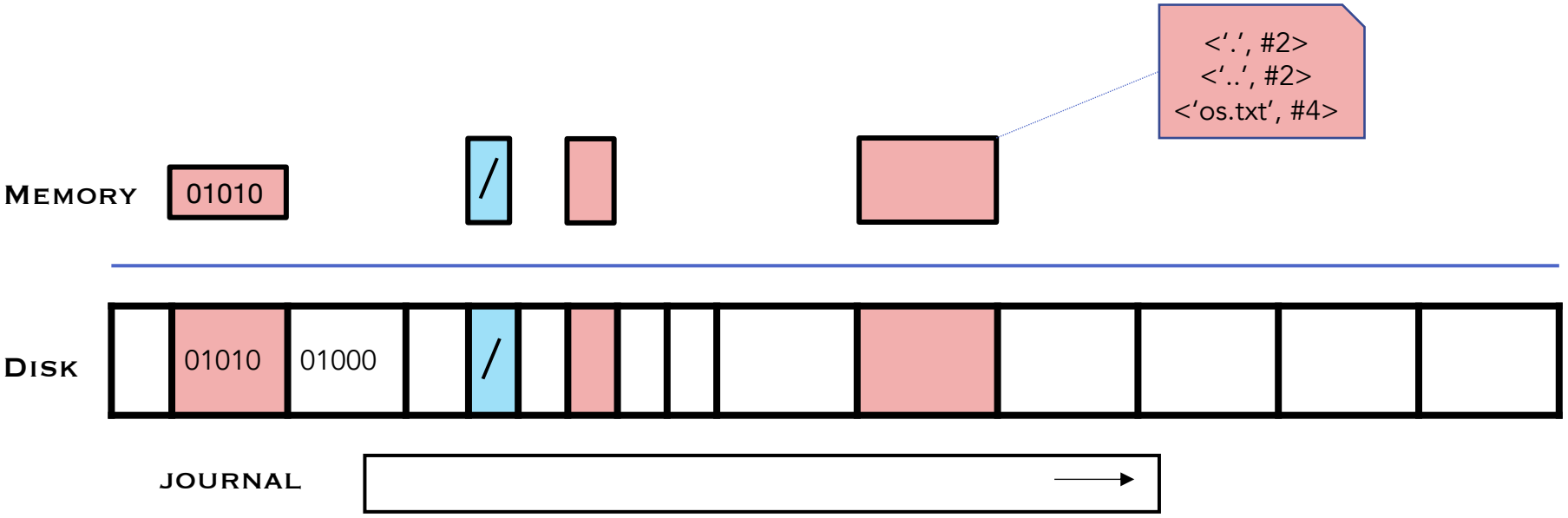


Step 3: Copy updates to data store

Called checkpointing



Step 4: Reclaim journal space

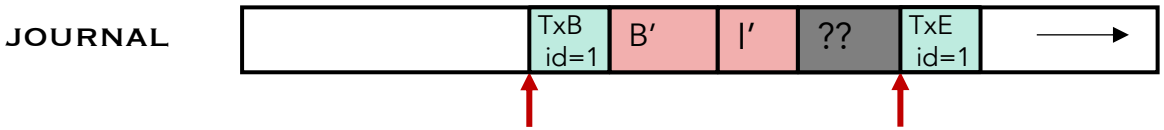


What if there is a crash?

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

What if …

- `TxBegin` but not `TxEnd` in log?
- `TxBegin` and `TxEnd` are in log, but D has not reached the journal?



- How could this happen?
  - Disk controller may reorder the writes from OS!
- Why don't we merge step 2 and step 1?
  - To avoid this situation by waiting for all updates to be in journal

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

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 FS first, then journal metadata

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

Another 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
  - e.g., B' I D, B I' D
- Or punt to admin
  - `lost+found`, rely on users to put the missing-link files to the correct place

Problem:

- Cannot fix all crash scenarios (e.g., can B' I D' be fixed?)
- Performance (sometimes takes hours to run)
- Not well-defined consistency