File System Reliability, Journaling File System

EE469, April 11

Yiying Zhang

Reading

- Chapters 11-12
- Comet: Chapter 42

Roadmap

- Functionality (API)
  - Basic functionality
    - Disk layout
    - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation
  - File system designs
  - Buffer cache
- Reliability
  - FS level
  - Disk level: RAID

File system reliability

- Loss of data in a file system can have catastrophic effect
  - How does it compare to hardware (DRAM) failure?
  - Need to ensure safety against data loss

- Three threats:
  - Accidental or malicious deletion of data → backup
  - Media (disk) failure → data replication (e.g., RAID)
  - System crash during file system modifications → consistency

1. Backup

- Copy entire file system onto low-cost media (tape), at regular intervals (e.g. once a day).
  - Backup storage (cold storage)

- In the event of a disk failure, replace disk and restore from backup media

- Amount of loss is limited to modifications occurred since last backup

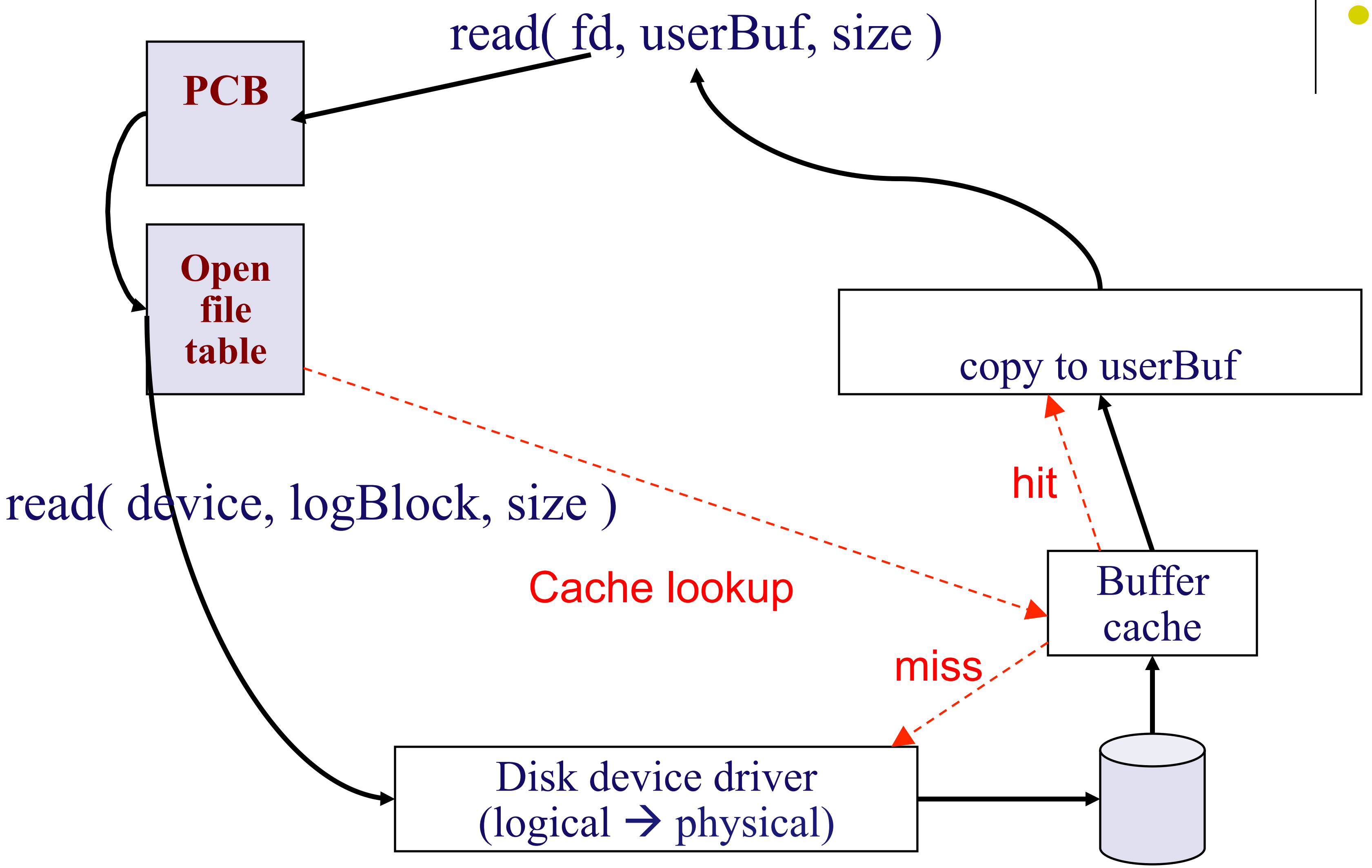
2. Data Replication

- Full replication
  - Mirroring across disks
  - Full replication to different machines (more next week)
- RAID (next lecture)
- Erasure Coding
  - Like RAID, use parity, but saves more space

3. Crash Recovery

- After a system crash in the middle of a file system operation, file system metadata may be in an *inconsistent state*

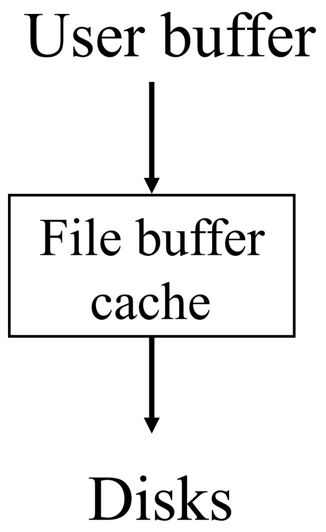
[lec23] Reading A Block



Modern disk drives are addressed as large one-dimensional arrays of logical blocks

[lec23] Write operations: Maintaining Consistency

- write(fd, buffer, n)
  - On a hit
    - write to buffer cache
  - On a miss
    - Read first
    - Then write (hit)



[lec23] File persistence under file caching

- Problem: fast cache memory is volatile, but users expect disk files to be persistent
  - In the event of a system crash, dirty blocks in the buffer cache are lost !
  - Example 1: creating "/dir/a"
    - Allocate inode (from free inode list) for "a"
    - Update parent dir content – add ("a", inode#) to "dir"
- Solution 1: use write-through cache
  - Modifications are written to disk immediately
    - (minimize "window of opportunities")
  - No performance advantage for disk writes

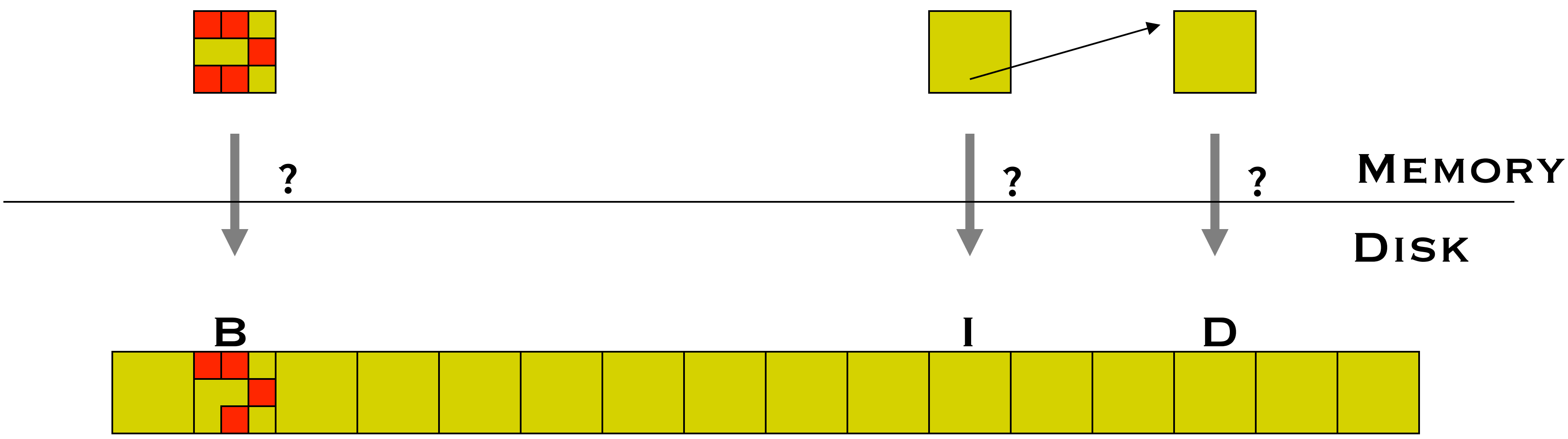
[lec23] File persistence under file caching

- Possible solution 2: write back cache
  - Gather (buffer) writes in memory and then write all buffered data back to storage devices
  - e.g., write back dirty blocks after no more than 30 seconds
  - e.g., write back all dirty blocks during file close

- Problem with this?

Many "dirty" blocks in memory: What order to write to disk?

- Example: Appending a new block to existing file
  - Write data bitmap B (for new data block),
    write inode I of file (to add new pointer, update time),
    write new data block D



Deep thinking

- One file operation may involve modifying multiple disk blocks (and hence multiple disk I/Os)

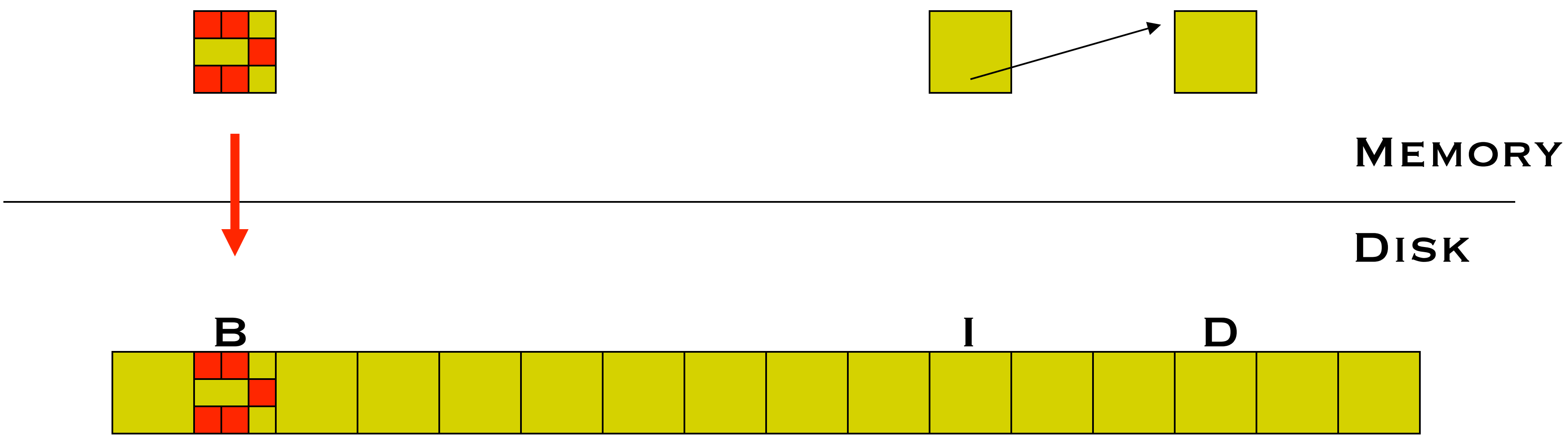
- After crashing, do we know which blocks were involved at the moment of crashing?

The 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

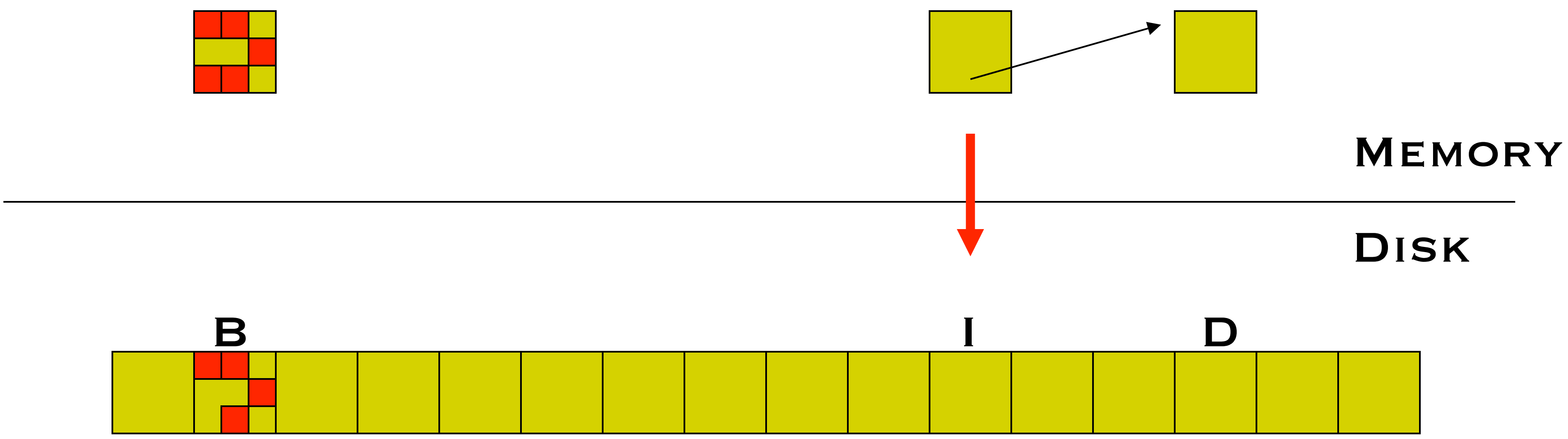
Example: Bitmap first

- Write Ordering: Bitmap (B), Inode (I), Data (D)
  - But CRASH after B has reached disk, before I or D
- Result?



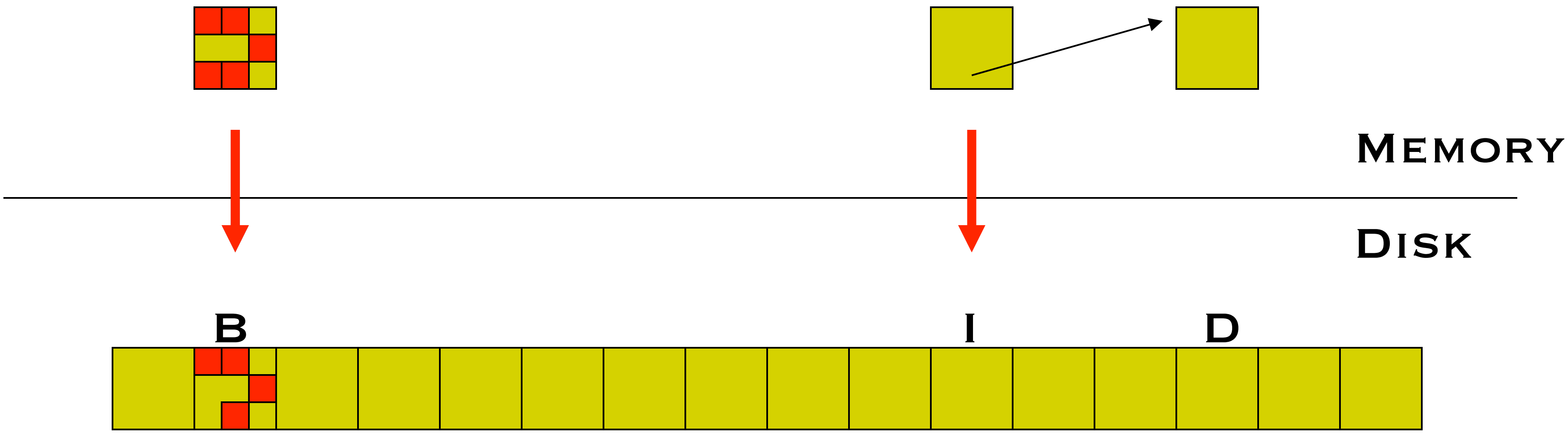
Example: Inode first

- Write Ordering: Inode (I), Bitmap (B), Data (D)
  - But CRASH after I has reached disk, before B or D
- Result?



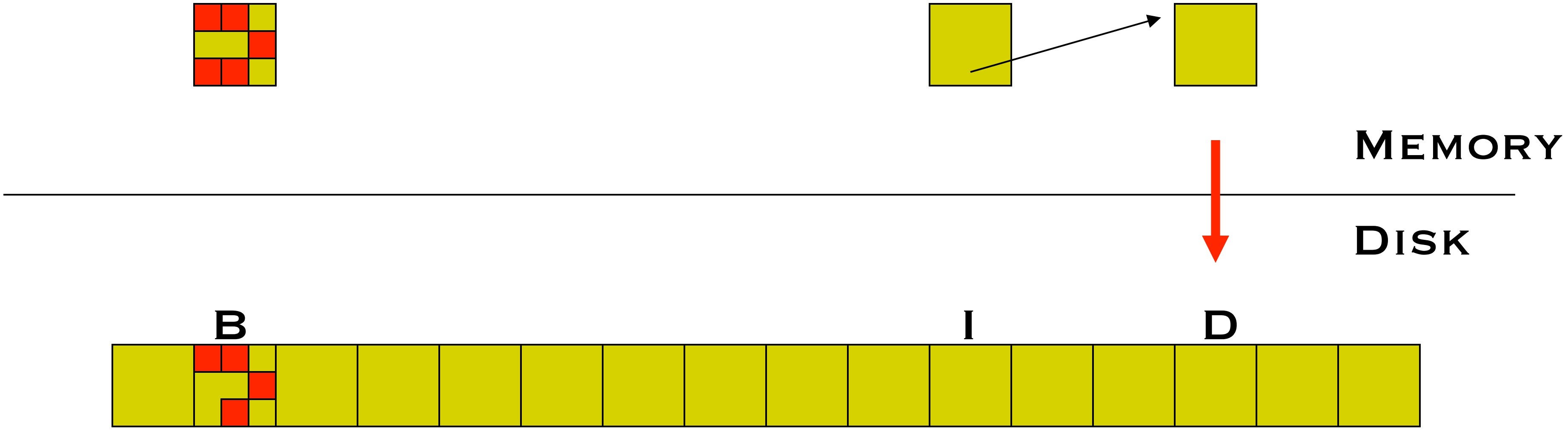
Example: Inode first

- Write Ordering: Inode (I), Bitmap (B), Data (D)
  - CRASH after I AND B have reached disk, before D
- Result?



Example: Data first

- Write Ordering: Data (D) , Bitmap (B), Inode (I)
  - CRASH after D has reached disk, before I or B
- Result?



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
  - Either fix automatically or punt to admin
  - Does fsck have to run upon every reboot?
- Main problem with fsck: Performance
  - Sometimes takes hours to run on large disk volumes

Larry Page and Sergey Brin

- Google Founder
- PhD students at Stanford
- PageRank paper
  - The PageRank Citation Ranking: Bringing Order to the Web (1998)

Jeff Dean

- Google Chief Scientist
- Distributed systems, deep learning
- MapReduce
- TensorFlow => AlphaGo
  - Tensor Processing Unit (TPU)

Andrew Ng

- Deep Learning
- Prof at Stanford
- Chief Scientist at Baidu leading deep learning group (recently quit)

How to ensure data consistency with arbitrary crash points?

- We need to ensure a “copy” of consistent state can always be recovered

- Either the old consistent state (before udpates)
- Undo Log
  - Make a copy of the old state to a different place
  - Update the current place

- Or the new consistent state (after updates)
- Redo Log
  - Write to a new place, leave the old place intact

The idea of Redo Log

- Idea: Write something down to disk at a different location from the data location
  - Called the "write ahead log" or "journal"
- When all data is written to redo log, write it back to the data location, and then delete the data on redo log

- When crash occurs, look through the redo log and see what was going on
  - Replay complete data, discard incomplete data
  - The process is called "recovery"

Journaling file systems

- Became popular ~2002, but date to early 80's
- There are several options that differ in their details
  - Ntfs (Windows), Ext3 (Linux), ReiserFS (Linux), XFS (Irix), JFS (Solaris)
- Basic idea
  - update metadata, or all data, *transactionally*
    - *"all or nothing"*
    - *Failure atomicity*
  - if a crash occurs, you may lose a bit of work, but the disk will be in a consistent state
    - more precisely, you will be able to quickly get it to a consistent state by using the transaction log/journal – rather than scanning every disk block and checking sanity conditions

Where is the Data?

- In file systems with memory cache, the data is in two places:
  - On disk
  - In in-memory caches
- The basic idea of the solution:
  - Always leave "home copy" of data in a consistent state
  - Make updates persistent by writing them to a sequential (chronological) journal partition/file
  - At your leisure, push the updates (in order) to the home copies and reclaim the journal space
  - Or, make sure log is written before updates

Journal

- Journal: an append-only file containing log records
  - <start t>
    - transaction t has begun
  - <t,x,v>
    - transaction t has updated block x and its new value is v
      - Can log block "diffs" instead of full blocks
      - Can log *operations* instead of data (operations must be idempotent and undoable)
  - <commit t>
    - transaction t has committed – updates will survive a crash
    - Only after the commit block is written is the transaction final
    - The commit block is a single block of data on the disk
- Committing involves writing the records – the home data doesn't need to be updated at this time

How does data get out of the journal?

- After a commit the new data is in the journal – it needs to be written back to its home location on the disk
- Cannot reclaim that journal space until we resync the data to disk

Journal checkpointing

- A cleaner thread walks the journal in order, updating the home locations (on disk, not the cache!) of updates in each transaction
- Once a transaction has been reflected to the home locations, it can be deleted from the journal

How does this help crash recovery?

- Only completed updates have been committed
  - During reboot, the recovery mechanism reapplies the committed transactions in the journal
- The old and updated data are each stored separately, until the commit block is written

If a crash occurs

- Open the log and parse
  - <start>, data, <commit> => committed transactions
  - <start>, no <commit> => uncommitted transactions
- Redo committed transactions
  - Re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any positive number of times with the same result.
- Undo uncommitted transactions
  - Undo updates from all uncommitted transactions
  - Write "compensating log records" to avoid work in case we crash during the undo phase

Case Study: Linux ext3

- Ext3: roughly ext2+journaling

- Ext3 grew out of ext2
- Exact same code base
- Completely backwards compatible *(if you have a clean reboot)*

ext3 and journaling

- Two separate layers
  - /fs/ext3 – just the filesystem with transactions
  - /fs/jdb – just the journaling stuff

- ext3 calls jbd as needed
  - Start/stop transaction
  - Ask for a journal recovery after unclean reboot

ext3 journaling

- Journal location
  - EITHER on a separate device partition
  - OR just a “special” file within ext2
- Three separate modes of operation:
  - Data: All data is journaled
  - Ordered, Writeback: Just metadata is journaled
- First focus: Data journaling mode

Transactions in ext3 Data Journaling Mode

- Same example: Update Inode (I), Bitmap (B), Data (D)
- First, write to journal:
  - Transaction begin (Tx begin)
  - Transaction descriptor (info about this Tx)
  - I, B, and D blocks (in this example)
  - Transaction end (Tx end)
- Then, "checkpoint" data to fixed ext2 structures
  - Copy I, B, and D to their fixed file system locations
- Finally, free Tx in journal
  - Journal is fixed-sized circular buffer, entries must be periodically freed

What if there's a Crash?

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

- What if …
  - Tx begin but not Tx end in log?
  - Tx begin through Tx end are in log,
    but I, B, and D have not yet been checkpointed?
  - What if Tx is in log, I, B, D have been checkpointed,
    but Tx has not been freed from log?

- Performance? (As compared to fsck?)

Complication: Disk/SSD Scheduling

- Problem: Low-levels of I/O subsystem in OS and even the disk/RAID itself may reorder requests
- How does this affect Tx management?
  - Where is it OK to issue writes in parallel?
    - Tx begin
    - Tx info
    - I, B, D
    - Tx end
    - Checkpoint: I, B, D copied to final destinations
    - Tx freed in journal

Complication: Disk/SSD Buffering

- Problem: Disks (SSDs) have internal memory to buffer writes. When the OS writes to disk, it does not necessarily mean that the data is written to persistent media
- How does this affect Tx management?
  - Tx begin
  - Tx info
  - I, B, D
  - Tx end
  - Checkpoint: I, B, D copied to final destinations
  - Tx freed in journal

Problem with Data Journaling

- Data journaling: Lots of extra writes
  - All data committed to disk twice (once in journal, once to final location)
- Overkill if only goal is to keep metadata consistent
  - Why is this sometimes OK?

- Instead, use writeback mode
  - Just journals metadata
  - Data is not journaled. Writes data to final location directly
  - Better performance than data journaling (data written once)
  - The contents might be written at any time (before or after the journal is updated)
- Problems?
  - If a crash happens, metadata can point to old or even garbage data!

Ext3 ordered journaling mode

- How to order data block write w.r.t. journal (metadata) writes?

- Ordered journaling mode
  - Only metadata is journaled, file contents are not (like writeback mode)
  - But file contents guaranteed to be written to disk before associated metadata is marked as committed in the journal
  - Default ext3 journaling mode

  - What happens when crash happens?
    - Metadata will only point to correct data (no stale data can be reached after reboot).
    - But there may be data that is not pointed to by any metadata.
    - How is this better than writeback in terms of consistency guarantees?

Conclusions

- Journaling
  - Almost all modern file systems use journaling to reduce recovery time during startup (e.g., Linux ext3, ReiserFS, SGI XFS, IBM JFS, NTFS)
  - Simple idea: Use write-ahead log to record some info about what you are going to do before doing it
  - Turns multi-write update sequence into a single atomic update ("all or nothing")
  - Some performance overhead: Extra writes to journal
    - Worth the cost?