Goals for Today

- <u>Learning Objective</u>:
  - Learn how to achieve reliability + availability in storage!!
- <u>Announcements, etc</u>:
  - MP3 is out! Due **April 18th**.
  - MP3 Walkthrough on Monday, slides up later today

Reminder: Please put away devices at the start of class

CS 423
Operating System Design: Reliable Storage

Professor Adam Bates
Spring 2018

Storage is hard ; - (

“In each cluster's first year, it's typical that 1,000 individual machine failures will occur; ***thousands of hard drive failures will occur****; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours;* 20 racks will fail, *each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; and the cluster will have to be rewired once, affecting 5 percent of the machines at any given moment over a 2-day span, Dean said. And there's about a 50 percent chance that the cluster will overheat, taking down most of the servers in less than 5 minutes and taking 1 to 2 days to recover.”*

- Jeff Dean, Google Fellow (2008)

Storage Goals

- Storage reliability: data fetched is what you stored
  - Problem when machines randomly fail!
- Storage availability: data is there when you want it
    - Problem when disks randomly fail!
    - More disks => higher probability of some disk failing
    - Data available ~ Prob(disk working)^k
      - If failures are independent and data is spread across k disks
    - For large k, probability system works -> 0

File System Reliability

- What can happen if disk loses power or machine software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete
- File systems need durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Transaction Concept

- A transaction is a grouping of low-level operations that are related to a single logical operation
- Transactions are atomic — operations appear to happen as a group, or not at all (at logical level)
  - At physical level of course, only a single disk/flash write is atomic
- Transactions are durable — operations that complete stay completed
  - Future failures do not corrupt previously stored data
- (In-Progress) Transactions are isolated — other transactions cannot see the results of earlier transactions until they are committed
- Transactions exhibit consistency — sequential memory model

Reliability Attempt #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely

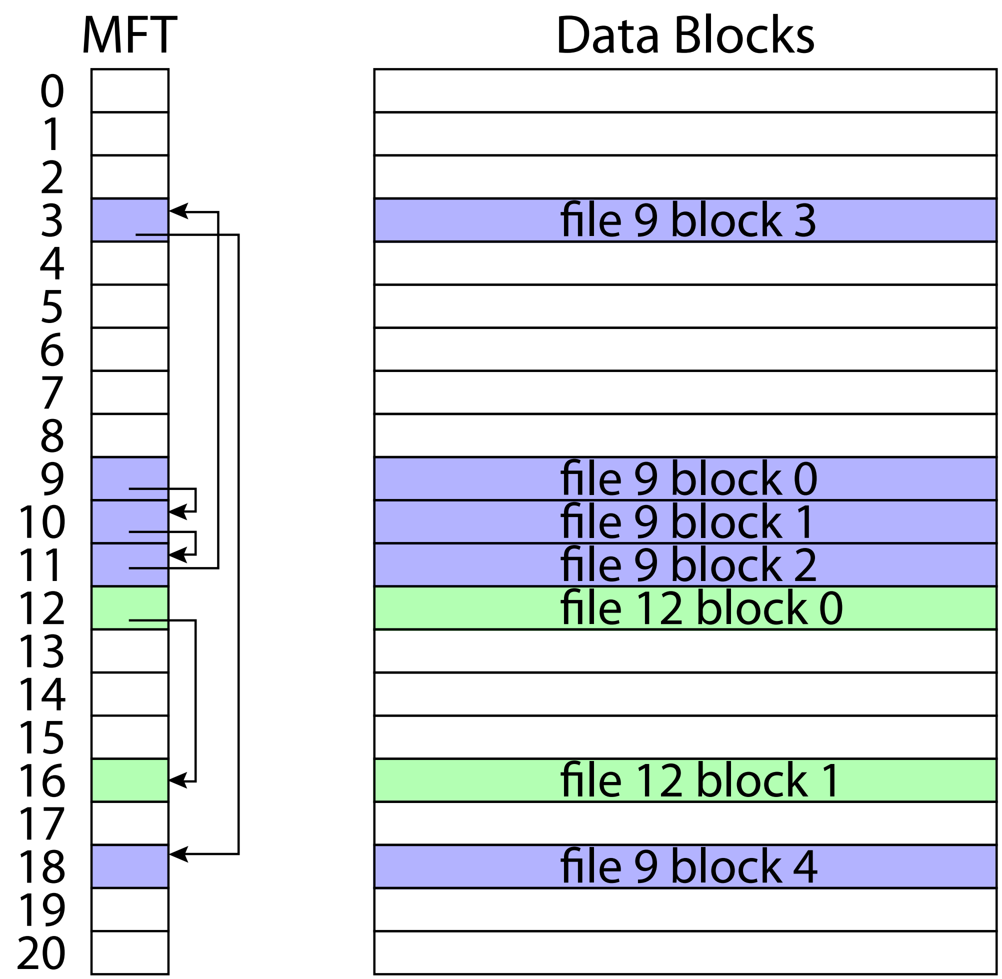
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed

- Approach taken in FAT, FFS (fsck), and many app-level recovery schemes (e.g., Word)

Reliability Attempt #1: Careful Ordering

<u>FAT: Append Data to File</u>

- Add data block
- Add pointer to data block
- Update file tail to point to new MFT entry
- Update access time at head of file



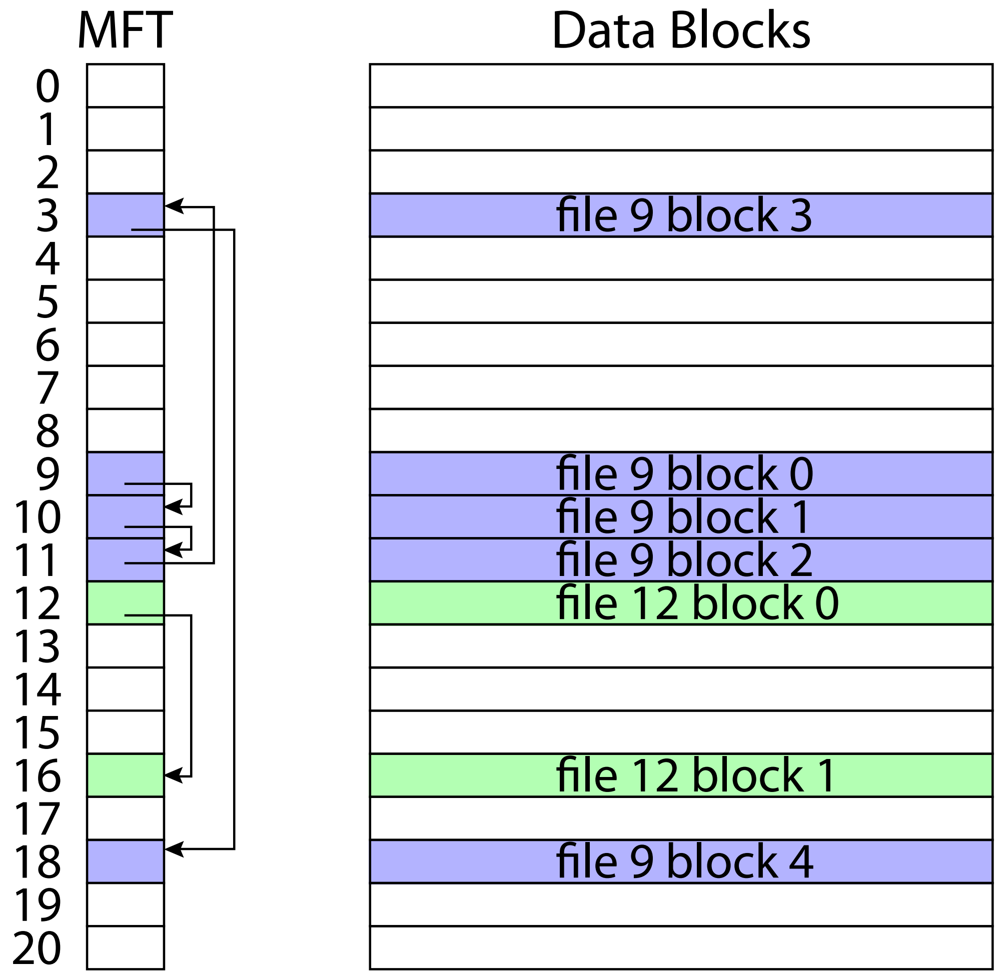
Reliability Attempt #1: Careful Ordering

FAT: Append Data to File

- Add data block
- Add pointer to data block
- Update file tail to point to new MFT entry
- Update access time at head of file

Recovery

- Scan MFT
- If entry is unlinked, delete data block
- If access time is incorrect, update

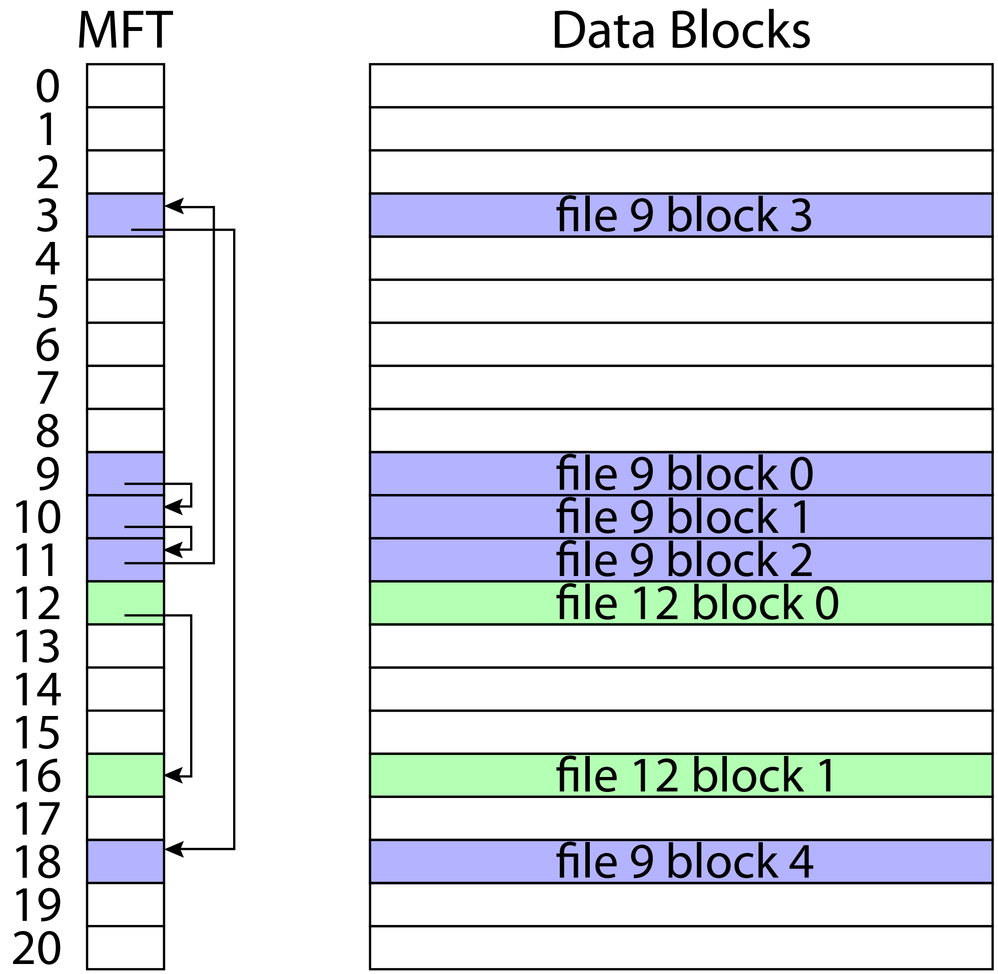


Reliability Attempt #1: Careful Ordering

FAT: Create New File

- Allocate data block
- Update MFT entry to point to data block
- Update directory with

  file name -> file number
- What if directory spans multiple disk blocks?
- Update modify time for directory



Reliability Attempt #1: Careful Ordering

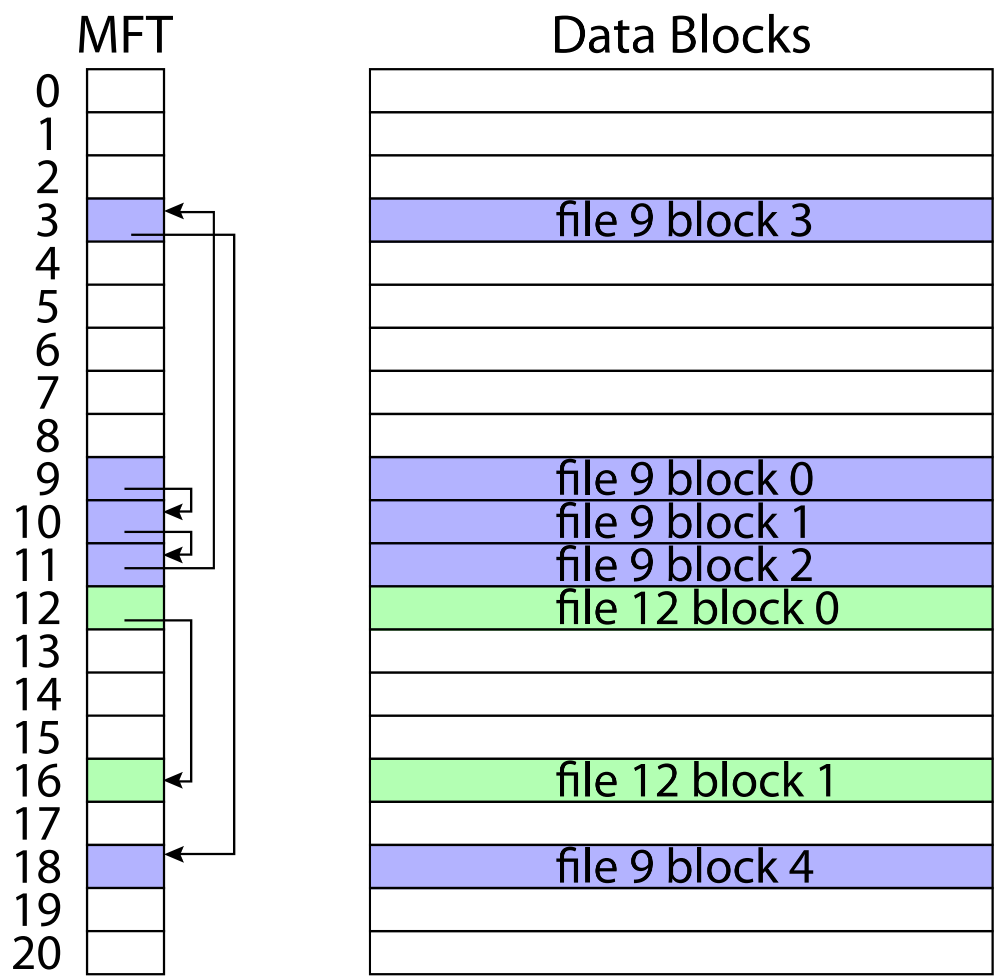
<u>FAT: Create New File</u>

- Allocate data block
- Update MFT entry to point to data block
- Update directory with

  file name -> file number
- What if directory spans multiple disk blocks?
- Update modify time for directory

<u>Recovery</u>

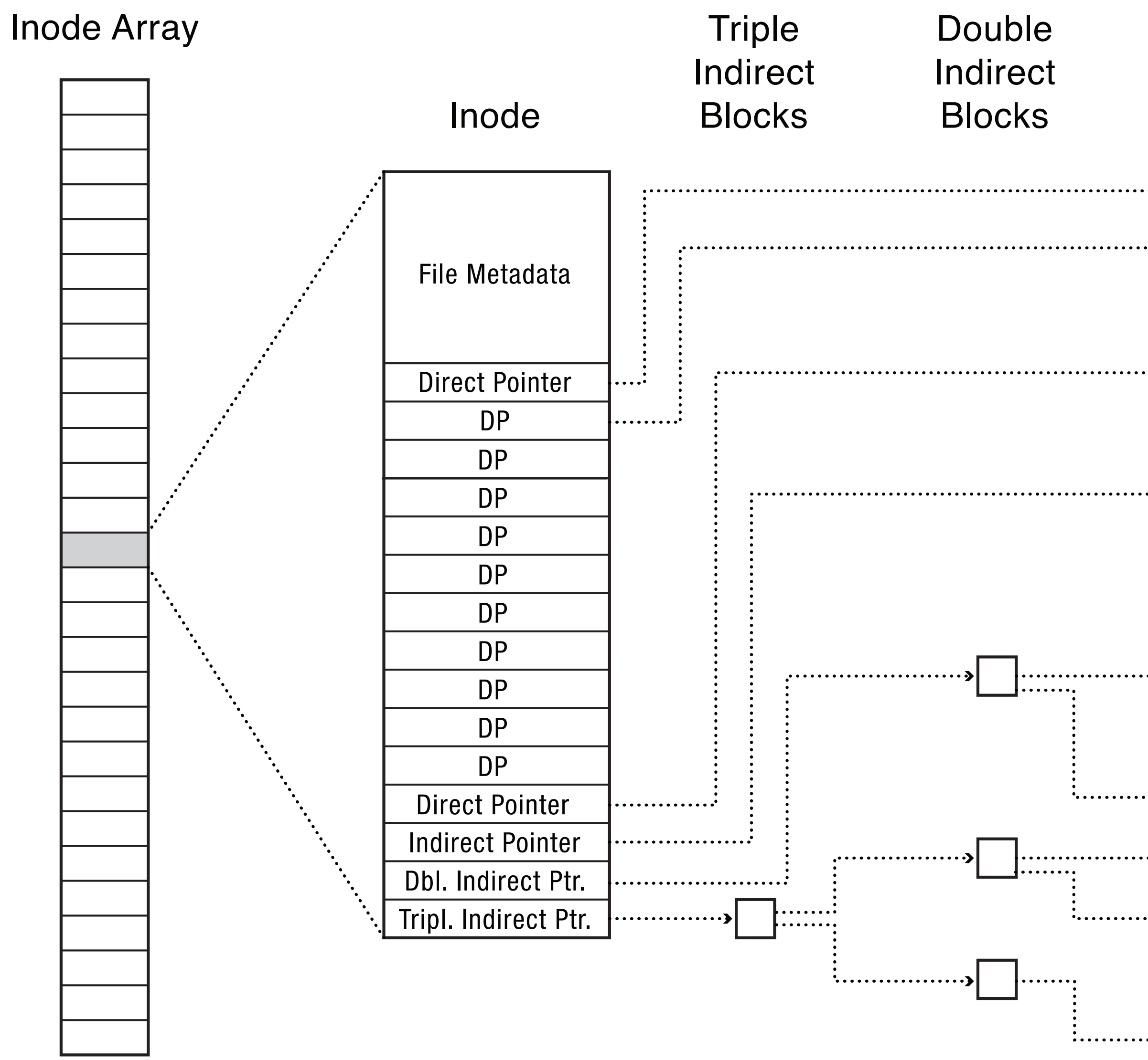
- Scan MFT
- If any unlinked files (not in any directory), delete
- Scan directories for missing update times



Reliability Attempt #1: Careful Ordering

FFS: Create New File

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks
- Update directory with
  file name -> file number
- Update modify time for directory



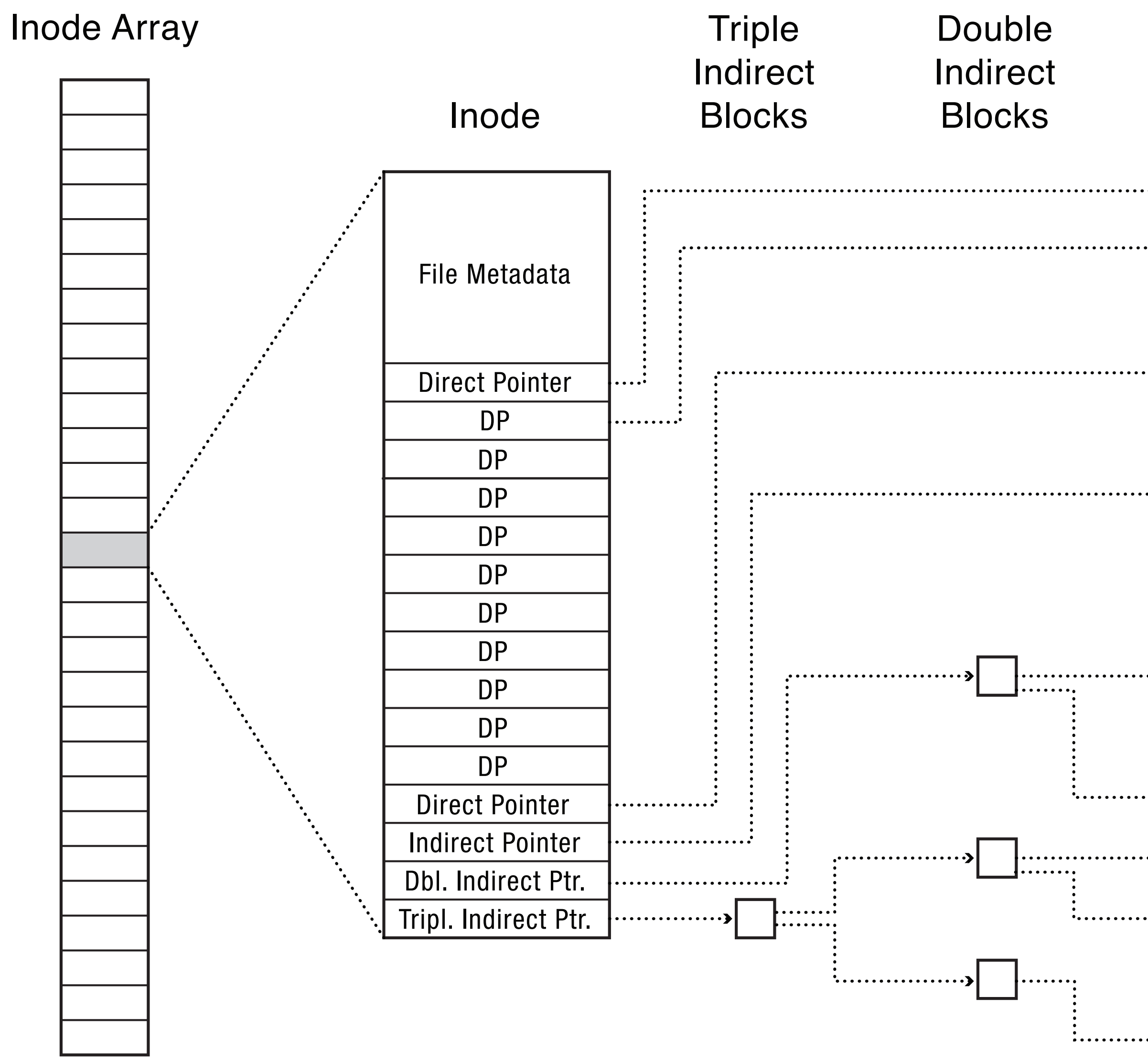
Reliability Attempt #1: Careful Ordering

FFS: Create New File

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks
- Update directory with
  file name -> file number
- Update modify time for directory

Recovery

- Scan inode table
- If any unlinked files (not in any directory), delete
- Compare free block bitmap against inode trees
- Scan directories for missing update/ access times

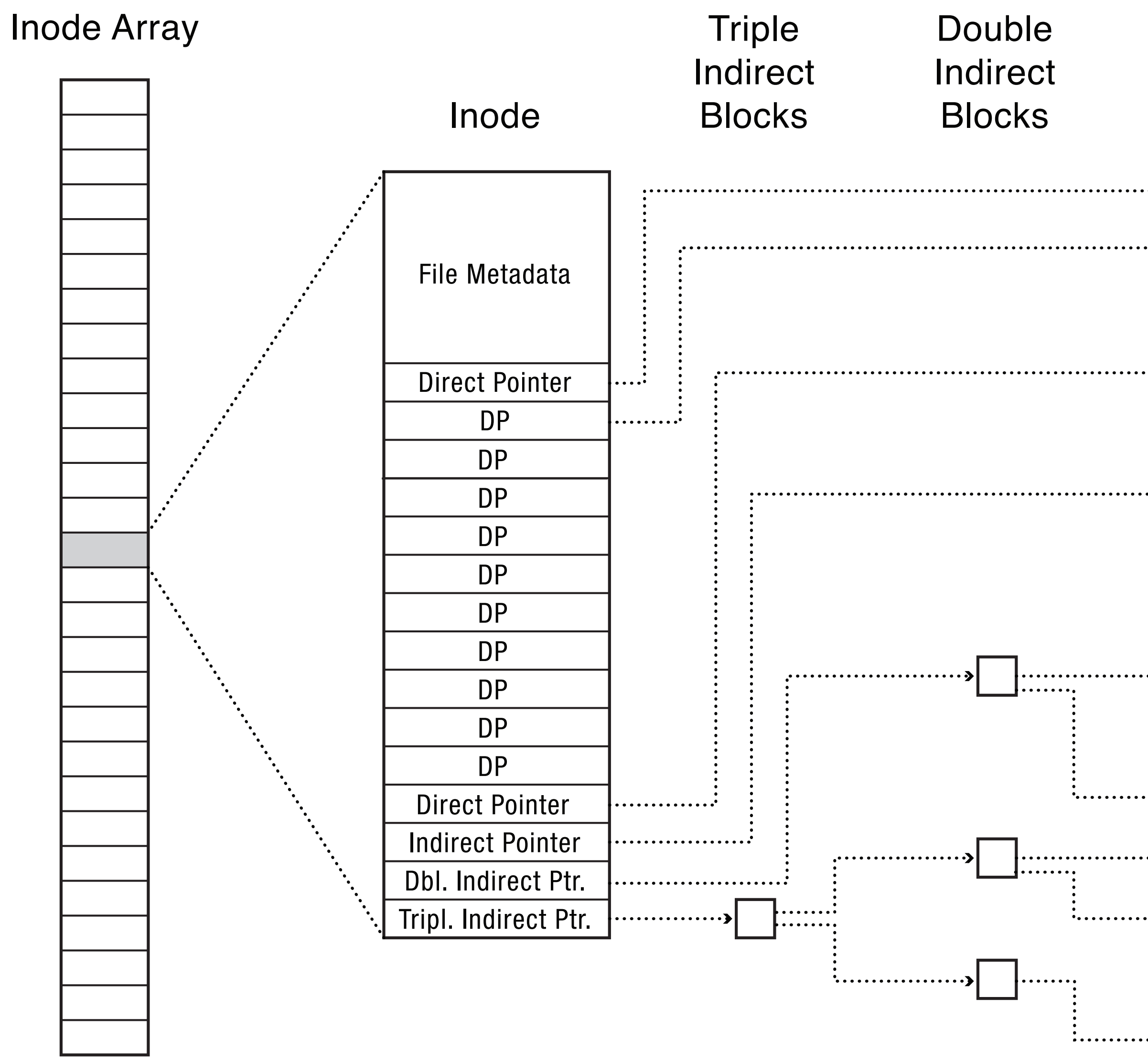


Recovery time is proportional to size of disk!

Reliability Attempt #1: Careful Ordering

FFS: Move a File

- Remove filename from old directory
- Add filename to new directory



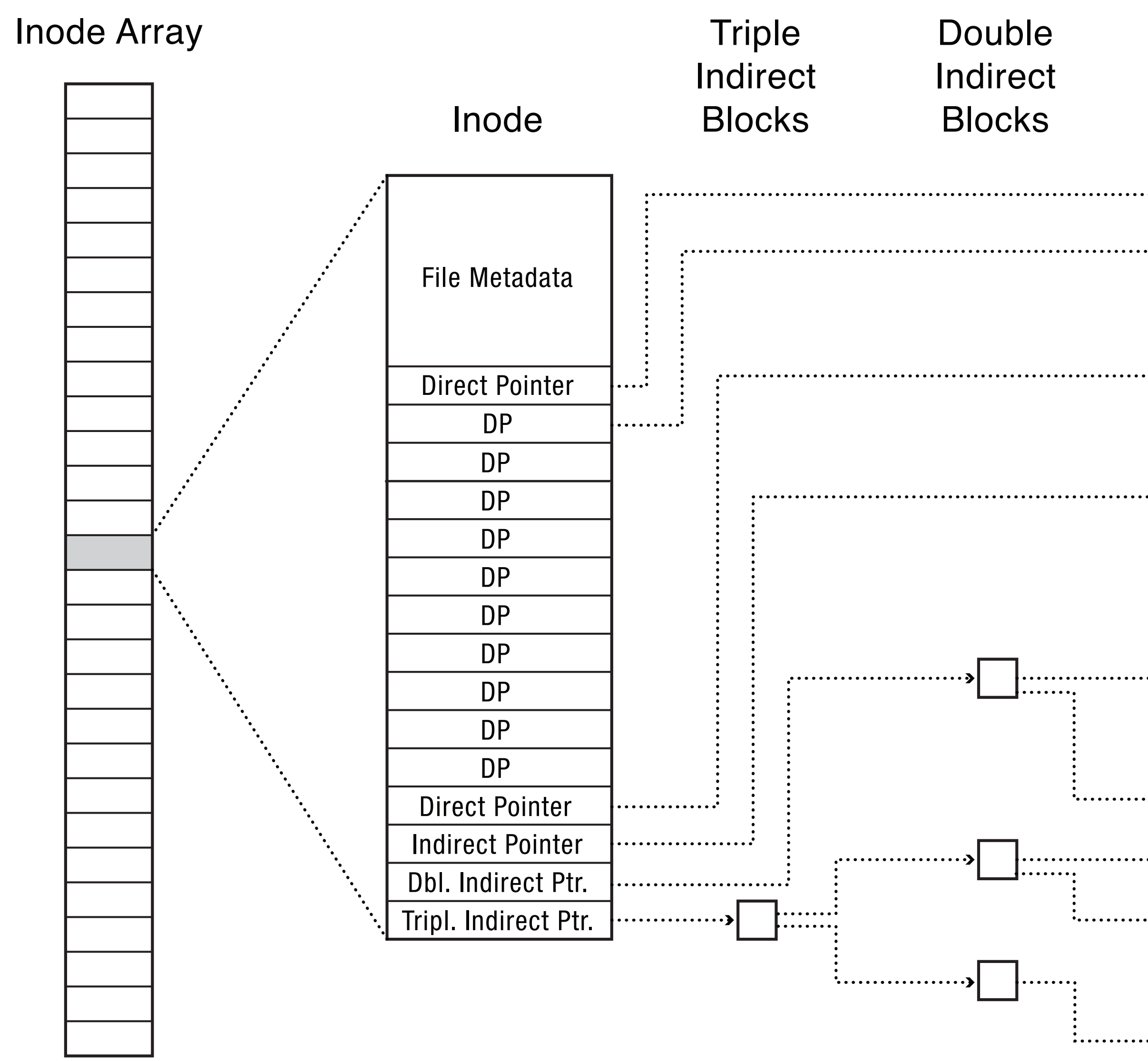
Reliability Attempt #1: Careful Ordering

FFS: Move a File

- Remove filename from old directory
- Add filename to new directory

Recovery

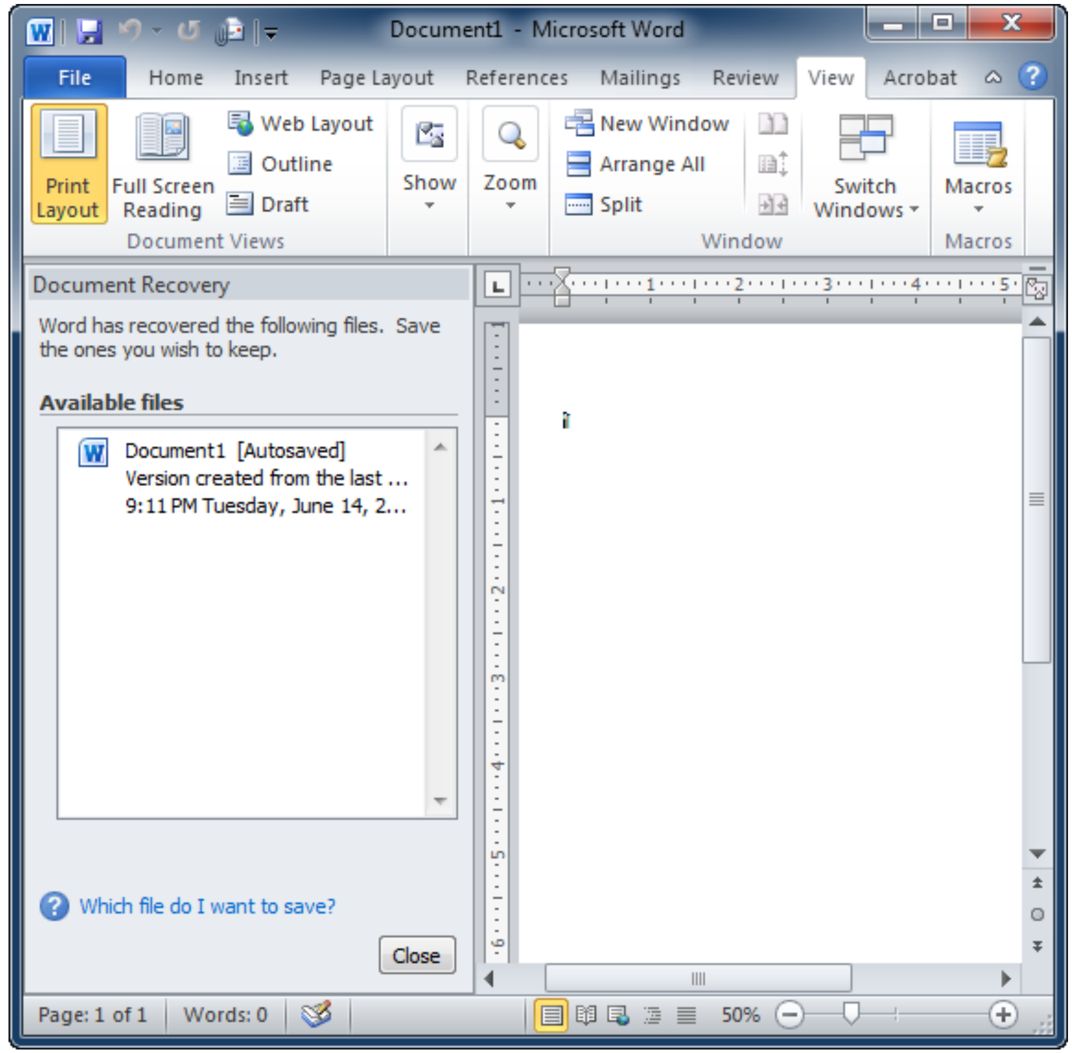
- Scan all directories to determine set of live files
- Consider files with valid inodes and not in any directory
  - New file being created?
  - File move?
  - File deletion?



Reliability Attempt #1: Careful Ordering

<u>Application Level</u>

- Write name of each open file to app folder
- Write changes to backup file
- Rename backup file to be file (atomic operation provided by file system)
- Delete list in app folder on clean shutdown



<u>Recovery</u>

- On startup, see if any files were left open
- If so, look for backup file
- If so, ask user to compare versions

Reliability Attempt #1: Careful Ordering

FFS: Move and Grep

- Observation — careful ordering is not a panacea…

Process A moves file from x to y

```
mv x/file y/
```

Process B greps across x and y

```
grep x/* y/*
```

- Will Process B always see the contents of the file?

Reliability Attempt #1: Careful Ordering

Pros

- Works with minimal support from the disk drive
- Works for most multi-step operations

Cons

- Can require time-consuming recovery after a failure
- Difficult to reduce every operation to a safely-interruptible sequence of writes
- Difficult to achieve consistency when multiple operations occur concurrently (e.g., FFS grep)

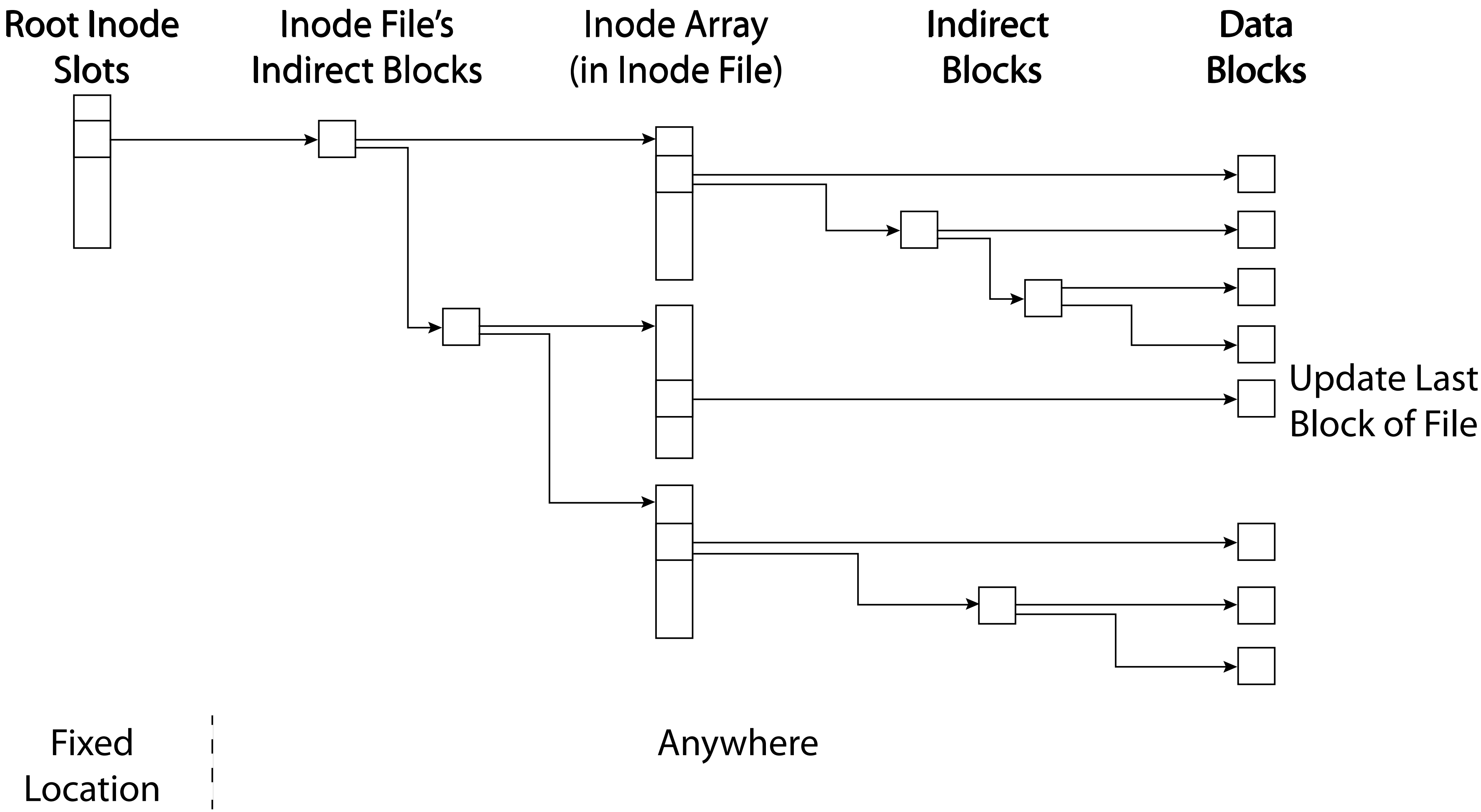
Reliability Attempt #2: Copy-on-Write

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks

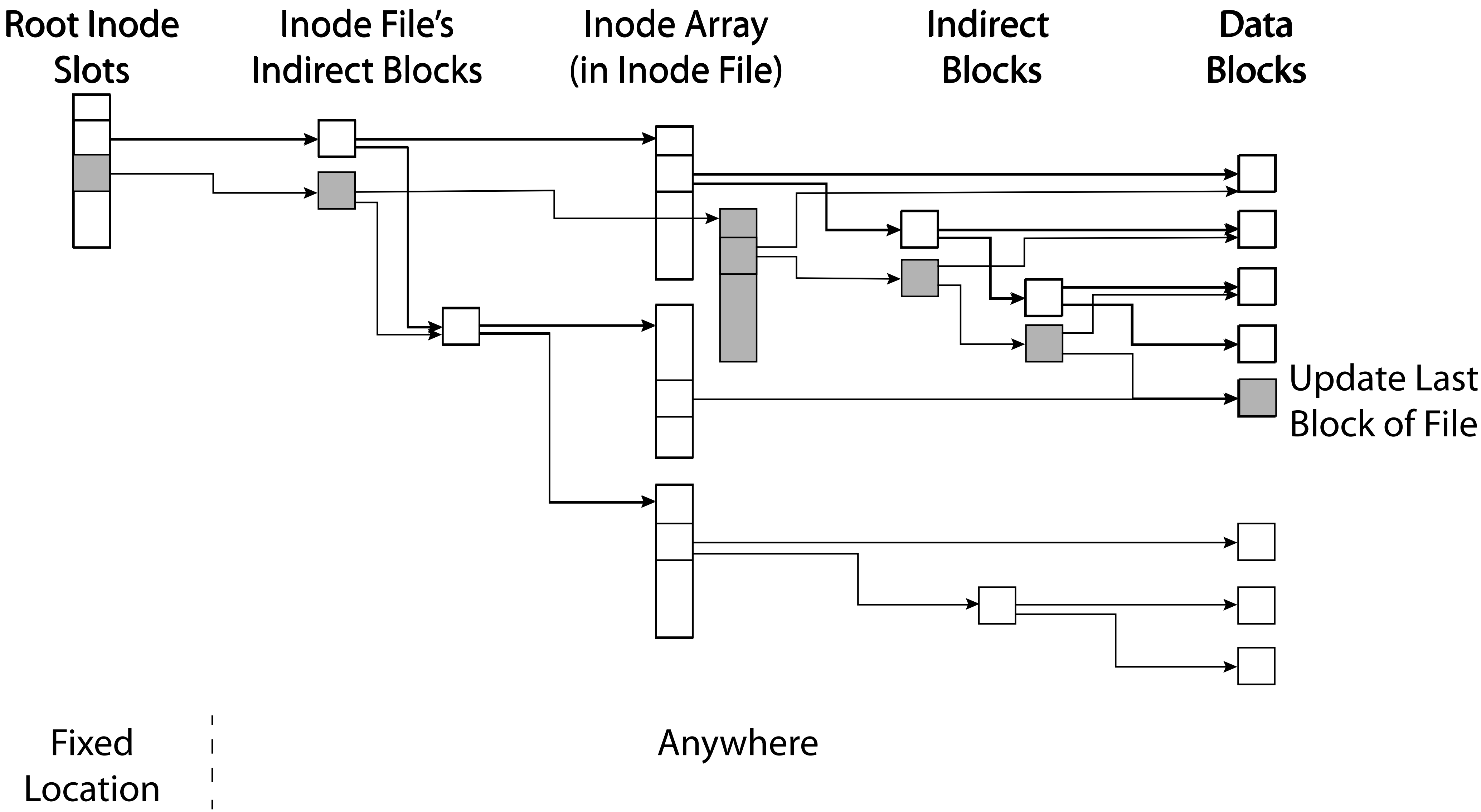
- Seems expensive! But…
  - Updates can be batched
  - Almost all disk writes can occur in parallel

- Approach taken in network file server appliances (WAFL, ZFS)

Copy on Write (Write Anywhere File Layout)

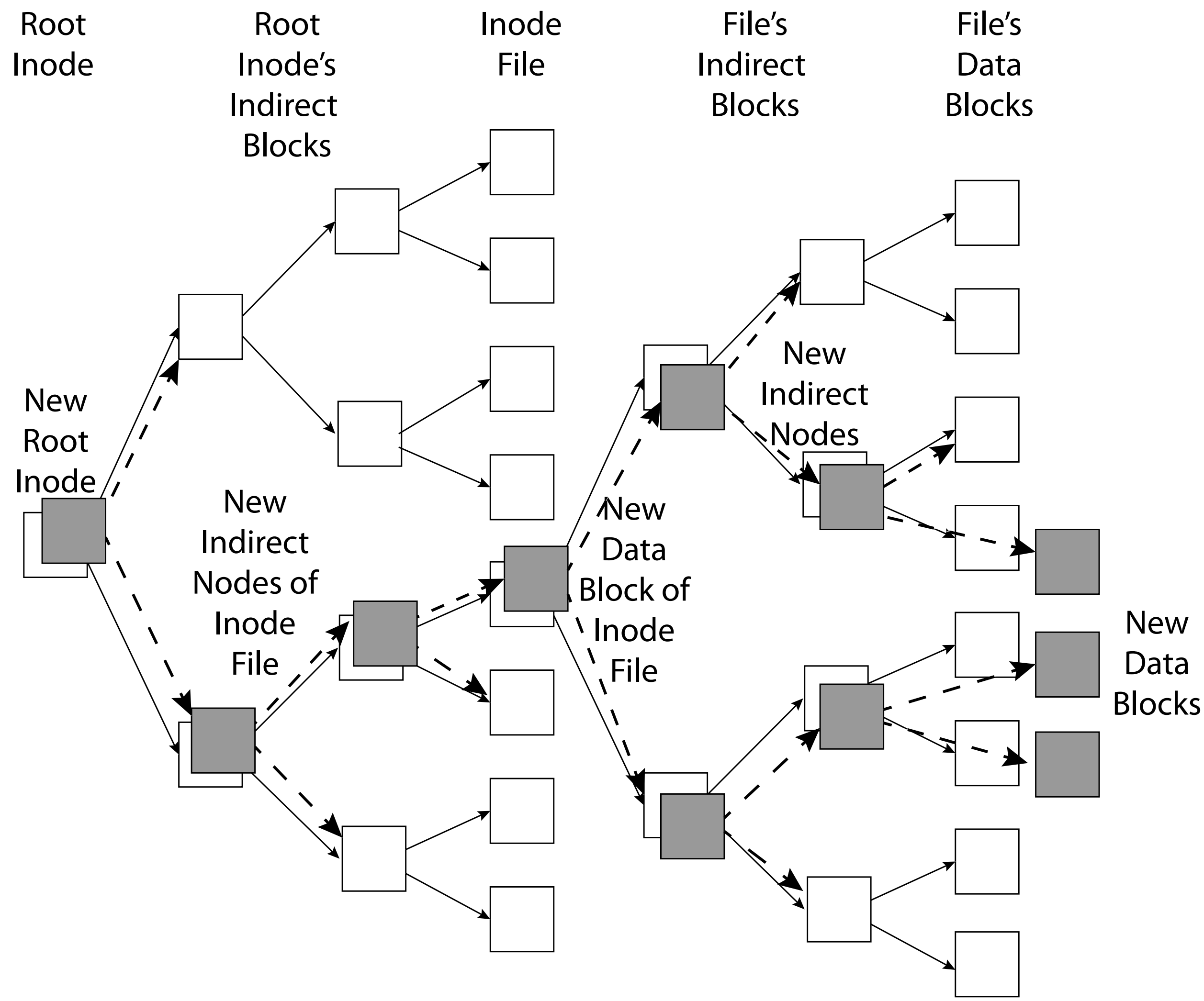


Copy on Write (Write Anywhere File Layout)

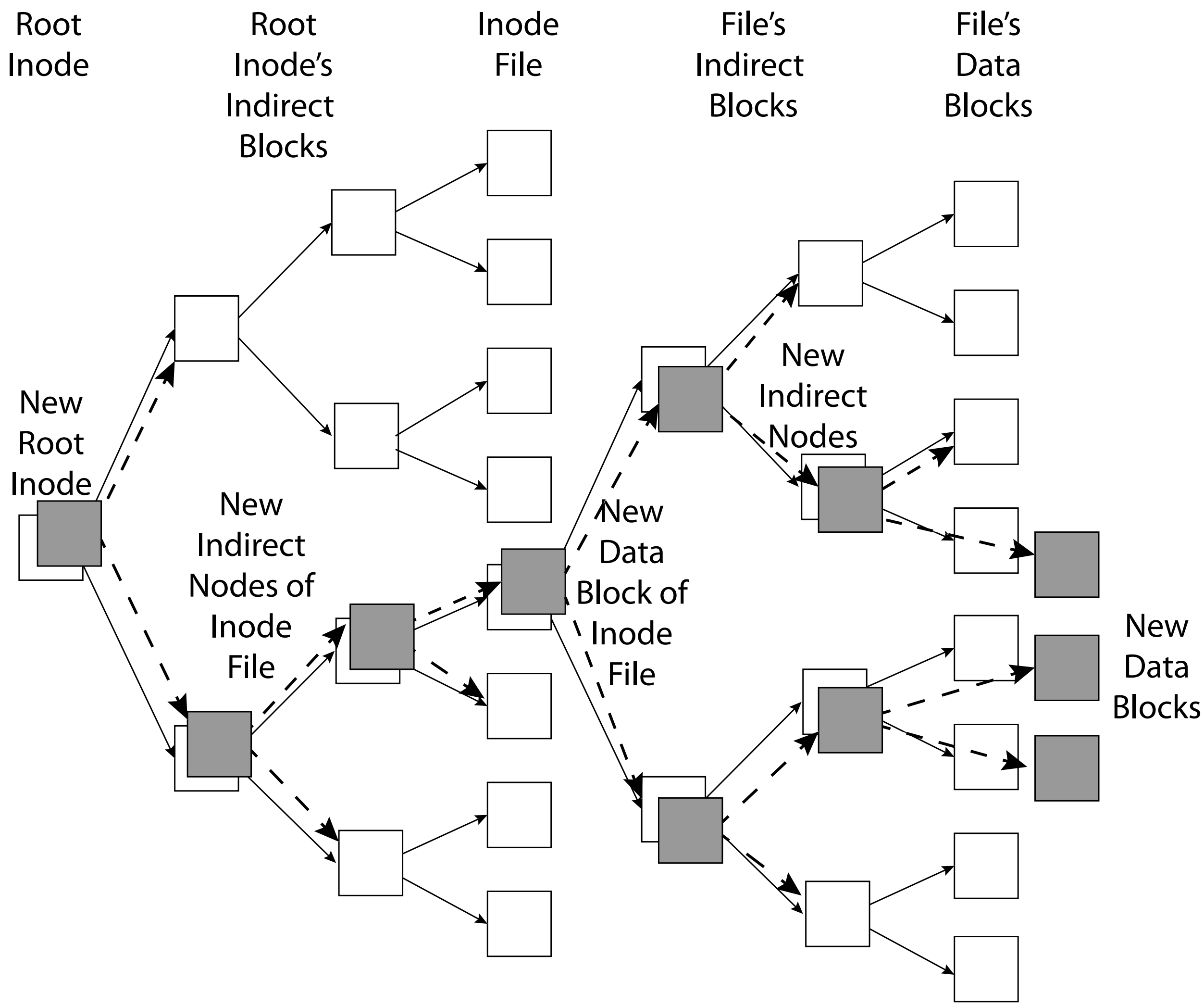


Reliability Attempt #2: Copy-on-Write

Batch Updates



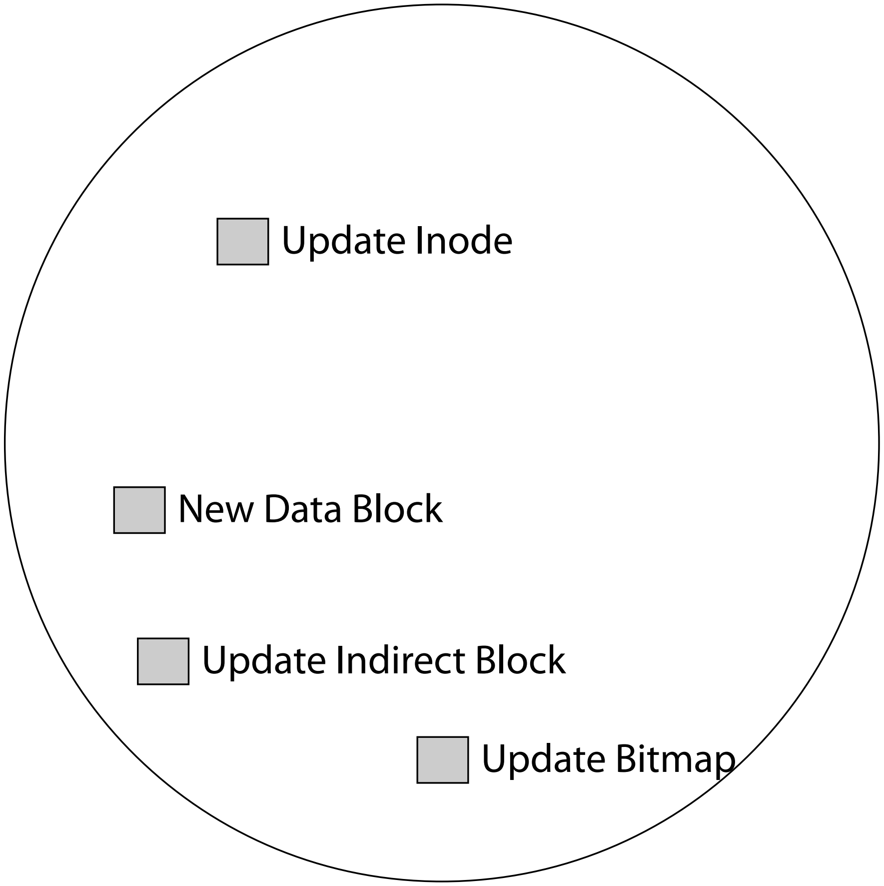
Batch Update



Reliability Attempt #2: Copy-on-Write

FFS Updates

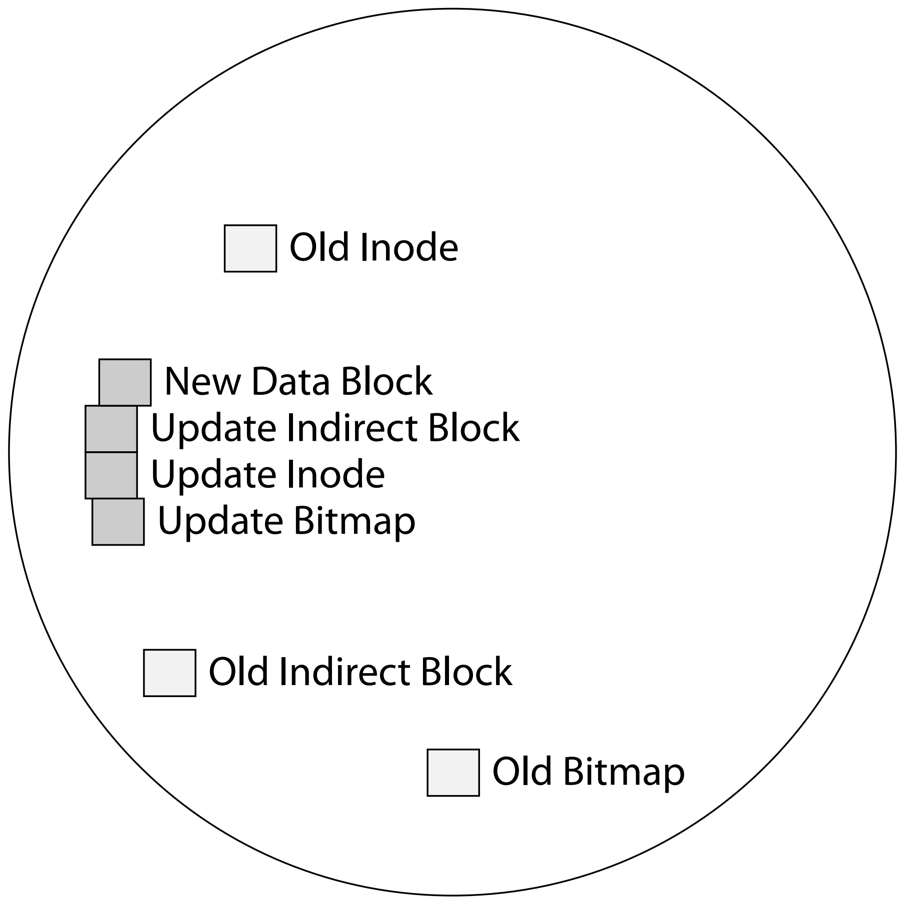
(updates are in-place)



Reliability Attempt #2: Copy-on-Write

<u>Write Anywhere File Layout (WAFL) Updates</u>

(Uses Copy-on-Write)



Reliability Attempt #2: Copy-on-Write

<u>Garbage Collection</u>

- For write efficiency, want contiguous sequences of free blocks
  - Spread across all block groups
  - Updates leave dead blocks scattered

- For read efficiency, want data read together to be in the same block group
  - Write anywhere leaves related data scattered

- Solution? Background coalescing of live/dead blocks

Reliability Attempt #2: Copy-on-Write

Pros
- Correct behavior regardless of failures
- Fast recovery (root block array)
- High throughput (best if updates are batched)

Cons
- Potential for high latency
- Small changes require many writes
- Garbage collection essential for performance

Logging File Systems

- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is append-only

- Once changes are on log, safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended

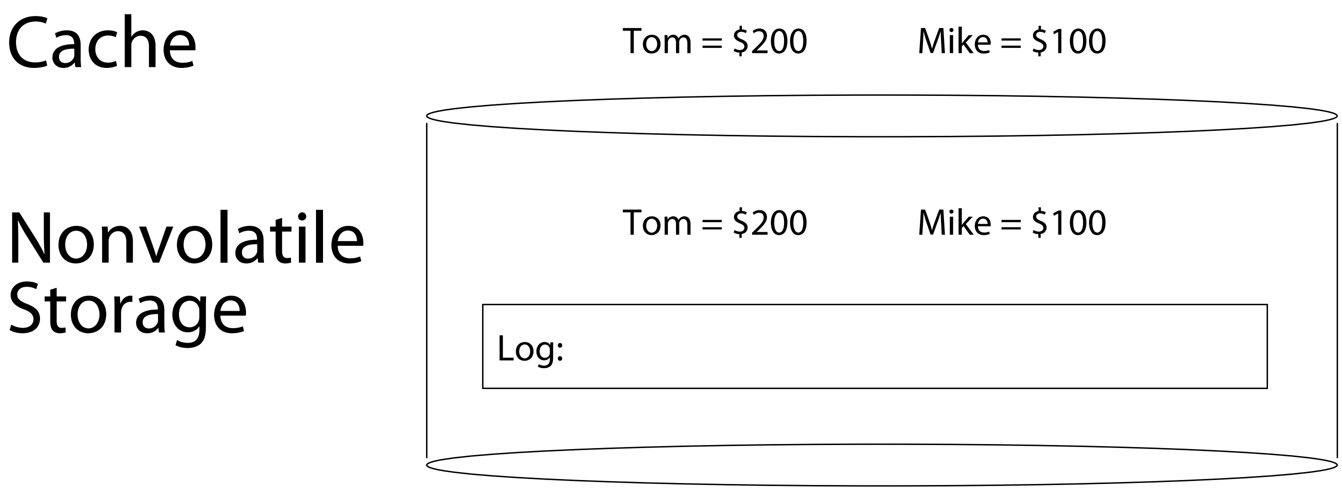
- Once changes are copied, safe to remove log

Redo Logging

- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

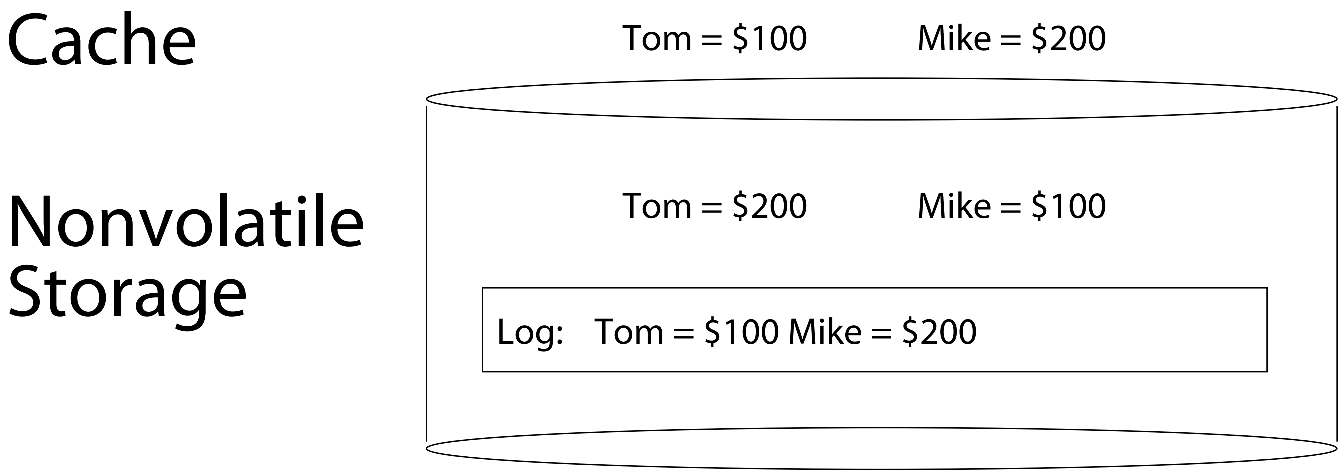
Redo Logging

Before transaction start



Redo Logging

After Updates are Logged

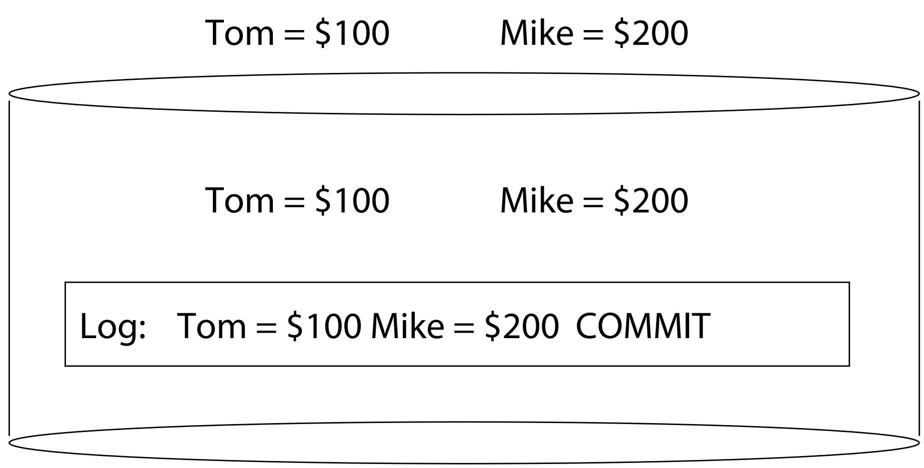


Redo Logging

After commit logged

Cache

Nonvolatile Storage



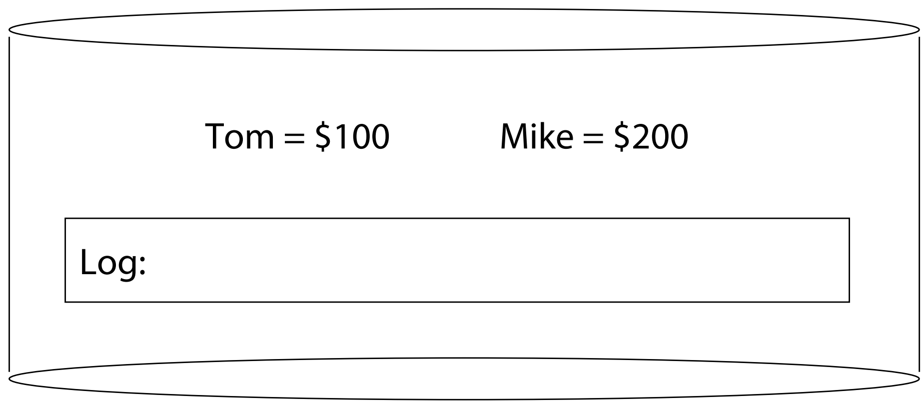
Redo Logging

After copy back

Cache

Tom = $100 Mike = $200

Nonvolatile Storage

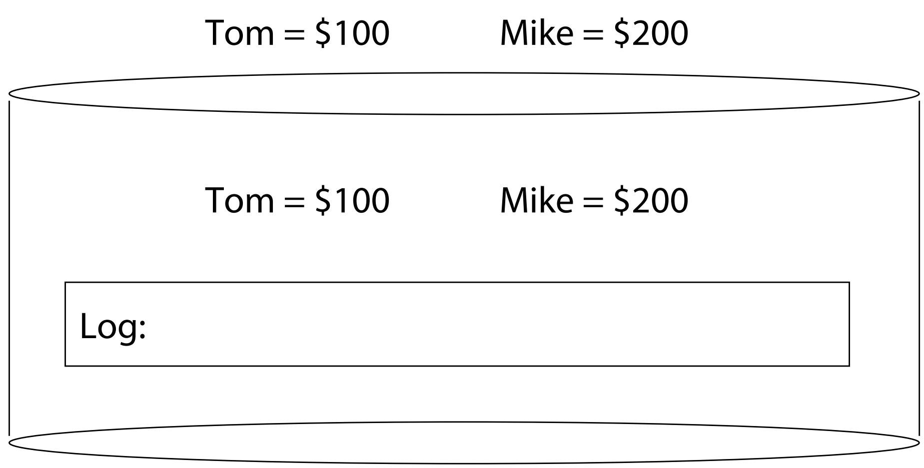


Redo Logging

<u>After garbage collection</u>

Cache

Nonvolatile Storage



Redo Logging

<u>Questions</u>

- What happens if machine crashes?
  - Before transaction start
  - After transaction start, before operations are logged
  - After operations are logged, before commit
  - After commit, before write back
  - After write back before garbage collection
- What happens if machine crashes during recovery?

Redo Logging

Performance

- Log written sequentially
  - Often kept in flash storage
- Asynchronous write back
  - Any order as long as all changes are logged before commit, and all write backs occur after commit
- Can process multiple transactions
  - Transaction ID in each log entry
  - Transaction completed iff its commit record is in log

Transaction Isolation

- What if grep starts after changes are logged but before they are commited?

Process A moves file from x to y

```
mv x/file y/
```

Process B greps across x and y

```
grep x/* y/*
```

Transaction Isolation

- What if grep starts after changes are logged but before they are commited?

Process A moves file from x to y

```
mv x/file y/
```

Process B greps across x and y

```
grep x/* y/*
```

- Two Phase Locking: Release locks only AFTER transaction commit.
  - Prevents a process from seeing results of a transaction that might not commit!

Process A moves file from x to y

```
Lock x, y
mv x/file y/
Commit & Release x, y
```

Process B greps across x and y

```
Lock x, y
grep x/* y/*
Release x, y
```

Serializability

- With two phase locking and redo logging, transactions appear to occur in a sequential order (serializability)
  - Either: grep then move or move then grep

- Other implementations can also provide serializability
  - e.g., <u>Optimistic concurrency control</u>: abort any transaction that would conflict with serializability

Storage Availability

- Storage reliability: data fetched is what you stored
  - Transactions, redo logging, etc.
- Storage availability: data is there when you want it
  - More disks => higher probability of some disk failing
  - Data available ~ Prob(disk working)^k
    - If failures are independent and data is spread across k disks
  - For large k, probability system works -> 0

RAID

"Redundant Array of Inexpensive Disks"

- Multiple disk drives provide reliability via **redundancy**.
- **Speeds up access times** even beyond sequential.
- Increases the **mean time to failure**

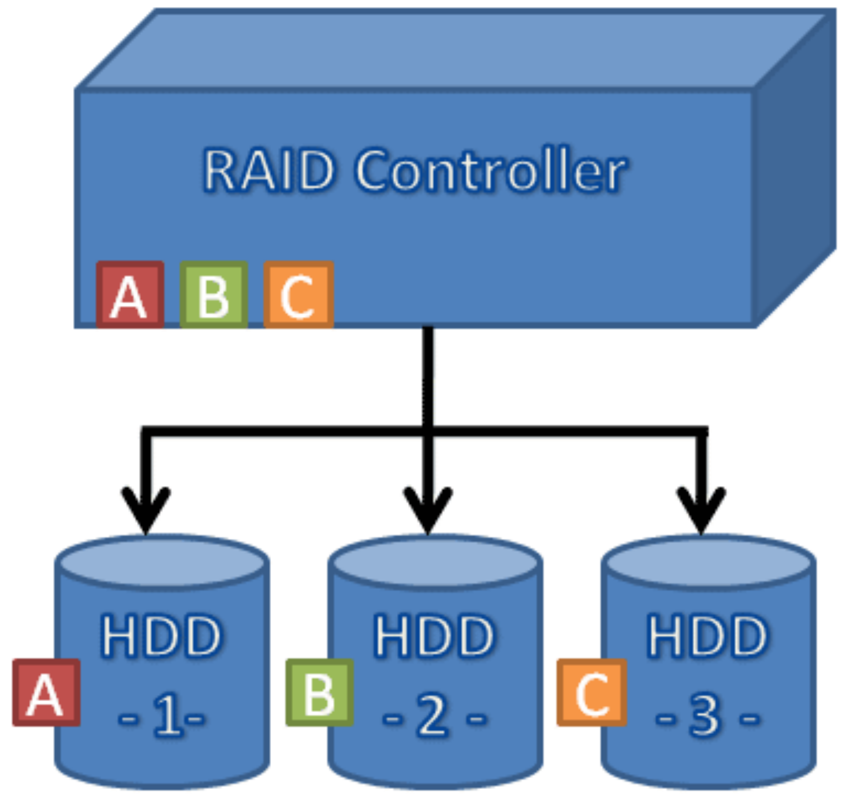


RAID

- RAID
  - multiple disks work cooperatively
  - Improve reliability by storing redundant data
  - **Striping** (**RAID 0**) improves performance with disk **striping** (use a group of disks as one storage unit)
  - **Mirroring** (**RAID 1**) keeps duplicate of each disk
  - Striped mirrors (**RAID 1+0**) or mirrored stripes (**RAID 0+1**) provides high performance and high reliability
  - **Block interleaved parity** (**RAID 4, 5, 6**) uses much less redundancy

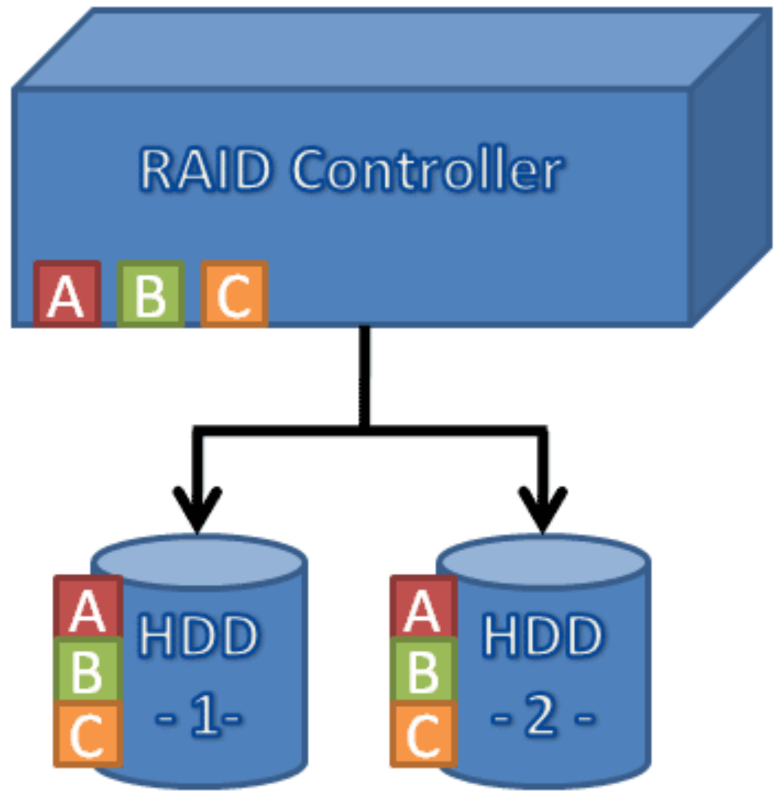
RAID Level 0

- Level 0 is nonredundant disk array
- Files are striped across disks, no redundant info
- High read throughput
- Best write throughput (no redundant info to write)
- Any disk failure results in data loss



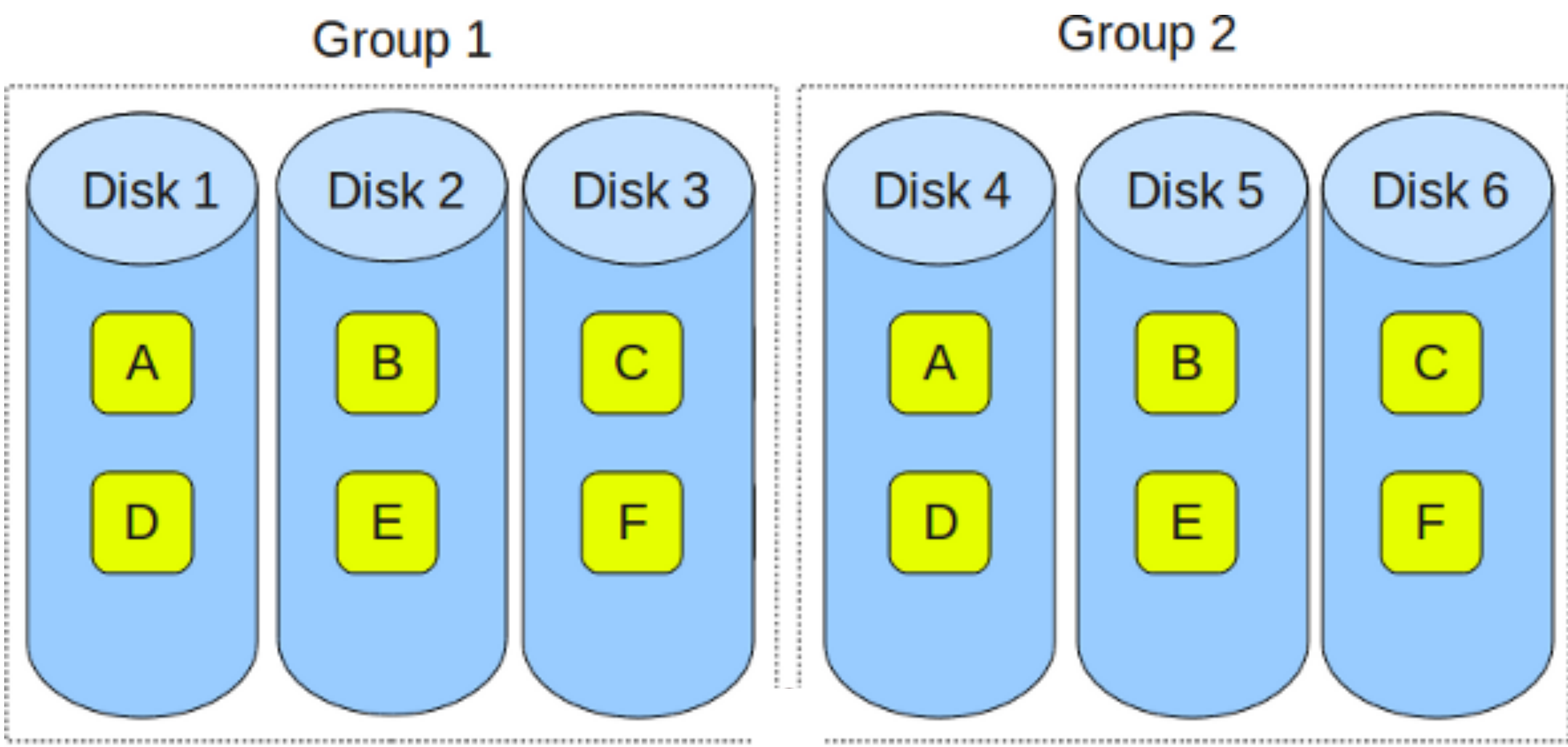
RAID Level 1

- Mirrored Disks
- Data is written to two places
  - On failure, just use surviving disk (easy to rebuild)
- On read, choose fastest to read
  - Write performance is same as single drive, read performance is 2x better
- Expensive (high space overhead)



RAID Level 0+1

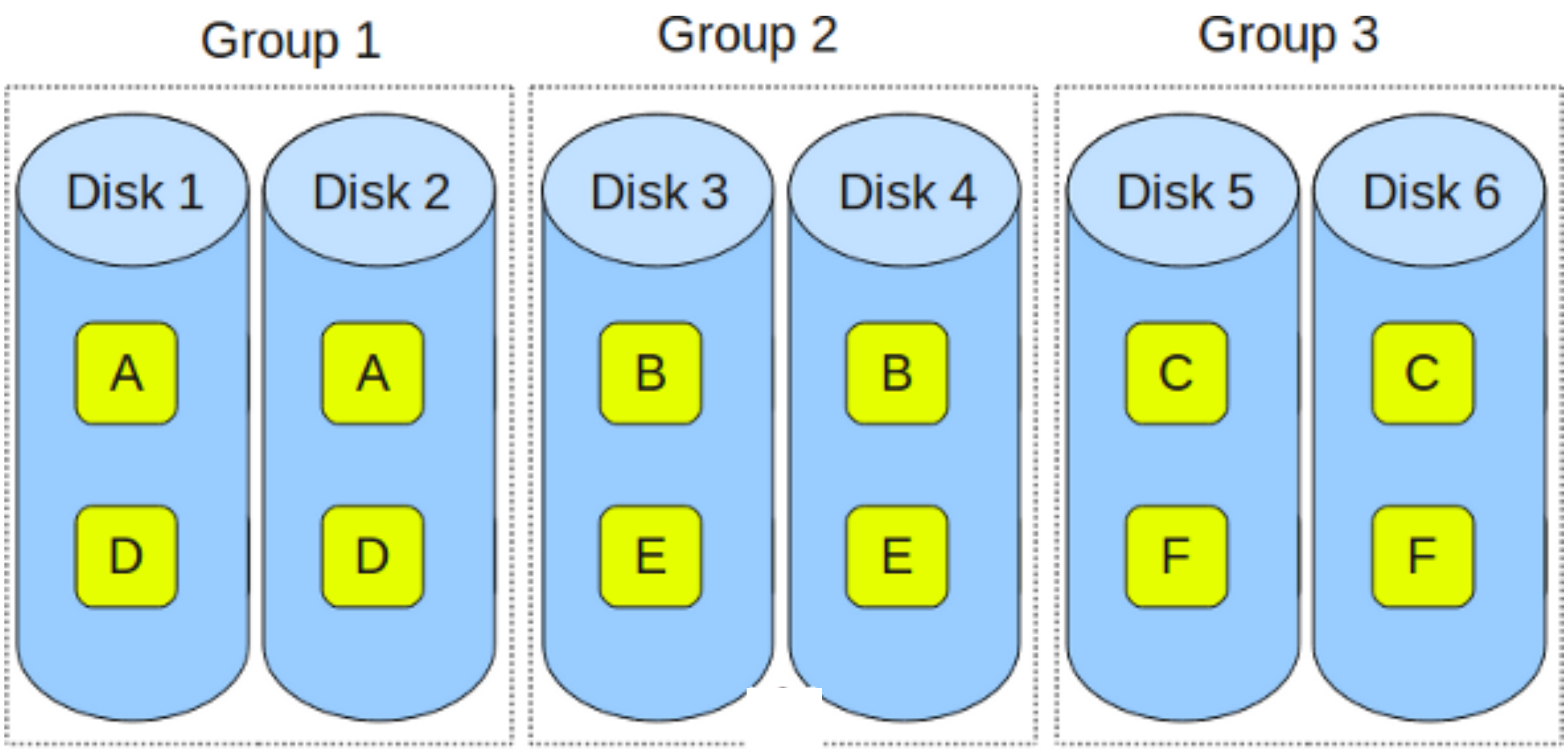
- Stripe on a set of disks
- Then mirror of data blocks is striped on the second set.



RAID 01 – Blocks Striped. (and Blocks Mirrored)

RAID Level 1+0

- Pair mirrors first.
- Then stripe on a set of paired mirrors



RAID 10 – Blocks Mirrored. (and Blocks Striped)