Transactions

Main Points

- Transaction concept
- Four approaches to implementing atomicity
    - Careful sequencing of operations
    - Copy-on-write (WAFL, ZFS)
    - Journalling (NTFS, linux ext4)
    - Log structure (flash storage)
- Two approaches to implementing consistency
    - Two-phase locking
    - Optimistic concurrency control

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 system wants 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, …
  - With remapping, single update to physical disk block can require multiple (even lower level) updates
- 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

- Transaction is a group of operations
  - Atomic: operations appear to happen as a group, or not at all (at logical level)
    - At physical level, only single disk/flash write is atomic
  - Durable: operations that complete stay completed
    - Future failures do not corrupt previously stored data
  - Isolation: other transactions do not see results of earlier transactions until they are committed
  - Consistency: sequential memory model

Reliability Approach #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)

FFS: Create a File

Normal operation:

- 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

Time proportional to size of disk

FFS: Move a File

Normal operation:

- 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?

FFS: Move and Grep

Process A

move file from x to y

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

Process B

grep across x and y

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

Will grep always see contents of file?

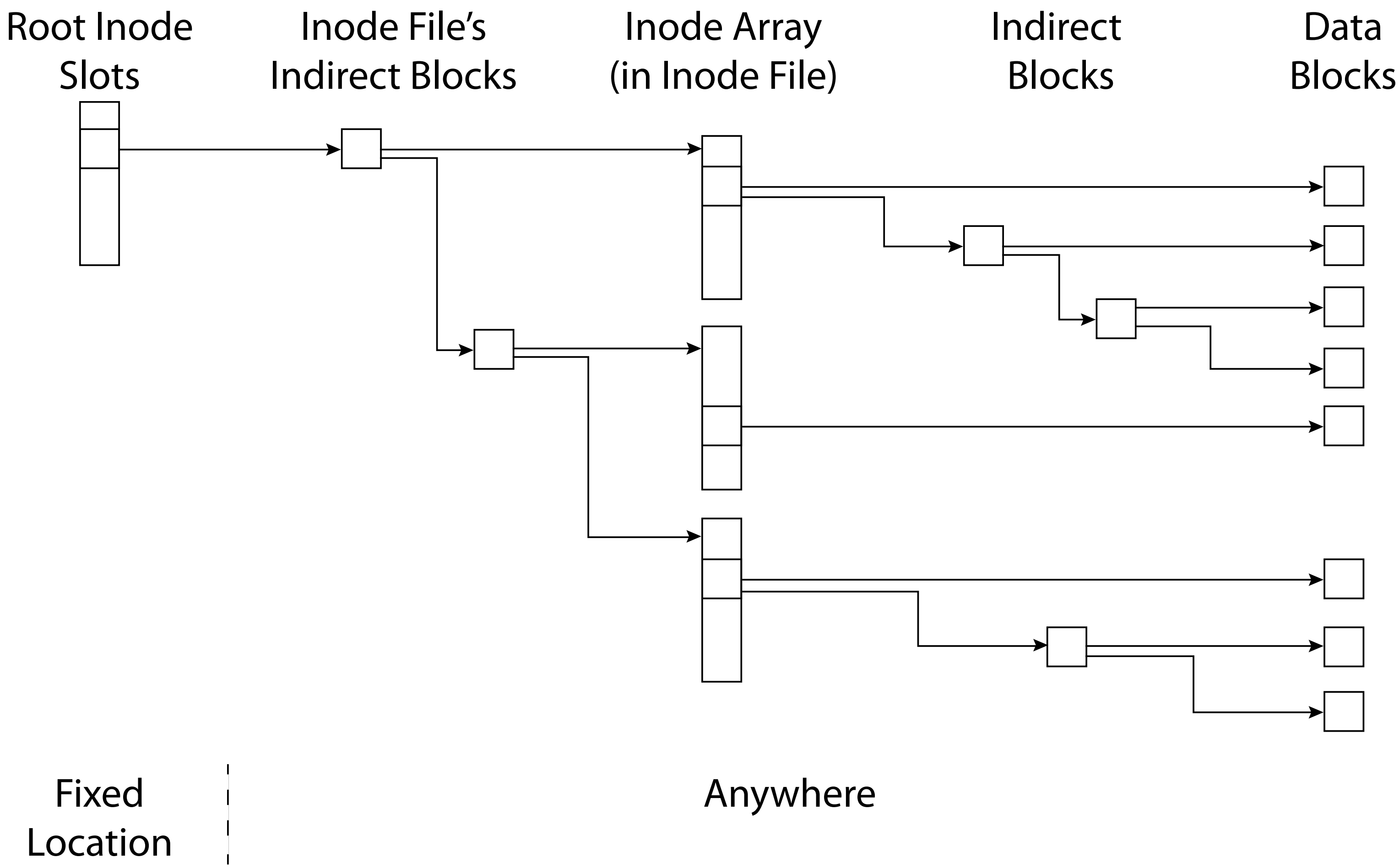
Careful Ordering

- Pros
  - Works with minimal support in 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

Reliability Approach #2: Copy on Write File Layout

- 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



Root Inode Slots

Inode File's Indirect Blocks

Inode Array (in Inode File)

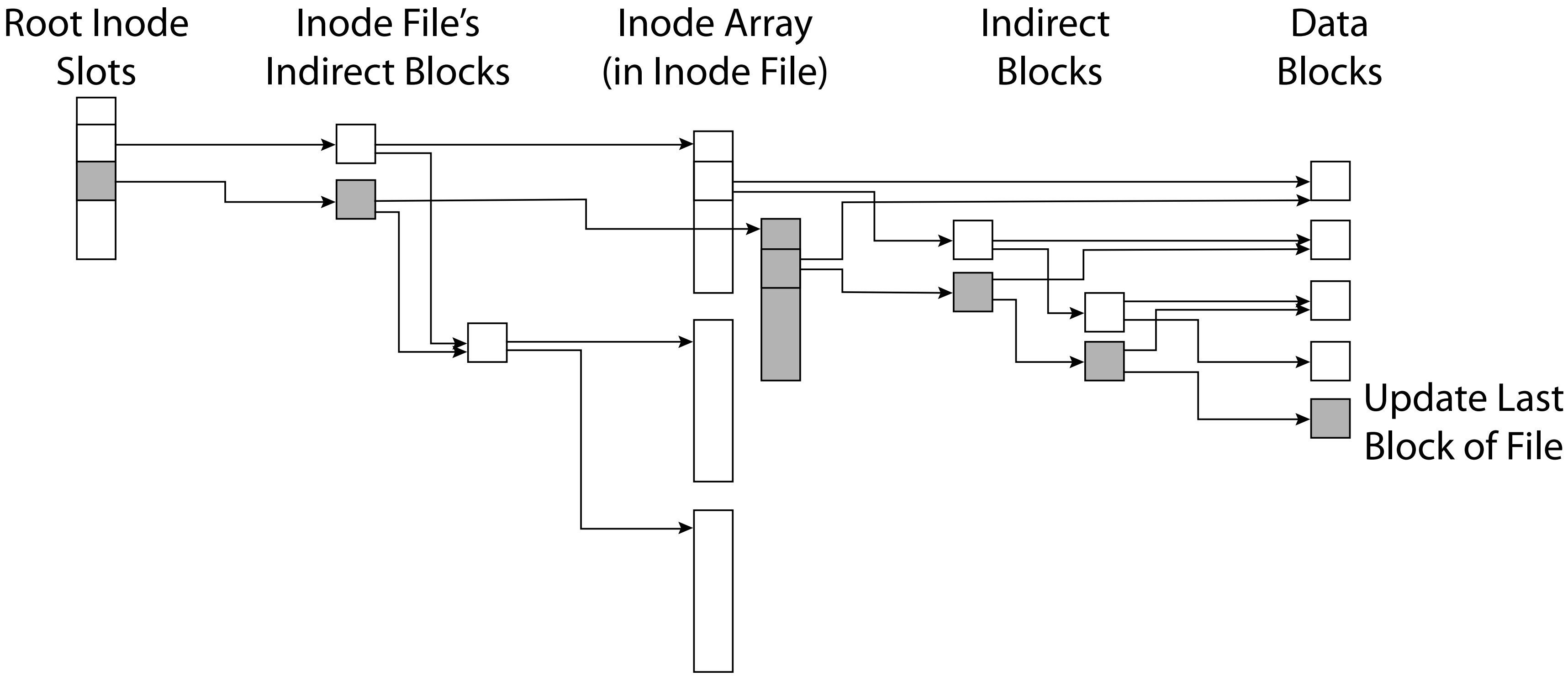
Indirect Blocks

Data Blocks

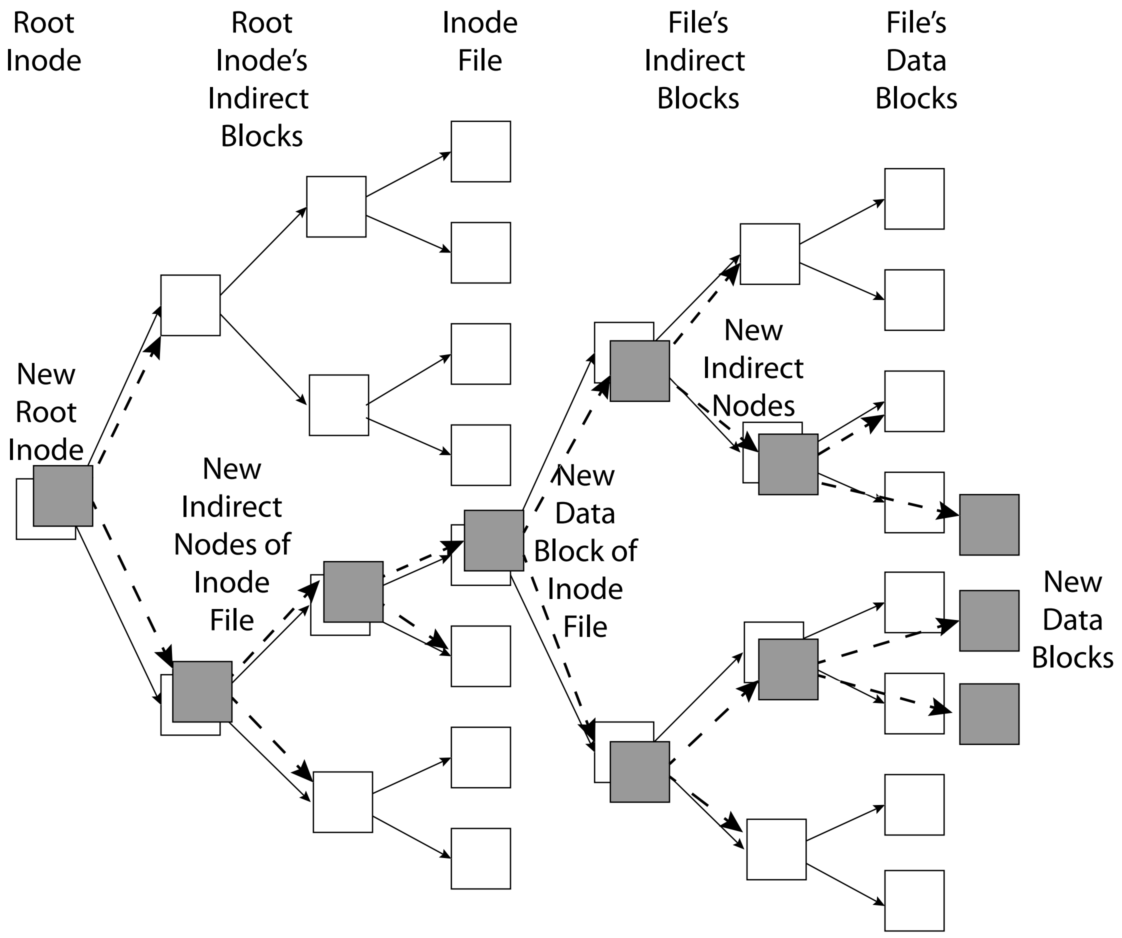
Fixed Location

Anywhere

Copy on Write/Write Anywhere



Copy on Write Batch Update



Copy on Write Garbage Collection

- 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

=> Background coalescing of live/dead blocks

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

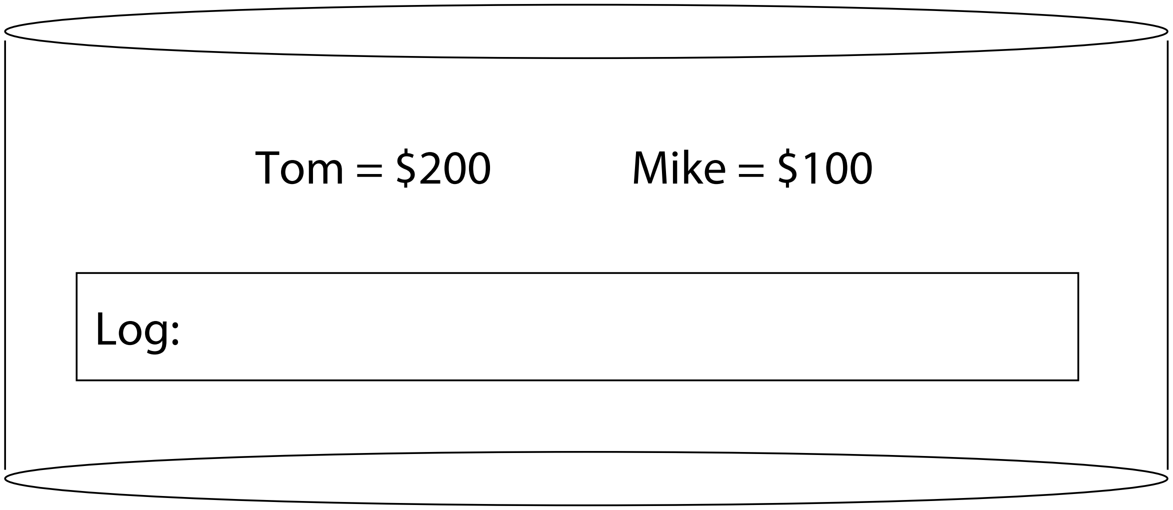
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

Before Transaction Start

Cache

Tom = $200 Mike = $100

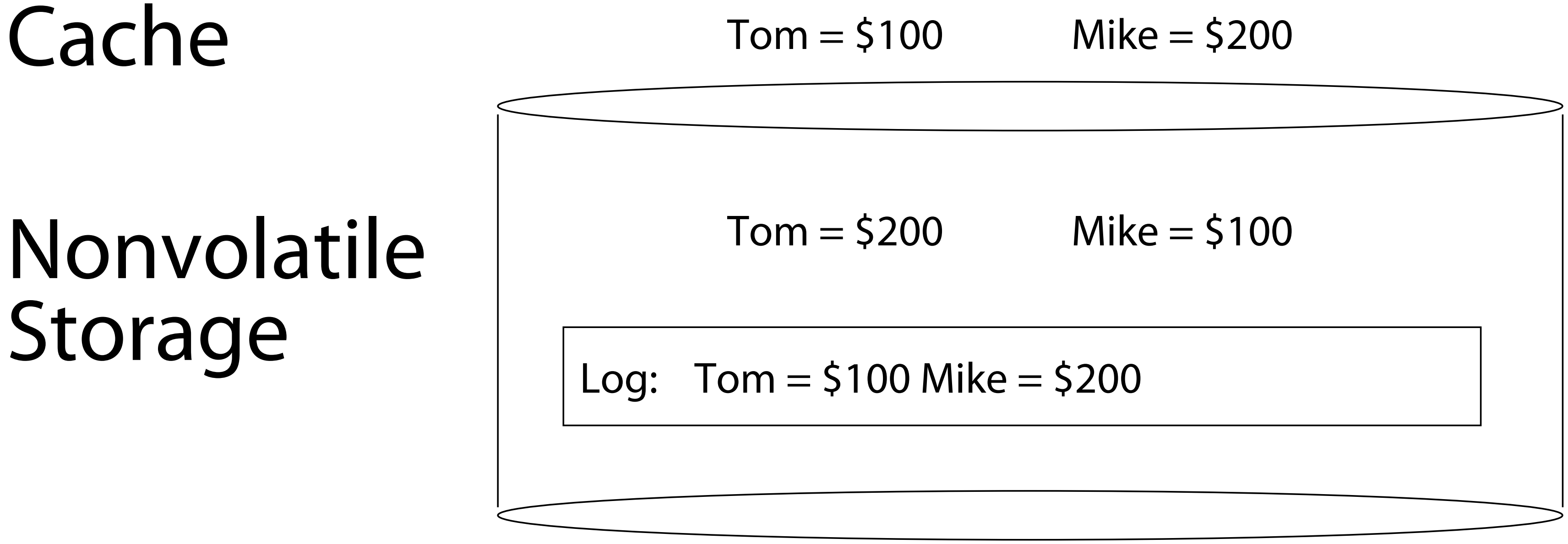
Nonvolatile Storage



Tom = $200 Mike = $100

Log:

After Updates Are Logged



Cache

Tom = $100 Mike = $200

Nonvolatile Storage

Tom = $200 Mike = $100

Log: Tom = $100 Mike = $200

After Commit Logged

Cache

Tom = $100 Mike = $200

Nonvolatile
Storage

Tom = $200 Mike = $100

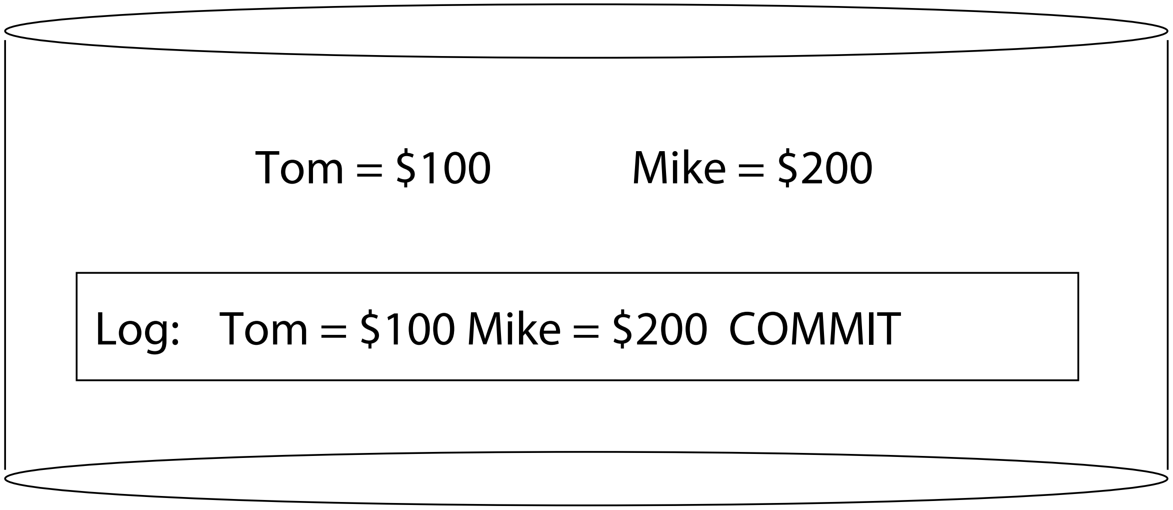
Log: Tom = $100 Mike = $200 COMMIT

After Copy Back

Cache

Tom = $100 Mike = $200

Nonvolatile Storage



Tom = $100 Mike = $200

Log: Tom = $100 Mike = $200 COMMIT

After Garbage Collection

Cache

Tom = $100 Mike = $200

Nonvolatile Storage

Tom = $100 Mike = $200

Log:

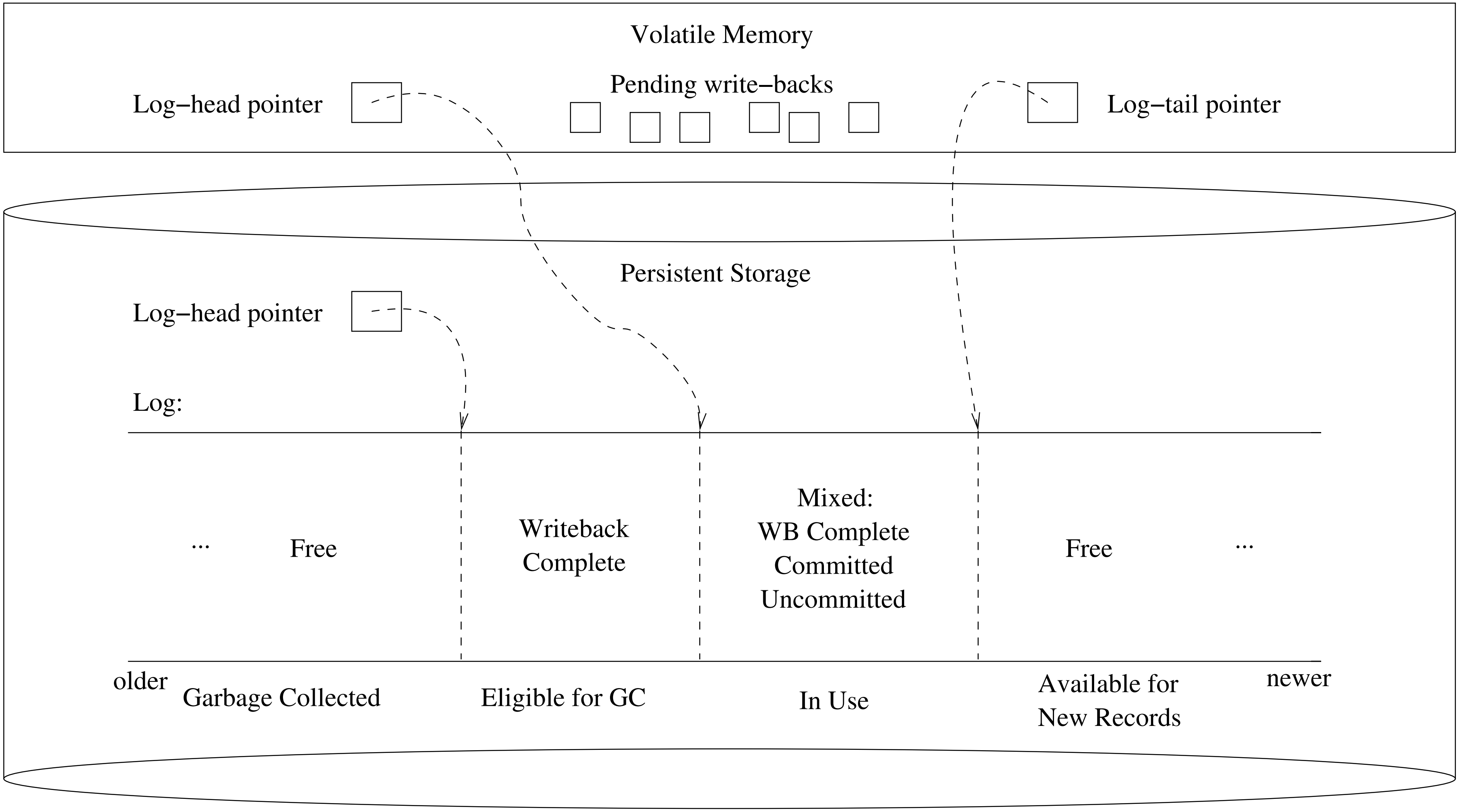
Questions

- 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?

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

Redo Log Implementation



Question

- Do we need the copy back?
  - What if update in place is very expensive?
  - Ex: flash storage, RAID

Log Structure

- Log is the data storage; no copy back
  - Storage split into contiguous fixed size segments
    - Flash: size of erasure block
    - Disk: efficient transfer size (e.g., 1MB)
  - Log new blocks into empty segment
    - Garbage collect dead blocks to create empty segments
  - Each segment contains extra level of indirection
    - Which blocks are stored in that segment
- Recovery
  - Find last successfully written segment

Transaction Isolation

Process A

move file from x to y

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

Process B

grep across x and y

```
grep x/* y/* > log
```

What if grep starts after changes are logged, but before commit?

Two Phase Locking

- Two phase locking: release locks only AFTER transaction commit
  - Prevents a process from seeing results of another transaction that might not commit

Transaction Isolation

Process A	Process B
Lock x, y	Lock x, y, log
move file from x to y	grep across x and y
mv x/file y/	grep x/* y/* > log
Commit and release x,y	Commit and release x, y, log
	Grep occurs either before or after move

Multiversion Concurrency

- Achieve serializability with no locks
  - Works well with distributed cache coherence
  - Non-blocking!
- On transaction start, pick a logical time for executing the transaction (usually, now)
  - All reads and writes execute at that logical time
  - Transactions can commit “out of order” in logical time
  - Requires keeping old versions of data in case needed
- On transaction commit, check if versions we used in this transaction are still valid
  - If can execute transaction without violating consistency, ok
  - Otherwise, abort and try again

Multiversion Conflicts

- If a write value at time T, and any committed transaction read (old) value after T
  - With two phase locks, one or the other of us would have needed to wait
- If read value at time T, and any committed transaction wrote (new) value before T
  - With two phase locks, one or the other of us would have needed to wait
- Are we guaranteed to make progress?