Ch 14: Reliable Storage

Bill Cheng

http://merlot.usc.edu/william/usc/

Main Points

- Problem posed by machine/disk failures
- Transaction concept
- Reliability
  - careful sequencing of file system operations
  - copy-on-write (WAFL, ZFS)
  - journalling (NTFS, linux ext4)
  - log structure (flash storage)
- Availability
  - RAID

File System Reliability

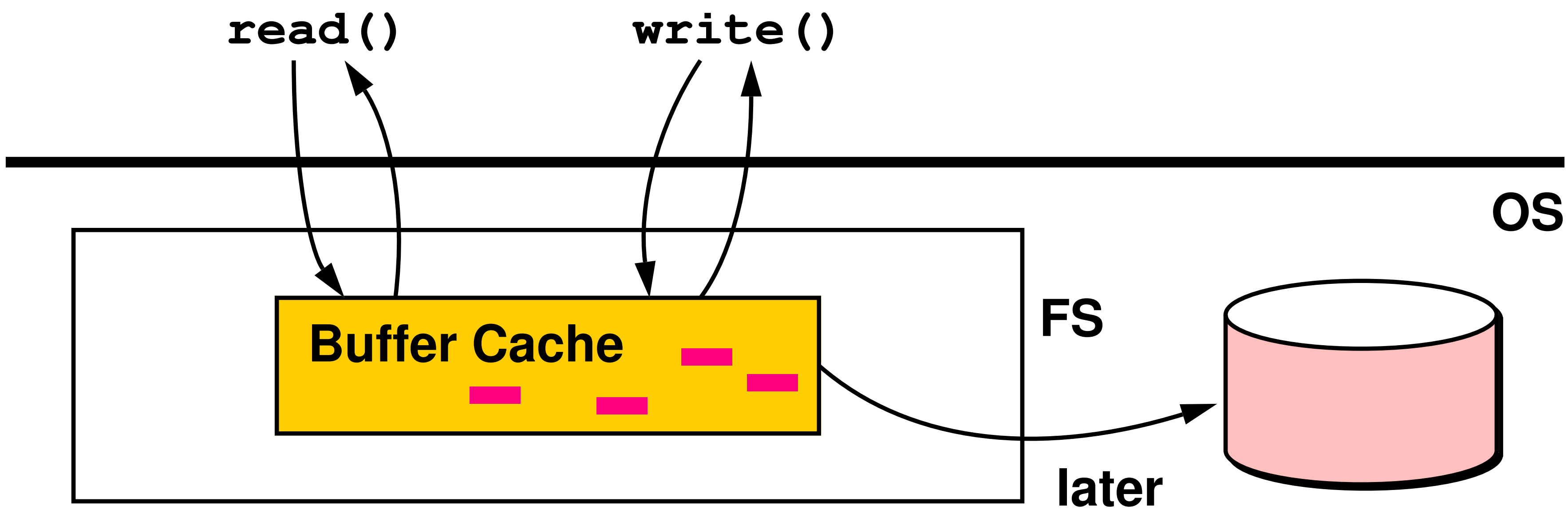
- A file system is *reliable* if it can be trusted to store and maintain data
  - this is different from *availability:* data can be accessed
- What can happen if disk loses power or machine software crashes?
  - some operations in progress may complete, but some operations in progress may be lost
  - 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
  - if *recovery takes too long* and the system is *not available* during recovery, customers are not going to be happy

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, etc.
  - 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?
  - *transations* for atomic updates (updates are all or nothing)
    - very popular in modern OSes
  - *redundancy* for media failure
    - RAID (Redundant Array of Inexpensive Disks)

(14.1) Transactions: Atomic Update

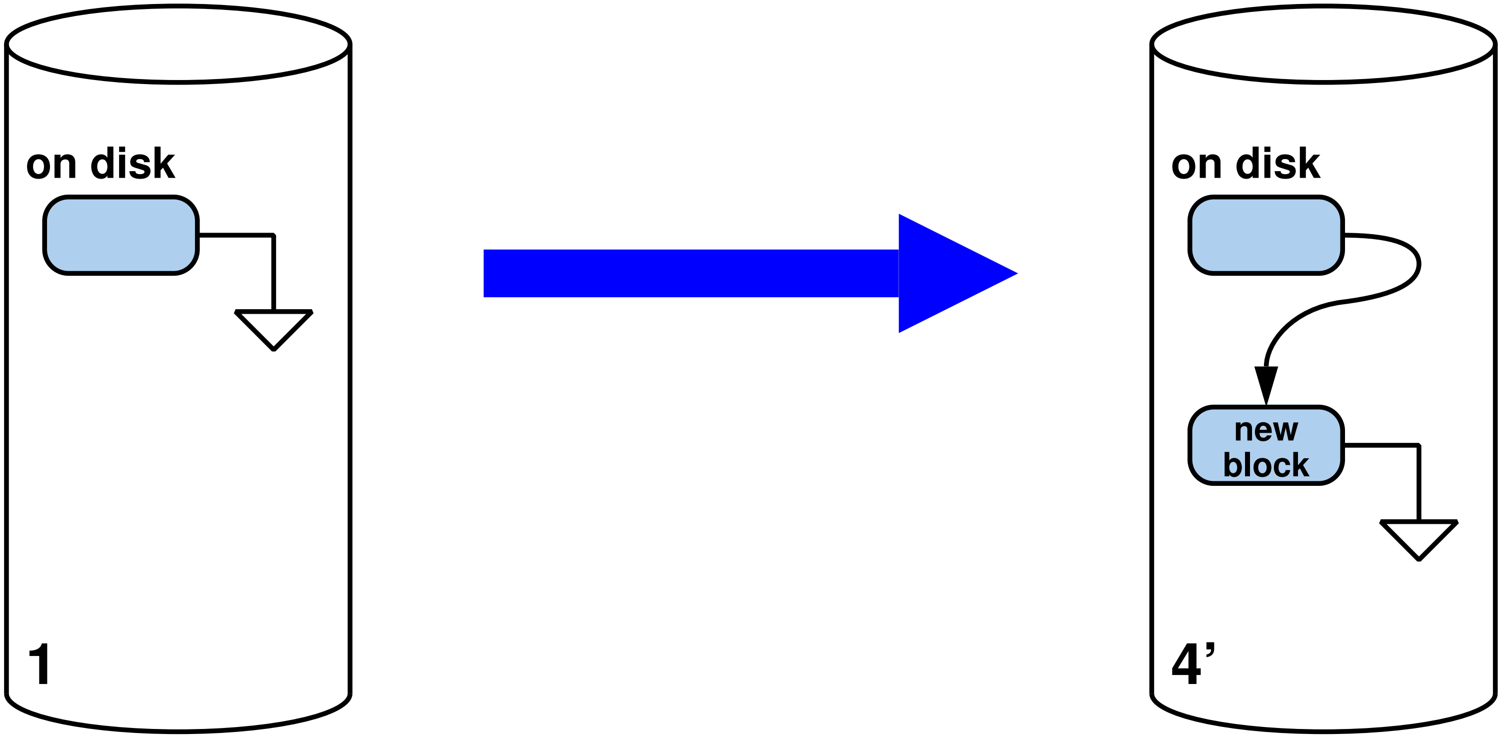
Block/Buffer Cache With Write-back



- *Dirty/modified blocks* in buffer cache
  - disk blocks are read in and cached in the buffer cache
    - originally "clean/unmodified"
  - a write operation would modify a disk block in the buffer cache
    - the block is labeled *"dirty/modified"*
  - *disk update:* the file system periodically gathers all the dirty blocks, update the disk, and clear the "dirty bits"
    - update is done *one disk block at a time*

Block/Buffer Cache With Write-back

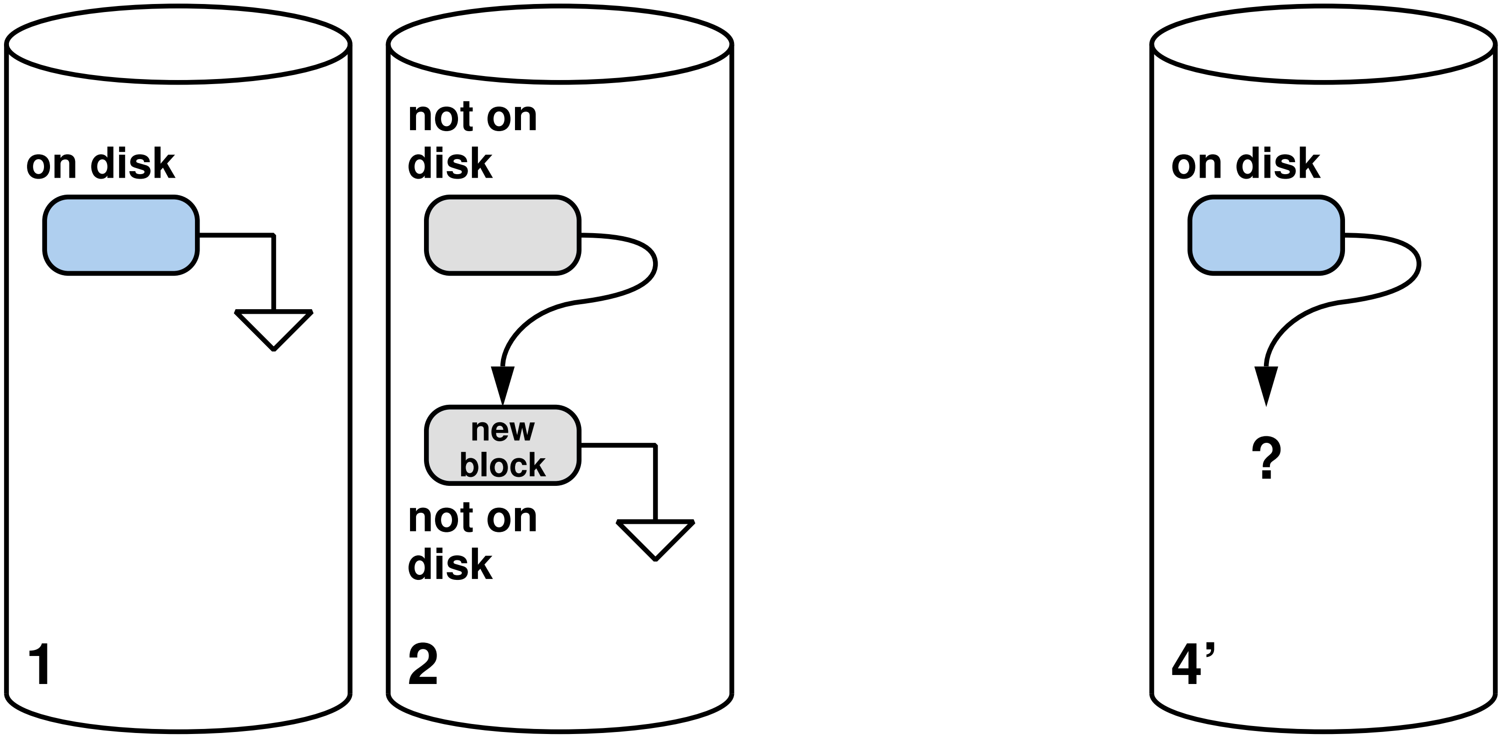
- In the event of a crash
  - file system can end up in an *inconsistent state*



- How to go from 1 to 4 *atomically*?
  - can we lock two disk updates in one atomic operation?
    - no way to ask the system not to crash in between updates

Block/Buffer Cache With Write-back

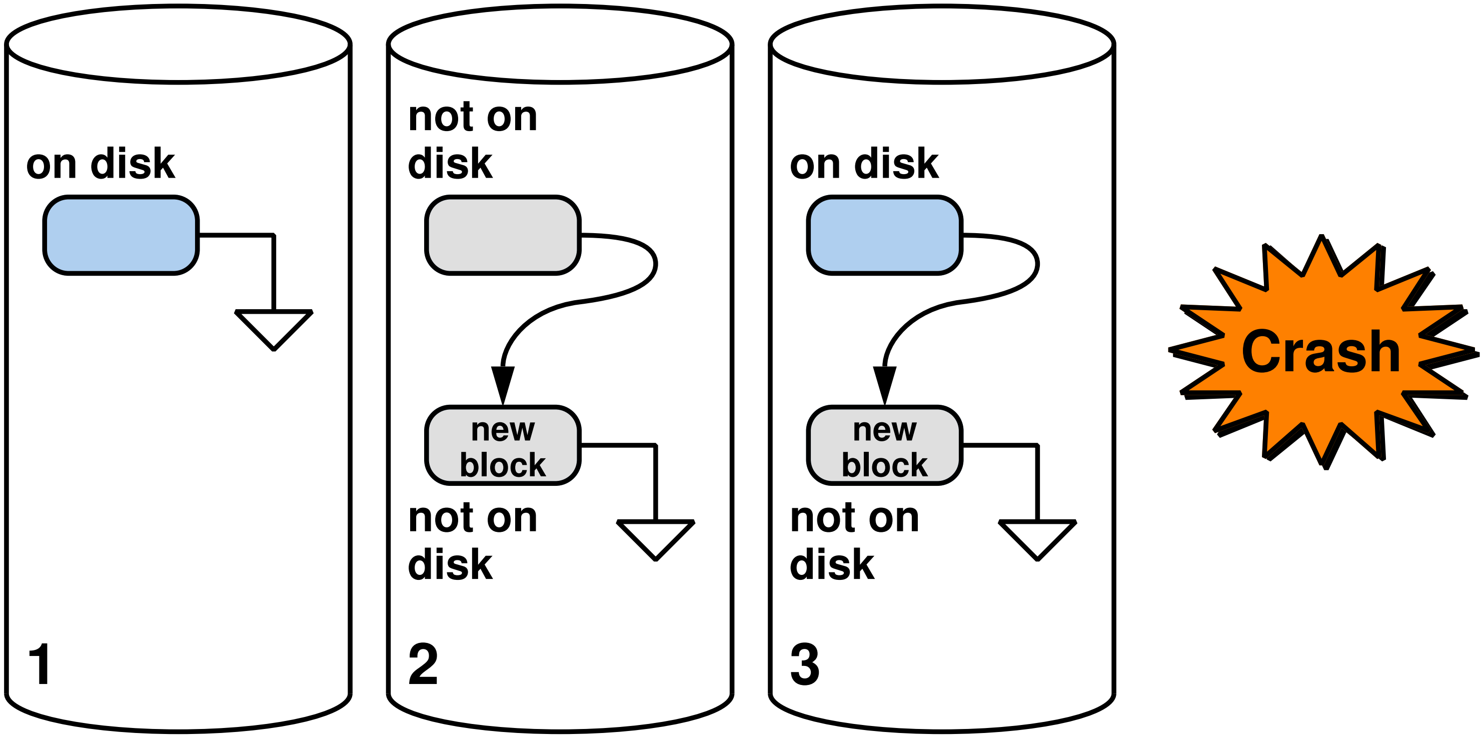
- In the event of a crash
  - file system can end up in an *inconsistent state*



- How to go from 1 to 4 *atomically*?
  - release dirty blocks to disk update thread (which has a mind of its own, unless you give it special instructions)

Block/Buffer Cache With Write-back

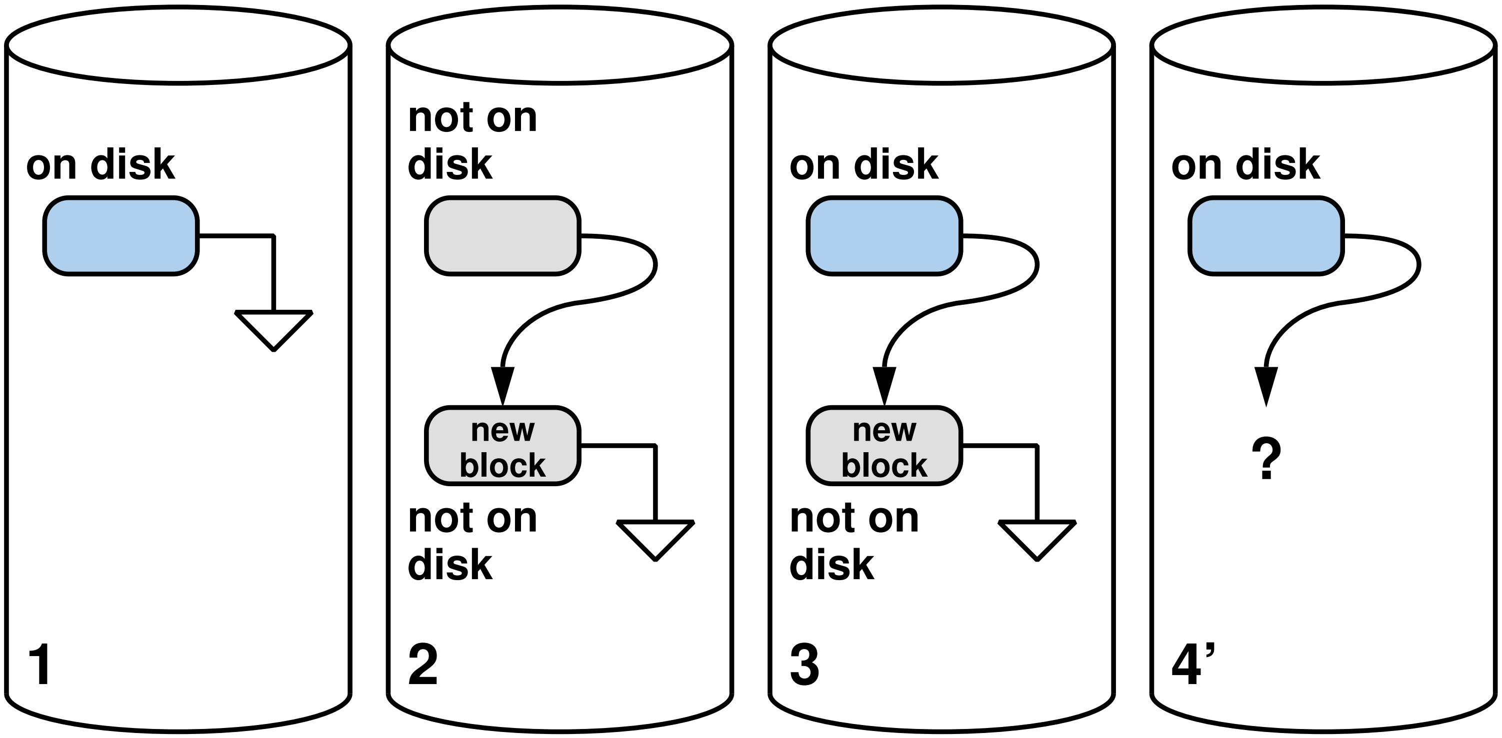
- **In the event of a crash**
  - file system can end up in an *inconsistent state*



- **How to go from 1 to 4 *atomically*?**
  - what if the disk update thread writes the top block to disk first and crash happens before the new block gets written?

Block/Buffer Cache With Write-back

- In the event of a crash
  - file system can end up in an *inconsistent state*



- How to go from 1 to 4 *atomically*?
  - after reboot, you would end up with this inconsistent state
- If something like SCAN is used, you cannot control which block gets written to disk first

Ad Hoc Approaches

- Until the mid-1990's, many file systems used ad hoc approaches to solving the problem of consistently updating multiple on disk data structures
  - *careful ordering:* e.g., FFS would carefully control the order that its updates were sent to disk
    - if a crash occurs in the middle of a group of updates, a scan of the disk during recovery could identify and repair inconsistent data structures
    - e.g., when creating a new file, FFS would:
      1) first update the free-inode bitmap, update disk
      2) initialize new file's inode, update disk
      3) update the directory that contains the new file
      - if crash happens, during reboot, run `fsck` (file system check) to scan all of the file system's metadata and repair them if needed
      - for ext2 file system, `fsck` would add discovered blocks to `/lost+found` and invite user to inspect them

Careful Ordering

- Problem with *carful ordering*:
  - complex reasoning
  - slow updates
    - to ensure that updates are stored in the order that allowed the system's state to be analyzed, file systems are forced to insert sync operations or barriers between independent operations
      - reducing amount of pipelining and parallelism in the stream of requests to storage devices
      - e.g., to perform 3 disk updates, you may end up waiting for 3 full disk rotations
  - extremely slow recovery (may be okay for small disks in the 1970's, by 1990's, this can take minutes)

Application-level Approach

- POSIX's `rename(oldpath, newpath)` system call
  - if `newpath` already exists, `newpath` will be replaced *atomically*
- Knowing the above, a text editor that's editing "foo.txt" will do the following when saving the file
  - save the file as "#foo.txt#", then call:

```
rename("#foo.txt#", "foo.txt")
```

- This is not a general solution

The Transaction Abstraction

- Transactions provide a way to *atomically* update multiple pieces of persistent state
- Ex: updating a web site
  - replace the current collection of documents in /server/live with a new collection of documents in /dev/ready
  - requirements:
    - you don't want users to see intermediate steps when some of the documents have been updated and others are not
  - a transaction file system like Windows Vista's TxF (Transactional NTFS) provides an API that lets applications apply all updates atomically like in the following pseudo-code:

The Transaction Abstraction

```
ResultCode publish() {
  transactionID = beginTransaction();
  foreach file f in /dev/ready that is not in /server/live {
    error = move f from /dev/ready to /server/live;
    if (error) {
      rollbackTransaction(transactionID);
      return ROLLED_BACK;
    }
  }
  foreach file f in /server/live that is not in /dev/ready {
    error = delete f;
    if (error) {
      rollbackTransaction(transactionID);
      return ROLLED_BACK;
    }
  }
  foreach file f in /dev/ready that differs from /server/live {
    error = move f from /dev/ready to /server/live;
    if (error) {
      rollbackTransaction(transactionID);
      return ROLLED_BACK;
    }
  }
  commitTransaction(transactionID);
  return COMMITTED;
}
```

The Transaction Abstraction

- A transaction can finish in one of two ways:
  - *commit:* all of its updates occur
  - *roll back:* none of its updates occur
- If the transaction commits, we are guaranteed that all of the udpates *will be seen by all subsequent reads*
- If the transaction encounters errors and rolls back or crashes, no reads outisde of the transaction will see any of the updates

Transaction Concept

- Transaction is a way to perform a set of updates while providing the following *ACID properties*
  - *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
  - *consistency:* sequential memory model
    - take the file system from one consistent state to another
  - *isolation:* other transactions do not see results of earlier transactions until they are *committed*
    - if multiple transactions are executing concurrently, for each pair of transactions T1 and T2, it either appears that T1 executed entirely before T2 or vice versa
  - *durable:* operations that complete stay completed
    - future failures do not corrupt previously stored data
    - `commitTransaction()` will not return until all of the transaction's updates have been safely stored in persistent storage

Transactions vs. Critical Sections

- Critical sections provide a way to update state that is atomic, consistent, and isolated (but not durable)
  - adding the durability requirement significantly changes how we implement atomic updates

Implementing Transactions

- The basic idea is to persistently store all of a transaction's *intentions* in some separate location of persistent storage first
  - only when all intentions are stored and the transaction commits should the file system begin overwriting the target data structures
  - if the overwrites are interrupted in the middle, then on recovery, the system can complete the transaction's updates using the persistently stored intentions

Redo Logging

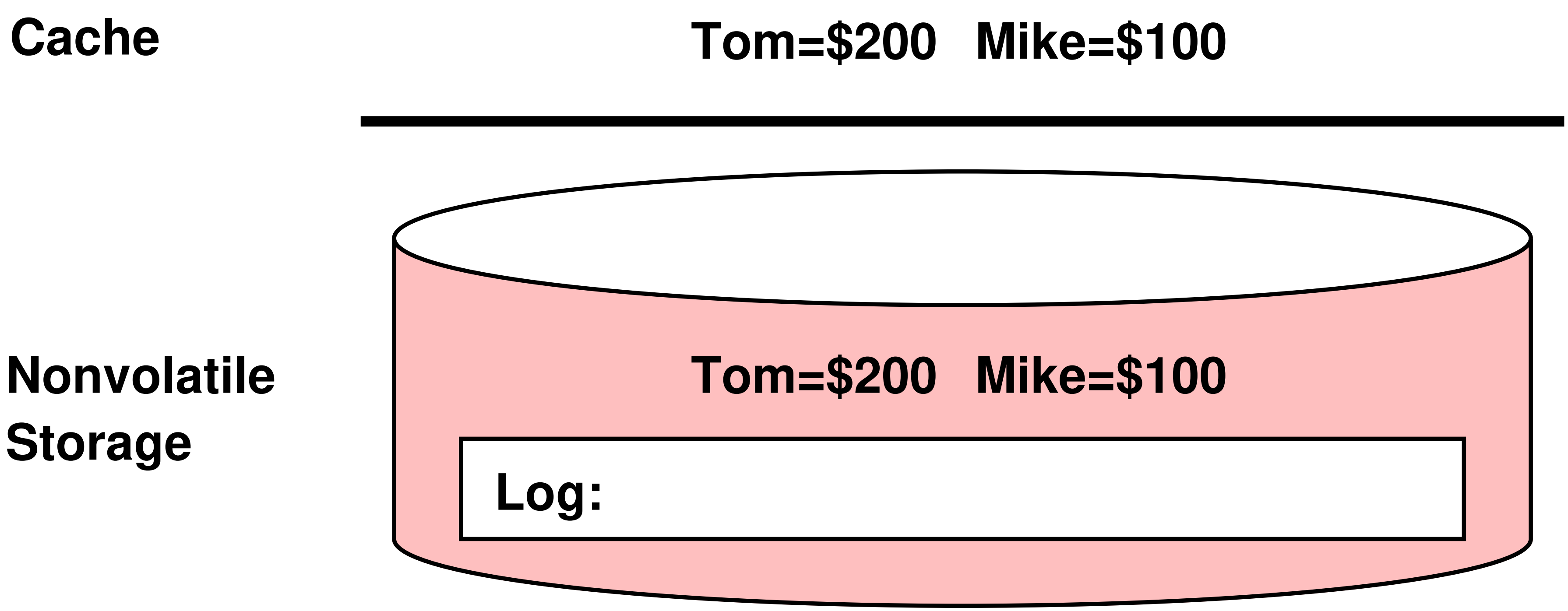
- ***Redo logging*** uses a persistent log for recording intentions and executes a transaction in 4 stages:
  - ***prepare:*** append all planned updates to log
  - ***commit:*** append a single commit record to the log, indicating that the transaction has committed
    - if a transaction is rolled back, a roll-back record may be placed in the log to indicate that the transaction has been abandoned
      - writing a roll-back record is optional
  - ***write-back:*** copy changes to disk
  - ***garbage collect:*** reclaim space in log
- The moment the sector containing the commit record is successfully stored is the ***atomic commit time***
  - before that moment, the transaction may safely be rolled back
  - after that moment, the transaction must take effect

Redo Logging

- If the system crashes in the middle of a transaction, it must execute a recovery routing before processing new requests
  - scan sequentially through the log, taking the following actions for each type of record in a transaction:
    - *update record:* add this record to a list of updates planned for the specified transaction
    - *commit record:* write-back all of the transaction's logged updates to their target locations
    - *roll-back record:* discard the list of updates planned for the specified transaction
  - when the end of the log is reached, the recovery process discards any update records for transactions that do not have commit records in the log

Redo Logging: Before Transaction Start

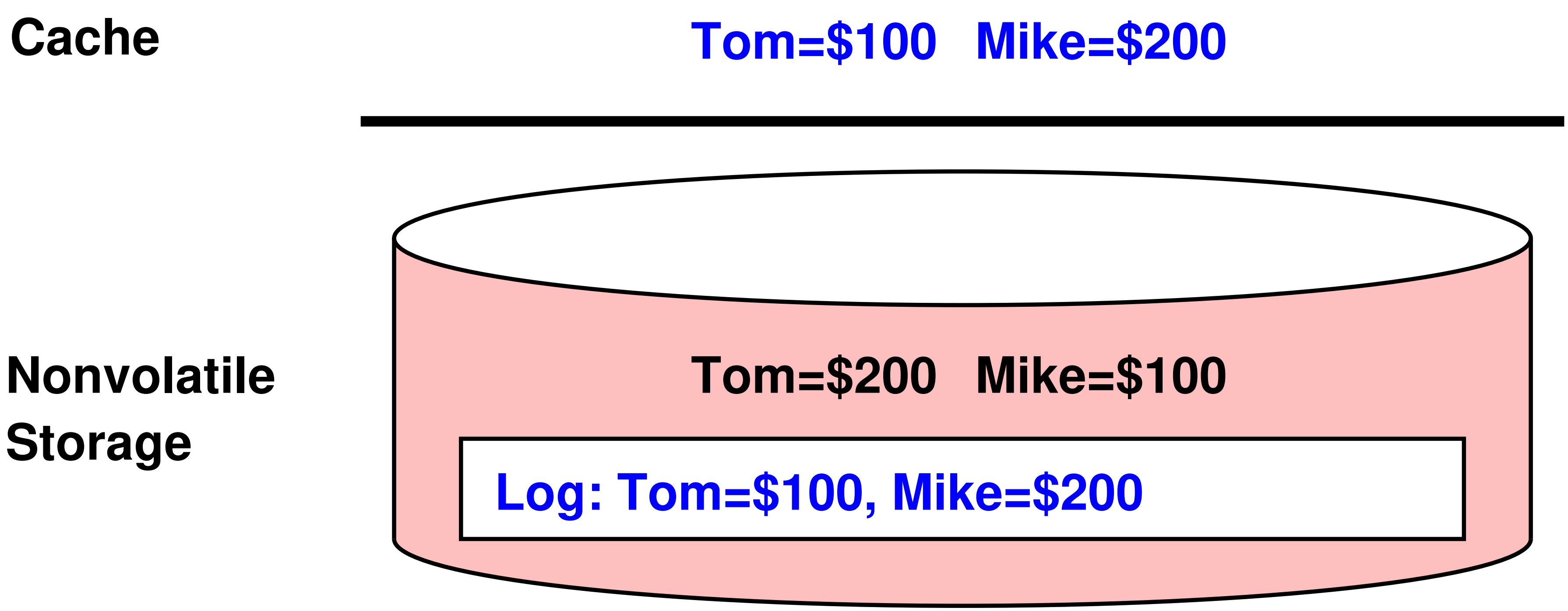
- Ex: transfer $100 from Tom's account to Mike's account
  - initially, Tom's account has $200 and Mike's account has $100



- Recall the 4 steps
  - ***prepare:*** append all planned updates to log
  - ***commit:*** append a single commit record to the log
  - ***write-back:*** copy changes to disk
  - ***garbage collect:*** reclaim space in log

Redo Logging: After Updates Are Logged

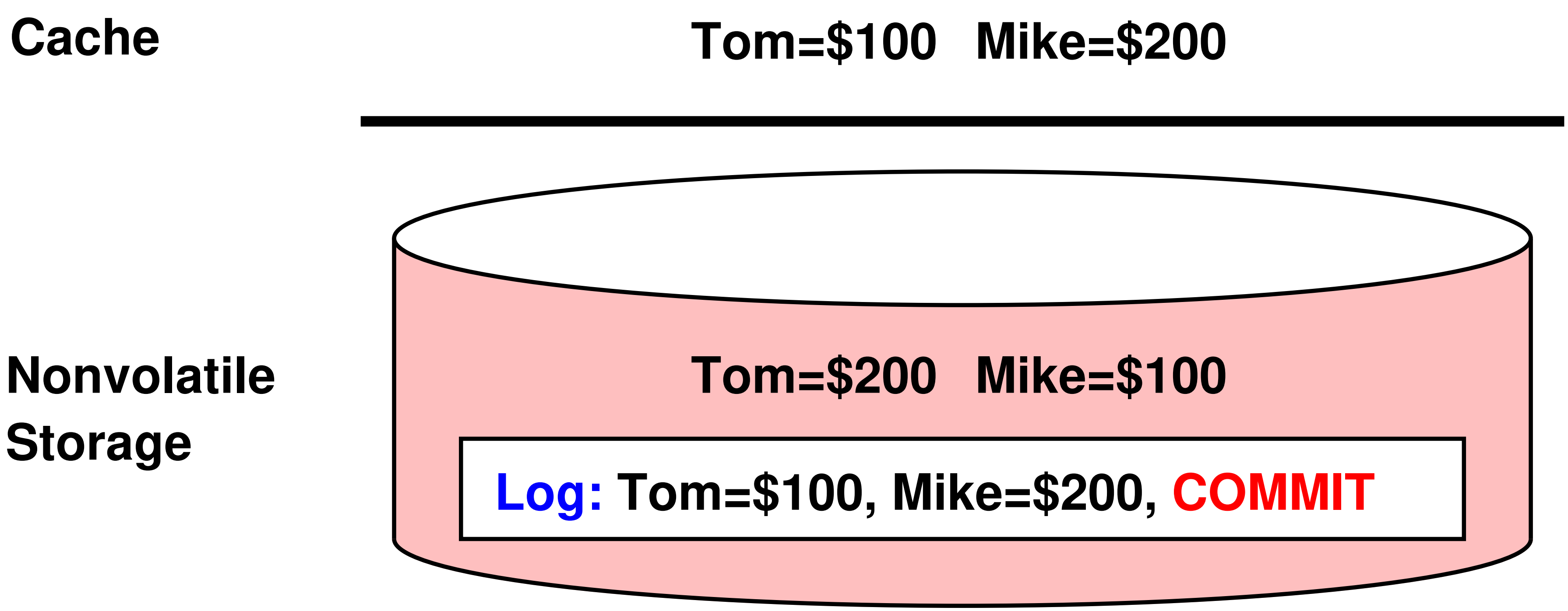
- Ex: transfer $100 from Tom's account to Mike's account
  - initially, Tom's account has $200 and Mike's account has $100



- Recall the 4 steps
  - *prepare:* append all planned updates to log
  - *commit:* append a single commit record to the log
  - *write-back:* copy changes to disk
  - *garbage collect:* reclaim space in log

Redo Logging: After Commit Logged

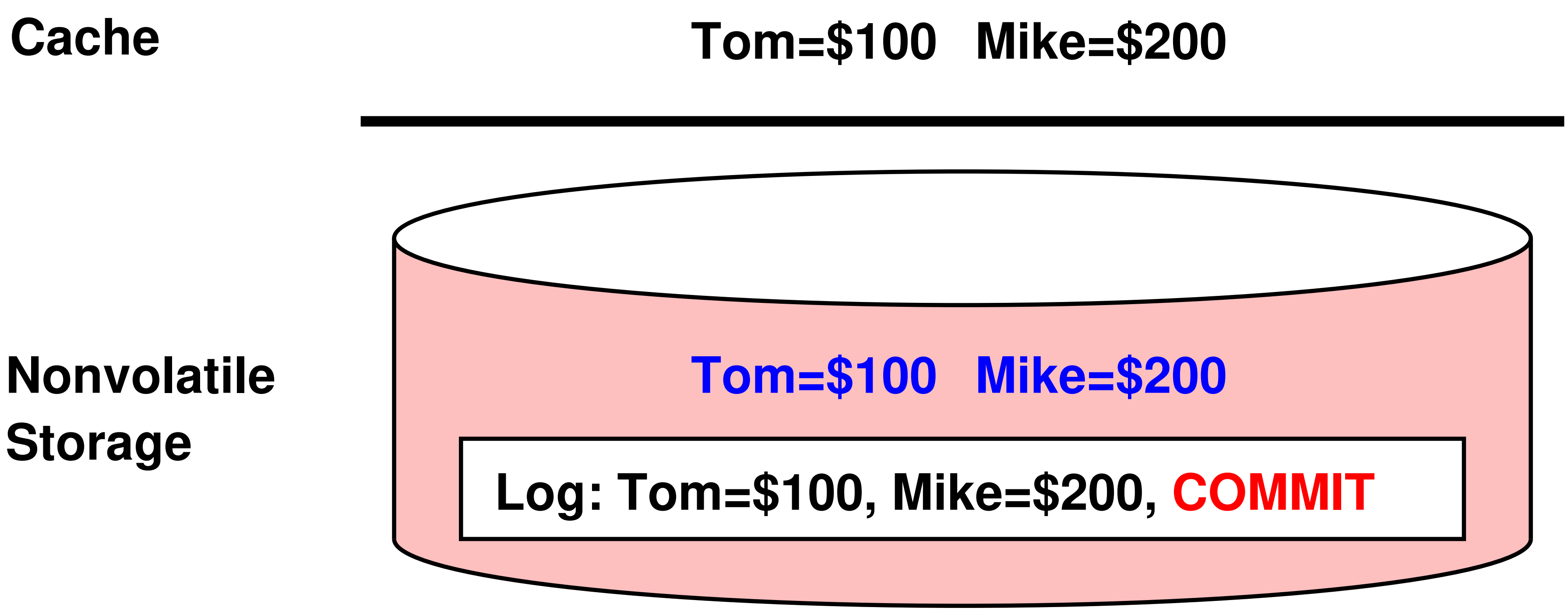
- Ex: transfer $100 from Tom's account to Mike's account
  - initially, Tom's account has $200 and Mike's account has $100



- Recall the 4 steps
  - *prepare:* append all planned updates to log
  - *commit:* append a single commit record to the log
  - *write-back:* copy changes to disk
  - *garbage collect:* reclaim space in log

Redo Logging: After Write Back

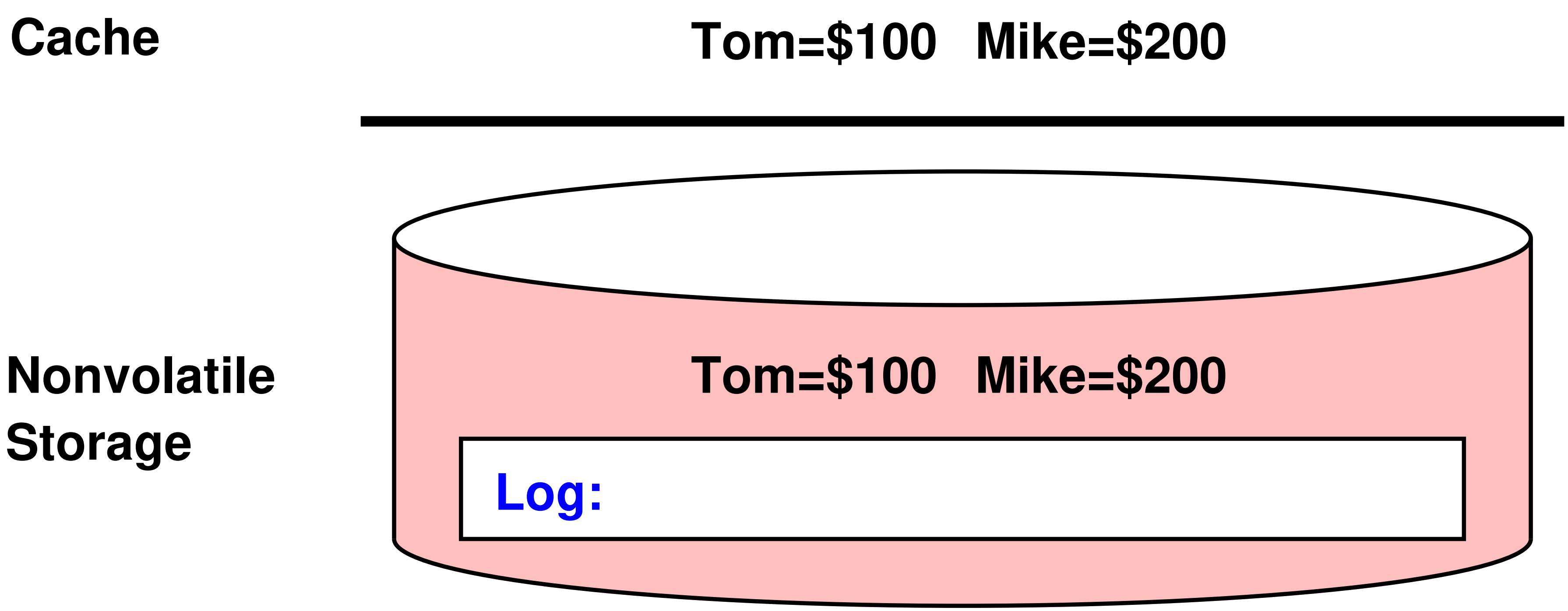
- Ex: transfer $100 from Tom's account to Mike's account
  - initially, Tom's account has $200 and Mike's account has $100



- Recall the 4 steps
  - ***prepare:*** append all planned updates to log
  - ***commit:*** append a single commit record to the log
  - ***write-back:*** copy changes to disk
  - ***garbage collect:*** reclaim space in log

Redo Logging: After Garbage Collection

- Ex: transfer $100 from Tom's account to Mike's account
  - initially, Tom's account has $200 and Mike's account has $100



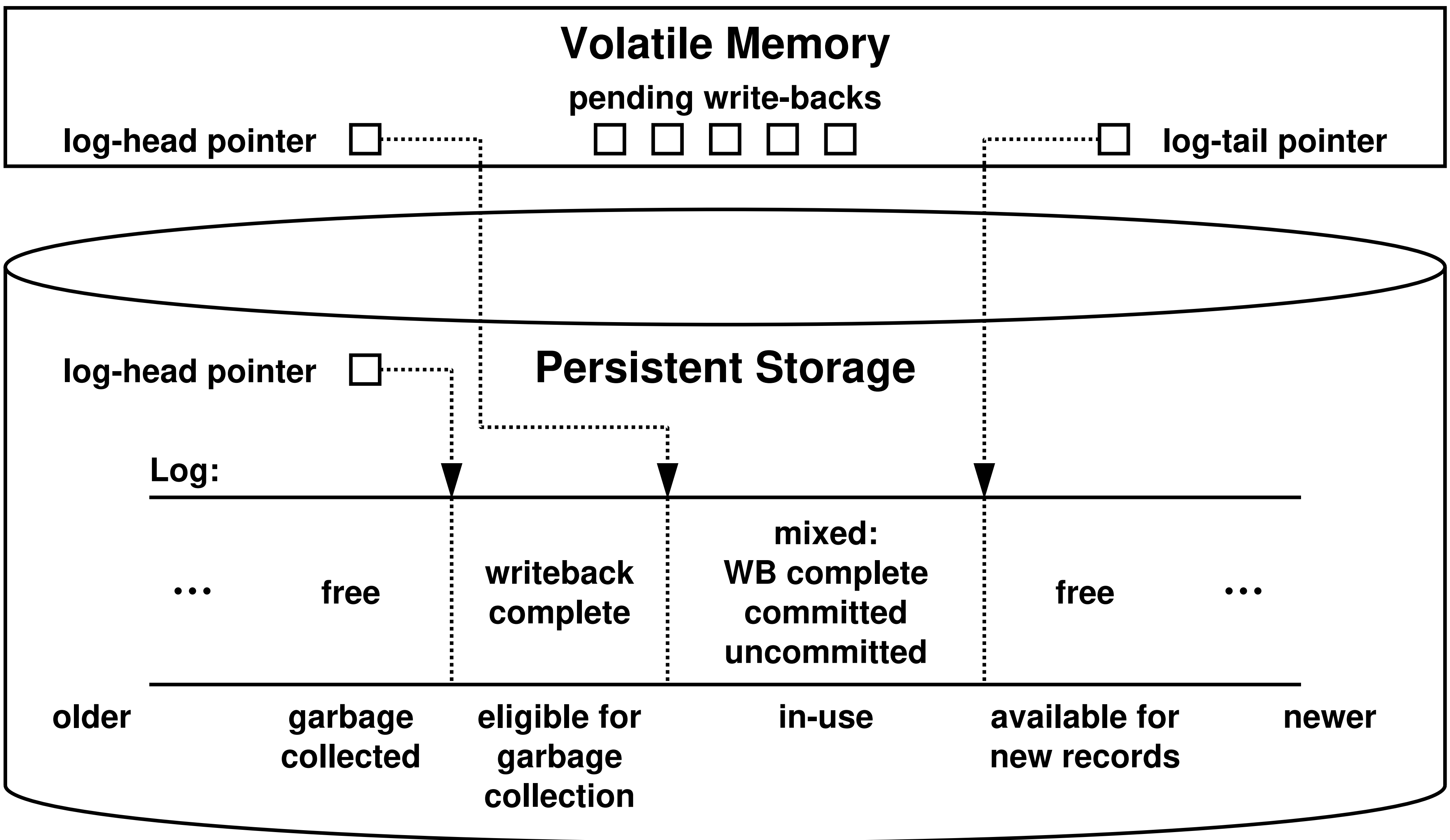
- Recall the 4 steps
  - *prepare:* append all planned updates to log
  - *commit:* append a single commit record to the log
  - *write-back:* copy changes to disk
  - *garbage collect:* reclaim space in log

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

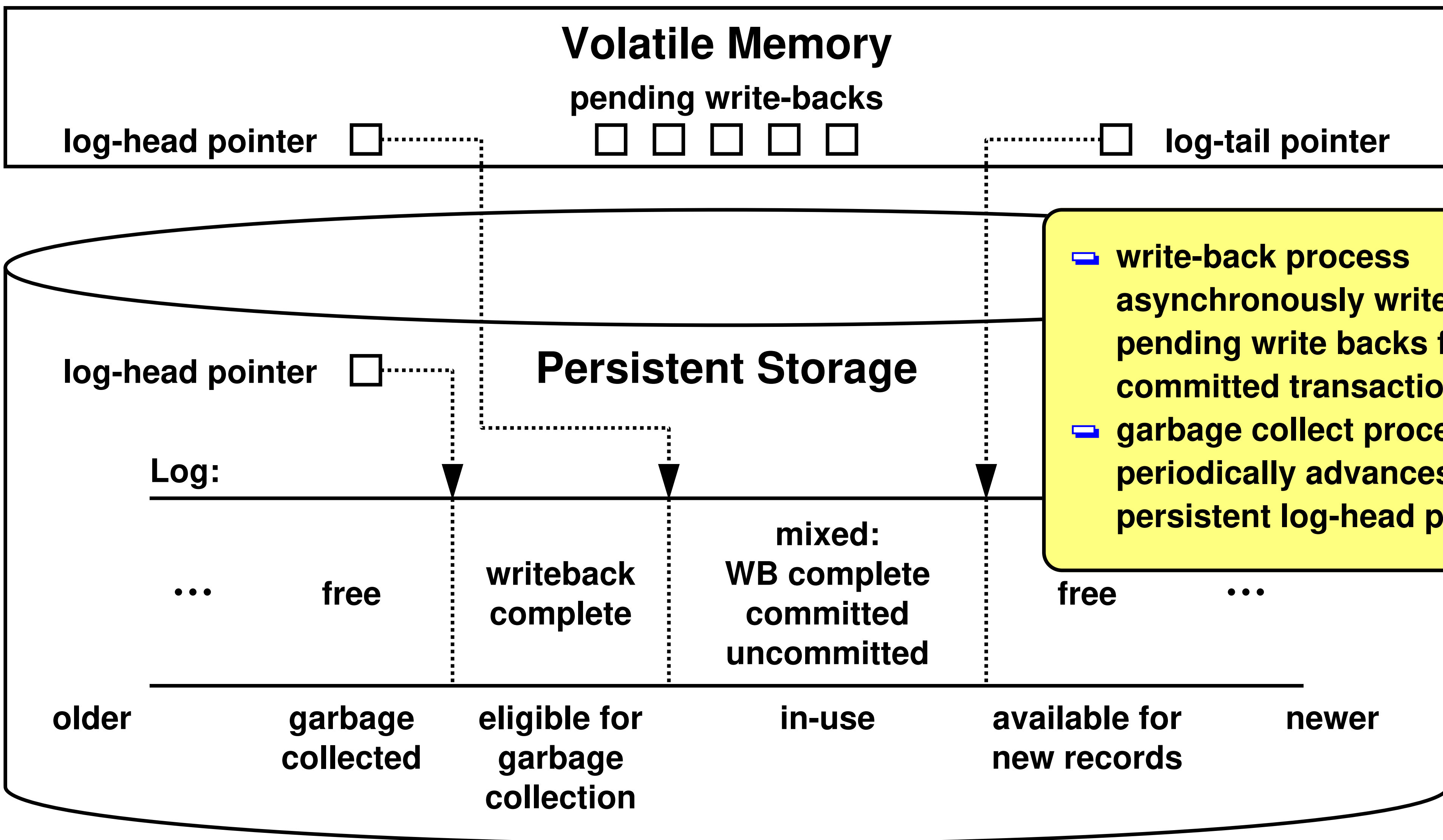
- What happens if machine crashes during recovery?
  - note that updates are *idempotent* operations, i.e., performing the operation twice has the safe effect as doing it once
    - account balance = $100 is idempotent
    - account balance += $100 is not idempotent

Redo Log Implementation



- Circular log implementation containing 3 regions

Redo Log Implementation



- write-back process asynchronously writes pending write backs for committed transactions
- garbage collect process periodically advances the persistent log-head pointer

⇨ **Circular log implementation containing 3 regions**

Transaction Isoluation

Process A

`move` file from x to y

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

Process B

`grep` across x and y

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

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

Two Phase Locking

- Recall two phase locking
  - expanding phase: locks may be acquired but not released
  - contracting phase: locks may be released but not acquired
- Common way to enforce isolution is to use *two phase locking:* release locks only AFTER transaction commit
  - prevents a process from seeing results of another transaction that might not commit
- *Deadlock is possible* for a set of transactions
  - can force one of more transactions to rollback, release their locks, and restart at some later time

Transaction Isoluation

Process A

lock x, y

`move` file from x to y

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

commit and release x, y

Process B

lock x, y, log

`grep` across x and y

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

commit and release x, y, log

- `grep` occurs either before or after `move`

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**
  - ***optimistic concurrency control:*** **abort any transaction that would conflict with serializability**

Performance Of Redo Logging

- One would expect redo logging to have poor performance since each update is written to disk twice
  - in practice, it can have good performance (often better than update in place, especially for small writes)

- Four factors allow efficient implementation of redo logging
  - log updates are *sequential* (append to the log is fast)
    - high performance systems use a separate disk for logging (i.e., no seeks)
  - write-back is *asynchronous* (all write backs occur after commit)
  - compared to careful ordering, this has fewer barriers or synchronous writes are required
  - group commits: can combine a set of transaction commits into one log write to improve performance

Caveat

- Most file systems implement a transactional model internally
  - copy on write
  - redo logging
- Most file systems provide a transactional model for individual system calls
  - file rename, move, ...
- Most file systems do NOT provide a transactional model for user data
  - when you are downloading a 3.5 GB file (such as our virtual appliance), do you want to automatically write all 3.5 GB of file data into the log?

Transactional File Systems

- Most modern file systems use transactions to make changes to the file system
- Journaling file systems
  - apply updates to the system's *metadata* via transactions, but update the contents of users' files in place
    - first write metadata updates to a redo log, then commit them, then perform write-backs, then garbage collect
    - if a program using a journaling file system requires atomic multi-block updates to the content of a regular file, it needs to provide them itself
  - e.g., Microsoft NTFS, Apple HFS+, Linux ext3 and ext4
- Logging file systems
  - include all updates to disk (both metadata and data) in transactions
  - e.g., Linux ext3 and ext4 can be configured to use either journaling or logging

(14.2) Error Detection & Correction

Storage Availability

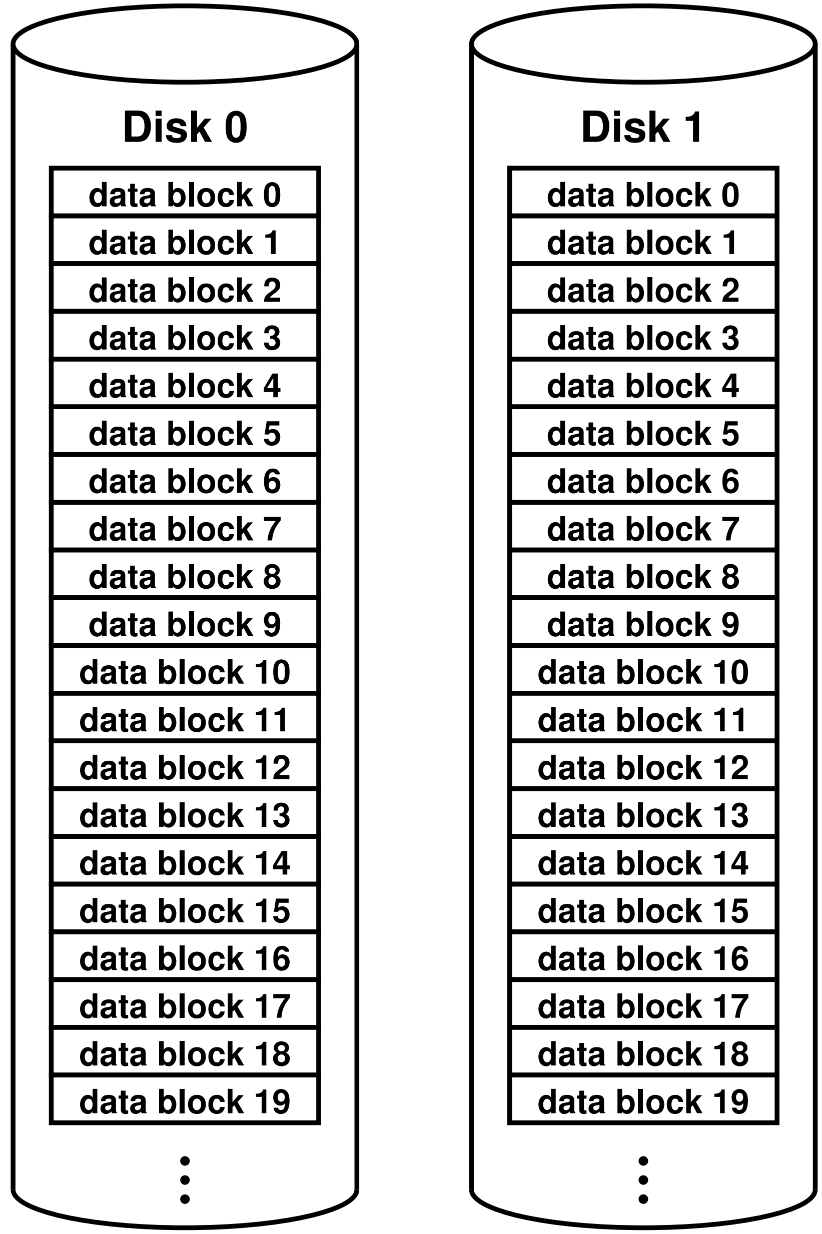
- Storage reliability: data fetched is what you stored
  - transactions, redo logging, etc.
- Storage availability: data is there when you want it
  - more disks → higher probability of some disk failing
  - data available $\propto$ Prob(disk working)^k
    - if failures are *independent* and data is spread across k disks
  - for large k, probability that system works → 0

RAID

- Replicate data for availability
  - RAID 0: no replication
  - RAID 1: mirror data across two or more disks
    - Google File System replicated its data on three disks, spread across multiple racks
  - RAID 5: split data across disks, with redundancy to recover from a single disk failure
  - RAID 6: RAID 5, with extra redundancy to recover from two disk failures

RAID 1: Mirroring

- Replicate writes to both disks
- Reads can go to either disk
- When the primary disk dies, switch to the backup disk
  - build a new disk by copying data from the backup disk
    - extra workload on the backup disk
      - degraded performance



Parity

- ***Even Parity:*** **parity block = block1 ⊕ block2 ⊕ block3 ⊕ ...**
  - **where ⊕ is the XOR operator**

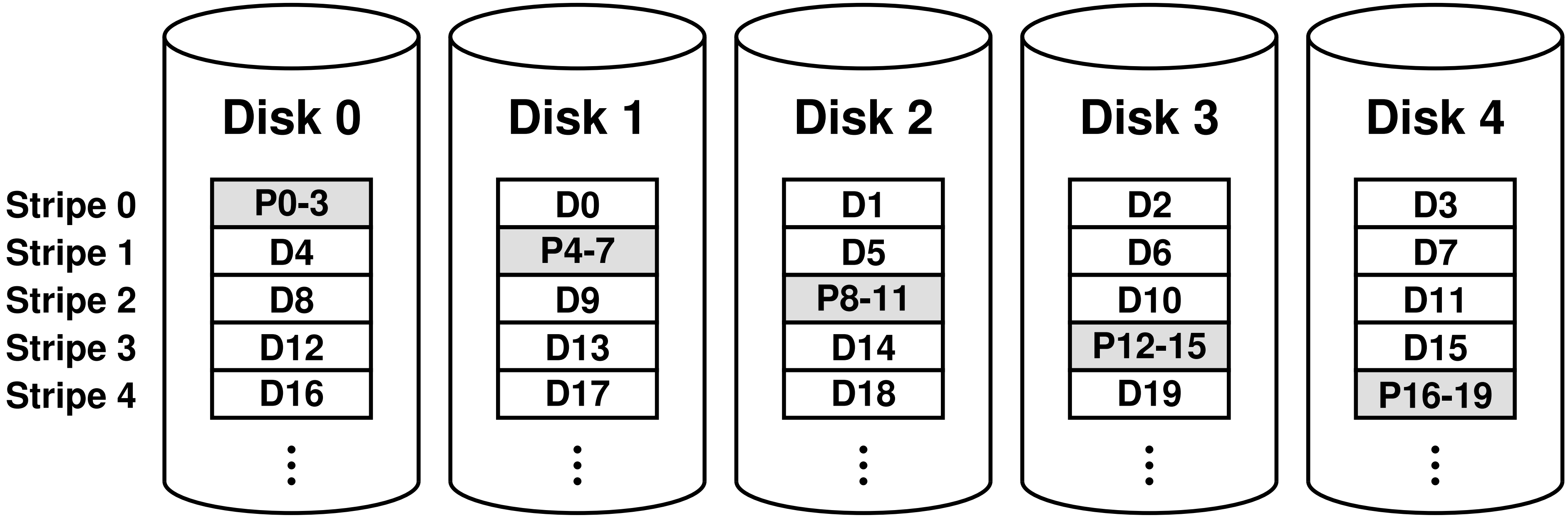
block 1:	10001101
block 2:	01101100
block 3:	11000110
even parity block:	00100111

- ***Odd Parity:*** **parity block = ~(block1 ⊕ block2 ⊕ block3 ⊕ ...)**
  - **where ~ is the bit compliment operator**

block 1:	10001101
block 2:	01101100
block 3:	11000110
odd parity block:	11011000

- **Can reconstruct any missing block from the others by XOR all remaining blocks and flip all bits if odd parity is used**

RAID 5: Rotating Parity



- If a parity disk is used instead, the parity disk can become a performance bottleneck (number of write requests to the parity disk is the sum of the number of write requests to all other disks)

RAID Update

- Mirroring
  - write every mirror
- RAID-5: to write one block
  - read old data block
  - read old parity block
  - write new data block
  - write new parity block
    - old data xor old parity xor new data
- RAID-5: to write entire stripe
  - write data blocks and parity

Non-Recoverable Read Errors

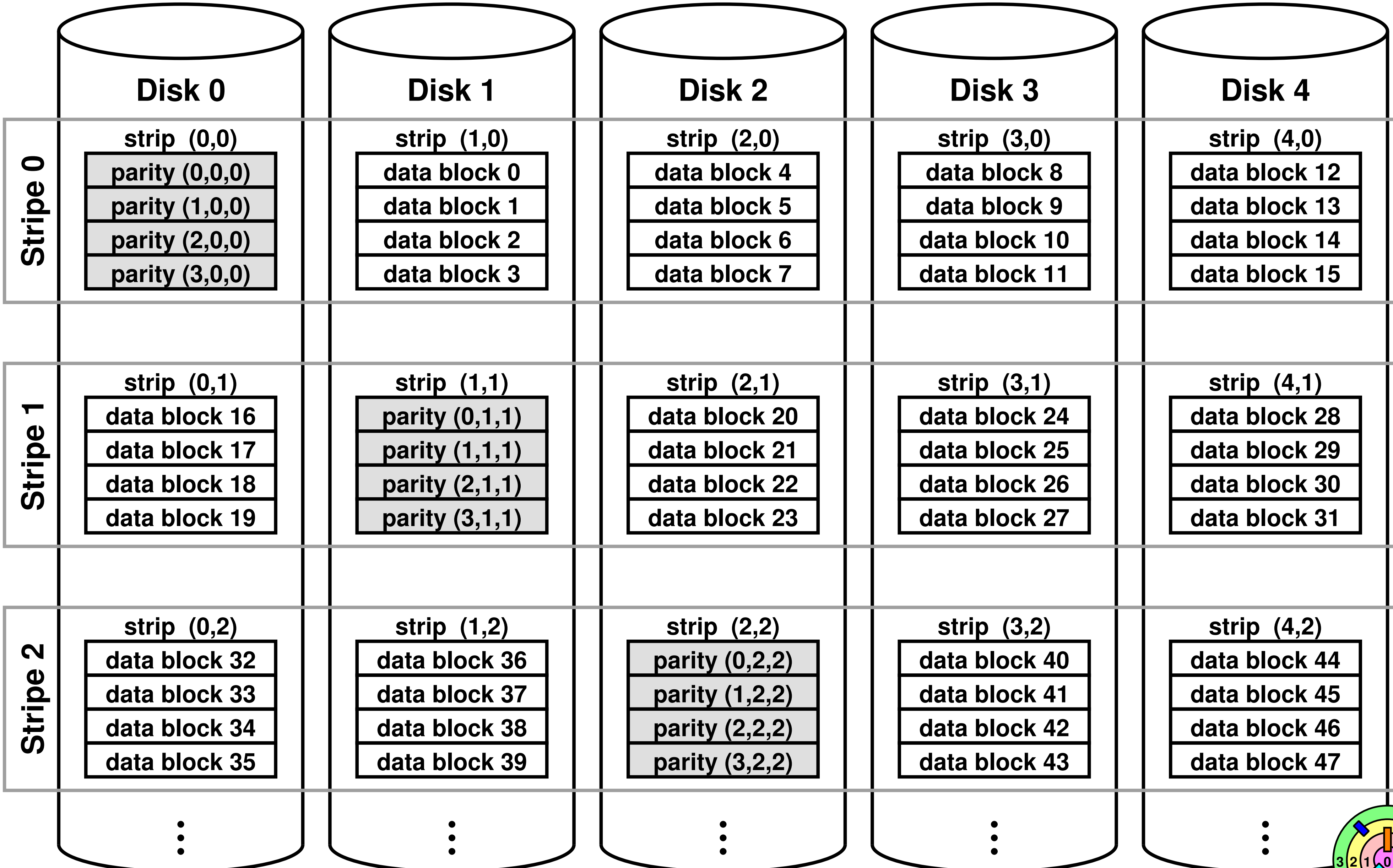
- Disk devices can lose data
  - one sector per 10^{15} bits read
  - causes:
    - physical wear
    - repeated writes to nearby tracks
- What impact does this have on RAID recovery?

Read Errors and RAID recovery

- **Example**
  - ten 1 TB disks, and 1 of them fails
  - read remaining disks to reconstruct missing data
  - cannot tolerate a block failure on any of the remaining disks
- **Probability of recovery** $= (1 - 10^{-15})^{(9 \text{ disks} \times 8 \text{ bits/byte} \times 10^{12} \text{ bytes/disk})}$ $= 93\%$
  - this failure rate is much higher than 2 random disk failures
- **Solutions:**
  - RAID-6: two redundant disk blocks
    - parity, linear feedback shift
  - scrubbing: read disk sectors in background to find and fix latent errors

Extra Slides

RAID 5: Rotating Parity

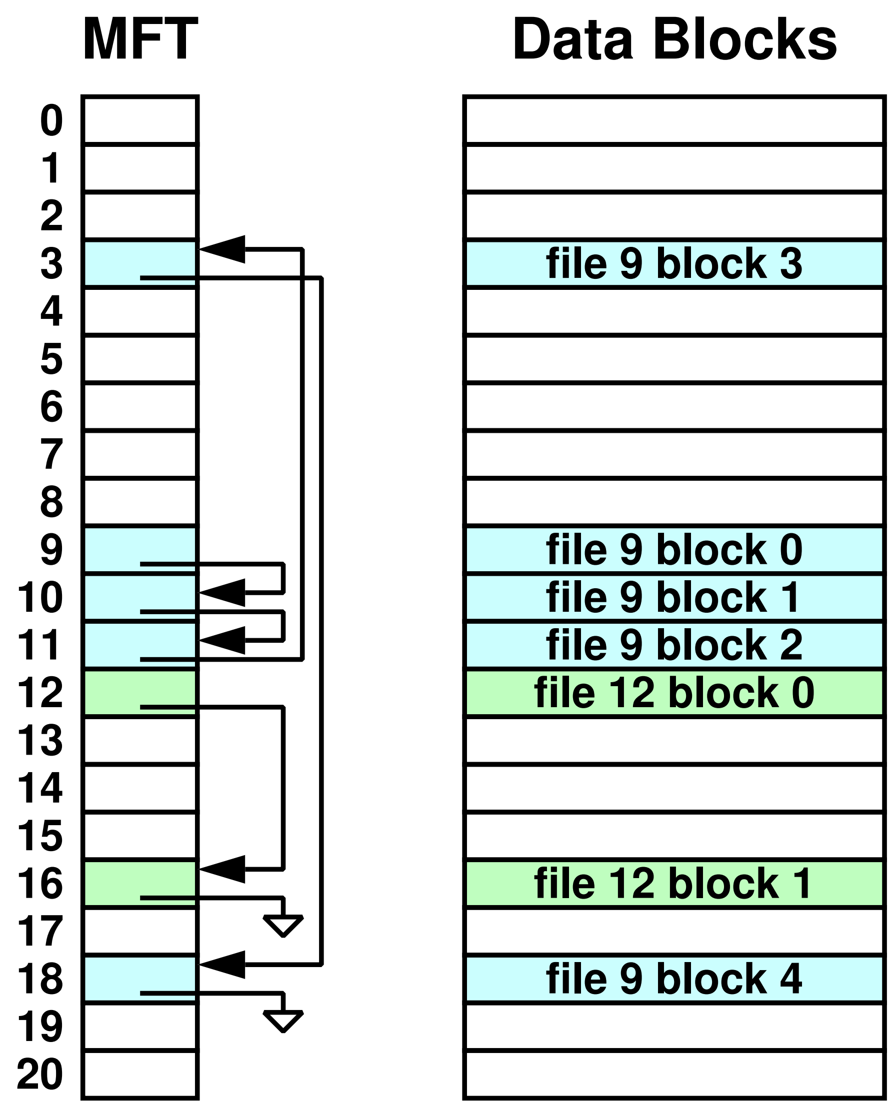


Reliability Approach #1: Careful Ordering

- 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)

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



FAT: Append Data To File

- Normal operation:
  - 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

FAT: Create New File

- Normal operation:
  - 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
- Recovery
  - scan MFT
  - if any unlinked files (not in any directory), delete
  - scan directories for missing update times

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/f1 y/
```

Process B

grep across x and y

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

Q: Will grep always see contents of `f1`?

Application Level

- Normal operation:
  - 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
- Recovery
  - on startup, see if any files were left open
  - if so, look for backup file
  - if so, ask user to compare versions

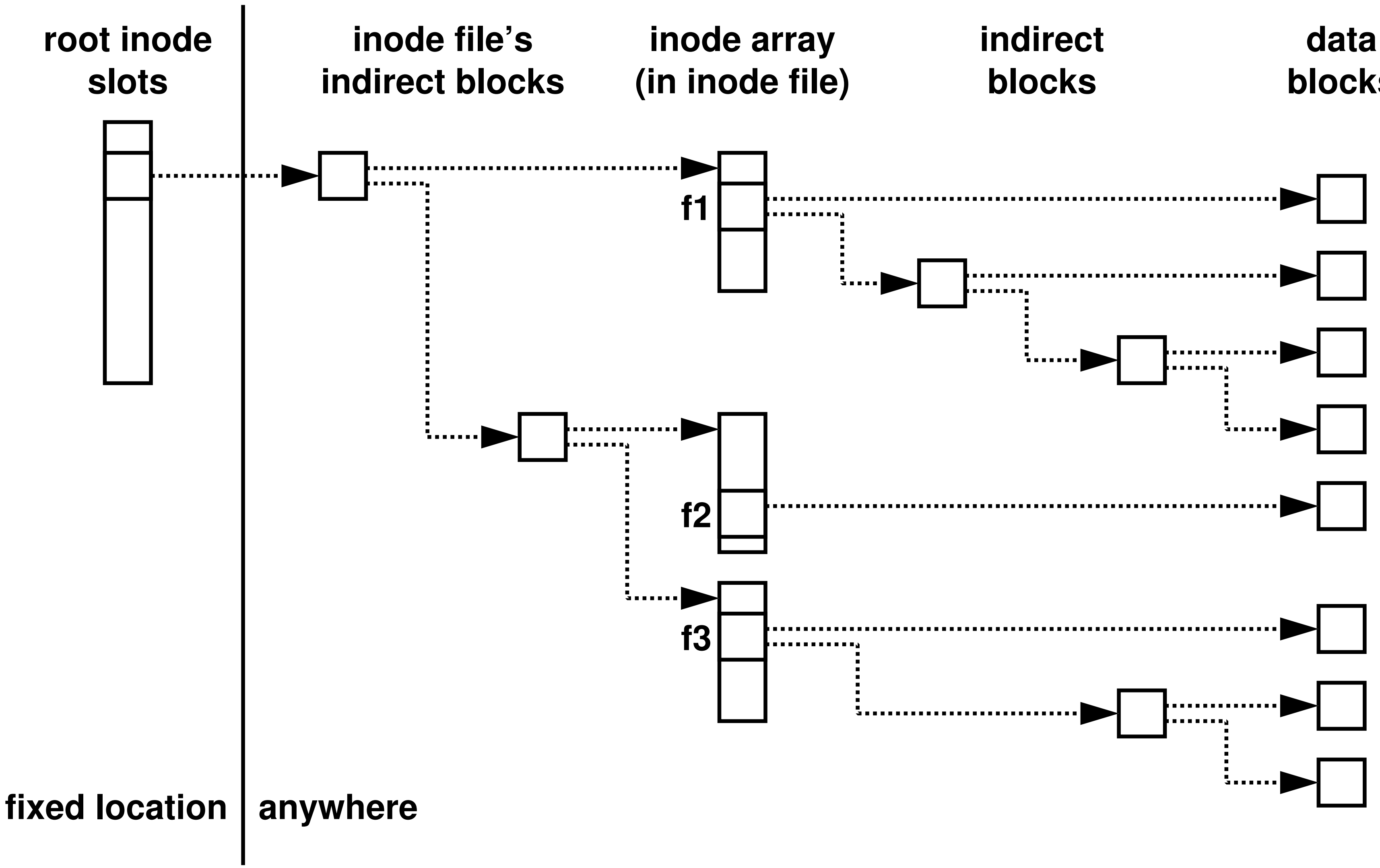
Careful Ordering

- **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

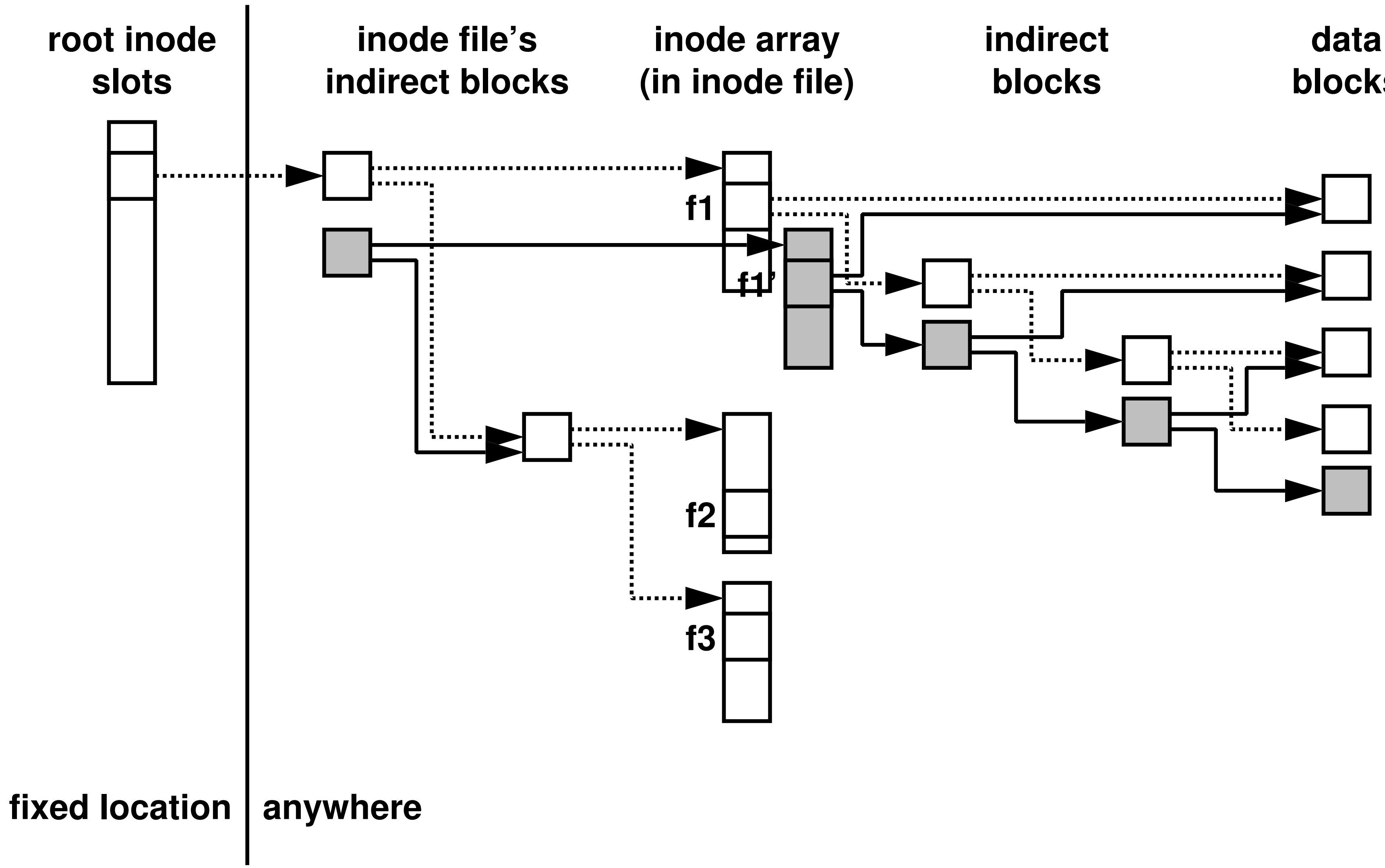
Reliability Approach #2: Copy on Write File Layout

- 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)

COW/Write Anywhere

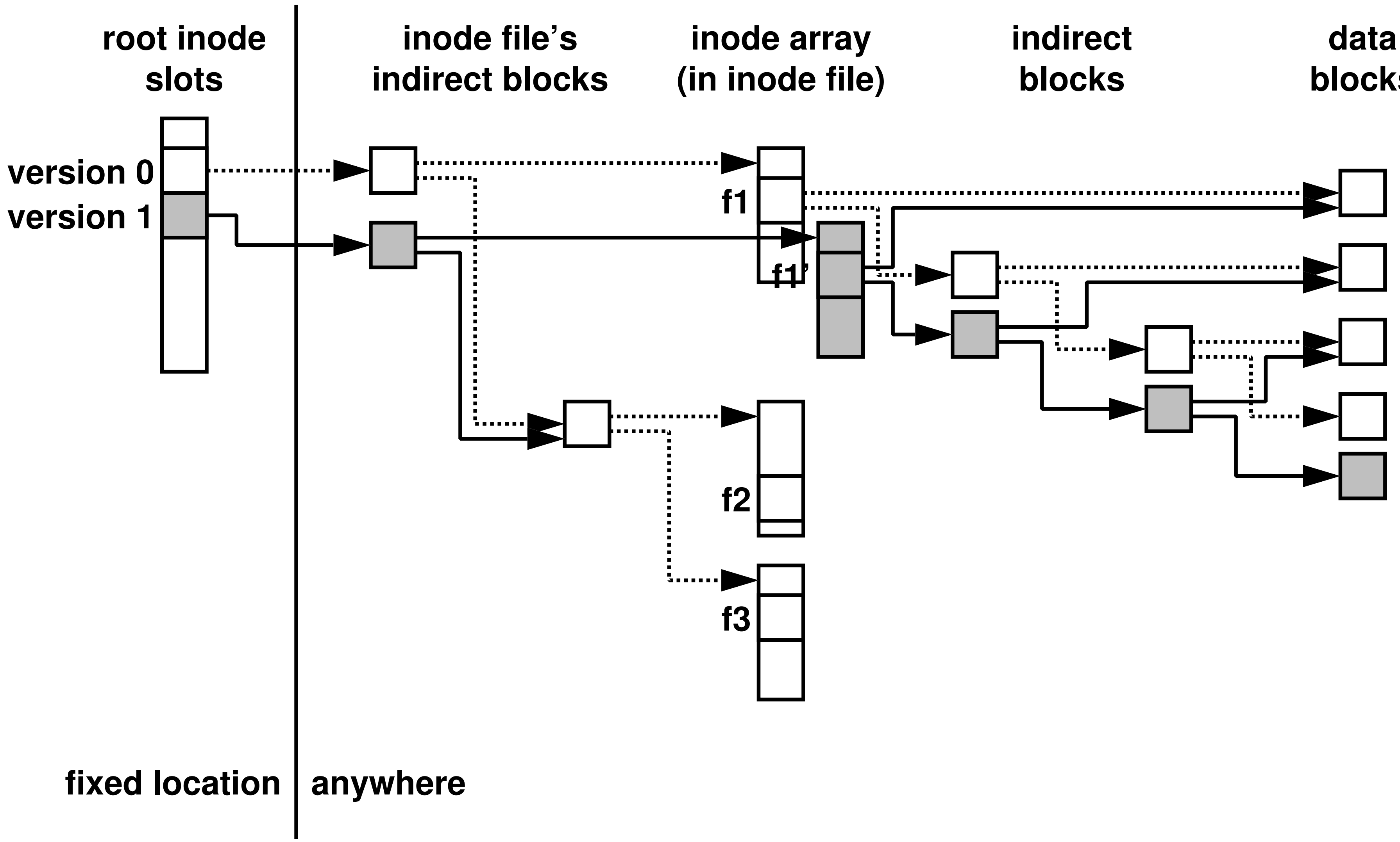


COW/Write Anywhere: Update Last Block Of F1



- Intermediate states of an update are *not observable*

COW/Write Anywhere: Update Last Block Of F1



- **Intermediate states of an update are not observable**
  - they ***atomically*** take effect ***when the root inode is updated***

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

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