Coping with System Failures: Transactions and Logging

Dave Eckhardt
Todd Mowry
Dave O'Hallaron

"If you don't know where you are going, you might not get there."
Yogi Berra

Reminder: Key File-System Data Structures

- Directories
  - Map from file names to file numbers
- Inodes
  - Map from file offsets to disk blocks
- Free-space map
  - Tracks free disk blocks

Big Picture for Today's Lecture

- What might go wrong in the file system when the system crashes?
  - e.g., while in the middle of moving a file between directories?

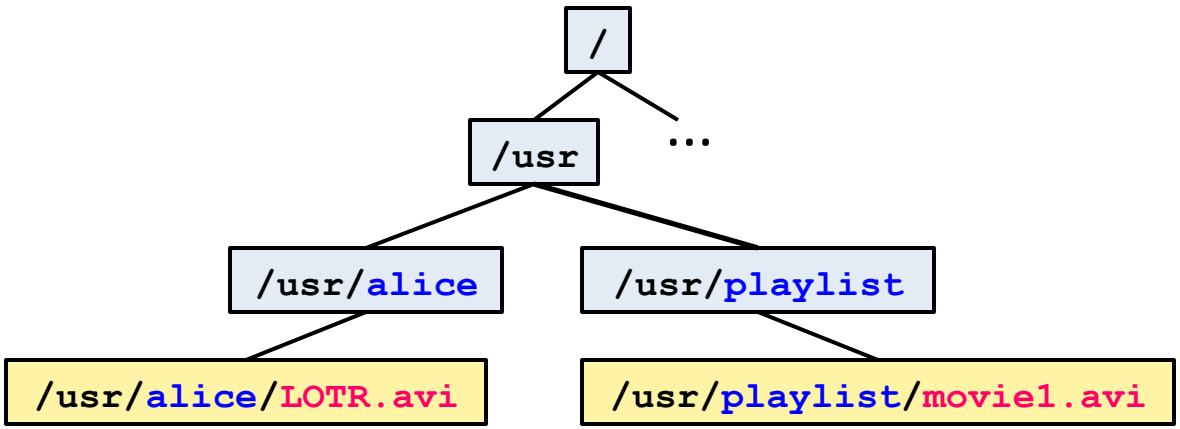
- How can we fix this problem?
  - A more ad-hoc, file-system-specific approach
    - (with some non-trivial limitations/downsides)

  - A more general approach: **Transactions**
    - including write-ahead logging
    - recurring theme in system design

- Reading material:
  - Anderson and Dahlin, "Operating Systems: Principles and Practices", 2nd Edition, Chapter 14.1 ("Transactions: Atomic Updates")

Motivating Example: Moving a File between Directories

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

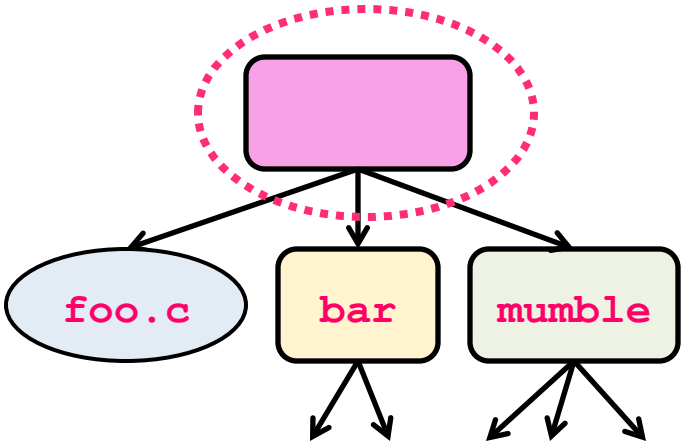


- What really happens to the file?
  - Do the bytes within the file get copied to another place on the disk?
    - No, but it gets a new name. (This operation is actually called "file rename".)
  - Where is the file name stored?
- What happens to the `/usr/alice` and `/usr/playlist` directories?
  - Remove "`LOTR.avi`" from the former, add "`movie.avi`" to the latter

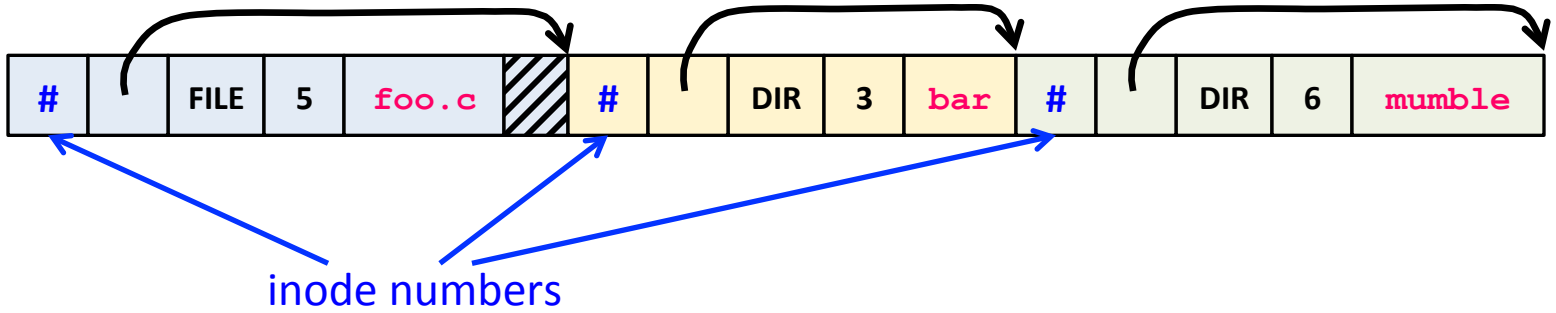
Format of Directory Entries (FreeBSD FFS)

Fields in a directory entry: (first 4 are fixed length)

1. inode number
2. size of the entry (in bytes)
3. type of the entry (e.g., file vs. directory)
4. length of the filename (in bytes)
5. filename
   - null-terminated, padded to 4-byte boundary



Directory block with 3 entries:

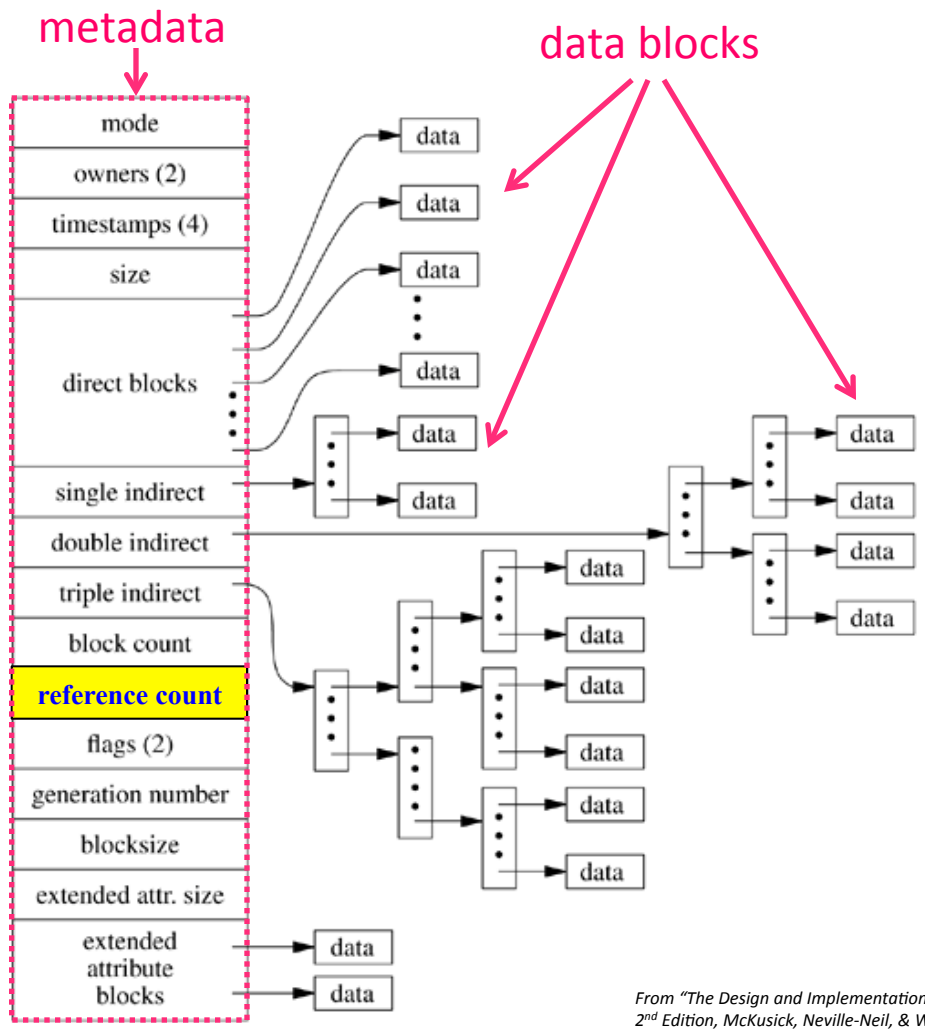


File Representation (FreeBSD FFS): inode ("index node")

- **inode** structure:

Note: *filename* is not stored here!

How many hard links point to this file?

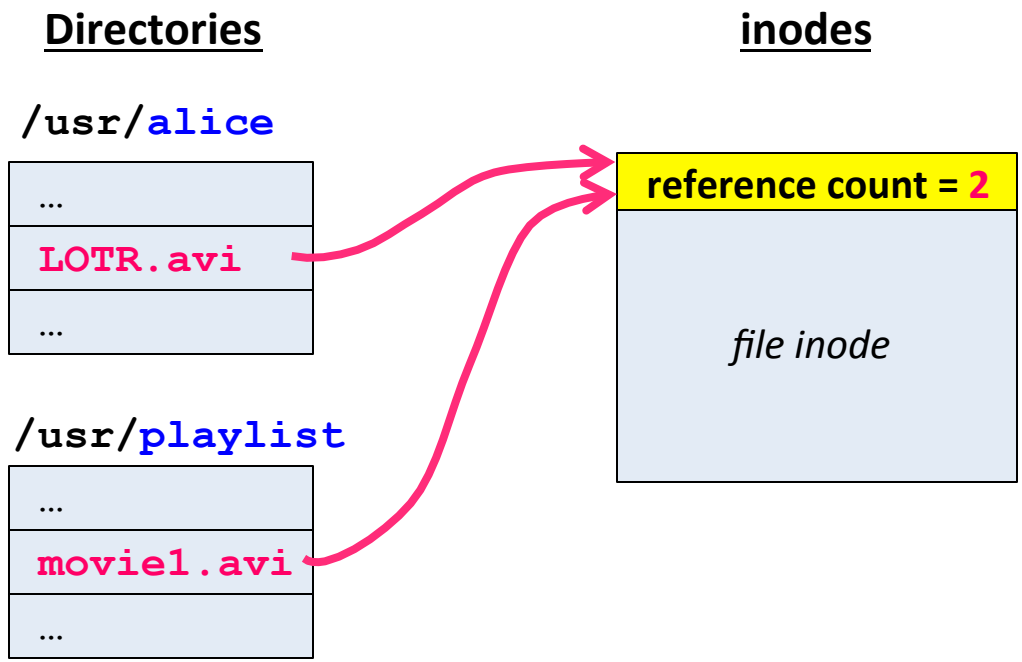


From "The Design and Implementation of the FreeBSD Operating System", 2nd Edition, McKusick, Neville-Neil, & Watson.

Multiple Hard Links to the Same File

```
> ln /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

Create hard link



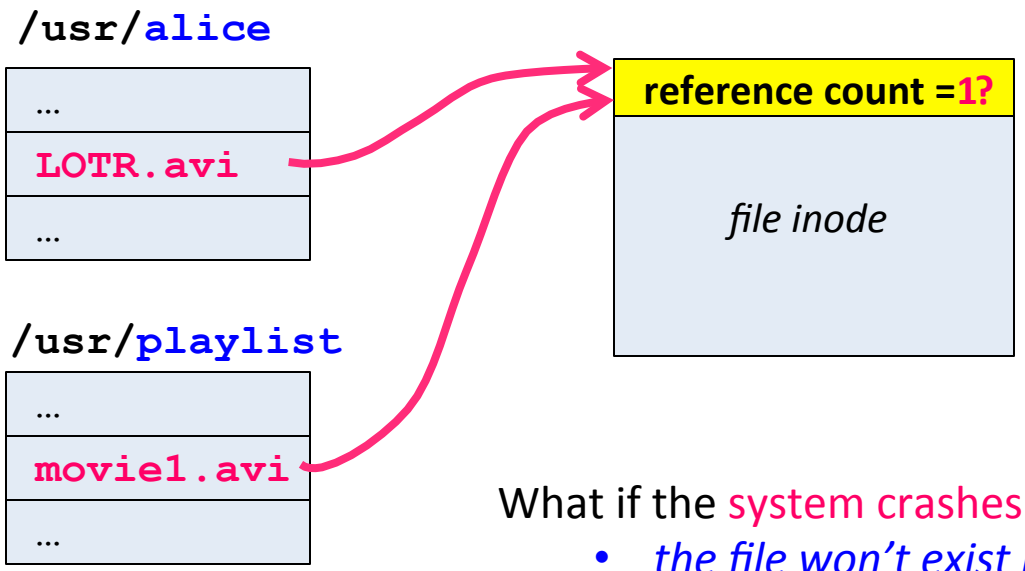
- Directory entries contain: filenames, pointers to inodes, etc.
- An inode can be deallocated when its *reference count == 0*

Steps in File Rename?

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

1st attempt:

1. Remove the `LOTR.avi` entry from the `/usr/alice` directory
2. Add a new `movie1.avi` entry to the `/usr/playlist` directory
   - that points to the original inode



What if the system crashes between steps 1 and 2?

- *the file won't exist in any directory!*

What Happens to Reference Count Between Steps 1 and 2?

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

Assume we are *between* these two steps:

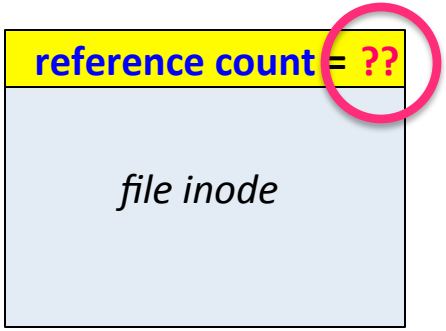
1. Remove the **`LOTR.avi`** entry from the **`/usr/alice`** directory
2. Add a new **`movie1.avi`** entry to the **`/usr/playlist`** directory
   - that points to the original inode

`/usr/alice`

…
…

`/usr/playlist`

…
…



If reference count = **0**:

- *inode (and data) might be reclaimed!*

If reference count = **1**:

- *doesn't match invariant (zero hard links to it)*

Rules for Ensuring Metadata Consistency

1. Never point to a structure before it has been initialized
   - e.g., an inode must be initialized before a directory entry references it
2. Never reuse a resource before nullifying all previous pointers to it
   - e.g., an inode's pointer to a data block must be nullified before that disk block may be reallocated for a new inode
3. Never reset the old pointer to a live resource before the new pointer has been set
   - e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written

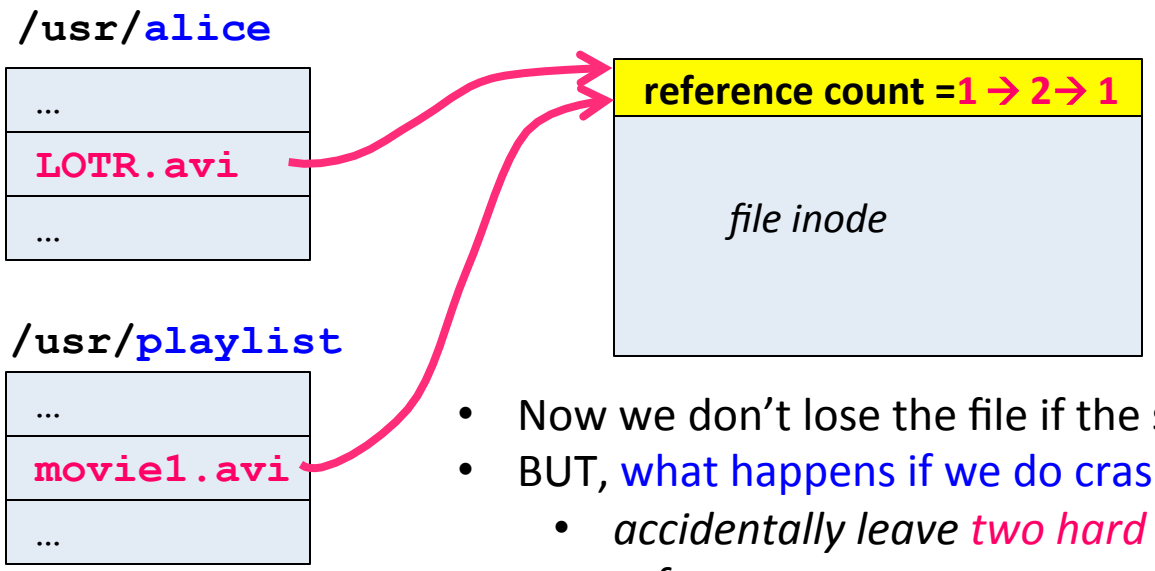
With that in mind, let's revisit the "file rename" implementation…

Revised Implementation of File Rename

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

Steps:

1. Increment reference count
2. Add a new `movie1.avi` entry to the `/usr/playlist` directory
3. Remove the `LOTR.avi` entry from the `/usr/alice` directory
4. Decrement reference count



- Now we don't lose the file if the system crashes
- BUT, what happens if we do crash between steps?
  - *accidentally leave two hard links to the file?*
  - *reference count may never go to zero (leakage)*

One Approach: Scan All Metadata to Find and Fix Inconsistencies

- Example: **`fsck`**
  - traditionally run on reboot by UNIX-based systems after a non-clean shutdown
- Examine all metadata (directories, inodes, etc.), looking for inconsistent state
  - when found, patch things up to restore invariants

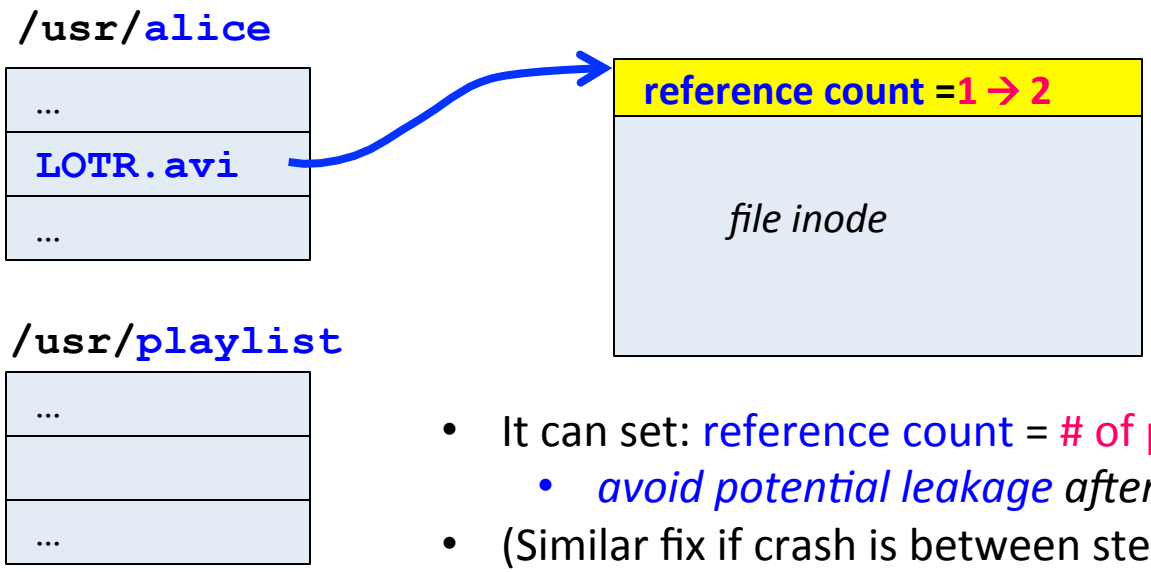
- Note: "restoring invariants" doesn't mean that operations completed properly
  - it simply fixes some obvious problems
  - doesn't know what operations were *partially completed* upon the crash

What would `fsck` do after a crash in the middle of File Rename?

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

Steps:

1. Increment reference count
2. Add a new `movie1.avi` entry to the `/usr/playlist` directory
3. Remove the `LOTR.avi` entry from the `/usr/alice` directory
4. Decrement reference count



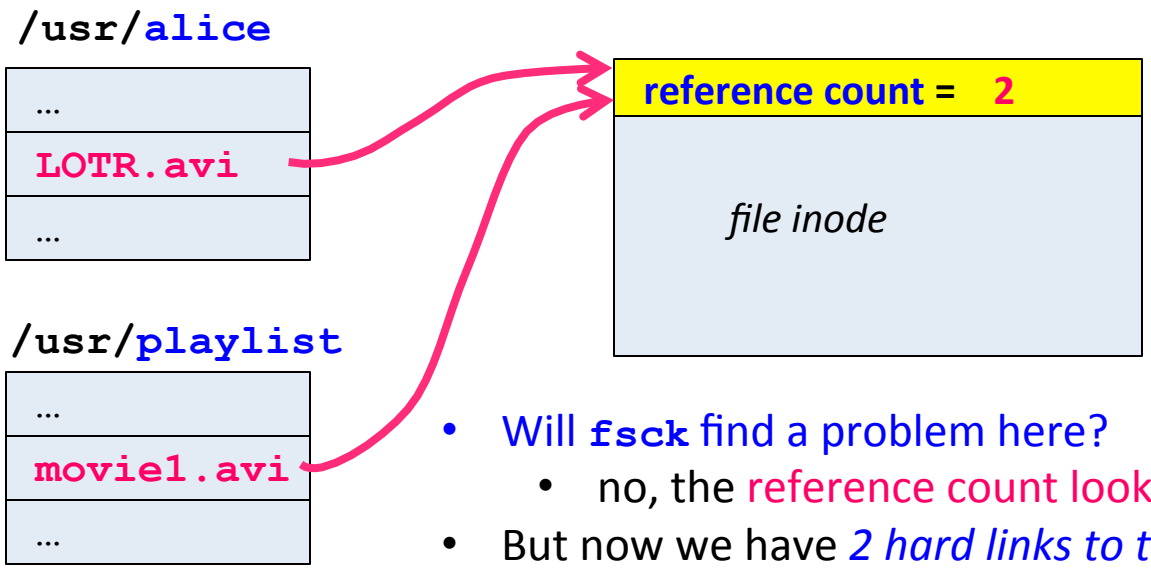
- It can set: reference count = # of pointers to inode
  - *avoid potential leakage after file deletion*
- (Similar fix if crash is between steps #3 and #4.)

What if the Crash Happens Here?

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

Steps:

1. Increment reference count
2. Add a new `movie1.avi` entry to the `/usr/playlist` directory
3. Remove the `LOTR.avi` entry from the `/usr/alice` directory
4. Decrement reference count



- Will `fsck` find a problem here?
  - no, the reference count looks fine
- But now we have *2 hard links to the same file (!)*

Limitations of the `fsck` Approach

- Finding and fixing problems:
  - only as powerful as the invariants that it understands
  - doesn't know what operations were in progress at the time of the crash

- Performance?
  - must scan all of the metadata on the disks
    - Yikes!!! That could take a *very* long time.
      - e.g., tens of minutes
    - disk capacities are increasing far more rapidly than disk bandwidth

Alternate Approach: Careful Ordering of Updates

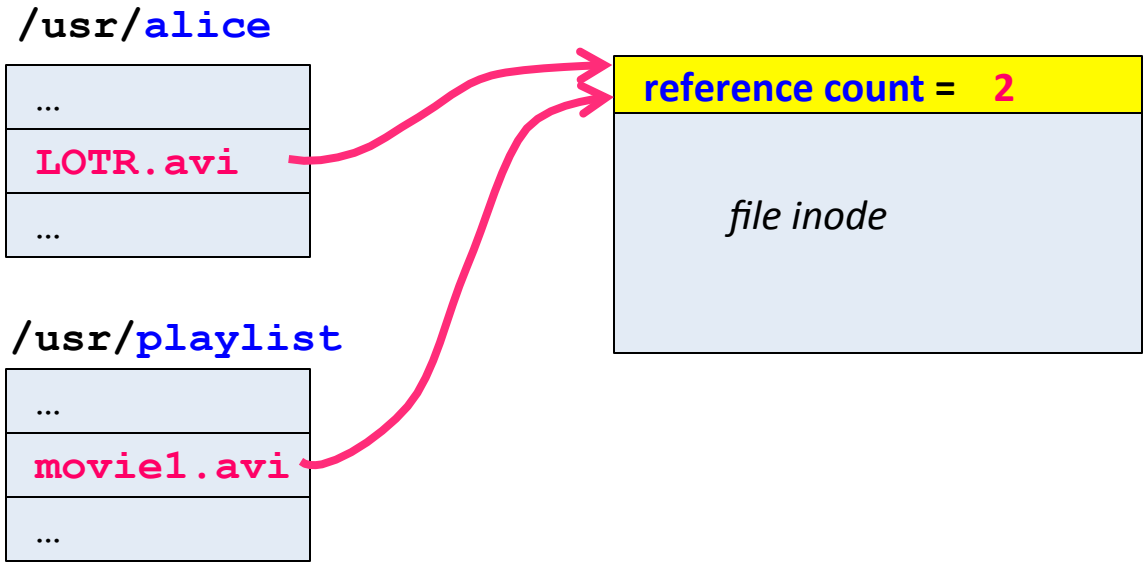
- "Soft Updates" (proposed by Prof. Greg Ganger et al.)
  - one of our reading list topics, if you are interested
    - appeared in ACM TOCS, 18(2), May 2000
- Basic idea:
  - carefully control how metadata is written back from memory to disk
  - to always obey the following types of rules:

    - Never point to a structure before it has been initialized
    - Never reuse a resource before nullifying all previous pointers to it
    - Never reset the old pointer to a live resource before the new pointer has been set

- Very interesting, but also quite specific to file system metadata
- Can we solve this problem with a more general approach?

Returning to Our File Rename Scenario

- Imagine that the file system looks like this after a crash and reboot:

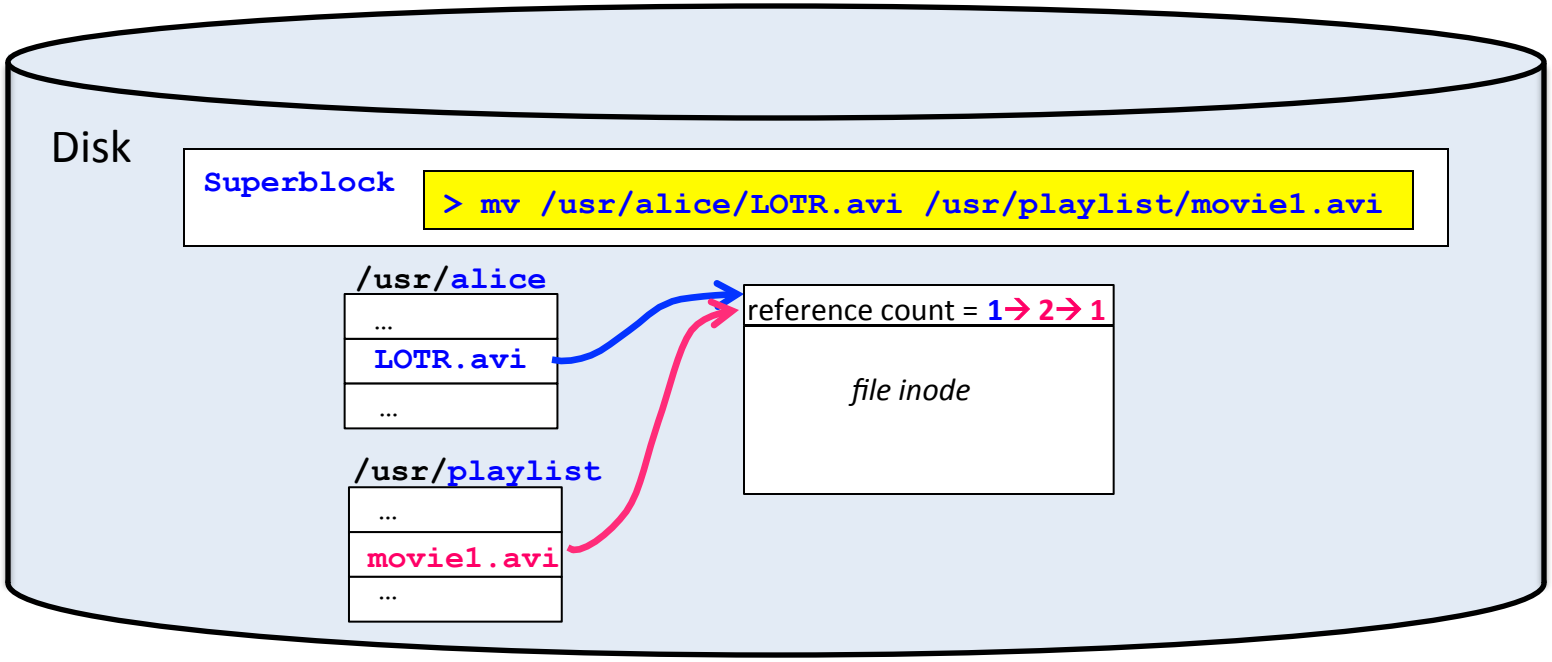


- Everything looks fine to **`fsck`** (but that is not the case!)
- Fundamental issue:
  - to fix this properly, we need to know what we were in the middle of doing

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

One Step in this Direction

- What if we fixed file rename by:
  - allowing only one file rename operation at a time
  - recording details of that operation to disk before it starts



Recovery Algorithm after a Crash

```
if (rename intent exists) {
    if (create-new-link fails) {
        // create-link could plausibly fail if disk is full
        decrement reference count;  // rename() is cleanly rolled back
    } else {
        // new link successfully created
        if (remove-old-link fails) {
            // this should NOT happen -- probably disk is broken, STOP panic
        }
    }
    // at this point rename() either 100% happened or 0% happened
    clear rename intent;
}
```

- Can we take this idea further, and generalize it?

Transactions

- A powerful technique for atomically updating persistent state
- Introduced originally in the context of databases (by Jim Gray)
  - but very useful throughout system design (including file systems, etc.)
- Mechanisms to implement transactions show up repeatedly throughout systems
  - e.g., write-ahead logging, abort/commit, two-phase locking, etc.

- Two sets of motivations/complications:
  1. Crash Recovery
     - do the right thing even if the system fails partway through the transaction
  2. Concurrency Control
     - avoid nasty surprises due to concurrent accesses to data structures

Transaction Properties

- A transaction is a way to perform a set of updates while providing the following "**ACID**" properties:
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**

- What do these mean?
  - let's go through them in a different order (ADIC)

Transaction Property: "Atomicity"

- We want "all or nothing":
  - either the entire transaction happens (commits)
  - or none of it happens (aborts)
- We should never see just part of a transaction happening

- File rename example:

1. Increment reference count
2. Add a new **`movie1.avi`** entry to the **`/usr/playlist`** directory
3. Remove the **`LOTR.avi`** entry from the **`/usr/alice`** directory
4. Decrement reference count

  - we should never end up in the accidental hard link scenario
    - between steps 2 and 3

Transaction Property: "Durability"

- Once a transaction commits, its side-effects should be persistent
  - i.e. they shouldn't suddenly disappear for no apparent reason

- How does this relate to Atomicity?
  - Durability is an orthogonal concept

- Why is Durability important?
  - Key issue: when can we safely forget about a transaction that has completed?
    - if it is not yet persistent and the system crashes, then it may be lost forever

Transaction Property: "Isolation"

- Relevant to the case where transactions are executing *concurrently*
  - (not directly relevant to crash recovery)
- Illustration: lots of concurrent node insertions into a linked list

```
insert_after(Node *prev, Node *new_node) {
    …
    Node *new_next = prev->next;
    prev->next = new_node;
    new_node->next = new_next;
}
```

…

```
insert_after(Node *prev, Node *new_node) {
    …
    Node *new_next = prev->next;
    prev->next = new_node;
    new_node->next = new_next;
}
```

1. While a concurrent transaction is executing:
   - it should not see partial results from the middle of other transactions
     - only "all or nothing"; other transactions are only visible once they *commit*
2. After a set of concurrently-running transactions finish executing:
   - it should appear as though they executed (in their entirety) one-at-a-time

Note: closely connected to how we think of Atomicity

Transaction Property: "Consistency"

- Has nothing to do with "memory consistency" on parallel machines
  - note that serializability of transactions was already covered by "Isolation"

- It has to do with preserving "integrity constraints" (invariants) in the database:
  - *"… a transaction performed on a database that is internally consistent will leave the database in an internally consistent state."* [Franklin '97]

- Unlike the other three properties, "Consistency" provides:
  - guidance for how people write transactions
  - NOT guidance for how systems designers implement transaction mechanisms

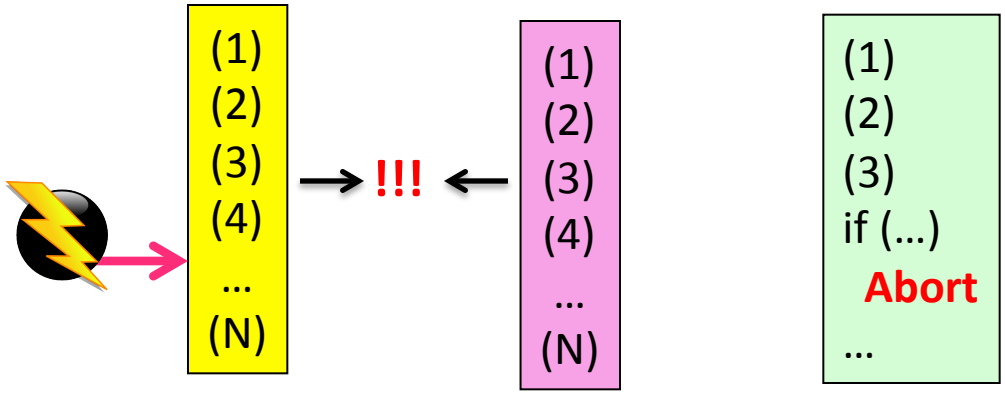
When the "ACID" Properties of Transactions Matter

- Atomicity ("all or nothing") is fundamental
- If you have concurrent transactions:
  - Isolation is important
    - natural extension of Atomicity to concurrency
- If you care about crash recovery:
  - Durability is important

Transactions in Practice:

- Transactional Memory: only Atomicity and Isolation matter (not Durability)
  - Intel TSX extensions, introduced in 2013, now disabled
- File Systems: Durability is also important!
  - Isolation matters if you perform concurrent operations
- Databases: everything is important
  - "Consistency" invariants are specific to database management

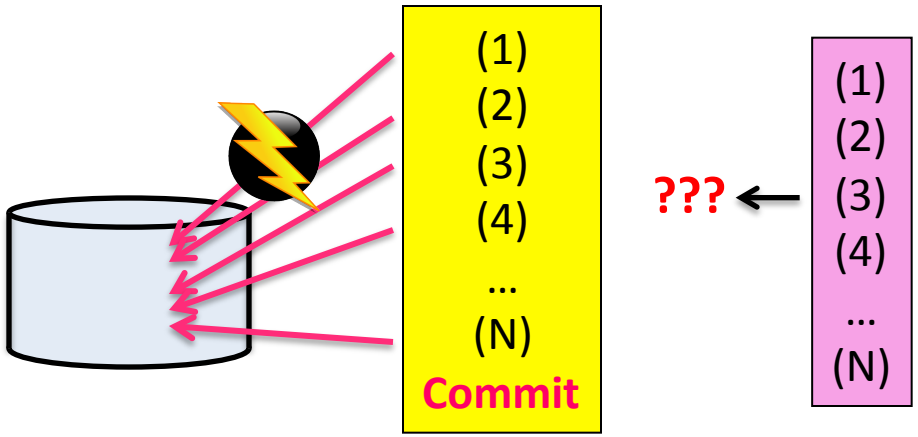
Rolling Back Transactions



- Why might we need to roll back a transaction?
  1. conflict with another concurrent transaction (optimistic concurrency)
  2. explicit software abort (e.g., a condition check failed)
  3. system crashed
     - depending upon implementation, might need to roll back to reach consistent state

- How might the system implement roll back?
  – (multiple designs are possible)

Committing Transactions



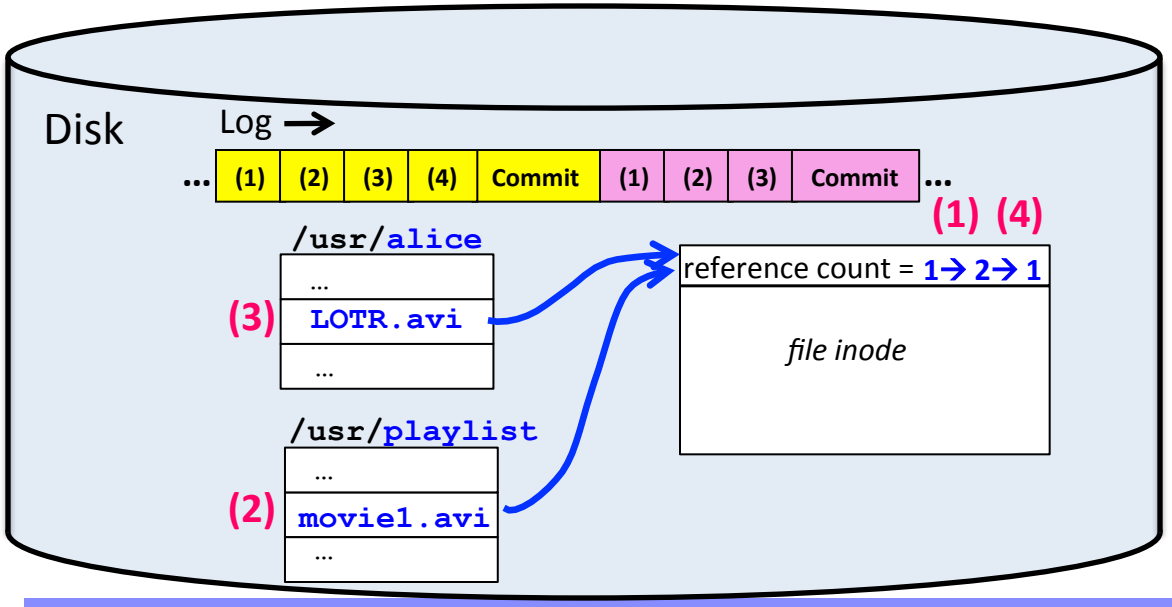
- What does it mean to "commit" a transaction?
  - Does it have something to do with making side-effects visible?
    - (Yes, but there are subtleties, as we will discuss later.)
- Isolation: have we been delaying the visibility of these side-effects until now?
- Durability: does "commit" mean "make persistent"?
- Practical issues:
  - writing all of the side-effects to non-volatile storage may take a while
  - what if we crash in the middle of these updates?

There Are Multiple Ways to Implement Transactions

- Our focus today: Transactions via **Write-Ahead Logging**
  - a common software-based approach that includes Durability

Write-Ahead Logging: Big Picture

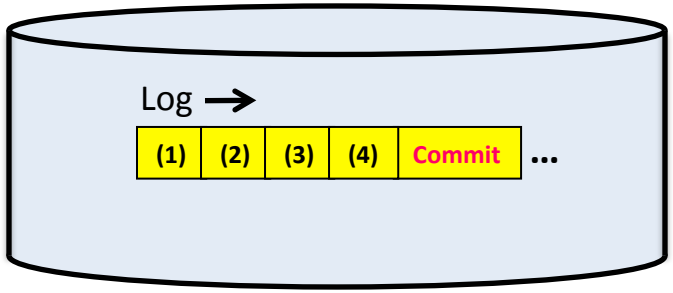
1. **Prepare**: record the changes that you are *planning to make* to a log
   - don't actually perform these changes yet!
2. **Commit**: append a "Commit" record to the log; store to non-volatile memory
3. **Writeback**: now you can perform the real updates in non-volatile memory
   - this can happen asynchronously (taking advantage of concurrency)
4. **Garbage Collect**: reclaim the completed portion of the log (usually circular buffer)



(1) Increment reference count
(2) Add a new `movie1.avi` entry to the `/usr/playlist` directory
(3) Remove the `LOTR.avi` entry from the `/usr/alice` directory
(4) Decrement reference count

(1) ...
(2) ...
(3) ...

Write-Ahead Logging: What it Means to Commit a Transaction



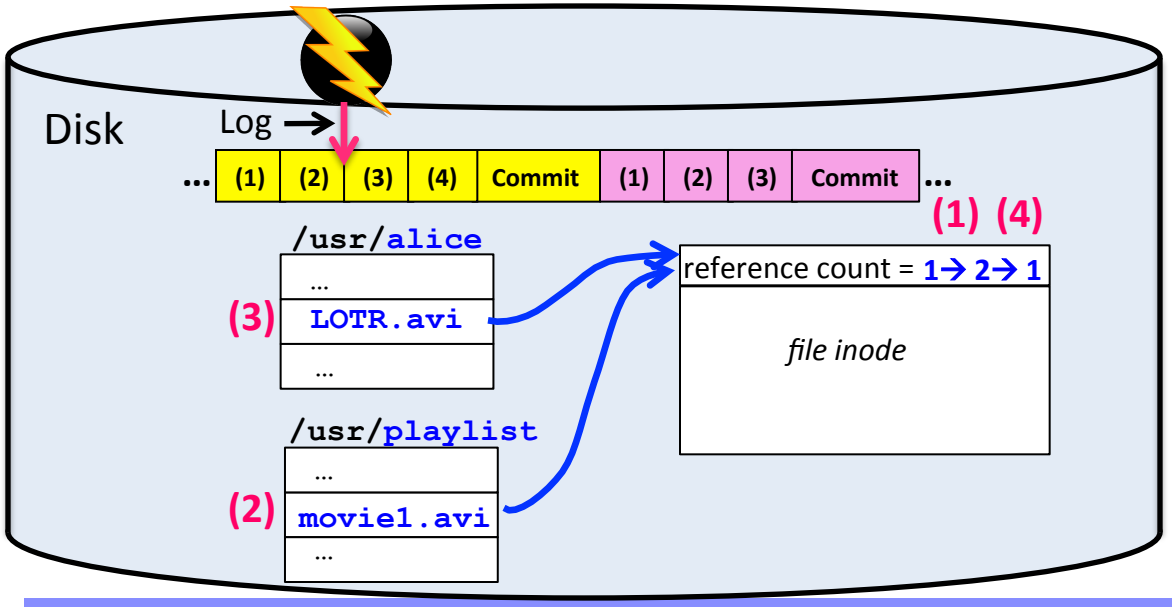
(1) Increment reference count
(2) Add a new `movie1.avi` entry to the `/usr/playlist` directory
(3) Remove the `LOTR.avi` entry from the `/usr/alice` directory
(4) Decrement reference count
COMMIT

- It does ***not*** mean waiting until all of the side-effects of the transaction are visible
  - that can take a while
  - it would greatly complicate Isolation and Durability if it did
- Commits when: a commit record is written to the log in non-volatile memory
  - this is a persistent, "point of no return" (it is now Durable)
  - the system guarantees that the side-effects will (eventually) become visible
    - with appropriate serializability with respect to other transactions (Isolation)

What Happens if the System Crashes?

Recovery Phase:

- Use the log to reconstruct the correct state of the system
- For committed transactions, replay planned actions
  - hence this implementation is called "redo" logging
  - these actions must be performed in a way that is *idempotent*
- For aborted (or non-committed) transactions, discard planned actions



(1) Increment reference count
(2) Add a new `movie1.avi` entry to the `/usr/playlist` directory
(3) Remove the `LOTR.avi` entry from the `/usr/alice` directory
(4) Decrement reference count

(1) ...
(2) ...
(3) ...

Questions

- What happens if the machine crashes:
  - before a transaction starts?
  - after a transaction starts, but before operations are logged?
  - after operations are logged, but before commit?
  - after commit, but before write back?
  - after write back, but before garbage collection?

What if the System Crashes During Recovery?

- Recall the recovery procedure for a redo log:
  - foreach transaction entry in the log (from oldest to newest):
    - if the transaction committed, then writeback planned actions
    - else discard the planned actions
- Consider our original file rename example:

```
> mv /usr/alice/LOTR.avi /usr/playlist/movie1.avi
```

(1) Increment reference count in inode
(2) Add a new `movie1.avi` entry to the `/usr/playlist` directory
(3) Remove the `LOTR.avi` entry from the `/usr/alice` directory
(4) Decrement reference count in inode

- are there any issues with this?
  - what if we crash between steps (1) and (4) during recovery?
- Planned actions in log must be *idempotent* in order to safely restart recovery
  - "increment reference count" → "set reference count = 2"
  - "decrement reference count" → "set reference count = 1"

Impact of Write-Ahead Logging on Performance?

- Logging involves some additional work: does this hurt performance?
  - (i.e. write intended action to log, and later perform that action)

- Usually not, because there is lots of good news:
  - The log is written to *sequentially*
    - great spatial locality; ideal for writing to flash storage
  - Write backs can occur *asynchronously*
    - any order is okay as long as:
      - all planned changes are logged before commit, and
      - all write backs occur after commit
    - usually much faster than waiting for a series of synchronous writes to complete
  - Can process multiple transactions at the same time
    - Record transaction ID in each log entry
    - A transaction has completed iff its commit record is in the log

File Systems Today

- Most modern file systems use one of these two flavors of transactions:
  1. Use transactions to protect file system metadata (but not regular file data)
     - that way the file system's structure is never corrupted after a crash
       - even though the contents of user files may be inconsistent
     - e.g., Microsoft's NTFS, Apple's HFS+, Linux's XFS, JFS, and ReiserFS
     - sometimes called "**Journaling**"

  2. Use transactions to protect both file system metadata and file data
     - better consistency after a crash, but more overhead
     - e.g., Linux's ext3 and ext4 support both this mode and the metadata-only mode
     - sometimes called "**Logging**"

Summary

- Recovering from crashes: file rename example

- Check invariants after a crash: **`fsck`**
  - won't notice accidental hard link in example; too costly to scan entire disk

- More powerful approach: **Transactions**
  - ACID properties (especially: Atomicity, Isolation, and Durability)
  - Write Ahead Logging:
    - record planned updates; record commit record; write back; garbage collect
  - Crash Recovery:
    - replay idempotent updates for committed transactions
  - Modern file systems:
    - Log-Structured File Systems, "Journaling", "Logging"

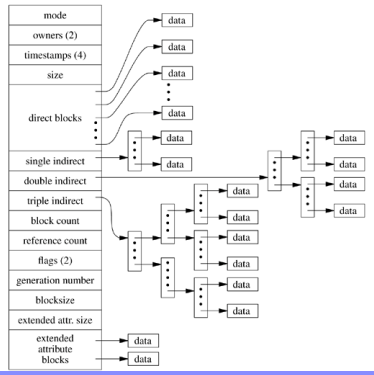
Applying Transactions to File Systems: A Semi-Radical Thought

- Do we actually need to perform the "write back" step (after the commit)?
  - what if "update in place" is very expensive?
    - e.g., random writes to Flash storage are very costly (write amplification)
  - in contrast, strictly sequential writes (for the log) are usually much cheaper

- What if the log *is* the representation of the data storage?
  - **Log-Structured File Systems**

Impact of Maintaining Only the Log

- Performance of Writes:
  - should improve, since the overhead of writebacks is eliminated
    - and because the log can be written sequentially in large chunks
- Performance of Reads?
  - how do we locate the data?
    - do we need to scan through much of the log to find it?
  - if we had arbitrary-sized updates of arbitrary data structures (e.g., TM):
    - this could be very costly
  - in file systems: all updates (data and metadata) are to fixed-size blocks
    - hence we simply need to find the right inode, and then everything works as normal



Impact of Maintaining Only the Log

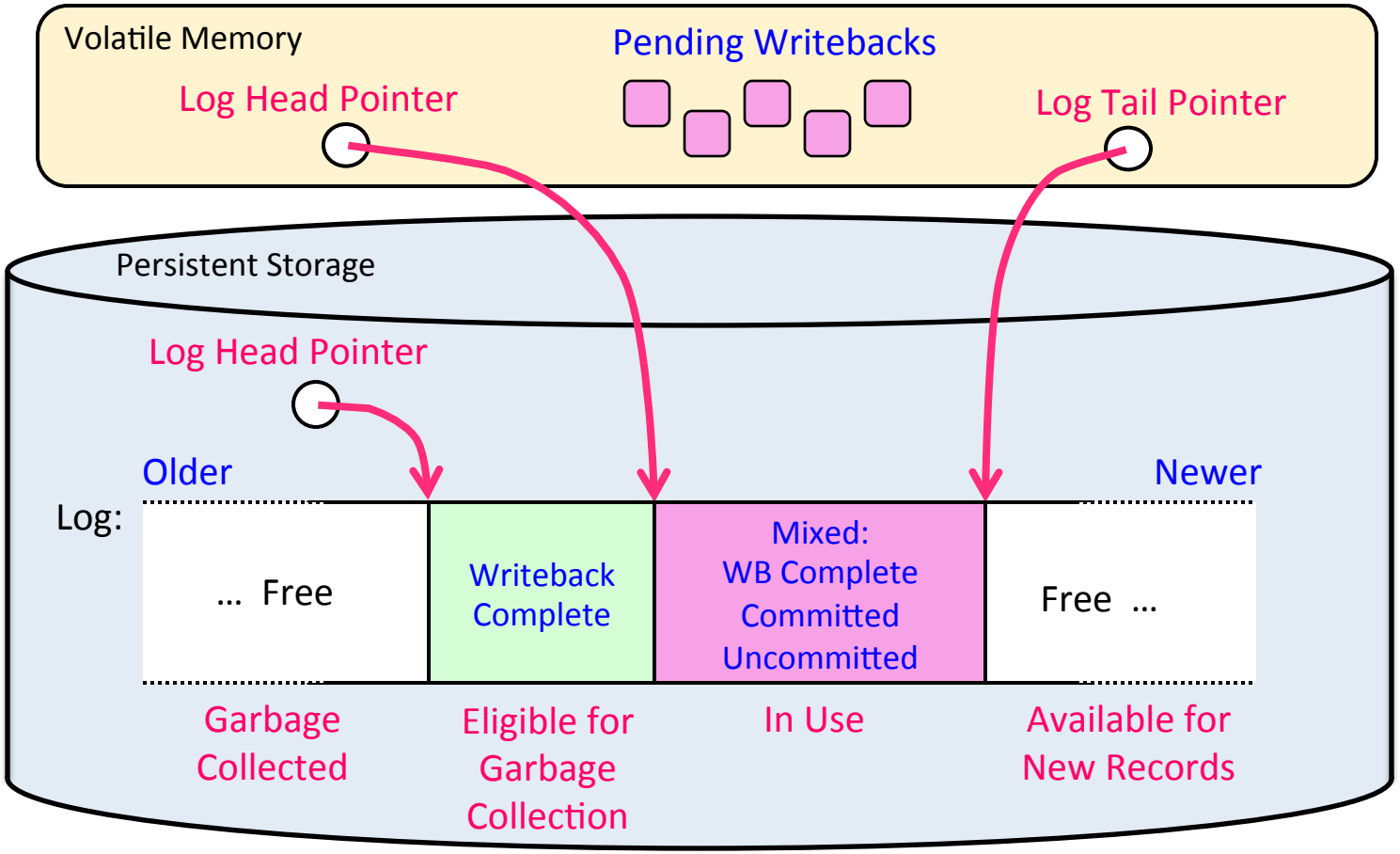
- Performance of Writes:
  - should improve, since the overhead of writebacks is eliminated
    - and because the log can be written sequentially in large chunks
- Performance of Reads?
  - how do we locate the data?
    - do we need to scan through much of the log to find it?
  - if we had arbitrary-sized updates of arbitrary data structures (e.g., TM):
    - this could be very costly
  - in file systems: all updates (data and metadata) are to fixed-size blocks
    - hence we simply need to find the right inode, and then everything works as normal

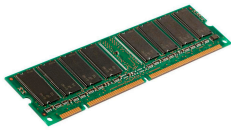
- Does the log continue to grow forever?
  - what would happen with a traditional circular buffer?

Log-Structured File Systems

- Log is the data storage; no write 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

Redo Logging Implementation





- Tail pointer must not pass head pointer: *which* head pointer?

Transaction Isolation: Motivating Example

- Two processes (A and B) performing *concurrent* transactions

Process A:

- move **`myfile`** from **`dir1`** to **`dir2`**

```
> mv dir1/myfile dir2
```

Process B:

- grep across **`dir1`** and **`dir2`**

```
> grep … dir1/* dir2/* > results
```

- What are some nasty surprises that might occur without proper Isolation?
  - how many times might grep matches from **`myfile`** appear in **`results`**?
  - and with what pathname?
    - **`dir1/myfile`**? **`dir2/myfile`**?

- What are valid outcomes with proper isolation?
  - grep matches from **`myfile`** must appear only once
  - from *either* **`dir1/myfile`** (B precedes A) or **`dir2/myfile`** (A precedes B)

One Way to Implement Isolation: Two Phase Locking

Two-Phase Locking:

- **Expanding** phase: locks may be acquired (but not released)
- **Contracting** phase: locks may be released (but not acquired)

To implement serializability (strong form of Isolation) in transactions:

- Contracting phase must wait until transaction commits (or rolls back)
  - this prevents other transactions from seeing updates that later disappear

Process A:

```
LOCK dir1, dir2
> mv dir1/myfile dir2
COMMIT
UNLOCK dir1, dir2
```

Process B:

```
LOCK dir1, dir2, results
> grep … dir1/* dir2/* > results
COMMIT
UNLOCK dir1, dir2, results
```

Now the grep will occur either before or after the move.

What about Deadlock?

Process A:

```
LOCK dir1, dir2
> mv dir1/myfile dir2
COMMIT
UNLOCK dir1, dir2
```

Process B:

```
LOCK dir1, dir2, results
> grep … dir1/* dir2/* > results
COMMIT
UNLOCK dir1, dir2, results
```

- Can we have deadlock with two phase locking?
- Fortunately, we can implement Deadlock Prevention
  - which of the four necessary conditions can we eliminate?
  - No Preemption:
    - roll back one (or more) transactions, release their locks, and restart them later

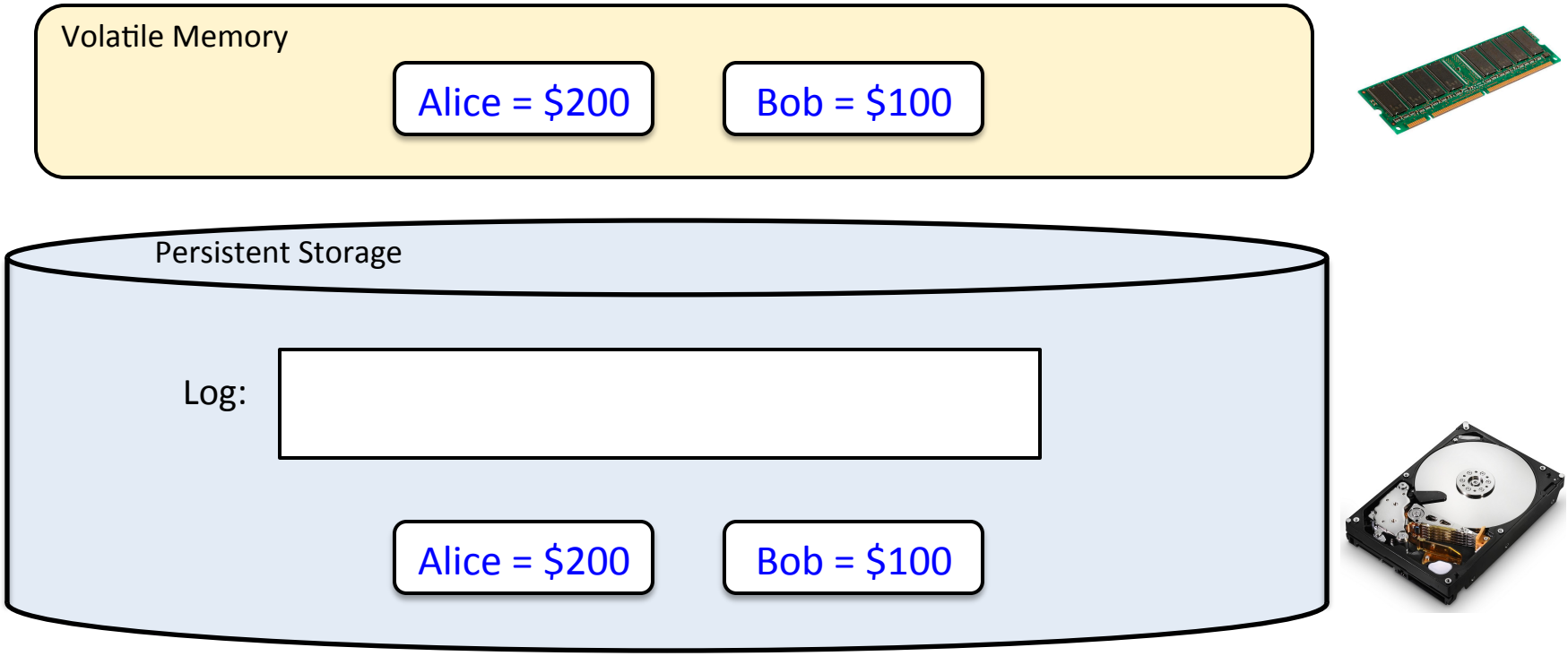
Serializability

- Redo Logging + Two Phase Locking → transactions appear to occur *sequentially*

- Other implementations can also provide serializability:
  - Optimistic Concurrency Control:
    - roll back any transaction that would conflict with serializability

Example Transaction: Move $100 from Alice to Bob

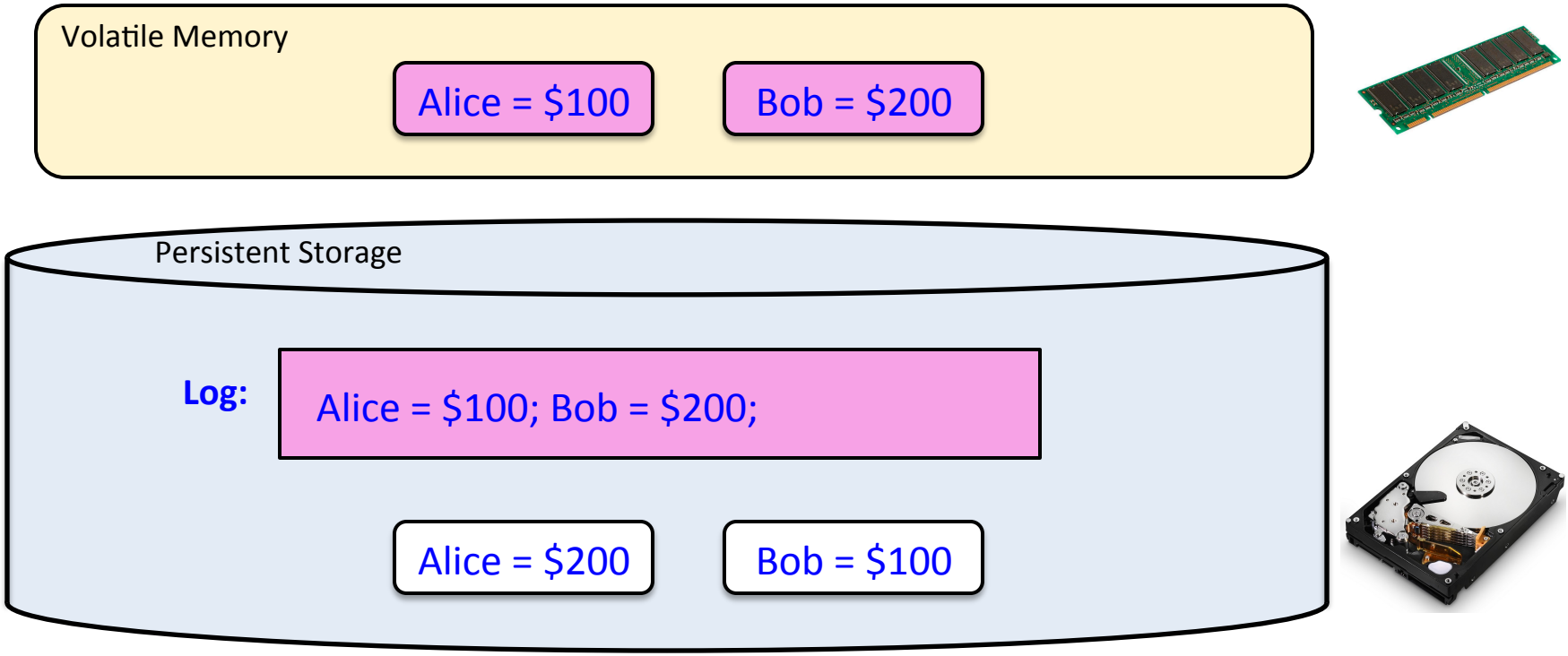
Original state:



- What happens if the machine crashes now?

Example Transaction: Move $100 from Alice to Bob

Updates appended to log:



- What happens if the machine crashes now?

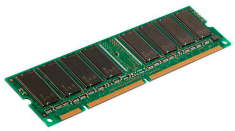
Example Transaction: Move $100 from Alice to Bob

Commit appended to log:

Volatile Memory

Alice = $100

Bob = $200



Persistent Storage

Log:

Alice = $100; Bob = $200; **COMMIT**

Alice = $200

Bob = $100

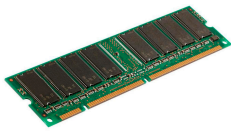
- What happens if the machine crashes now?

Example Transaction: Move $100 from Alice to Bob

Updates applied:

Volatile Memory

Alice = $100 | Bob = $200



Persistent Storage

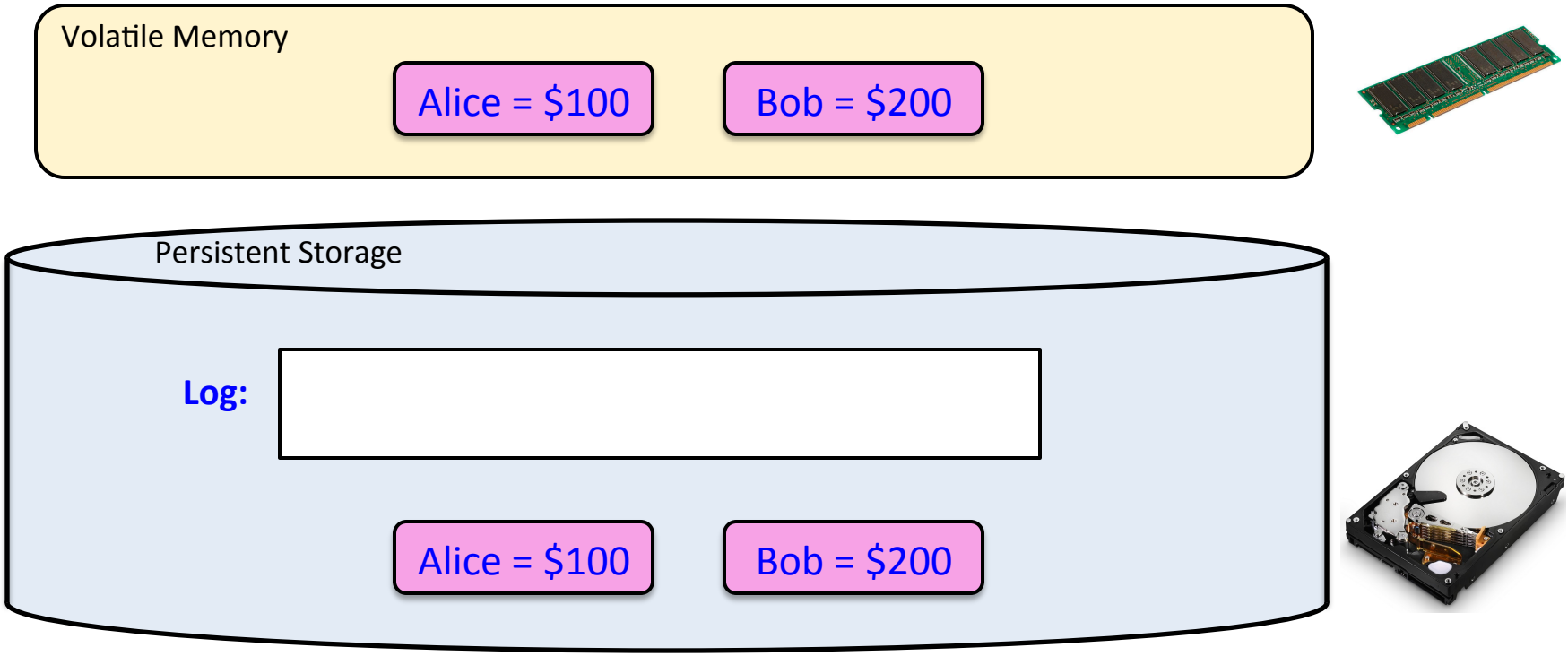
Log: Alice = $100; Bob = $200; **COMMIT**

Alice = $100 | Bob = $200

- What happens if the machine crashes now?

Example Transaction: Move $100 from Alice to Bob

Garbage collect completed transactions from the log:



- What happens if the machine crashes now?