Implementing Transactions for File System Reliability

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 27

October 31, 2014

Reading: A&D 14.1
HW 5 out
Proj 2 final 11/07

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

Achieving File System Reliability

- Problem posed by machine/disk failures
- Transaction concept
- Approaches to reliability
  - Careful sequencing of file system operations
  - Copy-on-write (WAFL, ZFS)
  - Journalling (NTFS, linux ext4)
  - Log structure (flash storage)
- Approaches to availability
  - RAID

Storage Reliability Problem

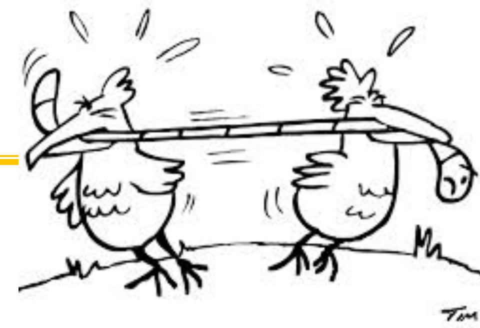
- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With remapping, single update to physical disk block can require multiple (even lower level) updates
- At a physical level, operations complete one at a time
  - Want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

The ACID properties of Transactions

Transaction is a group of operations:

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability:** if a transaction commits, its effects persist despite crashes

Fast AND Right ???



- The concepts related to transactions appear in many aspects of systems
  - File Systems
  - Data Base systems
  - Concurrent Programming
- Example of a powerful, elegant concept simplifying implementation AND achieving better performance.
- The key is to recognize that the system behavior is viewed from a particular perspective.
  - Properties are met from that perspective

Reliability

Performanc

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

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

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)

Transactional File Systems

- Journaling File System
  - Applies updates to system metadata using transactions (using logs, etc.)
  - Updates to non-directory files (i.e., user stuff) is done in place (without logs)
  - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4
- Logging File System
  - All updates to disk are done in transactions

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 in the log, it is safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended
  - Can take our time making the changes
    - As long as new requests consult the log first
- Once changes are copied, safe to remove log
- But, …
  - If the last atomic action is not done … poof … all gone

THE atomic action

- Write a sector on disk

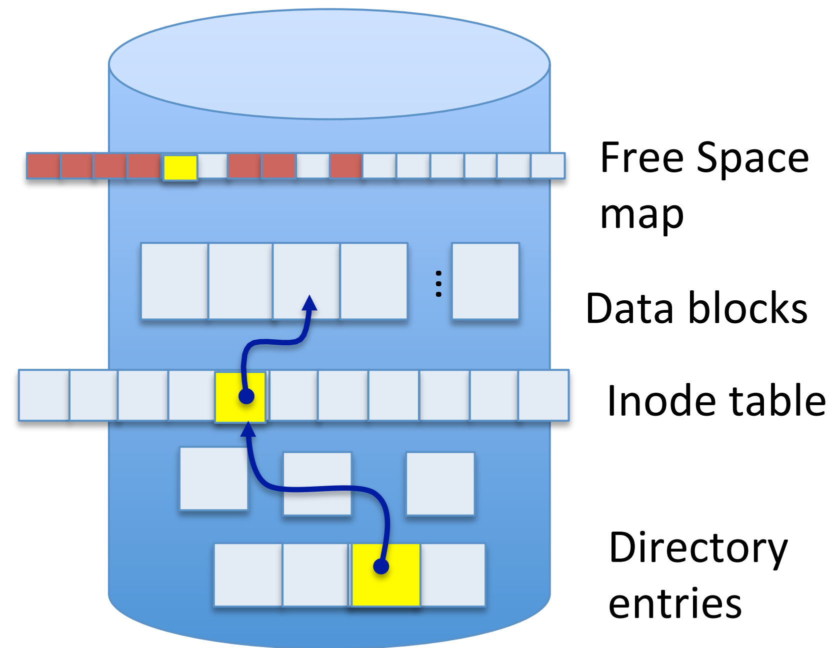
Redo Logging

- 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

Example: Creating a file

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Ex: Creating a file (as a transaction)

- Find free data block(s)
- Find free inode entry
- Find dirent insertion point

- Write map (used)
- Write inode entry to point to block(s)
- Write dirent to point to inode

Log in non-volatile storage (Flash or on Disk)

ReDo log

- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk

Crash during logging - Recover

- Upon recovery scan the long
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged

Log in non-volatile storage (Flash or on Disk)

Recovery After Commit

- Scan log, find start
- Find matching commit
- Redo it as usual
  - Or just let it happen later

Log in non-volatile storage (Flash or on Disk)

What if had already started writing back the transaction ?

- *Idempotent* – the result does not change if the operation is repeat several times.
- Just write them again during recovery

What if the uncommitted transaction was discarded on recovery?

- Do it again from scratch
- Nothing on disk was changed

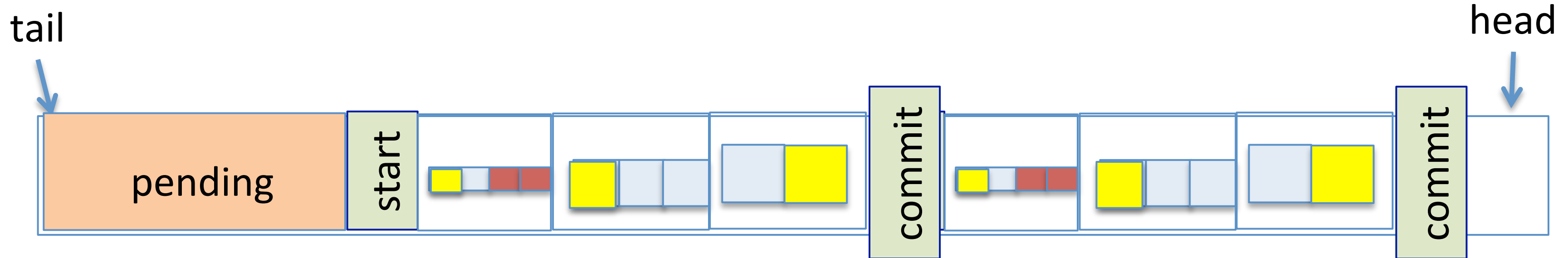
What if we crash again during recovery?

- Idempotent
- Just redo whatever part of the log hasn’t been garbage collected

Redo Logging

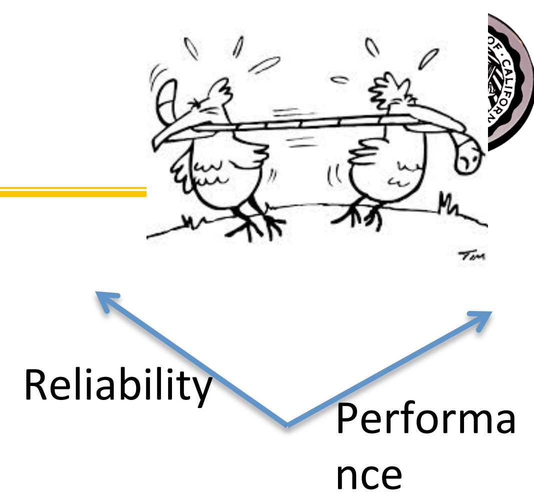
- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Ignore uncommitted ones
  - Garbage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

Can we interleave transactions in the log?



- This is a very subtle question
- The answer is "if they are serializable"
  – i.e., would be possible to reorder them in series without violating any dependences
- Deep theory around consistency, serializability, and memory models in the OS, Database, and Architecture fields, respectively
  – A bit more later --- and in the graduate course…

Back of the Envelope …



- Assume 5 ms average seek+rotation
- And 100 MB/s transfer
  - 4 KB block => .04 ms
- 100 random small create & write
  - 4 blocks each (free, inode, dirent + data)
- NO DISK HEAD OPTIMIZATION! = FIFO
  - Must do them in order
- 100 x 4 + 5 ms = 2 sec
- Log writes: 5 ms + 400 x 0.04 ms = 6.6 ms
- Get to respond to the user almost immediately
- Get to optimize write-backs in the background
  - Group them for sequential, seek optimization
- What if the data blocks were huge?

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

Volatile Memory

Log–head pointer

Pending write–backs

Log–tail pointer

Persistent Storage

Log–head pointer

Log:

...	Free	Writeback Complete	Mixed: WB Complete Committed Uncommitted	Free	...
older	Garbage Collected	Eligible for GC	In Use	Available for New Records	newer

Isolation

<table>
<tr>
<td>

Process A:

move foo from dir x to dir y

```
mv x/foo y/
```

</td>
<td>

Process B:

grep across a and b

```
grep 162 x/* y/* > log
```

</td>
</tr>
</table>

- Assuming 162 appears only in foo,
- what are the possible outcomes of B without transactions?
- If x, y and a,b are disjoint
- If x == a and y == b

Isolation

Process A:	Process B:
move foo from dir x to dir y	grep across x and y
`mv x/foo y/`	`grep 162 x/* y/* > log`

- Assuming 162 appears only in foo,
- And A is done as a transaction
- What if grep starts after the changes are loggend but before they are committed?
- Must prevent the interleaving
- Also what we do to isolate transactions

What do we use to prevent interleaving?

- Locks!
- But here we need to acquire multiple locks
- We didn't cover it specifically, but wherever we are acquiring multiple locks there is the possibility of deadlock!
  - More on how to avoid that later

Two-Phase Commit (2PC)

- Acquire all the locks & log the transaction
- Then commit
- And release the locks

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

Locks – in a new form

- "Locks" to control access to data

- Two types of locks:
  - shared (S) lock – multiple concurrent transactions allowed to operate on data
  - exclusive (X) lock – only one transaction can operate on data at a time

Lock Compatibility Matrix

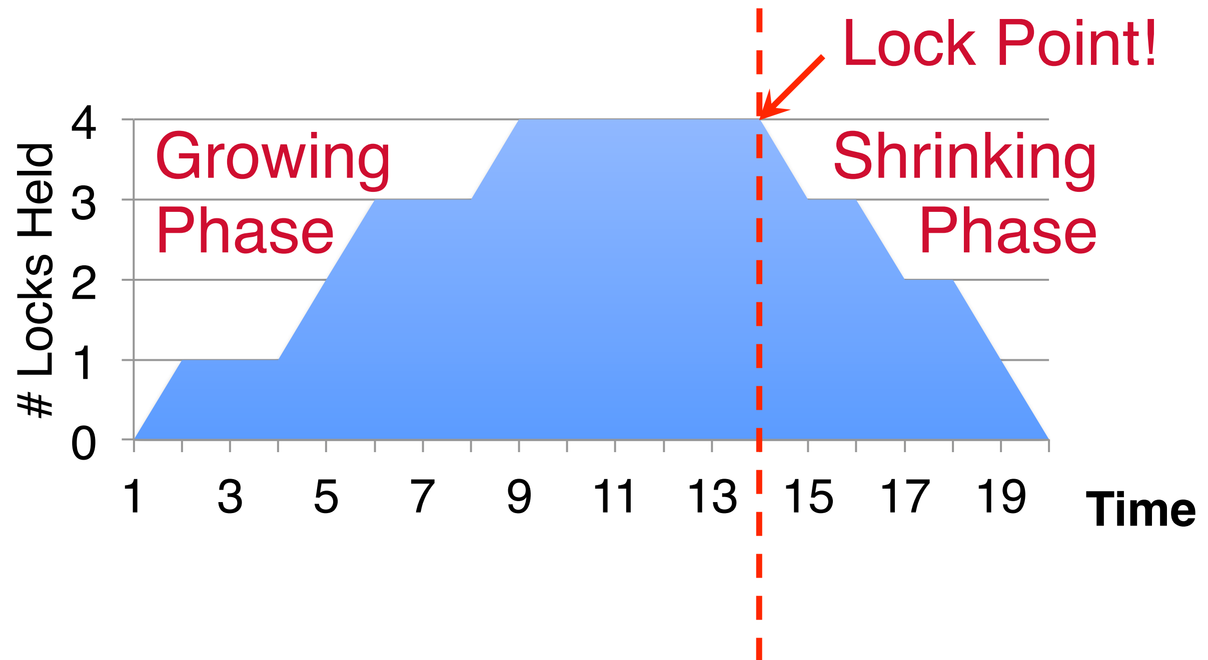
	S	X
S	√	–
X	–	–

Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - S (*shared*) or X (*exclusive*) lock on data before reading,
   - X (*exclusive*) lock on data before writing
2) A transaction can not request additional locks once it releases any locks

Thus, each transaction has a "growing phase" followed by a "shrinking phase"

Avoid deadlock by acquiring locks in some lexicographic order



Two-Phase Locking (2PL)

- 2PL guarantees that the dependency graph of a schedule is acyclic.
- For every pair of transactions with a conflicting lock, one acquires is first → ordering of those two → total ordering.
- Therefore 2PL-compatible schedules are conflict serializable.
- Note: 2PL can still lead to deadlocks since locks are acquired incrementally.
- An important variant of 2PL is **strict 2PL**, where all locks are released at the end of the transaction.

Transaction Isolation

Process A:	Process B:
LOCK x, y	LOCK x, y and `log`
move foo from dir x to dir y	grep across x and y
`mv x/foo y/`	`grep 162 x/* y/* > log`
Commit and Release x, y	Commit and Release x, y, log

- grep appears either before or after move
- Need log/recover AND 2PL to get ACID

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
  - If the operations from different transactions get interleaved in the log, it is because it is OK
    - 2PL prevents it if serializability would be violated
    - Typically, because they were independent
- Other implementations can also provide serializability
  - Optimistic concurrency control: abort any transaction that would conflict with serializability

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
  - Historical artifact ? - quite likely
  - Unfamiliar model (other than within OS's and DB's)?
    - perhaps

Atomicity

- A transaction
  - might *commit* after completing all its operations, or
  - it could *abort* (or be aborted) after executing some operations

- Atomic Transactions: a user can think of a transaction as always either *executing all its* operations, or *not executing any* operations at all
  - Database/storage system *logs* all actions so that it can *undo* the actions of aborted transactions

Consistency

- Data follows integrity constraints (ICs)
- If database/storage system is consistent before transaction, it will be after
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted)
  - A database enforces some ICs, depending on the ICs declared when the data has been created
  - Beyond this, database does not understand the semantics of the data (e.g., it does not understand how the interest on a bank account is computed)

Isolation

- Each transaction executes as if it was running by itself
  - It cannot see the partial results of another transaction
- Techniques:
  - Pessimistic – don’t let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen

Durability

- Data should survive in the presence of
  - System crash
  - Disk crash → need backups

- All committed updates and only those updates are reflected in the file system or database
  - Some care must be taken to handle the case of a crash occurring during the recovery process!