CS162
Operating Systems and Systems Programming
Lecture 20

Reliability, Transactions
Distributed Systems

April 13th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: File System Caching

- **Buffer Cache:** Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- **Read Ahead Prefetching:** fetch sequential blocks early
  - Exploit fact that most common file access is sequential
  - Elevator algorithm can efficiently interleave prefetches from different apps
  - How much to prefetch? It's a balance!
- **Delayed Writes:** Writes not immediately sent to disk
  - write() copies data from user space buffer to kernel buffer
    - » Other applications read data from cache instead of disk
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

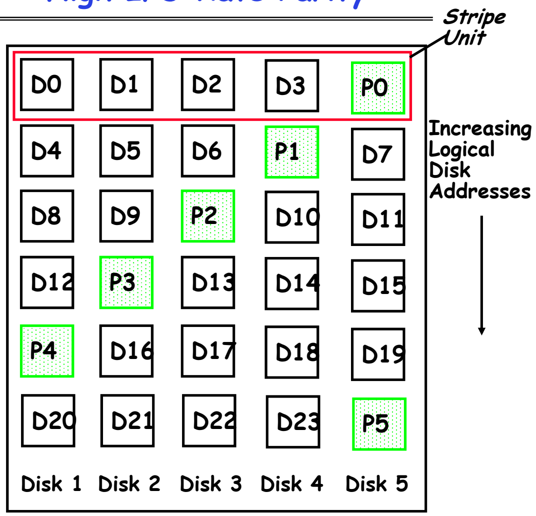
Recall: Important "ilities"

- **Availability:** the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

---

Recall: RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: D3=D0⊕D1⊕D2⊕P0



Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
D0	D1	D2	D3	P0
D4	D5	D6	P1	D7
D8	D9	P2	D10	D11
D12	P3	D13	D14	D15
P4	D16	D17	D18	D19
D20	D21	D22	D23	P5

- **Raid 6: More powerful code ⇒ can lose 2 disks of stripe**

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
- Having RAID doesn't necessarily protect against all such failures
  - Bit-for-bit protection of bad state?
  - What if one disk of RAID group not written?
- File system wants durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

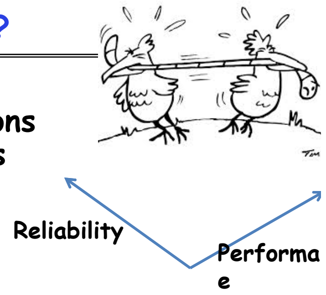
Storage Reliability Problem

- 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?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an *inconsistent state*.
  - e.g.: transfer funds from BofA to Schwab. What if transfer is interrupted after withdrawal and before deposit
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

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 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

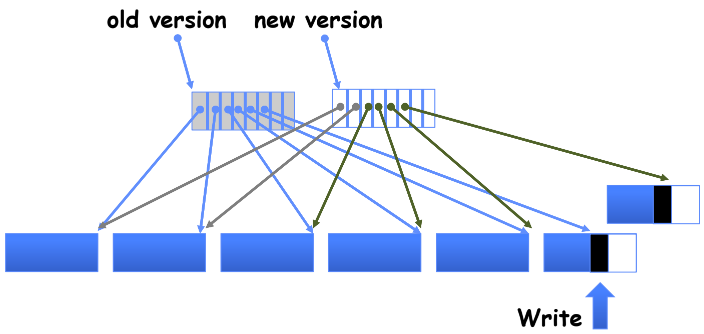
- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances (WAFL, ZFS)

Emulating COW @ user level

- Transform file foo to a new version
- Open/Create a new file foo.v
  - where v is the version #
- Do all the updates based on the old foo
  - Reading from foo and writing to foo.v
  - Including copying over any unchanged parts
- Update the link
  - ln –f foo foo.v

- Does it work?
- What if multiple updaters at same time?
- How to keep track of every version of file?
  - Would we want to do that?

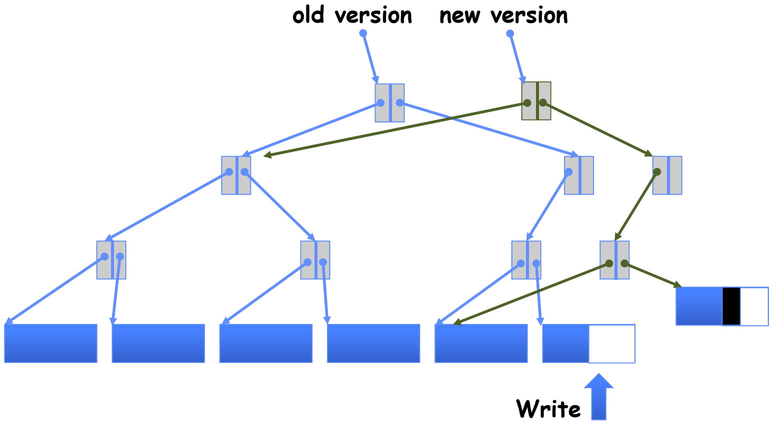
COW integrated with file system



- If file represented as a tree of blocks, just need to update the leading fringe

---

COW with smaller-radix blocks



- If file represented as a tree of blocks, just need to update the leading fringe

---

ZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

More General Solutions

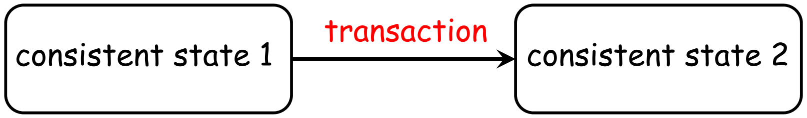
- Transactions for Atomic Updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either *all or none* of the updates
  - Most modern file systems use transactions internally to update the many pieces
  - Many applications implement their own transactions
- Redundancy for media failures
  - Redundant representation (error correcting codes)
  - Replication
  - E.g., RAID disks

Transactions

- Closely related to critical sections in manipulating shared data structures
- Extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Like flags for threads, many ad hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error, -- fsck
  - Applications use temporary files and rename

Key concept: Transaction

- An atomic sequence of actions (reads/writes) on a storage system (or database)
- That takes it from one consistent state to another



Typical Structure

- **Begin** a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, **roll-back**
  - Or, if any conflicts with other transactions, **roll-back**
- **Commit** the transaction

"Classic" Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';

 UPDATE branches SET balance = balance - 100.00
   WHERE name = (SELECT branch_name FROM accounts
   WHERE name = 'Alice');

 UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';

 UPDATE branches SET balance = balance + 100.00
   WHERE name = (SELECT branch_name FROM accounts
   WHERE name = 'Bob');


COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

The ACID properties of Transactions

- **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

Transactional File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- Journaling File System
  - Applies updates to system metadata using transactions (using logs, etc.)
  - Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional
  - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4
- Full 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
  - Single commit record commits transaction
- 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
- Basic assumption:
  - Updates to sectors are atomic and ordered
  - Not necessarily true unless very careful, but key assumption

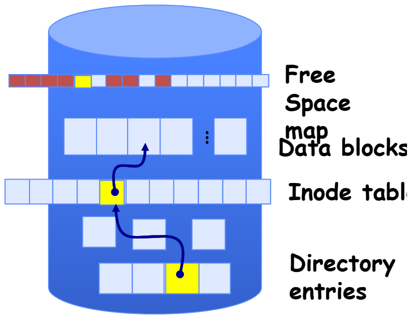
Redo Logging

- **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

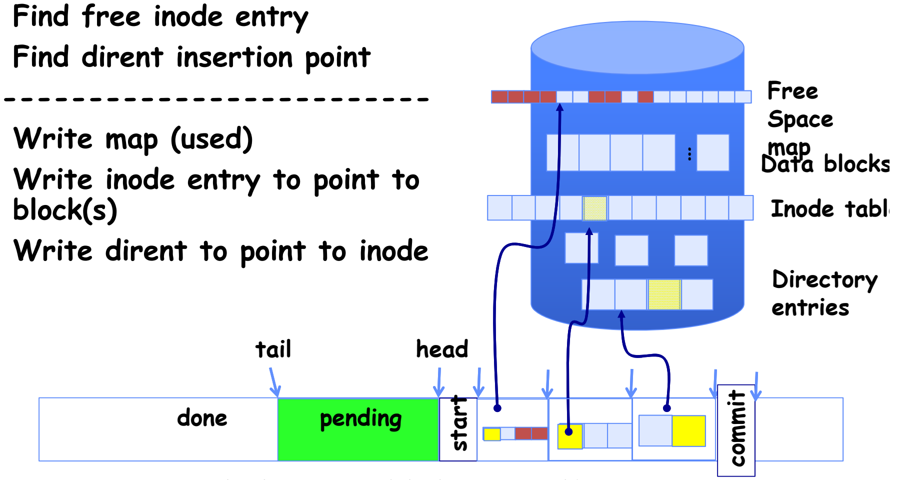
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Ex: Creating a file (as a transaction)

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

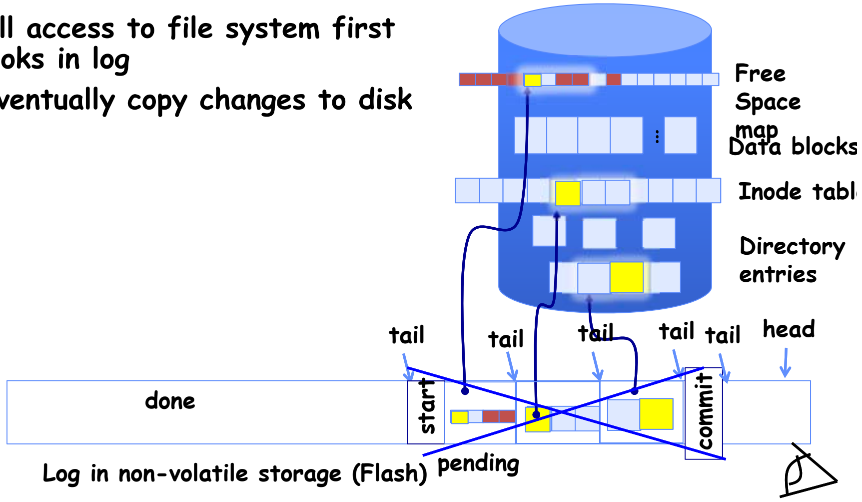
- Write map (used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



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

ReDo log

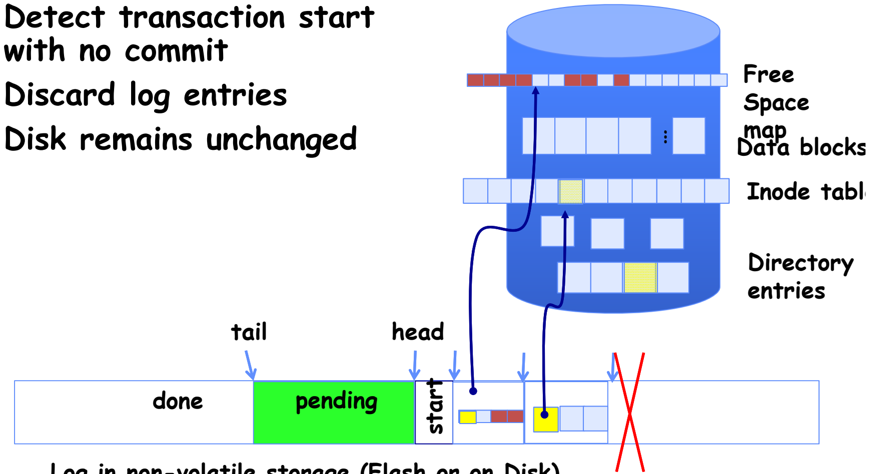
- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



Log in non-volatile storage (Flash)

Crash during logging - Recover

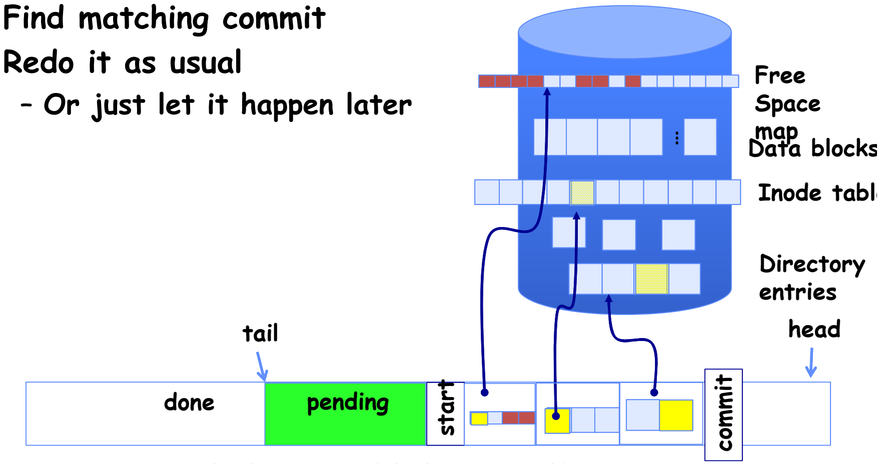
- Upon recovery scan the long
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



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?

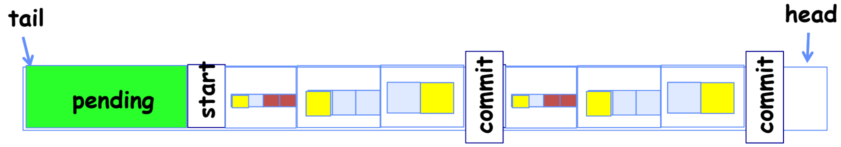
- Idempotent
- Just redo whatever part of the log hasn't been garbage collected

---

Redo Logging

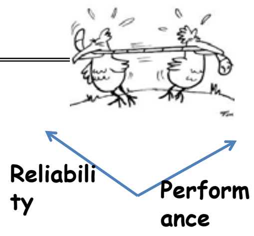
- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Ignore uncommitted ones
  - Garbage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

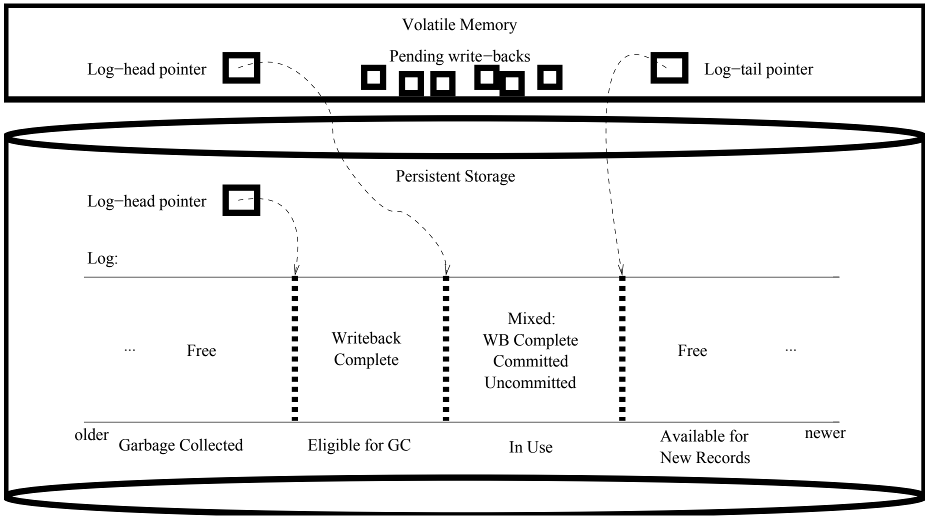


- Assume 5 ms average seek+rotation
- And 100 MB/s transfer
  - 4 KB block => .04 ms
- 100 random small create & write
  - 4 blocks each (free, inode, dirent + data)
- NO DISK HEAD OPTIMIZATION! = FIFO
  - Must do them in order
- 100 x 4 x 5 ms = 2 sec
- Log writes: 5 ms + 400 x 0.04 ms = 6.6 ms
- Get 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 ⇔ its commit record is in log

Redo Log Implementation



Isolation

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

- Assuming 162 appears only in foo,
- what are the possible outcomes of B without transactions?
- What if x, y and a,b are disjoint?
- What if x == a and y == b?
- Must prevent interleaving so as to provide clean semantics….

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

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	–	–

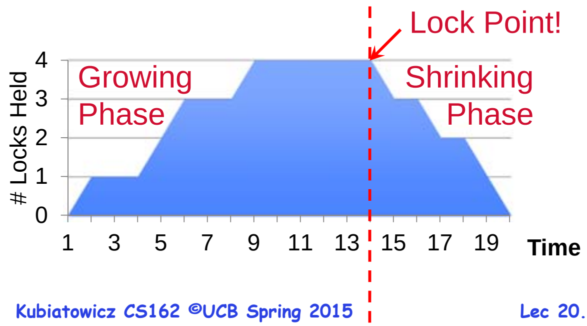
Two-Phase Locking (2PL)

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 it 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
  - Prevents a process from seeing results of another transaction that might not commit
  - Easier to recover from aborts

Transaction Isolation

Process A:

LOCK x, y

move foo from dir x to dir y

`mv x/foo y/`

Process B:

LOCK x, y and log

grep across x and y

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

Commit and Release x, y, log

Commit and Release x, y

- 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

Jim Gray

Computer Scientist

James Nicholas "Jim" Gray was an American computer scientist who received the Turing Award in 1998 "for seminal contributions to database and transaction processing research and technical leadership in system implementation." Wikipedia

Born: January 12, 1944, San Francisco, CA

Died: January 28, 2012

Books: Transaction Processing: Concepts and Techniques

Awards: Turing Award

Education: University of California, Berkeley (1969), University of California, Berkeley (1966)

Review: 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

Review: 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)

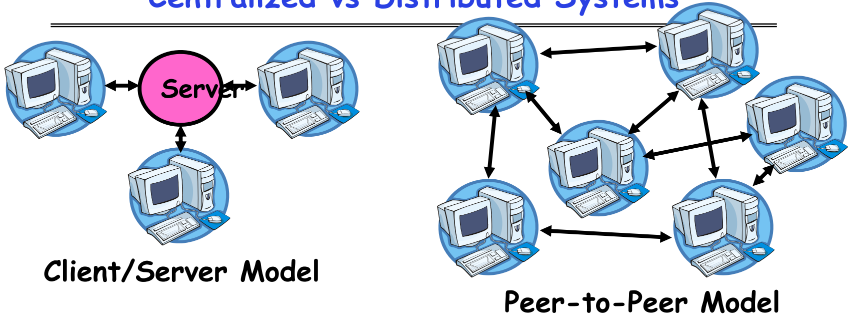
Review: Isolation

- 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

Review: 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!

Centralized vs Distributed Systems



- **Centralized System:** System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System:** physically separate computers working together on some task
  - Early model: multiple servers working together
    - » Probably in the same room or building
    - » Often called a "cluster"
  - Later models: peer-to-peer/wide-spread collaboration

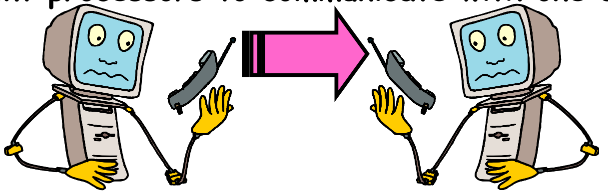
Distributed Systems: Motivation/Issues

- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: Much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
  - Higher availability: one machine goes down, use another
  - Better durability: store data in multiple locations
  - More security: each piece easier to make secure
- Reality has been disappointing
  - Worse availability: depend on every machine being up
    - » Lamport: “a distributed system is one where I can’t do work because some machine I’ve never heard of isn’t working!”
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

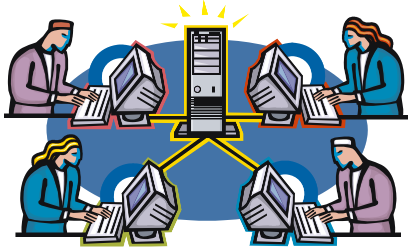
---

Distributed Systems: Goals/Requirements

- **Transparency:** the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - **Location:** Can’t tell where resources are located
  - **Migration:** Resources may move without the user knowing
  - **Replication:** Can’t tell how many copies of resource exist
  - **Concurrency:** Can’t tell how many users there are
  - **Parallelism:** System may speed up large jobs by spliting them into smaller pieces
  - **Fault Tolerance**: System may hide varoius things that go wrong in the system
- Transparency and collaboration require some way for different processors to communicate with one another



Networking Definitions



- **Network:** physical connection that allows two computers to communicate
- **Packet:** unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol:** agreement between two parties as to how information is to be transmitted

---

Summary

- Important system properties
  - **Availability**: how often is the resource available?
  - **Durability**: how well is data preserved against faults?
  - **Reliability**: how often is resource performing correctly?
- **RAID**: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Use of Log to improve Reliability
  - Journaled file systems such as ext3, NTFS
- **Transactions**: ACID semantics
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- **2-phase Locking**
  - First Phase: acquire all locks
  - Second Phase: release locks in opposite order