Recall: Important “ilities”

- **Availability:** the probability that the system can accept and process requests
  - Measured in “nines” of probability: e.g. 99.9% probability is “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
• Each disk is fully duplicated onto its “shadow”
  – For high I/O rate, high availability environments
  – Most expensive solution: 100% capacity overhead
• Bandwidth sacrificed on write:
  – Logical write = two physical writes
  – Highest bandwidth when disk heads and rotation synchronized (challenging)
• Reads may be optimized
  – Can have two independent reads to same data
• Recovery:
  – Disk failure ⇒ replace disk and copy data to new disk
  – Hot Spare: idle disk attached to system for immediate replacement
• 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 blocks in stripe
  – P0 = D0 ⊕ D1 ⊕ D2 ⊕ D3
  – Can destroy any one disk and still reconstruct data
  – Example: P0 = 0 ⊕ 1 ⊕ 0 ⊕ 1 = 0
    \[
    D0 \quad D1 \quad D2 \quad D3
    \]

• Suppose Disk 2 fails, then can reconstruct:
  – D2 = D0 ⊕ D1 ⊕ D3 ⊕ P0
  – Example: D2 = 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0

• Can spread information widely across internet for durability
  – RAID algorithms work over geographic scale
RAID 6 and other Erasure Codes

• In general: RAIDX is an “erasure code”
  – Must have ability to know which disks are bad
  – Treat missing disk as an “Erasure”
• Today, disks so big that: RAID 5 not sufficient!
  – Time to repair disk sooooo long, another disk might fail in process!
  – “RAID 6” – allow 2 disks in replication stripe to fail
  – Requires more complex erasure code, such as EVENODD code (see readings)
• More general option for general erasure code: Reed-Solomon codes
  – $m$ data fragments
  – generate $n - m$ extra fragments
  – can tolerate $n - m$ failures
• Erasure codes not just for disk arrays. For example, geographic replication
  – E.g., split data into $m = 4$ fragments, generate $n = 16$ fragments and distribute across Internet
  – Any 4 fragments can be used to recover the original data --- very durable!
Use of Erasure Coding for High Durability/overhead ratio!

- Exploit law of large numbers for durability!
- 6 month repair, FBLPY with 4x increase in total size of data:
  - Replication (4 copies): 0.03
  - Fragmentation (16 of 64 fragments needed): $10^{-35}$
Higher Durability through Geographic Replication

- Highly durable – hard to destroy all copies
- Highly available for reads
  - Simple replication: read any copy
  - Erasure coded: read m of n
- Low availability for writes
  - Can’t write if any one replica is not up
  - Or – need relaxed consistency model
- Reliability? – availability, security, durability, fault-tolerance
Recall: Important “ilities”

• **Availability:** the probability that the system can accept and process requests
  – Measured in “nines” of probability: e.g. 99.9% probability is “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
HOW TO MAKE FILE SYSTEMS MORE RELIABLE?
File System Reliability: (Difference from Block-level reliability)

• What can happen if disk loses power or 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
  – No protection against writing bad state
  – What if one disk of RAID group not written?

• File system needs durability (as a minimum!)
  – Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

• But durability is not quite enough…!
Storage Reliability Problem

• Single logical file operation can involve updates to multiple physical disk blocks
  – inode, indirect block, data block, bitmap, …
  – With sector remapping, single update to physical disk block can require multiple
    (even lower level) updates to sectors

• 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
  – Example: transfer funds from one bank account to another
  – 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
Two Reliability Approaches

Careful Ordering and Recovery
• FAT & FFS + (fsck)
• Each step builds structure,
• Data block \( \Leftarrow \) inode \( \Leftarrow \) free \( \Leftarrow \) directory
• Last step links it in to rest of FS
• Recover scans structure looking for incomplete actions

Versioning and Copy-on-Write
• ZFS, …
• Version files at some granularity
• Create new structure linking back to unchanged parts of old
• Last step is to declare that the new version is ready
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 by
  – FAT and FFS (fsck) to protect filesystem structure/metadata
  – Many app-level recovery schemes (e.g., Word, emacs autosaves)
Question

• Assume you need to store
  – A piece of data
  – A directory entry / pointer for the data
• Assume each of these operations is atomic

• Which one you should write first? Data or Pointer?
Berkeley FFS: Create a File

**Normal operation:**
- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

**Recovery:**
- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

*Time proportional to disk size*
Reliability Approach #2: Copy on Write File Layout

• Recall: multi-level index structure lets us find the data blocks of a file
• Instead of over-writing existing data blocks and updating the index structure:
  – Create a new version of the file with the updated data
  – Reuse blocks that don’t change much of what is already in place
  – This is called: Copy On Write (COW)

• Seems expensive!  But
  – Updates can be batched
  – Almost all disk writes can occur in parallel

• Approach taken in network file server appliances
  – NetApp’s Write Anywhere File Layout (WAFL)
  – ZFS (Sun/Oracle) and OpenZFS
COW with Smaller-Radix Blocks

- If file represented as a tree of blocks, just need to update the leading fringe
Example: ZFS and OpenZFS

• 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 Reliability Solutions**

- Use 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 filesystem structures and metadata
  - Many applications implement their own transactions

- Provide Redundancy for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)
Transactions

• Closely related to critical sections for manipulating shared data structures

• They extend concept of atomic update from memory to stable storage
  – Atomically update multiple persistent data structures

• 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

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.

- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from memory to persistent storage
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
Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions
Transactional File Systems

• Better reliability through use of log
  – 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 Systems

- Don't modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a journal or intention list
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied to file system
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
  - Some options: whether or not to write all data to journal or just metadata
- Other examples: NTFS, Apple HFS+, Linux XFS, JFS, ext4
Creating a File (No Journaling Yet)

- 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
Creating a File (With Journaling)

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

-----------------------------------------

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

Log: in non-volatile storage (Flash or on Disk)
After Commit, Eventually Replay Transaction

- All accesses to the file system first looks in the log
  - Actual on-disk data structure might be stale

- Eventually, copy changes to disk and discard transaction from the log

Log: in non-volatile storage (Flash or on Disk)
Crash Recovery: Discard Partial Transactions

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

Log: in non-volatile storage (Flash or on Disk)
Crash Recovery: Keep Complete Transactions

• 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)
Journaling Summary

Why go through all this trouble?

• Updates atomic, even if we crash:
  – Update either gets fully applied or discarded
  – All physical operations treated as a logical unit

Isn’t this expensive?

• Yes! We’re now writing all data twice (once to log, once to actual data blocks in target file)

• Modern filesystems journal metadata updates only
  – Record modifications to file system data structures
  – But apply updates to a file’s contents directly
Announcements

• PROJ3 design reviews this week (last round of design reviews!).
• HW5 due Monday, 11/22
Announcements

• PROJ 3 design reviews this week (last round of design reviews!).
• HW 5 due Monday, 11/22
• Midterm 2 grades were released. Please read over the solutions before requesting any regrades
  – Great job with the exam!
  – Mean: 59.76%
Recall: Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them
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
Centralized vs Distributed Systems

- **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/Promise

• 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
Distributed Systems: Reality

• Reality has been disappointing
  – **Worse availability**: depend on every machine being up
    » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
  – **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
  – What would be easy in a centralized system becomes a lot more difficult

• Trust/Security/Privacy/Denial of Service
  – Many new variants of problems arise as a result of distribution
  – Can you trust the other members of a distributed application enough to even perform a protocol correctly?
  – Corollary of Lamport’s quote: “A distributed system is one where you can’t do work because some computer you didn’t even know existed is successfully coordinating an attack on my system!”
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 splitting them into smaller pieces
  - **Fault Tolerance**: System may hide various things that go wrong

- **Transparency and collaboration** require some way for different processors to communicate with one another
How do entities communicate? A Protocol!

- A protocol is an agreement on how to communicate, including:
  - Syntax: how a communication is specified & structured
    - Format, order messages are sent and received
  - Semantics: what a communication means
    - Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram
  - Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
  - Stability in the face of failures!
Examples of Protocols in Human Interactions

• Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: “Hello?”
  6. Caller: “Hi, it’s John….”
     Or: “Hi, it’s me” (← what’s that about?)
  7. Caller: “Hey, do you think … blah blah blah …” pause
  8. Callee: “Yeah, blah blah blah …” pause
  9. Caller: Bye
  10. Callee: Bye
  11. Hang up
Distributed Applications

• How do you actually program a distributed application?
  – Need to synchronize multiple threads, running on different machines
    » No shared memory, so cannot use test&set

  – One Abstraction: send/receive messages
    » Already atomic: no receiver gets portion of a message and two receivers cannot get same message

• Interface:
  – Mailbox (mbox): temporary holding area for messages
    » Includes both destination location and queue
  – Send(message,mbox)
    » Send message to remote mailbox identified by mbox
  – Receive(buffer,mbox)
    » Wait until mbox has message, copy into buffer, and return
    » If threads sleeping on this mbox, wake up one of them
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

• Copy-on-write provides richer function (versions) with much simpler recovery
  – Little performance impact since sequential write to storage device is nearly free

• Use of Log to improve Reliability
  – Journaled file systems such as ext3, NTFS

• Transactions over a log provide a general solution
  – Commit sequence to durable log, then update the disk
  – Log takes precedence over disk
  – Replay committed transactions, discard partials

• **Protocol**: Agreement between two parties as to how information is to be transmitted