6.033 FINAL EXAM REVIEW SHEET

A case for RAID

RAID-1: Mirrored Disks
Mirrored disks are the most expensive option. Every write to a data disk is also a write to a check disk. Since these writes happen in parallel, it should be at least half as fast.
Main drawback: duplicating all disks means doubling the cost of the disks or using only halving storage capacity.

RAID-2: Hamming Code for ECC
If all data bits in a group are read or written together, there is no impact on performance. Reads of less than the group size require reading the whole group for error checking, and writes to a portion of the group require reading the whole group, modifying the data, and writing the full group, including check information.
1 parity disk can detect 1 error, but to correct an error we need enough check disks to identify the disk with the error. For 10 data disks, we need 4 check disks.
Good for supercomputers, bad for TPS.

RAID-3: Single Check Disk Per Group
Only 1 parity bit is needed to detect an error. Information on a failed disk can be reconstructed by calculating the parity of the remaining good disks and comparing bit by bit to the parity calculated for the original full group. If the check disk is the failure, read all data disks and store group parity in the replacement disk.
Drawback vs RAID-2: Soft errors can be corrected without rereading a sector. Also, the extra check information for each sector is not needed, increasing disk space.
Good for supercomputers but bad for TPS

RAID-4: Independent Reads/Writes
No longer spreads individual transfer information across several disks, keep each individual unit in a single disk. Allows parallelism for reads - the ability to do more than one IO per group at a time.
Parity calculation is much simpler: new parity = (old data xor new data) xor old parity. The check disk in a group must be read and written with every small write in that group, making it the bottleneck.

RAID-5: No single check disk
Distributes data and checks across all the disks - including the check disks. Because of this, RAID 5 can support multiple individual writes per group.
Best of both worlds: small read/writes perform close to the speed of a level 1 RAID and it keeps the large transfer performance and high storage capacity of RAID 3/4. RAID-5 is good for supercomputers or transaction processing.

The Google File System

GFS is a scalable distributed file system for large distributed data-intensive application. It has fault tolerance and delivers high aggregate performance to a large number of clients.

Assumptions
Component failures are common
Modest number of very large file
Reads are large streams or small & random
Large sequential writes that append data
Multiple clients concurrently append to the same file
High bandwidth is more important than low latency

New Operations
Snapshot creates a copy of a file or directory at low cost
Record append allows multiple clients to append to the same file and guarantees the atomicity of each append.

Architecture
A GFS cluster has one master and many chunkservers, and is accessed by many clients
Files are divided into 64MB chunks
Each chunk is replicated on multiple chunkservers
The master maintains all system metadata, chunk lease management, garbage collection, and chunk migration.
File data is not cached by the client or the chunkserver
Metadata is cached by the client
GFS does not have a per-directory data structure that lists all the files in that directory.
It does not support aliases for the same file (symbolic links)
GFS represents its namespace as a lookup table mapping full pathnames to metadata, stored in memory using prefix compression.

Master
Clients do not read and write files through the master, they ask the master which chunkservers it should contact, and interact with this chunkserver for subsequent operations
The master executes all namespace operations, manages chunk replicas, makes placement decisions, creates new chunks and replicas, coordinates system-wide activities to keep chunks fully replicated, to balance load, and to reclaim unused storage.

Metadata
Master stores 3 types of metadata: File/chunk namespaces, mapping from files to chunks, and locations of each chunk's replicas.
All metadata is kept in the master's memory, so master operations are fast.
Master does not keep a record of which chunkservers have a replica of a given chunk. It polls chunkservers for that information at startup. A chunkserver has the final word over what chunks it does or does not have on its own disks.
The operation log contains a historical record of critical metadata changes.
The operation log is the only persistent record of metadata and serves as a logical time line that defines the order of concurrent operations.
Files, chunks, and their versions are all uniquely identified by the logical times at which they were created.
The master recovers its state by replaying the operation log.
The master's state is checkpointed whenever the log grows too large, so it can recover by loading the latest checkpoint and replaying the log records after that.

Leases
The lease mechanism is designed to minimize management overhead at the master. Each mutation happens at all of the chunk's replicas.
Leases are used to maintain a consistent mutation order across replicas.
The global mutation order is defined first by the lease grant order chosen by the master, and then by a serial order that the machine granted the lease chooses.

Consistency model

GFS has a relaxed consistency model
File namespace mutations are atomic.
A file region is consistent if all clients will see the same data.
A file region is defined if it is consistent and all clients will see the last mutation in its entirety.
When a mutation succeeds without interference from other writers, the affected region is defined.
Concurrent successful mutations leave the region undefined but consistent.
A failed mutation makes the region inconsistent.
Mutations may be writes or record appends
A record append causes data to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing.
GFS may insert padding or record duplicates in between. These occupy inconsistent regions and are usually small.
After a sequence of successful mutations the file region is guaranteed to be defined and contain the data written by the last mutation
A chunk is lost irreversibly only if all of its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable rather than corrupted.

Atomic Record Appends

In a record append, the client specifies the data, and GFS appends it to the file at least once atomically at an offset of GFS's choosing and returns that offset to the client.
If a record append fails at any replica, the client retries the operation.
As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part.
GFS does not guarantee that all replicas are identical, just that the data is written at least once as an atomic unit.

Snapshot

The snapshot operation makes a copy of a file or directory tree almost instantaneously, using standard copy-on-write techniques.

Stale Replica detection

Chunk replicas become stale if a chunkserver fails and misses mutations to the chunk while it is down.
For each chunk, the master maintains a chunk version number to distinguish up-to-date and stale replicas
Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas.
The master removes stale replicas in its garbage collection.
Before that, it considers stale replicas to not exist when it replies to client requests for chunk information.

Fault Tolerance

We cannot trust the machines, or the disks
Two strategies keep the system highly available: fast recovery and replication
Fast recovery: Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they were terminated.
Chunk replication: Each chunk is replicated on multiple chunkservers on different racks
Master replication: The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines.
Each chunkserver independently verifies the integrity of its own copy by maintaining checksums

Concurrency Control and Recovery

Concurrency control ensures users see consistent states of the database even though their operations may be interleaved by the database
Recovery ensures that the database is not corrupted as the result of a software, system, or media failure.

Transaction

A transaction is a unit of work that must commit or abort as a single atomic unit.
If it commits, all updates it performed are made permanent and visible to other transactions.
If it aborts, all of its updates are removed from the database and the database is as if the transaction had never executed.

ACID Properties

Parity calculation is much simpler: new parity = (old data xor new data) xor old parity. The check disk in a group must be read and written with every small write in that group, making it the bottleneck.

RAID-5: No single check disk

Distributes data and checks across all the disks - including the check disks. Because of this, RAID 5 can support multiple individual writes per group.
Best of both worlds: small read/writes perform close to the speed of a level 1 RAID and it keeps the large transfer performance and high storage capacity of RAID 3/4.
RAID-5 is good for supercomputers or transaction processing.

The Google File System

GFS is a scalable distributed file system for large distributed data-intensive application. It has fault tolerance and delivers high aggregate performance to a large number of clients.

Assumptions

Component failures are common
Modest number of very large file
Reads are large streams or small & random
Large sequential writes that append data
Multiple clients concurrently append to the same file
High bandwidth is more important than low latency

New Operations

Snapshot creates a copy of a file or directory at low cost
Record append allows multiple clients to append to the same file and guarantees the atomicity of each append.

Architecture

A GFS cluster has one master and many chunkservers, and is accessed by many clients
Files are divided into 64MB chunks
Each chunk is replicated on multiple chunkservers
The master maintains all system metadata, chunk lease management, garbage collection, and chunk migration.
File data is not cached by the client or the chunkserver
Metadata is cached by the client
GFS does not have a per-directory data structure that lists all the files in that directory.

It does not support aliases for the same file (symbolic links)
GFS represents its namespace as a lookup table mapping full pathnames to metadata, stored in memory using prefix compression.

Master

Clients do not read and write files through the master, they ask the master which chunkservers it should contact, and interact with this chunkserver for subsequent operations
The master executes all namespace operations, manages chunk replicas, makes placement decisions, creates new chunks and replicas, coordinates system-wide activities to keep chunks fully replicated, to balance load, and to reclaim unused storage.

Metadata

Master stores 3 types of metadata: File/chunk namespaces, mapping from files to chunks, and locations of each chunk's replicas.
All metadata is kept in the master's memory, so master operations are fast.
Master does not keep a record of which chunkservers have a replica of a given chunk. It polls chunkservers for that information at startup. A chunkserver has the final word over what chunks it does or does not have on its own disks.
The operation log contains a historical record of critical metadata changes.
The operation log is the only persistent record of metadata and serves as a logical time line that defines the order of concurrent operations.
Files, chunks, and their versions are all uniquely identified by the logical times at which they were created.
The master recovers its state by replaying the operation log.
The master's state is checkpointed whenever the log grows too large, so it can recover by loading the latest checkpoint and replaying the log records after that.

Leases

The lease mechanism is designed to minimize management overhead at the master. Each mutation happens at all of the chunk's replicas.
Two phase locking forces all transactions to be well-formed and follow the rule: once a transaction has released a lock, it cannot obtain any additional locks.
Two phase locking is sufficient but not necessary to implement serializability.

Phantom problem

One transaction reads twice, another transaction adds a tuple that satisfies the query in between the reads.
The two read transactions will differ, which could not occur in a serial schedule

Isolation levels

Trades consistency for concurrency in a controlled manner.
Read uncommitted (Degree 0): No read locks obtained.
Risk is seeing updates that will be rolled back, or seeing incomplete transactions.
Read committed (Degree 1): Holds read locks on individual items for short duration.
Risk is seeing non-repeatable reads. Two reads could see two different values.
Repeatable read (Degree 2): Holds read locks on individual items for long duration.
Risk is the phantom problem.
Serializable (Degree 3): Holds read locks on predicates for long durations.
Solves the phantom problem.

Hierarchical locking

Concurrent transactions can obtain locks at different granularities
A lock at a particular level locks all items underneath it.
Lock escalation: If a transaction obtains many locks on one level, it could be given a lock at the level above.

ARIES

An implementation of recovery.
Uses WAL, with STEAL/NOFORCE buffer management policy:
Pages on stable storage can be overwritten and data pages do not need to be forced to disk to commit a transaction.
Its REDO repeats history: It redoes updates for transactions that will eventually be undone. This allows it to employ physiological logging.
It uses page oriented REDO and logical UNDO.
First pass processes the log forward from the most recent checkpoint, determines dirty pages and active transactions.
Second pass repeats history by processing the log forward.
Third pass proceeds backwards from the end, removing the effects of all uncommited transactions.
Uses fuzzy checkpoints that are extremely inexpensive.

Log Structured File System

Writes all modifications to disk in a log-like structure, speeding up file writing and error recovery.
Log is divided into segments, and a segment cleaner to compress live information from heavily fragmented sections.
An implementation, Sprite LFS, outperforms current unix file systems by an order of magnitude for small writes.
This system eliminates all seeks while writing.

Problems with existing file systems

Current file systems spread information around the disk in a way that causes many small accesses
Furthermore, the inode for a file is separate from its contents, as is the directory entry containing its name.
When writing small files, less than 5% of the disk's bandwidth is used for new data, the rest of the time is seeking.
The second problem is that they tend to write synchronously. The application must wait for the write to complete rather than continuing while the write happens in the background.

Log structured file systems

Fundamental idea: improve write performance
buffering a sequence of file system changes in the file cache and then writing all changes to disk sequentially in a single write operation.
For workloads that contain many small files, a log struc-

tured file system converts the many small synchronous random writes of a traditional file system into large asynchronous sequential transfers that can use 100% of a disk's bandwidth. Two key issues: How to retrieve information from the log, how to manage free space on disk so that large extents of free space are always available for writing new data.

File location and reading

Outputs index structures in the log to permit random access retrievals. For each file there is an inode. Unlike unix FFS, inodes are not in a fixed position in the disk, instead, there is an inode map, which maintains the current location of each inode. The inode map is small enough to be cached, so it rarely requires disk accesses.

Segments

The goal is to maintain large free extents for writing new data.
When the log reaches the end of the disk, two things can be done: Threading (writing the log to unused portions of the disk) or copying (moving live data out of the log and writing a compressed version of it at the head of the log.
Sprite LFS uses a combination of both. The disk is divided into large areas called segments. Each segment is written sequentially, and all live data must be copied out of the segment before it can be rewritten, but the log is threaded on a segment basis.

Segment cleaning

Copying live data out of a segment is called segment cleaning. Read a number of segments into memory, identify the live data, and write the live data back ot a smaller number of clean segments. After this operation is complete, the segments read are now clean.
A segment summary block identifies each piece of information written in the segment, so that the inodes can be updated when the segment is cleaned.
Sprite LFS cleans segments when the number of clean segments drops below a certain threshold, and it cleans until the number of clean segments rises above another threshold value.
Segments are divided into two types: hot and cold. Hot segments were written to very recently, and their free space is less valuable (since they will be written to again), and cold segments have old data, and their free space is more valuable.
To calculate hot and cold segments, there is a segment usage table that contains the number of live bytes in the segment and the most recent modified time of any block in the segment.
Hot segments are cleaned more regularly than cold segments.

Crash recovery

When a system crash occurs, the last few operations may have left it in an inconsistent state.
In traditional unix file systems, the system must scan all of the metadata structures on disk to restore consistency. The cost of these scans is very high.
In LFS, the locations of the last operations are easy to determine: they're at the end of the log.
Sprite LFS uses two methods to recover: checkpoints, which define consistent states of the file system, and roll-forward, which is used to recover information written since the last checkpoint.

Checkpoints

To write a checkpoint, Sprite LFS first writes out all modified information to the log, including data, inodes, inode map, and segment usage table. Second, it writes a checkpoint region to a special fixed position ont he disk.
The checkpoint region contains the addresses of all the blocks in the inode map and segment usage table, plus the current time and last segment written.

Roll-forward

Sprite LFS reads through the log entries that were written after the last checkpoint. A directory operation log entry is created for each directory operation and this is used to keep consistency between the directory entries and inodes.

PNUTS

PNUTS is a parallel and distributed database system used by Yahoo's web applications.
It provides data storage as hashed or ordered tables, with low latency for many concurrent requests, and per-record consistency guarantees.

Relaxed Consistency

No need for serializable transactions, or even transactions at all. Most users only modify one record at a time. One possible idea was eventual consistency, but that is too weak: if a user wants to remove his mother from the list of people who can access his photos before adding a photo, these updates must happen in that order.

Data and Query Model

Data is organized into tables of records with attributes.
Flexible schemas: new attributes can be added, and records are not required to have values for all attributes.
There are two main types of access: Point access involves a single record, range access involves a set of records. PNUTS allows hashed or ordered tables to support both types of requests.

Consistency Model

Between the extremes of serializability and eventual consistency. Per record timeline consistency: all replicas of a record apply all updates to the record in the same order. To implement, a master replica receives all updates to that record. There are several API calls with varying levels of consistency guarantees:
Read-any: returns a possibly stale version of the record
Read-critical(required-version): returns a version of the record that is at most as old as required-version
Read-latest: Returns the latest copy of the record
Write: Performs a normal write
Test-and-set-write(required-version): Performs a write to a record if and only if the version is the same as required-version. This is used for transactions that read a record and then modify it.

Architecture

Tables are horizontally split into groups of records called tablets.
Storage unit: Stores tablets, responds to get(), scan() and set()
Router: determines which tablet has a record, and which storage unit has that tablet
Tablet controller: Polled by the routers to obtain updated mappings. Determines when to move a tablet between storage units
Message Broker: A publish/subscribe system, replaces

redo log, used for replication. Provides partial ordering of published messages.
Scatter-Gather engine: Component of the router, responsible for multi-record requests.

Spanner
Google's database system. Distributes data at global scale, supports externally consistent distributed transactions.
It shards its data across sets of Paxos state machines. Applications can fine-tune parameters, and it provides externally consistent reads and writes, and globally consistent reads across the database at a timestamp.
Timestamps are used to reflect serialization order. This system is linearizable.

Implementation
A spanner deployment is called a universe. It has a placement driver which moves data across zones. It is organized as a set of zones, where each zone is analogous to a bigtable deployment.
A zone has one zonemaster and many spanservers. The zonemaster assigns data to spanservers, the spanservers serve data to clients.
A zone also has location proxies, used to find which spanserver is assigned to serve a user's data.

Spanserver
Each spanserver is responsible for many tablets.
A tablet's state is stored in a set of B-tree like files and a write-ahead log.
Each spanserver implements a Paxos state machene on top of each tablet. Each state machine stores its metadata and log in the corresponding tablet.
These Paxos machines are used to implement a consistently replicated bag of mappings. The set of replicas is called a Paxos group.
At every replica that is a leader, each spanserver implements a lock table for concurrency control, using two phase locking. Each leader also implements a transaction manager to support distributed transactions.

TrueTime
TT.now() - returns an interval from earliest to latest possible timestamp
TT.after(t) - returns true if timestamp t has definitely passed
TT.before(t) - returns true if timestamp t has definitely not arrived
Truetime uses both GPS and atomic clocks
A set of timemaster machines per datacenter
a timeslave daemon per machine.
The time uncertainty is under 7ms.

Cross Site Request Forgery
Cross Site Request Forgery is a widely exploited web vulnerability. A login CSRF logs the victim into an honest website as the attacker. The severity of CSRF varies, but can be as severe as XSS.

Common Defenses (ineffective)
Including a secret token with each request, validating that token with the user's session - bad because they overlook login requests because login requests lack a session to bind their token to.
Validating HTTP referer header, but if a request lack a referer header, the site must block the request (exclusing a significant proportion of legitimate users) or process the requests (making the defense ineffective).
XMLHttpRequest headers - require all state modifying requests to be made using XMLHttprequests.

Origin header
Proposal: Create an origin header, similar to HTTP referer, but addressing the privacy concerns.
Includes only information required to identify who initiated the request.
Sent only on POST requests.
Proposal: Make all state changing requests be POST requests, if Origin header is present, reject any requests whose origin header contains an undesired value.

What is CSRF?
Allows a malicious site to send requests as a user. Allows a userto access sites behind a firewall, read browser state, or write browser state.

Types of attacks
Forum poster: A malicious individual can create false images or links that force HTTP get requests, which are not supposed to modify the state based on a user's request, but frequently do.
Web Attacker: Malicious individual owns domain name, has a valid HTTPS certificate for that server, and operates that server. He can instruct the user's browser to mount a CSRF attack by instructing the browser to issue cross-site requests using GET or POST methods.
Network Attacker: Controls the user's network connection.

Out of scope attacks
These attacks are not considered CSRF attacks: XSS, Malware, DNS rebinding, Certificate errors, phishing, user tracking

Login CSRF
Logs the user into a malicious individual's account. The attacker forges a login request to an honest site using the attacker's username and password.
Possible attacks: Search history: Allows a malicious individual to store a user's search history.
Paypal: Lets the user log into a malicious individual's account and add his credit card to that account
iGoogle: Attaches to the user's iGoogle gadgets, allows the user to run the malicious individual's scripts, which create a fake login page, steals the user's autocompleted password, or reads the user's cookies to obtain the user's credentials.

Trusting Trust
You can make a C compiler with malicious code that inserts the malicious code into the compiled version of a C compiler (using a quine). With this, if you compile a program, it can insert bugs and no amount of reading the source will help. You must trust the people who wrote the software.

Beyond Stack Smashing
Traditional approach: Stack smashing - modifying the return address saved on the stack to point to malicious code.
Arc Injection (return to libc) transfers control to code that already exists in memory space. Used when stack is not executable.
Pointer Subterfuge changes control flow by attacking function pointers or modify arbitrary memory locations by subvering data pointers.

Heap Smashing allows exploiting buffer overflows in dynamically allocated memory, rather than on the stack.

Stack Smashing Enhancements
Trampolining: Allows you to apply stack smashing without knowing exactly where your malicious code is. Find a sequence of instructions at a predictable location that transfers control in a reliable way, then go there instead.
Separating the payload and the buffer overrun operation: Useful if your buffer is too small to hold your payload.

Effectiveness of Phishing Warnings
Phishing relies on confusing people, so it is hard to automatically detect with complete accuracy, because of this, anti-phishing tools use warnings rather than blocking sites.
Two types: Active and Passive, where active forces user to click past the warning.
Look and feel is the most important aspect to a user's trust, which means a well-designed malicious website can be really effective. Savvy users are just as susceptible to phishing as novice users.

SiteKey
Shows a userselected image after the user enters his username. 92% of users still logged in when the correct image was not present.

Extended Validation
Certificates saying the company is legally recognized. IE colors the URL bar green. Does not work, and makes users less suspicious of fraudulent websites that don't yield warnings.

Metrics
Hazard Matching - accurately using warning messages to convey risks
Arousal Strength - perceived urgency of the warning
Habituation - When a user sees the same warning many times, he ignores it. Only way to recapture attention is increasing arousal strength of warning.

Conclusion
Most people clicked the urls, and most people believed they came from eBay and paypal. 79% of people were saved by active warnings, whereas 13% of people were saved by passive warnings. Passive warnings are useless. Active warnings from IE are often ignored. Phishing warnings need to interrupt the user, provide clear choices, fail safely, and prevent habituation.

Transport Layer Security
TLS is a widely used protocol to establish a secure channel. It is a version of SSL.

TLS Handshake
Client sends a ClientHello message with the version that the client is running, a random sequence number, and a prioritized set of ciphers the client is willing to use.
Server responds with a ServerHello message announcing the version of the protocol that will be used, a random number, a session identifier, and a cipher selected from the set.
To authenticate the server, it sends a ServerCertificate message with a chain of certificates.
Server then sends a ServerHelloDone message to indicate it is done with the first part of the handshake.
Client verifies the certificates, generates a pre-master-secret, which it encrypts with the public key of the server.
The pre-master-secret is used by the server, along with both random numbers to generate the master-secret.
Client sends a ChangeCipherSpec message, specifying the cipher that it will use.
Client sends a Finished message, verifying the protocol sequence so far.
Server sends a ChangeCipherSpec message, specifying the cipher that it will use.
Server sends a Finished message, at which point there is a secure channel (encrypted and authenticated) over which they can communicate.

Authenticating Services
Client must convince itself that a service's public key is authentic.
To do this, the service buys a certificate from one or more CAs, each authority runs a check to validate the service, and then provides a certificate.
Client has a list of public keys of CAs it trusts, and uses them to validate one of the certificates.

Authenticating Users
Users can be authenticated by certificates, but this is uncommon due to inconvenience.
Using IP address is bad because it can be spoofed.
Passphrase authentication is better, but to avoid having the user type in passphrase on every request, the server uses cookies.

L14: Reliability
Goals: Keep running despite failures, include malicious failures.
Threats: Software faults, hardware failures, design issues, operation failures, environmental issues
Build reliable systems from unreliable parts. Basic way: redundancy.
Mean Time To Failure = MTTF
Mean Time To Repair = MTTR
Availability = $MTTF/(MTTF + MTTR)$
MTBF = $MTTF + MTTR$

L15: Atomicity
Atomicity is making a multi-step action look like a single action. For example, transferring money at a bank.
Related problem: Concurrency. If we have another function that sums the money in the bank, if it runs during a transfer it could return the wrong total.
Never modify the only copy if you want atomicity. Use a shadow copy, and rename it. Commit point is modifying the directory entry. If we crash, we might have too many refcounts on an inode, in which case we can remove the directory entry for the newfile.

L16: Logging
Key idea: keep a log of updates a transaction made while it ran and whether it committed or aborted.
Assign each all-or-nothing action a unique transaction id.
Two kinds of records: update and commit/abort records.

Writing a commit record is the commit point for an action, but writing a log record had better be atomic.
Two approaches: Make each record fit within one sector, or put checksum on each record.
To optimize for speed, keep both a log and cell storage.

Cell storage provides fast reads but is not atomic. Logging an update puts it into the log, installing an update puts it to cell storage.
To recover cell storage, scan log, determine loser actions and undo them. Scan backwards to undo newest to oldest.
Golden rule: Log update before installing it (WAL)
Further optimizations: Cache writes, truncate log by writing checkpoints.

L17: Isolation

Isolate running transactions from each other.
For each read object in T1, conflicts with all writes of that object in T2. For each write object in T1, conflicts with all reads or writes of that object in T2. A schedule is serializable if all conflicts run T1's action first or all conflict run T2's action first.
Plan: obtain locks for each variable before accessing it.
Wait until commit to release lock.
Problem: Deadlocks, to solve, kill one transaction.
Optimization 1: Read/Write locks separately, multiple readers are OK, conflicts are only when there is a writer.
Optimization 2: two-phase locking. phase 1: acquire read and write locks until transaction reaches lock point, phase 2: release locks after lock point and done with object (Idea: releasing locks early).
Strict 2PL: Hold write locks until end of transactions
Optimization 3: Relaxed consistency: Don't always need serializability.

L18: Multisite atomicity

Don't want one site to commit unless the other has also committed. One might commit and the other might crash and decide to abort.
Solution: 2 phase commit. Get nodes to agree that they are ready to commit, even if they crash, and then make them commit.
Coordinator sends tasks to workers, workers log transaction begin and updates as normal
Once all tasks are done, coordinator needs to get workers to enter prepared state.
Workers will definitely commit if in prepared state and coordinator tells them to commit.
If messages are lost, use timeouts to resend messages.
2PC provides a way for a set of distributed nodes to agree (commit vs abort), but it only guarantees they will eventually learn about the outcome, not agree at the same instant.
2 generals paradox: Can never ensure that agreements happen in bounded time, but they will eventually happen with high probability.

L19: Replicated state machines

Problem: Failures of nodes can make system unavailable.
Might want a long running service to continue working in the presence of a node failure.
Ideal goal for replicated system: single-copy consistency.
Operations appear to execute as if there's only a single copy of the data. Internally, there may be disagreements or failures, which have to be masked.
DNS and PNUTS settle for weaker semantics
Different clients can observe different states of the system.
Problem: replicas can become inconsistent if clients' requests arrive in different orders to different servers.
Solution: Replicated state machines
Start with the same initial state on each server, provide each replica with same input operations in same order, ensure all operations are deterministic.
Assumption: Independent failure of replicas
Client - Primary DB - Backup DB
Primary ensures all updates sent to backup before acknowledging client.
Primary chooses an ordering for all operations and decides non-deterministic values for random(), time(), etc.
Problem: What if primary fails?
Client could know about primary and backup and decide which to use.
Problem: multiple clients think different replicas are primary.
Solution: Have a human decide when to switch from primary to backup
Automated Solution: Have a view server maintain current primary
Clients contact view server to agree on primary.
Primary not allowed to respond until it gets Ack from backup.
Non-backup must reject forwarded requests.
Primary in view i must have been primary or backup in view i-1
Non-primary must reject direct client requests

L20: RSM with PAXOS

Need a way to pick new view server if old one fails. But what if old one is still alive?
View server is actually n nodes, where n¿=3. They run PAXOS (distributed consensus) to decide on some value (id of next view server).
Majority rule: to choose a new value, majority must agree on the value. Any 2 majority sets overlap by one server, so any majority will contain a node that heard about the previous majority.
Use PAXOS to choose the view server. View server nodes use PAXOS to agree on the set of all replicas and coordinator, the last operation executed by the current (dead) coordinator.

L21: security intro

Attacks are cheap, fast, and scalable.
Security is hard because it is a negative goal. Must consider all possible ways in which security might be broken.
Cannot guarantee security.

Policy: goals

Information security goals:
Privacy: limit who can read data
Integrity: limit who can write data
Liveness goals:
Availability: ensure service keeps operating

Threat model: assumptions

Adversary controls some computers, networks (but not all)
Adversary controls some software on computers he doesn't fully control
Adversary knows some information, passwords, or keys (but not all)
Adversary knows about bugs in your software?
Physical attacks?

Social engineering?
Resources are hard to estimate.
Many systems compromised due to incomplete threat model
Overly ambitious threat models are not good either:
Not all threats are equally important
Stronger requirements lead to complexity, which leads to subtle security problems

Guard model
We are worried about security at the server
Security goal relates to some resource in the server
Server checks all accesses to resource (consults a guard to make decision)
Complete mediation: only way to access involves guard
Must enforce client-server modularity
Must ensure server invokes guard in right places
Authentication: request - principal
E.G: username verified by password
Authorization: Request,principal,resource - allow?
E.G: consult access control list for resource

What goes wrong?
Software bug / Complete mediation
Policy vs Mechanism (policy is that students cannot get a copy of grades.txt, mechanisms are permissions or firewall rules)
Interactions between layers/components
Users make mistakes (social engineering, phishing)
Cost of security (users may be unwilling to pay cost of security)