By Myee Ye, as cheatsheet in OSDS final test. 2 cheatsheets allowed. You may use NotebookLM.

1 Introduction and Concurrency

1.1 Distribution System

A collection of independent computers that appears to its users as a single coherent system. “shared nothing architecture”
Features: 1.No shared memory - message-based communcation 2.Each runs its own local OS 3.Heterogeneity
Goals: 1. Resource Sharing 2. Transparency 3. Openness 4. Scalability

1.2 Concurrency

Concurrency is not parallelism, although it enables parallelism.
Concurrency is to realize **Mutual Exclusion**.
Mutual Exclusion: guarantee that only a single thread/process enters a CS, avoiding races
Properties of Mutual Exclusion:
1. Mutual Exclusion (Correctness): single process in CS at one time
2. Bounded Waiting (Fairness): No process waits for ever for a resource, i.e. a notion of fairness
3. Progress (Efficiency): Processes don’t wait for available resources, or no spin-locks => no wasted resources

Semaphores:
Semaphore has a non-negative integer value and supports the following two operations:
P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

1.3 Concurrency in GO

Goroutines: launched by `go`, independent call stack, very inexpensive
Channels: Passing information, synchronzation goroutines, providing pointer to return location
Channel Usage:
1. Bounded FIFO queue `c := make(chan int, 17)`, `d := make(chan string, 0)`
2. Insertion (If channel full, wait for receiver). Then put value at the end. `c <- 21`
3. Removal (If channel empty, then wait for sender. Then get first value) `s := <- d`
4. Capacity = 0: used for Sync Send / Recv
5. Capacity = 1: Pass token from sender to receiver

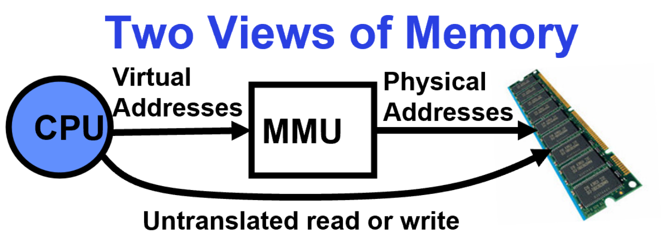
2 Virtual Memory

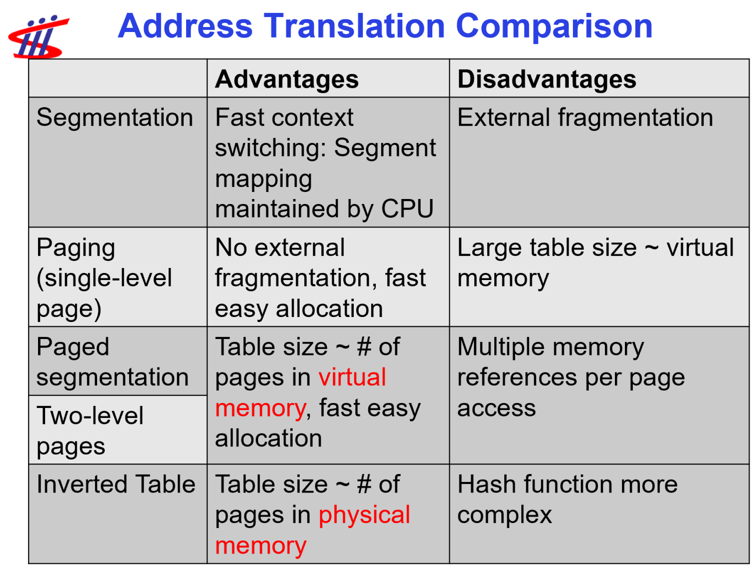
2.1 Segmentation

Segmentation: dynamic address translation
Realization: Address=Seg# + Offset, Seg# is BaseAddress+LimitLength+ValidBit
Advantage: giving program the illusion that it is running on its own dedicated machine, with memory starting at 0
Fragmentation: wasted space. 1. External: free gaps between allocated chunks 2. Internal: don’t need all memory within allocated chunks

2.2 Paging

Page: fix length of segment
Realization: Address=VirtualPage# + Offset, VirtualPage# is gotten through page table. Each page has valid bit, r/w able bit, dirty bit and so on.
Two-level page table: Tree of Page Tables.
Inverted Page Table: Hash table from VirtualPage# to PhysicPage#. Cons: Complexity of managing hash changes




Address Translation Comparison

	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Paging (single-level page)	No external fragmentation, fast easy allocation	Large table size ~ virtual memory
Paged segmentation	Table size ~ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Two-level pages		
Inverted Table	Table size ~ # of pages in physical memory	Hash function more complex

2.3 Caching general ideas & Demand paging

Temporal Locality: Locality in Time. Keep recently accessed data items closer to processor
Spatial Locality: Locality in Space. Move contiguous blocks to the upper levels
Demand Paging is Caching.
Factors Lead to Misses:
1. Compulsory Misses: Pages that have never been paged into memory before. Solution: Prefetching, Clustering, Working Set Tracking
2. Capacity Misses: Not enough memory. Must somehow increase size. Solution: Increase amount of DRAM / when multiple processes, adjust percentage of memory allocated
3. Conflict Misses: Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache.
4. Policy Misses: Caused when pages were in memory, but kicked out prematurely because of the replacement policy. Solution: Better replacement policy
FIFO (First In, First Out): Throw out oldest page. Fair. Bad, because throws out heavily used pages instead of infrequently used pages
MIN (Minimum): Replace page that won't be used for the longest time in the future. Great, but can't really know future
RANDOM: Pick random page for every replacement. Typical solution for TLB's. Simple hardware. Unpredictable
LRU (Least Recently Used): Replace page that hasn't been used for the longest time. Programs have locality, so if something not used for a while, unlikely to be used in the near future. Seems like LRU should be a good approximation to MIN.
In practice, people approximate LRU.
Bélády's anomaly: When use FIFO, pages faults rate may increase while number of frames increases.
Second Chance Algorithm: Approximating LRU. Replace an old page, not the oldest page. FIFO with `use` bit. Remove the front if `use=0`, otherwise change `use` to `0` and move to the back.
Clock Algorithm: efficient implementation of SCA. Arrange physical pages in circle.
Nth chance algorithm: give increment on counter if `use=0`. Normally $N = 1$ for clean pages and $N = 2$ for dirty pages.

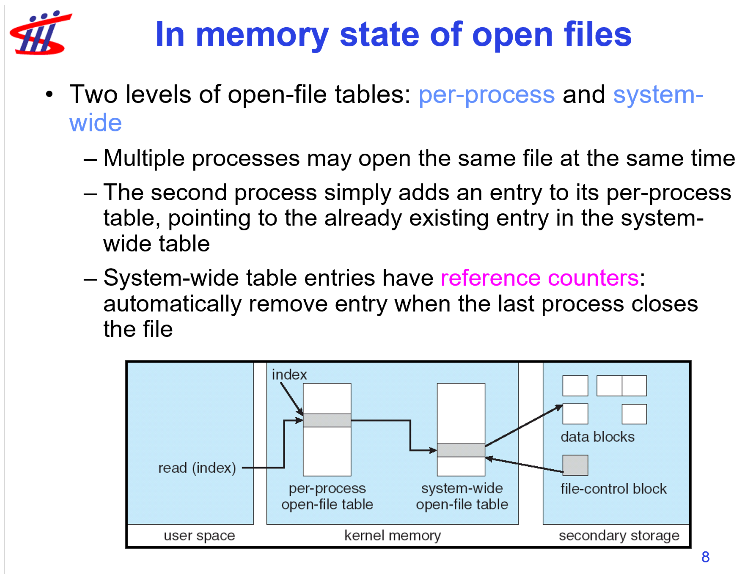
2.4 Buffer cache: File system caching

Key Idea: Exploit locality by caching data in memory
Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
Implemented entirely in OS software. Blocks go through transitional states between free and in-use
Caching Policy: LRU. Advantages: Works very well for name translation; Works well in general as long as memory is big enough to accommodate a host's working set of files. Disadvantages: Fails when some application scans through file system, thereby flushing the cache with data used only once. Example: `find . -exec grep foo {}`
Cache Size: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
Read Ahead Prefetching: fetch sequential blocks early
Delayed Writes: Writes to files not immediately sent out to disk. file systems need recovery mechanisms.

3 File Systems

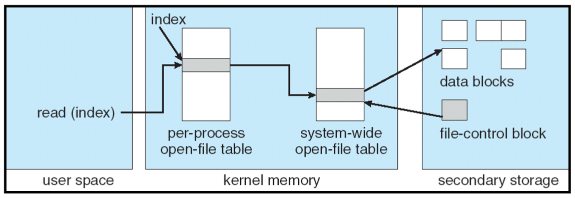
3.1 File Systems

Three views: 1. User's view: files 2. Syscall interface (between user and OS): collection of contiguous bytes 3. OS internal view: collection of blocks (may be incontiguous)
Disk management: organize disk blocks into files. Files: Naming, Protection, Reliability/durability
Metadata: attributes of a file a.k.a file control block a.k.a (sometimes) inode; directory.
All storage logic: store metadata the same way as data
Two levels of open-file tables: per-process and system-wide
Directory Structures: Data of a directory is an unordered list of <name, pointer> pairs. A special directory called the root (/), and has i-number of 2
link / unlink (rm): Link existing file to a directory.
Hard link: Sets another directory entry to contain the file number for the file. Creates another name (path) for the file. Both first class. Maintain ref-count of links to the file. Delete after the last reference is gone.
Soft link / Symbolic Link / Shortcut: Directory entry contains the path and name of the file. Map one name to another name.

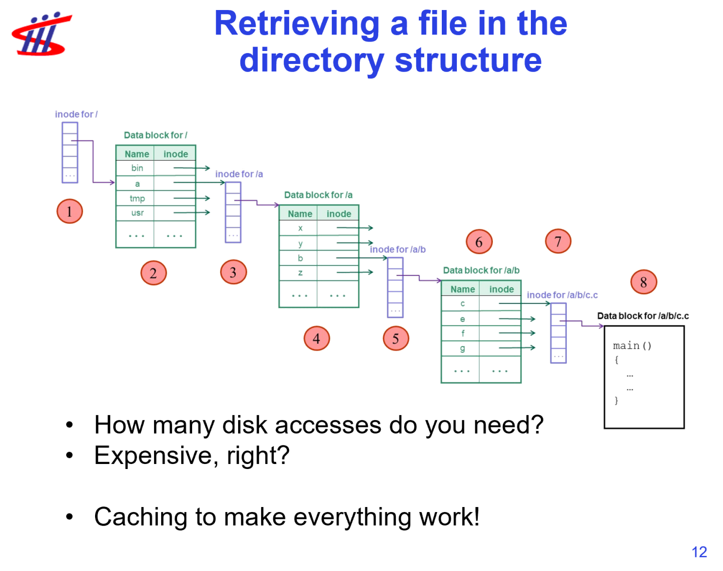


In memory state of open files

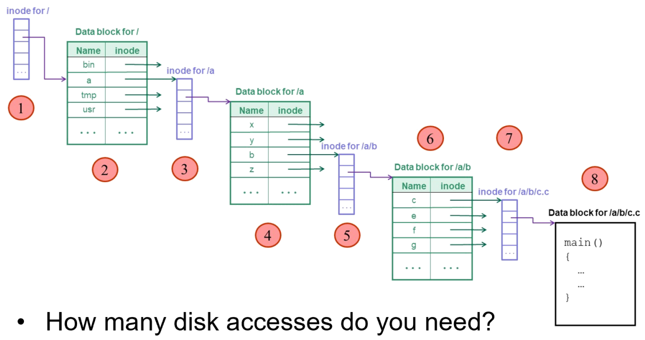
- Two levels of open-file tables: per-process and system-wide
  - Multiple processes may open the same file at the same time
  - The second process simply adds an entry to its per-process table, pointing to the already existing entry in the system-wide table
  - System-wide table entries have reference counters: automatically remove entry when the last process closes the file



8

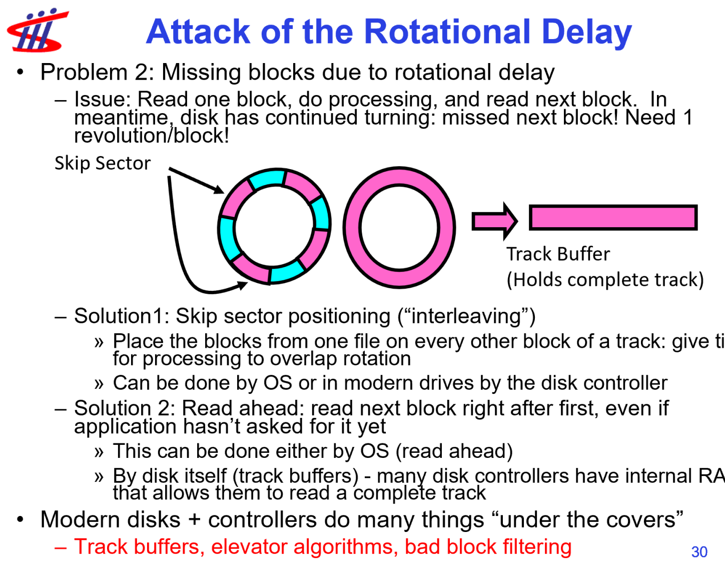


Retrieving a file in the directory structure



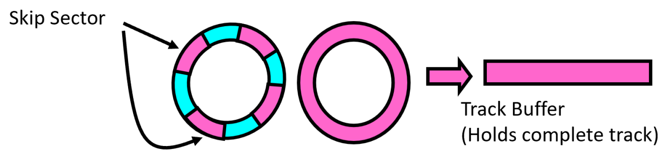
- How many disk accesses do you need?
- Expensive, right?
- Caching to make everything work!

12



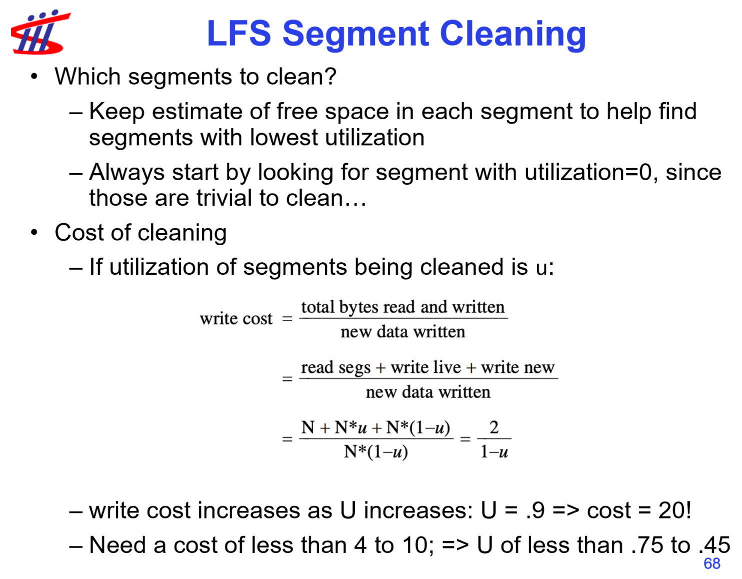
Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
    - » Can be done by OS or in modern drives by the disk controller
  - Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it yet
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track
- Modern disks + controllers do many things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

30



LFS Segment Cleaning

- Which segments to clean?
  - Keep estimate of free space in each segment to help find segments with lowest utilization
  - Always start by looking for segment with utilization=0, since those are trivial to clean…
- Cost of cleaning
  - If utilization of segments being cleaned is u:

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{\text{N} + \text{N}^*u + \text{N}^*(1-u)}{\text{N}^*(1-u)} = \frac{2}{1-u}$$

  - write cost increases as U increases: U = .9 => cost = 20!
  - Need a cost of less than 4 to 10; => U of less than .75 to .45

68

3.2 Performance optimizations

Sequential Access: bytes read in order
Random Access: read/write element out of middle of array
Characteristics of Files: Most files are small, growing numbers of files over time; Large files use up most of the disk space and bandwidth to/from disk

File Allocation Table (FAT): The most commonly used file system in the world!
File is collection of disk blocks. FAT is linked list 1-1 with blocks
File Number is index of root of block list for the file. Grow file by allocating free blocks and linking them in
FAT is simple, and has many Security Holes (no access rights, no header(metadata) in the file blocks). can't just list access control in the directory. reason: metadata is a single shared data structure
Fast File System (FFS, BSD 4.2): Uses bitmap allocation in place of freelist. Attempt to allocate files contiguously. 10% reserved disk space. Skip-sector positioning.
Use bitmap, store files from same directory near each other. The header information(inodes) stored closer to the data blocks.
FFS's order: no seeks
keep 10% or more free: Reserve space in the Block Group.
Improving Large Directory performance using B+Trees (dirhash)

3.3 General storage reliability

Availability: the probability that the system can accept and process requests
Durability: the ability of a system to recover data despite faults
Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period
Threats to Reliability: Interrupted Operation; Loss of stored data
How to be Durable:
1. Disk blocks contain error correcting codes (ECC) to deal with small defects in disk drive (recovery of data from small media defects)
2. Make sure writes survive in short term (Use special, battery-backed RAM called non-volatile RAM or NVRAM for dirty blocks in buffer cache)
3. Make sure that data survives in long term (replicate; independence of failure)
RAID (Redundant Arrays of Inexpensive Disks): Data stored on multiple disks (redundancy)
Redundancy necessary because cheap disks were more error prone
RAID 1: Disk Mirroring/Shadowing
Disk failure: replace disk and copy data to new disk
Hot Spare: idle disk already attached to system to be used for immediate replacement
RAID 5+: High I/O Rate Parity
Can spread information widely across the Internet for durability. RAID algorithms work over geographic scale
RAID 6: disks so big that RAID 5 not sufficient. RAID 6 allows 2 disks in replication stripe to fail.
Reed-Solomon codes: m data points define a degree m polynomial; encoding is $n > m$ points on the polynomial.
Any m points can be used to recover the polynomial; $n - m$ failures tolerated

3.4 File system reliability

File System Reliability: File system needs durability.
Operations on persistent storage are not atomic.
Reliability Approach: 1. Careful Ordering. Sequence operations in a specific order. Need Post-crash recovery. e.g. FAT, FFS. 2. Copy on Write (COW) File Layout. To update file system, write a new version of the file system containing the update. expensive, but updates are batched and writes can be in parallel. e.g. WAFL, ZFS, OpenZFS.
Transactions: A general way to solve the reliability issue.
Begin a transaction by get transaction id. Do a bunch of updates. roll-back if fail or conflict. Commit the transaction.
Useful for atomic updates. Closely related to critical sections for manipulating shared data structures.
Extend concept of atomic update from memory to stable storage.
ACID properties: 1. Atomicity: all actions in the transaction happen, or none happen
2. Consistency: transactions maintain data integrity
3. Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency
4. Durability: if a transaction commits, its effects persist despite crashes
Transactional File Systems:
1. Better reliability through use of log. All changes are treated as transactions, a transaction is committed once it is written to the log.
2. "Journaling File System". Applies updates to system metadata using transactions. Updates to non-directory files (i.e., user

stuff) can be done in place (without logs), full logging optional.
3. Full Logging File System. All updates to disk are done in transactions.
Log Structured File System (LFS):
Log all data and metadata with efficient, large, sequential writes.
Treat the log as the truth, but keep an index on its contents.
Rely on a large memory to provide fast access through caching.
Data layout on disk has "temporal locality" (good for writing), rather than "logical locality" (good for reading).
LFS Log Retrieval: A question of finding a file's inode. Solution: an inode map. inode map gets written to log like everything else. Map of inode map gets written in special checkpoint location on disk and is used in crash recovery
LFS Disk Wrap-Around: long-lived information gets copied over-and-over. log results in disk fragments, causing I/O to become inefficient again. Solution: segmented log. Do compaction within a segment; thread between segments. When writing, use only clean segments. Occasionally clean segments.
New Technology File System (NTFS): Attempt: both reliability and performance. Cost: implementation complexity.
Variable length extents rather than fixed blocks. Almost everything (Meta-data and data) is a sequence of <attribute:value> pairs. Mix direct and indirect freely. Directories organized in B-tree structure by default.
Master File Table: Database with Flexible 1KB entries for metadata/data. Extend with variable depth tree.

4 Scheduling

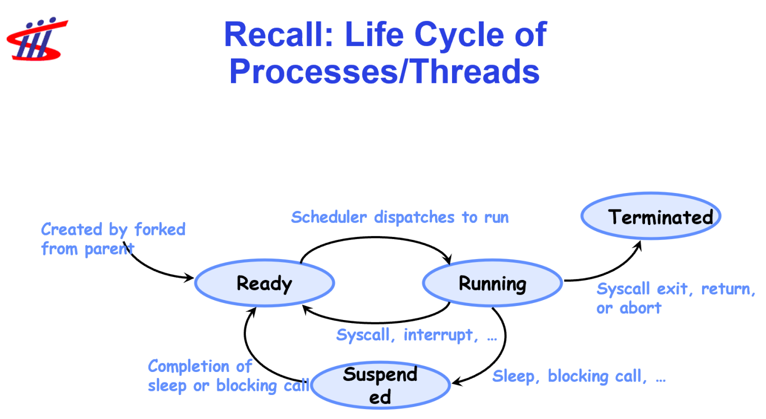
4.1 Scheduling Metrics & Policies

Waiting time: time when the job waits in the ready queue. Response time = waiting time + execution time
Throughput: number of jobs completed per unit of time.
Fairness: all jobs use resource in some equal way.
FCFS (First-Come, First-Served): run until done, non-preemptive. Convoy effect: short jobs are blocked behind long jobs.
RR (Round Robin): each job gets a small unit of time (time quantum, q) to execute, and then gets preempted and added back to end of queue.
SJF (shortest job first): run the job with the shortest execution time first.
SRTF/STCF (shortest remaining time first/shortest time to completion first): preemptive version of SJF.
SJF/SRTF is the optimal policy to minimize average response time.

4.2 Scheduling policy analysis

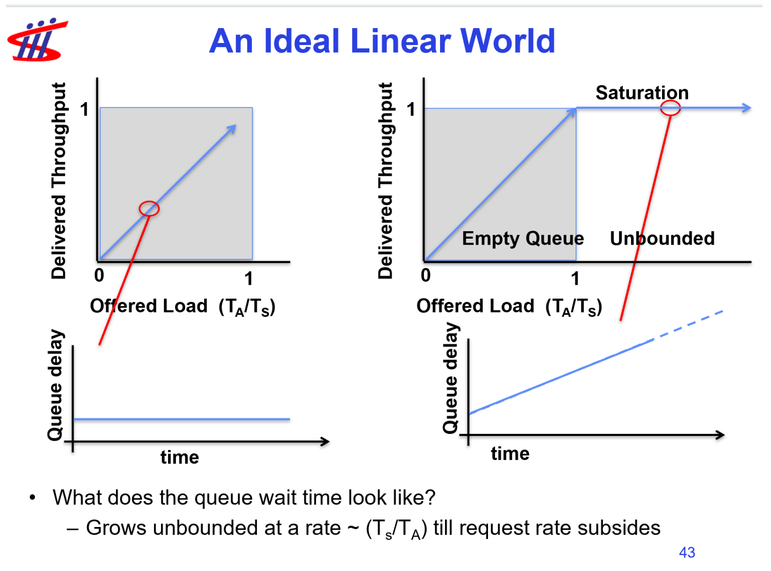
Comparison of SRTF with FCFS and RR: 1. What if all jobs the same length? SRTF becomes the same as FCFS
2. What if jobs have varying length? SRTF (and RR): short jobs not stuck behind long ones
SRTF problems: 1. SRTF can lead to starvation if many small jobs. Large jobs never get to run (unfair). 2. Need to predict future. 3. Bottom line, can't really know how long job will take
Adaptive: Changing policy based on past behavior
Multi-level feedback: Multiple queues, each with different priority. Each queue has its own scheduling algorithm.
Job starts in highest priority queue, drop one level if timeout, push up one level or to top if not timeout.
Between the queues, we need fixed priority scheduling and time slice.
The result approximates SRTF. CPU-bound jobs drop like a rock. Short-running I/O-bound jobs stay near top.
fairness: Strict fixed-priority scheduling between queues is unfair.
fairness gained by hurting average response time: give each queue some fraction of the CPU, or increase priority of jobs that don't get service.
Strict Priority Scheduling: Always execute highest-priority runnable jobs to completion. Each queue can be processed in RR with some time-quantum.
Starvation: Lower priority jobs don't get to run because higher priority jobs. Fix: Dynamic priorities
Deadlock: Priority Inversion, if low priority task has lock needed by high priority task. Fix: Dynamic priorities
Lottery Scheduling: Give each job some number of lottery tickets. On each time slice, randomly pick a winning ticket. On average, CPU time is proportional to number of tickets given to each job
To approximate SRTF, short running jobs get more tickets, long running jobs get fewer tickets
To avoid starvation, every job gets at least one ticket (everyone makes progress)

Advantage over strict priority scheduling: behaves gracefully as load changes
Evaluate a Scheduling algorithm: Deterministic modeling, Queuing models, Implementation/Simulation.



Recall: Life Cycle of Processes/Threads

Created by forked from parent
Scheduler dispatches to run
Terminated
Ready
Running
Syscall exit, return, or abort
Syscall, interrupt, ...
Completion of sleep or blocking call
Suspended
Sleep, blocking call, ...



An Ideal Linear World

- What does the queue wait time look like?
  - Grows unbounded at a rate ~ (T_s/T_A) till request rate subsides

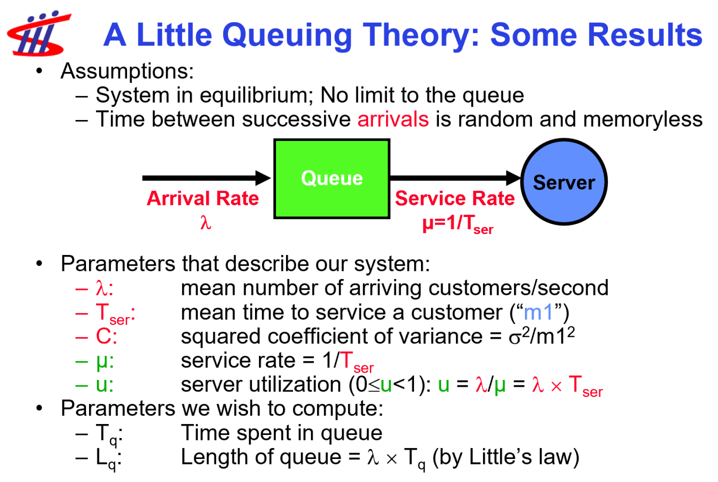
43

4.3 Real-time scheduling

Real-time scheduling: We need to predict with confidence worst case response times for systems. Real-time is about enforcing predictability.
Hard Real-Time: Attempt to meet all deadlines. EDF (Earliest Deadline First), LLF (Least Laxity First).
Soft Real-Time: Attempt to meet deadlines with high probability. Minimize miss ratio / maximize completion ratio (firm real-time).
Earliest Deadline First (EDF): Tasks periodic with period P and computation C in each period: (P_i, C_i) for each task i. The scheduler always schedules the active task with the closest absolute deadline. Schedulable when $\sum_i (C_i/P_i) \leq 1$.

4.4 Modeling performance: Intro to Queuing Theory

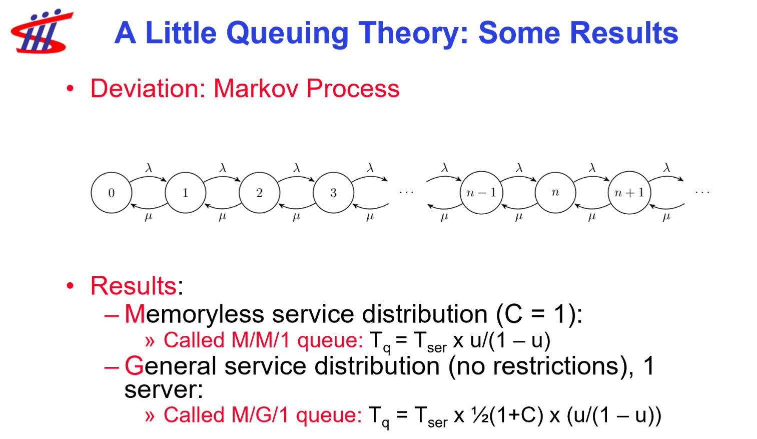
Assume requests arrive at regular intervals, take a fixed time to process, with plenty of time between...
Service rate: operations per sec, $\mu = 1/T_S$; **Arrival rate**: requests per second, $\lambda = 1/T_A$; **Utilization**: $U = \lambda/\mu < 1$
Model the service time. Mean(m1) $= \sum Tp(T)$, $\sigma^2 = \sum(T - \text{m1})^2 p(T)$. Squared coefficient of variance: $C = \sigma^2/\text{m1}^2$
Deterministic if $C = 0$, memoryless or exponential if $C = 1$, for disk response times $C \approx 1.5$ (majority seeks < avg)
e.g. If we model the arrival using $f(T) = \lambda \exp(-\lambda T)$, then $\mathbb{E}[T] = 1/\lambda$, Var$[T] = 1/\lambda^2$, $C = 1$.
Stable State: Arrivals characterized by some probabilistic distribution, Departures characterized by some probabilistic distribution.
Little's Law: In Stable State, we'd have Average arrival rate = Average departure rate, so the average number of tasks in the queuing system (N) is equal to the throughput (B) times the response time (L). $N(\text{ops}) = B(\text{ops/s}) \times L(\text{s})$.



A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless

- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = σ^2/m1^2
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \leq u < 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)



A Little Queuing Theory: Some Results

- Deviation: Markov Process

- Results:
  - Memoryless service distribution (C = 1):
    - » Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server:
    - » Called M/G/1 queue: $T_q = T_{ser} \times ½(1+C) \times (u/(1 - u))$

5 Time in Distributed Systems

5.1 Wall Clocks

Coordinated Universal Time (UTC) & Master Time Facility (MTF).
Computer clocks are not generally in perfect agreement, and time disagreement between machines can result in undesirable behavior.
In DS, we may have that: After 1 minute, errors almost 2 milliseconds. But we may need nanosecond accuracy!
Synchronizing wall clock in Perfect networks: Messages always arrive with propagation delay exactly d (constant). Then for message T, we need only to set the time as $T + d$.
Synchronizing wall clock in Synchronous networks: Messages always arrive with propagation delay at most d (constant). Then for message T, we need only to set the time as $T + d/2$. The error is no more than $d/2$.
Real networks: asynchronous (delays are arbitrary), unreliable (Messages don't always arrive).
Cristian's Time Sync:
Setting: A time server S receives signals from a UTC source Process p wants to know the time.
Algorithm: Process p requests time in message mr and receives time value t in message mt from S. p sets its clock to $t + \text{RTT}/2$, where RTT is the round trip time recorded by p.
Accuracy: Suppose min is an estimated minimum one way delay. Then the accuracy is $\text{RTT}/2 - \text{min}$.
Problems: useful only if RTT $\ll$ accuracy; reliance of the server; RTT increases if queue.
Network Time Protocol (NTP):
Goal: A time service for the Internet, synchronizes clients to UTC reliably from multiple, scalable, authenticated time sources.
Uses a hierarcy of time servers: Class 1 has highly-accurate clocks like atomic clocks, Class 2 servers get time from 1 or 2, Class 3 servers get time from the most servers.
Synchronization: similar to Cristian, but modified to use multiple one-way messages instead of immediate round-trip.
Algorithm: give a <RTT, Offset> pair per time, take the one with minimum packet delay after 8 measures.
Accuracy: Local $\sim$ 1ms, Global $\sim$ 10ms.
Berkeley Algorithm: An algorithm for internal synchronization of a group of servers. No UTC.
Select a master(a.k.a Time Daemon) to use Cristian's algorithm to get time from many clients, and take average.
Master sends time adjustments (average) back to all clients.
If daemon fails / timeout: elect a new one to take over.
How to update local clock: Change the update rate for the clocks. Prevents inconsistent local timestamps.
Wall Clock synchronization: Clocks never exactly synchronized.

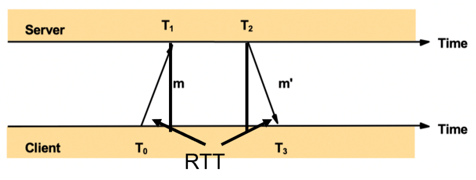
5.2 Logical Clocks

System is composed of a collection of processes, each process consists of a sequence of events.
Local sequence determines local event order.
Communication: sending and receiving messages. Each is an event.
Message transmission delay cannot be neglected.
Causal ordering: $a \to b$ if a may causally affect b.
Concurrent events: $a \| b$ if a and b cannot causally affect each other.
Happens-Before Relations (transitive): 1. Two events occurred at same process p. then they occurred in the order observed by p. 2. When message m is sent between two processes: send(m) happens before receive(m).
Logical Time (a.k.a. Lamport time): Capture just the happens-before relationship between events.
Lamport Clocks Algorithm: Each process p_i has a logical clock L_i which can be used to apply logical timestamps to events.
Rule 1:L_i is incremented by 1 before each event at process p_i.
Rule 2:when process p_i sends message m, it piggybacks $t = L_i$; when p_j receives (m, t) it sets $L_j \leftarrow \max(L_j, t)$ and applies Rule 1 before timestamping the event receive(m).
The Lamport clocks arbitrarily order some concurrent events.
Total-order Lamport Clocks: $L(e) = ML_i(e) + i$, $M \geq$ maximum number of processes, i is the process ID.

5.3 Vector Clocks

We want to give the event a vector time, so that $V(e) < V(e')$ if and only if $e \to e'$.
Method: Label each event by vector $V(e)[c_1, c_2, \cdots, c_n]$. $c_i = \#$ events in process i that causally precede e.
Algorithm: Initially, all vectors $[0, 0, \cdots, 0]$. For event on process i, increment own c_i.
Label message sent with local vector. When process j receives message with vector $[d_1, d_2, \cdots, d_n]$: Set each local entry k to $\max(c_k, d_k)$, increment value of c_j.

NTP Protocol: One round (2)

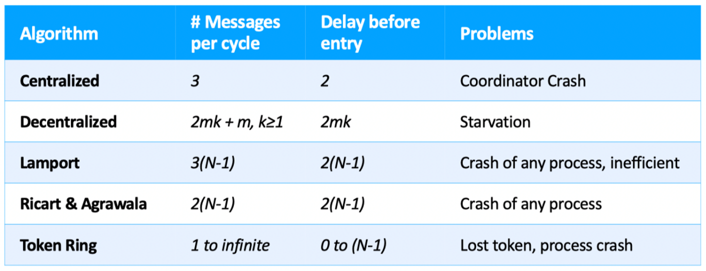


Observe: We can measure server processing time accurately!

RTT = wait_time_client – server_proc_time
= $(t_3-t_0) - (t_2-t_1)$
Time adjustment at client = (t_3 + Offset) = t_2 + RTT/2 (Cristain's)
So, Offset = t_2 + RTT/2 - t_3
= $((t_1-t_0) + (t_2-t_3))/2$
Why do we compute RTT and Offset?

19

A Comparison of the Five Algorithms



Algorithm	# Messages per cycle	Delay before entry	Problems
Centralized	3	2	Coordinator Crash
Decentralized	2mk + m, k≥1	2mk	Starvation
Lamport	3(N-1)	2(N-1)	Crash of any process, inefficient
Ricart & Agrawala	2(N-1)	2(N-1)	Crash of any process
Token Ring	1 to infinite	0 to (N-1)	Lost token, process crash

- k = number of retries in getting majority

47

6 Mutual Exclusion

6.1 Totally ordered multicast

Useful DS here: priority queue.
Totally ordered multicast: A multicast operation by which all messages are delivered in the same order to each receiver.
Assume: all messages sent by one sender are received in the order they were sent (TCP on all channels); no messages are lost.
Algorithm: 1. Message to be sent is timestamped with sender's logical time 2. Message is multicast (including to itself)
3. After received: put into local queue, ordered according to timestamp, and receiver multicasts acknowledgement
Problem: Squared messages.

6.2 Mutual Exclusion

Mutex Requirements: 1. Correctness/Safety: At most one process holds the lock/enter C.S. at a time.
2. Fairness (= no starvation) : Any process that makes a request must be granted lock eventually.
Main Distributed Mutex Req: 1. Low message overhead 2. No bottlenecks 3. Tolerate out-of-order messages
Extra Distributed Mutex Req: 1. Allow processes to join protocol or to drop out
2. Tolerate failed processes 3. Tolerate dropped messages
Our assumption: 1.Total number of processes is fixed at n 2. No process fails or misbehaves
3. Communication never fails, but messages may be delivered out of order.

6.3 Mutual Exclusion Algorithms

Centralized Algorithm:

```
@Coordinator:
while true:
    m = Receive()
    if m == (Request, i): if Available(): Send (Grant) to I; else: Add i to Q
    elif m == (Release) && !empty(Q): Remove ID j from Q; Send (Grant) to j
```

Leader election & Bully Algorithm: Select a unique process as the leader - in fact, the available with highest id.
Algorithm: If P notices that leader has failed, P would send an ELECTION message to all processes with higher ids.
If no one responds, P wins and becomes coordinator; If one of the higher-ups answers, it takes over and P's job is done.
Fully Decentralized Algorithm: Get a majority vote from $m > n/2$ coordinators. A coordinator replies immediately to a request with GRANT or DENY.
What if you get less than m votes? Backoff and retry later. It may cause starvation.
Lamport Mutual Exclusion: based on Lamport Totally Ordered Multicast.
ACK only to requestor (fewer messages), Release after finished (additional message).
When node i wants to enter CS, it multicasts time-stamped request to all other nodes (including itself).
Wait for replies from all other nodes. If own request is at the head of its queue and all replies have been received, enter CS
Upon exiting CS, remove its request from the queue and multicast a release message to every process.
For other nodes, after receiving a request, enter the request in its own request queue (ordered by timestamps) and reply with the timestamp.
After receiving release message, remove the corresponding request from its own request queue.
If its own request is at the head of its queue and all replies have been received, enter C.S.
Ricart & Agrawala Mutual Exclusion: Need no release message for the Lamport Mutual Exclusion, but only $n - 1$ replies from others to get into the C.S.
It's Deadlock free and Starvation free. (proved by the Lamport clock)
Token Ring Algorithm: Organize the processes involved into a logical ring. One token at any time $\rightarrow$ passed from node to node along ring.
None of these algorithms can tolerate failed processes or dropped messages.

7 Replication and Consensus

7.1 Replication

Machine failure modes:Fail-Stop:The machine does not do anything evil at failure;Byzantine:Everything else.
This section minds only Fail-Stop.
CAP thereom: consistency, availability, partition-resilience. A DS can only realize two. e.g. Paxos satisfies CP.
Replication: Store each data item on multiple nodes. Single-Master/Multi-Master.
Read-only is easy, Read-Write is hard.
Benefits when read-only: Scalability; Locality; Availability.
Primary-Backup for RW: Must wait for all backups to acknowledge before replying to writer client.
Quorum based replication: Define a replica set of size N. `put()` waits for acks from at least W replicas, `get()` waits for responses from at least R replicas. $W + R > N$.
Replicated state machine: Make a collection of machines to agree on the same value/command. A majority of the machines executes same commands in same order.

7.2 Consensus

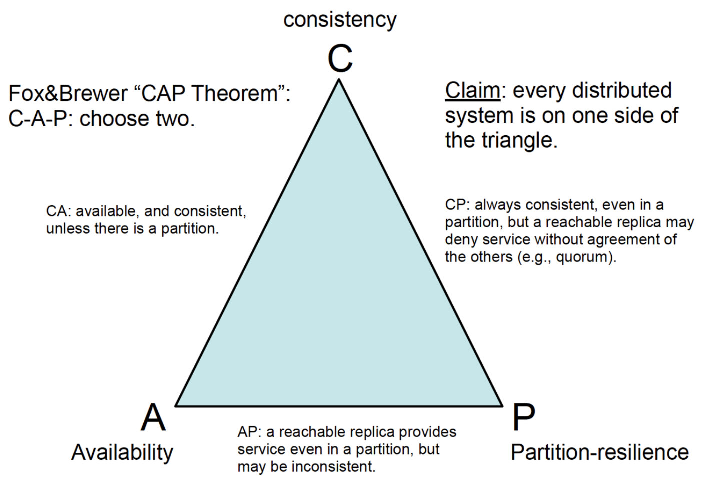
Fischer-Lynch-Paterson (FLP): No consensus can be guaranteed in an asynchronous communication system in the presence of any failures.
Properties for Correct Consensus:
1. Termination: All correct processes eventually decide. liveness property.
2. Agreement: All correct processes select the same value. safety property.
3. Integrity: All deciding processes select the right value. non-trivial property.
Paxos consensus protocol: An asynchronous consensus algorithm. guaranteed safe, not guaranteed live.
Key idea: if a value is chosen by the majority in an earlier round, we must let the value to remain unchanged.
Roler: The Proposers & Acceptors.
Prepare Phase: Broadcast a proposal, and invalidate older proposals that have not decided yet for someone.
Accept Phase: Broadcast accept message with the proposal, and value is chosen if received majority votes.
Paxos and Replicated state machine: Use Paxos on the logs of RSMs.
Membership Change: Add/Remove nodes, change node IDs, majority may change. Solution: Record config in log.

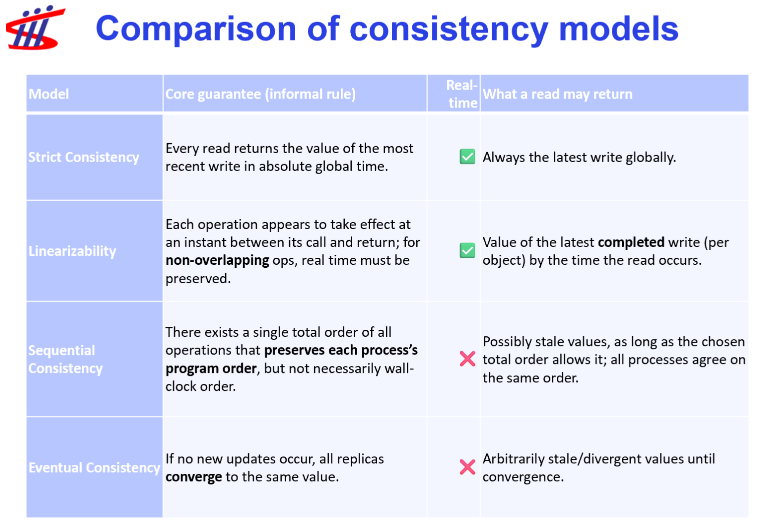
7.3 Two phase commit protocol (2PC)

Everyone can veto (say no). Need to wait for everyone.
Each transaction commit is led by a coordinator (C, or transaction manager TM); other participants in 2PC are called Resource Managers RMs or Participants (P); Widely taught and used.
Steps: 1. Tx requests commit, by notifying coordinator (C). 2. Coordinator C requests each participant (P) to prepare.
3. Each P validates the request, logs validates updates locally, and responds to C with its vote(commit/abort). if P votes commit, Tx is prepared at P.
4. Iff all P votes to commit, C writes a commit record to its log and Tx is committed. Else abort.
5. Coordinator notifies participants of the outcome for Tx. Each P logs the outcome locally and releases any resources held for Tx.
Problems: Might block forever if coordinator fails or disconnects; Might rollback indefinitely.

7.4 Consistency models

Strict Consistency: Read always returns value from latest write. Act just like a single machine
Linearizability:No real time. it obeys real time. Read always returns value from latest write
Sequential Consistency:All nodes see operations in some sequential order. Operations of each process appear in-order in this sequence. Allow to read some delayed results if such sequence exists.
Eventual Consistency:e.g.Resolve conflicting versions after failure recovery.


consistency
C
Fox&Brewer "CAP Theorem":
C-A-P: choose two.
Claim: every distributed system is on one side of the triangle.
CA: available, and consistent, unless there is a partition.
CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).
A
Availability
AP: a reachable replica provides service even in a partition, but may be inconsistent.
P
Partition-resilience



Comparison of consistency models

Model	Core guarantee (informal rule)	Real-time	What a read may return
Strict Consistency	Every read returns the value of the most recent write in absolute global time.	✓	Always the latest write globally.
Linearizability	Each operation appears to take effect at an instant between its call and return; for **non-overlapping** ops, real time must be preserved.	✓	Value of the latest **completed** write (per object) by the time the read occurs.
Sequential Consistency	There exists a single total order of all operations that **preserves each process's program order**, but not necessarily wall-clock order.	✗	Possibly stale values, as long as the chosen total order allows it; all processes agree on the same order.
Eventual Consistency	If no new updates occur, all replicas **converge** to the same value.	✗	Arbitrarily stale/divergent values until convergence.

8 Blockchains

Key problem: How to establish TRUST among distributed parties when: Each one does not trust another, and There is no trusted third party/intermediary?
The Goal System: decentralized with multiple users; Each member can track all history and transactions; Records are unchangeable once confirmed.

8.1 The offline blockchain data structure

Hash Pointer: pointer to where some info is stored, and it is a (cryptographic) hash of the info.
key idea: build data structures with hash pointers
Merkle Tree: Binary tree of hash pointer, only remember the root of the tree, any temper in data is obvious
Proof of membership / non-membership, Can verify membership in O(log n) time/space
Block: The time when the block producer generates the block; The hash of the previous block; A record (or so called transaction) is the major information in the block, e.g., "Alice transfers $5 to Bob"
Connecting Blocks with Hash Pointer
Immutability:Any changes/modifications to previous block can be easily detected by hash

Block Chain: The first block is genesis block, with no previous block; New blocks are appended at the end of the chain; Altering a block requires changing all blocks after

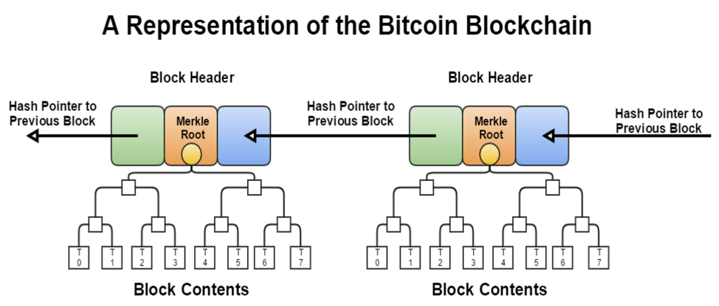
8.2 Distributed Chain: The P2P network protocol

Motivation for decentralization: Copy the entire chain to many many nodes, so no one can rewrite it (entirely). Everyone gets exactly the same copy, defined as consensus. How many coins you own is a consensus from the network!
Bitcoin P2P network: Ad-hoc protocol (runs on TCP port 8333), Ad-hoc network with random topology, No leader node (decentralized), New nodes can join at any time, Forget non-responding nodes after 3 hr
Transaction propagation: flooding

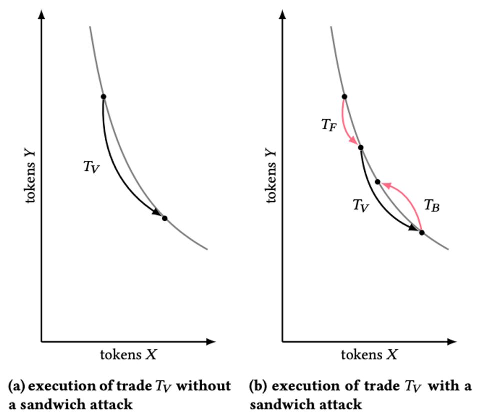
8.3 Nakamoto Consensus

P2P network solves the communication (broadcast) problem, but still no mention of ordering or consensus. This consensus is even harder
Consensus algorithm (Simplified): Letting a single node to decide the transactions and their ordering.
1. New transactions are broadcast to all nodes 2. Each node collects new transactions into a block 3. In each round a random/selected node gets to broadcast its block 4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures) 5. Nodes express their acceptance of the block by including its hash in the next block they create 6.(and resolve the inconsistency cases if they happen)
Goal: Select nodes in proportion to resource spent. Let nodes compete for right to create block. Competition takes resource (e.g. computing power)
Proof of Work: The more computing power you have, the more likely you win. Solve Hash Puzzles: find nonce that H(header|nonce) could be very small (lots of leading zeros)
Properties of a good PoW algorithm: Difficult to Compute for Miners; Parameterizable Cost; Trivial to Verify
Incentives for Miners: Creator of block gets to include special coin-creation transaction in the block, i.e., "coinbase" transaction; Receive all transaction fees (tips); choose recipient address of this transaction
# BTC created per block is predetermined: currently 3.125 BTC, halves every 4 years
The Longest-Chain Rule: When facing different versions of the chain, follow the longest one.
K-delay Confirmation: Forking should be resolved with long delay. So Normally give up K dalay. always $k = 6$.
Analysis: adversary has p fraction of power. the probability adversary catches up from z blocks behind is no more than $(p/(1-p))^z$.
Problem: Low transaction throughput

Chain Structure of Bitcoin


A Representation of the Bitcoin Blockchain
Block Header
Block Header
Hash Pointer to Previous Block
Hash Pointer to Previous Block
Hash Pointer to Previous Block
Merkle Root
Merkle Root
T 0
T 1
T 2
T 3
T 4
T 5
T 6
T 7
Block Contents
Block Contents

Sandwich Attack



9 Blockchains & BHT

9.1 Challenges to public blockchains

How to store the bitcoins: Professional mining centers
Pros: Security against botnets; Makes the miners interested in the long-term stability of the system
Cons: Makes the whole process "non-democratic"; Easier to attack by very powerful adversary?
Mining pools: lower variance, but 98% of the blocks in Bitcoin are mined by mining pools! Risk of centralization.
Performance Problem: too slow with full network validation.
Solution: Layer 2 and rollups: Optimistic / ZK; New, fast chain protocols
e.g. Etherum GHOST protocol, Conflux protocol
Off-chain data and Real world asset:
advantages: liquidity, transparency (and potentially privacy), participate in DeFi and can be heavily speculated
Requires extensive on-/off-chain interactions, including: Proving you actually own the off-chain assets, Real-world identity (KYC) verification, Using and settling with off-chain assets

9.2 DeFi

CeFi (Centralized Finance): Financial services run by firms that custody user funds
DeFi (Decentralized Finance): Smart-contract based services on public blockchains; users self-custody; code executes rules
Core mechanisms in DeFi:Collateralized lending, Automated market making (AMM), Staking / restaking
DeFi Lending Protocols:post one token as collateral and borrow another; When prices move and collateral value falls short, liquidation is triggered.
DeFi AMM:Invariants: $\#X * \#Y = K, \#X * P_X = \#Y * P_Y$. So, $P_X/P_Y = \#Y/\#X = K/(\#X)^2$
Knowing $\#X\#Y$, you know the price of both token after any trade.

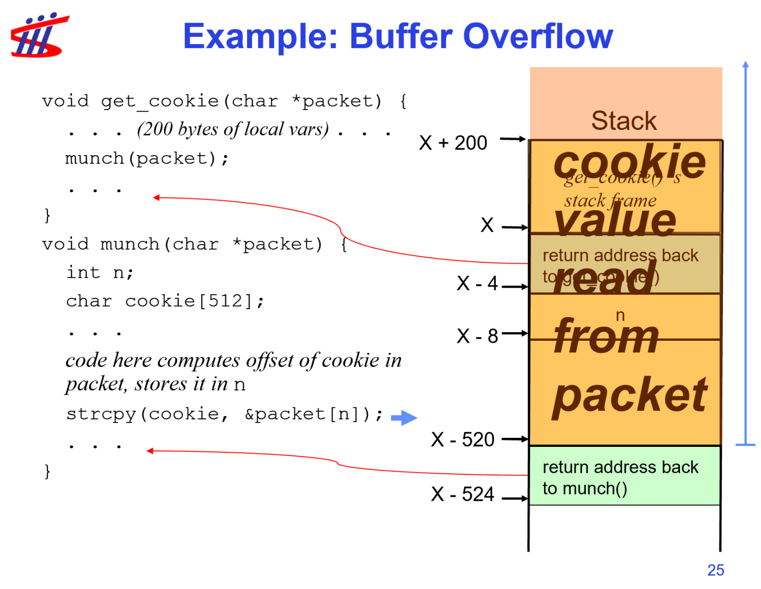
9.3 Defi-specific challenges

MEV: Miner (Maximum) Extractable Value. The maximal value that miners can extract from reordering, inserting, or censoring transactions, plus the gas fees.
Miners determines the ordering of the transactions:competition among miners for transaction order; can lead to unfair advantages
MEV strategies:e.g. Liquidation attacks, Sandwich attack
Liquidation MEV: 1. Reactive monitoring. Watch for liquidation transactions and try to insert a transaction ahead of them (front-run). 2. Proactive monitoring. Watch price-moving quotes/trades and judge whether they will trigger a liquidation. Immediately place a transaction right after that price-moving trade (back-run).
MEV Auction: miners pick the bundle that offers the highest payment (or do the MEV themselves) and execute it.

9.4 Adding a little trust: consortium chains

Byzantine General Problem: No solution for three processes can handle a single traitor. In a system with f nodes with Byzantine failures, agreement can be achieved only if there are $2f + 1$ (more than $2/3$) nodes correctly.
PBFT (Practical Byzantine Fault Tolerance):
property: tolerates <= f Byzantine failures using $3f + 1$ replicas.
idea: 1. Broadcast every message to everyone 2. Authenticate communications
Algorithm: 1. Pre-prepare phase. The primary assigns a unique sequence number n and broadcast PRE-PREPARE message to all replicas (of course, all messages signed) Acceptors: Checks authenticity of message, and whether n is fresh.
2. Every replica: broadcast a PREPARE message to everyone else, containing the sequence number n, (signed, of course). Everyone checks the message, accept only valid message
Enters PREPARED state if have received the PRE-PREPARED message before, and received $2f + 1$ distinct valid messages
3. Every replica: when PREPARED (i.e. received enough messages), broadcast a COMMIT message
Everyone checks the message, accept only valid message
Enters COMMITTED-LOCAL state if have already in the PREPARED state for the message and received $2f + 1$ distinct COMMIT messages

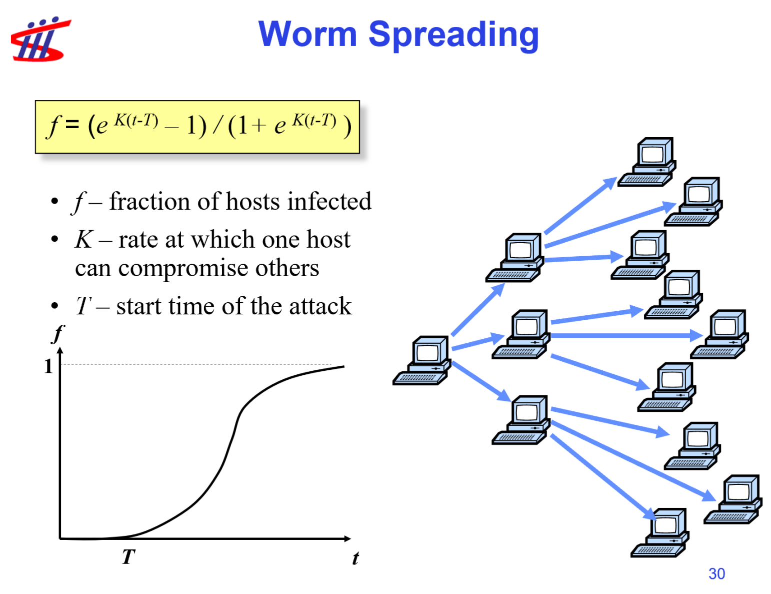
The global commit point: When at least $f+1$ non-faulty node enters into the PREPARED state. So COMMITTED-LOCAL at any node implies COMMITTED state will no longer change.
Change the primary: From replica p to $p+1$. Called "view change".
Performance: Need three rounds of communication, two multicasts. Long latency, high bandwidth



Example: Buffer Overflow

```
void get_cookie(char *packet) {
  . . . (200 bytes of local vars) . . .
  munch(packet);
  . . .
}
void munch(char *packet) {
  int n;
  char cookie[512];
  . . .
  code here computes offset of cookie in
  packet, stores it in n
  strcpy(cookie, &packet[n]);
  . . .
}
```

Stack
X + 200
cookie
get_cookie()'s stack frame
value
X
return address back to get_cookie()
read
X - 4
n
from
X - 8
packet
X - 520
return address back to munch()
X - 524
25



Worm Spreading

$$f = (e^{K(t-T)} - 1) / (1 + e^{K(t-T)})$$

- f – fraction of hosts infected
- K – rate at which one host can compromise others
- T – start time of the attack

10 Security

10.1 Access control

Authentication&Authorization:A&A. who the client is, has authorization for requests or not.
Access Control Matrix: Describes who (subject) can do what (rights) to what/whom (object/subject).
Allowed Operations (Rights): r,x,w
Access Control Lists(ACL): ACM Order by columns. e.g. `file1: {(Andy, rx)(Betty, rwx)(Charlie, rx)}`.
Capability List: ACM Order by rows. e.g. `Andy: {(file1, rx)(file2, r)(file3, rw)}`.
ACL-List solves: Given an object, what subjects can access it, and how?
C-List solves: Given a subject, what objects can it access, and how? e.g. privileged data structure, capability-based addressing
ACL-based systems are more common than C-List based.

10.2 Host Compromose

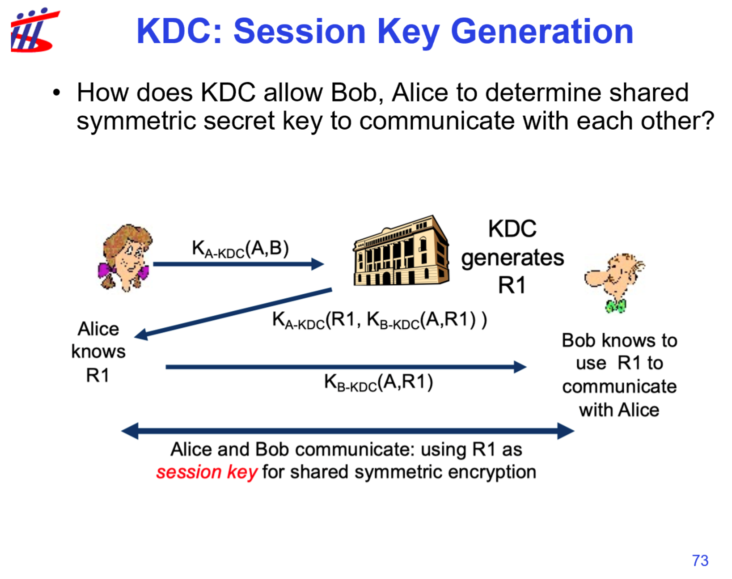
Worm: Replicates itself usually using buffer overflow attack
Virus: Program that attaches itself to another (usually trusted) program or document
Trojan horse: Program that allows a hacker a back door to compromised machine
Botnet (Zombies): A collection of programs running autonomously and controlled remotely. Can be used to spread out worms, mounting DDoS attacks
bypass OS protection: Buffer Overflow
As the worm repeatedly replicates, it grows exponentially fast because each copy of the worm works in parallel to find more victims.
OS and Hardware Protection:Don't write buggy software(Program defensively, Use code checkers); Use Type-safe Languages (Java, Perl, Python, ...); Use HW support for no-execute regions(stack, heap);Leverage OS architecture features;Add network firewalls
Firewall: Restrict traffic between Internet and devices (machines) behind it based on Source address and port number, Payload , Stateful analysis of data
Properties: Easier to deploy firewall than secure all internal hosts; Doesn't prevent user exploitation/social networking attacks; Tradeoff between availability of services (firewall passes more ports on more machines) and security

10.3 Denial of Service(DoS)

General Form: Prevent legitimate users from gaining service by overloading or crashing a server
Effect on Victims: Buggy implementations lead to crash; Better implementations limit the number of unfinished connections and drop users; Users can't access the targeted service on the victim because the unfinished connection queue is full (DoS)
SYN Attack: doing much SYN request as 3-Way Handshaking in TCP. Solution: SYN Cookies. Address would not be allocated if client address is spoofed.
Reflection DoS: Cause one non-compromised host to attack another.
Identifying and Stop Attacking Machines: Develop techniques for defeating spoofed source addresses 1. Egress filtering. 2. IP Traceback.
Distributed Denial-of-Service Attacks(DDoS): Zombie botnet used to generate massive traffic flows/packet rates. simple and real

10.4 Key distribution

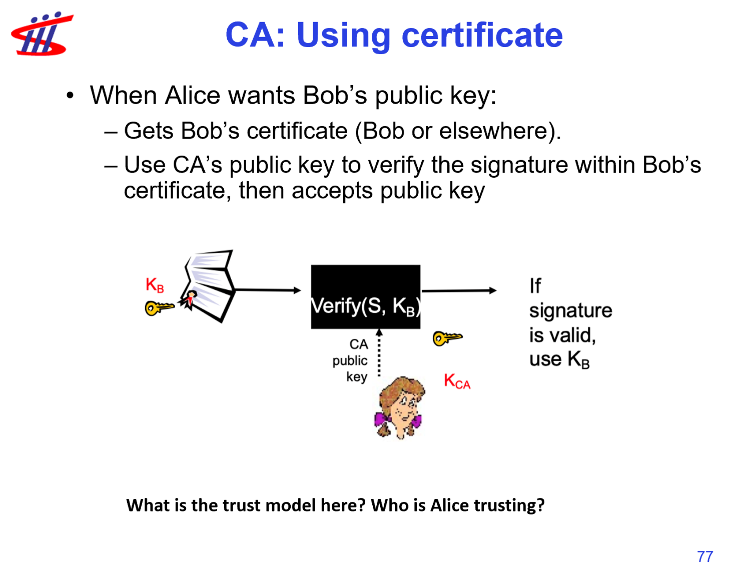
Key Distribution Center (KDC) in Kerboros: gives different secret symmetric key to users.
KDC can expose our session keys to others! Centralized trust and single point of failure.
KDC model is mostly used within single organizations.
Public Key Infrastructure (PKI): A system in which "roots of trust" authoritatively bind public keys to real-world identities.
Certification Authorities (CA): binds public key to particular entity, E. E provides "proof of identity" to CA. Certificate contains E's public key AND the CA's signature of E's public key.
Better scaling: CA must only sign once... no matter how many connections the server handles.
If CA is compromised, attacker can trick clients into thinking they're the real server.



KDC: Session Key Generation

- How does KDC allow Bob, Alice to determine shared symmetric secret key to communicate with each other?

$K_{A\text{-}KDC}(A,B)$
KDC generates R1
$K_{A\text{-}KDC}(R1, K_{B\text{-}KDC}(A,R1))$
Alice knows R1
Bob knows to use R1 to communicate with Alice
$K_{B\text{-}KDC}(A,R1)$
Alice and Bob communicate: using R1 as *session key* for shared symmetric encryption
73



CA: Using certificate

- When Alice wants Bob's public key:
  - Gets Bob's certificate (Bob or elsewhere).
  - Use CA's public key to verify the signature within Bob's certificate, then accepts public key

What is the trust model here? Who is Alice trusting?

10.5 Secure channels

Authentication: Who am I talking to?
Confidentiality: Is my data hidden?
Integrity: Has my data been modified?
Availability: Can I reach the destination?
Transport Layer Security (TLS): a.k.a Secure Socket Layer (SSL).Used for protocols like HTTPS. Special TLS socket layer between application and TCP (small changes to application). Handles confidentiality, integrity, and authentication. Uses "hybrid" cryptography.
Handshake of TLS Steps: 1. Clients and servers negotiate exact cryptographic protocols
2. Client's validate public key certificate with CA public key.
3. Client encrypt secret random value with servers' public key, and send it as a challenge.
4. Server decrypts, proving it has the corresponding private key.
5. This value is used to derive symmetric session keys for encryption & MACs.

Main idea: Symmetric and asymmetric primitives provide Confidentiality, Integrity, Authentication

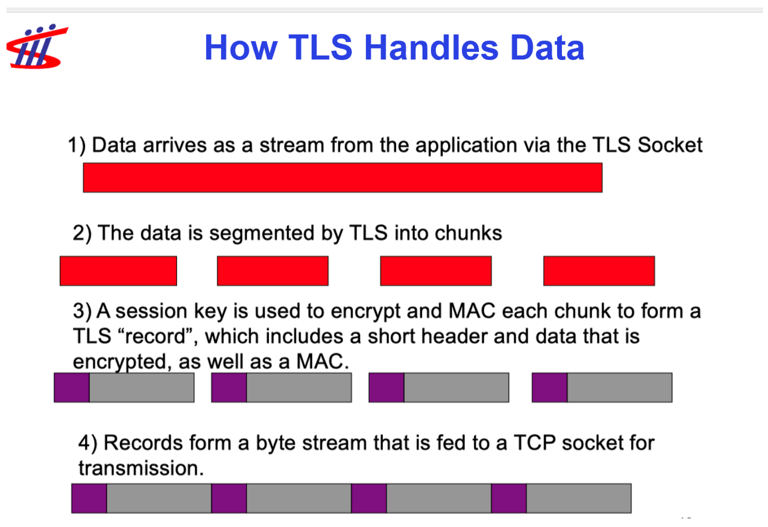
Diffie-Hellman Key Exchange: How to generate keys between two people, securely, no trusted party, even if someone is listening in.

vulnerable to man-in-the-middle attack. Attacker pair-wise negotiates keys with each of A and B and decrypts traffic in the middle.

solution: "perfect forward secrecy", take it exchanges in the protocols before.

Randomized Routing:Hide message source by routing it randomly. Routers don't know for sure if the apparent source of a message is the true sender or another router

Onion Routing:Sender chooses a random sequence of routers. Some routers are honest, some controlled by attacker. Sender controls the length of the path.



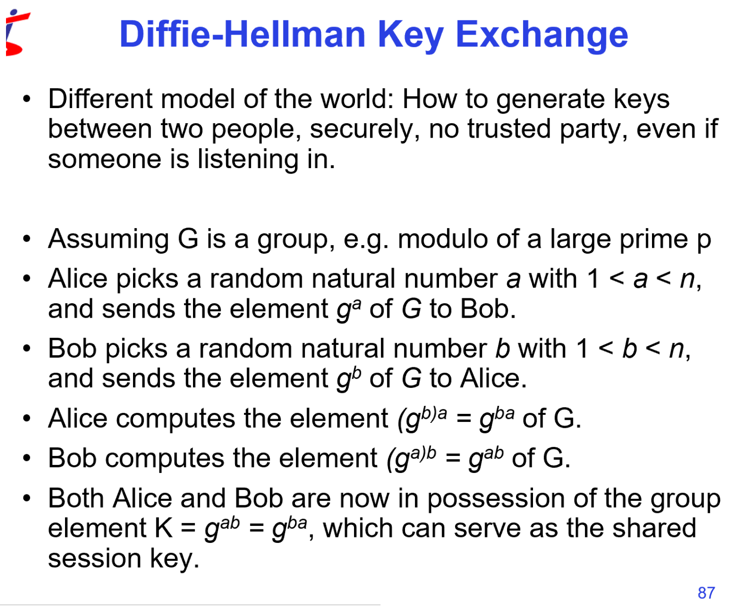
How TLS Handles Data

1) Data arrives as a stream from the application via the TLS Socket

2) The data is segmented by TLS into chunks

3) A session key is used to encrypt and MAC each chunk to form a TLS "record", which includes a short header and data that is encrypted, as well as a MAC.

4) Records form a byte stream that is fed to a TCP socket for transmission.

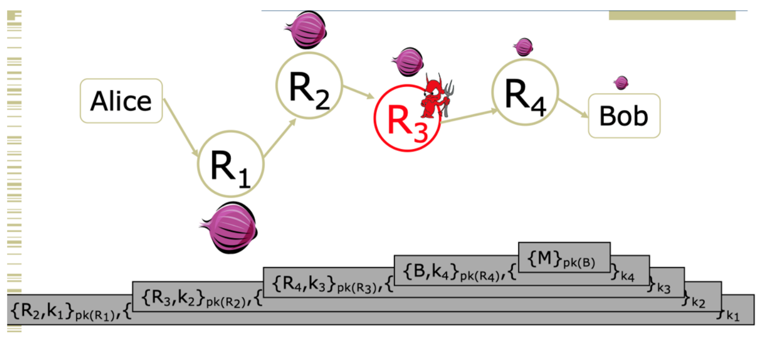


Diffie-Hellman Key Exchange

- Different model of the world: How to generate keys between two people, securely, no trusted party, even if someone is listening in.
- Assuming G is a group, e.g. modulo of a large prime p
- Alice picks a random natural number *a* with 1 < *a* < *n*, and sends the element g^a of *G* to Bob.
- Bob picks a random natural number *b* with 1 < *b* < *n*, and sends the element g^b of *G* to Alice.
- Alice computes the element $(g^b)^a = g^{ba}$ of G.
- Bob computes the element $(g^a)^b = g^{ab}$ of G.
- Both Alice and Bob are now in possession of the group element $K = g^{ab} = g^{ba}$, which can serve as the shared session key.

87

Route Establishment



Routing info for each link encrypted with router's public key Each router learns only the identity of the next router

Note: k1 , k2 , k 3 etc are session keys, so when each router (R1 , R2 , .. Rn) use their private keys to decrypt the packets, they can only then get the next hop (e.g. R2) and the session key (k1) to decrypt the rest of the packet and send it along.

94

11 Appendix

Nothing now.