

Parallel Job Scheduling

"Job" is a collection of processes/threads that cooperate to solve some problem (not independent). **Space Sharing** - each job has dedicated processors **Time Sharing** - Multiple jobs share same processors

- Synchronize over shared data -- other threads in job may not get far
- Cause/effect relationships (producer-consumer problem)
- Synchronizing phases of execution -- pace of slowest thread

**Scheduler-awareness:** 1. Know threads are related, schedule all at same time (space sharing; all threads are from same job; time sharing: group threads that should be scheduled together) 2. Know when threads hold spinlocks (don't deschedule lock holder, extends timeslice) 3. Know about general dependences (infer producer/consumer relations)

**Space Sharing Scheduling:** Divide processors into groups, assign job to dedicated set of processors (ideally one CPU per thread in job), job waits until required number of CPUs are available (batch scheduling)

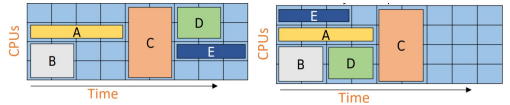
**Pros:** All runnable threads execute at the same time, reduce context switch overhead, strong affinity

**Cons:** Inflexible: CPUs in one partition may be idle while another has multiple jobs, difficult to deal with dynamically changing job sizes

**FCFS Space Share:** Scheduling convoy effect, long average wait times due to large job, leads to fragmentation of CPU space

**Solution:** Fill CPU "holes" from queue in FCFS order (not FCFS tho), Want to prevent "fill" from delaying threads that were queued earlier

**EASY:** Extensible Argonne Scheduling sYstem -- make reservation for next job in queue.



**EASY Variations:**

- Ordering alternative: include priority in queue, prevents starvation
- Reservation alternatives: queued jobs get reservation, queued job gets reservation if it has been waiting more than a threshold
- Queue lookahead: Use DP to determine optimal packing

**Co-scheduling:** Identify "working set" of processes (analogous to working set of memory pages) that need to run together

**Gang scheduling:** All-or-nothing: co-scheduled working set is all threads in the job (get scheduling benefits of dedicated machine)

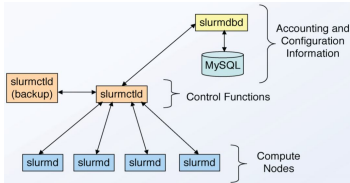
**Issues:** All CPUs must context switch together (to avoid fragmentation, construct groups of jobs that fill a slot on each CPU)

**Alternatives:** Paired gang scheduling (identify groupings with complementary characteristics and pair them, when one blocks, other one runs); Only use gang scheduling for thread groups that benefit

1. *Knows threads are related, schedules at the same time*
2. *Know when threads hold spinlocks:* thread acquiring spinlock sets kernel-visible flag, clears flag on release, scheduler will not immediately deschedule a thread with the flag set

**OS Noise:** Asynchronous OS activities perturb nice scheduling properties of running jobs together (up to factor of 2)

**SLURM:** Performs resource management within single cluster: allocates access to computer nodes and their interconnections, launches parallel jobs and manages them (I/O, signals, time limits, etc), manages contention in the queue.



Virtual Memory Management

**Goals:** Efficiency, transparency, protection and sharing

**Multi-level page table tradeoffs:** Saves space, adds more levels of indirection when translating addresses, more complexity in translation

**Paging time limitations:** Memory reference overhead (2 memory reads per address lookup, hierarchical page tables make it worse -- use hardware cache of lookups i.e TLB)

**Increasing TLB Coverage:** Increase TLB size, increase page size, use superpages (1 TLB entry per superpage, contiguous and aligned)

**Superpage TLB:** Superpage sizes must be power-of-multiples of the base page size, must be aligned in both virtual and physical memory

- TLB entry for superpage: only single reference bit, dirty and protection bit, includes page size, must be supported by MMU of processor

**Superpage Problem:**

- Allocation: what frame to choose on page fault
- Promotion: Combine base pages into superpage
- Demotion: break superpage into smaller superpages (or base pages)
- Fragmentation: Need contiguous physical pages to create superpage

**Relocation-based alloc:** When bringing a page in main memory, put it anywhere in RAM, will need to relocate it to a suitable place when we merge into a superpage

- Research: One superpage size only, designed to work with proposed partial sub-block TLBs (TLB entry: only one PPN, but may have multiple valid bits)

**Reservation-based alloc:** Put it in a location that would us "grow" a superpage around it, must pick a maximum size for the superpage

- Research: Move pages at promotion time (migrate pages to contiguous region when likely beneficial), however must recover copying cost

**Navarro Design:** Use preemptible reservations, promote to superpage when all the base pages are valid. Once an application touches the first page of a memory object, then it is likely that it will quickly touch every page of that object. *Optimistic Policy:* No a priori knowledge that other base pages on the same superpage will get accessed soon, superpages are large as possible and as soon as possible (even first page fault)

**Allocation Policy and Reservation Size:**

- On page fault, pick largest aligned superpage that contains faulting base page and does not overlap with other alloc pages or tentative superpages (i.e. reservation) | *Fixed:* Go to biggest size that is no larger than memory object, *Dynamic:* No such restriction, but limit to current size of object to avoid waste.

- Try preemption before resigning to a smaller size, last resort: assign smaller extent that desired

**Reservation lists:** Keep track of frame extents that are not fully populated, lists sorted by time of most recent page frame alloc

**Incremental promotions:** Superpage is created whenever any superpage-sized and aligned extent within a reservation fully populated

**Incremental Demotion:** When base page of superpage is evicted from memory or access rights changed, demote superpage until eviction

**Speculative Demotion:** One reference bit per superpage, on memory pressure, demote superpages speculatively (a bit each), unused base pages detected and evicted, re-promoted as pages are referenced

**Fragmentation Control:** Mostly done by buddy allocator (coalescing available memory regions), modified page replacement daemon (contiguity-aware page replacement), cluster wired pages (tend to get scattered over time, cluster them in contiguous region)

**Superpage take-aways:** Different applications benefit most from different superpage size -- let system choose among page size, contiguity-aware page replacement daemon can maintain enough contiguous regions, huge penalty for not demoting dirty superpages, overheads are small

**Page coloring:** Assign color to virtual and physical pages, on page fault, allocate a physical page with same colour as virtual, exploits spatial locality

Num Colors = (cache size) / (pg size \* associativity)

Page color = Page number % num\_colors

**Bin Hopping:** Assign colours to physical pages and keep per-colour free lists. On page fault, allocate physical page of next colour from last one previously allocated -- exploits temporal locality

**NUMA Multiprocessors:** Allocate "local" memory as much as possible (may not be local at access time), keep per-memory bank free lists.

Distributed Shared Memory

**Features:** Multiple computers, connected by communication network, cooperating to share resources and services

**Distributed Shared Memory:** Allow processes on networked computers to share physical memory through a single virtual address space

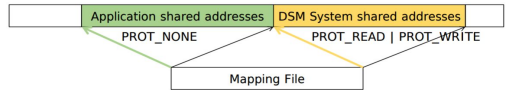
**Central Server DSM:** All read, write of shared data sent to server, server handles request and sends acknowledgement

**Sharing Granularity:** Object-based - pure software approach, allows granularity to be determined by object size (less false sharing). Page based -- can leverage paging hardware, unit of sharing is page size, false sharing is more likely

**Page-Based DSM:** Physical memory on each node holds pages of shared virtual address space, page table entry for a page is valid if the page is local. DSM protocol handles page fault to retrieve remote data. (can be implemented at user-level)

**DSM Page Fault:** DSM system maintains metadata about each shared page, install SIGSEGV handler to catch invalid access

**Atomic Page Update:** Multiple threads in process share OS page table and need to control multiple accessing remote page. Map file to two virtual addresses, one for application, one for DSM system. Use different protections on each, SIGSEGV allows access to DSM system address to update page, only grant access to application address when page fault is fully handled



**Locating remote data:** Page migrates to the node where most recent access happened | Problems: Central server is bottleneck, sol: distributed directory; virtual page exists on 1 machine (no caching) sol: allow replication

**Simple Replication (Read):** Multiple readers, single writer. One node is granted read-write copy or multiple nodes with read-only copies.

*On read:* Set access rights to read-only on any writable copy

*On write:* Revoke write permission from other writable copy, get read-write copy of page, invalidate all copies of page at other nodes

**Full Replication:** Multiple readers, multiple writers. More than one node can have writable copy of page, access to data controlled/consistent

*On request for page copy:* Add requestor to copyset, send page content

*On request to invalidate:* Send invalidation request to all nodes in copyset and wait for acknowledgements

**Consistency Model:** 1. Defines when modifications to data may be seen at a given processor 2. Defines how memory will appear to a programmer (restrict value return by a read of mem loc)

- Must be well-understood (reasoning correctness/what optimizations)

**Sequential Consistency:** Mem ops must execute one at a time, single processor appear to execute in program order, interleaving among processors is ok (but all processor observe same interleaving). Must ensure that previous mem op is complete before moving on (ack write has completed, with caching must send invalidate or update message, all messages must be acknowledged)

**Weak Consistency:** Relax either program order (or write atomicity) or data races and reordering constraints. Allow reads/writes to different memory locations to be reordered. Ok to relax consistency because no need to propagate writes sequentially, or at all until thread leaves CS

**Requirements:** Accesses to sync vars are sequentially consistent, no access to a sync var is allowed to be performed until all previous writes have completed, no data access is allowed to be performed until all previous accesses to sync vars have been performed

**Problems:** Inefficiency: syncing happens at begin and end of CS, system must make sure locally-initiated writes complete, remote writes obtained

**Eager Release Consistency:** Separate synchronization into 2 stages: acquire access (obtain all valid pages), release access (send invalidations for shared pages modified locally to remote node copies)

**Lazy Release:** Release requires ACKs, delay by sending invalidation to directory. On acquire, check with directory if new copy is needed.

**Diff at Release:** Changes are encoded into diff, twin discarded, page marked invalid, on next access, diffs are exchanged and applied

**Time, Clocks and Event Ordering**

**Physical Clocks:** 2 associated registers "counter" and "holding"

Counting: decremented by one on each oscillation, when zero, interrupt is generated (tick), adds 1 to time stored in memory, counter reloaded from "holding"

**Cristian's Algorithm:**

1. Client P request time from server S
2. S responds with time T from its own clock
3. P sets its time to be T + RTT/2

- Assume propagation delay same for send/receive - Accuracy improves by making multiple requests and using the minimum RTT

**Berkeley Algorithm:**

1. Master chosen by election
2. Master polls slaves who reply with their time
3. Master uses RTT of messages to estimate time of each slave
4. Master averages the save and own clock times
5. Master sends out the amount (+ or -) to each slave to adjust its clock

**Network Time Protocol:** Time servers form a tree, layers in tree called strata, Stratum 0 are reliable time source, other servers connected via network links UDP. Precision: ms in WAN, microsecond in LAN

**Precision Time Protocol:** Takes advantage of time sources in network switches and NICs, tree hierarchy (uses best master clock to select grandmaster device)

**"Happens Before" relation:** Given two events A and B, A => B (A happens before B) if: 1. A and B are executed at the same process, and A occurs before B 2. A = send(m) and B=receive(m) for some message m 3. There is an event C s.t. A => C and C => B

**Lamport Clocks:** Assign logical timestamp T: if A => B then T(A) => T(B)

1. The i-th process keeps counter T\_i, initially 0
2. When i-th process performs computation event, T\_i <- T\_i + 1
3. When i-th process send msg m, it computes T\_i <- T\_i + 1 and T(m) <- T\_i to m
4. When i-th process receives msg m, T\_i <- max(T\_i, T(M)) + 1

- For event A at i-th process, define T(A) = T\_i computed during A

**Problems:** Relation only goes one way, lamport clocks do capture causal dependencies, but may imply more dependencies than truly exists. That is: if A => B, then LC(A) => LC(B) !=> LC(A) => LC(B), then A => B

**Vector Clocks:** A=>B iff T(A) => T(B)

1. i-th process keeps vector T\_i with n elements
2. When i-th process performs any event, T\_i[i] = T\_i[i] + 1
3. Sends m, Appends vector T(m) <- T\_i to m
4. Receives m, T\_i[j][j] = max(T\_i[j][j], T(m)[j][j]) for each j != i

- For event A at i-th process, define T(A) = T\_i, computed during A.

$$T(A) < T(B) \equiv [\forall j: T(A)[j] \leq T(B)[j] \wedge \exists i: T(A)[i] < T(B)[i]]$$

**Properties of Distributed Algorithms:** Safe and Liveness, timing and failure assumptions affect how we reason about these props

**Timing Model:** Specifies assumptions regarding delays between execution steps of a correct process and send and receipt of a message sent b/w correct processes.

**Synchronization Assumptions:** Known bounds on message and execution delays (communication guaranteed in some # of clock cycles)

**Async Assumptions:** No assumption on message and execution delays (except they are finite) or process running slowly/failed

**Partial Synchrony Assumption:** Processes have some info about time, clocks are synced within some bound, approx bound on message-deliver time, use of timeouts

**Distributed Agreement**

**Failure Model:** A process behaves according to its I/O specification throughout its execution is called correct, deviating from spec is faulty. Either by communication link losing, duplicating, reordering message or process may die and be restarted (looking at Fail-Stop/Byzantine)

**Fail-Stop Failures:** Failure results in process P stopping (crash failure), P does not send any more messages, perform actions when messages are sent to it, other processes can detect P has failed.

**Heartbeat protocols:** Assumes partially sync. env. If process i does not hear from process j in some time  $T = T_{delivery} + T_{heartbeat}$ , then it determines that j has failed. Depends on T\_delivery being known/accur.

**Byzantine Failure:** Process P fails in an arbitrary manner, P is modeled as a malevolent entity (can send messages and perform actions that will have the worst impact on other processes) *Constraints on assumptions:* Incomplete knowledge of global state, limited ability to coordinate with other Byzantine processes, restricted to polynomial computation.

**Agreement Problems:** Processes in a distributed system reach agreement on a value. System model is critical to how to solve agreement problem - failure assumptions/timing assumptions

**Distributed Consensus:** N processes have to agree on single value, up to f < N processes may be faulty. Properties:

- *Agreement:* If any correct process believes that V is the consensus value, then all correct processes believe V is the consensus value
- *Validity:* If V is the consensus value, then some process proposed V
- *Termination:* Each process decides some value V

**Flood Set Consensus:** Sends initial value to each process f+1 times, collects f+1 vector and applies function (min, max, majority, etc). Assumes nodes are connected and links do not fail. Requires f+1 rounds because process can fail at any time.

**Byzantine Consensus:** All loyal generals decide on the same plan of action (obtain the same information) and a small number of traitors cannot cause the loyal generals to adopt a bad plan. Majority vote

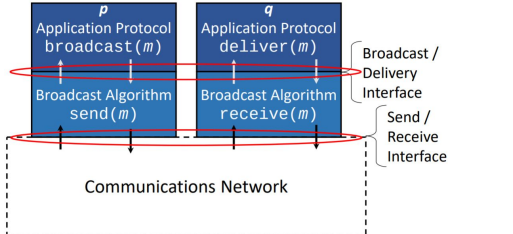
**Theorem:** There is no algorithm to solve consensus if only oral messages are used unless more than 2/3 of the generals are loyal

**Consensus:**  $n > 3f$  (n process, f faulty), Oral Message algorithm OM(f)

**OM(f) Recursive Algo:** f+1 rounds:


1. OM(f): commander sends his value to every lieutenant
2. For each lieutenant i, let v\_i be the i-th received msg, or some default action if no value was received. OM(f-1) sends v\_i to each of the n-2 other lieutenants
3. For each i, and each j != i, let v\_j be be msg from step 2, or some default action if no value was rec., Lieutenant i uses value of majority(v\_1, v\_2, ..., v\_n-1).

4. Continue until OM(0).  
Reducing Overhead: Faulty processes might change messages it received meaning low f tolerance. Each process signs messages can handle f faults with 2f+1 processes  
**Async Distributed Consensus:** Fail-stop or Byzantine failures are impossible to solve, async assumption makes it impossible to differentiate b/w failed and slow processes. Termination (liveness) not guaranteed, and it may violate agreement (safety)  
**Fault Tolerance, Group Communication, Replicated State Machines**  
**Fault Tolerance:** Ability of a system to continue operating in presence of faults (closely related to requirements on dependable systems)  
Masking/Hiding faults requires redundancy (information, time, physical)  
**Recovering:** Must detect/recover/tolerate f faults, covering before f+1 occurs. Requires restoring state of restarted process, *checkpointing*: save state to stable storage; *Replicated state machines*: rebuild state from other group members  
**Replicated State Machines:** Implement a service as a state machine (state variables, commands), replicate the sm on different servers (fault-tolerance/availability/reliability)  
**Commands:** Executed atomically (linearizable), modify state variables, produce outputs. Determined by initial state, sequence of commands  
- RSMs are t-tolerant, fail-stop: +1 replicas required, byzantine: 2t + 1 (one client must decide result, replicas don't have to agree)

**Group communication:**  
  
Properties of send/receive: (validity=liveness, integrity=safety)  
- Validity: both p,q and link are correct then q eventually receives m  
- Uniform Integrity: q receives at most one message from p iff p→q  
Properties of broadcast/deliver (validity, agreement = liveness, etc)  
- Validity: If correct process broadcasts, all correct processes deliver m  
- Agreement: If correct process delivers, all correct processes deliver m  
- Integrity: For any message m, every correct process delivers m at most once and only if m was previously broadcast by sender(m)  
**Message Order:** Causal Order: If broadcast of m causally precedes m', no correct process delivers m' unless previously delivered m.  
**Total Order:** All correct processes deliver messages in same order.  
Total Order does not imply either FIFO or causal, just same order for everyone. May be combined with any of the above delivery constraints.

**ReliableBroadcast(m):**  
tag m with sender(m), sequence\_number(m)  
send(m) to all neighbours including p  
// Event loop for receiving events  
upon receive(m) do  
if p has not executed ReliableBroadcast(m)  
then  
if sender(m) != p  
then  
send(m) to all neighbours  
**ReliableBroadcast(m)**  
- Works in synchronous or async systems - Assumes network does not partition - Failures assumed to be fail-stop - Floods network  
**FIFO Broadcast Algo:** Layered on top of ReliableBroadcast, keeps track of next sequence number of each process. Buffers until sequence number indicates messages maybe FIFO delivered  
**Causal Broadcast:** Build on top of FIFO, prepends list of message that m causally depends. Buffers FIFOdelivered message until all messages upon which m depends have CausalDelivered  
**Atomic Broadcast:** Form of distributed consensus, happens if all messages should be seen in same order.  
**Voting:** V - num votes, W - num writes, R - num reads  
Overlap constraint: V < R + W (recommended V < 2 \* W) - at least 1 voter has latest version to perform read/write  
- Data must contain version number/time stamp  
**Zookeeper**  
Hierarchical namespace (like file system, items in tree called znodes)  
Each znode has data, possibly children. Data always fully read or written  
Replicated in-memory database + persistent log, linearizable updates  
**Create(path, data, flags):** Ephemeral: znode deleted when deleted or creator fails. Sequence: append a monotonically increasing counter  
**Change Notifications:** Don't want service to manage client caches, clients can request notification changes instead.  
**Ordering Guarantee:** Updates are linearizable, each clients' requests are handled in FIFO order

**ZAB (Atomic Broadcast):** Uses FIFO message channels provided by TCP to build reliable delivery, total ordering, or/and causal ordering  
**ZAB Phases:** 1. Leader activation (leader establishes correct state of system, starts making proposal) - leader election 2. Active messaging

**Active Messaging:** Leader sends proposal to all followers using same order, followers process message in the order, leader will issue COMMIT to all followers when quorum of followers have ACK-ed. COMMITs are processed in order  
**Reliable, High Performance Storage**  
**Hard Disk:** Provide large-scale non-volatile storage, access time determined by seek time, rotational latency, transfer time.  
**Unix FS:** Uses logical block numbers (LBNs) that maps to low-level sector addresses on disk. Poor bandwidth utilization (many seeks), high fragmentation due to aging problem.  
**Cylinder Groups:** Addresses placement problems, data blocks in same file allocated in same cylinder group. Files in directory as well. Inodes allocated in same cylinder group as file data blocks. Superblock replicated to improve reliability  


**Metadata Update Problem:** Interdependencies must be handled carefully during disk updates (so FS is consistent after crash)  
**Check-and-repair (fsck):** Finds inconsistencies in ext FS and repairs them. Limitations: only FS integrity, correct repair is difficult, very slow  
**Journaling:** Before overwriting structures, log operations to be performed somewhere else on disk. If crash occurs during write, replay journal (good recovery, fast). If crash happens during journal write, doesn't matter since actual write hasn't happened, so not inconsistent.  
- Normal ops are slower (doubles volume of writes to storage)  
- May break sequential writing (jumping b/w journal and main region)  
**Redo Logging:** After reboot, scan journal and look for committed transaction, replay, after replay the FS is guaranteed to be consistent  
**Design Alts:** Where should journal be kept? File on FS/Special set of blocks in FS/separate device What should be written? Metadata/Data/Logical(record changes) or physical(whole block)/Log old value or Log new value Granularity of flushing updates? Single update/FS op/Group op How is journal space reclaimed? Checkpointing  
**Journal Entry Note:**  

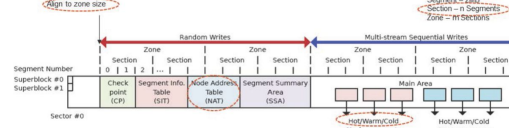
TxBegin (TID=1)	Updated inode	Updated Bitmap	Updated Data block	TxEnd (TID=1)
-----------------	---------------	----------------	--------------------	---------------

**Identify Complete Transaction:** Write each block synchronously (slow, unsafe since TxEnd might not exist, internal disk schedule might reorder ops to happen after TxEnd). Split transaction commit to 2 steps.  
**Metadata Journaling:** only write FS metadata to journal  

TxBegin (TID=1)	Updated inode	Updated Bitmap	TxEnd (TID=1)
-----------------	---------------	----------------	---------------

  
Write data to final location before writing metadata to journal. If failure while writing data, overwrites persist. If failure while journaling metadata, same outcome, as if no new allocations occurred.  
**Notes:** Recover complexity is in the order of size of journal not disk.  
**ext3:** Journal is ordinary (large) file, separates journaling from FS concerns, physical redo journaling, mount option to choose journal mode  
**Writeback:** Data blocks are not journaled and are not ordered with respect to (journaled) metadata blocks. File can contain garbage (or worse) after crash  
**Ordered:** Data blocks must be flushed to disk before metadata blocks are committed to journal. Challenging since blocks change type  
**Stale metadata:** Create directory, delete directory, file appended reusing bitmap block & inode block, crash. Replay overwrites datablock appended to file with old directory entry content. Sol: Make certain delete operations synchronous, don't reuse metadata blocks until checkpointed out of journal. Ext3: Delete op adds "revoke record" for freed directory block (recovery scans for revokes first and doesn't replay)

**Log-Structured File System**  
**Soft Updates:** Delayed metadata writes, protect consistency of disks by controlling order of writes (write-back caching, ensure propagate to disk)  
**Update Ordering Rules:** Ensure integrity of metadata pointers. Rules:  
1. Resource Allocation: Initialize resource before setting pointers  
2. Resource Deallocation: Nullify previous pointer before reuse  
3. Resource "Movement": Set new pointer before nullifying old one  
**Soft updates crash recovery:** Need to deal with dangling pointers, soft updates requires no post-crash recovery. Only inconsistencies are "leaked" blocks, can find free blocks in background while using FS  
**Issues:** Complex - each FS metadata structure uses subset of different data structs with different relationships. Anomalies with rm of large tree  
Space Overhead - dependence info + old/new versions of each change  
Computation & disk traffic overhead - may write single block multiple times to commit all updates  
**Log-structure FS:** Since memory is increasing, we care less about reads since most will be cached in memory. Assume write pose bigger I/O penalty. Pros: Write throughput improves, crash recovery is simpler Cons: Initial assumption may not hold, read might be slow  
**LFS:** Response to scaling disk bandwidth, and large main memories in machines. Takes advantage of both to increase FS performance  
Treat disk a single log, keep appending data and metadata updates. Collect write in disk cache, write out entire collection in one large disk request. All Info written to disk is appended to log.  
**Challenges:** Locating data written to log & Managing free space on disk  
1. Needs inode map (imap) to find inodes, imap located on fixed part of disk but is updated frequently (more seeks, poorer performance). Instead place imap next to new information have a checkpoint region (CR) point to latest imap. Contains addresses of imap blocks, segment usage table (for cleaning), address of last segment written, timestamp  
2. LFS append-only runs out of disk space. Fragment log into segments, write segment-at-once, chain active segments on disk (non-contiguous), reclaim space by cleaning segments. Reclaim: read segment, copy live data to end of log, free segment for reuse. High cleaning overhead!  
**Crash Recovery:** Most recent changes at end of log, borrows 2 database techniques: checkpoints, roll-forward

**Checkpoints:** Keep 2 CR to protect against crash while writing checkpoint. Write header (with timestamp), CR body, then one last block (also with timestamp). On recovery, use region most recent timestamp.  
**Roll-Forward:** recover data since last checkpoint, examine segments identified by CR, seg. contain special summary block and directory operation logs (changes to FS metadata). Operation log guarantee appear before modified directory or inode blocks. If directory or inode is inconsistent, operation log entry can be used to correct it.  
**SSDs**  
**Issues:** Must erase entire blocks to rewrite pages, limited endurance  
**Goals:** 1. Translate read/write to logical blocks into reads/erases/programs on physical pages + blocks (allows SSDs to export "block interface" and hides write-induced copying + garbage collection)  
2. Reduce write amplification (extra copy to deal with block-level erase)  
3. Implement wear-leveling (distribute writes equally to blocks)  
**Flash Translation Layer (FTL): Log-based mapping:** treat physical blocks like a log, send data in each page-to-write to end of log. Maintain mapping between logical pages and corresponding physical pages  
**Garbage Collection:** Expose logical page space that is smaller than physical page space, keep extra hidden pages around, defer GC to a background task. FTL occasionally shuffle live blocks that never get overwritten (enforces wear leveling)  
**F2FS Flash-friendly FS:** Multi-head logging, separates data+metadata into segments, takes advantage of parallel SSD ops, adaptive logging  
**Flash-friendly On-disk Layout:** Log-structured, with some update-in-place regions (take advantage of FTL). Section size is power-of-two segments, zones correspond to sets in FTL set-associative mapping  


**Wandering Tree Issue:** Each write to block changes its location, requires update to block that points to it. **Node Address Table (NAT)** stores location of block, other index structs store offset in nat. Cuts off propagation at one level.  
**Security**  
**Morris Worm:** Exploited buffer overflow in finger daemon, vuln in Unix sendmail daemon when it was running in debug mode (executing arbitrary commands), and password cracking dictionary.  
**Cryptolocker:** Trojan via email attachments (.pdf.exe file on windows), encrypts data files on infected computer using strong RSA, sends ransom note, attempts to delete Windows Shadow Copy backups  
**Heartbleed:** Vuln in OpenSSL (improper input validation), leaked memory contents between client and server (private keys).  
pl = 64k, data = "", i.e payload size != data size  
buffer = OPENSSL\_malloc(1+2+payload+pad);  
bp = buffer;  
memcpy(bp, pl, payload);  
r = dtls1\_write\_bytes(s, TLS1\_RT\_HEARTBEAT, buffer, 3 + payload + padding);

**Meltdown:** Load from protected address into pointer, use value to index an array. Time reads from array to detect which entry is cached → index of fast entry is value at protected memory. Speculative, out-of-order execution will fetch prevents exception raise until instruction reaches end of processor pipeline (retirement). Result will be cached.  
**Spectre:** Run attacker code on same core as victim code, train branch predictors to drive speculation down a particular branch in victim, apply timing analysis to extract victim data  
**OS Scalability & Multiprocessor Scheduling**  
**Areas of Concern**  
- Statistical Counters: widely tracked, frequently updated, rarely read  
- Processor Scheduling - Memory management  
**Per-CPU shared data:** Each CPU keeps count, can lead to false sharing due to several per-CPU variables being on the same cache line, modifying 1 invalidates other CPU caches. Pad array per-CPU counters  
**Centralized Queue:** Each CPU has local timer interrupt, current executing thread blocks or yields, event is handled that unblocks threads  
**Adv:** Simple scheduler implementation due to single ready queue, regardless of how many CPUs → best set of threads are picked to run  
**Disadv:** Single shared ready queue does not scale  
**Alternative Ready Queue:** Per CPU ready queue  
**Advantages:** Scalability, reduces contention, implement different policies for different queues  
**Disadvantages:** Load balancing  
**Load Balancing:** CPU should not idle while other CPUs are running, scheduling overhead may scale with size of run queue  
**Push Model:** Kernel daemon checks queue lengths periodically, moves threads to balance  
**Pull Model:** CPU notices its queue less than a threshold, and steals threads from other queues  
**Processor Affinity:** Attempt to keep warm cache on CPU (try to keep thread on same CPU it used last)  
**Symbiotic Scheduling:** Scheduling threads that can share nicely on same CPU (prefer threads with small cache footprint on the same CPU, large footprints share the same CPU due to lack of benefit anyway)  
**Completely Fair Scheduler (CFS):** One run queue where threads are globally sorted by runtime. When thread is done running for its timeslice, it is enqueued again (timeslice can have a niceness)  
**Load = weight x % CPU**  
**Linux Scheduler:** Per-CPU run queues, CPUs are organized into scheduling domains (load is kept balanced by kernel, each domain contains a set of groups). On each tick, recomputes local load statistics, checks if time to invoke load\_balance() for each domain from base to top level.  
**Linux Load Balancing:** Finds busiest group in current domain, finds busiest queue (CPU) in that group, invokes move\_tasks to move threads. (move\_tasks attempts to preserve affinity, can't be currently executing; target CPU must be allowable for task, idle, or not cache hot)