Page replacement algorithm:

- Delay next page fault for as long as possible
- Implies optimal algo exists → but we would need to know the future
- Do our best to approximate using algos like clock algos
- Poor replacement policy → constantly evicts pages needed in future
- Constantly swapping pages from disk → thrashing → performance hindrance

Most OS have a high watermark (HW) and a low watermark (LW). When there are fewer than LW pages, a background thread called the swap daemon is responsible for evicting pages until there are HW pages available. Grouping page swaps allows for more efficient use of disk

LRU

- Each time a page is accessed, record the time
- When you need to eject a page, look at all timestamps for pages in memory
- w/ 64GB and 4K pages, 16MB of timestamps—expensive
- when program loops over (n+1) pages w/ TLB size n, LRU misses upon every access (random is better in this case)

Clock algorithm: a surrogate for LRU

- Organize all pages in a circular list
- MMU sets a reference bit for the page on access
- Scan whenever we need another page
  - For each page, ask MMU if page has been referenced. If so, reset the reference bit in the MMU & skip this page. If not, consider this page to be the least recently used. Next search starts from this position, not head of list → Use position in the scan as a surrogate for age
- No extra page faults, usually scan only a few pages

FIFO and random are terrible

- LFU: need to update every time a page is accessed → overhead. Also, not a perfect predictor

Shared libraries: multiple processes accessing the shared global data at the same time would be a problem.

Flow control on IPC mechanisms like sockets, but not shared memory

- Sockets: process must read/write to them → requires flow control
  - Processes ask OS to perform read/writes, and OS can block processes so they can wait and send interrupt when data is ready.
  - Kernel can control running time of reader and writer—by doing this, we can match the speed of the writer and reader. OS transfers the bytes for the processes
  - Protocol stacks may be many layers deep, and data may be processed and copied many times. Limited throughput, high latency
- Shared memory: process have mapped shared file into VA spaces and can modify/read it any time. OS has no part in blocking processes on reads or polling for writes; up to processes
  - Fastest and most efficient way to share info between processes.
  - Only be used between processes on same memory bus
  - A bug in one process can easily destroy communications data structures. No authentication

API vs ABI

- API - Application Programming Interface. Specifies how a library can be used. Interface between different programs.
  - Source level interface such as functions, macros, data types
  - API compliant program must be recompiled for each different architecture
- ABI - Application Binary Interface, binary interface. Specifies layout and format of the module. Contract between OS & apps.
  - Specific conventions on linkage, data formats, dynamic libraries
  - ABI compliant program will run unmodified on a system with that ABI
  - Syscalls are a subset of ABI
- An ABI is an interface that binds a program's source code to a specific instruction set architecture (ISA). ISA define conventions in machine code and are device-dependent.
- Devices w/ different ISAs will probably not be to run the same binary for the program.
- We don't want to distribute the source code and have people compile it for their ISAs for themselves, so if we know the ABI ahead of time we can compile different binaries and distribute those so customers can just download the program and run it.

Segmentation

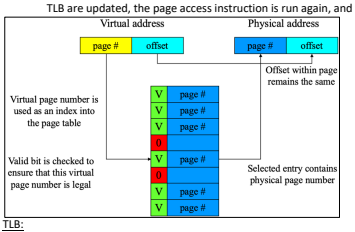
- Program divided to variable-sized chunks (code and data at beginning of VA space)
- Each segment needs to be in a contiguous PA (get rid of requirement w/ paging)
- Add value of offset into segment
- VM allows us to have VA in a process that translates to a PA, which can be computed based on the base address of a segment, based in a special hardware register. When relocating the memory, all we have to do is update the base address. All VM addresses in the process remain valid
- Solves problem of a relocation

4K to 1K page size

- Paging: Allow the virtual memory management system to load, relocate, and otherwise manage the space more flexibly
- Average internal fragmentation: half a page
- Makes external fragmentation non-existent since pages are all the same size.
- If running processes that don't require much memory (e.g. 100bytes), internal fragmentation decreases. Also, if a process needs 5KB of memory, then with 4KB page size, there is 3KB of waste, while no waste in 1K page size
- Decreased performance: many more page accesses needed, increasing chance of page fault. Also more page swaps

VM system with segmentation and demand paging:

1. TLB miss, PTE present & valid: Address is valid, page is in RAM page frame, MMU caches table entry → fast access
2. Address is valid, MMU already cached address translation (VPN + PTE) in the TLB. OS doesn't need to consult page table in RAM and just does the translation → even faster access
3. Address is invalid (page is unallocated/process doesn't have right permission) → OS kills process (segfault)
4. Address is valid but the page is not currently in RAM. (Page fault) If RAM is full, OS picks page to evict, then makes an I/O request to disk for the page to be swapped in. Process may block while waiting. When we get the page, the page table and TLB are updated, the page access instruction is run again, and we have access.



If we have 4K pages, and a 64 Gbyte memory, how many pages are there? $2^{36}/2^{12} = 2^{24}$
64 bit PA—2^64 bytes addressable RAM

VPN = TLBT followed by TLBI (TLBI = items in set)
PPO = VPO = 12 bits for 4KB page (2^12 = 4K)

TLB:

- Different processes have different address space
- They should not be allowed to read other processes' pages → hard modularity
- Solutions: 1. Flush the TLB or 2. ASID bit or similar indicators that tells ownership
  - ASID: won't have to waste time on TLB misses after a context switch

Paging/swapping

- Paging: individual pages are moved.
- Swapping: entire process is swapped back to disk. Usually done due to CPU scheduling
  - When a process yields, copy its memory to disk. When it is scheduled, copy it back
  - Each process could see a memory space as big as the total amount of RAM
  - If we actually move everything out, the costs of a context switch are very high: need to copy all of RAM out to disk, then copy other data from disk to RAM before the newly scheduled process can run

On-demand paging

- Kernel doesn't have to load all pages into physical memory
- Entire process needn't be in memory to start running
  - Start each process with a subset of its pages
  - Load additional pages as program demands them
- Demand paging will perform poorly if most memory references require disk access
- Avoid by ensuring that the page holding the next memory reference is already there
- Pre-fetching: get next page when a page is accessed. should be done when there's a reasonable chance of success
- When taking a page fault, read many pages instead of just one (good if program displays locality reference)

Page fault

- Mark the virtual pages not present in physical memory (**present bit** in Page Table Entry)
- Hardware finds out that the page is not present in physical memory, generates interrupt
- Traps to kernel, and control is transferred to **page fault handler**
  - Page fault doesn't have to invoke page fault handler, e.g. could be handled by hardware
- Checks permission and determines which pages are needed
- Schedule I/O to fetch it, then block the process. Load from disk (file or swap)
- Update Page Table Entries with allocated physical page frames and setting appropriate permission bits
- Returns to user mode, retry the execution

Dirty bits

- Dirty bit marks whether page has been written to or not. When pages need to be swapped out, an algorithm might look for clean pages as they have been unchanged and do not need to be written back to disk, saving a memory access, decreasing disk I/O, leading to a performance increase.
- Do background write-out of dirty, non-running pages (turning dirty pages into clean pages). Further reduces I/O involved in their eviction.

Working sets

- The pages we need: if we increase the number of page frames allocated to that process, it makes very little difference in the performance; if we reduce the number of page frames allocated to that process, the performance suffers noticeably.
- Age decisions are not made on the basis of clock time, but accumulated CPU time in the owning process. Pages only age when their owner runs without referencing them.
- If we find a page that has been referenced since the last scan, we assume it was just referenced.
- If a page is younger than the target age, we do not want to replace it, since recycling of young pages may lead to thrashing.
- If a page is older than the target age, we take it away from its current owner, and give it to a new (needy) process.
- If there are no pages older than the target age, we apparently have too many processes to fit in the available memory: we can replace the oldest page in memory. This works, but at the cost of the very expensive complete scan that the clock algorithm was supposed to avoid. If we think that our target age was well-chosen (to avoid thrashing) we need to reduce the number of processes in memory.

Page-stealing

- Every process is continuously losing pages that it has not recently referenced.
- Every process is continuously stealing pages from other processes.
- Processes that reference more pages more often, will accumulate larger working sets.
- Processes that reference fewer pages less often will find their working sets reduced.
- When programs change their behavior, their allocated working sets adjust promptly and automatically.

Dup:

- open the new input/output file
- closing the file descriptor (0, 1, 2) to be replaced
- duplicate the new input/output file to the (newly vacated) file descriptor to be replaced
- close the (now redundant) file descriptor on to which that file was originally opened

Scheduling—goals: maximize throughput, minimize wait time, ensure fairness, cater to other priorities

Metric 1: turnaround time: $turnaround\ time = completion\ time - arrival\ time$

Metric 2: response time: $response\ time = time\ of\ first\ run - arrival\ time$

	Pros	Cons
First In, First Out (FIFO)	- Easy to implement	- Turnaround time: chooses early arrivers over short jobs, so one long job could kill the average turnaround. - Convoy—short jobs get queued behind long jobs
Shortest Job First (SJF)	- Optimal when jobs arrive at the same time	- Jobs *don't* arrive at the same time - Response time: starving for longer jobs
Real-time scheduling	- Soft versus hard real time, schedules normally created beforehand - Good for video rendering, etc.	Not suitable for time-sharing computer systems: jobs may be dropped if deadlines aren't met, which would upset the user
Shortest Time-to-Completion First (STCF)/Preemptive SJF	- Preemption: scheduler performs a context switch, stopping one process and starting/reusing another	- Need good estimate of job time - Response time: starving of longer jobs
Round Robin (RR)	- Runs job for a time slice (multiple of the timer interrupt period) - Response time: the shorter the time slice is, the better	- Turnaround time: RR does not finish short jobs quickly, thus does not optimize ATT. → 1 of the worst for ATT - Time slice too short: overhead of context-switching—tradeoff
Priority Scheduling	- No preemption: If non-preemptive, priority scheduling is just about ordering processes. Much like SJF, but ordered by priority instead	- With preemption: possible starvation - Adjust priorities at runtime
Multi-Level Feedback Queue	- Observes execution of a job and prioritizes it accordingly. - Good overall performance (SJF/STCF) for short-running jobs, is fair, makes progress for long-running, CPU-intensive workloads	

Fixed partition allocation

- Pre-allocate partitions (>=1) for *n* processes
  - Reserving space for largest possible process.
  - Partitions come in one or a few set sizes
- Easy to implement: de/allocation cheap and easy. Common in old batch processing systems. Suited to known job mix.
- Need to know how much memory will be used ahead of time → internal fragmentation. Limits the number of

MLFQ

1. If Priority(A) > Priority(B), A runs
2. If Priority(A) = Priority(B), A&B RR
3. New job placed in highest priority first
4. When a job uses up its allotment, its priority is reduced (moves down 1 queue)
5. After some time period, move all jobs in the system to topmost queue

System call—trap handler

1. System recognizes a syscall, the system traps. Switch from user mode to supervisor mode
2. CPU loads new PC/PS from associated trap vector
3. CPU pushes PC/PS at time of trap onto supervisor mode stack
4. first level handler saves registers and status, forwards to 2nd level handler
5. 2nd level handler determines it was a user-mode segmentation fault and that there is a registered signal handler
6. push PC/PS at time of trap onto user mode stack, and changes the return PC (supervisor mode stack) to sig handler.
7. 2nd level handler returns to 1st level handler, 1st level handler restores registers and returns to user mode
8. user mode execution resumes in the registered signal handler

First level vs second level trap handler

- 1st level saves current process state—CPU registers, PC, processor status onto k-stack (same procedure for every trap)
- 2nd level handler handles the trap itself. Makes it simpler to add a new trap, or change the process-state-saving procedure.

Syscalls vs procedure calls

- System calls are like procedure calls in that they pass parameters, perform a service, and return a result when the operation is complete.
- System calls run with different privileges, in a different address space. Procedure calls are made with direct call instructions, while system calls are a requested with an illegal instruction that is recognized by the trap handler. Consquently, system calls are much more expensive than procedure calls.

Copy-on-write

- Copy-on-write can be implemented efficiently using the page table by marking certain pages of memory as read-only and keeping a count of the number of references to the page.
- When data is written to these pages, the kernel intercepts the write attempt and allocates a new physical page, initialized with the copy-on-write data, although the allocation can be skipped if there is only one reference. The kernel then updates the page table with the new (writable) page, decrements the number of references, and performs the write. The new allocation ensures that a change in the memory of one process is not visible in another's.

Segments found in a process's virtual address space

- text ... executable, read-only
- data ... read/write, can change at run-time. stores global variables, static variables, local static variables
- stack ... read/write, grown/shrunk by OS as needed
- DLLs/shared libs ... executable, read-only other code segments loaded at load- or run-time
- thread stacks ... like program stacks, but are probably allocated in the data segment and may not be managed by OS

Pipes

- Named pipes are device special files in the file system. Processes of different ancestry can share data through a pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.
- Blocking: if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well.
- If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline

Solaris x86

- What would have to be done to permit Windows binaries to be loaded into memory and executed on Solaris x86:
  - Need to support windows binaries by changing the Solaris/x86 to be able to read/execute windows ABI.
  - Need to support new load module format/basic syscalls
- What would have to be done to correctly execute the system calls that the Windows program requested?
  - A new 2nd level trap handler would be written to intercept the Windows system calls, and pass it on to an emulation layer, which would try to simulate the effects of each Window's system call, using Solaris mechanisms.
- How good might the performance of such a system be?
  - Performance should be okay since user-level instructions don't need to be emulated. Only system calls do.
- Besides new load module format and basic syscalls, that the system should be prepared to simulate:
  - Emulate functionality of Windows device drivers (specifically for displays and printers). Might be easier to provide own dynamic link libraries to directly implement the higher-level functionality on a Solaris display server.
- Would not work on Solaris/PowerPC or Solaris/SPARC system because those systems don't have x86 processors & so have different processors than the Windows machines.

Threads

Fork()

- Fork returns 0 in child thread, returns PID of child in parent thread. Results in copy-on-write
- New process has a different PID than parent. Useful because child process may do different things (like exec() a program while parent listens for output). In addition, if one process has multiple children, it can use the PIDs to differentiate between them, which is important for child reaping.
- Thread has its own PC, its own private set of registers, its own stack

- Thread control block (TCB) to store state of each thread of a process. Address space remains the same in context switch between threads, i.e. no need to switch page table
  - allocate memory for a (fixed size) thread-private stack from the heap. create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack. add to ready queue
- sleep/yield: if preemptive scheduling, the costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.
- Pros of multithreading: overlap of I/O (which would cause a process to be a blocked) with other activities within a single program. Speeds up program on system w/ multiple processors
- Use for: parallel activities in a single program, when there is frequent creation and destruction, when all can run with same privileges, when they need to share resources, when they exchange many messages/signals, when you don't need to protect them from each other
- Maximum stack size specified when thread is created: stack space must be reclaimed when thread exits

Exec()
- Kill() syscall is used to send signals for a process
- Exec: loads code and overwrites current code segment (and static data); heap and stack and other parts of the memory space are re-initialized
  - Never returns

Processes vs. Threads (tradeoffs):
- Multiple processes: application may run much more slowly, may be difficult to share some resources
- Threads: have to create and manage them, have to serialize resource use, program will be more complex to write

Critical sections
- A *critical section* is a resource that is shared by multiple threads: by multiple concurrent threads, processes or CPUs, or by interrupted code and interrupt handler
- Use of the resource changes its state: contents, properties, relation to other resources
- Correctness depends on execution order: when scheduler runs/pre-empts which threads, or relative timing of asynchronous/independent events

Atomicity
- Before/after atomicity – no mingling of intermediate steps
- All/nothing atomicity—an update that starts will complete; if an update doesn't complete, there is no effect

Ways to enforce mutual exclusion

Disable interrupts (can only do so at kernel level w/ privileged instruction; coarse-grained)
- Temporarily block some or all interrupts: no interrupts -> nobody preempts program; side-effect of loading new Processor Status Word
- Abilities: prevent time-slice End (timer interrupts), prevent re-entry of device driver code
- Dangers: may delay important interrupts, bug may leave them permanently disabled, won't solve all sync problems on multi-core machines, since they can have parallelism without interrupts
- When to use: situations that involve shared resources + non-atomic updates (pre-emption would lead to data corruption/deadlock) + interrupt disables are absolutely necessary
- Good fairness if disables are brief.
- Performance: much cheaper than syscall, but long disables may impact system performance.

Atomic instructions (compare-and-swap, test-and-set)—hardware mutual exclusion
- Limitations: unusable for complex critical sections, unusable as a waiting mechanism
- Test-and-set: note current value, store new value (1), and return the old value. If the return value is 0 (lock is released) then the lock is acquired.
  - int TestandSet (int *old_ptr, int new)
  - while (__sync_lock_test_and_set(&svar, 1) == 1) //svar an int
    ; *//do nothing*
    //do operation
    __sync_lock_release(&svar);
- Compare-and-swap: if the current value of **ptr* is *oldval*, then write *newval* into **ptr*. Return ptr.
  - int CompareAndSwap (int *ptr, int expected, int new)
  - do { //generate newval } while (__sync_val_compare_and_swap(&pointer,oldval,newval) != oldval)
- Test-and-set and compare-and-swap are spinlocks
  - Pros: properly enforces access to critical sections
    - Use when: awaited operation proceeds in parallel, awaited operation is guaranteed to be soon, spinning does not delay awaited operation, or when contention is expected to be rare
  - Cons: wasteful as spinning uses processor cycles, likely to delay freeing of desired resource b/c wasting cycles, burning memory bandwidth slows I/O. Possible deadlock, possible starvation
  - Yield and spin: spin a few times, then yield if can't get lock. Problems: extra context switches, still wastes cycles if spin each time scheduled, might not get scheduled to check until long after event occurs, works very poorly with multiple waiters (potential unfairness). Possible deadlock, possible starvation

Software locking
- Asynchronous completion: sleep and wake up (condition variables)
  - It blocks a process or thread when condition variable is used, observes when the desired event occurs, then unblocks the blocked process or thread
- Semaphores
  - P – "wait": decrement counter. If count >= 0, return. If counter < 0, add process to waiting queue
  - V – "post": increment counter. If queue non-empty, wake one of the waiting process
  - Initialize semaphore count to one (initial count reflects # threads allowed to hold lock)
  - Cons: lack many practical synchronization features, easy to deadlock, cant check lock without blocking, don't support reader/writer shared access, no way to recover from wedged V operation, no way to deal w priority inheritance
- Mutexes—advisory locks (low overhead)
  - Object level locking: mutex protects code critical sections for brief durations. Other threads operating in a single address space on the same data
- Persistent objects (e.g. files) more difficult: CS likely to last much longer, many different programs can operate on them, may not even be running on same machine. Solution: lock objects
  - Linux file descriptor locking: **int** flock. Applies to open instances of same *fd*, distinct opens unaffected
  - Linux ranged file locking: **int lockf(*fd, cmd, offset, len*)**. Lock applies to file, process-specific. Closing any fd for the file releases for all of a process' fds for that file

Locking performance

Requires syscall. If don't get lock, then block. Blocking expensive, overhead may be more than time spent in CS

$C_{expected} = (C_{block} * P_{conflict}) + (C_{get} * (1 - P_{conflict}))$

Problems:
- Convoy: one process gets the resource and all other processes line up & wait. No parallelism
- Priority inversion: high-priority thread waiting and blocking for lock held by low-priority process, which may be swapped out by the scheduler to run a mid-priority process
  - solution: raise priority level of low-priority process temporarily. Don't pre-empt low priority if blocked. Health-monitoring where if high-priority task didn't run for a long time, reset system
  - priority inheritance: low-priority task inherits high priority when it has the lock, then loses priority when release

Solutions:
- Reducing locking overhead: minor overhead, not much can be done as its usually highly optimized
- Eliminate CS (eliminate shared resource/use atomic ops)
- Reducing contention: reduce CS, reducing frequency of entry, fine-grained locks, spread the requests to multiple resources
  - Fine-grain locking instead of coarse-grain locking, e.g. dividing locks (read locks and write locks)
    - Reader/writer lock: possible starvation if writers must wait for all readers to leave
  - Allocate required memory before taking lock, minimize code in CS. Cons: may complicate code
  - Local variables that are then added all together in the end

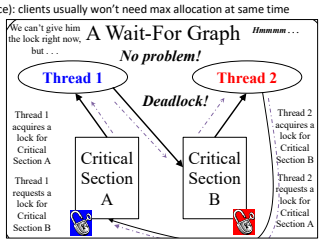
Deadlocks
- Commodity resources: clients need an amount of it (e.g., memory), deadlocks result from over-commitment. Avoidance can be done in resource manager
- General resources: need a specific instance of something (particular file, semaphore, message or request completion)

Four basic conditions:
1. Mutual exclusion. Solution: shareable resources use atomic ops, use private resources if possible
2. Incremental allocation: processes/threads allowed to ask for resources whenever they want, instead of everything at the beginning. Solution: allocate resources in single operation, non-blocking requests (all-or-nothing), disallow blocking while holding resources (release all held locks before blocking & all-or-nothing reacquire after return)
3. No pre-emption: when an entity has reserved resource, can't take it back. Solution: resource leases that expire & lead to resources being seized & reallocated to new client through revoking access by invalidating a process's resource handle. Resources must be designed w/ revocation in mind. Not possible to revoke if object a part of process's address space. Revoking access requires destroying the address space through killing the process.
4. Circular waiting: cycle in graph of resource requests. A wait on B wait on A. Solution: total resource ordering—all requesters allocate resources in same order. Assumes we know how to order resources (e.g. parent before children, etc.). May lead to lock dance: release R2, acquire R1, reacquire R2. In more complex systems, total ordering may be difficult; use partial ordering, where lock acquisition order requirements are grouped/there are general rules.

Reservations:
- Reservations for commodity resources. Only grant reservations if resources available. Client must be prepared to deal w/ failures. Failures happen at reservation time, not reserve time → behavior predictable

- OS grants until everything is reserved
- Beyond available amount (sometimes won't get reserved resource): clients usually won't need max allocation at same time

One more solution: ignore it!
- Deadlocks are improbable, and prevention may be expensive. Instead, let them happen.
- Deadlock detection: maintain wait-for graph/equivalent structure. When lock requested, structure updated & checked for deadlock.
- Health monitoring to reboot when deadlock happens.
  - internal monitoring agent watch message traffic or a transaction log to determine whether or not work is done
  - asking clients to submit failure reports to a central monitoring service when a is unresponsive
  - servers send periodic heart-beat messages to a central health monitoring service.
  - send periodic test requests to the service, see if they are being responded to correctly and in a timely fashion.
  - Kill and restart all affected software. As many as needed & as few as possible. Apps must design for cold (all data discarded, restart @ init state), warm (retentive data maintained, processing restarts @ init state), partial restarts. Highly available systems define restart groups (groups of processes, inter-group dependencies define restar



Monitors: each monitor object has a semaphore, lock acquired for any resource invocation. Very conservative

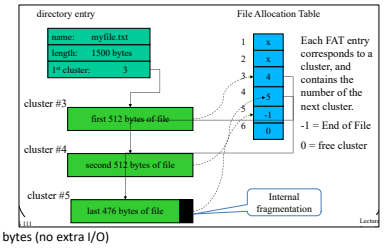
Java synchronized methods: each object has associated mutex, only acquired for specific methods. Granularity. Need to identify serialized methods.

Devices—when a device sits idle, its throughput drops
- Direct memory access—contiguously allocated buffers in PM. Gather data from physical memory pages into buffer before data gets sent out. Scatter: read from device to multiple pages; gather: writing from multiple pages to device. Minimal CPU overhead. Bigger transfers = better (amortize seek/rotation overhead). Data transfer at bus/device/memory speed
- Memory-mapped I/O—treat registers/memory in device as part of regular address space. Device accepts data at memory speed. Every byte transferred by CPU instruction, so no per-op overhead but considerable per-transfer overhead. Lower overhead per update, no interrupts to service. Relatively easy to program
- DMA better for occasional large transfers, MMIO better for frequent small transfers.
- Example of DMA: sound card may need to access data stored in RAM, but since it can process the data itself, it may use DMA to bypass the CPU.
- Example of MMIO: bit-mapped display adaptor: each word of memory corresponds to one pixel, application uses ordinary stores to update display.

Filesystem—store data and metadata. SSDs 50-70x faster than hard drive, no penalty for random access.
- Goals: persistence, ease of use, flexibility (no limit of file size/type, # of files), portability, performance, reliability, security

Virtual File Systems
- File is opened—in-memory structure is created. Device version points to device blocks, in-memory version points to RAM
- Boot block—0th block of device (reserved). FS start work at block 1

Linked extents: DOS FAT directory entries
- Space divided into clusters (1-N), file control structure points to first cluster of a file
- File allocation table, one entry per cluster. Contains number of the next cluster in file (0 means unallocated, -1 means EOF)
- FAT is kept in memory: no disk I/O
- No support for sparse files
- Width of FAT describes FS size: 32 bits now



File system blocks: Unix System V
- Linux: 10 direct pointers, 3 indirect (, double, triple). Support for very large files
- 10 direct usually enough as most files a few KB.
- Assuming 4k bytes per block and 4 bytes per pointer
  - 10 direct blocks = 10 * 4K bytes = 40K bytes (no extra I/O)
  - Indirect block = 1K * 4K = 4M bytes
  - Double indirect = 1K * 4M = 4G bytes
  - Triple indirect = 1K * 4G = 4T bytes

- 1-3 extra I/O operations per thousand pages–any block can be found with 3 or fewer reads
- Index blocks can support sparse files

EXT2: 15 (12 direct + 3 indirect)

Free space and allocation
- UNIX: search superblock free inode list, take first free inode. DOS: search parent directory for unused dirent
- Extending a file—find free space, associate w/ appropriate address in file. Deleting a file—for UNIX return each free block to free block list, for DOS garbage collection searches deallocated blocks & add to free list at later time. Deallocate file control block—UNIX, zero inode and return to free list, for DOS zero the first byte of the name in parent directory

FS structure
- Directories are a special type of file: used by OS to map file names into the associated files
- Directory tree organizes all files and directories into a tree, starting w/ root. To access a file, process uses syscall to request permission from the OS, returning a fd. Each fd private & per-process, & an entry in open file table. Tracks offset too.
- DOS: file "absolute name" is path. UNIX: actual file descriptors are inodes.
- Association of name w/ inode is a hard link. Inode keeps refcount. Symblink: do not reference inode, just indirect reference
- Unix dirent: name (relative to directory) + pointer to inode
- Files can have different names if they have different paths (namespaces)

Guarantee crash consistency
- I/O required for adding new file: write to disk for inode, bitmap, and data block. May crash during write of any of the 3.

Journaling
- Journal is a circular buffer that can be recycled after old ops completed. After system restart, review journal to see which entries have not completed writing, and replay operations in order. Truncate journal and resume normal operation. Fast because all writes are sequential, it's write-only, journal very small compared to disk

Log-structure file system: redirect-on-write
- The journal is the file system: all inodes and data updates written to the log, in-memory index caches inode locations
- Redirect-on-write: 1 disk I/O. don't make new copy, directly write a new version into file system & update pointer to file
  - Once written, blocks and inodes are immutable. Clones and snapshots are almost free
  - Write buffering: when sufficient updates are received, write them all to disk at once—efficient use of disk
  - $Time\ taken\ to\ write = rotation\&seek\ time + (bytes/peak\ transfer\ rate)$
- Recovery: most recent snapshot (checkpoint region) will be read into memory & all updates after that snapshot will be scanned and replayed for any new inode mappings that need to be indexed.
- Issues: recovery time (to reconstruct index/cache), log defragmentation and garbage collection → compaction by reading partially-used segments, determine which ones are live, and write out a new segment, freeing up old segment for writing.
- Key-value stores: divide filesystem based on hash value of keys; file referred to by hash value of key in the inode.
- Ring buffer log FS: Garbage collection: reference-counting for each block

Copy-on-write: 3 disk I/O's—read data block, copy data block, and write to a new location, where it is modified. Update pointer to file. Need to keep track of old versions to allow for replays in case of a crash.

Security
- Access control list: for each protected object, maintain list. Capabilities: each user keeps set of items they can access. Applied to much broader context, e.g. get capability from server, go ahead and access resources
- Symmetric cryptography: key same. Fast, encryption and authentication done at once, no centralized authority required.
- Asymmetric: public key crypto. To send a message, encrypt w receiver's public key. Receiver decrypts with his private key. To sign a message, encrypt with sender's private key. Receiver verifies by checking w sender's public key
  - Don't know if key corresponds to right person—needs a trusted key distribution infrastructure/certificates
- Checksum: before sending a message from one machine to another, compute a checksum over the bytes of the message. Then send both message and checksum. At destination, the receiver computes a checksum over the message as well; if they match, then the data likely did not get corrupted

Man-in-the-middle attacks: use SSL (certificate-based authentication of server)

Distributed systems
- Goals: scalability and performance, reliability and availability, ease of use
- Excellent scalability: 100x servers = 100x performance. Limited by management and not hardware
- Good for embarrassingly parallel jobs/mapreduce. Mapreduce: divide-and-conquer. Divide data into pieces, perform function on each piece on a separate node (map), combine results to obtain output (reduce)
- Cryptographic hashes: strong checksums. Both parties need the hash to decrypt it, so need to transmit hash securely