OS

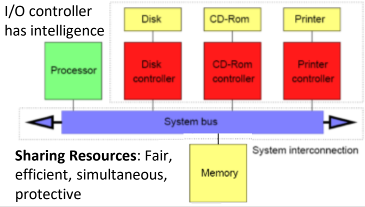
Resources: OS has to make efficient use of resources and share them to multiple users. *Processors*: CPU cores, multi-socket CPU, SMT/hyper-threading, *Memory*: caches, RAM, *I/O devices*: displays, GPUs, NICs, printers, *Internal Devices*: clocks, timers, interrupt controllers, *Persistent Storage*: disks, SSDs, DVDs.

Concurrency: Single core = time sharing, Multi core = true, both non-deterministic. **Persistence**: File system, access controls, failure protection, organise devices. **Portability**: OS written in HLL (C) then compiled down as specific interactions. Platform specific = re-impl'd, platform independent = translate down directly. **Kernel Mode**: r/w access to all memory, full access to all instrs/hardware. **User Mode**: use kernel interface, limited access, isolated. **Splitting**: **A** = protection (kernel can cause damage), reliability (u-mode restarted without entire sys), security (isolation restricts malice). **D** = Performance (interrupt overhead when switch k/u), abstraction (restricted to kernel interface)

Monolithic Kernel: single executable with 1 addr space (Linux, most pop). **A** = performance (fast comms), easy to write (no abstractions). **D** = Complexity (no separation), correctness (hard to prove), robustness. **Hybrid Kernel**: kernel small and basic, all else user mode. **A** = complexity (simple + small), correctness (ez prove), robustness (servers fail w/o entire sys too). **D** = IPC overhead (lots of inter-process comms). **Microkernel**: Combination. **A** = more separation (micro) w lower IPC reqs, **D** = performance (user-level servers lower performance). **Linux**: Monolithic, interr handlers interact w devices, static in-kernel components + dyn loadable modules, portability, services+devices exposed as files. **Windows**: NT is hybrid, DLL impl OS services modularly, ker layer synch, device drivers dyn loaded

Computer Architecture Overview:

I/O controller has intelligence



Sharing Resources: Fair, efficient, simultaneous, protective

Clean Interfaces: OS abstracts hardware complexity/variability. **API** – for given OS there is spec of syscalls for interactions. **POSIX**: unix-like interface, covers wide variety of OS interactions e.g. synch so devs can support multiple OSs.

OS Zoo: **Multiprocessor** = Windows, MacOS, Linux, **Server** = optimise throughput, Solaris, Linux, **Mainframe** = bespoke, limited workload, OS/390, **Embedded** = trusted software, low power hware (washing machine), QNX, **Mobile Devices** = power efficient, iOS, **Real-Time** = precision and time constraints, FreeRTOS

Concurrency: Higher utilisation, kernel shares CPU by switching processes, needs fairness. **Time switching**: illusion of parallel on single core. CPU time split in quanta (time slices), at each scheduler chooses which process gets quanta.

Context Switch: When CPU switches process (context = mem map, process state). **D** = overhead (process state saved + restored), caches (some invalidated, leading to misses)

PCB: Process Control Block, data struct storing process id, pointer to it's mem map, saved state, allocated resources. Linux: task_struct. **Process end**: normal, abnormal (error/exception), aborted (p kill c), never (daemon providing service) pstree –p = show process hierarchy

Signals: IPC (inter-process comm) similar to hardware interrupts. When get signal, call appropriate signal handler. All sigs except SIGKILL/SIGSTOP can be handled + ignored (not much info in them). SIGINT = keyboard interrupt, SIGABRT, SIGFPE, SIGKILL, SIGSTOP = suspend, SIGSEGV, SIGPIPE = p closed, SIGALRM = alarm timer, SIGTERM, SIGCONT. **Pipes** (ps): 1-way comms between processes (pcs). Connect stdout with other stdin. Can buffer, if pipe full, write end blocks until read. Temp unnamed ps used to comm by pcs. Named ps stored in filesys (used like file), can outlive creating pcs. *int pipe(int fd[2])* – create pipe. Params: fd = where to put file desc (fd[0] = Read fd[1] = W). Returns: 0/-1. *close(fd[0])*

Unix Processes: *int fork(void)* – creates copy of parent prcs. Returns: -1 = am p+err, 0 = am c, 1 = am p+c's pid. Fail: max thds reached, low memory, p killed. Linux: clone() syscall + COW. *int execve(const char *path, char *const argv[], char *const envp[])* – replaces currently running program with new (new stack/heap). Params: path = pathname of new prog, argv = args to main, envp = env var. Returns: 0/-1. Combined with fork() to make new child prcs to run program (Unix philosophy – basic blocks easily combined). Wrappers: execl, execle, execvp, execv. Windows: CreateProcess. *void exit(int status)* – kills prcs, returns exit code. *int kill(int pid, int sig)* – sends any signal to prcs/p group. Params: pid = >0 → sig to pid, 0 → every prcs in curr prcs's group, -1 → all ths prcs can send to, <-1 → every prcs in p group with id –pid, sig = 0 → no signal, any → signal is int, Returns: 0/-1. *int waitpid(int pid, int *stat, int options)* – suspend exec of calling prcs until terminates normally/signal received. Params: pid = -1 → wait for any child, 0 → wait any c in prcs group, >0 → c in pg =pid, stat = pointer to place info on child, options = e.g. WNOHANG (return if c status not available). Returns: pid → pid of child wated on/terminated (e code at *stat), 0 → WNOHANG set + no terminated children, -1 → err.

Process: Running program, user mode (kernel through interface), some foreground (user interaction), others b-grnd (mail/print server, called *daemons*). Contain state: *addr space* (own mem, heap/stack/static data(e.g. instrs)), *registers*, *OS resources* (e.g. open files). **A** = isolation (only kernel requests affect each other), safety, simplicity (prog-er not worried abt others), (kernel has) control, concurrency. Processes created at system initialisation, user request, or a syscall by running process.

Thread Abstraction: Thread = stream of instrs being executed, state defined by CPU, stack per thd, addr space shared. Processes have >=1 thd. Processes are slow + expensive to create/destroy, and comms are complex (pipes/sig handlers). **A** = shared data (all thds in process share addr space, ez R/W to same mem to comm), cheap switching (onoly change per-thd state – registers, SP), cheap management (basic, cheap). **D** = concurrency bugs, forks (holding locks in children never released), blocking (if 1 thd blocked, all in process blocked). **Pthreads** (POSIX Ts): POSIX interface for managing thds. *int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)* – creates new thd. Params: thread = pointer to store thd at, attr = attributes, start_routine = func to start new thd at, arg → arg to pass start_routine. Returns: 0 → success, EAGAIN → lack of resources, EINVAL → invalid attr, EPERM → caller lacks permissions. *void pthread_exit(void *value_ptr)* – exit thd, provide value_ptr to joins onto thd. Called implicitly at end of start_routine. Process terminate → terminate thds, main thd exit → waits for children/exit call. Remember main has implicit call to exit added by compiler. *int pthread_yield(void)* – yield CPU to another thd. Returns: 0/-1. Always succeeds on Linux. *int pthread_join(pthread_t thread, void **value_ptr)* – block curr thd until other thd terminates. Params: thread = thd to wait on, value_ptr = location to put thd termination val. Returns: 0 → success, EDEADLK → deadlock (thd already joined), EINVAL → thd param not valid, ESRCH → couldn't find thd.

Per process items	Per thread items
Address space	Program counter (PC)
Global variables	Registers
Open files	Stack
Child processes	
Signals	

Pre-Emption: kernel interrupts+suspends task before it completes/yields. Done via interrupts. IHandler saves process/thread state before potential context switch to other scheduled process.

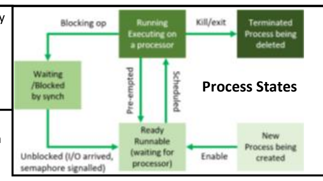
User Level Threads: kernel unaware of thds – only interacts with pcs. Pcs maintains thd table for scheduling. Thd management impl by library. **A** = performance (no expensive kernel involvement in thd ops e.g. creation, switching, synch), customisation (each app has own scheduler for that prog). **D** = blocking (any block syscall on 1 thd blocks all in same pcs), page faults (during interrupts for page faults all thds in pcs are blocked), pre-emptive (pre-emptive scheduling hard for u-ts). **Kernel Level Threads**: kernel can manage individual thds with pcs. **A** = blocking (thds individually blocked), simpler (easier scheduler as kernel can get interrupts but must be general). **D** = performance (still cheaper than proc switching but more expensive kernel interactions, can be mitigated by thd pool apps), customisation (no app specific schedulers). **Hybrid**: take adv of thd-specific blocking, but cheaper ops. OS = kernel level thds. Program manages pool of kernel provided thds (scheduled by kernel). Can use user-level synch (faster).

Scheduler Goals: Fairness (comparable processes get comparable CPU time), all run (avoid indefinitely postponing processes), maximise utilisation (maximise resource allocation), minimise overhead (context switches, running scheduler), correctness (scheduler correctly impl). **Fair-Share Scheduling**: each user has a ready queue used by scheduler, which RRs between them to fairly distribute CPU time. We study div time fairly for processes. **Batch Systems**: Optimise for throughput (jobs/unit time) or turnaround time (time to completion). **Interactive Systems**: Response time critical. **Real-Time Systems**: Run jobs to meet deadlines (soft deadlines e.g. slow video recoverable, hard e.g. factory arms colliding non-recoverable).

CPU Bound Process: Pcs spends maj of time using CPU. Time to run limited by CPU performance. **I/O Bound Process**: waiting for I/O, briefly use CPU to issue I/O reqs. TTR limited by len I/O wait.

Non-Pre-Emptive Scheduler: Processes run until voluntarily yield. Software must be trusted. Bad for interactive. Good for batch systems. **Pre-Emptive Scheduler**: Requires timer interrupts. On interrupt, kernel takes control and context switches. Good for interactive processes (smaller quanta = better interactivity).

Process States



Time Estimation: SRT and SJF need to estimate. Runtime rarely known in adv or knowable at all. Heuristics based on history (not always available/representative e.g. for I/O). User/process provided estimates (inaccurate/malicious – can penalise incorrect estimates but adds overhead). Instead, most OSs use priority-based system. **General Purpose Scheduling**: favour short + I/O bound jobs (OS must determine nature of jobs + adapt to changes). Good resource util (after I/O uses CPU, descheduled) and short response times. Often I/O bound (get data) → CPU to process data. **Priority Scheduling**: schedule based on priority – always run job with highest, ps can be externally def (e.g. by user) or process-specific metrics (e.g. expected CPU burst). Ps can be static/dynamic (changed during exec), SRT can be considered priority scheduling (less time = higher p).

Schedulers: FCFS: No pre-emption. Assume all pcs sched eventually terminate/block (to allow others to run). **A** = all run (eventually), correctness (simple+ez impl), minimise overhead (simple queue = low overhead, low context switches as no pre-emption). **D** = turnaround time (TT) (small pcs stuck behind large one). **Round Robin** (RR): Periodic timer interrupt, prev pcs at back of ready queue and next run. Larger quanta → small overhead, higher response time. Smaller quanta → opposite. RR with ∞ quanta = FCFS. **A** = fairness (all ready jobs get eq CPU time), response time (low for small num jobs + small quanta), TT (low when runtimes differ), correctness (ez). **D** = TT (high when runtime similar), context switches (lowers response time, inc overhead). 100ms quanta by default (sched_rr_get_interval). **Shortest Job First** (SJF): Schedule ready thd with shortest time remaining. Assumes we know time for each pcs, non pre-emptive. **A** = TT (optimally low, if shorter added to queue current finishes first still), correctness (simple if can estimate time), context switches (very few). **D** = response time (can be high as non-pre), indefinite postponement (if many short added, long never scheduled). **Shortest Remaining Time** (SRT): pre-emptive ver of SJF. If shorter job (than curr remaining time) pre-empt curr. Need to track time of jobs. **A** = TT (low as SRT always run). **D** = indef post (many short, long never sched). **Throughput = no. jobs / sum of time for each, Turnaround = time taken for job (and all before it) to finish, RR overhead = context switch / quantum + context switch.**

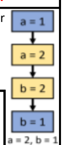
Multilevel Feedback Queues (MLFQs): Each priority level has queue, each can sched each diff (usually RR). Highest p always running (pre-empt if higher comes along), ps recomputed periodically (uses aging / recent CPU time). Can only set niceness not priority. **A** = reactive (to changing behaviour of pcs/thds), all run (priority recomputation). **D** = inflexible (apps little control, ps no guarantees), warm-up (determining factors for estimation takes time), cheating (apps add meaningless I/O to get higher p), no donation (cannot donate priority – only based on feedback from running pcs/thds). **Lottery Scheduling**: each job gets number of tickets, each scheduling decision chooses random ticket to schedule, share of tickets ≈ share of CPU time, tickets can be transferred between jobs, can use for resources other than CPU. **A** = meaningful (proportions tickets and CPU time), all run (no job starved, any with tickets run), donation (exchange tickets if job blocked by another). **D** = proportions (meaningfulness of tickets relies on proportions – adding more affects all jobs), unpredictable (random, more freq decisions help but inc overhead).

Synchronisation: For concurrent apps, cannot make assumptions about relative speed of exec for threads (context switch at any time), must consider op interleaving, real (true) concurrency consideration in multiprocessor systems. **A** = parallelism (independent ops in parallel reduce time taken), blocking (to avoid blocking ops e.g. I/O to progress, can separate them to diff threads/processes).

Race Condition: bug when result depends on non-deterministic order of ops

		Thread B	
		Read	Write
A	Read	Fine	Race Condition!
	Write	Race Condition!	Race Condition!

Happens-Before Relationships: describe order of operations in concurrent program. Strict partial order between events. Determine constraints on potential interleavings. Blue = thread 1 (pre-empted), yellow = thread 2



Synchronising Access: **Critical Section**: code accessing shared resource/data which requires mutual exclusion. Synch required at start/end of CS. **Mutual Exclusion**: 1 thd in CS at a time, no thd outside CS can prevent other thds from entering. No thd needing access to CS delayed indefinitely. No ass made about speed/scheduling. **Busy Waiting**: program continually checks on wait cond instead of blocking and being awoken (used when wait time small, wastes CPU time).

Synchronisation Methods: Disabling Interrupts: ASM: cli = clear interrupt flag (off), sti = set interrupt flag (on). **A** = simple (already impl by arch). **D** = dangerous (processes could monopolise and never get de-scheduled – only run in privileged mode), pre-emption (scheduler cant pre-empt to run others), multi-core (only disables is on single core so multi core concurrency still an issue). *Strict Alternation*: 1 thd runs at a time, busy wait in meantime, once complete set turn for next thd. Any number of thds. Thd outside CS can block others from entering. *Peterson's Solution*: Allow thds to register interest, if not int then others can run. No alternation required. Thds outside CS cannot block others from entering (as they are not interested). Enforces mutual exclusion.

Atomic Operations: typically single ASM instr cannot be interr e.g. reg/mem R/W. Use special instrs to ensure atomicity of ops. Lock prefix can be used to ensure atomicity, and TSL (test-and-set-lock) allows a boolean in mem to be read and set to true atomically. **Semaphores**: counter (no. threads which can enter, >1 useful for producer-consumer) and list (of blocked threads waiting to enter). Can enforce ME (init sema with val 1) without caring about num of thds. *int sem_init(sem_t *sem, int pshared, unsigned int value)* init sema with val i. *int sem_wait(sem_t *s)* (aka down()/P()) val--. *int sem_post(sem_t *s)* (aka up()/V()) val++. *int sem_getvalue(sem_t *s, int *valp)*, *int sem_destroy(sem_t *s)*. Enforce ordering (init to 0, thd 0 start with down(), thd 1 start with up() – 1 goes first as sema can't be downed from 0). **Locks**: enforce ME. 1 thd hold L at given time. If attempt to get L but already acq, block requesting thread. Only holding thd can release L. Methods: init, lock, unlock. Re-entrant Ls: allow L to be acquired many times by same thd. If holder attempts to lock again, will deadlock. **Spin Locks**: if expected wait low, busy waiting may be better than blocking threads. Locks busy wait (in while loop, while (TSL(l->locked) != 0);), no kernel involvement (no block/unblock syscalls), ensure check on acquiring is atomic, can run into priority inversion. **Priority Inversion**: when using priority-based scheduling alg, low-p threads can block high-p ones. LP holds resource which HP wants e.g. lock, HP tries to acq but blocked and busy waits, LP not scheduled as P is low so cannot release resource. Solution = **priority donation** → HP donates to LP so it can finish and release resource. Aging can also let LP execute if it has been a while. Spin locks mean HP always ready to run as no blocking used, but then LP cannot release lock. **Read/Write Locks**: allow many readers to hold lock, or 1 writer. Reduced lock contention as readers don't block. Some impls allow R → W upgrade or W → R downgrade w/o release + re-acquire.

Producer-Consumer: Constraints: *Producer*: can only deposit (produce) when space in buffer, only deposit if ME ensured. *Consumer*: only fetch (consume) if buffer not empty, only fetch if ME ensured. *Buffer*: limited capacity (0-N spaces). **P-C w/ Semaphores**: uses semas to count num items+space (if can't sema down then know we have reached lim so block). **P-C w/ Monitors**: CV to model producer/consumer's constraints.

Monitor: Enabled thds to wait on a cond, and signal other thds that cond has been met. Thds outside M call entry procedures (no access to internal data otherwise). Internal procedures only called from within M lock. Implicit M lock (sometimes explicit) ensure ME in M. Usually impl as language construct (e.g. Java wait + notify). **Condition Variable**: a flag representing some high-level condition (e.g. "there is space in the buffer"). 3 main ops: *void wait(condition_variable *c)* releases M lock and wait for c to be signalled, *void signal(c_v *c)* wake up one thd waiting for c, *void broadcast(c_v *c)* wake up all thds waiting for c. Signals do not accumulate (if no pcs waiting on signalled CV, signal discarded). **Hoare Impl**: thd waiting for signal immediately scheduled. **A** = simple (ez to reason about). **D** = inefficient (thd that signals switched out despite not being finished with M), scheduler (more constraints). **Lampson Impl**: sending sig + waking from wait not atomic. When CV signalled, waiting thd may be ready but waits until sched to run (cond may not be true at this point). Used over Hoare. **A** = efficient (signalling thd not switched away, no extra sched constraints). **D** = complex (extra care when waking from wait – recheck cond).

Design Decisions: *Lock Overhead*: Measure of extra resources required for using locks (memory alloc for lock structure, time to init/destroy, time for acq/release). *Lock Contention*: measure of num thds waiting on a lock (more contention = more thds blocked waiting on lock or waiting w/ spinlocks and less concurrency as blocked rather than running in parallel). *Lock Granularity*: measure of amt of data protected by a lock (coarse = whole list, fine = each list elem). In designing the synchronisation scheme, goals are: correctness, reduce overhead+contention.

Coarse Grained: few locks, large amt of data protected by each lock. **A** = simpler, lower lock overhead. **D** = higher contention, less parallelism. **Fine Grained**: many locks, protect small parts of data. **A** = lower lock contention, more parallelism. **D** = more complex, higher lock overhead.

Memory Models: Sequential consistency (all ops happen in order in source code, besides thds non-determinism) as assumed above is one mem model. Typically a **weak memory model** is used (allows compiler and hardware optimisation to improve sys performance) which can be affected by: *hardware*: reordering instrs, per-core caches on multicore processors (cache coherence) and *compiler optimisation*: instr reordering. This can affect code when conditions are changed by reordering. Typically compilers+archs have features to enforce control over ordering e.g. barriers around which instrs can't be reordered.

Deadlock: Set of pcs/thds waiting for an event only another (also waiting) can cause. Resource deadlock most common, can happen with single pcs/thds (re-acquire non-reentrant lock). **Coffman Conditions for Dlock**: *mutual exclusion* (each resource available or assigned to single thd), *hold&wait* (thread can request resources while holding others), *no pre-emption* (thread cannot have resources held forcibly revoked – holds until releases), *circular wait* (closed chain of thds s.t. each holds resource needed by another).

Resource Allocation Graphs: Directed graph to model resource allocation. Cycle = deadlock. Resource $\rightarrow_{\text{owned by}}$ thd/pcs. Thd/Pcs $\rightarrow_{\text{waits to acquire}}$ resource

Communication Deadlock: lost messages result in Dlock. Cannot use other methods to prevent DLock here. Use timeouts (wait set period for message, if none report failure+progress to other task and retry message).

Livelock: Pcs/Thds not blocked, but system as whole does not make progress. Common pattern: attempt to acq res, then back off+repeat. No thds blocked, sys continues attempting and failing progress. Often seen with spinlocks as do not block when conds for DLock met. Hence they are livelocked.

Most archs require valid pcs running at all time, so many OSs have an idle pcs when no others are running, this is not a deadlock situation as even though it is not doing anything as soon as another pcs comes along it can spring into action.

DLock Prevention: **Ostrich**: ignore it. **Detection and Recovery**: Dynamically build RAG, look for cycles (DFS), remove member in cycle if one found (e.g. revoke resource/end pcs). *Recovery*: pre-emption (take resource from owner and give to another temp), rollback (take periodic snapshot of sys and roll back+restart), killing pcs in cycle (dangerous). **Dynamic Avoidance**: for each resource request, check if it will Dlock, e.g. Banker's Algorithm. **Prevention**: write code so 4 reqs for DLock never met. *Mutual exclusion*: remove ME and share resource. **A** = no overhead (no synch overhead), more concurrency (less lock contention). **D** = race conditions. *Hold and wait*: all pcs specify what res they need before starting. **D** = redundant requests (res which may be used req'd – waste), advance knowledge (don't know what res used in advance). *No pre-emption*: allow resources to be forcibly (sometimes temp) revoked. **D** = correctness (logic issue if revoked during operations). Allow pcs to attempt to get res, req can be revoked, release other res if reqs are revoked. **D** = complexity (code must account for reqs being rejected). pthread_mutex_lock, _trylock, _unlock. *Circular wait*: force single res per pcs. **D** = inefficient, makes impl even basic programs difficult. Ensure res are acq and released in same order by pcs/thds. **A** = simple (no mechanisms needing interrupts e.g. killing pcs/revoking res), efficient (less overhead due to simple point). **D** = ordering (need to order large num of res and ensure all pcs acq in order).

Memory Management: needs to provide allocation and protection. Reqs: no knowledge of generation/use of addrs for security (isolation to hide data/prevent mem corruption). **Mem Hierarchy**: regs (1ns), L1/L2/L3 (10ns), mem (100ns), disk/SSD(25μs R, 250μs W), network (10ms-1s). L1+L2 per-core, L3 shared. Regs only store small amt + expensive.

Logical/Physical Addr: L = gen by CPU, addr space seen by pcs. P = addr seen by mem unit, refers to phys sys mem. Same in compile/load time addr-binding schemes (decide binding at comp/when prog loaded to mem), diff exec-time sch (dynamic decision).

Memory Management Unit (MMU): hardware device for mapping logical to physical addrs. Base reg = smallest phys addr, limit reg = highest logical addr, relocation reg = offset. Mmu ensures **base <= translated addr (base + rel reg) <= base + limit**. User pcs only deals with log addrs. Fast → impl in hardware. MMU keeps memory of kernel (low mem) and other pcs protected.

Fragmentation: **External**: Enough mem available, no hole large enough. Fix: *compaction* → shuffle memory to put all free mem in one block (can result in I/O bottleneck – large amt of copying). **Internal**: Allocated mem larger than req'd (wasted memory).

Multiple Partition Allocation: Hole: contiguous section of unalloc'd mem. OS keeps track of alloc/free holes for when pcs are created/destroyed. **Dynamic Storage Allocation**: First-Fit: allocate first hole that is big enough (fast+simple), Best-Fit: smallest hole large enough (unless hole list sorted, search full list). Worst-Fit: largest hole (searches entire list, worse than other 2).

Swapping: Num pcs limited by available main mem (only running pcs need to be in main mem). Supplement main mem with *swap space* (partition or file) on secondary storage (SSD, HDD) → swap non-running pcs to disk, bring back when exec again. Transfer time is long.

Virtual Memory with Paging: Abstraction to separate log mem from phys. Not all of exec pcs needs to be in mem for exec. Log addr space can be >> phys addr space. Addr spaces can be shared between pcs. Pcs creation more efficient. Fixed pg size → ext frag impossible. **Virtual Addr Space**: Impl via paging or segmentation. **0, code, data, heap → ← stack, max**. **Frames** (F): fixed-size blocks of phys mem, OS tracks free Fs. **Page**: block the same size as F in log mem (effectively a log F). **Program Start**: Prog size n frames: 1. find n free frames, load prog into that mem, 2. create page table for pcs mapping log to phys. **Termination**: destroy pg table, freeing all associated Fs. **A of VM w/ Paging**: less int (if page unused, not alloc in mem) and ext (when alloc lots of mem, split into pgs+dist across phys mem) fragmentation, and pcs alloc pgs when phys mem available (contiguous log addr can be phys non-contiguous). **Page Sizes**: Small = less int frag, potentially less mem to swap, larger pg table, best for efficient mem usage. Large = more int frag, smaller pg table, best for low addr lookup overhead. **TLB**: Translation Lookaside Buffer used to cache mappings of virtual to phys addrs. When virt addr used + corresponding phys addr found, added to cache to avoid re-computation. **Context Switch**: OS locate pg table for new pcs, set base reg in MMU to new pg table, clear now-invalid cache addr calcs from TLB.

Address Translation: Mem addr split into page num (p) and offset (d). P = index of pg in pg table, gets base addr of frame. D = offset through pg/F, combined with base addr to get phys addr. For log addr space 2^m and pg size 2^n:

p → f num	d
m - n	n

V → P addr: v / 2^n = page num, check PTE valid+dirty bits, PTE / 2^n = F, add F to front of offset.

Memory Protection: Associate protection bits with each pg. **Valid/Invalid** bit: V → associated pg in pcs' log addr space. I → page not present (page fault, load in from swap), or incorrect access (by programmer?)

Page table implementation: PT kept in main mem. PTBR (PT base reg) points to start, PTLR (PT len reg) is size. Inefficient → each data/instr access needs 2 mem accs (PT, data/instr). Use fast-lookup hardware cache as associative memory (supports parallel search). Called CAM (Content Addressable Mem), more expensive than RAM and specific to fixed mem alloc format.

Kernel Pg Access in Syscalls: when pcs makes syscall, handler runs in privileged mode so may need access to kernel pages. To ensure safe access, PT has supervisor bit (if set for translation, page can be accessed in kernel mode). Alt is to use separate PT for user/kernel.

TLB (More Deets): assoc mem cache, stores VP → PF mapping. Some store ASIDs (address-space IDs) in entries so not fully wiped in context switch (recog old entried inv using ASID). Imp performance w/ short CS (no cycles wasted in removal). Used by MIPS+ARM. **Reach** = num addrs can be cached, reach larger for larger pg sizes.

Page Table Types: **Multi-Level/Hierarchical**: break up log addr space into multiple PTs. **PT size = (addr space size / pg size) * entry size.** Large addr space+small pg size = large PT – issue (if PT larger than F, added complexity, PT uses lots of mem even if pcs doesn't access any/many pgs). Fix = use multi-level PTs (a tree). For n levels, pg num partitioned into n sections. **Pg Accessing**: **1.** get outermost PT (typically in single frame), **2.** for each partition of pg num: a. index from partition, b. entry at index in curr frame (a PT), c. in entry invalid/empty/dirty, page fault, d. else get frame from entry (set as curr frame), **3.** get phys addr (in curr frame add pg offset). **D** = more mem accesses for pcs to access mem (in TLB miss). **P2 →**

2-Level Paging

outer pg	inner pg	offset
42	10	12

3-Level Paging

2^{nd} o pg	outer pg	inner pg	offset
32	10	10	12

Hashed Page Table: Hash table to map pg nums to frame nums (normal PT is hashed with id as hashing func). PT contains linked list chains of pgs hashing to same location (more complex hash func reduces PT size at expense of conflicts). Search for match of virt pg num in chain. Extract phys frame if match found.

Segmentation: Paging gives 1-D virt addr space – what about separate addr space for code/data/stack? **Segment**: indep addr space from 0 to some max, can grow/shrink indep, support diff types of protection (r/w/x), unlike pgs, programmers aware of segments. A = good for shared libs. **D** = memory alloc harder due to variable size, may suffer external frag.

Shared Memory: pcs can access shared mem by having pgs in 2 pcs point to same frame (imp/exp mapped to files on disk). After set up, no more kernel involvement. Useful for IPC+sharing libs (more common to use mmap). **Comparison to Pipes**: higher performance (less kernel intervention, data kept in space so less copying), bi-directional comms (would need 2 pipes), less useful for uni-directional comms (kernel has synch for pipes). **System-V API**:

shmget	Allocates a shared memory segment
shmat	Attaches a shared memory segment to the address space of a process
shmctl	Changes the properties associated with a shared memory segment
shmdt	Detaches a shared memory segment from a process

Meltdown Attack: abuses speculative execution to access memory from cache side channel which wouldn't be accessible when running the code (e.g. to read kernel memory without correct privileges).

Effective Access Time: **w** = mem cycle time (μs), **e** = assoc lookup time, **a** = TLB hit ratio (related to num assoc regs), **x** = PT level + 1: **EAT = (e + w)a + (e + xw)(1 – a).** If hit, 1 mem lookup else **x** + 1.

Inverted PT: PT w/ entry for every pg/F of mem w/: virt addr of pg at location, info on owning pcs (e.g. PID). Single PT used, fast find pcs+pg num assoc w/ any F. Comp to standard PT: dec mem needed to store PT, inc time to search table (filter through entries w/ other PIDs). Can use hash table to limit linear search to one/few entries.

Linux Virt Mem Layout: 1:1 Mapping → turn log addr to phys addr for kernel pgs by subtracting 3GB. V efficient for kernel mem access, no change of PT (no change in context when pcs switching so TLB not flushed when user pcs makes syscall), on-demand mapping contains temp mapping for use of >896MB of mem in remaining 128MB of virt mem. **IA32**: 4KB pg size, 4GB virt addr space, 2-level PT (up to 3 w/ physical addr extension), offset bits contain page status (dirty, R only etc). **AMD64/x86_64**: larger pg sizes (e.g. 4MB), 48-bit addrs, up to 4-level PT, offset bits can contain can-exec (prevent malicious code taking over process).

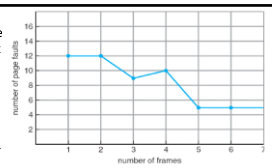
Demand Paging: Only load pages from swap when user attempts to access them. **A** = lower I/O load (unused pgs never loaded), less mem required (fewer pgs resident in mem), faster response time (no need to wait for all pgs to be loaded), supports more users (lower mem usage allows this). **Valid/Invalid Bit**: 1 = in mem, 0 = not, all pg entries init = 0, if pg with 0 accessed, page fault and trap to kernel. K uses table to determine if ref invalid (abort) or valid but pg not in mem. **Valid req**: 1. get empty frame, 2. swap pg to F, 3. reset tables (valid bit = 1), 4. restart last instr.

Performance (Demand Paging): Pg Fault Rate 0 <= p <= 1 (0 = no PFs, 1 = every ref is PF). Effective Access Time (EAT) = ((1 – p) * memory access) + p * (pg fault overhead + swap pg out + swap pg in + restart overhead).

Page Replacement: When out of free mem+new pg needs creating, find unused pg to swap out. Victim does not have to be pg of same pcs. 1. Access pg not loaded in mem (PT bit shows this), 2. update PT (ensure no RCs when involving multiple pcs on multiple cores), 3. write victim back to disk, 4. read req pg from disk, 5. restart op. **Goals**: reduce num pg faults (in general, more frames = fewer PFs), prevent over-alloc of mem (PF service routine should incl pg replacement), reduce redundant I/O (use dirty bit to only load modified pgs back to disk).

Virt Mem Tricks: **Copy-On-Write (COW)**: Pcs accessing identical pgs use same frame, only copy when one wants to write/modify it. Parent and child init share same pgs in mem. Efficient pcs creation (copy only modified pgs). Free pages alloc from pool of 0'd out pgs. **Using fork()**: C's PT points to P's pgs (marked read-only in both PTs). Protection fault causes trap by kernel. Kernel alloc new copy of pg to process altering, replaces old one in PT. P and C PT sets page to R-W. **Memory Mapped Files**: Map files into virt addr space using paging. Only load parts of file when accessed. Simplified programming model for I/O (easy access to stdin/out).

Replacement Algos: **FIFO**: replace oldest pg. **A** = simple to impl. **D** = may replace heavily used pg. Belady's Anomaly: more frames → more page faults (see graph). **Optimal Algorithm**: Replace pg which will not be used for longest time period. **D** = impossible in practice (can be used as benchmark to compare other algos tho). **LRU** (Least Recently Used): each pg has counter, every time referenced copy clock to ctr. When replacing, choose pg with lowest ctr. Proper LRU is expensive (search for lowest ctr+storing ctrs) so use approximations instead: *Reference Bit*: each pg has ref bit r = 0 (init), when pg referenced r = 1 (done by MMU), periodically reset bits, when evicting choose pg with r = 0. *Second Chance*: Evict pgs in order, but if r = 1 give second chance to stay in mem. Keeps circular queue of pgs to evict (like RR). If pg to be replaced (in RR order) has r = 1, set r = 0 and try another pg. **Counting Algos**: keep counter of num of references to each pg. *LFU (Least Freq Used)*: replace pg with smallest count, may replace very new pg just brought in, never forgets heavy pg usage (reset counters/use aging). *MFU (Most Freq Used)*: rep pg w/ largest cnt. Newly accessed pgs have low count → prioritised, pgs barely used hog mem+heavily used pgs evicted quickly.



Locality of Reference: progs tend to req same pgs across space+time. Design for this to ensure good performance. If we don't, could res in thrashing. **Thrashing**: excessive paging causing low processor util. Prog repeatedly reqs pgs from secondary storage, destroying performance.

Working Set Model: working set of pgs W(t, w) = set of pgs ref'd by a pcs while running from time interval t – w to t. Can use this in clock repl algo, keeping track of pgs in working set by adding "time of last use". At each pg fault: **1.** r = 1 → set r = 0 and move to next pg, **2.** r = 0: a. calc age, b. if age < w (working set age) → move to next pg, c. age > w → if pg clean, replace, otherwise start write-back+continue to find another pg (once w-b of frame complete, pg marked as free+clean – don't want to halt while waiting for pg to write back to disk). Effectively, only repl pgs if haven't been ref'd in w time, to take adv of temporal locality. **Working Set Size** (w): processes transition between diff working sets (transition has higher pg frames alloc to pcs, then const again in new WS). Do not dealloc pgs ref'd within w time (OS keeps some outside WS hanging around). In transitions num alloc pgs tend to be higher as pgs from 2 sets referenced within w time. WS too large → too many allocs, pcs uses too much mem. WS too small → too few allocs, many pg faults, slow. Don't want to misalloc (OS alloc for pgs prog doesn't want anymore). If many faults → alloc more pg frames (interfault time inc). When alloc all pgs in pcs, no pg faults so IFT = exec time. Graph (x = num pg frames alloc to pcs, y = IFT) follows S curve (top stops at (num pgs in pcs, total exec time)).

Local vs Global Pg Repl: **Local**: when repl pg, pcs can choose a pg belonging to same pcs. Reqs keeping track of changes in working set size + some partitioning to determine how many pgs each pcs owns/can be alloc'd – 1. fixed partitioning (each pcs gets fixed size), 2. balanced set algos. As pcs manage pg faults indep, more scalable than global. **Global**: Can choose pg from any pcs. Mem dyn shared between pcs. Init alloc mem prop to pcs size. Use pg fault freq to tune alloc (more PFs → more alloc). Use all pgs to determine rep. More efficient than local. No best sol – Linux – global, Windows – local. **Linux Page Repl**: variation of clock algo to approx. LRU pg repl. Active list → contains active pgs, MRU near head of list. Inactive list → opp, LRU near tail. Only repl pgs in inactive list. Ref bit determines when pgs moved between A+I lists. **kswapd** (swap daemon) → pgs in inactive list reclaimed when mem low, uses dedicated swap partition/file, must handle locked/shared pgs. **pdflush** (kernel thread) → periodically flushes dirty pgs to disk.

I/O Device Management Objectives: Fair access to shared devices (prevent pcs hogging res, slloc dedicated devices), exploit parallelism (can use devs in par – e.g. send packets using NIC while writing to disk – allows for multiprogramming), provide uniform+simple I/O view (abstract devs from pcs, uniform naming+err handling, hide complexity of dev handling). **Device Independence**: dev indep from it's type (e.g. terminal, disk, DVD drive) and instance (disk 1, 2, or 3). **Device Variations**: unit of data transfer (char (bytes) or block), supported ops (R, W, seek), synch or asynch (s = disk, as = NIC), speed differences (NVMe SSD vs tape drive), types of err conds (disk errs vs GPU ºC warning).

Character vs Block Dev: **C** = minimise latency (keyboard/terminal – mem = dev file for phys mem of OS, pty = pseudoterminal – bidirectional comms channel, usb = USB devs). **B** = maximise throughput (disks/NIC – ramdisk = access RAM disk in raw mode, fd = floppy disk, loop = loop device - maps data blocks to file in filesys, or other block dev).

I/O Layers:
- User-level I/O Software
- Device-Independent OS software
- Device drivers
- Interrupt Handlers
- Hardware

Interrupt Handler: Interrupt is sig from dev to CPU that dev needs attending to (dev connecting, finish reading, error etc). Drivers register handlers to deal w/ diff interr types. **Block**: on transfer completion, sig dev handler. **Character**: when char transferred, process next. Modern Systems – **Hybrid**: (e.g. nvme storage) polling (queue of reqs+resps) very fast but uses CPU time, wait for interr (better for longer waits as can sched other pcs).

Device Driver: handles type of dev, can control mult of same type. Impl block R/W, access dev regs (write control info to dev), init ops (e.g. start dev at boot), sched reqs (if dev shared), handle errs.

Buffered I/O: Output → user data transferred to OS output buffer. Pcs cont (only suspend when buffer full). Input → OS reads ahead, reads taken from buff, pcs blocks when buff empty. Smooths I/O traffic peaks (limited load balancing), allow diff data transfer unit sizes between devs (buff contains blocks). **Unbuffered I/O**: data transferred directly between dev+user space, each R/W causes phys I/O (dev does smth, not just buffer), dev handler used for every transfer, high switching overhead (every R req driver to take over+do phys action).

Dev-Indep OS Layer: standard interfaces for drivers of dev types → simps OS des, interface to write new drivers to, no OS changes to supp new drivers (just new interfaces). Provides **dev indep**: map log to phys devs (naming+switching, map 1:many l:p or p:l), req validation against dev (check dev+driver working correctly), alloc (which pcs can access which devs), buffering (performance+block size indep), err reporting.

Device Drivers: **Mem Mapped I/O**: devi addr as mem location hence can use virt mem setup to restrict acc (supervisor bit). **Ways to do I/O**: **Programmed**: simple but ineff, wait for device (spin) then continue exec. **Interr Driven**: large overhead, good for long expected waits. Hardware sends interr when op complete, do other work while waiting. **Direct Mem Acc** (DMA): requires hardware, reduces CPU intervention. DMA controller (often in dev) waits for dev to respond, then once full res avail, places direct into mem.

Device Alloc: **Dedicated Device**: (DVD writer, terminal, printer) – 1 pcs gets exclusive accs, if another tries to acc it fails (can keep queue of open reqs), typically alloc for large periods of time, only alloc to authorised pcs (avoid malicious pcs blocking accs to res). **Shared Device**: (disks, window terminals) – OS provde sys for sharing e.g. filesys to use disk. **Spooled Device**: blocking user accs to alloc (non-sharable)devs causes delays+bottlenecks – spool to intermediate medium (disk file). Daemon pcs has dedicated accs, pcs send reqs/jobs to daemon which provides sharing on non-sharable res, and red I/O time (greater throughput).

User-Level I/O Interface: syscall interface to allow user progs to interact, often with 3rd party libs. Basic I/O ops (close, R, W, seek), set up params (dev indep), can be synch (blocking) or asynch (non-blocking). *UNIX Files*: accesses virt devs as files (standard I/O calls). FD:Name → 0:stdin, 1:stdout, 2:stderr. /dev/kdb for keyboard accs.

Loadable Kernel Module (LKM): Dev drivers loadable modules, loaded+linked dyn with running kernel. Required binary compatibility (mod specific to kernel ver). **Kmod**: kernel subsys manages mods w/o user intervention, determines sumbod deps, loads mods on demand. *init_module* all init code, *cleanup_module* clean shutdown. Kernel can open file, look for symb table (gen by compiler) and call corresponding funcs. *sudo insmod some_module.o* insmod loads mod to kernel. **I/O Management**: *Device classes*: group sim types of dev. *ID numbering* Major = det which driver is controlling, Minor = distinguishes dev of same class. *Special files* most devs rep by dev/special files in /dev dir. **Device Access**: accessed via virtual file system (VFS). Most drivers impl R/W/seek but all have other ops too. *ioctl(cdrom, CDROMEJECT, 0)* syscall supports special tasks. **Character Dev I/O**: transmits data as stream of bytes. Represented by char_device_struct (driver name, registered maj+min nums, ptr to driver's file_operations struct via cdev). All registered drivers ref'd by chrdevs vector. file_operations (ops supported by dev driver, stores funcs called by VFS when syscall accesses dev special file). **Block Dev I/O**: **Block I/O Subsys**: several layers, modularised ops (common code in each layer). 2 main strats to min time accessing block devs: 1. caching data, 2. clustering I/O ops (store up tasks, exec many at once). **Direct I/O**: bypass kernel cache when accessing drive, good for databases (+others) where caching can reduce performance/consistency (e.g. vals rarely accessed more than once). **Linux API**: **I/O Classes**: character (unstructured, files+devs), block (structured, devs), pipes (message, interprocess comms), socket (message, network interface). **Sockets**: can be local or across network (pipes use machine-specific fds so cannot go over network). Types: TCP, UDP. *fd = create(filename, permission)*, *fd = open(filename, mode)* mode = 0 (R), 1 (W), 2 (RW), *close(fd)*, *numbytesread/write = read/write(fd, buffer, numbytes)*, *pipe(&fd[0])*, *newfd = dup(oldfd)*, *ioctl(fd, op, &termios)*, *fd = mknod(fname, perm, dev)*. **File Descriptors**: each pcs has fd table, 3 fds when created (0=stdin, 1=out, 2=err) all refer to terminal from which prog was started.

Disk Layout: cylinder = stack of disks, track = concentric circles on disk, sectors = radii sections. Inter-sector/-track gap as expected. R/W head (1 per surface).

HAMR: (Heat Assisted Magnetic Recording) to ensure high precision, platter can only be written to when heated (laser heats platter, cold sections ignore R/W ops).

Blocking: process suspended until op complete, I/O call only returns once complete (appears instantaneous from prog perspec). Pcs making call could be blocked for a while, threads get around this (complex). **Non-Blocking**: I/O call ret as much as is avail. Constantly R until get all data (could return none sometimes). App-level polling for I/O (constantly req R from stdin, op only if data at moment). Turn on using **fcntl** syscall for sending comms to manage fds). **Asynch**: pcs runs in parallel w/ I/O ops (non-blocking). When ops complete pcs notified, pcs can check/wait for I/O completion. Flexible+efficient, more complex code, potentially less secure if buffs mismanaged.

Sector Layout: surface div into 20+ zones, outer = more sectors per track. Ensures sects have same phys len. Zones hidden using virt geometry. **Disk Addressing**: phys addr = (cylinder, surface, sect) (hidden from OS). Modern sys use **logical sector addressing/logical block addresses (LBA)**: sects numbered consec 0..m, makes disk management easier (just need sector num+offset in sect). Helps work around BIOS limitations.

Disk Scheduling: **FCFS**: reqs completed in order received. **A** = fine for light loaded disk (time between reqs larger than time to fulfil any req), fair (no bias). **D** = low performance for heavy load. **SSTF** (Shortest Seek Time First): order reqs on shortest seek dist from curr head pos. **D** = biased against inner/outermost tracks (middle on ay close), unpredictable performance, pcs can use dummy reqs to keep control of head), can delay reqs indefinitely. **SCAN**: elevator – select reqs w/ shortest seek time in preferred direction. Only change dir when at inner/outermost cylinder (no more reqs in dir). Base for most common algos used. **D** = same delay issue as SSTF (but reduced – only in 1 dir), long delays for reqs not in dir of algo/on extremes. **C-SCAN**: SCAN in 1 dir, jumping to start (innermost) when at the end (outermost). **A** = lower variance of reqs on extreme tracks, largely red indef wait issue from SSTF. **N-Step SCAN**: only services reqs waiting when sweep began (for each sweep). 1 sweep = inner → outer → inner. **A** = reqs arriving during sweep serviced before end of sweep (no long waits), no indef waits pcs.

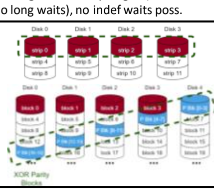
Disk Formatting: Must format disk before use. **Low Level Format**: [preamble | data | ECC]. P = id type of block, D = storage, ECC = Error Correction Codes (correct small errs in R/W). *Interleaving*: seq data spread apart s.t. CPU has time to process sector before next R. *Cylinder Skew*: skew n means when head at sector x, next is x + n). **High Level Format**: boot block (block dedic to starting sys), free block list, root dir, empty filesys.

Disk Delay: *Typical Disk* → sector size = 512 bytes, seek time (adj cyl) = <1ms, seek time (avg) = 8ms, rotation time (avg latency) = 4ms, transfer rate = +100MB/s. *Disk Scheduling*: minimise seek/latency times (seek approx. 2-3x larger than latency time), ordering pending disk reqs w.r.t. head pos.

Disk Performance: **b** = num bytes to transfer, **N** = num bytes/track, **r** = rot speed (rev/s). Seek time: t_{seek}, lat time = $t_{latency}$ = **1/2r**, trans time: $t_{transfer}$ = **b / rN**, total acc time = $t_{access} = t_{seek} + t_{latency} + t_{transfer}$.

Disk Sched: I/O reqs in req list, one for each dev in sys, **bio** structure assoc mem pgs w/ reqs. Block dev drivers def req op called by kernel, kernel passes ordered req list, driver performs ops in list, dev drivers do not define file R/W ops. Some drivers bypass kernel ordering and do it themselves (RAID), done for complex setups where ass made by kernel ordering algo don't apply. **Default**: var of SCAN – kernel attempts to merge reqs to adj blocks, but synch R reqs may starve during large Ws. **Deadline Scheduler**: ensures Rs performed by deadline, eliminating starvation. **Anticipatory Scheduler**: delay after R req completes – pcs will issue another synch R op before it's quantum expires. Reduced seeking behaviour → reduced throughput if pcs does not issue R req to nearby location → anticipate pcs behaviour from past behaviour.

RAID (Redundant Array of Inexpensive Disks): Disk performance not kept pace with CPU performance → RAID inc disk based sys performance by using many disks in parallel. Arr of phys drives appears as single virt drive, distribute stores data over disks to allow parallel operation (imp performance). Use redundant disk capacity to respond to disk failure (more disks → lower mean time to failure). **Levels**: 0: (diag 1) spread blocks in RR on disks, concurrent seek/transfer of data (for blocks on diff phys disks), sometimes balance load across disks, no redundancy. 1: level 0 but all disks duplicated. Rs serviced by either disk, Ws must update both disks in parallel, failure recovery easy, low space efficiency, high cost. 5: (diag 2) most commonly used, distributes parity so potential for concurrency. Some potential for W concurrency as parities on diff disks, good storage/efficiency tradeoff, reconstruction of disk non-trivial+slow.



Category	Level	Description	I/O Data Transfer (read/write)	I/O Request Rate (reads/writes)
Striping	0	Non-redundant	+ / +	+ / +
Mirroring	1	Mirrored	+ / 0	+ / 0
Parallel access	2	Redundant via Hamming code	++ / ++	0 / 0
	3	Bit interleaved parity	++ / ++	0 / 0
Independent access	4	Block interleaved parity	+ / -	+ / -
	5	Block interleaved distributed parity	+ / -	+ / - or 0

SSD: **Sched**: no sched algo as many mem modules can be R/W in parallel, and R/W speed approx. constant. Drivers need to overcome issues with limited Ws, tracking virt to phys blocks+assignment of free blocks. **Comp to HDDs**: high bandwidth (1GB/SSD vs 100MB/HDD), lower latency, high parallelism.

Disk Caching: Can cache sectors of disk in main mem to reduce access times. Buffer in main mem contains copies of disk sectors, OS manages disk in terms of blocks (likely much larger than sectors so loads multiple). Must ensure contents saved in case of failure (lazy writing complex). **Cache** has finite space → need replacement policy. **LRU** (Least Recently Used): replace block in cache longest with no refs. Cache = stack of pointers to blocks in mem, when ref'd pushed to top of stack, algo evicts bottom of stack. **D** = doesn't track num of accesses, only rel time. **LFU** (Least Frequently Used): replace block w/ fewest refs – each block has counter. Some blocks ref'd many times in short period → misleading ref count – use freq-based repl. **Frequency-Based Replacement**: Divide LRU stack into 2 secs – new+old. Block ref → move to top of stack, only inc ref count if not already in new. **D** = blocks age out too quickly → use 3 secs, only replace blocks from old.

File Systems: Organise info. Main obj: non-volatile+long-term storage, sharing info, concurrent access, convenient organisation, easy management of data, security. **File Types**: hard links (file aliases data of another file), soft links (aliases path to a file by another – ln command), regular (.bat, .sh, .lib, .zip, .docx…), directory, char special (access to char I/O dev), block special (same but for block devs). **Filesystem Support Functions**: name translation (conv paths to disks+blocks for log use by driver), management of disk space (alloc/dealloc storage for files), file locking for exclusive access (important when concurrency – fcntl syscall), performance optimisation (caching/buffering), protection against sys failure, security. **File User Funcs**: truncate (erase contents but keep all other attrs), reposition/seek (set file curr pos to given val), R attrs (creation date, size, archive flag), W attrs (protection, immutable flag).

File Attrs: *Basic*: name, type, organisation, creator. *Addr Info*: volume, start addr, size used, size alloc'd. *Access Control Info*: owner, authentication, permitted actions. *Usage Info*: creation timestamp, last modified/read/archived, expiry date, access activity.

File Syscalls: *fd = open(file, how, …)*, *s = close(fd)*, *n = read(fd, buffer, nbytes)*, *n = write(fd, buffer, nbytes)*, *pos = lseek(fd, offset, …)* – move fp, *s = stat(name, &buf)* – get file metadata, *s = fcntl(fd, cmd, …)* – file locking/other ops.

Inodes (index blocks): on file open, OS opens inode table/make = inode entryin mem. Inode contains type+access control, num links to inode, user+group ID, access +modification time, inode change time, direct, indirect, doubly indirect, triple indirect ptrs to data blocks, disk dev+inode num, num pcs with open file, maj/min device num.

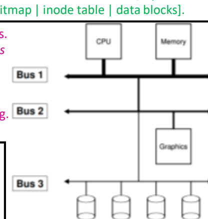
Directory Syscalls: *s = mkdir (path, mode)*, *s = rmdir (path)*, *s = link (oldpath, newpath)* new hard link, *s = unlink (path)*, *s = chdir (path)*, *dir = opendir (path)*, *s = closedir (dir)*, *dirent = readdir (dir)* R 1 entry from dir, *rewinddir (dir)* rewind dir to re-read.

Filesys Directory Organisation: map symbolic names (text.txt) to logical disk locations (Disk 0, block 234). Helps with file organisation+prevents naming collisions (all names in given dir unique). Hierarchical FS → root = where root dir begins, root dir points to various dirs each having own entries for it's files. **Link**: ref to dir/file in other part of FS (allows alt names/diff locations in tree). *Hard Link*: ref to addr of file (not dirs). *Soft Link*: ref full pathname of file/dir, created as dir entry. *Problems*: file deletion (search for links+remove, leave links and cause exception when used (symbolic links), keep link count in file+delete when cnt=0, looping → if files point at self/loop of others, they never get deleted). **Directory Operations**: open/close, search (pattern match on strings), create/delete files, link/unlink files, change dir, list files, read/write attrs, mount. **Mounting**: combine (make links to) multiple FSs into one namespace. Allows ref from single root dir, support soft links to files in mounted FS (not hard as they are dependent on FS impl). *Mount point* = dir in native FS assigned to root of mounted FS. FSs manage mounted dirs. With mount tables (info abt location of mount points/devices – when native FS encounters mount point, use MT to determine dev and type of mounted FS). **Directory Representation**: dirs map symbo names to inodes (can be other dirs. – typically only single block) or files.

Filesys Organisation: **Space Alloc**: File size variable (can inc/dec) and is allocated on disk in blocks (512-8192 bytes). Block size determined by filesys. *BS Too Small*: high overhead for managing large files (many blocks to keep track of), high file transfer time (more blocks = more seeking back+forth). *BS Too Large*: internal frag (small files wasteful), caching based on blocks → large cache space/unable to cache many blocks. **Contiguous File Alloc**: Files at contiguous addrs on storage device. **A** = successive logical records typically phys adj. **D** = external frag, poor performance if files grow/shrink, file grows beyond init size specified and no contig free block avail → transferred to new area of adequate size → many additional I/O ops. **Block Linkage/Chaining**: Each file contains linked list of blocks. When locating block, chain searched from beginning (O(n)), if blocks dispersed through disk searching is slow. Waste pointer space in each block. Insertion/Deletion by modifying pointer in prev block. **Block Allocation Table**: Uses dir mapping files to first block. Table maps blocks to next block in file, indicating free spots (block with no next is EOF). **A** = file allocation table (**FAT**) can be cached in mem for quick lookup, no lengthy seeks to traverse block nums for file. **D** = files can become fragmented reducing R/W speed → needs periodic de-fragmenting (expensive), FAT can become impractically large using lots of mem (>1 block of storage). **Index Blocks**: Each file has >=1 index blocks containing list of pointers to file data blocks (effectively page table for filesys). File's dir entry points to index block. *Chaining*: reserve last free entries in index block to store ptrs to more index blocks. **A** (over simple linked-list impl) = searching in index blocks themselves, index blocks placed near corresponding data blocks → quick data access, cache index blocks in mem.

ext2fs: the second EXTended File System is high-performance and robust (formerly standard, remains robust tho). Uses block sizes typically 2^ 10, 11, 12, or 13. 5% block reserved for root (safety mech to ensure root pcs can always run even after malicious ones use all disk space). ext2 inode used to rep files+dirs, uses 12 direct pointers, 13th indirect, 14th doubly, 15th triple (fast access to small files, also allowing for large files). **Block Groups**: clusters of contiguous blocks. FS attempts to store related data in same BG. Reduces seek time for accessing groups of related data. BG structure: [superblock | group descs | block alloc bitmap | inode alloc bitmap | inode table | data blocks].

Buses: *Bus 1*: High bandwidth, low latency devices. Devs that can function at same speed as RAM. *Bus 2*: High bandwidth, medium latency devs. Devs have high throughput+req low latency but slower than mem (e.g. graphics cards, network). *Bus 3*: Other. Slower devs/dev with lower bandwidth (e.g. block devices – disk).



Free Space Management: When alloc new block for file, need to quickly determine which is available. **Free List**: linked list of blocks containing locations of free blocks. Bs allocated from start of free list, newly freed blocks appended to end of list. **A** = freeing+alloc fast (O(1)). **D** = files unlikely to be contiguously alloc → when reading files have to seek across many random locations on disk which is slow. **Bitmap**: 1 bit in mem for each B, ith bit = ith B. **A** = uses little mem (single bit/entry), quickly determine contiguous blocks at locations, highly optimised bit ops as standard on most CPUs. **D** = may need to search entire bitmap to find free spot (O(n)).

Filesys Layout: boot block, super block, free inode bitmap, free block (zone) bitmap, inodes+data. *Super Block*: contains crucial info about filesys: num inodes, num data blocks, start of inode+free space bitmaps, location of 1st data block, block size, max file size.

← **P1 Multi-Level/Hierarchical Page Table**: Allocating non-pg sized mem can be hard so nice if indiv parts of pg table fit in 1 frame each (partially what multi-level PTs achieve). Single level wastes lots of mem.

Security: **Goals**: data confidentiality (theft of data), data integrity (destruction/alteration), system availability (DoS). **Policy v Mechanism**: P = what security if provided (what is protected, who has access, what access permitted), M = how to impl P. Same M can support diff Ps. **People Security**: attacks by insiders (abuse privileges), social engineering (phishing, blagging, shouldering), convenience (not changing pwords often enough), ignorance/lack of knowledge. **Hardware Security**: phys access (damage machine, read/alter contents of disk, snoop on/forge network traffic), hardware flaws (side channel attacks e.g. cache attack, timing attack, data remnance, or badly impl access control). **Software security**: buffer overflow, integer overflow, string formatting. Can gain root privileges/crash app/steal data/deny access to sys.

Access Control: Principals = users/pcs. **Authentication**: Personal characteristics: (key based on user, usually biometrics/signature). **A** = hard to forge, convenience. **D** = special hardware (expensive), false +ves/-ves. Possessions: RFID cards, implants, secure keys, 2FA. **A** = convenience. **D** = lost/stolen (impersonation), sometimes expensive (complex locks/card scanners). Knowledge: secret known to user (password, with freq turnover). **A** = cheap, secure if secret. **D** = password reuse diminishes security, only as secure as password storage, dictionary attacks. **Authorisation**: who can access what and how. *Principle of Least Privilege*: user gets min reqs to carry out task. Sometimes more given by default/for convenience. Access rights defined as objects with operations permitted on them. Principal executing in domain D has access rights specified by D. Access control matrix (col = obj, row = principal). Access control list → 2D arr too large to store, so store each col as list + associate with object. Capability list = rows of matrix as lists assoc with principals.

UNIX/Linux File Access: users are principals, each has UID. Can belong to +1 groups. UID = 0 = root (all access rights). All files are objects, can belong to at most one group. **-rw-r--r-- 1 prp lsds 2517522 Mar 9 09:14 test.txt** → file type (- = reg, d = dir, b = block f, c = char f, l = symbo link, p = pipe, s = socket), owner rights, group rights, else rights, owner, group name. Permission bits (- = none, r = read, w = write, x = exec, s = setuid/gid, t = sticky bit). File exec w/ privs of user executing. SUID set → use owner's privs. Each user has effectively 3 UIDs: 1. Real UID = ID of user that started pcs, 2. Effective UID = ID of executing process, 3. Saved UID = UID the effective UID can be switched to.

Access Right	File (f)	Directory
Read	Read f	List dir contents
Write	Write to f	C/D owned files
Exec	Exec f	Enter dir + access fs

Password protection: only store 1-way hash of password (etc/passwd → plaintext passwords, etc/shadow → hashed pwds). **Rainbow Tables**: table of hashes for most popular passwords. Hash func cannot be inverted so must brute force. **Password Salting**: salt created upon account creation, stored with hash (which takes salt as param). Use salt to calc password when user logs in.

Capability Lists: capabilities are protected pointers to objects, specifying permitted operations. Kernel owns capability info, user uses fd as a key of sorts for indirect access to info.

Action	Best	Access Control Lists	Capabilities
Principle of Least Privilege	Cap	Use of groups mean all users in a group get access rights.	Can be very fine grained, only giving capabilities to specific users.
Revocation	ACL	Can revoke all access by changing one ACL for the object.	Must search all capability lists, and remove capability.
Rights Transfer	Cap		Can transfer a single capability from one user to another.
Persistence	ACL	File systems are designed to be persistent.	Very complex.

DAC vs MAC: DAC = Discretionary Access Control = principals determine access rights to their objects (default). MAC = Mandatory Access Control = system rules/policies determine ARs. **Bell – La Padula Model**: MAC policy where info doesn't travel down security hierarchy. Pcs only read at it's security level or lower, only write at same/higher. **Biba Model**: opp.

Design Principles: Least privilege = pcs run with lowest priv poss, default no access. Simple+uniform mechanism, psychologically acceptable (if too complex/burdensome ppl will not impl), public (can be critiqued for flaws).