Process: unit of activity characterized by exec of sequence of instructions, current state & associated set of system resources, consists of program code & associated data at least
Trace: behavior of individual process by listing sequence of instructions
Dispatcher: switches processor from one process to another
Process Creation: assigns identifier, allocates space, initializes process control block, sets appropriate linkages, creates or expands other data structures
Process States: New, Ready, Blocked, Ready/Suspend, Blocked/Suspend, Running, Exit; process creation state is special (not in memory yet, code not executed yet); suspended process may not be removed until agest (process that sent command to suspend, ex: itself, parent, OS) orders removal explicitly
OS Control Structures: memory tables (allocation of main & secondary memory, protection attributes, info needed to manage virtual memory), I/O tables (manage devices & channels, has status of operation & location in main memory used as source/destination of transfer), file table (existence of files, location on secondary memory, current status, info may be maintained & used by file management system), process tables (has reference to memory, I/O & files, tables themselves subject to memory management)
Process Image: program, data, stack, PCB
Process Control Block (PCB): most important data structure in OS, for protection only handler is allowed to read/write
PCB parts: Consists of ids, processor state (user visible registers, control & status registers (program counter, condition codes, status), stack pointer (to the top)), process control (sched & state info (process state, priority, scheduling info, event (waiting for)), data structuring (if linked list has pointer to next process), inter process communication, privileges, memory management (pointer to segment & page tables), resource ownership & utilization)
Kernel Functions: process management (creation & termination, scheduling & dispatching, switching, synchronization & inter process communication support, process control blocks), memory management (allocation of address space, swapping, page & segment management), I/O management (buffer management, allocation of channels & devices), support functions (interrupt handling, accounting, monitoring)
Process Switch Steps: save processor context, update PCB of running one, move PCB to appropriate queue, select another process, update selected PCB, update memory management data structures, restore processor context for selected process
OS Execution Types: non-proc kernel (user proc on top of kernel), exec with user proc (user proc all have OS functions, all on top of proc switching functions), proc based OS (OS functions are in separate procs alongside user procs, all above proc switching functions)
Thread Parts: exec state, saved thread context when not running, exec stack (user stack plus kernel stack), some static storage for local vars, and access to shared mem and resources of process
Thread Uses: foreground & background work, asynchronous processing, speed of execution, modular program structure
Thread Types: User Level Thread (ULT), Kernel Level Thread (KLT)
ULT Advantages: no need for kernel mode privs, app specific sched possible, OS indep
ULT Disadvantages: can block whole process on system call but thread itself is not blocked (thread blocked status is for waiting on another thread), no multiprocessing
KLT Advantages: opposite of ULT disadvantages, kernel routines can be multithreaded
KLT Disadvantages: thread switch requires switching to kernel mode
Scheduling: important in multitask & multiuser sys, ex: app then daemon, deadlines
When to schedule: new process, process exits, I/O wait, blocks on lock, I/O interrupt (resume waiting process or interrupted process?), *generally* when process/thread can no longer continue or activity results in more than 1 ready process
Scheduling Types: long term (add to list of processes to execute), medium term (add to main memory), short term (actual execution), I/O (which request to handle)
Short Term Scheduling Criteria: user oriented (ex: turnaround time, response time, deadline, predictability), system oriented (ex: throughput, processor utilization, fairness, enforcing priorities, balancing resources)
Short Term Scheduling Parts: selection func & decision mode (preemptive does not monopolize and better overall service to procs but more overhead, opp for non preemptive)
Performance Indices: Arrival, Service (total execution time), Turnaround (total time spent), Normalized Turnaround (turnaround divided by service, 1.0 is best)
FCFS: imple, non preemptive, performs better for long processes, favors CPU bound, performs badly given wildly varied jobs
Round Robin: preemptive, regular interrupt, next process chosen using FCFS, effective for general purpose time sharing systems, short time quantum improves responsiveness, long time quantum improves efficiency (less time switching), still favors CPU bound (fix is to let previously blocked processes finish their time quantum before other ready processes, which is virtual round robin)
Shortest Process Next: non preemptive, choose proc with shortest expected run time (need to know), gives minimum avg wait time, possibility of starvation for longer procs, not suitable for time sharing, can be unfair to short procs given varied mix (like FCFS)
Shortest Remaining Time: preemptive, when new proc arrives OS will preempt running proc if expected time to completion for current is longer than the new one (need to know), long procs may be starved, no bias for long procs like FCFS (is actually against), no additional interrupts like in RR, better turnaround time than SPN because short jobs get immediate attention, but higher overhead (need to know elapsed times)
Highest Response Ratio Next: non preenptive, chooses process with highest $RR = \frac{w+s}{s}$, where w is waiting time and s is expected service time (need to know them), after it $RR = \frac{T_r}{T_s}$, considers age of process, if more waiting time response ratio increases, so more likely to be chosen, and longer processes will not be starved
Priority Scheduling: priority per process, can have queue per priority, each queue can have own algorithm, priority boosting needed to prevent starvation (do this for old processes)
Feedback: preemptive, priority 0 is highest, queues per priority, new process start at 0, when preempted or blocked priority reduced, FCFS for each queue (round robin for last), can starve long processes (partial fix: increase time quantum for lower priority queues (ex: priority k can get 2^k time), also possibly do priority boosting after waiting for some time)
Fair Share Scheduling: decide based on process group/user instead of individual thread/process, give fair share to each group, give fewer resources to group with more than fair share & more to group with less than fair share, scheduling done on basis of process priority, recent processor usage of process & group
Tightly Coupled Multiprocessing: set of CPUs which share OS and often large mem size
Parallelism Classes: Independent (no explicit sync, typical use: time sharing sys, lower response time), Coarse grained (multiple procs, ex: parent spawns child, results accumulated in parent, sync every 200 – 1,000,000 i), Medium grained (multithread, sched of one thread affects whole app perf, sync every 20 – 200 i), Fine grained (programmer must use special i & write parallel programs, tends to be very specialized and fragmented with many different approaches, sync every < 20 i, possibly 1)
Multiprocessor Scheduling Issues: *assignment of processes to processors*: static (for uniform MP, assigned to a processor for total life of process, short term queue for each processor, simple, little overhead, one processor may be idle while other has long queue), dynamic (for uniform MP, process may change processor during lifetime, single global queue, more efficient, if shared memory context info available to all processors, cost of scheduling independent of processor identity), master-slave (simple but master is bottleneck), peer (complicates OS, need to handle case where 2 processors want same job/resource); *multiprogramming on individual processors*: may be useful to leave some processors idle to handle interrupts and allow cooperating processes/threads to run simultaneously, so multiprogramming may not be needed; *process dispatching*: FCFS is best (less overhead), RR can handle a varied mix of jobs better but very small for 2 processors, even smaller for more
Process vs Thread Scheduling: thread switching has less overhead, threads share resources, some principle of locality applies
Load Sharing/Self Scheduling (LSSC): single ready queue shared by all processes
LSSC Advantages: even load distrib, no central sched req, global q can be orgd appropriately (FCFS, smallest n of threads 1st (both preemptive or not), FCFS apparently better)
LSSC Disadvantages: mutual exclusion on the central queue must be enforced (can be bottleneck), local caching less effective (same thread unlikely to go to same processor), unlikely that all threads of program run together (limits thread communication)
Gang Scheduling: set of related threads sched on set of CPUs at same time, reduce blocking due to sync, less proc switching, less sched overhead (decide once for a group)
Gang Scheduling Types: uniform (app gets $\frac{1}{m}$ of available time in n processor), weighted (amount of processor time is weighted by number of threads)
Dedicated Processor Assignment: group of CPUs assigned to a job for the whole duration of job, extreme gang sched, each thread assigned to a CPU, results in idle CPUs (no multiprogramming), but not so important in highly parallel sys, and no proc switch at all
Dynamic Scheduling: both OS and app are involved in scheduling decisions, OS mainly allocates processors to jobs, while jobs allocate processors to threads
Dynamic Scheduling Steps: when job requests processors (new job or new thread), if there are idle processors then request satisfied, else if it is a new job take one processor from another job (if they have multiple) and allocate, if cannot satisfy then wait in queue or job rescinds request, when processors released scan queue (assign 1 processor per new process, then allocate to other requests using FCFS)
Memory vs Processor Management: similar, processor thrashing occurs when scheduling of threads needed now induces de-scheduling of threads which will soon be needed, processor fragmentation occurs if leftover processors not enough to satisfy waiting jobs

Multicore thread scheduling issues: prioritize reducing access to off chip mem instead of max CPU utilization, use caches, which are sometimes shared by some but not all cores
Cache Sharing: part of above, cooperative resource sharing (multiple threads access same memory locations), resource contention (multiple threads competing for cache use)
Contention Aware Scheduling: allocate threads to cores to maximize shared cache & minimize need for off chip memory access
Real Time System (RTS) Types: hard (must meet deadline always, unacceptable damage otherwise, ex: airbag, fly-by-wire, ABS), soft (must mostly meet deadlines, desirable but not mandatory, still makes sense to schedule and complete task even if deadline has passed, ex: multimedia, navigation, washing over time)
RTS Properties: arrival time, max exec (service) time, deadline (start or end)
RTS Task Categories: periodic (regular interval, max exec time is the same each period, arrival time is start of period, deadline is the end of period), aperiodic (arrive any time)
RTS Characteristics: *determinism*: delay before acknowledging interrupt, operations performed at fixed, predetermined times or within predetermined time intervals; *responsiveness*: time taken to service interrupt after acknowledgement, includes time to initially handle interrupt and begin executing service routine, time required to perform routine and interrupt nesting; *user control*: much broader in real time than standard; *reliability*: more important than normal; *fail safe operation*; *stability*: system will meet deadlines of most critical, highest priority tasks even if some less critical task deadlines are not always met
Static Table Driven: precomputed schedule for periodic tasks
Static Priority Driven: precomputed prio, use preemptive prio sched, also for periodic
Dynamic Planning Based: task arrives prior to execution, scheduler determines whether new task can be admitted, works for both periodic & aperiodic
Dynamic Best Effort: priority assigned on task arrival based on characteristics, tries to meet all deadlines, abort processes which have missed the deadline, no guarantee of timing constraint of task until complete, typically aperiodic, used by many current systems
Preemption only for ending deadlines, NOT starting
Periodic Load: given n events, event occurs in period T and requires C time, $U = \frac{C}{T}$, $\sum U \leq 1$ to be possible to schedule, timeline repeat when all task deadlines align
Rate Monotonic Scheduling: static priority driven, shorter period higher priority, $\sum U \leq n(2^{\frac{1}{n}} - 1)$ (conservative), performance difference is small relative to earliest deadline first, most hard real time systems also have soft parts which are not used with rate monotonic scheduling, also stability is easier
Priority Inversion: high priority task forced to wait for a lower prio task
Unbounded Priority Inversion: duration depends on unpredictable actions of other unrelated tasks as well as the time to handle the shared resource, ex: P1 needs to wait for P3, but P2 is higher priority than P3, so it runs, so P3 cannot proceed, so P1 cannot proceed as well, to solve increase priority of P3 to above P1, so it runs before P2 (implementation: resource priority is highest priority user + 1, then that priority is assigned to processes that use the resource, after that the process priority returns to normal)
Concurrency: can be interleaved only or both overlapped and interleaved
Concurrency Contexts: multiple applications (allow processing time to be shared among active apps), structured apps (extension of modular design & structured programming), OS structure (OS implemented as set of processes/threads)
Concurrency Key Terms: atomic op (guaranteed to exec as a group or not exec at all), crit sec (code within proc that req shared resources, must not be exec while another proc is in corresponding section), deadlock (procs unable to proceed, waiting on each other), livelock (procs constantly change states in response to others without useful work), mutex (when one proc in crit sec no other proc can be in crit sec that access same shared resource), race cond (multiple threads/procs r/w a shared data item and final result depends on relative timing of exec), starvation (situation in which runnable proc overlooked indefinitely by sched)
Concurrency Difficulties: sharing of global resources, hard for OS to manage alloc of resources optimally, hard to find prog errors as results nondeterministic and nonreproducible
Concurrency OS concerns: must be able to keep track of various procs, alloc & dealloc resources for each active proc, protect data and phys resources of each proc against interference by others, ensure that procs and outputs are independent of processing speed
Process Competition: each process unaware of others, each process should be unaffected by execution of others, only one process can safely access resource at a time, others must wait, control problems are mutex, deadlock, starvation
Process Data Sharing: procs interact indirectly by sharing data, but may not be explicitly aware, still compete, but also need coop to ensure integrity and coherence of shared data
Process Communication: way to synchronise ot coordinate activities, use message passing protocol, mutex not big problem, but deadlock & starvation still possible
Mutex Reqs: must be enforced, proc halting in noncrit sec must do so without interfering with other proc, no deadlock, no starvation, if no proc in crit sec a proc must be permitted to enter without delay, proc remains inside crit sec for bounded time only
Mutex Memory Assumption: simultaneous memory access to same location is prevented by hardware, requests serialised and granted in unspecified order
Mutex Hardware Support: disable interrupts on multiCPU, special atomic i
Mutex HW Advantages: applicable to any number of processors sharing main memory, simple, mutex easy to verify, can support multiple critical sections (each with own variable)
Mutex HW Disadvantages: busy waiting, starvation (next process selection arbitrary), deadlock (P2 high priority & P1 low, P1 critical, interrupted & switch to P2, P2 attempt to access resource P1 used, P2 wait but P1 never run due to priority)
Compare & Swap: parameters are memory location, test value and new value, if old value in memory equals the test value store the new value, return the old value; while (cs (mem, 0, 1) == 1 do nothing, then afterwards set mem to 0)
Exchange: params are register and memory, swap contents of them; reg = 1, do ex (reg, mem) while reg $\neq 0$, afterwards set mem to 0
Mutex OS & PL Support: semaphore, monitor, message passing
Semaphore: an int with queue, can init (set sem to non neg value), wait (sem-1, if become neg then block), signal (sem+1, if still non pos then unblock waiting proc), all ops atomic, strong uses FIFO (starvation free), weak uses unspecified order, may starve
Binary Sem: wait (if 1 set to 0, else block), signal (if queue empty set to 1, else unblock)
Sem Consequences: no way to know before process decrements sem if will block, which process will continue on uniprocessor when running concurrently, whether another process is waiting so may or may not unblock process
Producer/Consumer: one or more prod gen data & place them in buffer, single cons take items out 1 at a time, only 1 prod/cons may access buffer at one time, problem is to ensure prod can't add to full buffer and cons can't remove data from empty buffer
Monitor: software module consisting of one or more procedures, init sequence, local data (only accessible by monitor procedure), process enters monitor by calling procedure, only one process may be executing in monitor at a time
Monitor Synchronization: achieved by condition variables (local in monitor), cwait (suspend exec of caller on condition), csignal (resume exec of process waiting on condition)
Monitor vs Sem: mutex is enforced by monitor itself, synchronization is confined to monitor but still programmer responsibility to call cwait/csignal, easy to verify & debug, sem needs programmer to enforce mutex, synchronization, programming/debugging difficult
Hoare Monitor: textbook, thread that signals gives up lock, waiting thread gets it, once done lock return to signalling thread
Mesa Monitor: signalling thread keeps lock, waiting thread waits for lock
Readers/Writers: data area shared among many processes, some only read, some only write, any number of readers may simultaneously read the file, only one writer may write to the file, if a writer is writing to the file no reader may read it
Mutex Software Support: assume only basic mem mutex, executing on single or multiprocessor **Peterson's Algorithm**: (flag[id]=true, turn=otherId, while flag[otherId] and turn=1 do nothing, critical, flag[id]=false), mutex enforced using the flags, deadlock prevented using turn (can only be either 1 or 0)
Deadlock Conditions: mutex, hold-and-wait (process may hold resources while waiting for others), no preemption (forcibly unlock resource), circular wait (closed chain of processes exist that each process holds at least one resource needed by next process in chain), first 3 necessary but not sufficient, last one is both (unresolvable is definition of deadlock)
Deadlock Prevention: indirect (prevent first 3, preemption practical if possible to save state, ex: memory allocation, not others), direct (prevent circ wait by defining linear order of resources, if process alloc R_i can get R_j if $i < j$)
Deadlock Avoidance: decision dynamically made whether current resource request will potentially lead to deadlock, can block resource allocation only or block process initiation
Deadlock Avoidance Advantages: not necessary to preempt and rollback processes, less restrictive than deadlock prevention
Deadlock Avoidance Restrictions: maximum resource requirement must be stated in advance, processes must be independent (no synchronization requirements for order of exec), must have fixed number of resources to allocate
Banker's Algorithm: resource allocation denial, use 2 vectors (resource/total, available), 2 matrices (claim/needed per process, allocation/already allocated), safe state if at least one sequence of resource allocation to processes that does not result in deadlock (can finish all)
Deadlock Detection: grant resource request whenever possible, can have checks every request (early detection, simple, but frequest checks consume processor time)
DD Algorithm: use alloc matrix, request matrix and avail vector, also create temp vector as copy of avail, start with marking all 0 in allocation matrix, afterwards unmarked proc involved in deadlock, loop steps:

1. find unmarked process where requests are $\leq$ available (means it can complete), if none break
2. mark the process, then add its allocated to temporary (copy of available), goto 1
DD Advantages & Disadvantage: never delays process init, facilitates online, but inherent preemption losses
Deadlock Recovery: abort all deadlocked processes, rollback all deadlocked to checkpoints and restart, successively abort deadlocked processes or preempt resources (doing this also requires rollback to before acquired) until deadlock gone
Dining Philosophers: 5 forks, no 2 philosophers can use the same fork at same time (mutex), no philosopher must starve (also no deadlock), with semaphore wait room, wait left, wait right, eat, release right, release left, release room, room init to 4, with monitor cwait & get left & right, release in same order (if empty set true, else csignal)
Memory Management: memory divided to OS & user, user divided to each process, need to make sure of reasonable supply of ready processes, reqs:
Relocation: may be in different place, support for this also allows support for next two
Protection: against interference from others, accidental or intentional, must be provided by hardware, memory references must be checked at run time
Sharing: (allow controlled access to shared areas without compromising protection, processes running same code should run same address space)
Logical Organization: (memory organized into linear address space, programs organized to modules, modules can be written & compiled spearately with cross referencing resolved at runtime, different degrees of protection available to different modules, modules should be shareable as designated by programmer)
Physical Organization: (could be left to programmer but impractical & undesirable, memory available for program + data may be insufficient, never know how much space is avaialble if multiprogramming)
Fragmentation: Internal (in partition), External (outside partition)
Fixed Equal-Size Partitioning: simple with minimal overhead (easy placement), but program may be too big to fit, inefficient utilization, internal fragmentation
Fixed Unequal-Size Partitioning: can use one q per partition or one global q, more flex but small jobs will not use space efficiently, n of partitions limits n of active proc
Dynamic Partitioning: variable number and size of partitions, allocated exactly per process, has external fragmentation, can manage with compaction but waste of time
Dynamic Placement Algorithms: Best-fit (worst, external frag), First-fit, Next-fit
Address types: logical (reference to memory location independent of current assignment of data to memory), relative (to some known point), physical/absolute (actual location in main memory, relative + base, compare with bound before access, if OK access, else interrupt)
MM Terms: frame (fixed length block of main memory), page (fixed length block of data in secondary memory, may be temporarily copied to frame), segment (variable length block of data in secondary memory, whole segment may be copied into main memory or divide it into pages and load them separately)
Paging: use pages and frames, all pages need to be in memory (no VM yet), only internal fragmentation (at the end of the last page of the process)
Page Size Effect: smaller gives less internal frag, closer modelling of loc of ref, less page faults, but more pages per proc, larger page tables, maybe double page fault, if larger then smaller page tables, but loc of ref weakens, page fault rate increases unless most of proc fits into one page, also n of alloc frames is a factor
Page Table: maintained by OS for each process, contains frame location for each page in process, used to produce physical address (relative from base is logical, first few bits are page number, replace it with frame number to get physical)
Segmentation: divide program to segments, require all segments to be in memory (without VM), no internal fragmentation, less external than dynamic partitioning (can split app to multiple parts instead of one contiguous), usually visible to programmer, typical code & data in different segments, to get physical address need to get base address + offset via segment table, needing to check offset is protection, sharing can be achieved by segments referencing multiple processes
Segmentation Advantages: data structures of unpredictable size can be handled, program structure can be organized around segments (separate compilation), segments can be shared among processes, can be organized into segments of same protection level (more powerful, easier to control than with paging)
Virtual Memory: ensure currently exec parts of proc in mem when req, rest can be on disk, gives very large virt addr space, allow secondary mem to be addressed as part of main mem, addr used by app is diff from phys addr, size limited by addressing scheme and amount of secondary mem instead of main mem
VM Terminologies: virtual address (address assigned to location in VM to allow it to be accessed like part of main memory), virtual address space (virtual storage assigned to process), address space (range of memory addresses avaialble to a process), real address (address of storage location in main memory)
Resident Set: pages/segments of process currently in main memory, marked in table
VM execution: if request is in resident set then proceed, else fault (interrupt, OS blocks process, issues I/O request for required piece, dispatches another process, when I/O complete OS puts process to ready queue)
VM Advantages & Disadvantages: more processes may be in memory at once, possible for process to be larger than main memory (programmer no need to care about address trasnlation), some overhead (page/segment tables need updating), but still big gain in efficiency (main memory constraint gone, multiprogramming more effective)
Thrashing: sys spend most time swapping pieces rather than exec proc, try to guess using history which pieces are least likely to be used, principle of loc applies, for practicality HW must suport paging & seg, OS must incl SW for managing movement of pages/segs between secondary mem and main mem
VM Address Translation: page number + page table pointer to get frame number, then concatenate with offset, max page number usually more than max frame number
Page Table with VM: will need some extra bits (ex: currently in main memory, has been modified since loaded) may need multilevel page table (first few bits as index to root page table, next few bits as index to next level page table to actually get frame number), or use inverted page table (one entry per frame, to get frame hash page number, if same hash but not right then follow chain to find right one, each entry has page number, process id, chain pointer, control bits $\rightarrow$ flags, protection, locking info)
Segmentation with VM: segment table needs some extra bits, to translate to physical seg number + seg table pointer to get base, then + offset and check if within limit
Paging & Segmentation: user address space broken up to segments, each segment broken up to pages, so first need seg number + seg table pointer to get page table location, then + page number to get frame number, then concatenate with offset
Translation Lookaside Buffer: cache page table entries to avoid 2x memory accesses, contains recently used page table entries, associative lookup in cache to speed up search, works with standard main memory cache
OS Policies for VM: below
Fetch Policy: demand paging (seeme best for most purposes, bring in page only when memory reference is made, can cause large number of page faults before settling down), Pre-paging (bring in pages likely to be referenced, such as those adjacent to referenced, if stored contiguously then overhead small, can be triggered by page faults)
Placement Policy: no perf effect on paging, seg same as dynamic partitioning
Replacement Policy: due to principle of locality often there is high correlation between recent referencing history & near future referencing patterns, more elaborate policies will have greater overhead, may lock frames to avoid replacement (for kernel, key control structures, I/O buffers, time-critical areas)
Optimal: choose page for which next ref is furthest in future, not practical, but least n of page faults
Least Recently Used: choose page which has not been refd for the longest time, matches principle of loc, almost as good as opt, but many impl difficulties
FIFO: simple to implement, removes page which has been in memory the longest, no principle of locality, performs badly
Clock: when page loaded or referenced use bit 1, any frame with use bit 1 is passed over and set to 0, after replacing move clock position forward, can use modify bit as well to prefer replacing unmodified pages
Page Buffering: improves paging perf, allows simpler repl policy, when repl page keep in free or modified page list (keep small n of frames free, frame at head of free list used, add to list at tail, so if was repl but needed again just get it back)
Resident Set Management: decide how many pages to bring to main memory, if smaller per process then more process, but small number of pages will increase page faults, beyond certain size more frames will have no effect (locality)
Fixed-Local: fixed n of frames for proc, page repl is chosen from alloc frames of proc
Variable-Global: easiest to impl, OS has list of free frames, when page fault add free frame to proc res set, if no frames avail then OS must repl page, could use page buffering
Variable-Local: when load process allocate number of frames, when page fault occurs replace a page in process resident set, reevaluate allocation and increase/decrease to improve performance, more complex but better performance
Cleaning Policy: determine when modified page should be written out, demand cleaning (write out only when selected for repl), precleaning (allows writing of pages in batches)
Load Control: number of processes resident in main memory (multiprogramming level), too few then need to swap to get ready process, too many then thrashing, to reduce level need to choose process to swap out (lowest priority, faulting, last process activated, smallest resident set, largest process, largest remaining execution window)
I/O Device Categories: human readable, machine readable, communication (modem)
Differences in I/O Devices: data rate, application/use, complexity of control, unit of transfer (block vs stream), data representation (encoding), error conditions
OS Design Objectives for I/O: Efficiency, Generality (way processes view I/O devices & OS manages devices & ops, diversity makes it hard, so use hierarchy)
Hierarchical Design: functions of OS should be separated according to complexity, characteristic time scale, level of abstraction, leads to series of layers in OS organization, layers should be defined so changes on one layer do not require changes to other layers
Local Peripheral Device: logical I/O (managing general functions, use device in terms of id, open, close, read, write), device I/O (convert ops and data to sequences of I/O instructions, channel commands, controller orders, may use buffering), scheduling and control (queue & schedule of I/O ops, control of ops, interrupt handling, status collected and reported, direct interaction with hardware)
Communication Port: difference with above is communications arch instead of logical I/O, may itself consist of layers, ex: TCP/IP
File Structure: diff with above is 3 layers instead of comms arch, which are:
Directory Management: symbolic file names converted to identifiers that either reference the file directly or indirectly through file descriptor or index table, also concerned with user ops that affect directory of files, ex: add, delete, reorganize
File System: logical struct of files and ops (open, close, r/w), also manage access rights
Physical Organization: convert logical refs to files & records to physical addresses (tracks, sectors), also allication of space and main storage buffers
I/O techniques: Programmed I/O (no interrupt, transfer through processor), Interrupt driven I/O (through processor but no busy wait), Direct Memory Access
DMA Configurations: single bus detached DMA, single bus integrated DMA (I/O devices behind DMA), I/O bus (all behind single DMA)
I/O Buffering: smooth out peaks in I/O demand (eventually all buffers full and no more advantage), transfer input in advance of request, transfer output sometime after, no buffer then time per block is T (input time) + C (compute time), 1 buffer then max (T,C) + M (move from buffer to user memory), 2 then max (T,C) only (OS fill one, user take from other, or other way), or circular
Block Oriented Single Buffer: anticipated input, when transfer complete block moved to user space and request another, general speedup but complicates OS & swapping logic
Stream Oriented Single Buffer: line or byte at a time (each keystroke, or sensors, controllers)
Disk Performance Parameters: seek time (move head), rotational delay, access time
FIFO: fair to all, approximates random if there are many processes competing (bad)
Shortest Service Time First: least movement away from current position
SCAN/LOOK: elevator, arm moves in one direction, SCAN satisfies all requests until last track then reverse, LOOK reverses after no more requests in direction, favours jobs near innermost or outermost
N-step-SCAN: avoid starvation by arm stickiness, segments the disk request q to sub q of length N, each sub q using SCAN, when sub q processed new must go to other, if fewer than N avail all processed, large N approach SCAN, N=1 FIFO
Priority: imposed outside of disk management to meet other objectives in OS, not for optimising disk use, ex: interactive (and maybe short batch) jobs given higher priority, good response time but long jobs slow
Redundant Array of Independent Disks: Availablility: (A0: lower than single, A1: higher than single, A2: higher than A1, A3: max), 0 (A0), 1 (A2), 2 (Hamming code, N+m, A1), 3 (Bit-interleaved parity, N+1, A1), 4 (Block-interleaved parity, N+1, A1), 5 (Block-interleaved distributed parity, N+1, A1), 6 (Block-interleaved dual distributed parity, N+2, A3), table below compare to single disk by default

n	large transfer	small request
0	very high	very high
1	> read, == write	up to 2x read, == write
2	highest of all	2x approx
3	highest of all	2x approx
4	== L0, sig < write	== L0, sig < write
5	== L0 read, < write	== L0 read, general < write
6	== L0 read, < L5 write	== L0 read, sig < L5 write

File Management (FM): consists of system utils that run as privileged apps, at least needs special services from OS, at most part of OS
File Properties: long term exist, sharable between proc, struct (int & ext)
File System (FS): means to store data organized as files as well as ops (create, delete, open, close, read, write)
File Structure Terms: Field (basic element of data, single val), Record (collection of related fields, can be treated as unit), file (collection of similar records, treated as single entity, may be refd by name, access control apply here), Database (collecion of related data, relationships explicit, consists of one or more files)
FM Objectives: meet data management needs of user, guarantee valid data in file, optimize performance, provide support for variety of storage device types, minimize potential for lost or destroyed data, provide standardized interface to process, provide multi-user support
FM User Requirements: create, delete, read, modify, write files; have controlled access to other users' files; control what type of access allowed to files; restructure files in form appropriate to problem; move data between files; backup & recover files in case of damage; access by name rather than by identifier
FS Architecture: Access Method (file struct, access, proc data), Logical I/O (files), Basic I/O Superv (sched), Basic FS (place & buff of blocks), Device Driver (start & end of req)