Process: unit of activity characterized by execution 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 Identifiers, processor state (user visible registers, control & status registers (program counter, condition codes, status), stack pointer (to the top)), process control (scheduling & 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 process control block of running one, move process control block to appropriate queue, select another process, update selected process control block, update memory management data structures, restore processor context for selected process
OS Execution Types: non-process kernel (user processes on top of kernel), execution with user processes (user processes all have OS functions, all on top of process switching functions), process based OS (OS functions are in separate processes alongside user processes, all above process switching functions)
Thread Parts: execution state, saved thread context when not running, execution stack (user stack plus kernel stack), some static storage for local variables, and access to shared memory 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 privileges, app specific scheduling possible, OS independent
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 multitasking & multi user systems, ex: app first then daemon, also 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 function & decision mode (preemptive does not monopolize and better overall service to processes but more overhead, opposite 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 process with shortest expected running time (need to know), gives minimum average waiting time, possibility of starvation for longer processes, not suitable for time sharing, can be unfair to short processes given varied mix (like FCFS)
Shortest Remaining Time: preemptive, when new process arrives OS will preempt running process if expected time to completion for current is longer than the new one (need to know), long processes may be starved, no bias for long processes like FCFS (is actually against), no additional interrupts like in Round Robin, better turnaround time than Short Process next 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 processors which share OS and often large amount of memory
Parallelism Classes: Independent (no explicit sync, typical use: time sharing system, lower response time), Coarse grained (multiple processes, ex: parent spawns children, results accumulated in parent, sync every 200 – 1,000,000 instructions), Medium grained (multiple threads, scheduling of one thread affects whole application performance, sync every 20 – 200 instructions), Fine grained (programmer must use special instructions & write parallel programs, tends to be very specialized and fragmented with many different approaches, sync every < 20 instructions, 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 distribution, no centralized scheduler required, global queue can be organized appropriately (FCFS, smallest number of threads first (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 scheduled on set of processors at same time, reduce blocking due to synchronization, less process switching, less scheduling 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 processors is assigned to a job for the whole duration of the job, extreme gang scheduling, each thread assigned to a processor, results in idle processors (no multiprogramming), but not so important in highly parallel systems, and no process switching 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 memory instead of maximizing processor 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/release time, maximum execution (service) time, deadline (starting or completion)
RTS Task Categories: periodic (regular interval, max execution 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 priorities, use preemptive priority scheduler, 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: circumstances within the system force a higher priority task to wait for a lower priority 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: more than one process/thread operating at the same time, 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 operation (guaranteed to execute as a group or not execute at all), critical section (code within process that requires shared resources, must not be eecuted while another process is in corresponding section), deadlock (processes unable to proceed, waiting on each other), livelock (processes constantly change states in response to others without useful work), mutual exclusion (when one process in critical section no other process can be in critical section that access same shared resource), race condition (multiple threads/processes read/write a shared data item and final result depends on relative timing of execution), starvation (situation in which runnable process overlooked indefinitely by scheduler)
Concurrency Difficulties: sharing of global resources, hard for OS to manage allocation of resources optimally, difficult to locate programming errors as results nondeterministic and nonreproducible
Concurrency OS concerns: must be able to keep track of various processes, allocate & deallocate resources for each active process, protect data and physical resources of each process against interference by others, ensure that processes 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: processes interact indirectly by sharing data, but may not be explicitly aware, still competition, but also need cooperation 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 Requirements: must be enforced, process halting in noncritical section must do so without interfering with other process, no deadlock, no starvation, if no process in critical section a process must be permitted to enter without delay, process remains inside critical section 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 multiprocessor, special atomic instructions
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 negative value), wait (decrement sem, if become negative then block), signal (increment sem, if still non positive then unblock waiting process), 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 producers generating data & placing them in buffer, single consumer taking items out one at a time, only one producer/consumer may access buffer at one time, problem is to ensure producer can't add to full buffer and consumer 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-style Monitor: textbook, thread that signals gives up lock, waiting thread gets it, once done lock returned to signalling thread
Mesa-style Monitor: most real OS & Java, signalling thread keeps the 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, mutex impossible, hold-and-wait inefficient, preemption practical if possible to save state, ex: memory allocation), direct (prevent circular wait by defining linear ordering of resources, if process allocated 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 allocation matrix, request matrix and available vector, also create temporary vector as copy of availability, start with marking all 0 in allocation matrix, afterwards unmarked processes 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: (need to be swapped in/out, 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 avaialble for program + data may be insufficient, never know how much space is avaialble if multiprogramming)
Fragmentation: Internal (data loaded smaller than partition), External (memory outside partitions small pieces)
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 queue per partition or one global queue, more flexible but small jobs will not use space efficiently, number of partitions limits number of active processes
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 fragmentation), 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, only internal fragmentation (at the end of the last page of the process)
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, 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