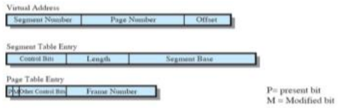
The rate monotonic scheduling algorithm selects the tasks with the highest rate: **True**
Frame: fixed-length block of main mem. Available chunks of mem
Page: fixed-length block in secondary mem (ex: disk). Can be temp copied into frame of main mem. Chunks of a process
Segment: variable-length block in secondary mem. May temporarily be copied into available region of main mem (segmentation), or divided into pages that can be copied into main mem (paging & seg)
Mem Mgmt satisfies reqs: Relocation, Protection, Sharing, Logical Org, Physical Org
Relocation: typically unknown which other programs in main mem @ time of program exe. Active processes need to be swapped in/out main mem to max processor util. Specifying proc must be placed in same mem region when swapped back in would be limiting: may need to relocate process to diff area
Protection: proc need to get perms to ref mem locations for reading/writing/ Location of prog in main mem unpredictable. Mem refs generated by proc checked @ runtime. Mechs supp relocation supp prots
Sharing: adv to allow each proc access to same copy of program instead of own copy. Mem mgmt. must allow controlled access to shared areas of mem w/o compromising prots. Mechs supporting relocation supp sharing capabilities.
Logical Org: mem org as linear; modules written & compiled independently & can have diff degrees of prots, sharing on mod lvl corresponds to users' way to view prob. Segmentation most readily satisfies reqs
Phys Org: can't leave programmer w/ responsibility to manage mem; doesn't know how much space available. Mem available for prog + its data may be insufficient -> overlaying allows various mods to be assigned to same region of mem, but is time consuming.
Mem Partitioning: mem mgmt brings procs into main mem for exe by processor. involves virt mem & based on seg & paging. Partitioning doesn't involve main mem
Fixed Partitioning: main mem divided into # of static parts @ sys generation time. equal-sized parts where any proc whose size <= part size can be loaded into available part. OS can swap out proc if all parts are full & no proc is in ready/running state.
Adv: simple to implement; little OS overhead.
Disadv: inefficient use of mem due to internal frag (wasted space since any program occupies entire part regardless of size); max # of active proc is fixed
Unequal Size Partitions: helps lessen problems
Disadv: # parts specified @ sys gen time limits # of active procs. Small jobs use part space inefficiently
Dynamic Partitioning: parts variable length & #. Proc allocated exactly as much is required. Used by IBM's mainframe OS, OS/MVT.
Adv: no internal fragmentation; more efficient use of main mem.
Disadv: inefficient use of proc due to need for compaction to counter external fragmentation; mem util declines. (compaction is ;time consuming & wastes CPU time).
Placement Algos:
Best-fit: chooses block closest in size
First-fit: begins to scan mem from beginning & chooses 1st available clock large enough
Next-fit: begins scan mem from last placement & chooses next available large enough block
Buddy Sys: comprised of fixed & dynamic partitioning schemes. Space available for alloc treated as single block. Mem blocks available of size 2^k words, $L \le K \le U$, where 2^L smallest alloc block & 2^U largest block alloc (usually size of entire mem available)
Address
Logical: ref to mem location independent of curr assignment of data to mem
Relative: addy expressed as location relative to some known point
Phys/Absolute: actual location in main mem
Paging: parts mem into relatively small equal fixed-size chunks (frames). Proc divided into small fixed-size chunks of same size (pages). Proc loaded by loading all pgs into available frames.
Adv: no external fragmentation
Disadv: small amount of internal fragmentation
Page Table: maintained by OS for each proc, contains frame location for each pg in process, proc must know how to access for curr proc, used by processor to produce phys addy.
Segmentation: program can be subdivided into segments that may vary in length (but there is a max length). Addressing consists of 2 parts: seg # & offset. Similar to dynamic partitioning. Proc loaded by loading all segs into dynamic parts
Adv: no internal frag; improved mem util & reduced overhead
Disadv: external fragmentation.
Security Issues: If proc has not declared portion to be sharable, then no other proc should have access to contents of that mem portion. If proc declares that mem portion may be shared by other designate procs, OS security must ensure only those procs have access
Buffer Overflow Attacks (buffer overrun): security threat related to mem mgmt that can occur when proc attempts to store data beyond limits of fixed-size buffer. prevalent & dangerous type of security attacks
Defending Against Buffer Overflows: Prevention, Detecting & aborting. Countermeasure categories: Compile-time defenses that aim to harden programs to resist attacks in new programs & Run-Time Defenses that aim to detect & abort attacks in existing programs
Select the parameter used in deadline scheduling that specifies the time a task must begin: **Starting deadline**
Hardware & Control Structs: Fundamental characteristics so not all pgs/segs must be in main mem during exe: 1) all mem refs are logical addies dynamically translated into phys addies @ runtime, 2) proc may be broken up into pieces that don't need to be contiguously located in main mem during exe.
Terms:
***Virtual Mem**: storage alloc scheme where secondary mem can be addressed as if part of main mem. Addies a program may use to ref mem distinguished fr addies mem sys uses to identify phys storage sites, & program-gen addies translated automatically to their machine addies. Size limited by addressing scheme of comp sys & amount of secondary mem available & not by actual # of main storage locations.
***Virt Address**: addy assigned to a location in virt mem to allow it to be accessed as if part of main mem.
***Virt Address Space**: virt storage assigned to a process
***Address Space**: range of mem addies available to a process
***Real address**: Address if storage location in main mem
Process Execution: OS brings few pieces of program into main mem (resident set). Interrupt generated when a needed addy is not in main mem. OS places proc into blocking state. Piece of proc w/ logical addy brought into main mem: OS issues I/O Read request; another proc runs un while disk I/O takes place; interrupt issues when I/O disk complete, causing OS to place affected proc in Ready state.
Implications: More procs may be maintained in main mem: only load some pieces of each proc; w/ so many procs in main mem, likely a proc in Ready state @ any time. Proc may be larger than main mem.
Real & Virt Mem: Real mem is main memory, actual RAM. Virt mem is mem on disk, & allows for effective multiprogramming & relieves user of tight constraints of main mem
Characteristics of Paging & Segmentation

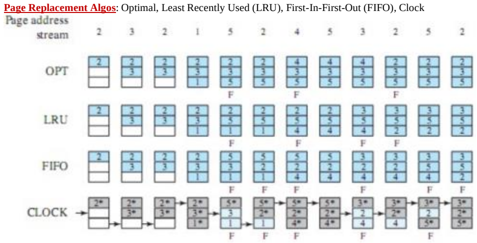
Simple Paging	Virt Mem Paging	Simple Segmentation	Virt Mem Segmentation
Main mem partitioned into frames	Main mem partitioned into frames	Main mem not partitioned	Main mem not partitioned
Progs broken into pgs by compiler / mem mgmt sys	Progs broken into pgs by compiler / mem mgmt sys	Program segs speced by programmer to compiler	Program segs speced by programmer to compiler
Internal frag in frames	Internal frag in frames	No internal frag	No internal frag
No external frag	No external frag	External Frag	External Frag
OS must maintain pg table for each proc showing the frame each pg occupies	OS must maintain pg table for each proc showing the frame each pg occupies	OS must maintain seg table for each proc showing load addy & seg length	OS must maintain seg table for each proc showing load addy & seg length
OS must maintain free frame list	OS must maintain free frame list	OS maintain free hole list in main mem	OS maintain free hole list in main mem
Processor uses pg #, offset to calc abs addy	Processor uses pg #, offset to calc abs addy	Processor uses seg #, offset to calc abs addy	Processor uses seg #, offset to calc abs addy
All pgs of proc must be in main mem for proc to run, unless overlays used	Not all pgs need be in main mem frames for proc to run. Pgs may be read in as needed	All segs of proc must be in main mem for proc to run, unless overlays used	Not all segs need be in main mem for proc to run. Seg may be read in as needed
	Reading pg into main mem may require writing pg out to disk		Reading seg into main mem may require writing 1+ seg to disk

Thrashing: state where sys spends most of its time swapping proc pieces rather than exec instructions. To avoid this, OS tries to guess based on recent history the pieces least likely to be used in near future
Principle of Locality: program & data refs w/in proc tend to cluster. Only few pieces of a proc needed over short pd of time -> possible guess which will be needed in the future. Avoids thrashing.
Supp needed for Virt Mem: hardware must supp paging & segmentation. OS must include software for managing movement of pgs and/or segs b/w secondary mem & main mem
Paging: term virt mem usually associated w/ systems that employ paging. Each proc has own pg table, where each entry contains frame # of corresponding pg in main mem
Select the function from processor scheduling that deals with virtual memory: **Medium-term Scheduling**
Select the block-oriented device: **Disk**
The main benefit of Gang scheduling is to reduce the overhead when executing a set of related threads: **True**
Select the RAID level that does NOT include redundancy: **RAID 0**
A page is a fixed-length block of main memory: **False**
Select the memory management technique which is visible to the programmer: **Segmentation**
On a fixed partitioning memory system, the number of processes in main memory can be greater than the number of partitions: **False**



(c) Combined segmentation and paging

Inverted Page Table: pg # portion of virt addy is mapped into hash value -> points to inverted pg table. Fixed proportion of real mem required for tables regardless of # of procs or virt pgs supped. Struct inverted bc it indexes pg # entries by frame # instead of virt pg #.
Each entry includes pg #, process identifier (pid that owns this page), control bits (includes flags & prot locking info), and chain pointer (index val of next entry in chain)
Translation Lookaside Buffer (TLB): special highspeed cache virt mem schemes use to overcome doubling mem access time (each virt mem ref cause 2 phys mem accesses: fetch pg table entry, fetch data)
Associative Mapping: TLB only contains some pg table entries -> can't index into TLB based on pg #; each TLB entry must inc pg # & complete pg table entry. Processer equipped w/ hardware to simultaneously interrogate TLB entries to determine if there's a match on page #.
Page Size: smaller page size -> less internal frag. But more pgs required per process -> larger pg tables. For large programs in multiprogrammed environ, portion of page tables must be in virt mem instead of main mem. Phys characteristics of most secondary-mem devices favor large page size.
New programming techniques used in large programs tend to decrease locality of refs w/in a process. Page design issue related to phys main mem size & program size -> main mem & addy space used by apps getting larger (most obvious on PCs where apps become increasingly complex)
Segmentation: allows programmer to view mem as having multiple addy spaces/segments
Adv: simplifies handling of growing data structs, allows programs to be altered & recompiled independently, lends itself to sharing data among processes, lends itself to protection
Seg Organization: each seg table entry contains starting addy of corresponding seg in main mem & seg length. A bit needed to determine if seg already in main mem & a bit to determine if seg has been modded since being loaded
Combo Paging & Segmentation: user's addy space broken into segments & each segment is broken into fixed-size pgs that are equal in length to a main mem frame. Segmentation visible, paging transparent.
Protection & Sharing: segmentation lends itself to prot & sharing policies. Each entry has base addy & length so inadvertent mem access can be controlled. Sharing can be achieved by segs refing many procs.
OS Software: design of mem mgmt portion of OS depends on whether or not to use virt mem techniques, use of paging/segmentation/both, algos employed for various aspects of mem mgmt.
Policies for Virt Memory: Key issue: performance (min pg faults). Includes: **Fetch Policy** (demand paging, prepaging), **Placement Policy**, **Replacement Policy** (Page buffering, Basic Algos: Optimal, LRU, FIFO, Clock), **Resident Set Mgmt** (fixed/variable resident set size, global/local Replacement Scope), **Cleaning Policy** (Demand, Precleaning), **Load Control** (Degree of multiprogramming).
Fetch Policy: determines when page should be brought into memory. **Demand Paging**: only brings pgs into main mem when ref to location on pg is made. Many page faults when proc first started. Locality suggests that as more pgs brought in, most future refs will be to pgs recently brought in, & pg faults should drop. **Prepaging**: pgs other than the one demanded by pg fault brought in. exploits trait of most secondary mem devices. if pgs of proc stored contiguously in secondary mem, more efficient to bring in # of pgs @ a time. Ineffective if extra pgs not reffed. Shouldn't be confused with "swapping"
Placement Policy: determines where in real mem proc piece will reside. Important design issue in seg sys. Paging/combo paging w/ seg placing irrelevant bc hardware performs functions w/ equal efficiency.
Replacement Policy: deals w/ selection of pg in main mem to be replaced when new pg brought in: removed pg least likely to be reffed near future. More elaborate -> greater hardware & software overhead.
Frame Locking: when frame locked, the curr stored pg in that frame may not be replaced. OS kernel & key ctrl structs held in locked frames. I/O buffers & time-crit areas may be locked into main mem frames. Locking achieved by associating a lock bit with each frame
Page Replacement Algos: Optimal, Least Recently Used (LRU), First-In-First-Out (FIFO), Clock



***Optimal**: selects pg where time to next ref is the longest. Produces 3pg faults after fram alloc filled
***Least Recently Used (LRU)**: replaces page that hasn't been referenced for longest time. By locality, should be page least likely to be refed in near future. A lot of overhead
***First-In-First-Out (FIFO)**: Treats page gram malloced to proc as circular buffer. Pgs removed round-robin style (simple replacement policy). Page in mem longest is replaced
***Clock Policy**: requires association of additional bit w/ each frame (use bit). When page first loaded in mem/reffed, use bit set to 1. Frame set considered to be circular buffer. frame w/ use bit = 1 passed over.
Page Buffering: Improves paging performance & allows use of simpler page replacement policy. A replaced page isn't lost its either in a) free page list (list of pg frames available for reading in pages) or b) modified page list (pages are written out in clusters).
***Replacement Policy & Cache Size**: with large caches, replacement of pages can have performance impact. If page frame selected in cache, it's lost with its page. In systems using page buffering, cache performance can be improved w/ policy for page placement in page buffer. Most OS place pgs by choosing arbitrary page frame from page buffer
Resident Set Mgmt: OS must decide how many pages to bring into main mem. Smaller amount of mem alloc to each proc, the more procs can reside in mem. Small # of pgs loaded increases page faults. Beyond a certain size, further allocations of pages will not affect page fault rate.
Resident Set Size: Fixed-Alloc: *Local Replacement*: fixed # of frames allocated to a proc. when pg fault occurs, one of the pgs must be replaced. *Global replacement*: not possible. Variable-Alloc: *Local Replacement*: the # of pg frames allocated to a proc can be changed over the lifetime of the process. Page replaced chosen from frames alloc to that process. *Global Replacement*: page replaced chosen from all available frames in main mem; causes resident size to vary.
***Fixed Alloc, Local Scope**: Need to know amount of all for a process before. If too small, high page fault rate. If too large, too few programs in main mem: increased time spent in swapping & processor idle time
***Variable Allocation Global Scope**: easiest to implement. OS maintains list of free frames. Free frame added to resident set of process when page fault occurs. If no frames available, OS must choose pg curr in mem. One way to counter potential problems is to use page buffering.
***Variable Allocation Local Scope**: when new proc loaded into main mem, alloc certain # of pg frames as resident set. When page fault occurs, select pag to replace from resident set of proc suffering fault. Reevaluate alloc provided to proc & increase/decrease to improve overall performance based on assessment of likely future demands of active processes.
Page Fault Frequency (PFF): requires use bit to be associated w/ each page in mem. Bit set to 1 when page accessed. When page fault occurs, OS notes virtual time since last page fault for that process. Does not perform well during transient periods when there's a new shift to a new locality.
Variable-Interval Sampled Working Set (VSWS): evals working set of proc @ sampling instances based on elapsed virt time. Driven by min & max duration of sampling interval, # of page faults allowed to occur b/w sampling instances.
Cleaning Policy: Concerned w/ determining when modded page should be written out to secondary memory. **Demand Cleaning**: page written to secondary mem only when selected for replacement. **Precleaning**: allows writing in batches
Load Control: Determines # of proc resident in main mem (multiprogramming lvl). Crit in effective mem mgmt. Too few procs, many occasions when all procs blocked & much time spend in swapping. Too many procs leads to thrashing.
Process Suspension: if degree of multiprogramming reduced, 1+ curr resident procs must be swapped. 6 possibilities exist: 1) lowest-prio proc 2) faulting proc 3) last proc activated 4) proc w/ smallest resident set 5) largest proc 6) proc w/ largest remaining exec window
Kernel Mem Allocator: kernel generates & destroys small tables & buffers frequently during course of exec, each requires dynamic mem allocation. Most blocks significantly smaller than typical pgs. Allocations & free operation must be made asap.

CHAPTER 9 – UNIPROCESSOR SCHEDULING

Processor Scheduling: Aims to assign procs to be exec by processor in a way that meets sys obj, e.g. response time, throughput, & processor efficiency. Broken down into 3 functions: Long-term scheduling, Medium-term scheduling, Short-term scheduling
Types of Scheduling:
***Long-term**: adds to pool of procs to be exec; determines programs admitted to sys for processing.
A reference to a memory location independent of the current assignment of data to memory is: **Logical Address**

Controls degree of multiprogramming; more procs created, smaller % of time each proc can be exec & may limit to provide satisfactory service to curr proc set: It creates procs from queue when it can, but must decide 1) when OS can take on 1+ addition procs, 2) which jobs accept & turn to procs, 2.1) FCFS, 2.2) prio, expected exec time, I/O requirements
***Med-term**: decision to add to # of proc partially/fully in main mem. Part of swapping func & determines when program brought into main mem for exec.
***Short-term**: determines which ready procs will be exec next by processor. Known as dispatcher. Exec most frequently. Makes fine-grained decision of which proc exec next. Invokes when event occurs that may lead to blocking of curr proc or may provide opportunity to preempt curr proc in favor of another. Ex. Clock interrupts, I/O interrupts, OS calls, Signals (e.g., semaphores)
~Criteria: Main obj - alloc process time to optimize certain aspects of sys behavior → set of criteria to eval sched policy. Classified into a) performance related; quantitative, easily measured (e.g. response time & throughput) or b) non-performance related; qualitative, hard to measure (e.g. predictability)
***User-oriented criteria**: relates to behavior of sys as perceived by user or proc (e.g. response time in interactive sys). Important on virtually all systems.:
Performance Related:
Turnaround time: interval of time b/w proc submission & completion. Includes actual exe time + time waiting for resources (includes processor). Appropriate measure for batch job.
Response time: time from request submission til proc begins to be received.
Non-performance related:
Predictability: a given job should run in about the same amount of time & a about same cost regardless of sys load. Wide variation in response time/turnaround time is distracting; may signal wide swing in sys workloads or need for sys tuning to cure instabilities.
***Sys-oriented criteria**: focus on effective & efficient util of processor (proc completion rate). Minor importance on single-users sys.
Performance Related:
Throughput: sched policy should attempt to max # of procs completed per unit of time. Measure of how much work is being performed. Dependent on avg proc length but also influenced by sched policy.
Processor Util: % of time processor is busy. For expensive shared sys, this is significant. In single-user sys & some others, this is less important than some others.
Non-performance related:
Fairness: in absence of guidance from user/sys-supplied guidance, procs should be treated the same & none should suffer starvation
Enforcing priorities: when procs assigned prios, sched policy should favor high-prio procs.
***I/O Scheduling**: decides which process's pending I/O request will be handled by available I/O device
Alternative Scheduling Policies

	FCFS	Round Robin	SPN	SRT	HRRN	Feedback
Selection Function	max[w]	constant	min[s]	min[s – e]	$max\left(\frac{w+s}{s}\right)$	(see text)
Decision Mode	Non-preemptive	Preemptive (@ time quantum)	Non-preemptive	Preemptive (@ arrival)	Non-preemptive	Preemptive (@ time quantum)
Throughput	Not emphasized	May be low if quantum too small	High	High	High	Not emphasized
Response Time	May be high, esp if large variance in proc exec times	Provides good response time for short procs	Provides good response time for short procs	Provides good response time	Provides good response time	Not emphasized
Overhead	Min	Min	Can be high	Can be high	Can be high	Can be high
Effect on Processes	Penalizes short procs; penalizes I/O bound procs	Fair treatment	Penalizes long procs	Penalizes long procs	Good balance	May favor I/O bound procs
Starvation	No	No	Possible	Possible	No	Possible

Selection Function: Determines which ready proc selected next for exec. May be based on prop, resource requirements, or exec characteristics of proc. If based on exec characteristics, then important quantities are w = time spend in sys so far, waiting; e = time spent in exec so far; s = total service req by proc including e.
Decision Mode: Specifies instants for selection func exercised. 2 categories: Preemptive & Non-preemptive.
Non-preemptive: one proc is running state, it will continue until it terminates/blocks itself for I/O
Preemptive: curr running proc may be interrupted & moved to ready state by OS. Preemption may occur when new proc arrives, on an interrupt, or periodically.
***First-Come-First-Served (FCFS)**: Similar to FIFO. As each process arrives, it is executed for its entire service time and then the next process is executed in order of arrival time. In the example below, A, B, C, and D arrive sequentially. So, A arrives first and executes for 3 seconds and then terminates. Since B arrives while A is still executing, B is added to the queue. Once A finishes, it is popped off of the queue and since B is at the front of the queue, it gets executed next. We do this for all of the processes, until we have none left in the queue. However, in this example, we only need to calculate Tfinish and Tr for each process. The formula is right below.
Execute FCFS for the following group of processes and complete the following table:
Tfinish = cumulative sum of Ts
Tr = Tfinish - Tarrival

Process	A	B	C	D
Tarrival	0	2	4	6
Ts(service)	3	5	4	1
Tfinish	3	3 + 5 = 8	3 + 5 + 4 = 12	3 + 5 + 4 + 1 = 13
Tr	3 - 0 = 3	8 - 2 = 6	12 - 4 = 8	13 - 6 = 7

Round Robin (RR): As each process executes it only runs for a maximum of the quantum time (in this example it is 4). If the process service time is longer than the Quantum time you still execute it, but the process is then popped, subtract the service time by quantum time, and add the process to the end of the queue. For instance, in this example, A arrives first and runs to completion since its service time is 3 which is < 4. Next, B executes only for 4 seconds, then its popped off the queue and added to the end since it has 1 second remaining.
Execute RR (Q=4) for the following group of processes and complete the following table:

Process	A	B	C	D
Tarrival	0	2	4	6
Ts(service)	3	5	4	1
Tfinish	3	13	11	12
Tr	3 - 0 = 3	13 - 2 = 11	11 - 4 = 7	12 - 6 = 6

A: 3 - 3 = 0
B: 5 - 4 = 1 - 1 = 0
C: 4 - 4 = 0
D: 1 - 1 = 0

A: 3
B: 3 + 4 + 4 + 1 + 1 = 13
C: 3 + 4 + 4 = 11
D: 3 + 4 + 4 + 1 = 12

Shortest Process Next (SPN): In this example here is the step-by-step: Process A arrives, executes, and finishes, since it passes B's Arrival Time (ARR TIME) = 1, B is added to the queue. Process B arrives, executes, and finishes. C & D are pushed onto the queue in order because they arrived during process B's execution. The processes in the queue are sorted based on shortest service time so the queue would look like so: D←C.

Process	A	B	C	D
Tarrival	0	1	5	6
Ts(service)	4	2	3	1
Tfinish	4	4 + 2 = 6	4 + 2 + 1 + 3 = 10	4 + 2 + 1 = 7
Tr	4 - 0 = 4	6 - 1 = 5	10 - 5 = 5	7 - 6 = 1

Shortest Remaining Time (SRT): preempt version of SPN. When a process arrives compare its service time with the remaining processes that are in the queue. The shortest one has priority of execution. In this example, A finishes, then B arrives, executes, then compares its remaining execution time with process C. Since 1 (B's remaining service time) < 5 (C's remaining service time), B continues and finishes, and C is added to the queue. C then executes for 1 second, then compares its remaining execution time with process D. Since, 1 < 4, C is popped and added to the end of the queue, and D executes for 1 second and completes. As of right now, we are at time 7, and C is the only process in the queue (4 seconds remaining). C executes for 1 second then is compared with E because E arrived at 8 seconds. C has a remaining time of 3 and E has a remaining time of 4, thus 3 < 4 and C continues to execute while E waits at the end of the queue. C finishes at 11 seconds, and then E executes to completion since no other processes arrive. We finished at 15 seconds. **If multiple processes have the same service time, then they execute in terms of arrival time (processes at the front of the queue go first).**

Process	A	B	C	D
Tarrival	0	2	4	6
Ts(service)	2	3	~~5~~ → 4	1
Tfinish	2	2 + 3 = 5	~~5 + 1~~ → 7 + (1 + 3) = 11	5 + 1 + 1 = 7
Tr	2 - 0 = 2	5 - 2 = 3	11 - 4 = 7	7 - 6 = 1

Highest Response Ratio Next (HRRN): chooses next proc w/ greatest ratio. Attractive bc it accounts for proc age. While shorter jobs are favored, aging w/o service increases ratio so longer proc will eventually pass shorter jobs. As each process arrives, we will execute for its entire service time but with the following process arrivals we will compare the ratios of the processes **in the queue**. Works the same as SPN, but in HRRN we compare using the "Ratio" formula below instead of comparing by service times for the processes **in the queue. When sorting, the process with the larger ratio goes ahead of the process with the smaller ratio.**

Process	A	B	C	D	E
Tarrival	0	2	4	6	8
Ts(service)	2	3	5	1	4
Tfinish	2	5	10	11	15
Tr	2 - 0 = 2	5 - 2 = 3	10 - 4 = 6	11 - 6 = 5	15 - 8 = 7

Wait = (Current Time) - Arrival
Ratio = (Wait + ServiceTime) / ServiceTime

WaitD = 10 - 6 = 4; RatioD = (4 + 1) / 1 = 5
WaitE = 10 - 8 = 2 RatioE = (2 + 40 / 4 = 1.5

Notes:
-For as long as only one process is in the system at a time we don't have to follow any ratio rules
-Choose the process with the biggest ratio

Performance Comparison: ant sched discipline that chooses next item to be served independent of service time obeys relationship: $T_r / T_s = 1 / 1\text{-}p$ where T_r = turnaround time or residence time (waiting + exec), T_s = average service time (running state), p = processor util.
Fair-Share Scheduling: based on proc sets. Each user is assigned a processor share. Obj: monitor usage to give fewer resources to users who've had more than fair share & more to those who had less than fair share.

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$
$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$
$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

$CPU_j(i)$ = measure of proc util by proc j through interval i
$GCPU_k(i)$ = measure of proc util of group k through interv i
$P_j(i)$ = prio of proc j @ beginning of interv i; low vals = high priority
$Base_j$ = base prio of proc j
W_k = weighting assigned to group k, w/ constraint $0 \le W_k \le 1$ and $\sum_k W_k = 1$

In a non-preemptive scheduling algorithm, the transition from running to ready is valid: **False**
The objective of a real-time system is to minimize the deadline of the tasks: **False**
DMA does not use interrupts: **False**
In contiguous file allocation, compaction is performed to deal with the external fragmentation problem: **True**
A reference to a memory location independent of the current assignment of data to memory is: **Logical Address**
Fair-Share Scheduling Example:

	Group 1			Group 2		
Time	Process A			Process B		
	Priority	Process	Groupt	Priority	Process	Groupt
0	45	0	0	45	0	0
1	75	30	30	45	0	0
2	59	15	15	75	30	30

1. The base priority is equal to 45.
2. The processor is interrupted 60 times per time instant (the number of counts of the process that is currently running will be increased).
3. The weight of Group 1 is equal to the weight of Group 2.
4. If the priority of the two processes is the same, you will use the lowest PID criterion (using lexicographical order).

1 sec:
60/2 = 30
60/2 = 30
45 + (30/2) + (30/2) = 75

2 sec:
60/2 = 30
60/2 = 30
45 + (30/2) + (30/2) = 75

30/2 = 15
30/2 = 15
45 + floor(15/2) + floor(15/2) = 59

CHAPTER 10 – MULTIPROCESSOR & REAL-TIME SCHEDULING

Classes of Multiprocessors:
*Loosely Coupled/Distributed Multiprocessor or cluster: consists of collection of relatively autonomous systems, each processer having its own main me,m & I/O channels
*Functionally Specialized Processors: there's master, gen-purpose processor ctrling & provide services to specialized processors
*Tightly Coupled Multiprocessor: consists processors that share common main mem & under integrated ctrl of an OS
Synchronization Granularity & Processes (**Grainsize**: *Description*; Synch Interval by # Instructions))
***Fine**: *Parallelism inherent in single instruction stream*; < 20
***Medium**: *Parallel processing/multitasking w/in single app*; 20-200
***Course**: *Multiprocessing of concurr procs in multiprogramming environ*; 200-2000
***Very Course**: *Distributed processing across network nodes to form single computing environ*; 2000-1M
***Independent**: *Multiple unrelated processes*; n/a
Independent Parallelism: No explicit synchronization among processes; each represents a separate, independent app or job. Typically used in time-sharing sys. Each use is performing a particular app, multiprocessor provides same service as multiprogrammed uniprocessor, bc >1 processor available, avg response time to users will be less.
Course & Very Course-Grained Parallelism: Synch among procs, but @ very gross lvl. Good for concurr procs on multiprogrammed uniprocessor; multiprocessor can supp w/ little/no change to user software
Med-Grained Parallelism: single app can be effectively implements as collection of threads w/in single proc; programmer must explicitly specify potential parallelism of an app & there need to be a high degree of coordination & interaction among threads of an app. Bc of various threads of an app interact so frequently, sched decisions concerning one thread may affect entire app performance.
Fine-Grained Parallelism: represents more complex use of parallelism than found in threads. Is a specialized & fragged w/ many diff approaches
Design Issues: approach taken will depend on defree of granularity of apps & # available procs. Includes a) actual proc dispatching, b) use of multiprogramming on individ processors, assgmt of procs to processors.
Assgmt of Procs -> Processors:
Assuming all processors equal, simplest to treat processors as pooled resource & assign procs on demand -> static or dynamic needs to be determined.
If proc permanently assigned to 1 proc from activation til completion, then dedicated short-term queue maintained for each processor -> adv: may be less sched func overhead -> allows group/gang sched.
Disadv static assgmt: one processor can be idle, w/ empty queue while another processor has backlog.
Approaches:
***Master/Slave**: Key kernel funcs always run on specific processor. Master schedules -> slave send service request to master. Conflict resolution simplified bc 1 process crtls all mem & I/O resources.
Disadv: master can become performance bottleneck, & master fails brings down whole sys
*Peer Architecture: kernel can exec on any processor. Each processor does self-sched fr procs pool. Complicates OS since it must ensure processors don't choose same proc & not somehow lost from queue
Process Scheduling: usually procs not dedicated to processors. A ingle queue used for all processors; if some priority scheme used, multiple queues based on prio. Sys view as multi-server queueing architecture.
Thread Scheduling: thread exec separated from rest of proc definition. An app can be set of threads that coop & exec concur in same addy space. On uniprocessor: threads can be used as program structuring aid & to overlap I/O w/

processing. In multiprocessor: threads can be used to exploit true parallelism in an app. Dramatic gains in performance possible in multiprocessor sys. Small diff in thread mgmt & sched can have impact on apps that req significant interaction among threads

Approaches:

*Load Sharing: Procs not assigned to specific processor. Simplest approach & carries over most directly from uniprocessor environ. Ex: FCFS, (preempt) smallest # threads first. Adv: load distrib evenly, no centralized scheduler reqiored. Disadv: central queue occupies region of mem that must be accessed that uses mutual exclusion -> bottlecks, preept threads unlikely to resume exec on same processor (caching less efficient), if all threads treated as common thread pool, unlikely that all will gain access to processors @ same time (may compromise performance).

***Gang Scheduling**: set of related thread sched to run on set of processers @ same time, on 1-to-1 basis. simultaneous sched of thread that make up single proc. Useful for med-grained to fine-grained parallel apps whose performance degrades when any part of app isn't running while others are ready. Benefits: sync blocking may be reduced, less proc switching may be needed, & performance will increase; may reduce sched overhead.

Dedicated Processor Assgmt: when app scheduled, each thread is assigned to a processor that remains dedicated to that thread until app runs to completion. If thread is blocked waiting for I/O or for synch w/ another thread, then that thread's processor remains idle; no multiprogramming of processors.

Defense: in highly parallel sys, processor util no longer so important as metric for effectiveness/importance. Total avoidance of proc switching during lifetime of a program should result in a substantial speedup.

Dynamic Scheduling: for some apps it's possible to provide lanf & sys tool that permit # of thread in proc to be altered dynamically, allowing OS to adjust load to improve util. Both OS & app involved in sched decisions. OS responsibility primarily limited to processor alloc. This approach > gang sched or dedicated processor assgmt for apps that can take adv of it.

Real Time Systems: OS & scheduler most important component. Correctness of sys depends on both logical computation result & time results are produced. Tasks/procs attempt to ctrl or react to events that take place in outside world & occur in "real time" -> tasks must be able to keep up.

Hard Real-Time Tasks: must meet deadline, otherwise will cause dmg/fatal sys error.

Soft Real-Time Tasks: has desirable (but not mandatory) associated deadline. Still makes sense to sched & complete task even if deadline has passed.

Periodic & Aperiodic Tasks:

Periodic: requirement may be stated as "one per period T" or "exactly T units apart"

Aperiodic: has deadline by which it must finish/start, both of which may have a constraint.

Real-Time System Characteristics:

*Determinism: how long an OS delays before acknowledging an interrupt. Operations perform at fixed, predetermined times or w/in predetermined intervals; when multiple procs competing for resources & processor time, no sys is fully deterministic. Extent an OS can satisfy requests deterministically depends on a) the speed it can respond to interrupts & b) if it has capacity to handle all requests w/in required time.

*Responsiveness: w/ determinism, makes up response time to external events; crit for real-time sys that must meet timing req from individuals, devices & external data flows. Concerned w/ how long after acknowledgment it takes the OS to service the interrupt. Includes a) amount of time required to hand interrupt & begin exec of interrupt Service Routine (ISR), b) amount of time required to perform ISR, c) effect of interrupt nesting.

User Control: much broader in real-time OS than norm OS. Allows fine-grained ctrl over task prio & allows user to specify characteristics like pagin/proc swapping, which must always stay resident in main mem, what disk algos should be used, what rights the procs have.

*Reliability: real-time sys > non-real time. Real time sys respond to & ctrl events in real time; loss/performance degradation may have catastrophic consequences.

*Fail-Soft Operation: refers to ability to fail but preserve data /7 capability as possible. Important aspect: stability. Stable if system meets deadlines of most critical high prio tasks, even if other deadlines not met.

Real-Time Scheduling approached depend on a) if sys performs sched analysis & if static/dynamic b) if result of analysis produces sched plan according to which tasks are dispatched at run time

Classes:

***Static table-driven**: performs static analysis of feasible scheds of dispatching. Result: sched that determines when task must begin execution

Static prio-driven preemptive: static analysis performed, but no sched drawn up. Analysis used to assign task prios so traditional prio-driven preemptive scheduler can be used

***Dynamic planning-based**: feasibility determined at runtime rather than offline prior to start of exec. 1 result of analysis is a sched/plan used to decide when to dispatch this task.

***Dynamic best effort**: no feasibility analysis performed. Sys tries to meet all deadlines & aborts any started proc whose deadline is missed

Deadline Scheduling: real-time OS designed w/ obj of starting real-time tasks asap & emphasize rapid interrupt handling & task dispatching. Real-time apps generally not concerned w/ speed but with completing/starting tasks @ most valuable times. Prios provide a cruel tool & don't capture reqs of completion/initiation @ most valuable time. Uses:

*Ready Time: time task become ready for execution

*Starting deadline: time task must begin

*Completion deadline: time task must be completed

*Processing Time: time required to execute the task to completion

*Resource Requirements: resources required by task while executing

*Priority: measures relative importance of the task

*Subtask Scheduler: task may be decomposed into mandatory & optional subtask

Priority Inversion: can occur in any prio-based preemptive scheduling scheme. Relevant in the context of real-time scheduling. Occurs when circumstances w/in sys force a higher prio task to wait for a lower prio task. Unbounded: duration of prio inversion depends on time required to handle a shared resource & the unpredictable actions of other unrelated tasks.

CHAPTER 11 – I/O MANAGEMENT AND DISK SCHEDULING

Categories of I/O Devices:

*Human Readable: suitable for communicating w/ computer user; printers, terminals, vid display, keeb, mouse

*Machine Readable: suitable for communicating w/ electronic equipment; disk drive, USB keys, sensors, controllers

*Communication: suitable for communicating with remote devices; modems, digital line drivers

Differences:

*Data Rate: there may be differences of magnitude b/w data transfer rates

*Application: use to which a device is put has an influence on the software

*Complexity of Control: the effect on the OS filtered by the complexity of the I/O module that controls the device

*Unit of Transfer: data may be transferred as a stream of bytes of characters or in larger blocks

*Data Representation: different data encoding schemes are used by different devices

*Error Conditions: the nature of errors, the way in which they're reported, their consequences, & available range of responses differs from one device to another.

Techniques for performing I/O:

*programmed I/O: processor issues I/O command on behalf of a proc to an I/O module; that proc then busy waits for operation to be completed before proceeding'

*Interrupt-Driven: processor issues I/O command on behalf of a proc. If nonblocking: processor continues to execute instructions form proc that issued command. If blocking: next instruction processor exec is from OS, which will put curr proc in blocked state & schedule another proc

*Direct Memory Access (DMA): DMA module controls exchange of data b/w main mem & I/O module

	No Interrupts	Use of Interrupts
I/O-to-mem transfer via processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-mem transfer		Direct memory Access (DMA)

Evolution of I/O Function: Processor directly controls peripheral device -> controller or I/O module is added -> Same config as previous step, but now interrupts are employed -> I/O module given direct ctrl of mem via DMA -> I/O module enhanced to become a separate processor w/ specialized instruction set tailored for I/O -> I/O module has local mem & is a computer in its own right.

Design Objectives:

Efficiency: major effort in I/O, important bc I/O operations form bottleneck, most I/O devices are extremely slow compared w/ main mem & processor, area that has received the most attention is disk I/O

Generality: desirable to handle all devices in uniform manner, applies to the way procs view I/O devices & the way the OS manages I/O devices & operations

Hierarchical Design: Functions of the OS should be separated according to their complexity, their characteristic time scale, & their level of abstraction. Leads to an org of the OS into a series of layers. Each layer performs a related subset of the func required of the OS. Layers should be defined s.t. changes in 1 layer don't require changes in other layers

Buffering: perform input transfers in adv of requests being made & perform output transfers some time after the request is made. Block-oriented device: stores info in blocks that are usually of fixed size, transfers made one block @ a time, possible to ref data by block #, disks & USB keys are examples. Stream-oriented device: transfers of data in & out as byte stream, no block structure, terminals, printers, comm ports, & most other devices that aren't secondary examples

Block-Oriented Single Buffer: input transfers made to the sys buffer. Reading ahead/anticipated input; done in expectation that block will be needed eventually, when transfer is complete, procs move block into user space & immediately requests another block. Generally provides a speedup compared to the lack of sys buffering.

Disadv: complicates OS logic, swapping logic is also affected

Stream-Oriented Single Buffer:

***Line-at-a-time operation** – appropriate for scroll-mode terminals (dumb terminals), user input & output are 1 line @ a time (input w/ carriage return signaling end of a line)

***Byte-at-a-time operation**: used on forms-mode terminals, when each keystroke is significant, other peripherals (sensors & controllers)

Double buffer (buffer swappig): uses 2 sys buffers. Proc can transfer data to/from 1 buffer while OS empties/fills other buffer

Circular Buffer: 2+ buffers used. Each buffer is 1 unit in circular buffer. Used when I/O operation must keep up w/ proc

Utility of buffering: technique that smooths out peaks in I/O demand; w/ without demand all buffers fill & adv lost. Where there is a variety of I/O prcs activities to service, buffering can increase OS efficiency & proc performance

Disk performance parameters: actual disk I/O operation details depend on computer sys, OS, nature of I/O channel & disk ctrl hardware

Positioning Read/Write heads: when disk driver operating, disk rotating @ const speed. To read/write head must be positioned @ desired track & beginning of desired sector of that track. Track selection involves moving the head in movable-head sys/electronically selecting 1 head on fixed-head sys. On a movable-head sys, the time it takes to position the head @ track is seek time. Time takes for beginning of sector to reach head is rotational delay. Access time = seek time + rotational delay.

FIFO: processes in seq order. Fair to all procs. Approxs random sched performance if many procs competing for disk

Priority (PRI): ctrl of scheduling is outside disk mgmt ctrl software. Goal is not to optimize disk util but to meet other obj. short batch jobs & interactive jobs given higher prio. Provides good interactive response time. Longer jobs may have to wait an excessively long time. A poor policy for database sys

Shortest Service Time First (SSTF): select disk I/O request needing least disk arm mvt. choose min seek time

SCAN (elevator algo): arm moves in 1 direction only, but after it satisfies all outstanding requests in that track direction, the direction is reversed. Favors jobs whose requests are for tracks nearest to both inner- & outermost tracks

CSCAN (circular scan): restricts to scanning one direction only. When last track visited in one direction, arm returns to opposite end of disk & scan begins again

N-Step-SCAN: segs disk request queue into subqueues of length N. subqueue processed 1 at a time using SCAN. While queue is being processed

FCAN: uses 2 subueues. When scan begins, all requests are in 1 queue w/ the other empty. During scan, all new requests put into other queue. Service of new requests deferred until all old requests have been processed

RAID (Redundant Array of Independent Disks): set of phys disk drives viewed by OS as single logical drive. Capacity is used to store parity info, which guarantees data recoverability in case of disk failure. Data distributed across phys drives of an array in a scheme known as striping.

RAID 0: not true raid since it doesn't include redundancy. User and sys data distributed across all disks in array. Logical disk divided into strips.

RAID 1: redundancy is achieved by simple expedient of duping all data. No "write penalty". When drive fails, data may still be accessed from second drive. Principal disadv is cost

RAID 2: makes use of parallel access technique. Data striping used, hamming code used. Effective choice in an environ where many disk errors occur

RAID 3: requires only single redundant disk, no matter how large disk array. Employs parallel access w/ data distrib in small strips. Can achieve very high data transfer rates

RAID 4: makes use of independent access technique. Bit-by-bit parity strip is calced across corresponding strips on each disk & parity bits stored in corresponding strip on parity disk. Involves write penalty when I/O write request of small size performed

RAID 5: like RAID4 but distribs parity bits across all disks. Typical alloc is RR scheme. Has the characteristic that loss of any 1 disk doesn't result in data loss

A block-oriented device transfers data in and out as a stream of bytes: **False**

RAID 6: 2 diff parity calcs are carried & stored in separate blocks on diff disks. Provides extremely high data availability. Incurs substantial write penalty bc each write affects 2 parity blocks

Category	Level	Description	Disks required	Data availability	Large I/O data transfer capacity	Small I/O request rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

N = number of data disks; m proportional to log N

Disk cache: buffer in main mem for disk sectors. Cache mem: used to apply to mem that is smaller & faster than main mem & is interposed b/w main mem & processor. Reduces avg mem access time by exploiting locality. Contains copy of some of sectors on disk. When I/O request made for particular sector, check is made to determine if its in disk cache. If YES, request is satisfied via cache; if NO, requested sector read into disk cache from the disk.

Least Recently Used (LRU): most common algo dealing w/ design issue of replacement strat. Block that's been in cache longest w/ ref is replaced. Stack of pointers ref cache; most reffed block is on top of stack, when block reffed/brought into cache, its placed on top.

Least Frequently Used (LFU): block that's experienced fewest refs is replaced. Counter associated w/ each block & is incremented each time block is accessed. When replacement is required, block w/ smallest count is selected.

CHAPTER 12 – FILE MANAGEMENT

Files: data collections created by users. File system is one of the most important parts of the OS to a user. Desirable properties of files: Long-term existence: files stored on disk or other secondary storage & don't disappear when a user logs off. Shareable between processes: files have names & can have associated access perms that permit ctrlled sharing.

Structure: files can be organized into hierarchical / more complex structs to reflect file relationships

File Systems: provide a means to store data organized as files as well as a collection of funcs that can be performed on files. Maintain a set of attributes associated w/ the file. Typical operations: create/del, open/close, read/write

File structure:

Field: basic elem of data. contains single value. Fixed/variable length

Record: collection of related fields that can be treated as a unit by some app program. Fixed/variable length

File: collection of similar records. Treated as single entity. May be reffed by name. Access ctrl restrictions usually apply @ file level.

Database: collection of related data. Elem relationships explicit. Designed for use by diff apps. 1+ more file types

File Mgmt Sys Obj: a) meet data mgmt needs of user b) guarantee file data is valid c) optimize performance d) provide I/O supp for variety of storage device types e) minimize potential for lost/destroyed data f) provide standardized set of I/O interface routines to user procs g) provide I/O supp for multiple users in the case of multi-user sys

Minimal User Requirements: Each user 1. should be able to create, del, read, write & mod files 2. may have ctrlled access to other users' files 3. may ctrl type of accesses allowed to files 4. Should be able to restructure files in form appropriate to the prob 5. Should be able to move data b/w files 6. Should be able to back up & recover files 7. Should be able to access their files by name rather than numeric identifier

Device Drivers: lowest lvl. Comms directly w/ peripheral devices. Responsible for starting I/O ops on a device. Procs the completion of an I/O request. Considered to be part of the OS

Basic File Sys (Phys I/O Lvl): primary interface w/ environ outside computer sys. Deals w/ block of data exchanged w/ disk/tape sys. Concerned w/ block placement on secondary storage device.

Basic I/O Supervisor: responsible for all file I/O initiation & termination. Ctrl structs that deal w/ device I/O, scheduling, & file status are maintained. Selects device where I/O is performed. Concerned w/ scheduling disk & tape accesses to optimize performance. I/O buffers assigned & secondary mem alloc @ this lvl. Part of OS

Logical I/O: enables users & apps to access records -> provides gen-purpose record I/O capability -> maintains basic data about file.

Access Method: - Level of file sys closest to user. - Provides standard interface. – diff access methods reflect diff file structs & diff ways of accessing & processing data

A physical or absolute address represents an actual location in main memory: **True**

Select the RAID level that requires 2*N disk (where N is the number of data disks): **1**

The best-fit placement algorithm (dynamic partitioning), chooses the block that is closest in size to the request: **True**

Thrashing is a state in which the system spends most of its time swapping process pieces rather than executing instructions: **True**

The priority inversion problem occurs when a low priority task waits for a high priority task: **False**

C-SCAN (disk scheduling algorithm) restricts the scanning of the tasks to one direction only: **True**

Device drivers communicate directly with the user of the computer system: **False**

In paging, given a logical address with an offset field with a size equal to 10 bit, the page size is equal to: **1K**

The resident set management combination where the page to be replaced is chosen from all available frames in main memory is: **Variable Allocation - Global Replacement**

The page replacement algorithm that looks into the future to select the page to be replaced is: **Optimal**

Select the approach to thread scheduling that carries over most directly from a uniprocessor environment: **Load sharing**

Select the I/O technique that does not use interrupts: **Programmed I/O**

Internal fragmentation is not possible on a system using simple segmentation: **True**

The placement policy (virtual memory) is an important design issue on a system using segmentation: **True**

Prepaging (virtual memory) only brings pages into main memory when a reference is made to a location on the page: **False**

Assuming that the disk head is located at track 100, select next track chosen by the shortest service time first (SSTF) algorithm: **90**

Given a system using dynamic partitioning as a memory management technique, select the free partition that is chosen by the placement algorithm for a memory request Of 16 MB: **Free Partition Size = 18 MB**

File Org & Access: - File org is the logical structuring of records as determined by the way they're accessed. – In choosing file org, important criteria: short access time, ease of update, economy of storage, simple maintenance, reliability. – Prio of criteria depends on the app using the file.

File Organization Types:

*Pile: - least complicated form of file org. data collected in order they arrive. each record consists of one data burst. purpose: accumulate data mass & save it. Record access is by exhaustive search

*Sequential: most common form of file struct. Fixed format used for records. Key field uniquely identiies record. Used in batch apps. Only org that is easily stored on both tape & disk

*Indexed Sequential: adds an index to file to supp random access. Adds an overflow file. Greatly reduces time required to access a single record. Multiple lvls of indexing can be used to provide greater access efficiency

*Indexed: records are accessed only through indexes. Variable-length records can be employed. Exhaustive index contains one entry for ever record in main file. Partial index contains entries to records where field of interest exists. Used motly in apps where timeliness of info is critical. Ex: airline reservation sys & inventory ctrl sys

*Direct/Hashed File: access directly any block of a known address. Makes use of hashing on key val. Often used where very rapid access required, fixed-length records used, records are always accessed one at a time. Ex: directories, pricing tables, schedules, name lists.

B-Trees: balanced tree struct w/ all branches of equal length. Standard method of organizing indexes for databases. Commonly used in OS file sys. Provide for efficient searching, adding & deleting of items. Characteristics:
a) a number of nodes & leaves where each node contains 1+ ey that uniquely identifies a file record & 1+ pointer to child nodes (leaves) b) each node is limited to the same # of max keys c) keys area stored in non-decreasing order; each node has 1 more pointer than keys.

B-tree is characterized by its min degree *d* & satisfies: a) every node has at most 2d – 1 keys & 2d children, or equivalently 2d pointers b) every node, except for the root, has at least d-1 keys & pointers, as a result, each internal node, except root, is at least half full & has at least d children c) root has at least 1 key and 2 children d) all leaves appear on the same lvl & contain no info. This is a logical construct to terminate the tree; the actual implementation may differ d) a nonleaf node w/ k pointers contains k-1 keys

Operations Performed on a Directory: Types of operations: search, create files, delete files, list directory, update directory

Two-Level Scheme: There is one directory for each user & a master directory. Master directory has an entry for each user directory providing address & access ctrl info. Each user directory is a simple list of the files of that user. Names must be unique only w/in the collection of files of a single user. File sys can easily enforce access restriction on directories.

A bit table (disk free space management) uses one bit for each block on the disk: **True**

In the two-handed clock page replacement algorithm (UNIX SVR4), if the front-hand finds a page with the reference bit equal to zero, then: **The reference bit remains unchanged**

Tree-Structured Directory: master directory with user directories underneath it & each user directory may have subdirectories & files as entries

File Sharing: issues that arise when allowing files to be shared among users:

*mgmt. of simultaneous access

*Access rights:

-none: user wouldn't be allowed to read user directory that includes the file.

-Knowledge: user can determine file exists & its owner & can then petition the owner for additional access rights mgmt. of simul access

-Execution: user can load & execute a program but cannot copy it

-Reading: user can read file for any purpose, including copying & execution

-Appending: user can add data to the file but cannot mod or del any of file's contents

-Updating: user can mod, del, & add to the file's data

-Changing Protection: user can change access rights granted to other users

-Deletion: user can del file from file sys

User Access Rights:

*Owner: initial creator usually, has full rights, may grant rights to others

*Specific users: individ user designated by user ID

*User Groups: set of users not individ defined

*All: all user s who have access, public files

Record Blocking: Blocks are unit of I/O w/ secondary storage. Given block size, there are 3 blocking methods:

*fixed-length blocking: fixed-length records are used, and an integral # of records are stored in a block (internal frag)

*Variable-length Spanned Blocking: variable-length records used & packed into blocks w/ no unused space

*Variable-length Unspanned Blocking: variable-length records used, spanning not employed

File Allocation: on secondary storage, file consists of collection of blocks. OS/file mgmt sys is responsible for allocating blocks to files. The approach taken for file alloc may influence approach taken for free space mgmt. Space is allocated to a file as 1+ portions (contiguous set of allocated blocks). File Allocation Table(FAT): data struct used to keep track of portions assigned to a file

Preallocation vs Dynamic Allocation: prealloc policy requires that max size of a file be declared @ the time of file creation request. For many apps it's diff to estimate reliably the max potential size of file; tends to be wasteful bc users & app programmers tend to overestimate size. Dynamic allocation allocs space to a file in portions as needed

Portion Size: In choosing a portion size there's a trade-off b/w efficiency from the POV of a single file vs overall sys efficiency. Items to be considered:
*contiguity of space increases performance, esp for Retrieve_Next ops, & greatly for transactions running in a transaction-oriented OS.
*having large # of small portions increases size of tables needed to manage the alloc info
*having fixed-size portions simplifies the realloc of space
*having variable-size or small fixed-size portions mins waste of unused storage due to overallocation.
Alternatives:
*Variable, Large Contiguous Portions: provides better performance, variable size avoids waste, file alloc tables are small
*Blocks: small fixed portions provide greater flexibility, they may require large tables/complex structs for their alloc, contiguity has been abandoned as primary goal, blocked alloced as needed

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed/Variable size portions?	Variable	Fixed Blocks	Fixed Blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation Frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	1 entry	Large	Medium

Contiguous File Allocation: Single contiguous set of blocks is alloced to a file at the time of file creation. Prealloc Strat using variable-size portions. Is best from POW of individual sequential file.

Chained Allocation: Allocation is on an individual block basis. Each block contains a pointer to next block in chain. The file alloc table needs just a single entry for each file. No external frag. Best for sequential files

Free Space Management: just as alloced space must be managed, so must the unalloced space. To perform file alloc, it's necessary to know which blocks are available. Disk Alloc Table needed in addition to file alloc table

Bit Tables: This method uses a vector containing one bit for each block on disk. Each entry of 0 corresponds to a free block & each 1 corresponds to a block in use. Adv: works well w/ any file alloc method, and it is as small as possible

Chained Free Portions: free portions may be chained together by using a pointer & length value in each free portion. Negligible space overhead bc there's no need for a disk alloc table. Suited to all file alloc methods. Disadv: leads to frag; every time you alloc a block, the block needs to be read first to recover the pointer to the new first free block before writing data to that block.

Indexing: treats free space as a file & uses an index table as it would for file alloc. For efficiency, the index should be on the basis of variable-size portions rather than blocks. This approach provides efficient supp for all of the file alloc methods.

Free Block List:
*Each block is assigned a # sequentially; list of #'s of all free blocks is maintained in a reserved portion of the disk.
*Depending on the size of the disk, either 24 or 32 bits will be needed to store a single block #; size of free block list is 24 or 32 times the size of the corresponding bit table & must be stored on disk.
*There are 2 effective techs for storing a small part of the free block list in main mem: 1.the list can be treated as a push-down stack w/ the first few thou elems of stack kept in main mem. 2. List can be treated a FIFO queue, w/ a few thou entried from both head & tail in main mem

Volumes: collection of addressable sectors in secondary mem that an OS/app can use for data storage. The sectors in a vol don't need to be consecutive on phys storage device; only need to appear that way to OS or app. Vol may be the result of assembling & merging smaller vols

Access Matrix: basic elements are a) subject: entity capable of accessing objects, b) object: anything to which access in ctrlled, c) access right: the way an object is accessed by a subject

Access Control Lists: matrix may be decamped by columns, yielding access control lists that lists users & their permitted access rights

Capability Lists: Decomp by rows yields capability tickets, which specifies authorized objects & operations for a user

Unix File Mgmt: file types:

*Regular/Ordinary: contains arbitrary data in 0 or more data blocks

*Directory: contains list of file names + pointer to associated inodes

*Special: contains no data but provides a mech to map phys devices to file names

*Named Pipes: an interprocess comm facility

*Links: an alt file name for existing file

*Symbolic links: data file containing name of the file it's linked to

Inodes: All types of UNIX are administered by OS by means of inodes (index nodes) which are ctrl structs that contain the key info needed by OS for a particular file, Several file names may be associated w/ a single inode: active inode is associated w/ exactly one file, and each file is controlled by exactly one inode.

File Allocation: done on a block basis & is dynamic, as needed, rather than using preallocation. An indexed method used to keep track of each file, w/ part of index stored in inode for the file. In all UNIX implementations the inode includes includes a # of direct pointers & 3 indirect pointers (single, double, triple)

Capacity of a FreeBSD file w/ 4Kbyte block size

Level	Number of Blocks	Number of Bytes
Direct	12	48K
Single Indirect	1024 512	4M 2M
Double Indirect	1024 x 1024 512 × 512 = 256K	1G 4 G – 4M – 48 K
Triple Indirect	512 × 256K = 128M	512G

UNIX directories & inodes: directories are structed in a hierarchical tree. Each directory can contain files and/or other directories. A directory that's inside another directory is referred to as a subdirectory

Volume Structure: A Unix file sys resides on a single logical disk/disk partition & is laid out w/ following elements:

***Boot block**: contains code required to boot OS

***Super block**: contains attribs & info about file sys

*__Inode table__: collection of inodes for each file

*__Data blocks:__ storage space available for data files & subdirectories

Access Control Lists in UNIX:

*Free BSD allows admin to assign a list of UNIX user IDs & groups to a file.

*Any number of users & groups can be associated w/ a file, each w/ 3 prot bits (read, write, execute)

*A file may be protected solely by the traditional UNIX file access mech

*FreeBSD files include an additional prot bit that indicated whether file has an extended ACL

The Translation Lookaside Buffer (TLB) is used to overcome the effect of doubling the memory access time: **True**

What is main goal of the translation lookaside buffer?

To overcome the effect of doubling memory access time in a virtual memory scheme

Main benefit of Gang scheduling? *To reduce the overhead when executing a set of related threads*

Explain priority inversion problem in the context of real-time scheduling.

It is a condition where the system forces a higher priority task to wait for a lower priority task.

What is the difference between SCAN and CSCAN (disk scheduling algos)? *SCAN satisfies all outstanding requests until it reaches last track in that direction, then reverses direction. CSCAN restricts scanning one direction only.*

Describe main goal of long-term, medium-term sched, & short-term scheduling.

Long-term: program becomes a process. Med-Term: process uses virt mem. Short-term: selects process to exe.

What is the major disadvantage of static memory partitions and dynamic memory partitions?

Static: Internal Fragmentation. Dynamic: External fragmentation

1. Select the RAID level that does NOT include redundancy:
   a) RAID 1 b) RAID 2 c) RAID 3 d) RAID 5 **e) none**
2. Select the page size of a system where logical memory address has offset field size equal to 12 bits:
   a) 4 KB b) 1 KB c) 2 KB d) none of the above
3. What happens when the back-hand checks a frame with the use bit = 1 when executing the two-handed clock page replacement algorithm (UNIX)
   a) Frame is ignored b) use bit = 0 c) frame gets replaced e) none
4. Select type of address representing mem location independent of current assignment of data to mem
   a) Logical b) Physical c) Relative d) Absolute e) none of the above
5. Select the resident set mgmt. combo that is ***NOT*** feasible
   a) Fixed Alloc / Local Replacement **b) Fixed Alloc / Global Replacement**
   b) Variable Alloc / Local Replacement d) Variable Alloc / Global Replacement
6. A fixed-length block of main mem is:
   a) Page b) Segment c) Virtual Memory **d) Frame** e) None of the above
7. What is the main objective of a real-time system ?
   a) Meeting all deadlines b) Minimizing waiting time c) Maxing CPU utilization d) None
8. Select the scheduling algorithm that ***IS*** preemptive
   a) FIFO b) FCFS **c) SRT** d) HRRN e) None of the above
9. The max size of a partition in the Buddy System mem management solution is
   a) Entire mem b) Entire mem / 2 c) Entire mem / 4 d) None of the above
10. Execute RR (Q = 3), SPN, SRT, and HRRN for the following group of processes

Process	A	B	C	D
$T_{arrival}$	0	1	2	3
T_S	2	3	4	1

11. Execute the page replacement algos FIFO, LRU, and Clock for a sys with 3 frames & the following string of page references: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2

FIFO (First In First Out)

7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2
		1	1	1	1	0	0	0	3	3	3	3
			F		F	F	F	F	F	F		

Explanation: FIFO general premise is *literally* the name; 7 was *first in*, so it's *first out* when a page needs to be replaced. You can think of it like this: Track all pages in mem in a queue whose is always length of # frame (rows); pop the oldest page in front when a new page needs to be replaced. The tracking queue when 2 is added is 7,0,1 so 7 is popped and replaced. The new queue is 0,1,2. When 3 is added, 0 is popped, so the new tracking queue is 1,2,3, etc. Fault whenever a new page gets added.

LRU (Least Recently Used)

7	7	7	2	2	2	2	4	4	4	0	0	0
	0	0	0	0	0	0	0	0	3	3	3	3
		1	1	1	3	3	3	2	2	2	2	2
			F		F		F	F	F	F		

Explanation: The key here is looking at the actual page references. Best way to memorize how many digits you look back is by the N frames and UNIQUE values. So, if you have 3 frames, like in the example, you reference the third least recently used unique value to the left. If it's 4 frames, then it's 4th unique value. For instance, 2 replaces 7 because 7 is the third least recently used unique value to the left.

CLOCK (a gray frame represents the pointer)

7^1	7^1	7^1	2^1	2^1	2^1	2^1	4^1	4^1	4^1	4^0	3^1	3^1
	0^1	0^1	0^0	0^1	0^0	0^1	0^0	2^1	2^1	2^0	2^0	2^1
		1^1	1^0	1^0	3^1	3^1	3^0	3^0	3^1	0^1	0^1	0^1
			F		F		F	F		F	F	

Clock algo Explanation:
1. only replace on a frame w/ use bit = 0 (it's basically a circular buffer, so any frame w/ use bit = 1 gets passed over. If ALL are = 1
- after replacing, SHIFT pointer down one
- while curr frame ref bit = 1, flip bit and shift pointer down one (repeat until replacement, like a clock)
2. if request number is already in a page/frame, if ref bit = 0, flip but **don't** move pointer; its moved ONLY when a new page is added.

Consider a 32-bit file sys & a 4 KB block size with an inode format that has 12 blocks for direct access, 1 block for single indirect access, 1 block for double indirect access. Calculate the following parameters *formulas are BOLD*

LEVEL	NUMBER OF BLOCKS	NUMBER OF BYTES
Direct	12 (given)	**# Blocks direct Access * Block Size** 12 blocks * 4 KB/block = **48 KB * 1024 = 49,152 Bytes**
Single Indirect	**= Block size / bit file sys** (in **bytes)** Bits -> Bytes: # bits / 8 32 bits / 8 = 4 bytes (bit file sys) 1KB = 1024 bytes, so 4KB = 4 * 1024 = 4096 bytes (block size) 4096 byte block size/4 bytes = **1024 blocks**	^^ same formula 1024 blocks * 4 KB/block **= 4096 Kbytes = 4 Mbytes** **= 4,194,304 bytes**
Double Indirect	**= (# blocks single indirect)²** $(1024 \text{ blocks})^2$ = **1,048,576 Blocks**	1,048,576 Blocks * 4Kbytes/block **= 4,194,304 Kbytes = 4 GBytes** **= 4,294,967,296 bits** = MAX FILE SIZE (Max = 2^{32} bits = 4,294,967,296 bits)
This is Extra, but Triple Indirect	**= (# blocks single indirect)³** (1024 blocks)3	THIS EXCEEDS MAX FILE SIZE!