Synchro-problems. Face them when have concurrent processes execute in an interleaving fashion, and share some modifiable resource. In more details: execution of single sequential proc is deterministic (repeated execution gives same result); execution of concurrent proc may be non-deterministic → we get a "race condition": refer to situations where execution depend on order in which the shr-resources are accessed/modified. SOLUTION is designate segment with race conditions as a CRITICAL SECTION. Critical Section. Mutex, progress, bound-wait/no-starve, independence (not block other proc). Deadlock (need mutex, hold-wait, no preemption, circular-wait), livelock (keep change state), starve.

Assembly-level. We have assembly-level/machine-level instruction [*TestAndSet*] with "parameters" [*Register,MemoryLocation*]. Behavior: we load current content at *MemoryLocation* into *Register*, AND change the value at *MemoryLocation* to (the immediate/constant) 1. The above is performed as a *single* machine operation (i.e. atomic; cannot be interrupted).

Code for Process i: $<i \in \{0,1\}$: *Want*$[i] = 1$; *Turn* $= 1$; *while* (*Want*$[1-i]$ && *Turn* $== 1$); [// CRITICAL SECTION] *Want*$[i] = 0;>$

H-L-Abstr-s. SEMAPHORES AND MUTEXES. A generalised synchro mechanism that specify behavior (not implementation). It provides a way to block a number of procs, which will be known as SLEEPING PROCESSES. We can see a semaphore S as a protected integer, with a non-negative initial value. A general semaphore or counting semaphore can have values $S \geq 0$. A binary-semaphore/mutex has values $S \in \{0,1\}$. API: {*wait*(): takes in a semaphore, if semaphore value is ≤ 0, it blocks the current proc and decrements the value, this is an atomic operation; *signal*(): takes in a semaphore, wakes up 1 sleeping proc (if any) and increments semaphore value [regardless if any sleeping process is waken up], this is an atomic operation, and never blocks}. *sem_init*(*sem_t**sem, pshared [nonzero is sharable], value); *sem_wait*(*sem_t**sem); *sem_post*(sem).

The invariant is $S_{\text{current}} = S_{\text{initial}} + \#signal(S) - \#wait(S)$, where $S_{\text{initial}} \geq 0$; $\#signal(S)$ is the number of *signal*() executed, $\#wait(S)$ is the number of *wait*() operations completed. [end, m-alloc begins]

2 assumptions to remove: 1. p-mem-space is contiguous, 2. small enough to fit. Memory Hardware. Physical mem storage: Random Access Memory (RAM) → can be treated as an array of bytes, and each byte has a unique index (known as PHYSICAL ADDRESS). The OS needs to perform following tasks: {Allocate mem-space to a new-proc; manage mem-space for proc; protect mem-space between procs; provide mem-related syscalls; manage mem-space for internal use}.

No Memory Abstraction. Pros: {straightforward, fast, addresses fixed during (i.e. unchanged after) compile time}. Cons: {Both procs assume they start at 0, resulting in conflicts; it is also hard to protect memory}. Solution 1: relocate addresses: recalculate memory references when procs is loaded into mem, e.g. if proc B is located at addr 8000, add 8000 to all mem addr-s; however, loading time is slow and it's not easy to distinguish a mem-addr from any arbitrary int. Solution 2: base + limit registers: Use a special BASE REGISTER that stores start-addr of proc's mem-space. All mem-references will be offset by this reg-value; We then use a LIMIT REGISTER to indicate the range, i.e. cannot access past the limit; However, there is a lot of overhead since we need to perform an addition and comparison per access; this is later generalised in segmentation.

Fixed-Size Allocation Algos. Phys-mem is split into a fixed num of partitions, then a proc will occupy 1 of them. Behaviors: {Internal fragmentation: when a proc does not occupy the (its) entire partition; easy to manage and fast to allocate: just give any free partition, it's all the same; need to fit largest proc: this also means ALL the smaller proc-s will waste space}.

Variable-Size Allocation Algos. Also known as dynamic partitioning. We allocate just the right amount of space needed, forming HOLES between such partitions. Behavior: {No internal frag: all proc-s get the exact space required; external frag: when holes between proc-s become unusable, waiting space, can be fixed via compaction, but slow (still better than internal, since internal is unreachable/unfixable); need to maintain more info (in the OS); slower to allocate (need to find appropriate region)}.

Dynamic/Var-Size Allocation Algos — the actual algos. Algos: {FIRST-FIT ALLOC: takes the first hole that's large enough, split it into 2, give the process the required space, and the remaining space is a new hole; BEST-FIT ALLOC: take smallest hole that is large enough, rest is same as above; WORST-FIT ALLOC: take largest hole (why? so that have lots of medium-sized holes), rest is same as above; DEALLOCATION AND COMPACTION: try to merge freed partition with adjacent holes (we can also do compaction, i.e. move partitions around, BUT compaction is expensive and should not be done frequently}. Partition Info: partition info can be maintained as either a linked list or a bitmap; for a linked-list, each node contains {1. Status: TRUE = occupied, FALSE = free; 2. Start Address; 3. Length of partition/hole; 4. Pointer to next node}.

BUDDY SYSTEM. Popular imple of multiple free lists. Buddy memory allocation provides efficient {1. Partition splitting; 2. Locating good match for a free partition (hole); 3. Partition de-allocation and coalescing}. [Contgs end here]

Paging. The physical mem is split into PHYSICAL FRAMES, and logical mem is split into LOGICAL PAGES of the same size (i.e. size of page == size of frame; so a page can be loaded into any frame). Frame size decided by hardware (mostly by MMU/memory management unit). At execution time, pages of a proc are loaded into any available mem-frame. So, logical mem can remain contiguous, while occupied physical mem-region can be disjoint (each page can be mapped into disjoint frames).

FORMULA. Phys-addr = Frame_number · sizeof(physical_frame) + offset [offset := displacement from beginning of physical frame]. Essential tricks: two impt design decisions to simplify addr translation: {1. Keep frame-size/page-size as power-of-2; 2. Phys-frame-size == Logic-frame-size}. Illustration is Logic-addr is $[p \mid o]$, p has $m-n$ bits and o has n bits, and translation mechanism maps p to f (frame number) while keeping offset unchanged. Conclusion of Addr-Translation Formula: Given page/frame size of 2^n and m bits of logical addr; for Logic-addr LA, set $p :=$ most significant $m-n$ bits of LA, $o :=$ remaining n bits of LA; use p to find frame-number f (from mapping mechanism like page-table); and finally, $PA := f \cdot 2^n + o$. Page table: to support translation, we will use a page table, which is an array where {index: page number, value: frame number}. Two tricks are employed in calculation: {keep frame/page size as (a) power of 2, keep frame and page size equal}.

Analysis of paging: {No external fragmentation (NOT POSSIBLE): no leftover physical mem-reg (every single free frame can be used); has internal frag (YES, BUT INSIGNIFICANT): when a logical mem-space is not a multiple of page size (max one page per process is not fully utilized); clear separation of logical and physical address-space: flexibility and simple translation}.

(A Pure-software) Implementation: each proc has its own page table, stored in its PCB, this PCB is in RAM. [Improved Understanding] So, this (page table) is included in the memory context of a process (or pointers to page tables; today's page tables are quite big). However, this means we require 2 RAM accesses for each mem-reference: the two accesses are {1. read the indexed page table entry to get frame number; 2. access to actual mem-item}.

Description: bottom is 2 lookups (through page-table), but top is through TLB ("page-table entries cache").

Translation Look-Aside Buffer. The TLB is on the chip, and provides hardware support for paging. Caches 4KB of page table entries (??? what does this mean ??? Done: check if have; if have then hit; if miss then take extra penalty to go through entire "normal uncached" run-through) and can cache multiple (table entries) at once; like associative cache (have to look this up again later, what does this mean; Done: see above). Fast: takes 1 ns, vs the 50 ns for RAM. ALGO: CPU ⇒ $[P \mid O]$; part of CPU is TLB + page-table; P goes through TLB (which maps some page# to frame#) else p-tb$[P]$ = frame#. $[F \mid O]$ then points to phys-mem-space.

TLB-Hit vs TLB-Miss: RAM is only accessed upon the latter, and TLB is updated after that. TLB's Impact on Mem-Access-Time: AMAT (average-mem-access-time) is $= TLBhit + TLBmiss$. Say hit rate is 90%, this equals $P(TLBhit) \cdot$ latency$(TLBhit)$ plus same thing for TLB miss. This equals (assuming 1 ns, 50 ns) equals $90\% \cdot (1ns + 50ns) = 10\% \cdot (1ns + 2 \cdot 50ns)$. Compare with no TLB: memory access two time (100 ns). Notes: latency of filling in TLB entry can be hidden by the hardware; this impact of caches is ignored.

Context switching; what happens to PTBR (Page-table Base Register): There are 3 things to note in context/process switch: {Page table: we change the pointer in a special register called "base register" from P to Q's base; Physical frames: OS need correction schemes or protection mechanisms so newly running proc Q not interfere with everything else; TLB: This is a global object. In CS2106, can assume TLB is "fully flushed" (???), as it is a part of a proc's hardware context. What does this mean? It means that if not flush, we CAN have incorrect translation (moreover, we flush to avoid correctness, security and safety issues), This flushing ("emptying" whatever was inside that belonged to previous proc) makes each proc encounter initial latency in the form of a couple of TLB misses, until its TLB "fills up".}

Protection. We can extend the paging entries to include bits to support mem-protection. Properties: {(To facilitate) Access(ing the)-right bits (a note: right here does not mean "opposite of left", right here means "privilege/power (Indonesian: hak)": on whether the page itself is writable, readable or executable, e.g. you cannot write over the text of the proc, but you can execute or read OR you cannot execute data of the proc, but can read and write; Valid bit ("OS as security guard"): some pages may be out of range for certain processes for certain reasons (OS will set these valid bits when running). If out-of-range access is done, OS will catch.} BOTH is checked in HARDWARE; and out-of-range accesses will trigger some interrupt and caught by OS to be passed/handled by some interrupt handler.

Page Sharing. We can naturally now have mutliple pages pointing to the same physical frame (??? actually yes, very magical, like pipelining but with memories and their accesses; which if not shared, will result in efficient memory usage because the physical mem-space will have multiple copies of same code across multiple processes). For example, when some library code are shared between processes. We can use a SHARED BIT in the page table entry to track whether the page is shared, OR when parent forks a child. Properties: {Shared code page: as mentioned, when there's library code, syscalls, etc; Copy-on-write: when a parent forks, the parent and child share same pages; i.e. page table is copied, but frames remain (why this happen, and what does "frame remains" mean?) This means that initially (after forking), every frame is pointing to the same frames. Using shared bit, when the child needs to update a shared page (AND only when a child *writes*), then the frame is dup-ed and page "unshared" (seems like lazy writing; only dup-ing when it is necessary). When any process tries to write (regardless of when it's parent or child), that is the moment when we make a copy and make corresponding changes. Note: this is why fork is fast! The pages are shared between multiple processes}.

Segmentation. Mem-space usually contains multiple reg(ion)-s, all with different usages in a proc. Some remain const in size (i.e. data and text), some will shrink and grow during execution time (i.e. stack, heap, library code regions). It is thus difficult to put all the regions together in a contiguous (logical) mem-space AND allow them to shrink/grow freely. Also hard to check if a mem-access is in-range.

Segmentation Scheme. Basic Idea: in Paging, mem-space considered "a single entity". We don't consider them as 4 distinct regions, we just "blindly allocate". The Motivation for this scheme is then some regions may shrink/grow during execution time. This is hard to achieve is when process uses 1 piece of memory space; e.g. at compile time, not know how much the stack (of a particular process) may/is-going-to-grow.

This scheme turns the mem-space into "a collection of segments" where each segment is mapped into contiguous physical parts (of same size). Each mem-segm has name and limit.

1. MEMORY SEGMENTS: split the regions into their own segments; {name: for reference, usually translated to an index, e.g. {Text = 0; Data = 1; Heap = 2}; base: physical base address; limit: indicate range of segment}. 2. SEGMENT TABLE: we keep a table of [Base, Limit], indexed by the name indices/segment ids (so, in my mental space, I imagine it as segtable[index] := pair({Base, Limit})). The algorithmic "execution" of a segtable: {1. With [SegId, Offset], we use SegId to get the [Base, Limit]; 2. We check if Offset < Limit, if so, seg(mentation) fault; 3. Else we access base + offset}. We store this seg-table inside registers since its size is fixed and small, AND it's (registers are) fast. The motivation for this is the segtable is equal to # of proc-s (i.e. *Segtable <<<< Page Table*). 3. ANALYSIS OF SEGMENTATION: {Independent segments: each segment is contiguous and independent, and can shrink and grow independently (i.e. not enough memory? Can just find another pace if not enough 4head!); (Unavoidable) external frag: variable size is contiguous mem-reg-s, resulting in the same problem as always; Not-the-same-as-paging: solving different problems}.

Hybrid (segment-with-paging) Most commonly used; combination of the 2. Each segment is now composed of pages and has a page table of its own (can be of different sizes). The seg-table now points to page-table address instead of base address. Page limit remains unchanged. ALGO: CPU gives $[S \mid P \mid O]$, segment-table$[S]$ points to <pg-limit, pg-tb-base>. If not addressing-error (checks with P), pg-tb$[P]$ = frame#. Put together with O. [Disj end here]

Rm assumption: {What if logical mem-space of proc is >> than physical memory?; What if same program is executed on a computer with less phys-mem?}

Basic Idea. Key observations: {1. Secondary Storage Capacity (HDD/hard-disk-drives, SDD (? I think it's SSD or solid-state-drives)) >> Physical Memory Capacity; 2. Some pages are accessed much more often than others}. So the basic idea is split logical address space into small chunks: {Some chunks reside in physical mem; other are stored on SECONDARY STORAGE}. Why useful? {separates logic-mem from phys-mem (no restrict sz logic-mem-addr); more efficient use of phys-mem (not-needed-pages can be on sec-stor); allows more processes to reside in memory}. Extended Paging Scheme. Basic idea remain unchanged: use PAGE-TABLE for Virtual/Logical → Physical address translation.

New addition: 2 page types: {Memory resident (pages in phys-mem); Non-memory resident (pages in sec-stor)}, we use RESIDENT BIT in page-table entry (indicates whether page is resident (in mem)). CPU can only access mem-resident pages: {Page Fault: when CPU tries to access non-mem res-page; So, OS needs to bring a non-mem res-page into phys-mem}.

Accessing Page X: General (6) Steps. First step (DONE BY HARDWARE): check page table "is page X mem-resident?", {Yes: access phys-mem-loc. Done; No: raise an exception!} Next (5) steps: {2. Page Fault, OS takes control; (3-6 DONE BY OS) 3. Locate page X in secondary storage; 4. Load page X into a phys-mem; 5. Update page table; 6. Go to step 1 to re-execute the SAME instruction, this time with the page in memory.}

Issues of Virt-Mem. 3 issues: {1. Elephant in the room: Secondary-Storage access time » Physical memory access time. ~ 5 orders of magnitude: milliseconds » nanoseconds; 2 (still related to 1). If mem-access results in page-fault most of the time (i.e. to load non-resident pages into memory), the entire system CAN SLOW DOWN SIGNIFICANTLY (known as thrashing); 2'. How to ensure thrashing not happen? (related to 2': how we know after page is loaded into mem, that it's likely to be useful for future accesses)}

Locality. Unlikely that page-faults occur repeatedly. Defn: most programs exhibit these behaviors: {most time spent on small part of code + accesses to (relatively) small data}. Properties: {Temporal: mem-addr has-been-used likely be-used-again; Spatial: nearby mem-addr-s likely be-used-next. They are included in 1 page.}

Virt-Mem + Loc-Princ. Both forms of locality help amortize the cost of loading the page (may be needed again after loaded, and a page that contains (many) consecutive locations that are likely to be used in future [so later access less page-fault-s]). Exceptions to this amortization: poorly-designed or maliciously-intentioned programs (i.e. those that exhibit bad behavior).

Intro: What happens if we don't do demand-paging? IF all text-/instruction + data be allocated in memory, considerations: {large startup cost if there are large num of pages to init+alloc (how to pick, if we don't load everything?); need to reduce footprint of proc-s in phys-mem SO THAT more proc-s can use mem}.

Demand Paging. OS starts with no memory-resident page only copies a page into mem IF a page fault occurs (AS OPPOSED/COMPARED TO anticipatory paging). This makes it so that un-needed pages will almost-never be loaded. Pros: {more efficient use of phys-mem; fast startup time for new process}. Cons: {proc-s may appear sluggish at start due to multiple page-faults; and those page faults may have cascading effects on other proc-s (i.e. thrashing); with large num of pages are on disk, the page tables themselves still take up a lot of space in mem; results in high overload and fragmentation (since table itself needs to occupy several pages)}.

BIG-SECTION 1. Page Table Structures. How to structure page-table for EFFICIENCY? (much bigger problem than you might realize; since we have HUGE PAGE TABLE with LARGE LOGICAL-MEMORY-SPACE).

IF we do direct-paging. Keep all entries in single page-table, and we allocate each proc this huge page table. HOWEVER, current modern-computer-systems provide huge logic-mem-space. (Current) Situation: {4GB (32-bit) was norm in past, 8TB+ is possible now; huge logic-mem-space ⇒ huge num of pages ;(indirect cause-and-effect) → (IF) each page has a page-table-entry ⇒ huge page table}. Problems (with huge page table): {high overhead, page-table alloc (page-table can occupy several frames)}.

Examples. 1: mid 1980s. V-Addr-s: 32-bits, Page-size: 4KB, $p = 32 - 12 = 20$ (the 12 is a fixed number; since 4KB = $2^2 \cdot 2^{10}$ bytes. We divide entire logical-mem-space of 2^{32} bits into 2^{12}-sized-pages. Can always find # of pages by dividing entire space by size-of-each-page). Since the "residue" p bits can (and is also an upper bound) be used to specify one unique page, so there are 2^{20} allowed page(-table-entry)s/PTE-s. So P-tab-size is $2^{20} \cdot 2B = 2MB$ (why 2 bytes per entry? 1 byte is not enough since there is extra info besides just frame no; e.g. valid bit, access-right-bits etc; nowadays, range from 4-8 bytes, more often, we see 8-byte p-table entries).

2. Today's midrange laptop: V-Addr-s: 64-bits ($16*1024$GB of V-Addr-space!), Page-size: 4KB (12 bits for offset, as in mid-1980s model), Physical memory 16GB with P-Addr-s = 4+10+10+10 = 34-bit. How many V-pages: $2^{64}/2^{12}$ PTE entries, How many P-pages: $2^{34}/2^{12}$ PTE entries, in reality, PTE size is 8B (with other flags). So, p-table-size: $2^{52} * 8B = 2^{55}$B per process! Compare with phys-mem-size 2^{34}B.

2-Level-Paging! Observation: process may not use entire v-mem-space; so full-page-table is a waste! Idea: page the page table:{Split whole page table into regions, only a few regions are used (as mem-usage gros, new region can be alloc-ed); this is similar to splitting logic-mem-space into pages; AND we need a "directory" to keep track of the regions (analogue to the relationship between page-tables and (real/actual) data-pages}. ALGO: V-Addr is $[page-dir\# \mid page\# \mid offset]$. Traverse twice to get frame, then phys-addr is $[frame\# \mid offset]$.

Description: {split page-table into smaller page-tables, each with a page-table-num; if original page-table has 2^p entries: {with 2^m smaller page-tables, m bits needed to uniquely identify (one page-table), each smaller page-table contains 2^{p-m} entries}; to keep track of smaller page-tables: {a single page-directory is needed; page-dir contains 2^m indices to locate each of smaller page-table}}.

Advantages. If V-Addr 32-bit, P-size 4KB = 2^{12}B, PTE sz = 2B, p-dir = $2^{20}/(2^{12}/2)$, overhead = 1KB + total "direct" page tables (total: 13KB [if 3 p-tab-s] vs 2MB [direct]). So, adv-s: {enables p-tab-struct to grow beyond 1-frame-sz (not need to be contiguous in mem); can have empty entries in p-dir}. Disadvantages/Problems. Requires TWO serialized memory access JUST to get frame number (one access for directory, another for page-table, and only then can access data; and that doesn't count if page-table is on virtual-mem-space — this is possible by tutorial question). TLB-as-a-solution? Since it eliminate page-table accesses? HOWEVER, TLB-miss-es experience longer page-table-walks (traversal of page-tables in hardware). Hierarchical p-tab: radix-tree structure; n-levels meaning including root (as first) (can have empty & inval entries (too)).

Inverted Page-Table. ALGO: CPU ⇒ $[PID \mid P \mid O]$, trav one-by-one until get entry $< PID, P >$ (say i), then frame no is $[I \mid O]$. Basic Idea: {p-table is a per-process info (if M proc-s in mem, there are M indep p-tables); observations: {only N mem-frames can be occupied, and out of M page tables, only N are valid; so it's a huge waste, sincd $N << M$ (page tables) idea: keep single mapping of [phys-frame → (pid, page#)], pid = proc-id, page# logic-page-num corresponding to the proc (note: page# not unique among proc-s, another-note: pid + page # can uniquely identify a mem-page)}.

Lookup: {Normal-page-table: PTE-s ordered by page-num (look-

up page X: access X^{th} entry); inv-page-table: PTE-s ordered by frame-num (lookup page X: need search whole table)}. Pros/-Cons: {Adv: huge saving (one table for all (maybe insanely many) proc-s); Dis: slow translation}. Application: in practice, inv-tables often used as "auxiliary structure" (e.g. to answer questions like: who are (all) sharers of phys-frame X?).

BIG-SECTION 2: Page-Repl Algos. Which page shld be replaced (evicted) when needed (i.e. when a page-fault occurs)? When a page is evicted: {if clean(-page) (i.e. not-modified) → no need to write-back; else dirty(-page) (i.e. modified) → need to write-back}. Algos to find a suitable repl-page: OPT, FIFO, LRU, Second-Chance (clock) [only talk abt these 4 in CS2106]. Modelling mem-ref-s: in actual mem-ref [logic-addr = page-num + offset]; however, for studying (page-repl-algo) purposes, only page-num is important. Therefore, to simplify discussion, mem-ref-s are often modeled as mem-ref-strings (i.e. a sequence of page-num-s).

Evaluations of Page-Repl-Algos. Mem-access time $T_{\text{access}} = (1 - p) * T_{\text{mem}} + p * T_{\text{page-fault}}$, with p is probability of page-fault (others are self-explanatory; T stands for (access) time). Since $T_{\text{page-fault}} \gg T_{\text{mem}}$, need to reduce p to keep T_{access} reasonable. (Good algo should minimize total-num of page-faults + be fast!)

1. OPT (optimal page repl). General idea:{replaces page that WILL NOT be needed again for the LONGEST PERIOD of time; guarantees minimum num of page-faults (try to prove it! Proof-sketch: if replace a more-upcoming-page that is to-be-used in near(er)-future, additional page-faults can occur)} Problem: Not realizable, need future knowledge of mem-ref-s. Still useful: as a base of comparison for other algo-s. 2. FIFO (page repl). Details of imple: {OS maintains queue of resident page-num-s; remove first page in queue if repl is needed; update queue during page-fault-trap}. No need hardware support. Problems: Belady's anomaly (exhibits unintuitive behavior: ↑ frames → ↑ page-fault-s, while good algo "should" not downgrade with more RAM (i.e. bad performance). Example: {1,2,3,4,1,2,5,1,2,3,4,5}; 3 frames gives 9 PFs, 4 frames gives 10 PFs. 3. LRU (least-recently-used). General idea: makes use of temporal locality: {honor the notion of recency; replace the page that hasn't been used in longest time ("least recent" one)}. Another way to put this assumption/filtration-algo is "we expect a page to be reused in a short time window" (practical temporal-locality); so the contrapositive is "have not been used for some time → most likely will not be used again". Good results (not suffer Belady's). LRU imple details: not easy, need to keep track of "last access time" somehow. Hardware support: 2 options. A. "Counter" (logical-time ctr-s, entry has "time-of-use" field); prob: need search all pages, possible overflow. B. "Stack" (stack of p-num-s); prob: not pure stack (can remove anywhere), hard to imple in hardware. Second-Chance Page Repl (a.k.a. CLOCK). General Idea: {Modified FIFO, to give a second chance to pages THAT ARE ACCESSED; each PTE now maintains a ref-bit: {1: accessed since last reset; 0: not accessed}}. Algo: {1. Oldest FIFO page is selected (victim page); 2. if ref-bit == 0 → page is repled. Done, algo exits; 3. If ref-bit == 1 → page is skipped (i.e. given a 2^{nd} chance): {ref-bit cleared to 0; effectively resets arrival time → page taken as newly-loaded' next FIFO page (victim) is selected (i.e. go to Step 2 but for next page after this)}}.

BIG-SECTION 3: Frame Allocation. How to distribute limited phys-mem-frames among proc-s? (Assume, for notation, N phys-mem0frames, M proc-s competing) Simple: equal alloc (N/M frames) or proportional alloc (each proc $p$get size$_p$/size$_{\text{total}} * N$ frames).

Assumption lifting. Implicit assumption for page-replo-algo-s so far: victim pages ARE SELECTED AMONG PAGES OF THE PROCESS that causes page-fault (local replacement). If victim page can be chosen AMONG ALL PHYS-FRAMES, proc P can take (a) frame from proc Q, evicting Q's frame during replacement (global replacement).

Pros/Cons: {Local:{Pros: num of frames alloc to a proc remains constant → performance stable between multiple (re)runs; Cons: if alloc-ed frames not enough → hinder progress of a proc}; Global:{Pros: allow self-judgement betwen proc-s, proc needs more frame can get from others that need less; Cons: badly behaved proc-s steal frames from other proc + num-frames alloc-ed to proc can be different on rerun}}.

Thrashing, in relation to Local/Global Repl. If thrashing, need heavy I/O to bring non-resident pages into RAM (to be resident). It's hard to find right # of frames. If global is used: {thrashing proc "steals" page from other proc; causes other proc to thrash (cascading thrashing)}. If local is used: {thrashing can be limited to that (one) proc; BUT that proc can use up I/O bandwidth + degrade performance of other (affected) proc-s}.

The observation that "set of pages referenced by a proc is relatively constant in a (particular) period of time" gives the defn of a working set. However, as time passes, this set can change (different prog phases require different data). Example: function is executing (references likely on local variables that define function's locality) and terminates (ref-s change to another set of pages (abruptly!!!)).

Working Set Model: monitors mem-usage patterns of proc-s and adjusts frame-alloc-s based on working-set of each proc. Define work-set-window Δ as a particular interval of time (say, interval of 5 mem-ref-s), and $W(t,\Delta)$:= active pages in the interval at time t. The algo allocates enough grames for pages in $W(t,\Delta)$ to reduce possibility of page-fault. [mem ends, f-sys begins]

(So, OS helps the) f-sys (to) provides: {abstraction on top of physical media (portability!); high-level resource-management scheme (persistent, not volatile); protection between proc-s and users; sharing between proc-s and users}}. Mem-mngmt vs f-mngmt: {storage in RAM vs Disk; access-spd: const vs variable (disk I/O time); unit-of-addr-ing: phys-mem-addr vs disk-sector; usage: implicit when proc runs vs explicit; organization: paging/segmentation (determined by OS) vs many different FS (Linux: ext*, Win: FAT*, Mac OS: HFS*)}.

File-System General Criteria: self-contained, persistent (beyond OS lifetime), efficient. Note: when we use program, we have "no choice" but to invoke RAM (implicit usage) (since RAM is necessary addr-space); if we want to use data from stored memory (i.e. file), need to explicitly declare our (intention to) use.

Basic Defn. Represents a logical unit of info created by process. An abstraction (essentially an Abstract-Data-Type); files are (+ contains) a set of common operations with various possible imple. Contains (i) data: info structured in some ways; (ii) metadata (f-attributes): additional associated info. Metadata. {name; id (used internally by FS); type; sz; prot; time + date + owner info (creation, last modification time, owner id, etc); table of content (info for FS to determine how to access f)}. Name. Different FS has different naming-rule (to determine valid f-name). Common: {length; case sensitive; allowed special symbols; file extension}. Common f-types: {reg-files (contains user info (ASCII + binary files); directories (sys-files for FS structure); special files (char/block-oriented)}. Operations on F-metadata: {rename (change f-name); change attr-s (f-access-permissions, dates, ownership, etc); read-attr (get f-creation-time)}.

What is "perm-b-s"? Classify users into 3 classes: {Owner (user who created f); group (set of users who need similar access to a file); universe (all other users in system)}. Unix: define permission of 3 access types (r,w,x) for the 3 classes of users. F-Protection (Access Ctrl List): minimal (same as perm-b-s), or extended (added named users/group). Illustration: [*$getfacl$* (f-name) command to get ACL info, this specifies {perm for specific user (*user* : *sooyj* : *rwx*); specific/general group; upperbound (*mask* :: *rwx*)}]

File Data. Structure (possibilities): {Arr-of-bytes (each byte has unique offset from f-start. We talk about offset to reach the n^{th} byte here); fixed-len-records: {array of records, can grow/shrink; can jump to any record easily (offset of n^{th} record is size-of-rec $*(n-1)$)}; var-len-records (flexible but harder to locate a record)}. Access Methods: {sequential-access: data read in order (can't skip but can be rewound); random access: data can be read in any order. Can be provided in 2 ways: {*read*(*offset*): every read operation explicitly state the position to be acc-ed; *seek*(*offset*): a special operation, provided TO MOVE to a new loc in f}; direct-access: used for f that contains fixed-len-records. Allows random access to ANY record DIRECTLY. Very useful where there is a large amount of rec-s. Basic random access method can be viewed as a special case (of this method), where "each-record == one byte"}.

File Operations. As syscalls, OS provides f-oper-s as syscalls (provide protection, concurrent, and efficient access). Unix-syscalls Header Files: *#include* <*sys/types.h*>, *#include* <*sys/stat.h*>, *#include* <*fcntl.h*>. General info: {opened-f has an id (file-desc, which is an integer). This is used for other operations; file is acc-ed on a byte-to-byte basis (no interpretation of data)}.

File Information in the OS. Info (that is) kept for an opened-file: {F-ptr (keep track of current position within a file); f-descriptor (unique identifier of the f); disk location (actual f loc on disk); open count/reference count (how many proc has this f opened?), this is useful to determine when to remove entry in table}. Since several proc-s can open same file + several diff files can be opened at any time, the COMMON APPROACH is to USE 3 TABLES: {per-process open-file table: {to keep track of open files for a proc, each entry points to the system-wide open-file table entries}; system-wide open-file table: {to keep track of all open files in the system, each entry points to a V-node entry}; system-wide v-node (virtual-node) table:{to link with the f on phys-drive; contains info about phys-loc of the f.}}. Sys-wide open-f-tab: {op.type; f-offset; v-node; ref-count}, sys-wd v-nod-tab: {type; fn-ptr-s; I-Node; ref-cnt}.

Dir. used to: provide logical grouping of files (user view of directory) + keep track of files (actual system usage of directory). Single-Level. Self-explanatory. Unix: hard-link not allowed for directories. What is Unix Hard-Link (for DAG)? Consider: dir A is owner of file F + dir B wants to share F. Hard-link: {A and B has separate ptr-s point to actual file F in disk; pros: low overhead, only ptr-s are added in dir; cons: deletion problems (e.g. if B deletes F? If A deletes F?)}. Unix Command: "*ln*". In Unix: symbolic link IS ALLOWED to link to directory, so general graph CAN BE CREATED.

UNIX Context. *open*() takes in *char**path and *int* flags (many options can be set; using bit-wise-OR: {read, write, or read+write mode; truncation, append mode; create file (if not exists); many, many more :)}). Returns file descriptor (fd) integer (−1 failed to open file, ≥ 0: unique index for opened file). By convention: default fd-s: {STDIN (0); STDOUT (1); STDERR (2)}.

Open existing file for read only: *fd = open*("*data.txt*", *O_RDONLY*); Create file if not found, open for rd+wr: *fd = open*("*data.txt*", *O_RDWR* | *O_CREAT*);

read(): {fn-call: *int* read (*int* fd, *void**buf, *int* n); purpose: reads up to n bytes from curr-offset into buffer *buf*; returns num of bytes read (can be $0 \ldots n$), if $< n$: EOF is reached; param-s: {*fd*: file-descriptor (must be opened-for-read); *buf*: an array "large-enough" to store n bytes}; *read*() is sequential read, starts at curr-offset and increments offet by bytes read}.

write(): {fn-call: *int* write (*int* fd, *void**buf, *int* n); purpose: writes up to n bytes from curr-offset into buffer *buf*; returns num of bytes read (can be $0 \ldots n$), otherwise −1: error; param-s: {*fd*: file-descriptor (must be opened-for-write); *buf*: an array at least n bytes with values to be written}; POSSIBLE ERRORS: {exceeds file-size limit; quota; disk space; etc}; *write*() is sequential write: {starts at curr-offset and increments offet by bytes written; CAN increase file size beyond EOF (by appending new data}}.

lseek(): {fn-call: *off_t* lseek (*int* fd, *off_t**offset, *int* whence); purpose: move current position in file by *offset*; returns current offset in file (≥ 0), otherwise −1: error; param-s: {*fd*: file-descriptor (must be opened); *offset*: positive = move forward, negative = move backward; *whence*: point-of-reference for interpreting the *offset*: {SEEK_SET: absolute offset (count from file start); SEEK_CUR: relative offset, from curr-position (+/-); SEEK_END: relative offset from EOF (+/-)}}; can seek anywhere in file, even beyond end of existing data}.

close(): {fn-call: *int* close (*int* fd); returns 0: successful, or −1: error; param-s: (only) *fd*: file-descriptor (must be opened); after/with *close*(): {*fd* no longer used anymore; kernel can remove associated data-structures; indentifier *fd* can be reused later}; by default: process termination automatically closes all open files}.

More Overview. File-sys-s are stored on storage media (so, we use data in media to utilize the media). We concentrate on hard disk (in this lecture). General Disk-Structure: {1-D array of logical blocks; logical blocks := smallest accessible unit (usually 512B-4KB); logical block is mapped into disk-sector(s) (the layout of disk-sector is HARDWARE-DEPENDENT)}. Note: partition in windows is similar to your "C:" and "D:" drives in windows. If you want, each partition may have its own OS as well. Another: depending on imple, the three middle-sections may be combined. Also, file info is minimally metadata. Generic Disk Organization: {Simple Boot Code; Partition Table} in MBR, {OS Boot Block; Part-Details + Dir-Struct + F-s-Info; F-Data} in Partition(s).

Disk Organisation: {MBR/Master-Boot-Record at sector 0 ("small area in front of partition table"), with partition table; followed by 1 or more partitions (each partition can contain an independent file system)}. A FS generally contains: {OS Boot-up info; partition details: {total num of blocks; num and location of free disk blocks}; dir-structure; info of files; actual f data}.

File Imple Overview. A file can be logically viewed as a collection of logical blocks. When file size != multiple of logical blocks, last block may have internal fragmentation. A good f-sys imple must: {keep track of the logical blocks; allow efficient access; effective utilization of disk space} ⇔ the difference of file systems is their focus on "how to allocate" file data to logical blocks.

Contiguous. allocate consecutive disk blocks to a file. Pros: simple to keep track, fast access; Cons: external fragmentation, file size need to be specified in advance. LinkedList: {keep a linked list of disk blocks; each disk block stores: {the next disk block number (i.e. stored as and act as a pointer); actual file data}; f-info stores first and last disk block num}. Pros: solve fragmentation problem; Cons: {random access in a f is very slow (i.e. "a lot of moving around"); part of disk block is used for ptr; less reliable (what if 1 of those pointers is incorrect?)}. (note: can be optimized, by adding additional ptr-s)

FAT16/32. {move all the block ptr-s into a single table (known as FAT, which stays in memory at all time); simple yet efficient}. FAT allocation table details. FAT entry contains either: {FREE code (block is unused); block-num of next block; EOF code (i.e. NULL ptr to indicate it's the last block); BAD block (unusable block, i.e. disk error)}. FAT version 16 implies 2^{16} disk blocks; version 32 implies 2^{32} disk blocks. Example: block 3 → 5 → 8 → EOF. Pros: faster random access, since now the linked list traversal takes place in memory; Cons: FAT keeps track of all disk blocks in a partition, {can be huge when disk is large, and so consume valuable mem space (expensive overhead)}.

Can do better: allocate a memory block for "positions where (currently) open files are stored", known as *index table*. Why? Don't need every file information at every time, just need those of which are open in RAM. (from what I'm catching, every file has a (unique) index block). Idea: {instead of keeping track of every f in the system, we maintain blocks for each file; each f has an index block: {array of disk block addresses; with *IndexBlock*[N] == N^{th} Block address}}. Pros: {less(er) memory overhead, since only index block of opened file needs to be in mem; fast direct acess}; Cons: {limited max f size, since "max num of blocks" == "num of index block entries"; (existence of) index block overhead}.

Variation (schemes to allow larger file): {linked scheme: {keep a linked list of index nodes; cons: expensive since traversal cost (can be large)}; multilevel index: {similar idea as multi-level paging; can be generalized to any num of levels}; combined scheme (used in Unix, as Indexed-Node or I-Node): {combination of direct indexing and multi-level index scheme; fast access for small files (use direct index); handle large files (use multi-level)}}.

I-Node Data Block Example. Every f/dir will have a I-node associated with it (uniquely-numbered I-node). Design of I-Node allows fast access to small file AND flexibility in handling huge file. The actual example: e.g. consider 4-bytes block addr + 1KB disk block (i.e. 1 disk-block is sized 1KB). We assume "main directory of disk blocks" has 12 pointers (i.e. 12 direct blocks) + single, double, triple indirect blocks (how big of a file can be stored in a single structure of single/double/triple indir-blocks?)

Dir-blocks: 12 × 1KB = 12 KB. Single indirect block: {num of entries in a disk block = $2^{10}/4$ (i.e. 4B addresses, so total/4B addresses) which is equal to 256 entries; 256 × 1KB = 256KB}. Double indirect block: 256^2 × 1KB = 64MB. Triple indirect block: 256^3 × 1KB = 16GB.

Free Space Management. Overview (of "Partition Details" section of the slides) here. To perform file alloc: need to know which disk block is free. Free space management: {maintain free space info; allocate: {remove free disk block from free space list; needed when f is created/enlarged (i.e. appended)}; free: {add free disk block to free space list; needed when f is deleted/truncated}}.

Bitmap: each disk block is represented by 1 bit (in slides: free blocks are "green-colored and 1-labeled", occupied are "red-colored and 0-labeled"). Pros: provide a good set of manipulations; Cons: this bitmap needs to be kept in memory (otherwise, hardware access is not efficient) (though, a note on the overhead: nowadays, the size of your hardware can be HUGE

How to do better? Can think of the same argument as with indexed-allocation: "do I need ALL the data/information of which memory-spaces are free (at all times)? i.e. do I need every free block which is present in my memory?" Response: just need a small set of which blocks are free, and when they are exhausted, bring up the next set of free blocks.

Linked List: each disk block contains {a number of free disk-block num-s, or; a ptr to the next free space disk block}. Pros: {easy to locate free block; only the 1st ptr is needed in mem (though other blocks CAN be cached for efficiency)}; Cons: high overhead (we still need to store these in our "partition detail" section; to reduce overhead: can allocate the free blocks themselves to store this "what are the next free blocks" info).

Directory Structure Overview. 2 main tasks of dir-struct: {1. keeps track of files in a directory (possibly with f metadata); 2. map f-name to the f-info}. Details: {f must be opened before use (use *open*("data.txt")); the purpose of this operation is to locate f-info using pathname + f-name ("+" as in appended)}. How does the *open* thing work (???) given a full pathname, need to recursively search the dir-s along path to arrive at the f-info " *dir2/dir3/data.txt* " (note: sub-dir usually stored as f-entry with special type, in a directory).

Linear List. Each entry of linear list represents file (f-name + metadata, f-info or ptr to f-info). Locate file using list (linear search; inefficient for large dir-s and/or deep tree traversal). Common sol: use cache to remember latest few searches. Hash Table. Each dir contains a hash table, size N. To locate f by f-name: f-name is hashed into indek K, $0 \le K \le N-1$. *HashTable*[K] is inspected to match f-name. Pros: fast lookup; Cons: limited-size hashtable; depends on good hash fn. File Information. Consists of f-name + metadata and disk-blocks info. 2 common approaches to store: store everything in dir entry OR store only f-name, and points to some other data structure for more info (i.e. first approach but abstracted/lazily evaluated). Note: reserved (bits) is if need to extend some part of the directory entry. [Structure!] First number is I-Node num, second is offset/"ptr" to next entry, 3rd is F/dir/subdir, 4th is f-name size.

File Systems In Action. CREATE file operation (walkthrough). To create file *.../.../parent/F*: {Use full pathname to locate parent dir (search for filename F to avoid duplicates (if found, f-creation terminates with error) and search could be on cached dir structure); use free space list to find free disk block(s) (depends on alloc scheme); add entry to parent dir (with relevant f-info: f-name, disk-block info, etc}.

OPEN file operation (walkthrough). When proc P open file *.../.../.../F*: {Use pathname to locate file F (not found → open operation terminates with error); when F located, its f-info loaded to new entry E in sys-wide-table and V in I-node-table (if not already present, do this for each of the two); creates an entry in P's table to point to E (i.e. f-descriptor) AND ptr from E's entry to V (same thing, but this is a bridge from second table to third table); return f-desc of this}. Note that the returned f-desc is used for further r/w operation(s).