TUTORIAL 5

Demand Paging

Varun Gupta

MP2?

Mini-Project 2
by Akhila Matathammal - Wednesday, 1 October 2025, 7:17 PM

Dear All,

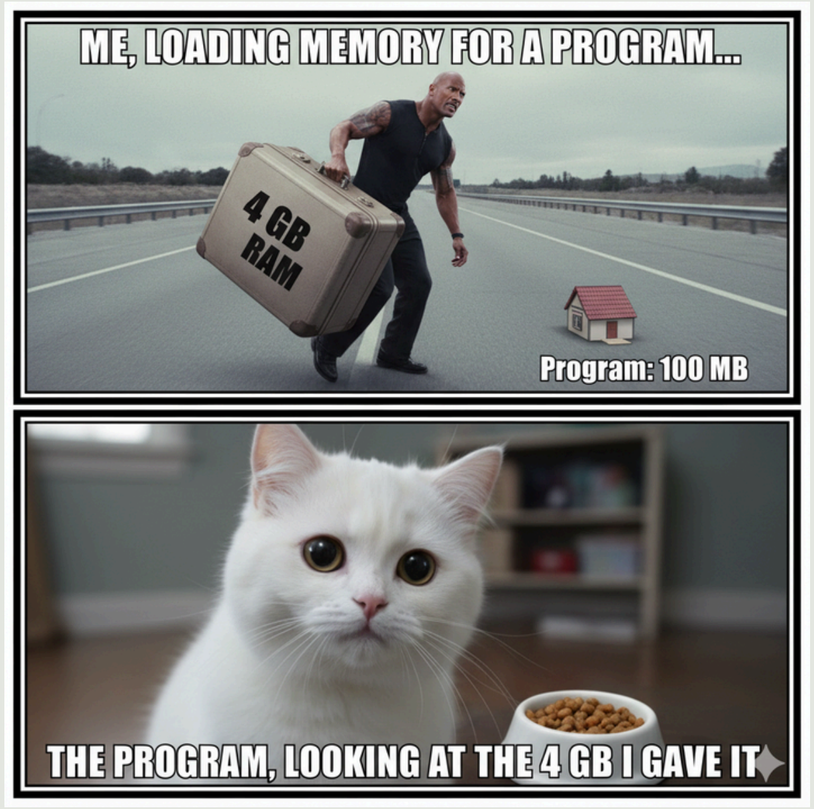
Due to the course logistics and upcoming evaluation schedules, we will not be able to accommodate or respond to any new doubts raised after October 11th regarding Mini Project 2.

Please make sure to clarify all your queries well before the deadline. We encourage you to use the discussion forums and contact the TAs before the cutoff date to ensure smooth progress.

Thank you for your understanding and cooperation.

MOTIVATION

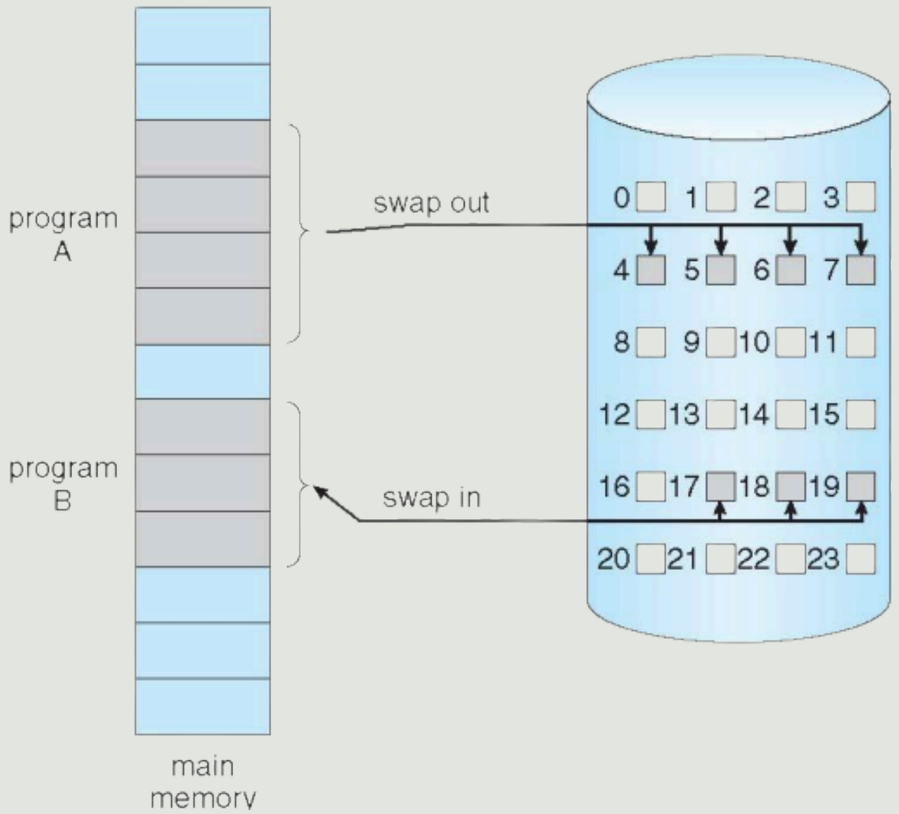
- Large Programs and limited Physical Memory
- Programs rarely use all code/data
- Loading everything → wasted RAM
- Partially loaded execution?
- How VM is implemented(in practice)?
- XV6?



DEMAND PAGING

- Bring a page into memory only when it is needed (Lazy / On-Demand)
- Similar to paging system with swapping
- Page is needed → reference to it
  - invalid reference → abort
  - not-in-memory → bring to memory
- How its done?
- Missing Page → raise page fault
- Detect Missing Page?

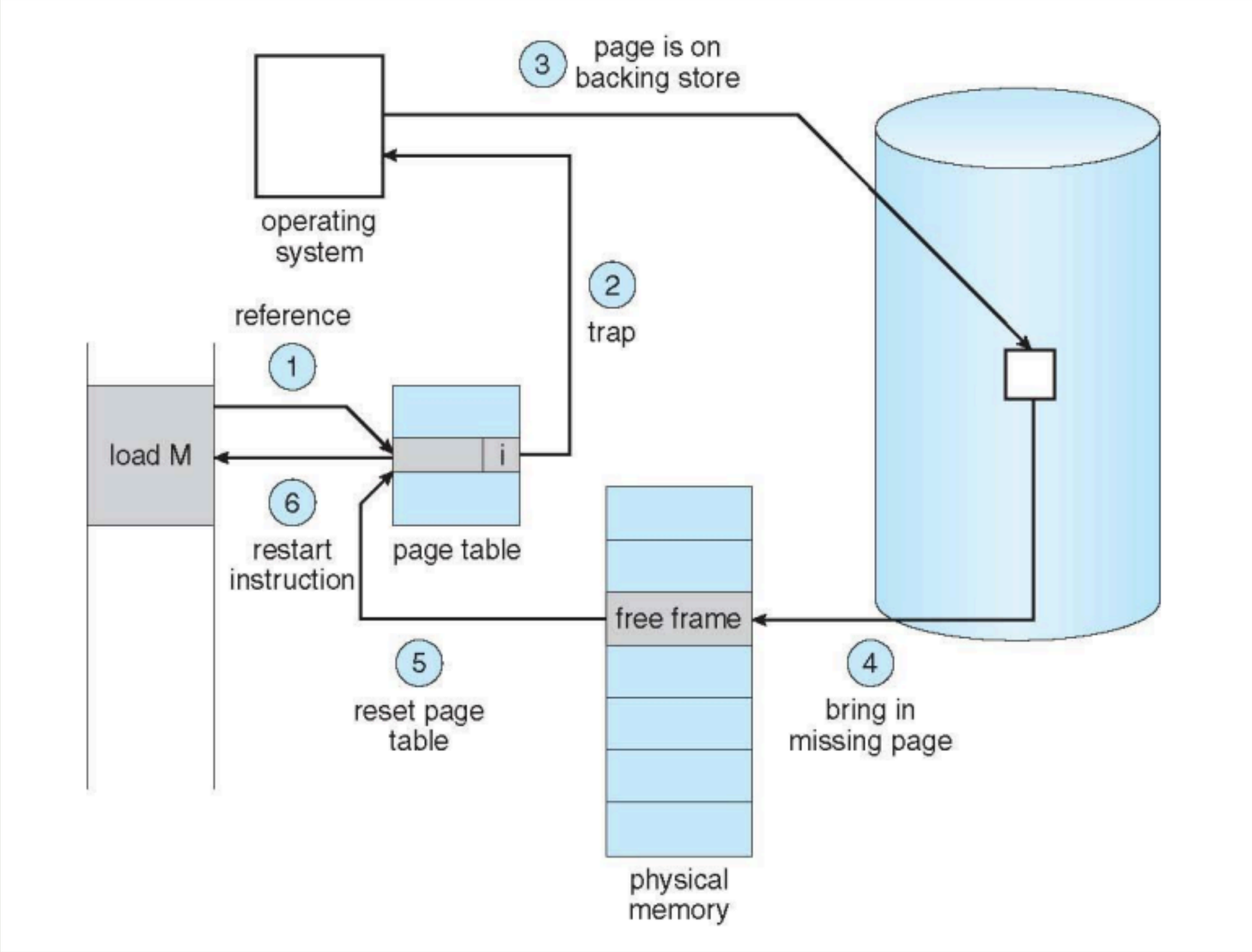
VPN	PFN	Valid	Prot	ASID



PAGE FAULT

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
- Operating system need to decide:
  - Invalid reference → abort
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory
- Restart the instruction that caused the page fault

PAGE FAULT


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

ISSUES?

- Valid and Invalid Address?
  - Need to check if faulting VA lies in code, data, heap, stack, or swapped pages. (check assignment spec)
  - Requires process metadata tracking.
- What if No Free Page?
  - If kalloc() fails → need replacement policy.
  - Must choose a victim page (Page Replacement Algorithm)
- Cost of page fault?
  - Servicing a fault means disk I/O → ~100,000× slower than RAM.
  - Too many faults = thrashing.
  - is it Worth it?

PERFORMANCE

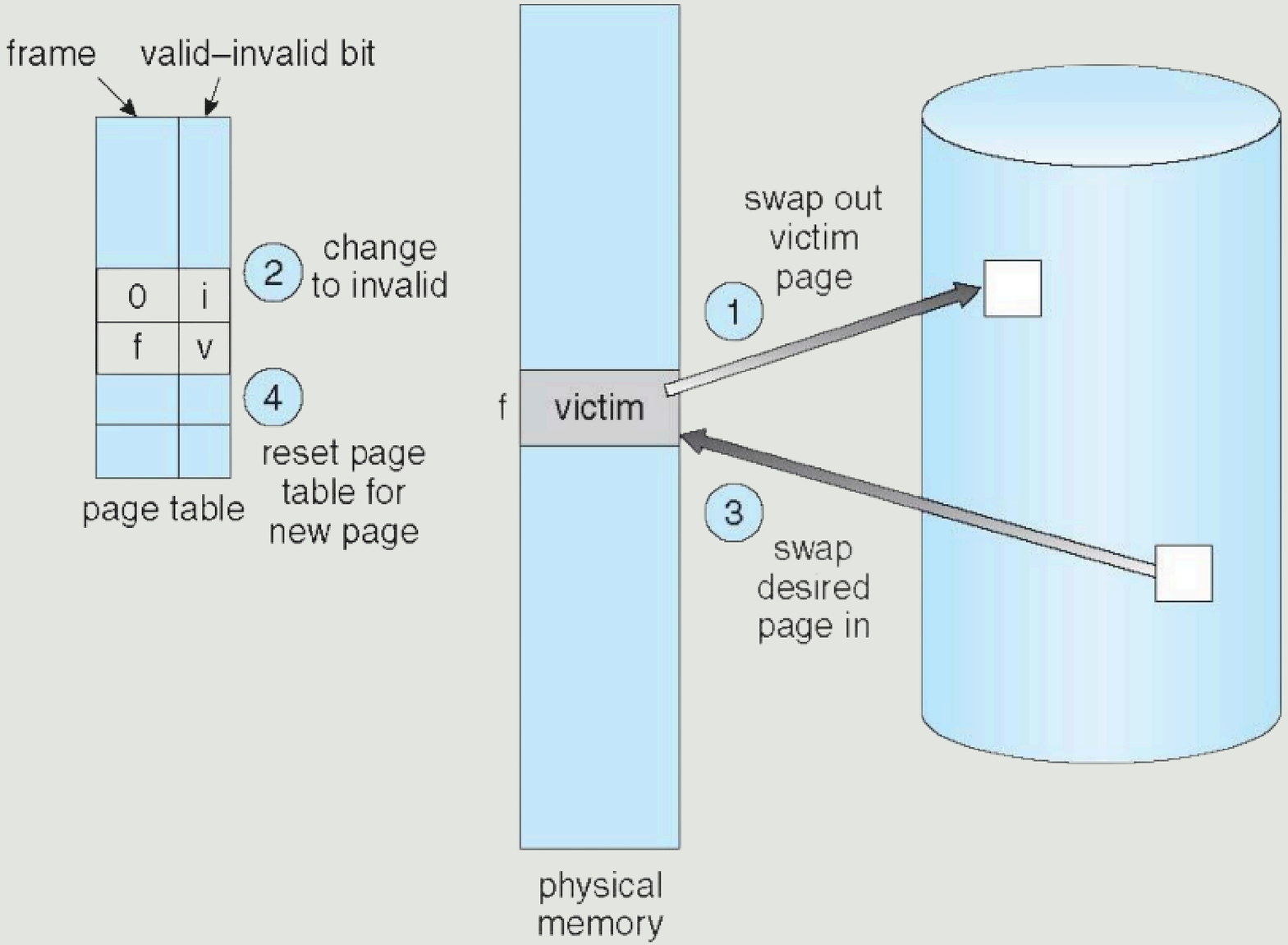
Critical Areas:-

- Service the interrupt(quick)
- Read the page (major)
  - Device Queue
  - Seek + Rotational Latency
  - Transfer (DMA)
- Restart the process (small)

- Page Fault Rate $0 \leq p \leq 1$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

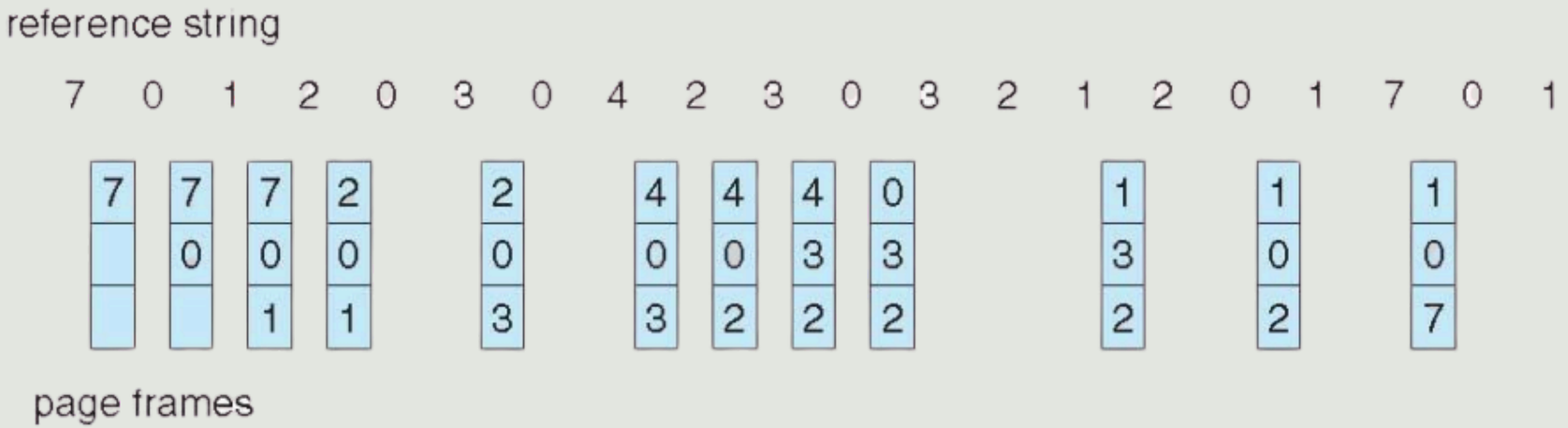
$$EAT = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

NO FREE FRAME?



LRU ALGORITHM

- Use past knowledge rather than future(not like optimal algo)
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

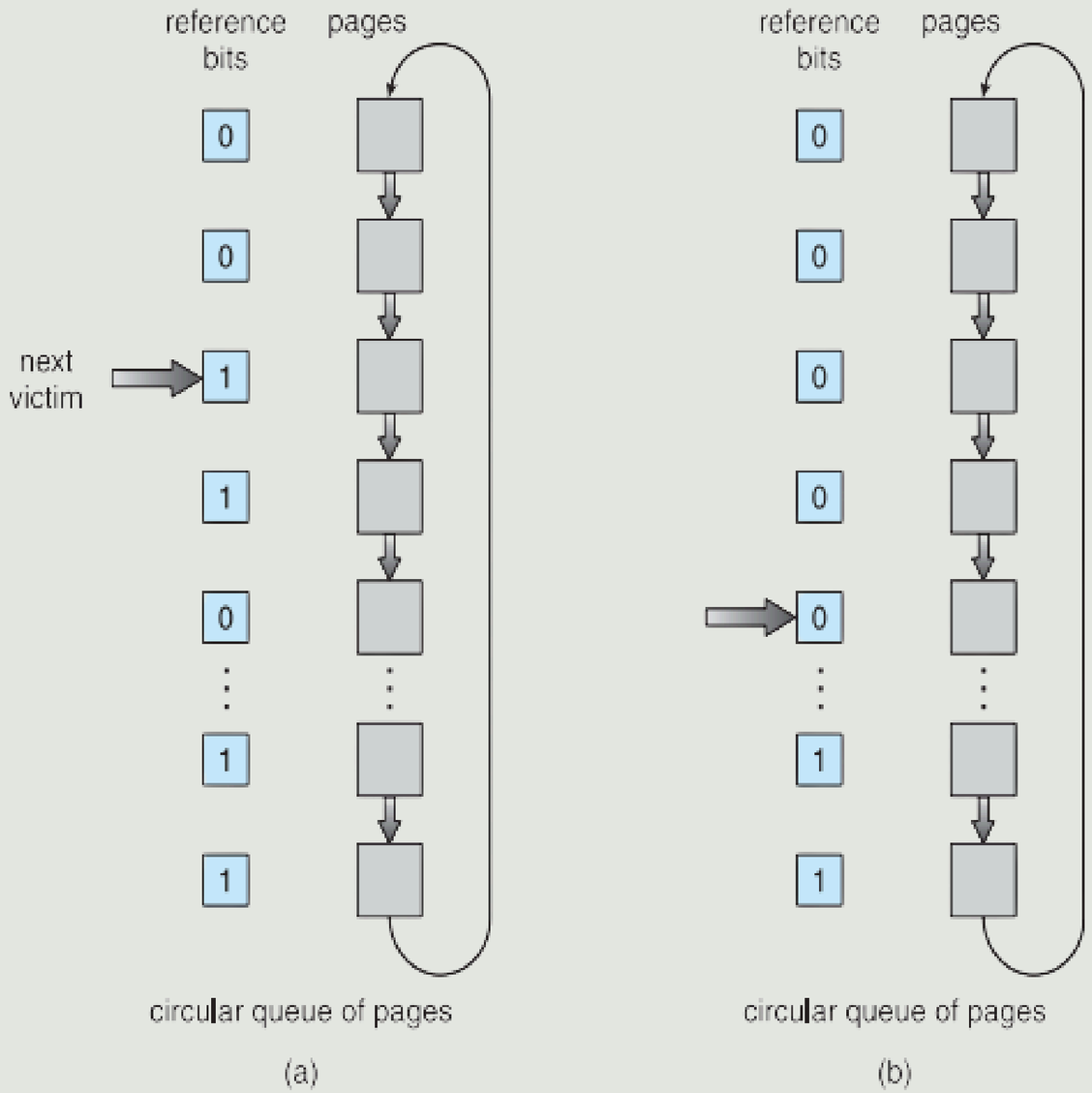


- FIFO: 15 and LRU: 12 but still more than optimal(9)
- Implementation?

IMPLEMENTATION

- Counter Based
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced: move it to the top
  - requires 6 pointers to be changed
  - But each update more expensive; No search
- Approximations?
  - Using Ref bit(no ordering?)
  - Second Chance (Clock Based - Circular Queue)

SECOND-CHANCE (CLOCK)



Okay, but what if the page I evict is dirty? Where do I put it?

SWAPPING

- When a page is evicted:
  - Clean page → just discard (can reload later).
  - Dirty page → must save contents to disk.
- Saved pages go to a swap area on disk.
- Later, if the process needs that page again → swap it back in.
- Why you need?
- Without swapping:
  - Evicting dirty pages = lost data (program correctness breaks).
- With swapping:
  - Process can continue seamlessly.
  - Virtual memory illusion is maintained.

IMPLEMENTATION

- Swap Area Design?
  - A dedicated swap partition or swap file Challenges?
  - Assignment: per-process swap file
  - Swap space divided into slots (fixed size)
- Page Eviction
  - Check if the victim is clean or dirty:
    - Clean page → no need to write; just discard.
    - Dirty page → must save it to disk
- For Dirty Page:
  - Find a free swap slot in process's swap file.
  - Write page contents to that slot.
  - Update process metadata: map VA → slot ID.
  - Mark the page table entry invalid

IMPLEMENTATION (CONTD...)

- Swap-In
  - Page fault handler checks metadata: was this VA swapped out earlier?
  - If yes:
    - Allocate a free physical frame (or evict another page first).
    - Read contents back from swap slot into RAM.
    - Free the swap slot (mark available again).
    - Update page table → mark valid.
- Managing Swap Slots
  - Need a bitmap/array to track free vs. occupied slots per process.
  - On swap-out: allocate a free slot.
  - On swap-in: free the slot.
  - If no slot available → log SWAPFULL and kill process.

IMPLEMENTATION (CONTD...)

- Process Lifecycle and Swap Cleanup
  - On process exit:
  - Delete its swap file.
  - Free all slots associated with it.
  - Ensures no data leakage between processes
- Challenges in Implementation
  - Slot Management: prevent overwriting an occupied slot.
  - Consistency: keep VA ↔ slot mapping in sync with pg table
- Edge Cases:
  - Swap file full → kill process.
  - Evicting pages that are in use by kernel/I/O → forbidden.
  - Correctly handling multiple processes with their own swap files.

THRASHING

- Thrashing occurs when a process spends more time servicing page faults than executing actual instructions.
- Caused by insufficient frames for the process's working set.
- Leads to very high page fault rate → CPU utilization drops drastically.

Working Set Model

- Working set, $W(t, \Delta)$: set of pages referenced by a process in the most recent Δ references (time window of size Δ).
- Working Set Size $wssi(t) = | Wi(t,\Delta) |$
- Total Demand $D(t) = \sum_{i=1}^{n} wssi(t)$
- If $D(t) > \text{Available frames}$ → Thrashing otherwise Safe

OS Decision Making

- If thrashing risk detected:
  - Reduce degree of multiprogramming (swap/suspend a process).
  - Or allocate more frames to processes with larger working sets
- If Safe allow all the processes to run.

HOW FAR FROM REAL OS?

- We built: lazy allocation, demand paging, FIFO replacement, swapping.
- Real OS have:
  - Multi-level replacement (Clock-Pro, Aging).
  - Demand zero + demand paging + copy-on-write.
  - Shared memory + mmap.
  - Huge pages (2MB/1GB).
  - Transparent swapping policies.

REFERENCES

Questions?