CS 347M (Operating Systems Minor)

Spring 2022

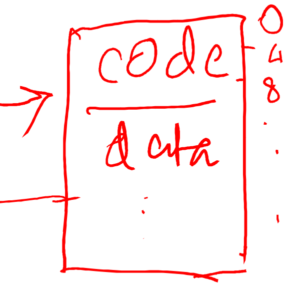
Lecture 9: Demand Paging

Mythili Vutukuru

CSE, IIT Bombay

Recap: Virtual addresses and paging

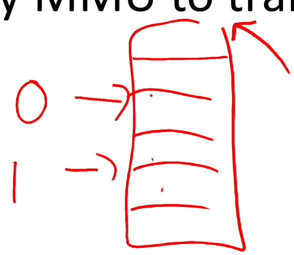
CPU PC eax

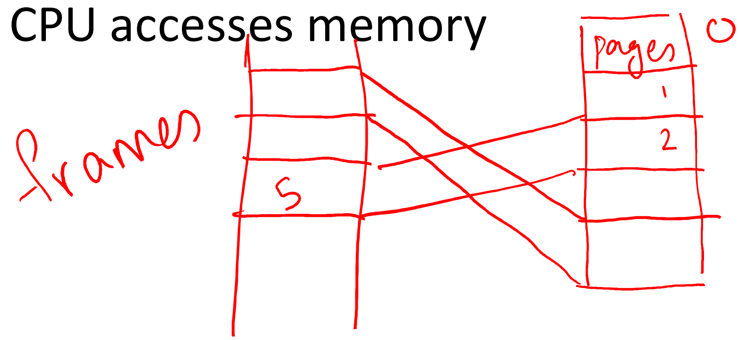


- Instructions and data of a process in memory assigned virtual addresses
  - Starting at 0 for user code (OS code also assigned high virtual addresses)
- Virtual address space of a process divided into fixed size logical pages, stored in a fixed size physical frames in memory
  - Prevents external fragmentation, cannot prevent internal fragmentation
- Page table maps logical page numbers to physical frame numbers
  - One per process, maintained by OS as part of PCB
  - Used by MMU to translate VA to PA when CPU accesses memory

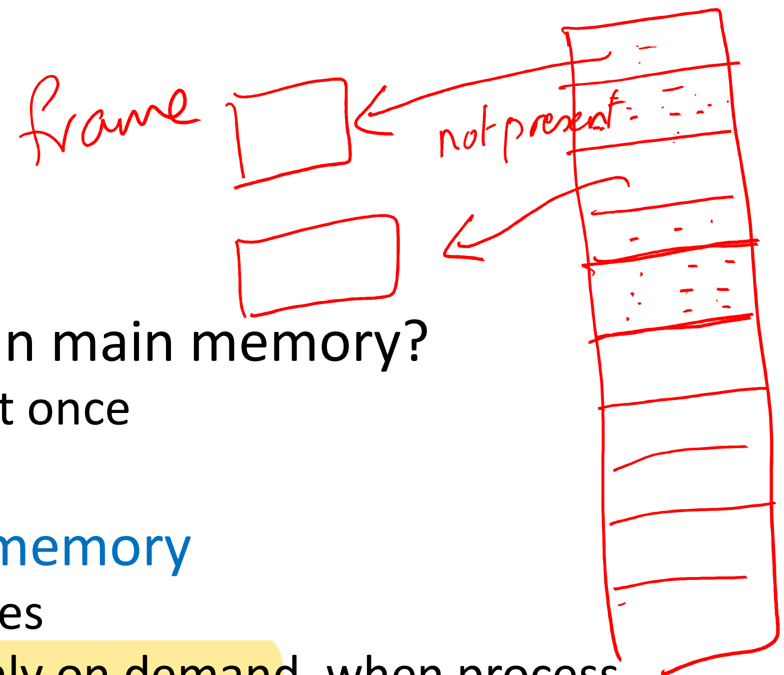
VA - PA

TLB



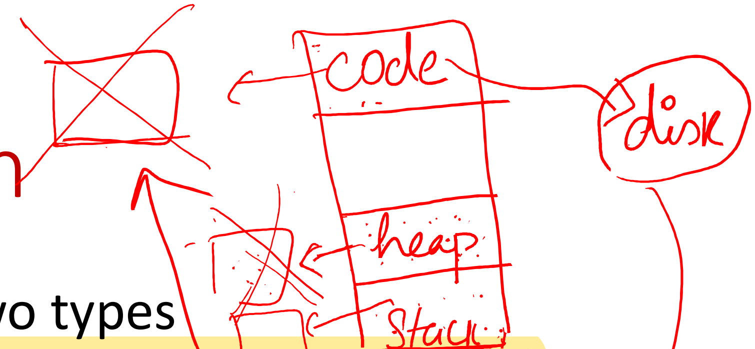


Demand Paging

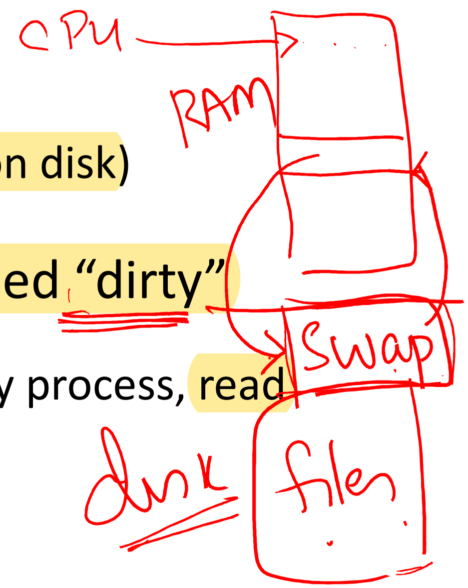


- Are all pages of all active processes always in main memory?
  - Not necessary, as process will not use all of it at once
  - Not possible, with large address spaces
- Modern operating systems provide virtual memory
  - Not all logical pages are assigned physical frames
  - OS allocates physical frames to logical pages only on demand, when process accesses the memory contents of the page
  - OS can reclaim some physical memory of process that is not in active use
  - For some pages in page table, physical frame number is not stored, page table entry is marked as "not present"
- Virtual memory of processes can be much more than physical memory in the system, OS overcommits memory

On-demand memory allocation

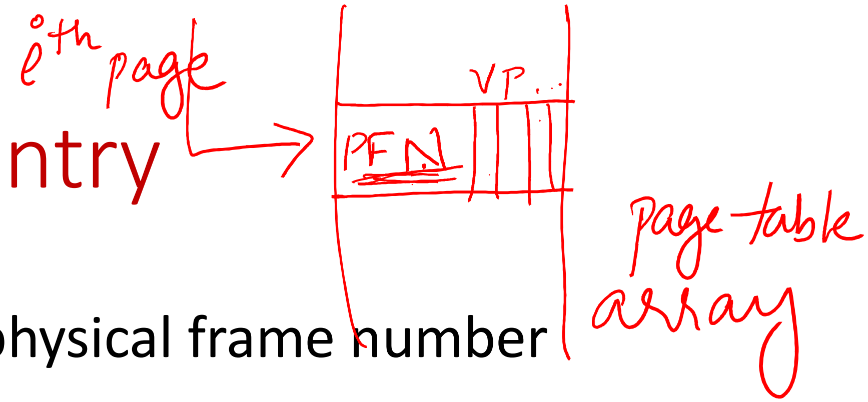


- Pages in the memory image of a process are of two types
  - File-backed pages contain data from files on disk (e.g., page with executable code)
  - Anonymous pages are not backed by files on disk (e.g., pages containing stack, heap)
- On-demand memory allocation: allocate when accessed for first time
  - File-backed page content copied from disk on first access
  - Anonymous pages allocated empty physical frames on first access
- How to reclaim unused memory from processes? active
  - File-backed page content can be deleted from memory (copy exists on disk)
  - Cannot simply delete content of anonymous pages (data can be lost)
- Swap space: space on hard disk used to store copies of modified "dirty" anonymous pages (different from file storage)
  - Dirty anonymous pages are written to swap space when not in use by process, read back from swap into main memory when required

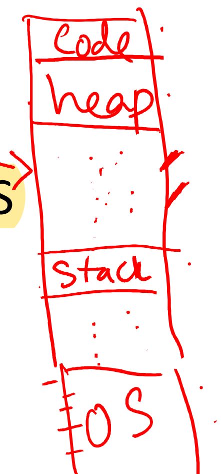


how many frames? over commit

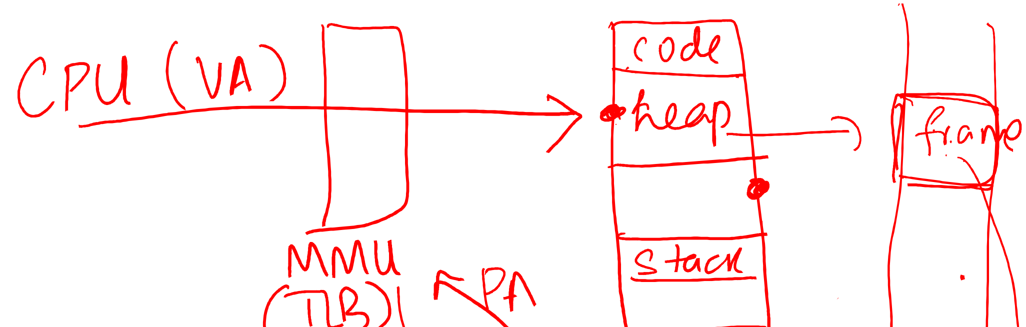
Information in page table entry



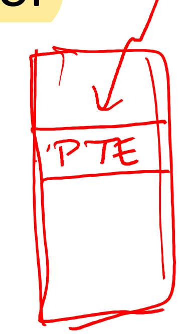
- Page table entry maps page number to physical frame number
- Page table entry also contains several bits to indicate status of page
  - Valid bit indicates if the page is in use by process in its virtual address space (invalid addresses should never be accessed by a process)
  - Present bit indicates if a physical frame number is assigned to the logical page
- Process accesses page with
  - Valid bit not set → illegal memory access (segmentation fault)
  - Valid bit set, present bit not set → OS has not allocated memory yet, or OS has reclaimed memory of this page (content in disk / swap space)
- Other bits set by MMU: dirty bit, accessed bit (more later)



Page fault handling (1)



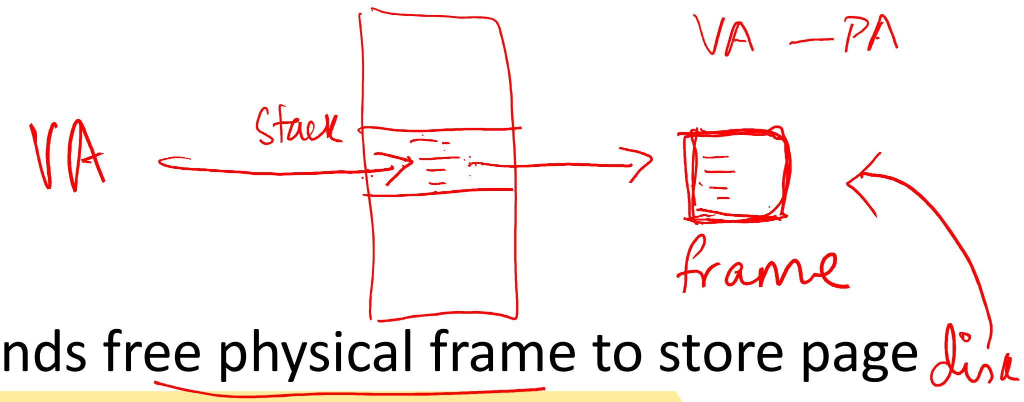
- When MMU walks page table to translate a virtual address to physical address, the various bits in page table entry are also examined
- MMU traps to the OS (page fault) in case of any unexpected behavior
  - Illegal access, e.g., process tries to write to a read-only page
  - Invalid access, e.g., process tries to access an entry with valid bit not set
  - Valid bit is set but present bit is not set, e.g., due to on-demand memory allocation not done yet by OS



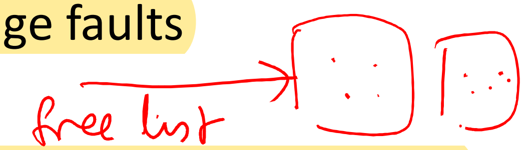
traps
syscalls
intr
page fault

- Trap instruction is invoked, OS code to handle page fault runs
  - For illegal/invalid accesses, OS may terminate the process when servicing the page fault
  - If the virtual address is valid but not present in physical memory, OS will service page fault by assigning a free frame to the page

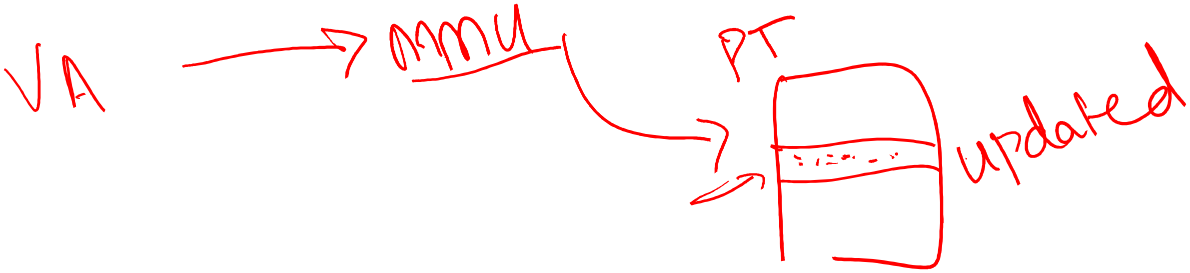
Page fault handling (2)



- If page not assigned a frame, OS finds free physical frame to store page
  - OS maintains list of free physical frames to assign during page faults

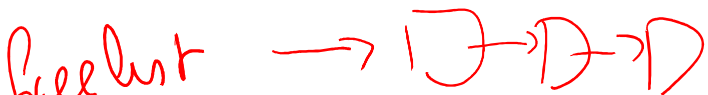


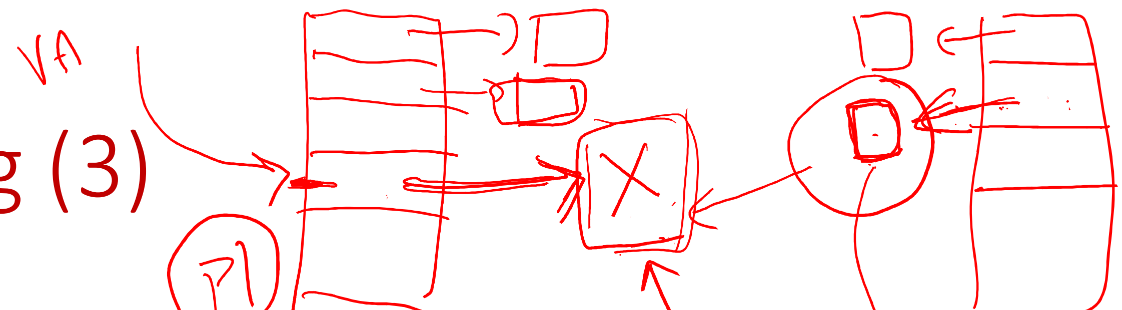
- Next, OS fills the physical frame with page contents
  - If page is in swap space or file-backed, contents are read from disk into free page
  - Reading the page contents from disk may block the process
- Once physical frame is ready with page content, OS updates page table entry (add new physical frame number), updates MMU, process restarts
  - Hopefully runs correctly this time and MMU doesn't raise trap



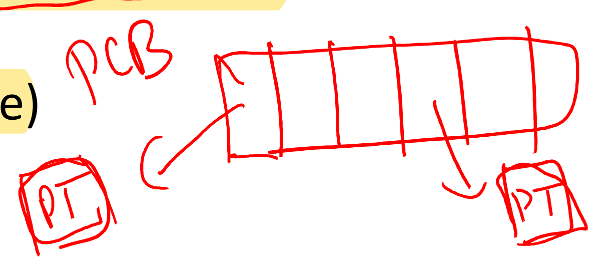
TLB flush

Page fault handling (3)

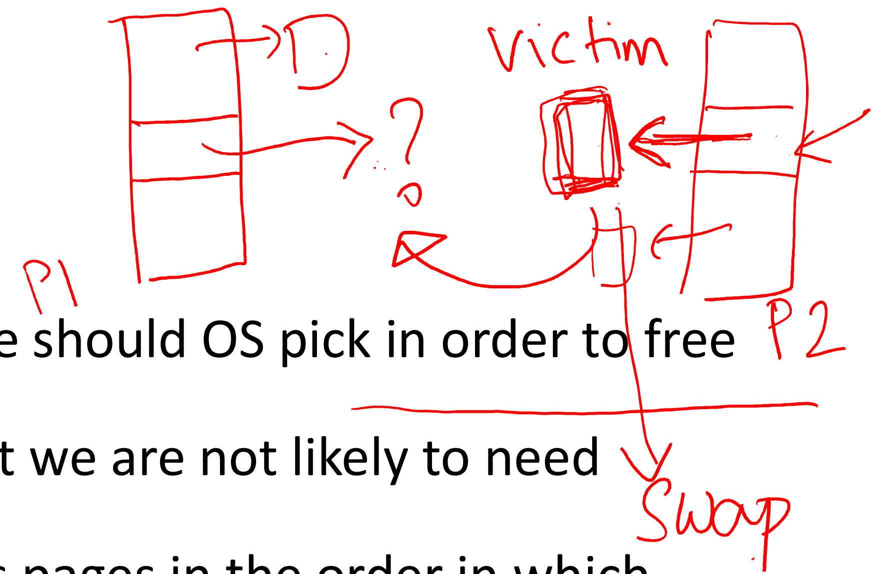




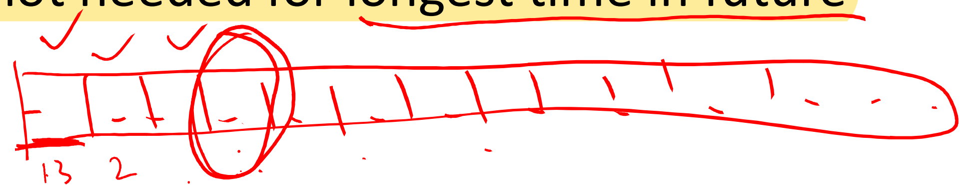
- What if all physical frames are occupied and OS cannot find free physical frame to store page?
  - If there are no free physical frames, OS can evict a victim page (i.e., take away a physical frame assigned to another page) to free up a physical frame
  - Page replacement policy helps OS identify victim pages
  - OS writes victim page contents to swap space (in case of modified "dirty" anonymous page)
  - Page table of victim process updated to delete old mapping
  - Victim page can belong to same process or another process
- Servicing page fault may potentially involve two disk accesses
  - Store old contents of victim page to disk (if dirty)
  - Read new contents of page from disk (if not empty page)



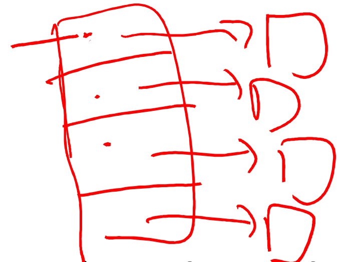
Page replacement policies



- Page replacement policy: which victim page should OS pick in order to free up a physical frame?
- Goal: Minimize page faults, evict pages that we are not likely to need immediately
- Simple policy: First In First Out (FIFO) evicts pages in the order in which they have been assigned frames
  - May be suboptimal, e.g., the first assigned pages may be important pages that are in use very often, leading to another page fault in near future
- Most commonly used policy: evict the Least Recently Used (LRU) page
  - Page has not been used for sometime now, so less likelihood that it will be immediately used in future
- Ideal optimal policy: replace page not needed for longest time in future (not practical!)



Example: Optimal policy



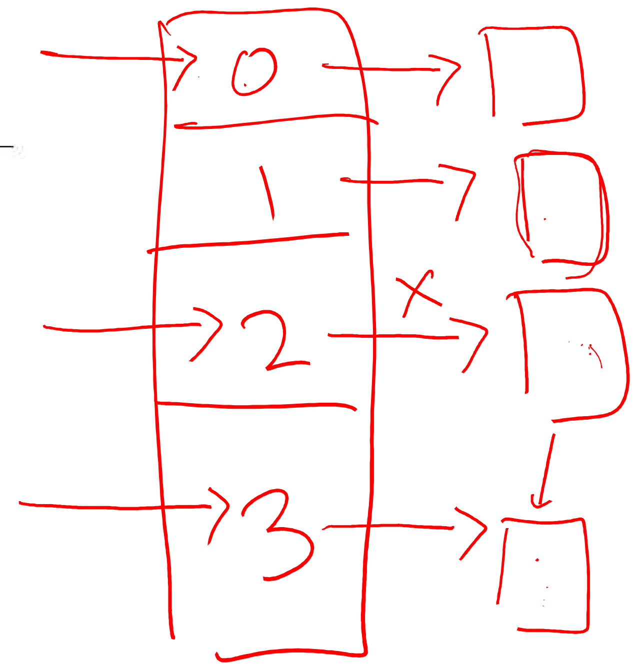
- Example: Process accessed 4 pages (0,1,2,3), only 3 physical frames in memory
- First few accesses are cold (compulsory) misses

references to memory

pagefault

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**



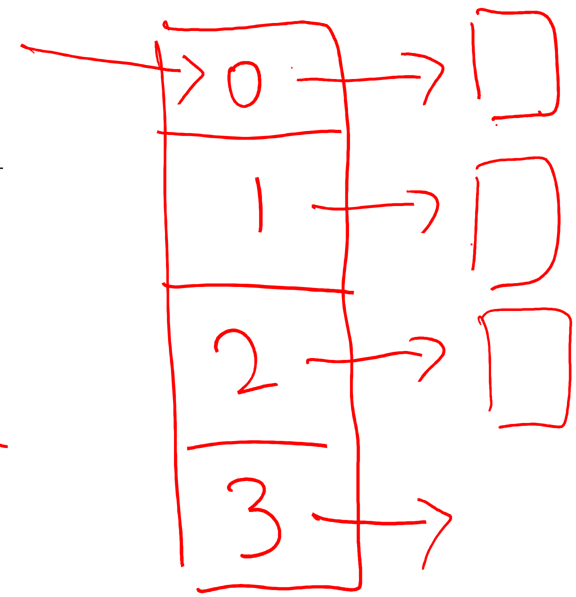
Example: FIFO

- Usually worse than optimal
- Belady's anomaly: performance may get worse when memory size increases!

reference

memory

Access	Hit/Miss?	Evict	Resulting ~~Cache~~ State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2



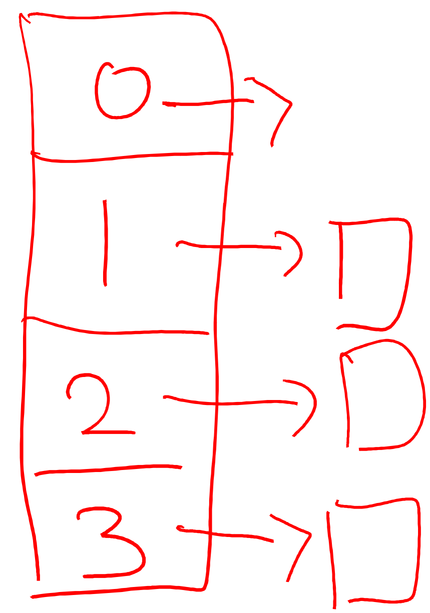
Example: LRU

- Equivalent to optimal in this simple example
- Works well due to locality of references (recently used pages accessed again with high probability)

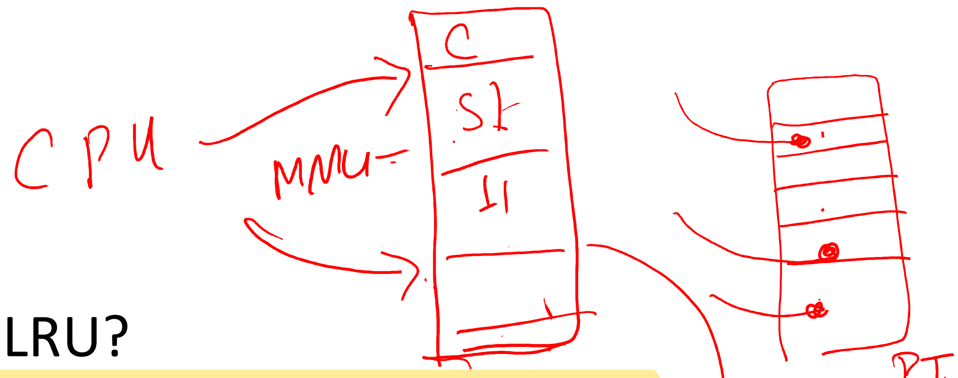
access pattern

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**



LRU implementation



- How does OS know which page is LRU?
  - OS is not involved in every memory access, so doesn't know which pages have been recently used
- Solution: MMU sets the accessed bit for every page table entry it accesses
  - Accessed bit is set implies page has been recently used
- Modern operating systems implement approximate LRU

OS

  - Periodically, look at accessed bit of pages to classify pages into active and inactive pages
  - Pick pages that have been inactive for eviction
  - May also avoid dirty pages for eviction, since it requires extra disk write

CPU VA MMU TLB PA code data

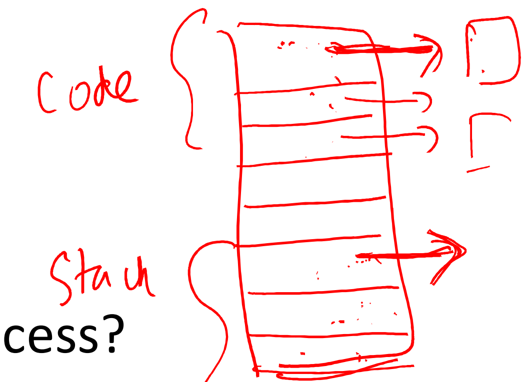
What happens on a memory access?

RAM

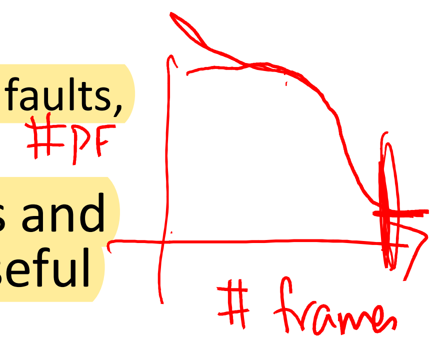
- CPU has requested data (or instruction) at a certain memory address
  - CPU checks caches. If hit in CPU cache, data is directly available (within ns) 10^{-9}
  - If CPU cache miss, CPU has to access main memory (hundreds of ns)
  - MMU checks TLB. If TLB hit, the physical address is available, the data is fetched from memory. Otherwise, MMU has to walk page table
  - If address is valid and present in page table, permission checks pass, translate address and access memory (MMU adds translation to TLB)
  - For any error in address translation, MMU traps to OS for page fault
  - OS may need to read/write from disk to service page fault (few millisec)
- Causes for application slowdown: CPU cache misses, TLB misses, page faults, …

PT

Thrashing



- How much physical memory should OS assign a process?
- Every process has a working set: frequently used pages in memory image
  - Working set can change from time to time, based on code being executed
  - Working set is usually smaller than total virtual memory of process
  - If memory assigned to process is less than working set, frequent page faults, frequent interruptions to process execution



- Thrashing = system spends too much time servicing page faults and swapping back and forth from disk, and too little time doing useful application work
  - Significant slowdown in application performance will be noticed by users
- Solution: users can reduce working set of processes, OS can terminate some processes or clean up unnecessary memory, …