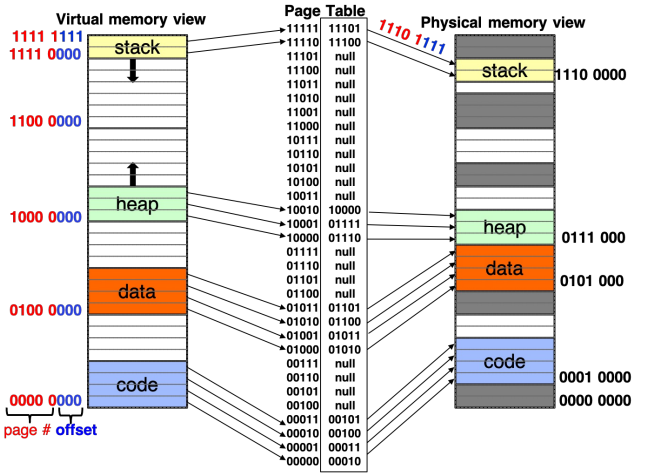
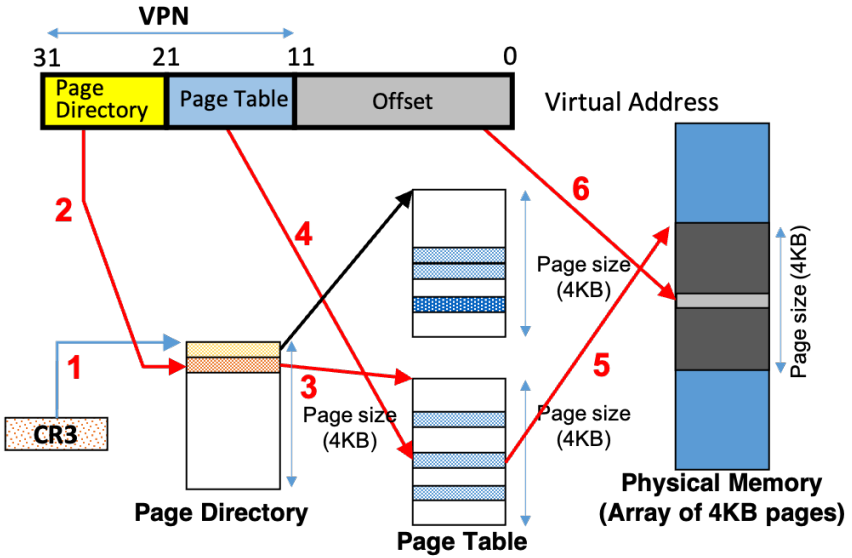
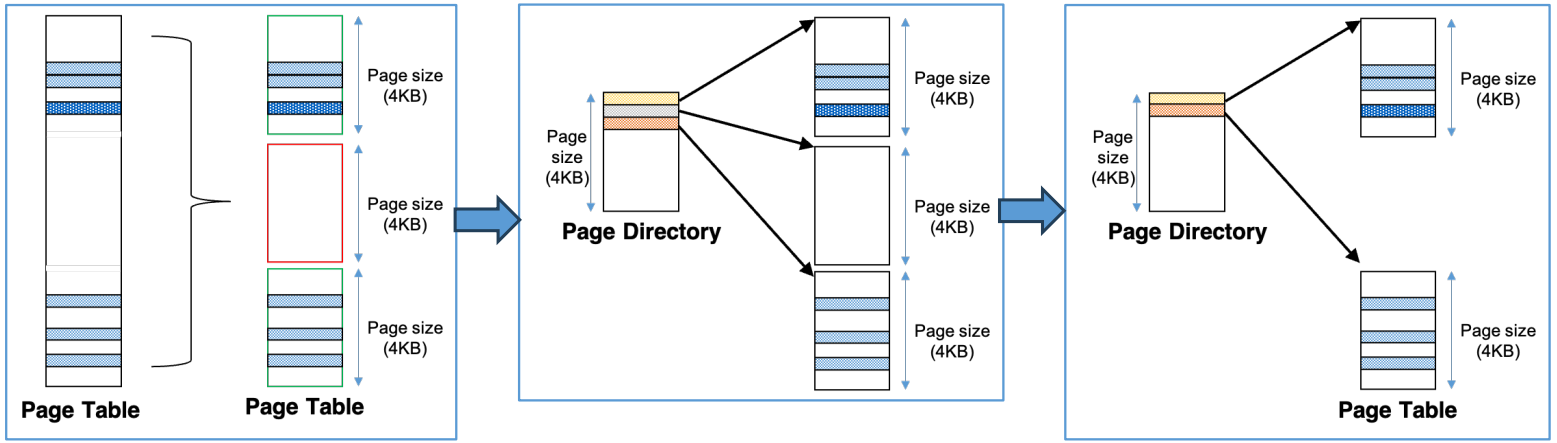
Lecture 20: Demand Paging

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture







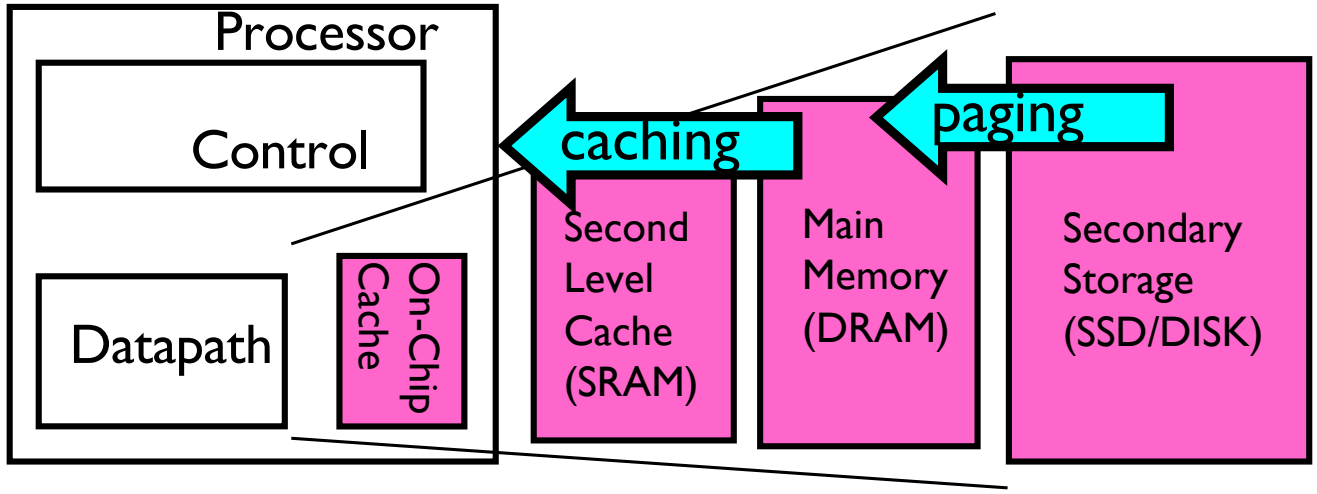
- Process address space is not contiguous. Hence, having a linear page table would lead to huge space overheads
- Increasing the page size can alleviate the overhead, but it would lead to fragmentation
- Multi-Level Page Table (MLPT)

Today's Class

- Demand paging
- Page replacement policies
  - FIFO
  - Random
  - LRU
    - Implementing LRU

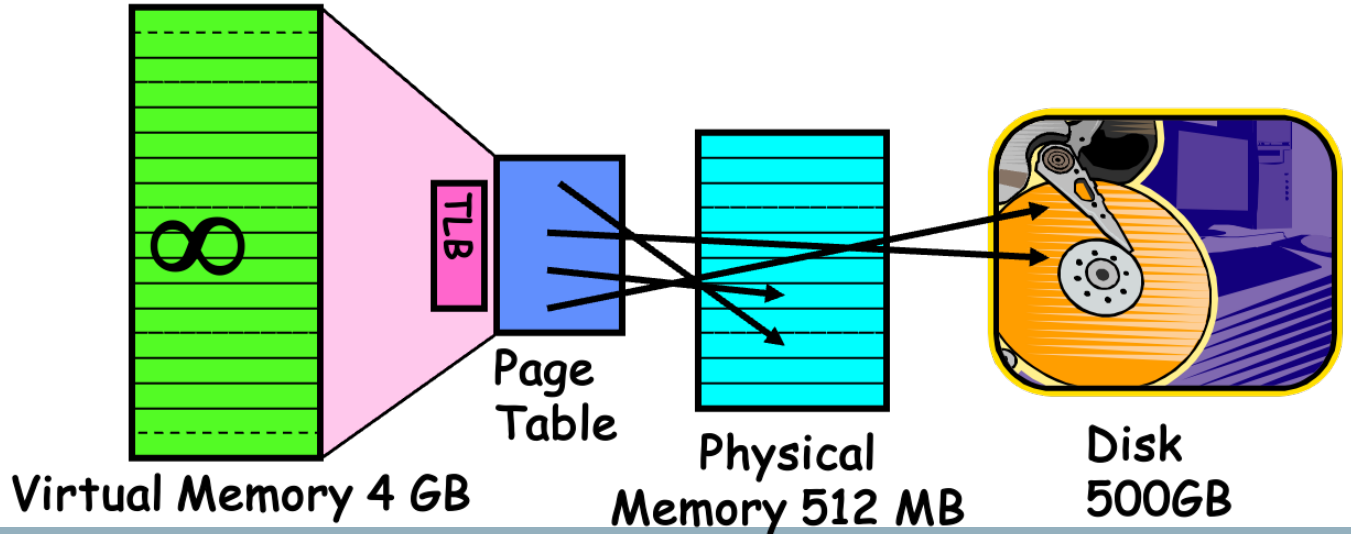
Migration of Pages across DRAM & DISK

- Modern programs require a lot of physical memory
- But they don't use all the memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as "cache" for disk



The Illusion of Infinite Memory

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - Performance issue, not correctness issue



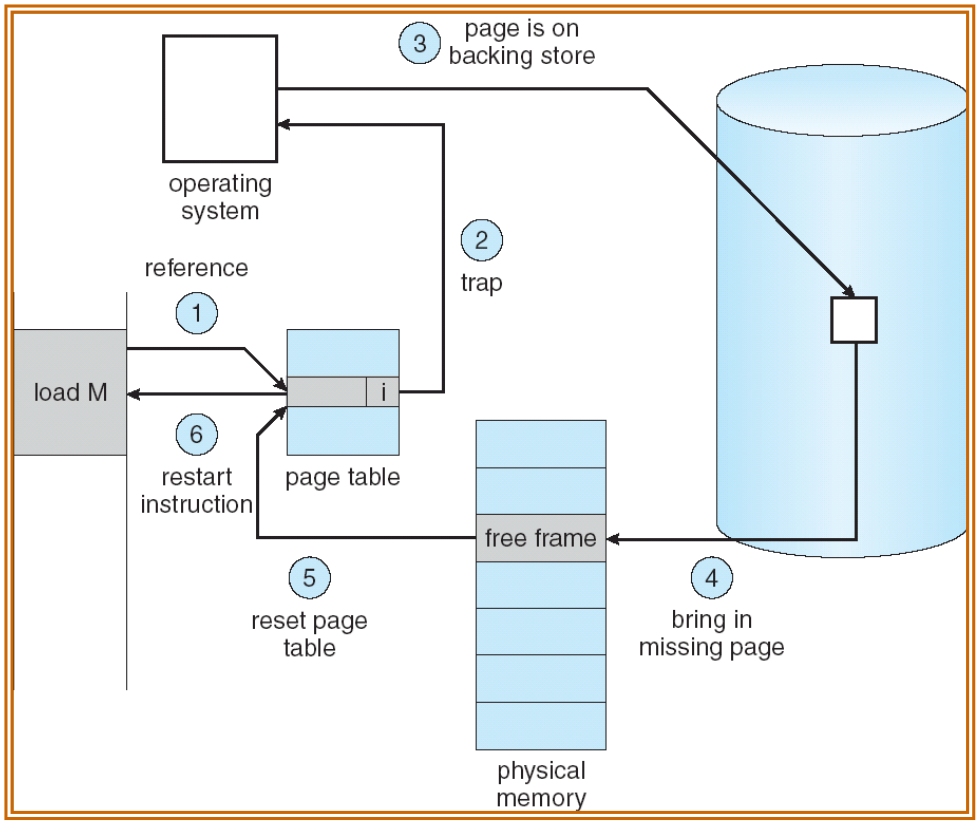
Mapping Virtual to Physical Address

- Approaches
  1. Virtual address is same as the physical address
  2. Virtual address is not the same as physical address
     a) Using base & bound for VA to PA mapping
     b) Using segmentation for VA to PA mapping
     c) Paging
        a) Using Page Table (PT) in RAM and having a base register for PT
        b) Using Page Table (PT) in RAM and having a base register for PT as well as a TLB cache
        c) Using Multi-level Page Table
        d) Demand paging

Demand Paging

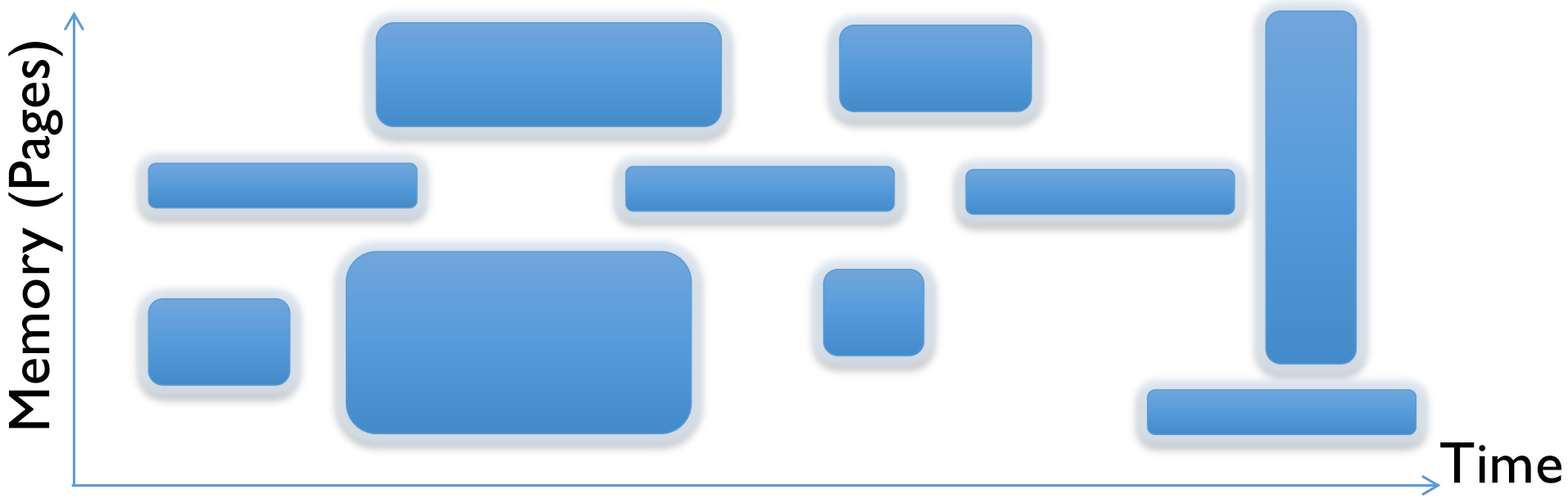
- It the mechanism to provide the illusion of infinite memory
- OS brings the pages into memory when it is accessed (i.e., on demand)
- PTE (Page Table Entry) makes demand paging implementable using the Present bit (a single dedicated bit in the PTE)
  - P=Valid $\Rightarrow$ Page in memory
  - P=Not Valid $\Rightarrow$ Page not in memory

Handling Page Fault in Demand Paging



- TLB miss happens for VA to PA mapping
- MMU reads present bit inside page table during VA to PA translation
  - Present bit in page table entry indicates if a page of a process resides in memory or not
- If page present in memory, directly accessed, otherwise generate page fault exception and move into kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address
- OS context switches to another process
  - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process is scheduled again the OS restarts the instruction that caused page fault

Working Set Model



As a program executes, it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space. If fewer pages remain active over time then the OS may move those inactive pages to the disk

When to Move Unused Pages to Disk?

- OS keeps some amount of DRAM always free by deciding some upper cap
- Whenever the upper cap memory level is breached, unused pages are moved out to disk
  - OS uses a daemon process for carrying out this activity
- To help decide which pages are really “old”/”unused”, there are several **page replacement policies**

Cost of Demand Paging

- Average Memory Access Time (AMAT)
  - Memory Access Time (DRAM) + Probability of Page Fault (a miss) x Page fault service time
    - MAT + P_{Miss} x PF_{Time}
- Example
  - Memory access time = 200 nanoseconds
  - Suppose p = Probability of page fault (miss)
  - Average page-fault service time = 8 milliseconds
  - Then, we can compute AMAT as follows:

    AMAT = 200ns + p x 8 ms = 200ns + p x 8,000,000ns
  - If one access out of 1,000 causes a page fault, then EAT = 8.2 µs

Page Replacement Policy (1/2)

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
    - Recall, with demand paging we are using DRAM as a cache for keeping pages
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out

Page Replacement Policy (2/2)

- FIFO
  - Throw out oldest page
  - Bad – might throw out heavily used pages instead of infrequently used
- Random
  - Pick random page for every replacement
  - Better than FIFO, but it goes by luck – no guarantees!
- MIN (Minimum) – a.k.a. the optimal policy
  - Replace page that will be used furthest in the future
    - But we can't really know future!
- LRU (Least Recently Used)
  - Replace page that hasn't been used for the longest time
    - We cannot know the future, but we can use the history to predict the future!
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future

FIFO Policy

- Suppose we have the following page reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since we need A again right away

MIN Policy (= LRU in this Stream)

- Suppose we have the same reference **stream**:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future
- LRU will also function exactly the same for this reference stream

LRU May Perform Bad Also..

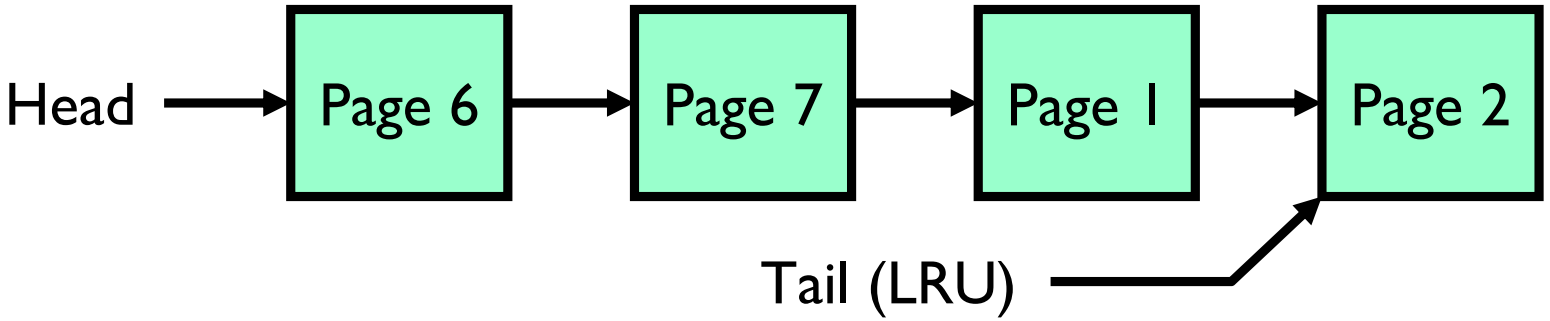
- Consider another reference stream: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!

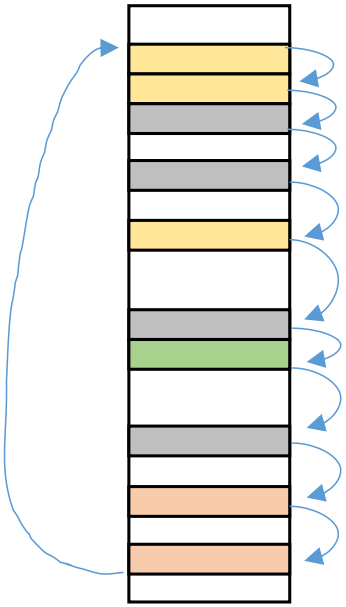
Implementing LRU

- LRU can be implemented using a list



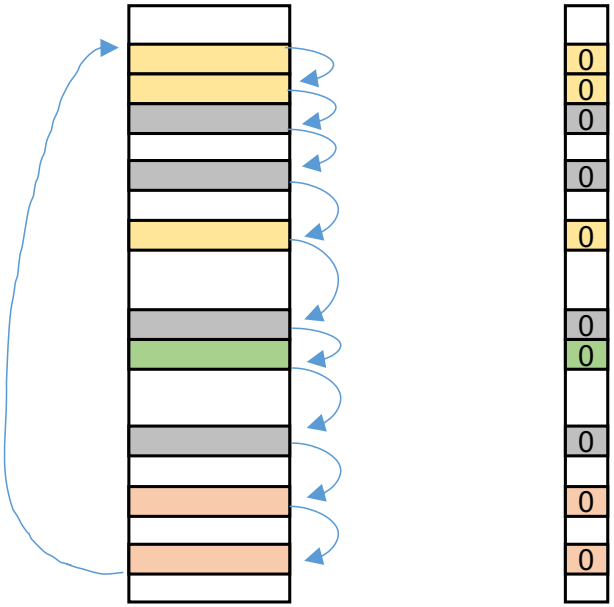
- Upon each use, remove that page from list and place at the **head**
  - Least Recently Used (LRU) page is at the **tail**
- Problems with this scheme for paging?
  - Need to know immediately whenever page is used so that can change position in list
  - Lots of work required at each memory access for this implementation

Approximating LRU: Clock Algorithm (1/7)



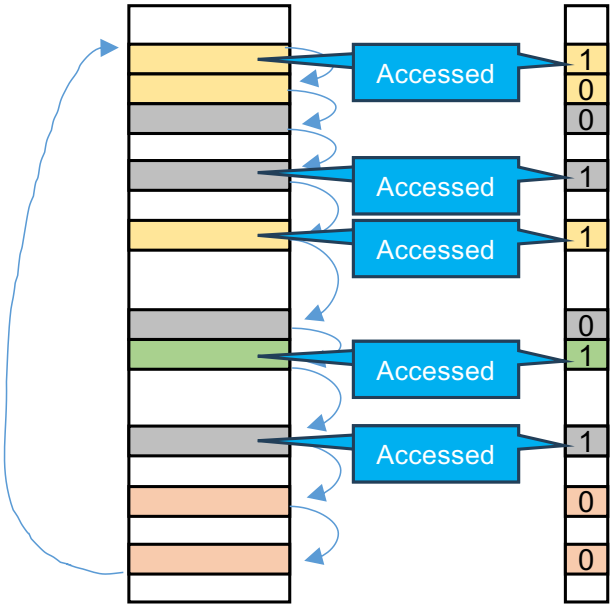
- Each physical pages (**used ones**) are arranged as circular linked list

Approximating LRU: Clock Algorithm (2/7)



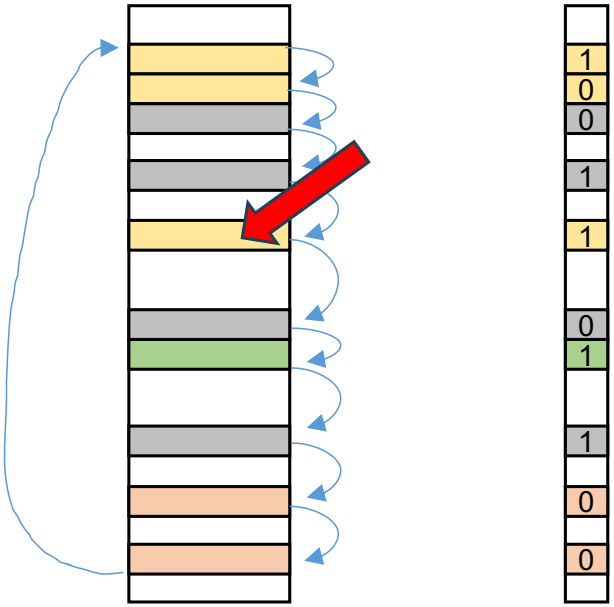
- Each physical pages (**used ones**) have a dedicated bit stored in RAM known as “use” bit
  - Imagine OS maintaining another array of “use” bits inside DRAM, where each index is a “use” bit for one dedicated physical page
- Values could be 0 or 1
  - Default value set to 0

Approximating LRU: Clock Algorithm (3/7)



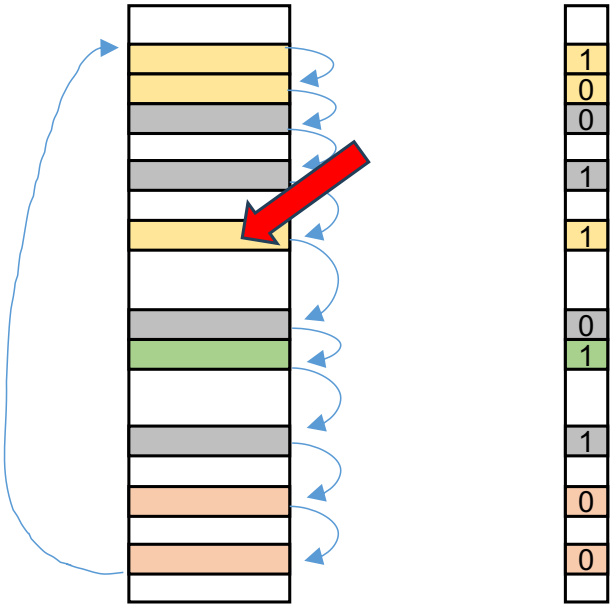
- Whenever a page is accessed (load/store of some address inside this page) by the process, the **hardware** changes its "use" bit to 1

Approximating LRU: Clock Algorithm (4/7)



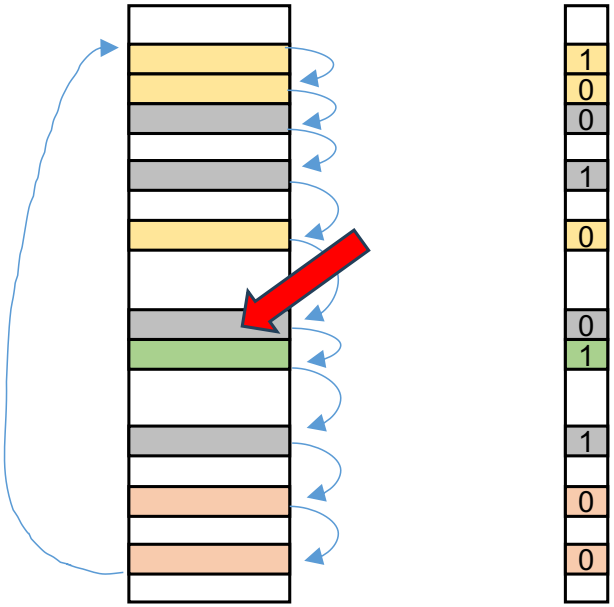
- A clock hand (single hand) points to any one of the page in this list
  - It doesn't matter which one
- This single clock hand advances to a next page in the list **only** upon a page fault for demand paging

Approximating LRU: Clock Algorithm (5/7)



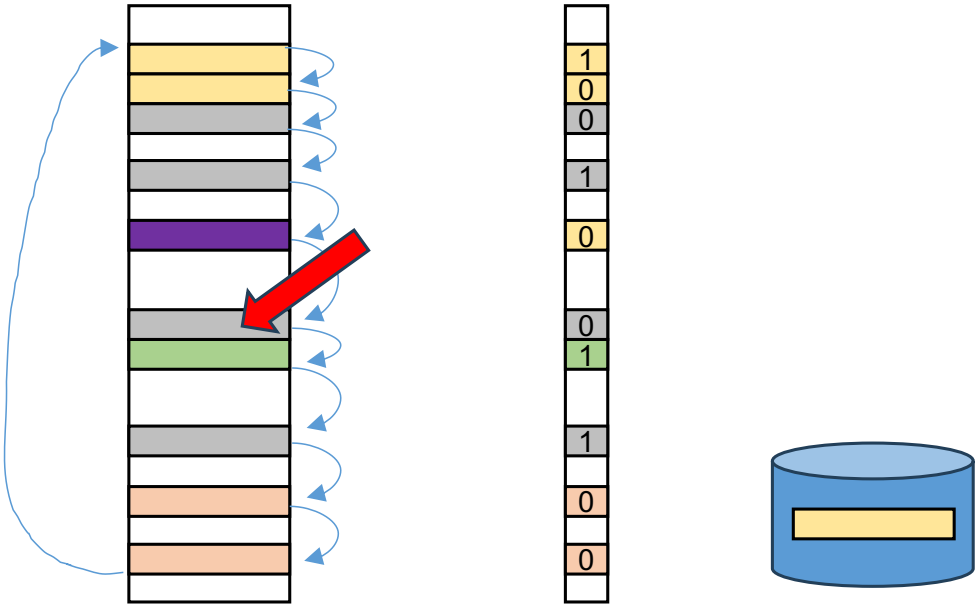
- At each page fault (request for demand paging), advance the clock hand (not real time), and check the **use** bit
  - **1→ used recently; set to 0 and move to next page**

Approximating LRU: Clock Algorithm (6/7)



- At each page fault (request for demand paging), advance the clock hand (not real time), and check the **use** bit
  - **0→ selected candidate for replacement**

Approximating LRU: Clock Algorithm (7/7)



- The candidate page is moved to the disk and required page from the disk is brought into the newly created page slot on the DRAM
- There are several optimizations possible in this simple clock algorithm

Next Lecture

- Introduction to multithreading
  - Start of a new module – concurrency!
    - Total 3 lectures