L26- VIRTUAL MEMORY

Overview of Memory Management

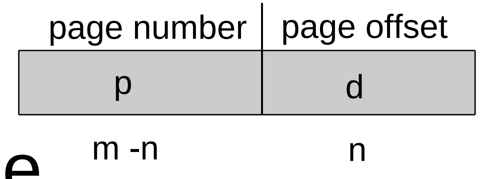
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files

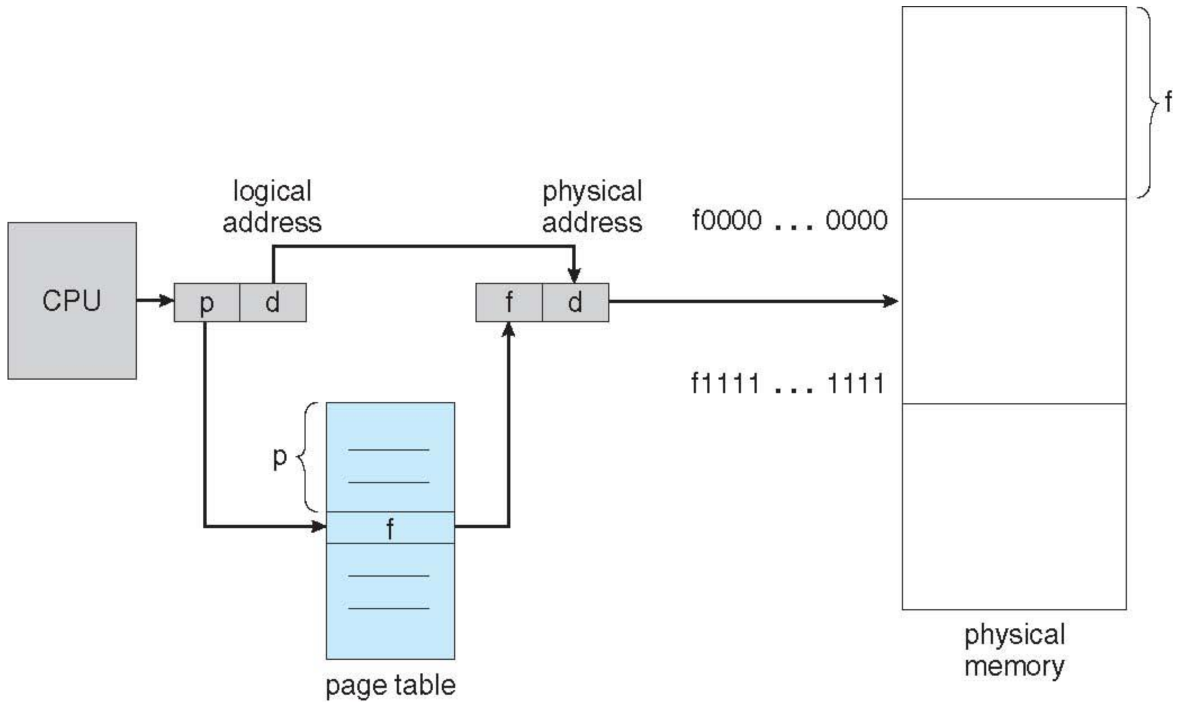
Objectives

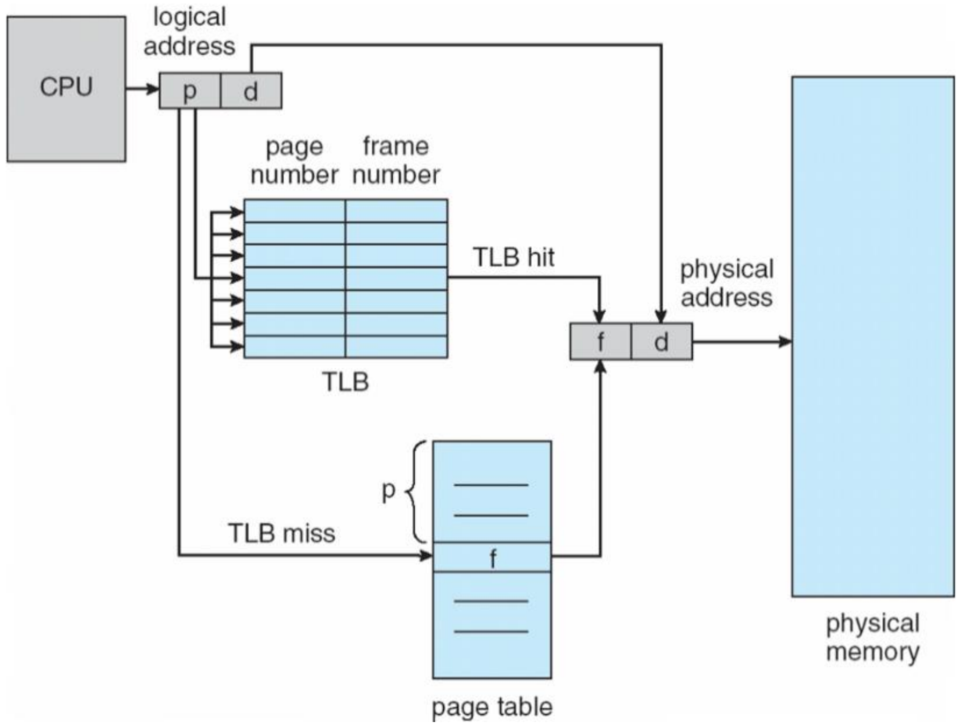
- ❖ To describe the benefits of a virtual memory system
- ❖ To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- ❖ To discuss the principle of the working-set model
- ❖ To examine the relationship between shared memory and memory-mapped files

Paging & Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (*p*) – used as an index into a page table
  - **Page offset** (*d*) – displacement within a page







Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Background

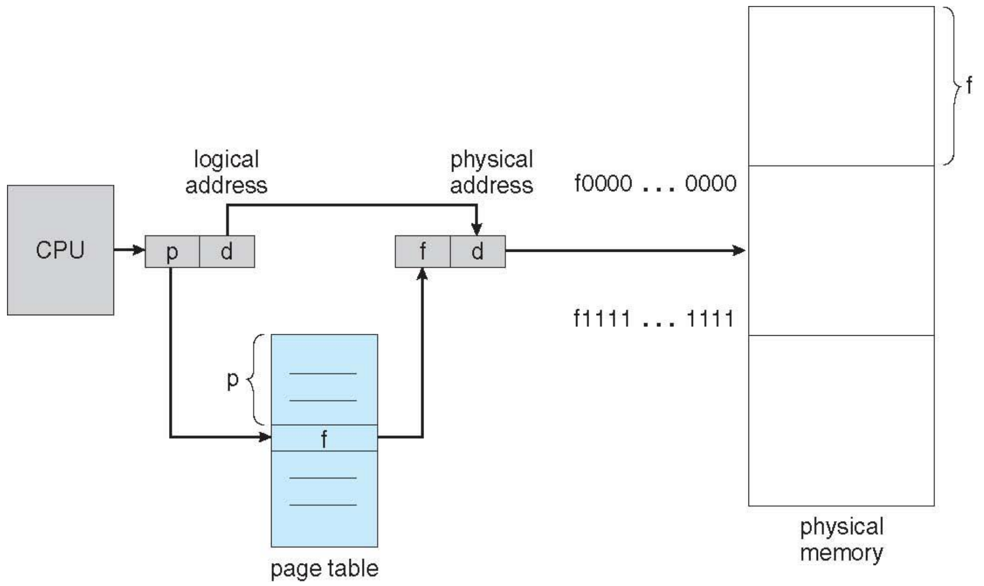
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running → more programs run at the same time
  - Increased CPU utilization and throughput with no increase in response time or turnaround time

Background

- **Virtual memory** – separation of user logical memory from physical memory
- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- More programs running concurrently
- Less I/O needed to load or swap processes

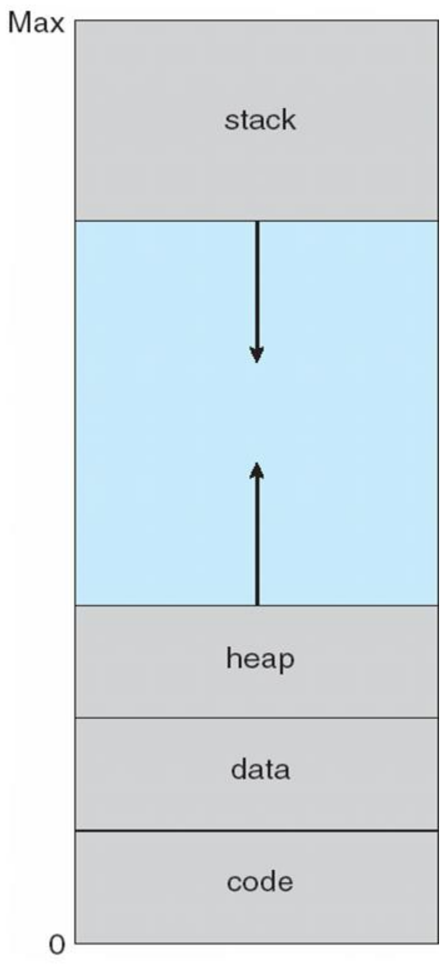
Background

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

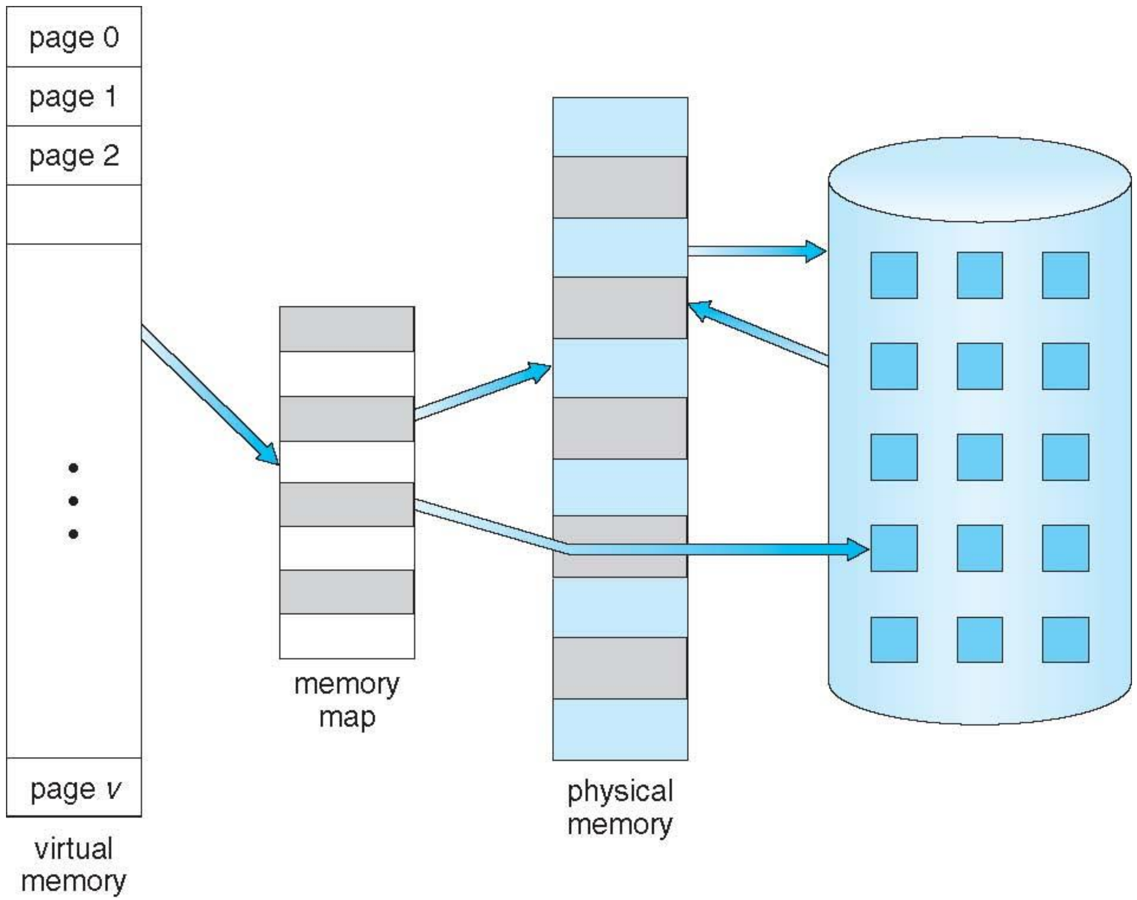


Virtual-address Space

- ❖Logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
- ❖Unused address space between stack and heap is hole
- ❖Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

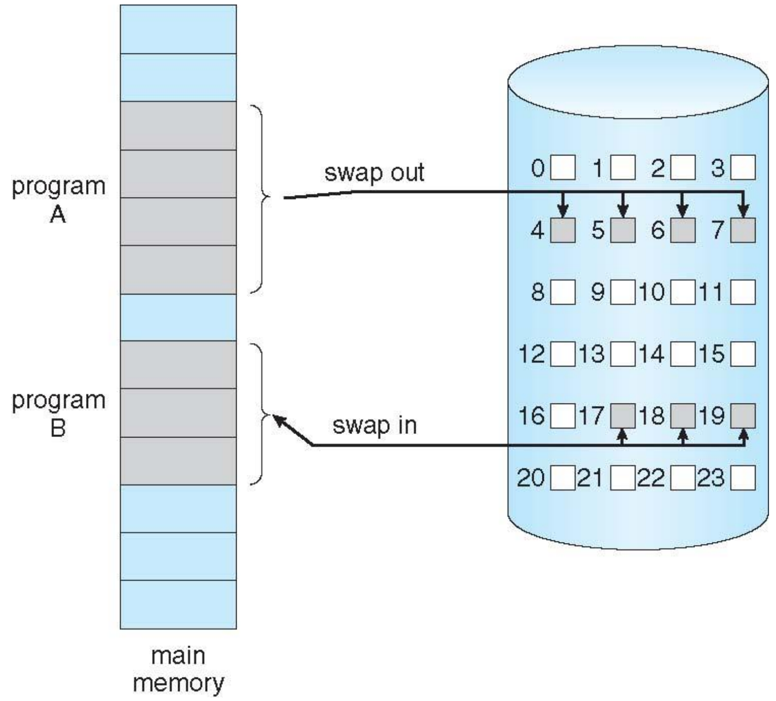


Virtual Memory To Physical Memory Mapping



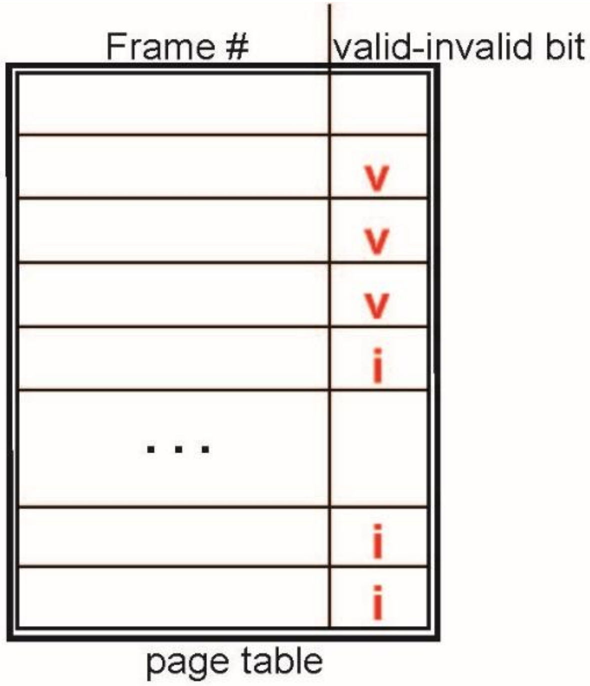
Demand Paging

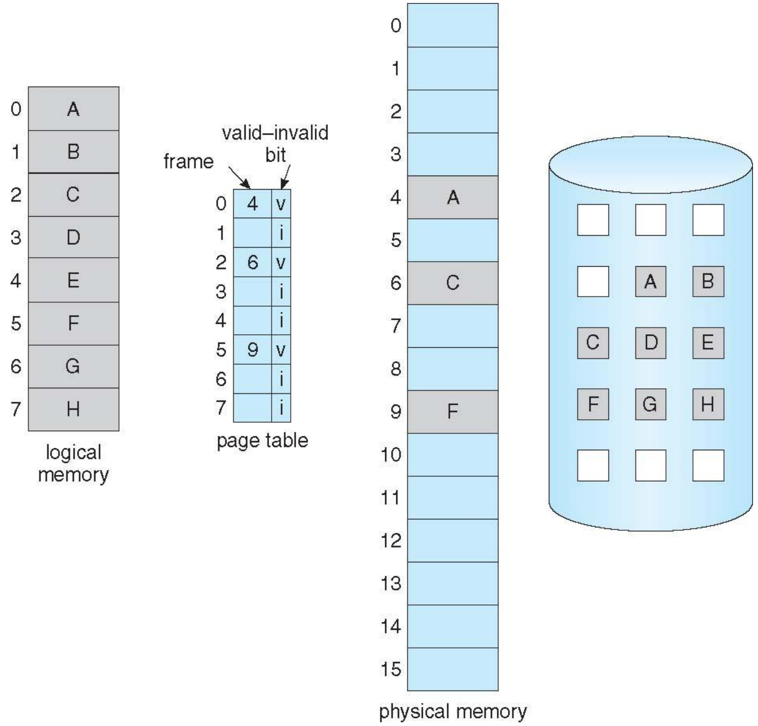
- Bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries



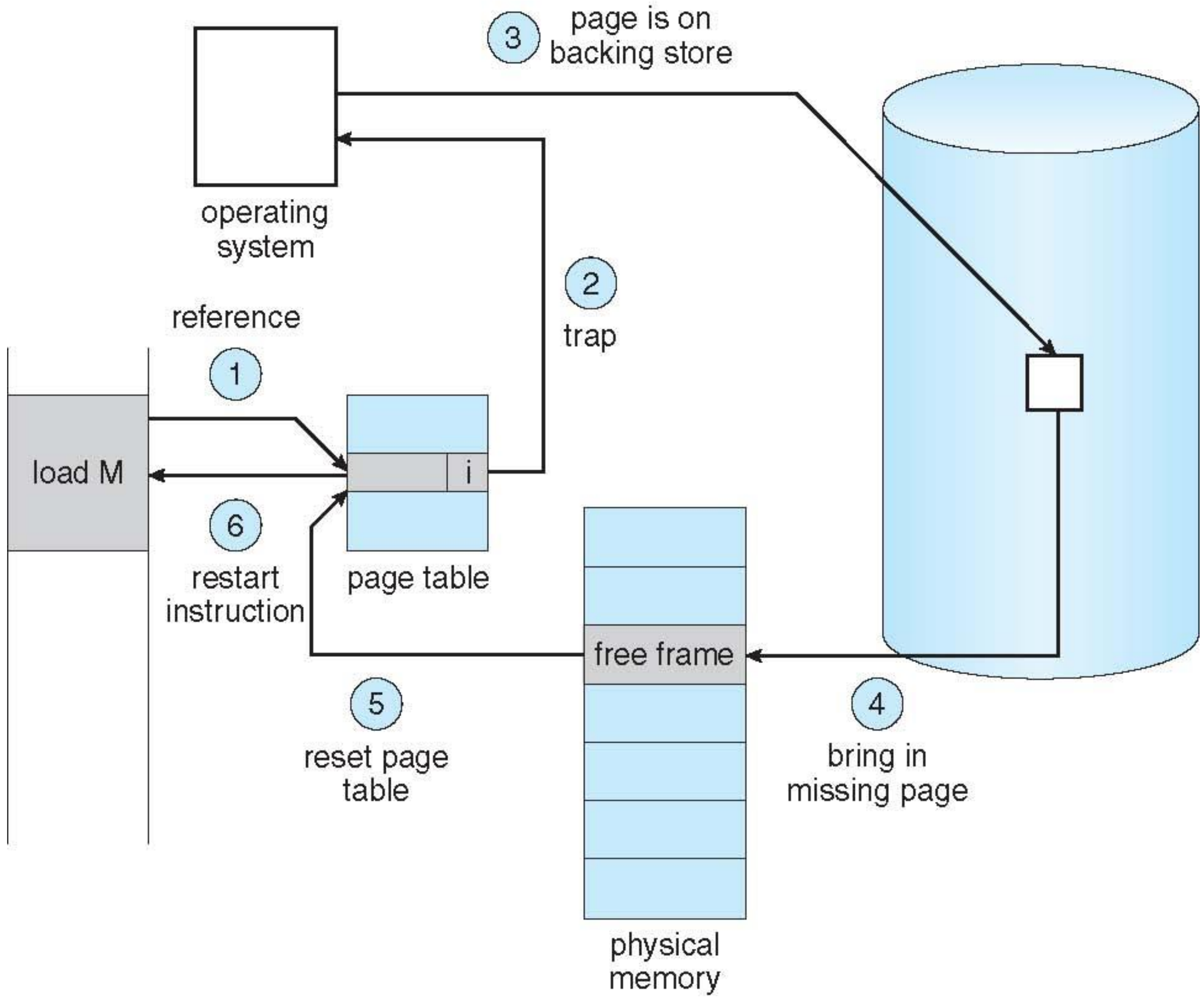


Page Table When Some Pages Are Not in Main Memory

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: **page fault** (page not found in main memory)
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory: Set validation bit = **1**
- Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Demand Paging Overhead

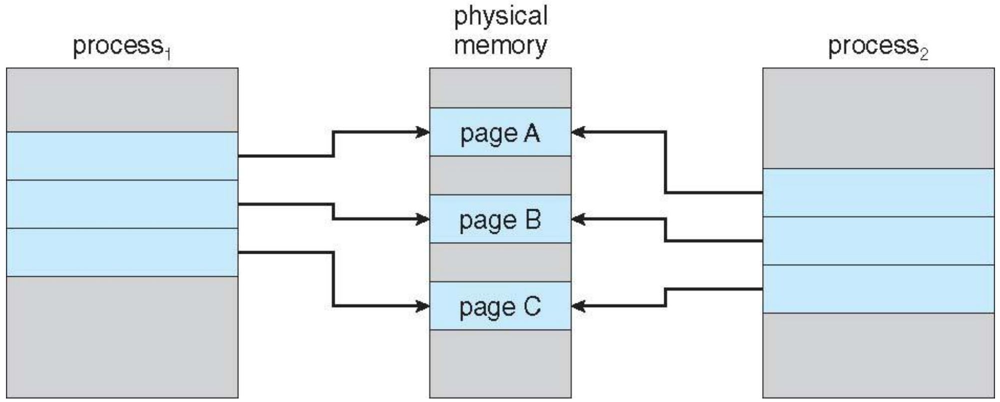
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

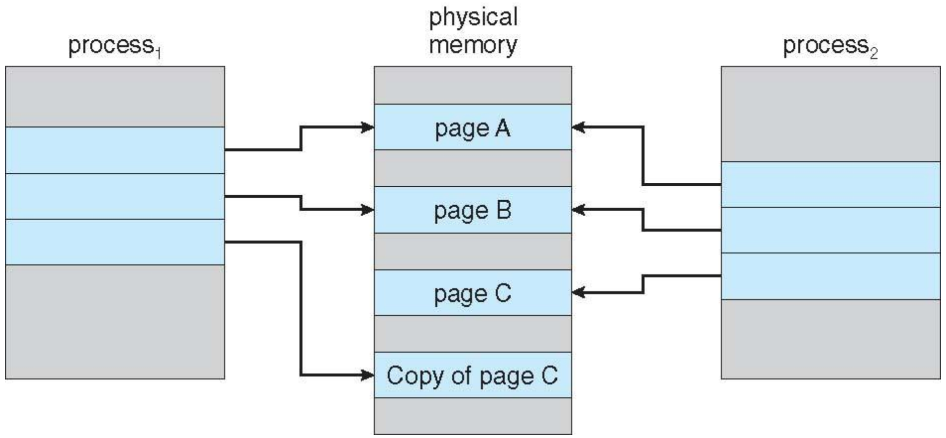
  EAT = $(1 – p)$ x memory access + p (page fault overhead

  + swap page out + swap page in)

Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same pages in memory
- If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

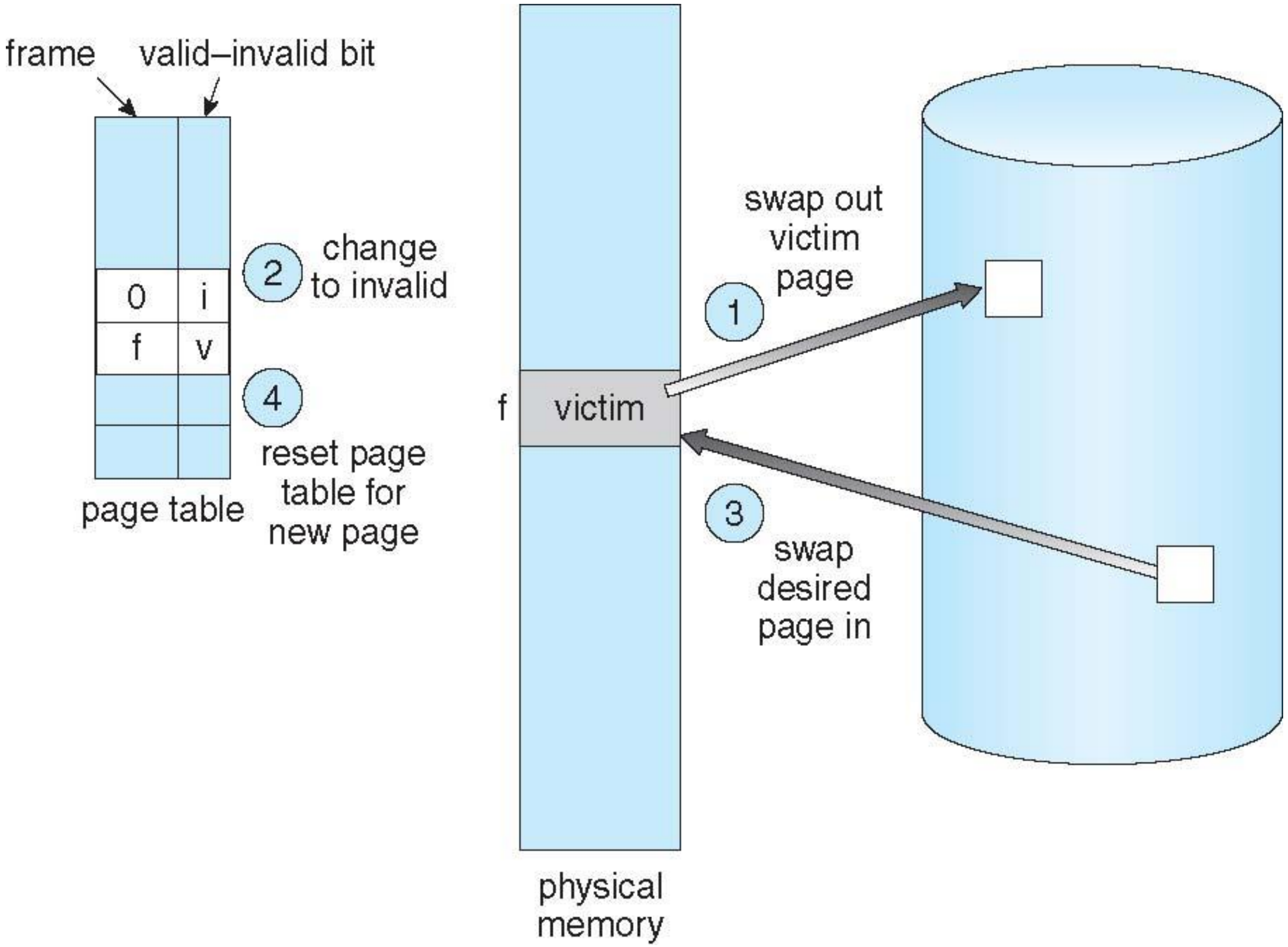




Basic Page Replacement

- ❖ Find the location of the desired page on disk
- ❖ Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
  - Write victim frame to disk if dirty
- ❖ Bring the desired page into the (newly) free frame; update the page and frame tables
- ❖ Continue the process by restarting the instruction that caused the trap

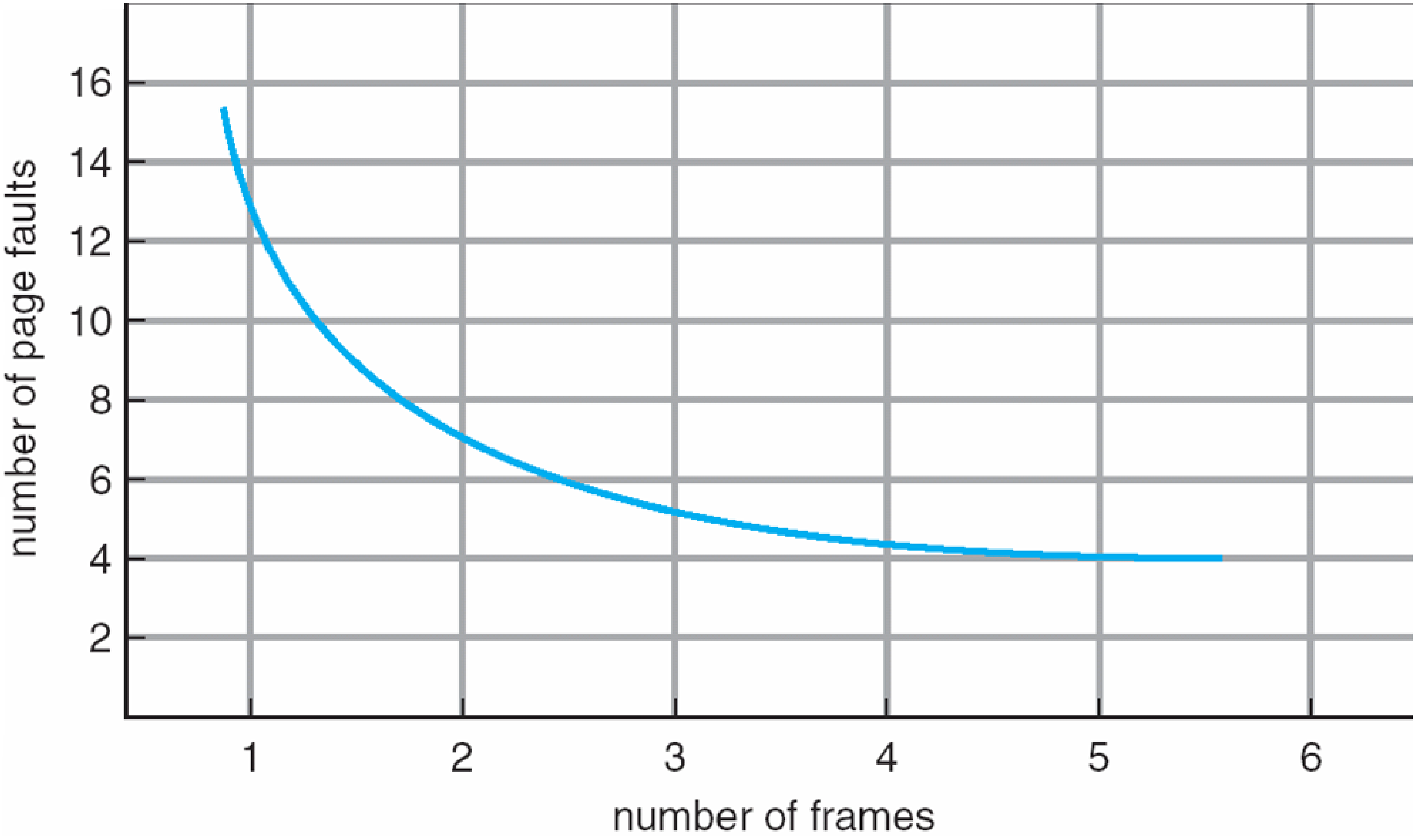
Page Replacement



Page Replacement Algorithm

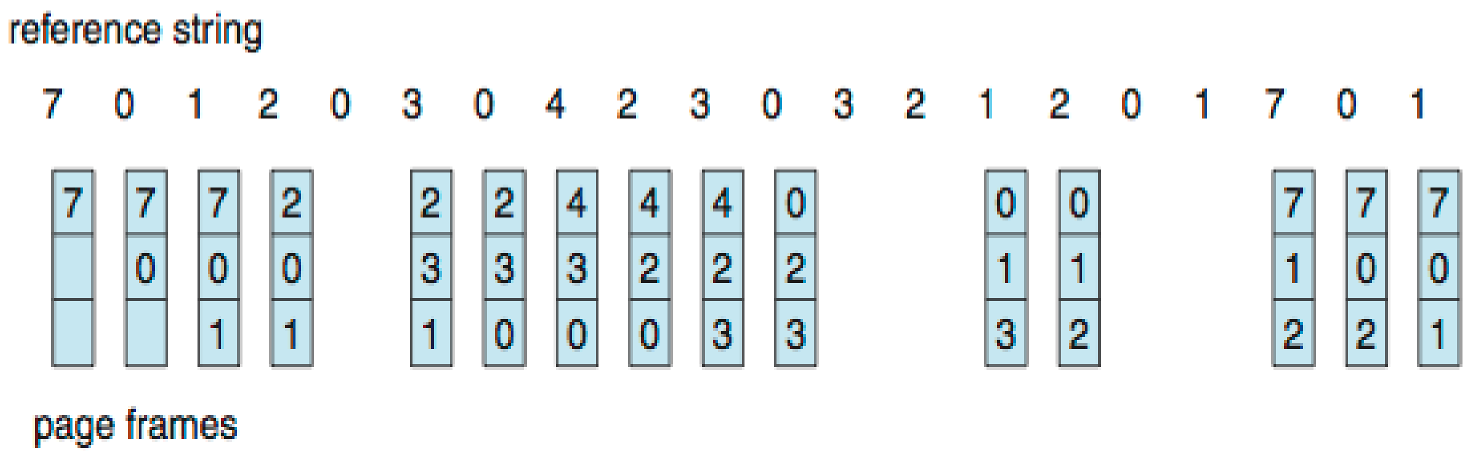
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - **FIFO, LIFO,**
  - **Optimal,**
  - **LRU, LRU approximations,**
  - **LFU, MFU**

Page Faults Vs Number of Frames



First-In-First-Out (FIFO) Algorithm

- Ref. string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

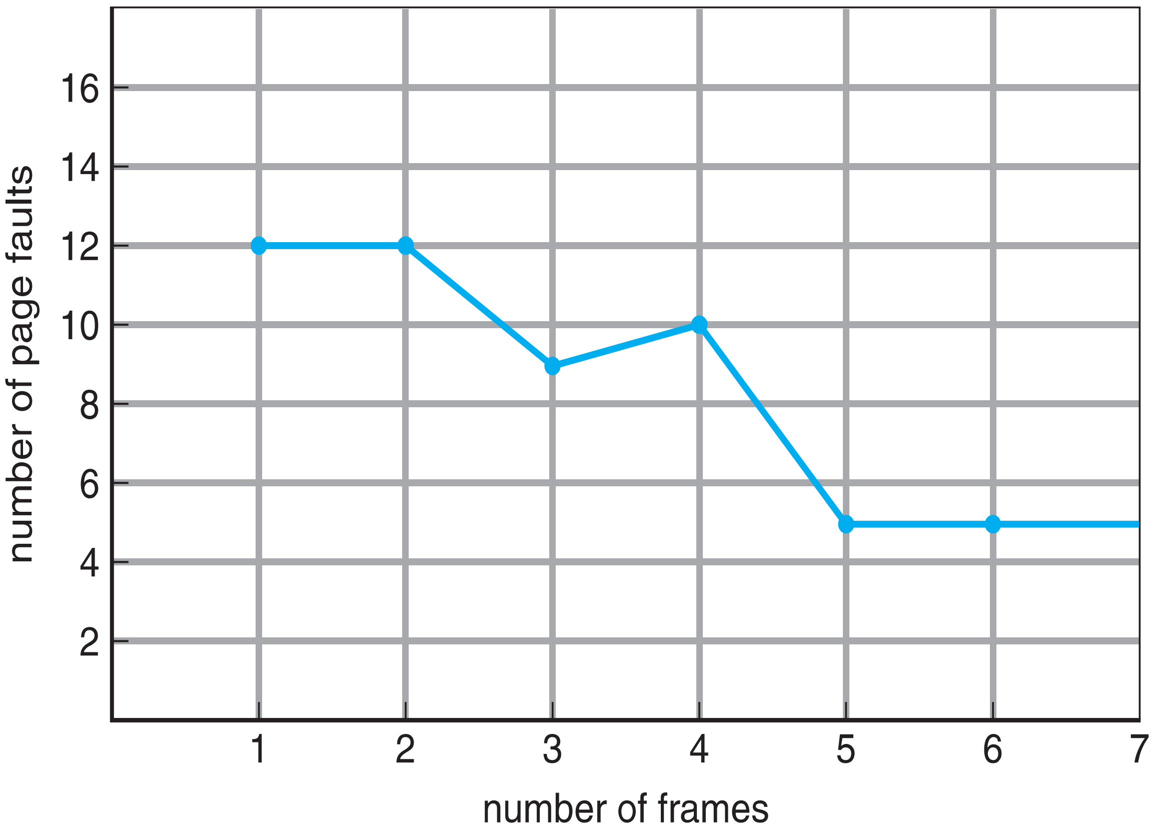


15 page faults

- How to track ages of pages? - Use a FIFO queue

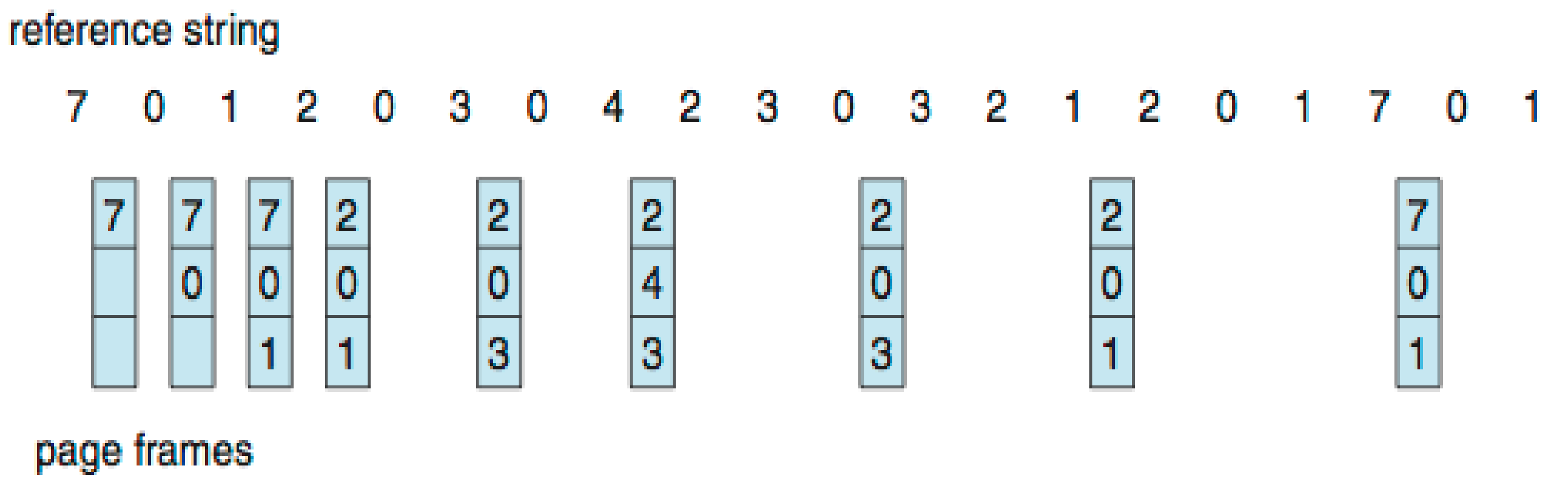
Belady's Anomaly

- Consider 1,2,3,4,1,2,5,1,2,3,4,5
- Adding more frames can cause more page faults! - **Belady's Anomaly**



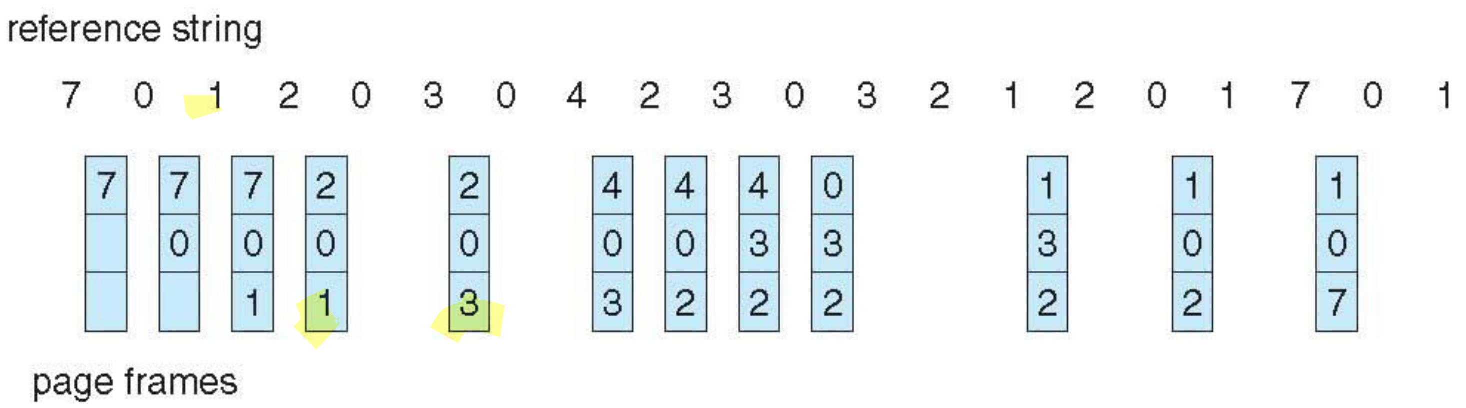
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- Practical difficulty- Can't read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

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



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

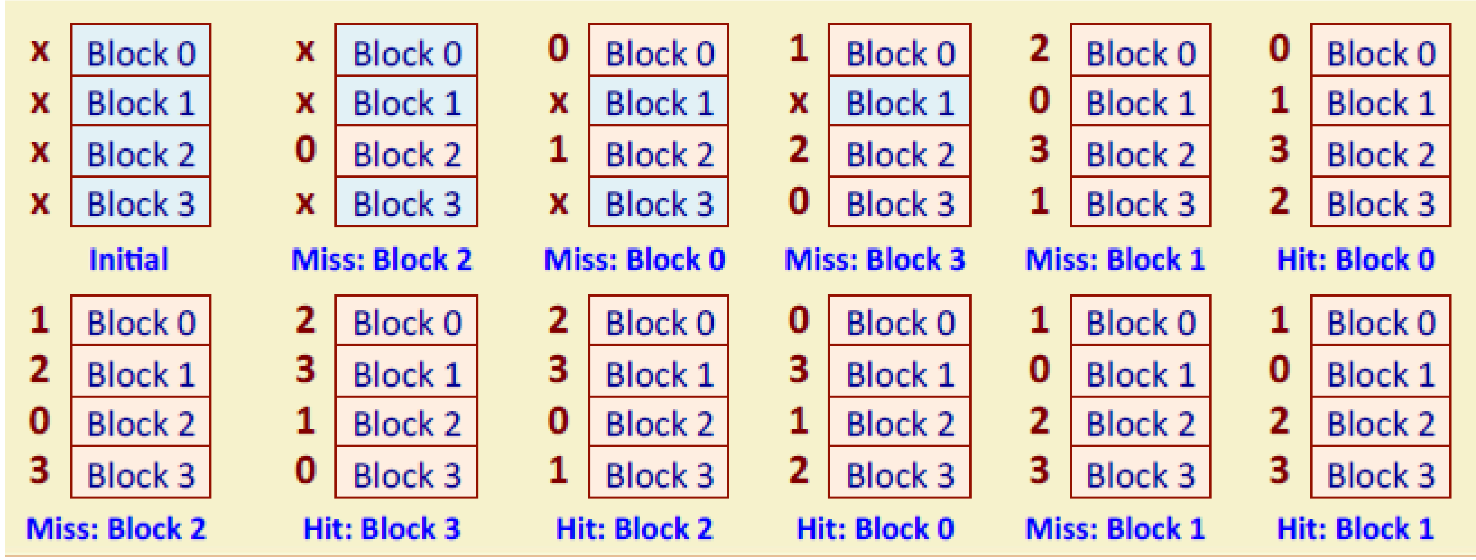
LRU Algorithm Implementation

- Counter implementation
  - 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.

- To implement the LRU algorithm, the memory controller must track the LRU block as the computation proceeds.
- Example: Consider a Page Table with 4 pages.
  - For tracking the LRU block within a Page table, we use a 2-bit counter with every block.
  - When hit occurs:
    - Counter of the referenced block is reset to 0.
    - Counters with values originally lower than the referenced one are incremented by 1, and all others remain unchanged.

– When miss occurs:

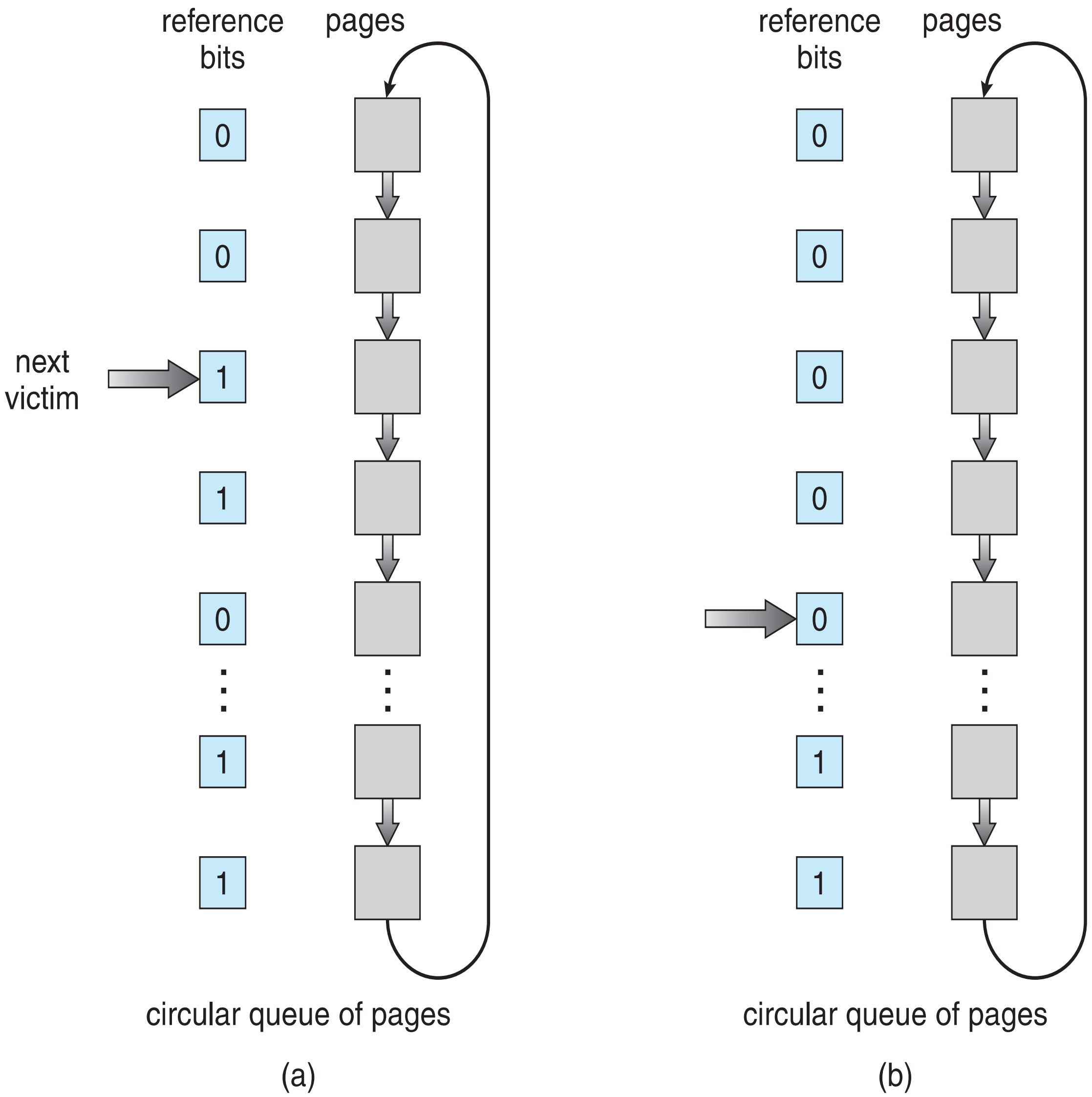
- If the set is not full, the counter associated with the new block loaded is set to 0, and all other counters are incremented by 1.
- If the set is full, the block with counter value 3 is removed, the new block put in its place, and the counter set to 0. The other three counters are incremented by 1.



LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
  - Replace any with reference bit = 0 (if one exists)
- **Second-chance algorithm**
  - If page to be replaced has
    - Reference bit = 0 →replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance Page-Replacement



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit
- Take ordered pair (reference, modify)
- (0, 0) neither recently used not modified – best page to replace
- (0, 1) not recently used but modified – not quite as good, must write out before replacement
- (1, 0) recently used but clean – probably will be used again soon
- (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

Counting Algorithms

- Keep a counter of the number of references that have been made to each
- **Lease Frequently Used (LFU) Algorithm**: replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Always keep a pool of free frames
  - When needed, frame is always available, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Keep list of modified pages
  - When free, write to backing store and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Thank You