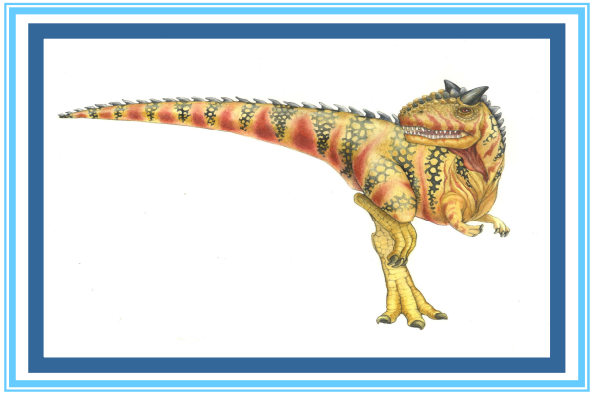
Chapter 9: Virtual-Memory Management

School of Computing, Gachon Univ.
Jungchan Cho



Most slides from "Operating System Concepts – 10th Edition".
Many slides are taken from lecture notes of Prof. Joon Yoo.

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging
- To explain the page-replacement algorithms

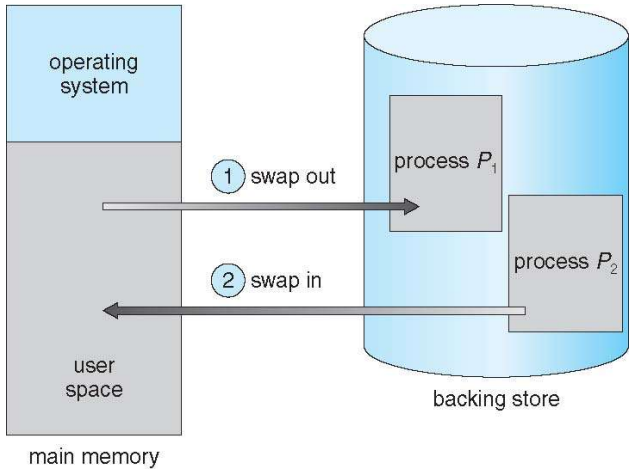
Chapter 9: Virtual-Memory Management

- **Background**
- Demand Paging
- Page Replacement
- Page Replacement Algorithms

Chapter 8: Swapping

- Q: What if memory needed by process exceeds physical memory size?
  - In other words, can process size > physical memory?
- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program

- **Swapping**
  - A whole process can be swapped temporarily out of memory to a backing store (swap device – usually HDD, SSD, or Flash) then brought back into memory for continued execution



Chapter 8: Swapping

- Context switching time of swapping system is fairly high
  - e.g., user process 100MB, hard disk 50MB per second: then, 100MB / 50MB per second = 2 seconds. Swap-in + swap-out = 2 seconds x 2 = 4 seconds

- Swapping takes too much time! Modern OSs try to avoid swapping – only enable if available physical memory is small

- Can we do better?
  - Swap only portions of process (rather than entire process) to decrease swap time: **Demand Paging**

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures (arrays)
- Entire program code not needed at same time
- Consider ability to execute **partially-loaded program**
  - e.g., 500MB program. Only need to load 100MB to start program. The rest can be loaded on-demand!

Chapter 9: Virtual-Memory Management

- Background
- **Demand Paging**
  - Basic Concepts
  - Valid-Invalid Bit
  - Page Fault
  - Performance of Demand Paging
- Page Replacement
- Page Replacement Algorithms

Demand Paging Concept

- We can view a process as
  - a sequence of pages (rather than large contiguous address spaces)

- How can an executable program be loaded from disk into memory?
  - **Option 1**
    - Load the entire program into memory
  - **<u>Option 2</u>**
    - **<u>Load pages only as they needed</u>**

Demand Paging

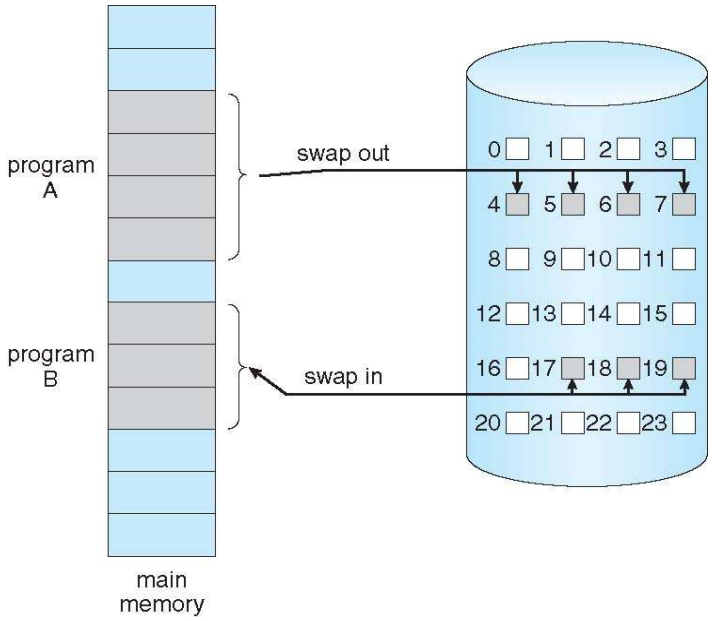
- Concept:
  - Instead of loading the whole program into memory, bring pages into memory only when it is needed

- Why? (Advantages)
  - Less I/O needed from disk to load memory
    - Faster response to load programs
  - Less memory needed
    - Larger programs
    - Better multi-program – more users for server

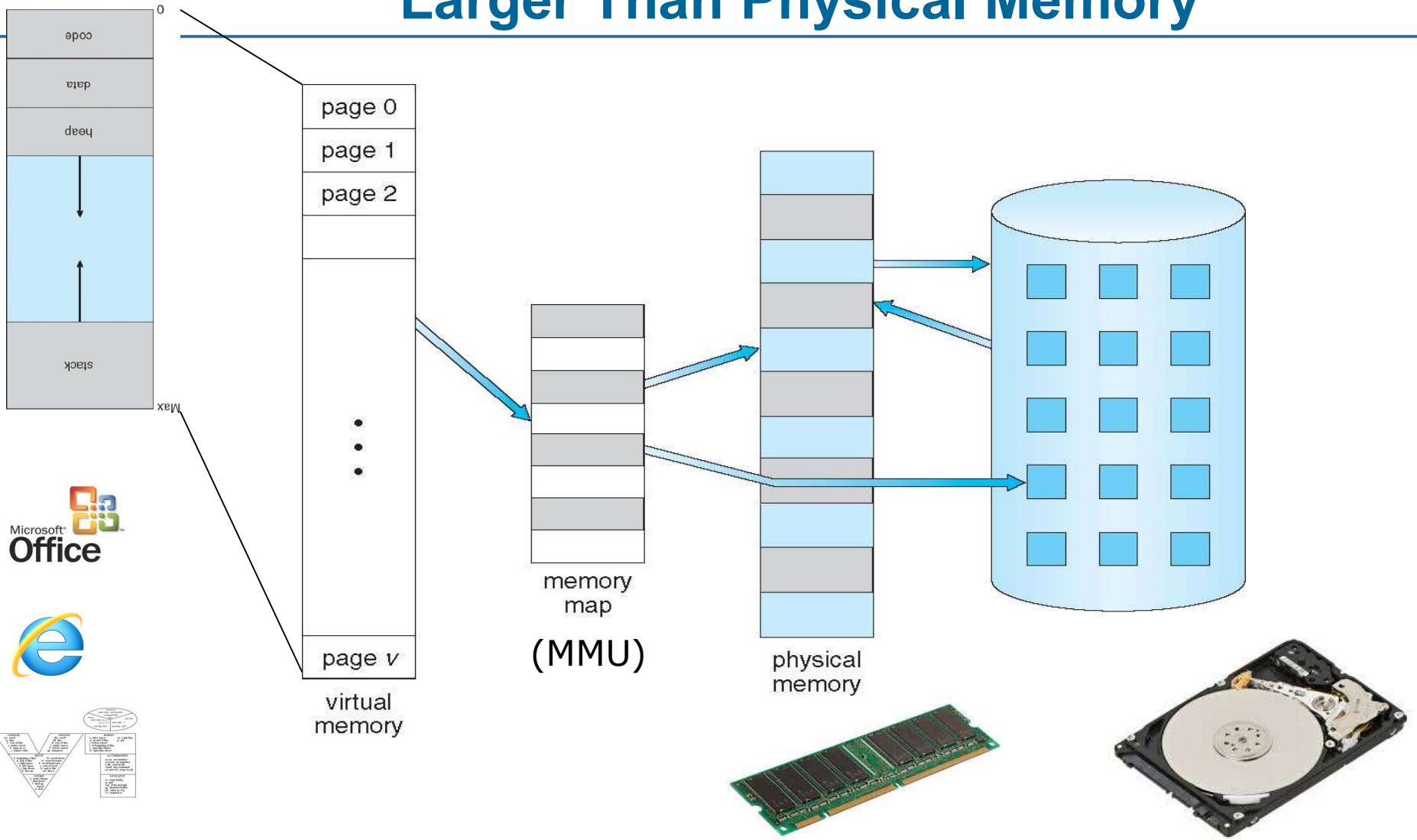
Demand Paging

- Pages are loaded only when they are demanded during execution
  - Pages that are never accessed are never loaded into physical memory

- **Pager**
  - Conducts **demand paging**
  - c.f., swapper: swaps in/out the whole process

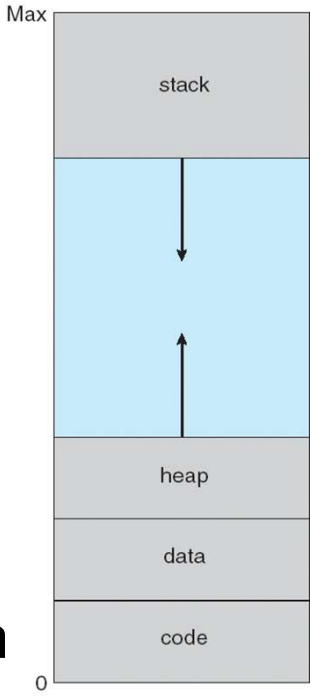


Virtual Memory That is Larger Than Physical Memory



Demand paging usage: Heap/Stack

- Heap: grow upward in memory as it is used for dynamic memory allocation
- Stack: grow downward in memory through successive function calls
- Large blank space is part of virtual address space, but will require actual physical pages only if the heap and stack grows
- Using **sparse** address spaces with holes left for growth can be filled during execution via **demand paging**
  - as the stack or heap segments grow



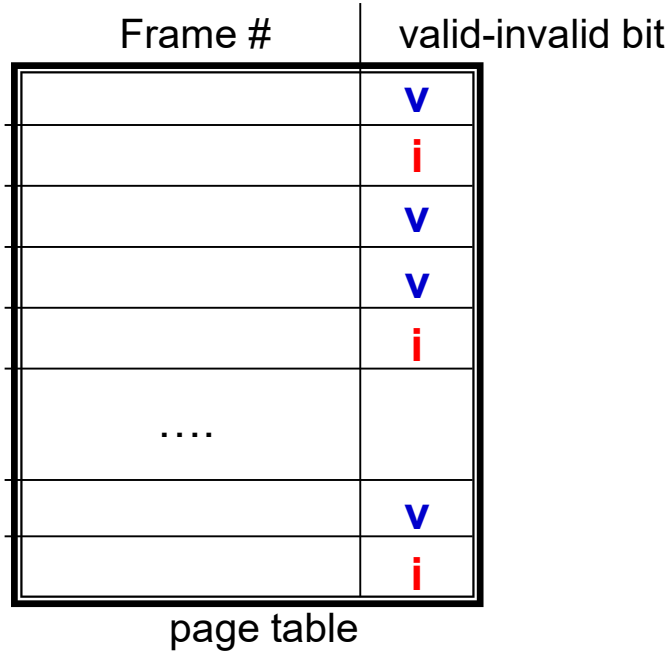
Demand paging usage: Dynamically linked libraries (DLL)

- **Dynamically linked libraries** (**DLL**)
  - Library linking is postponed until execution time

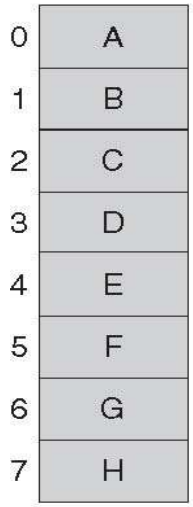
- Now some pages are in memory but some are still in disk (not in memory)
- Q: How can distinguish this? - **Need H/W support**
  - **Valid-Invalid Bit**

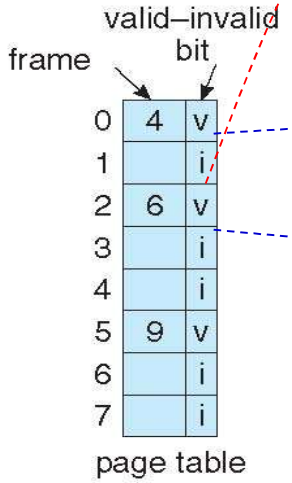
Valid-Invalid Bit

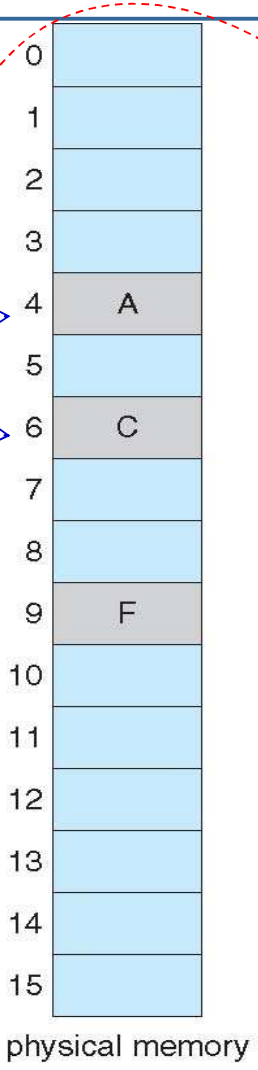
- With each page table entry, a **valid–invalid bit** is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
  - or it can be represented as **1** / **0** for valid/invalid.
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ **page fault**

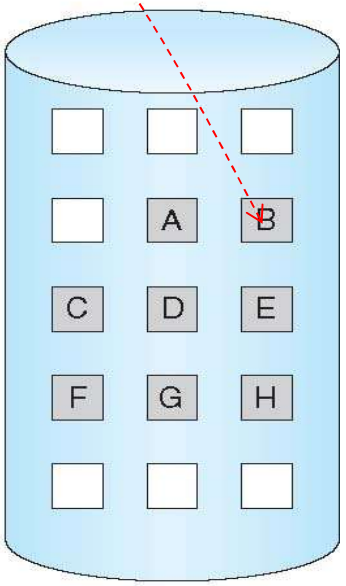


Page Table When Some Pages Are Not in Main Memory



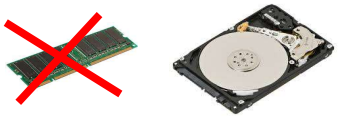






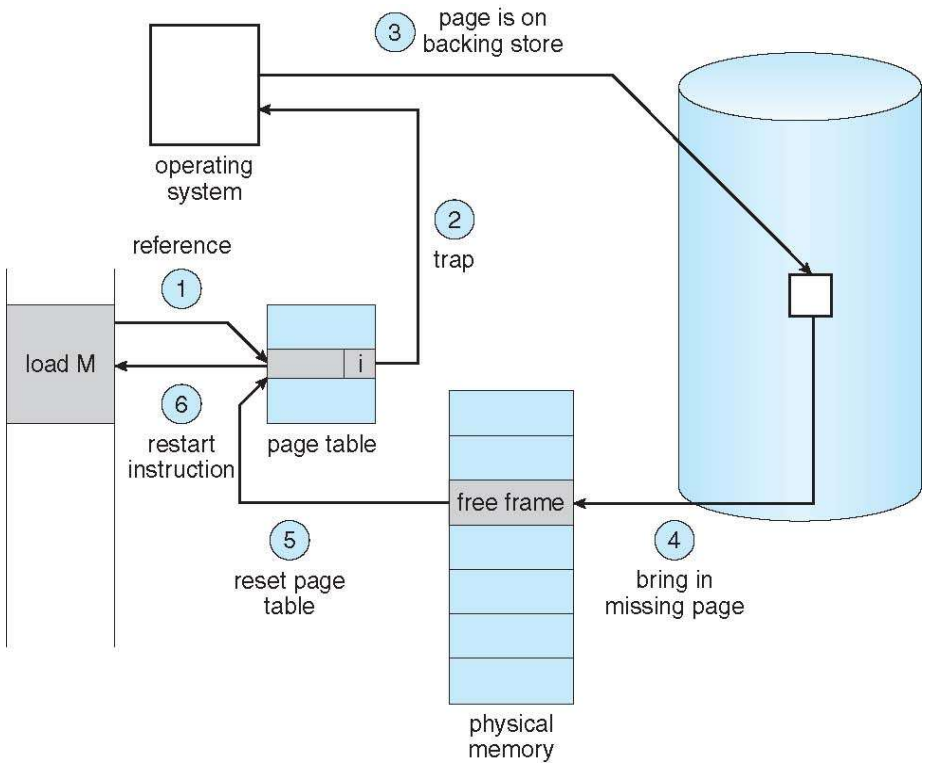
Page Fault

- **Page fault**



  - If there is a reference to an **invalid** page
  - Reference to that page will **trap (software interrupt)** to operating system

- OS handling page fault
  1. Operating system looks at page table to decide:
     - Illegal reference (address outside of virtual memory space) ⇒ abort
     - **validation bit = i** ⇒ Just not in memory
  2. Get empty frame from **physical memory**
  3. Swap page from **disk** into **physical memory frame**
  4. Reset **page table** to indicate **page** now in memory
     Set **validation bit = v**
  5. **Restart** the instruction that caused the page fault



Aspects of Demand Paging

- Hardware support needed for demand paging
  - **Page table** with valid / invalid bit
  - **Secondary storage**
    - Swap device with **swap space** (usually disk)
  - Restart Instruction (after page fault)

Performance of Demand Paging

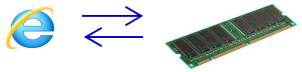
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

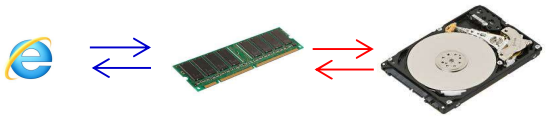
- **Effective Access Time (EAT)** for *demand paging*

EAT = (1 – *p*) x **memory access** + p x **page fault time**

page fault time = page fault overhead
+ swap page out
+ swap page in
+ restart overhead

Demand Paging Example





- Memory access time = **200 ns**
- Average page-fault service time = **8 ms** (=8,000,000 ns)

- EAT = (1 – p) x **200 ns** + p (**8 ms**)
  = (1 – p) x 200 + p x 8,000,000 (ns)
  = 200 + p x 7,999,800 (ns)
- If one access out of 1,000 causes a page fault (p=0.001), then
  EAT = 8,200ns (= **200ns** x 41)
  This is a slowdown by a factor of 41x!! (or 97.6% degradation)
- If we want performance degradation < 10 % (or EAT < 220ns)
  - 220 > EAT = 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses
- **<u>It is important to keep the page-fault rate low in a demand-paging system!</u>**

Chapter 9: Virtual-Memory Management

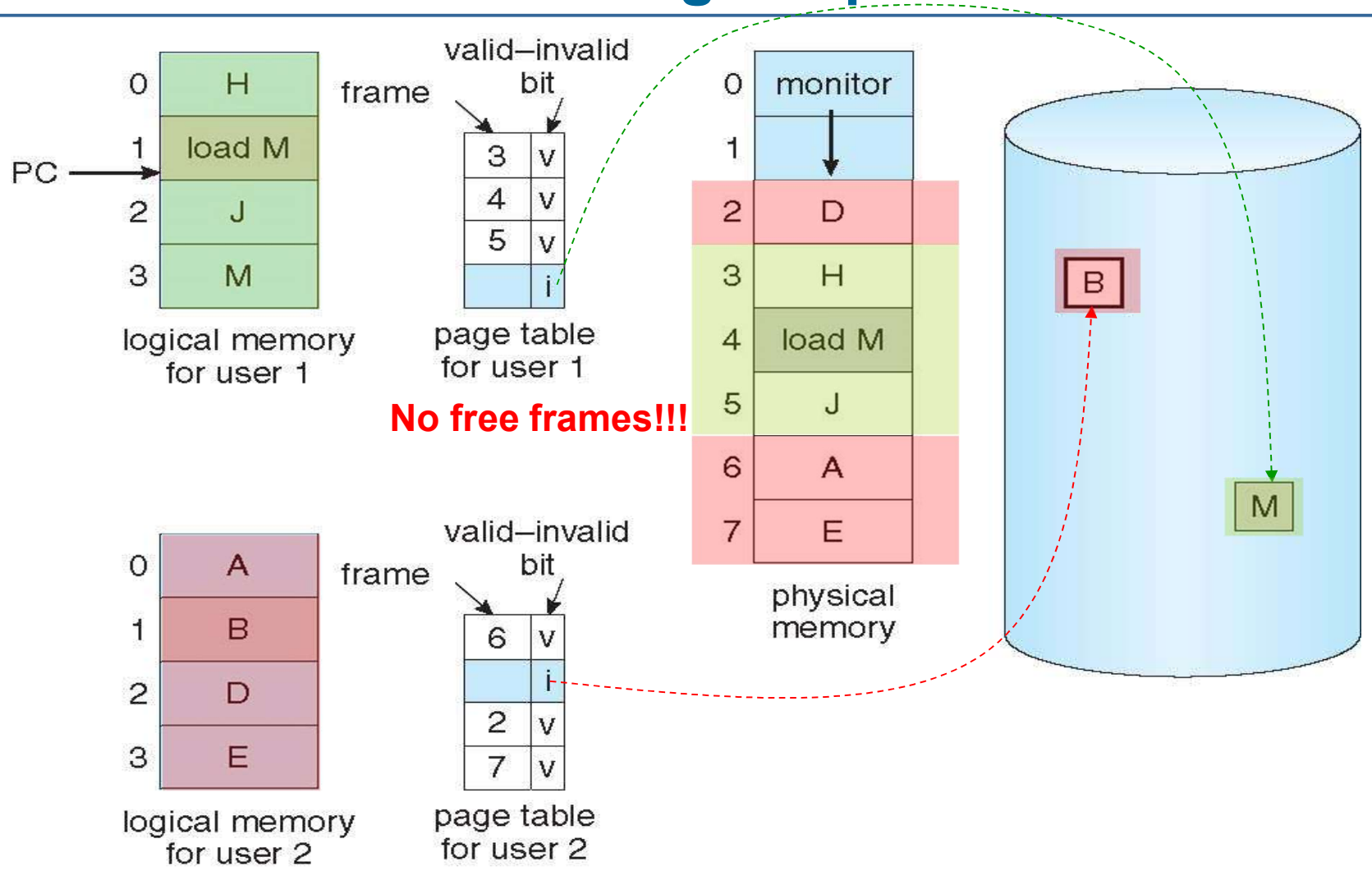
- Background
- Demand Paging
- **Page Replacement**
  - Basic Page Replacement
  - Modify (Dirty) Bit
- Page Replacement Algorithms

What Happens if There are no Free Frames?

- All memory may be in use
  - Multiprogramming – multiple processes share the memory
  - Page fault → OS finds the desired page on the disk → find the free-frame
  - If NO free frames on the **free-frame list** ? – i.e., physical memory is full!

- **Page replacement**
  - Find some page in memory, but not really in use, **page out**
  - Performance – want an algorithm which will result in minimum number of page faults
    - **Page replacement Algorithms**

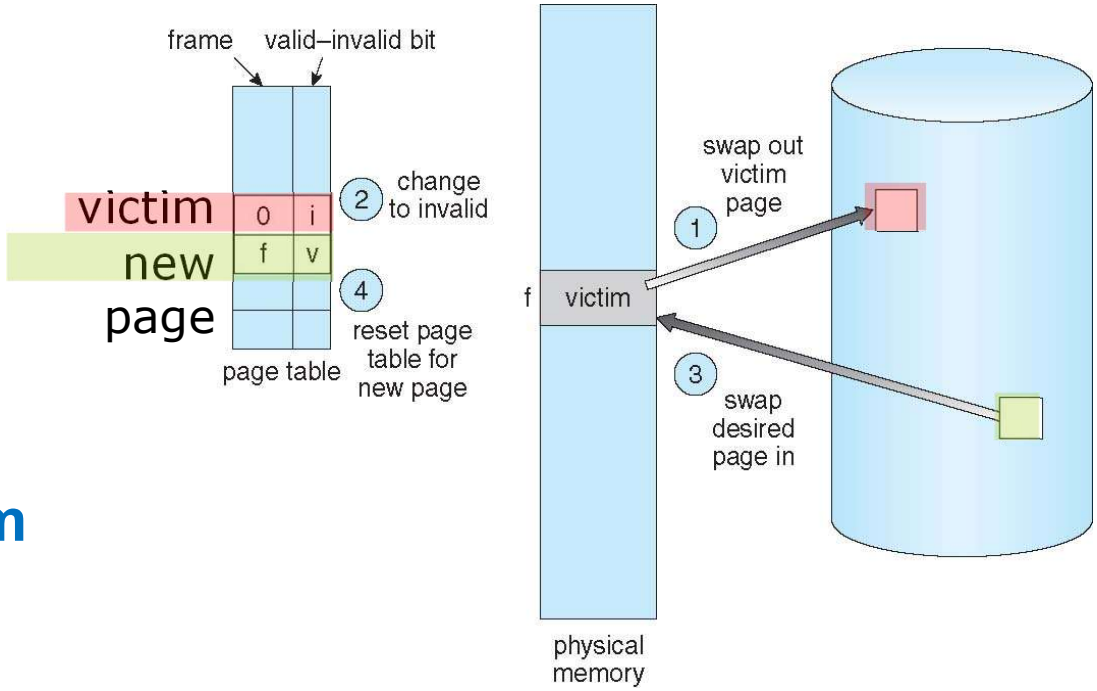
Need For Page Replacement



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**



- Read the desired page into the (newly) free frame. Update the page and frame tables.
- Restart the process

Modify Bit (= Dirty Bit)

- If no frames are free → **TWO** page transfers are required
  - One page out and one page in (see previous slide)
- Can we reduce the number of disk writes (page-outs)?
- Solution: **Modify bit (dirty bit)**
  - Initially set to zero
  - Set the bit (to 1) whenever any word or byte in the page is modified
    - only modified pages are written to disk – unmodified pages can just be deleted without writing on disk (it has **not** changed!)

  ➔ can reduce the page-transfer overhead

Benefits of Virtual Memory (using Demand Paging)

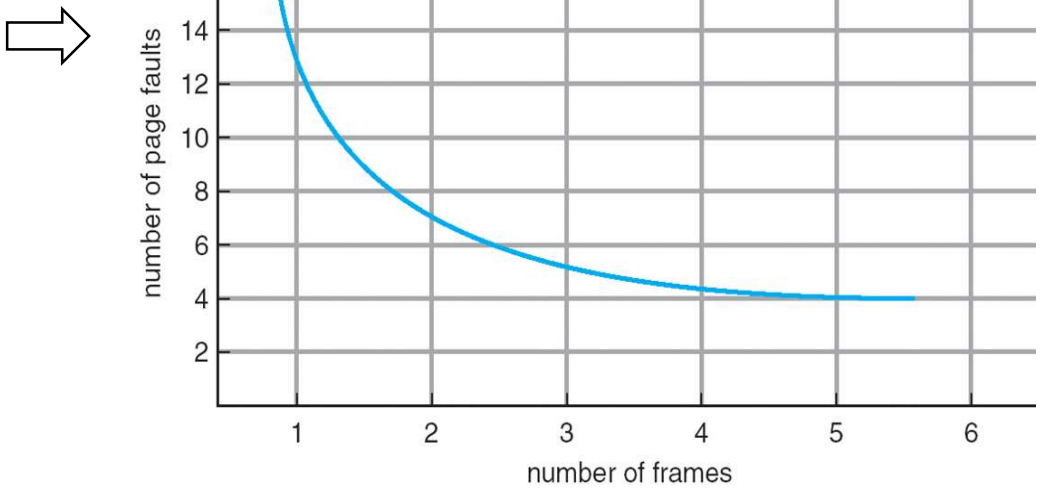
- **Protection**
  - A bug in one process can corrupt memory in another
  - A process can only access the virtual address (pages)
- **Transparency**
  - A process shouldn't require particular physical memory
  - A process sees a contiguous virtual memory space (pages)
- **Resource exhaustion**
  - Sum of sizes of all processes often greater than physical memory
  - **Demand paging**
    - e.g., If a process is 20 pages, we can execute with 10 frames and the rest in the swap space (disk) – use demand paging and page-replacement

Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Page Replacement
- **Page Replacement Algorithms**
  - FIFO Page Replacement
  - Optimal Page Replacement
  - LRU Page Replacement

Page and Frame Replacement Algorithms

- **Page-replacement algorithm**
  - Which frame should be replaced?
  - Want lowest page-fault rate
- Evaluate algorithm by computing the number of page faults
  - The number of page faults will decreases as the number of frames available increases

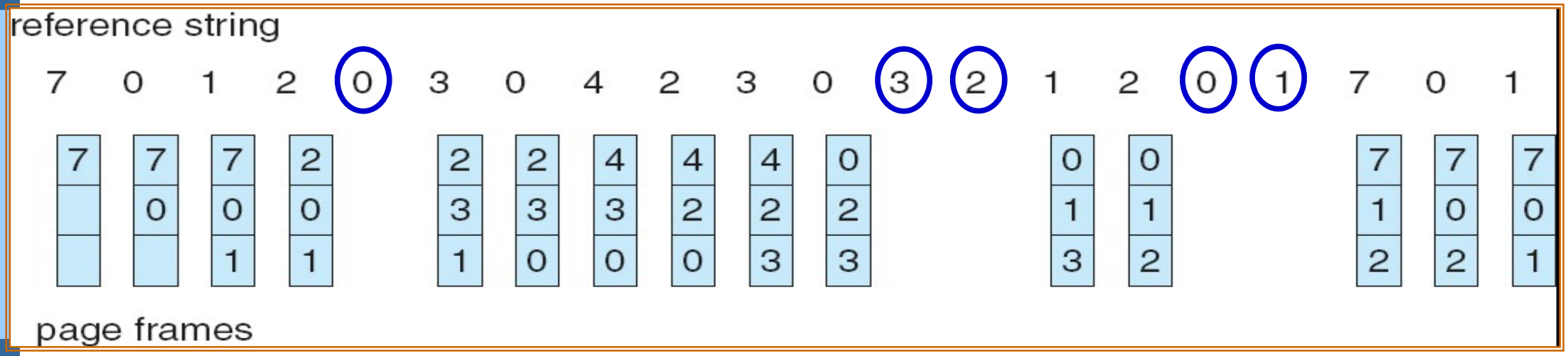


Page Replacement Algorithms

- FIFO page replacement
- Optimal page replacement
- LRU (Least-Recently-Used) algorithm
  - clock, stack, reference-bit

First-In-First-Out (FIFO) Algorithm

- When a page must be replaced, the oldest page is chosen



15 Page Replacements!!

- How to track ages of pages?
  - Just use a FIFO queue

Problem of FIFO Algorithm

- Example 2
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

-3 frames (3 pages can be in memory at a time per process)

1	**1**	4	5	
2	**2**	1	3	9 page faults
3	**3**	2	4	

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

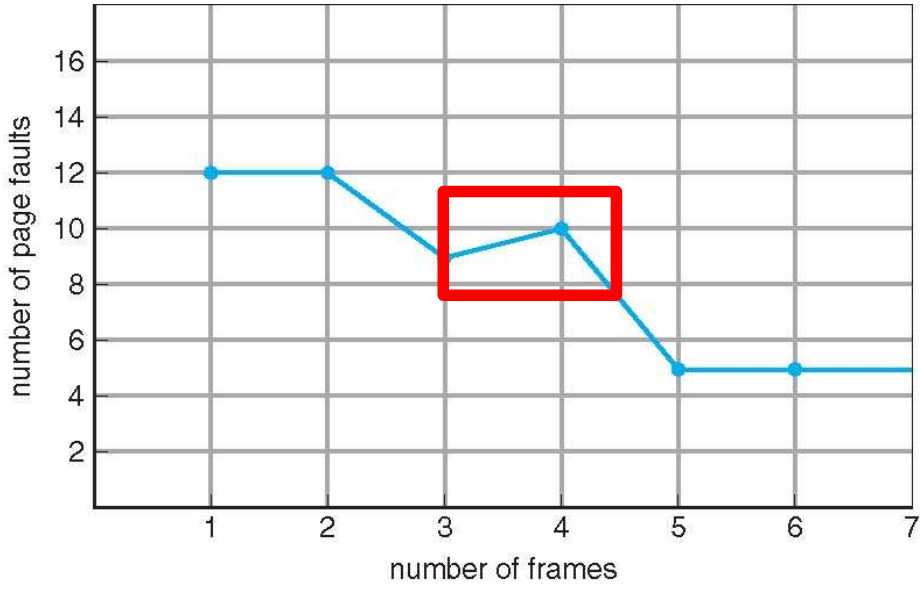
-4 frames:

1	**1**	5	4	
2	**2**	1	5	10 page faults
3	**3**	2		
4	**4**	3		

Why did this happen???

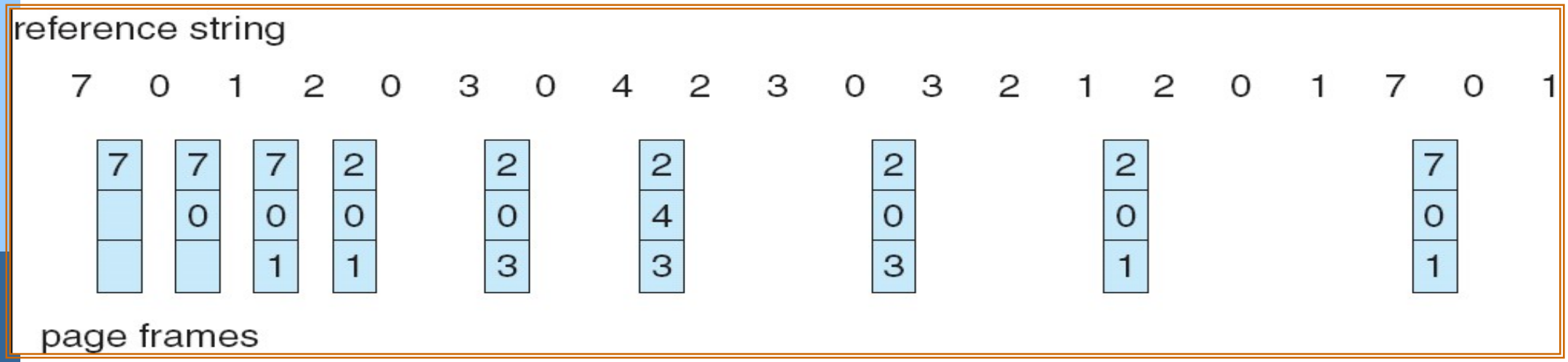
FIFO Illustrating Belady's Anomaly

- FIFO replaces the **oldest** page – is this always good?
  - An old page may contain a **frequently used variable**
- **Belady's anomaly**
  - Sometimes, page-fault may increase as the number of allocated frames increase!



Optimal Algorithm

- Algorithm that has the lowest page-fault rate
- Replace page that **will *not*** be used for longest period of time
  - This is a design to guarantee the lowest page-fault rate for a fixed number of frames



9 Page Replacements!!

Optimal Algorithm

- Unfortunately, the optimal page-replacement algorithm is difficult to implement
  - Why?
    - Require **future knowledge** of the reference string – We do not know the future!

- Mainly used for comparison studies
  - Evaluating a new algorithm
  - e.g. “the new algorithm is within 12.3% of optimal at worst and within 4.7% on average”

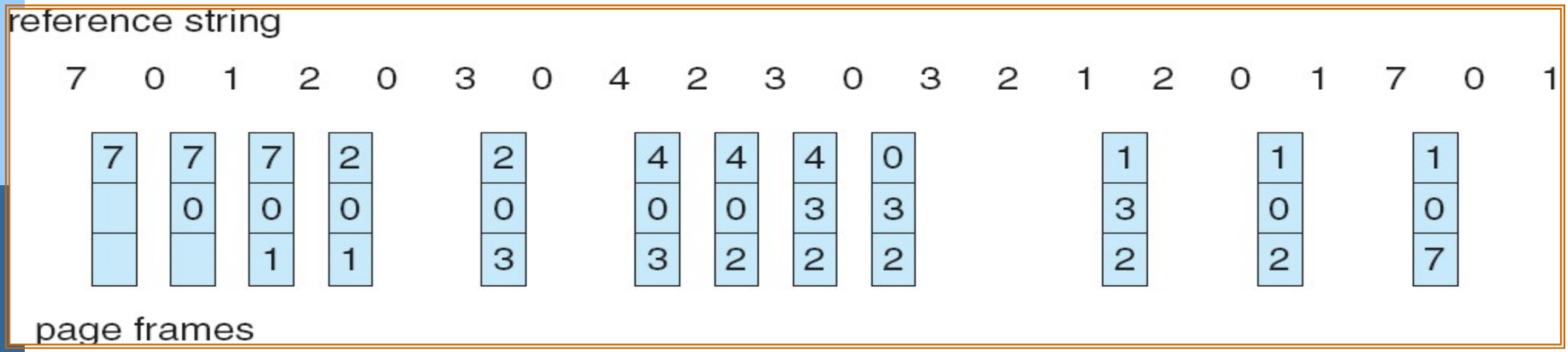
Least Recently Used (LRU) Algorithm

- FIFO: Use time when a page **was brought** into the memory
- OPT: Use time when a page is to be **used**
- Let us predict the future by using the past!

- **Least Recently Used (LRU) Algorithm**
  - Use past knowledge rather than future
  - **Replace** page that has **not been used** in the most amount of time
  - Generally good algorithm and frequently used

Least Recently Used (LRU) Algorithm

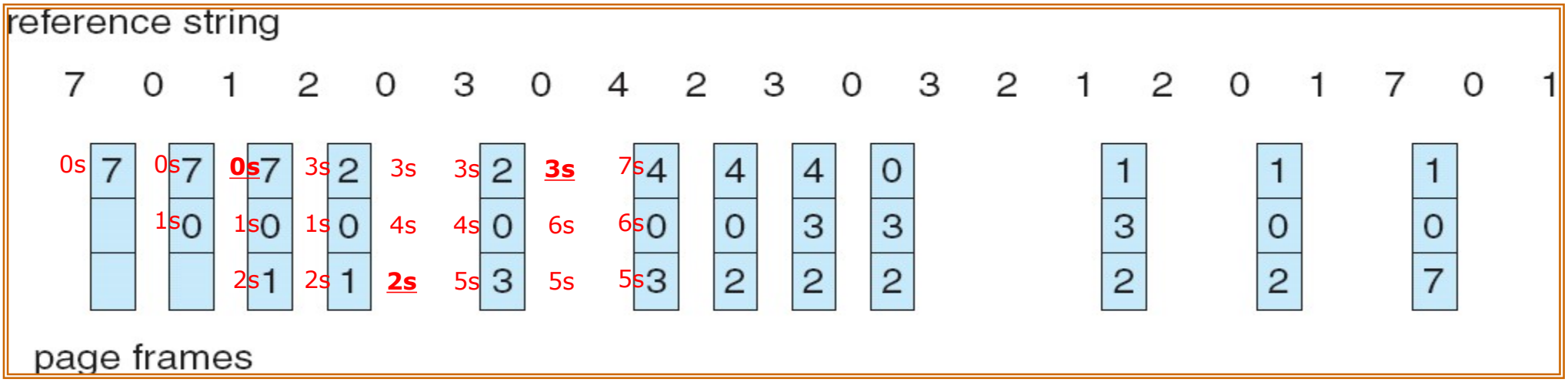
- LRU replacement associates with each page the time of that page's last use
- Replace the page that has not been used for the longest period of time



12 Page Replacements!!

Implementation of LRU Algorithm

- The major problem is **how to implement LRU replacement**
  - Use ***counter*** or ***stack***
  - LRU implementation requires H/W support (due to speed)
- Counter implementation example

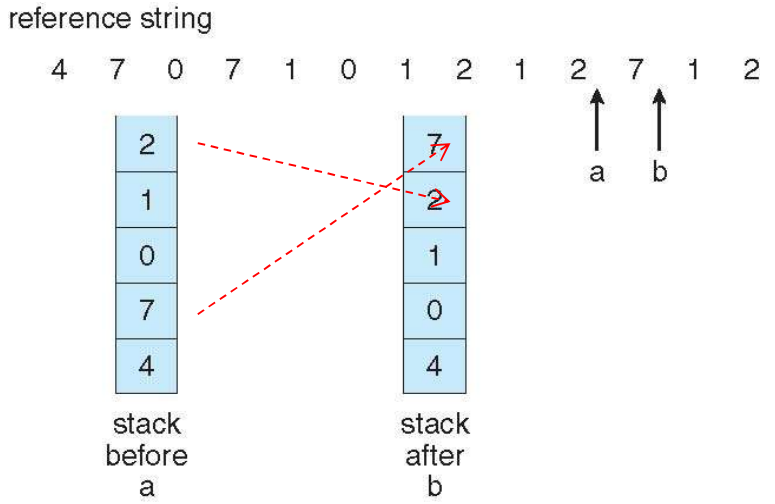


Implementation of LRU Algorithm

- 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 – replace page with **smallest** timer value
    - Con1: Need to **search** to replace page
    - Con2: Timer has to be updated for each memory reference

Implementation of LRU Algorithm

- Stack implementation
  - Page referenced:
    - move it to the top
  - Use doubly-linked list
    - Why not array?
  - Con: Replacement requires 6 pointers to be changed at worst
  - No search for replacement



LRU Implementation

- Updating of **clock fields** or **stack** must be done for ***every*** memory reference
- LRU needs special hardware and still slow
  - Timer and stack approach use too much resources
- LRU Approximation: **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)
  - In this method, although we can not know the **order** of page reference, we can know that whether the page is **referenced or not**

Summary

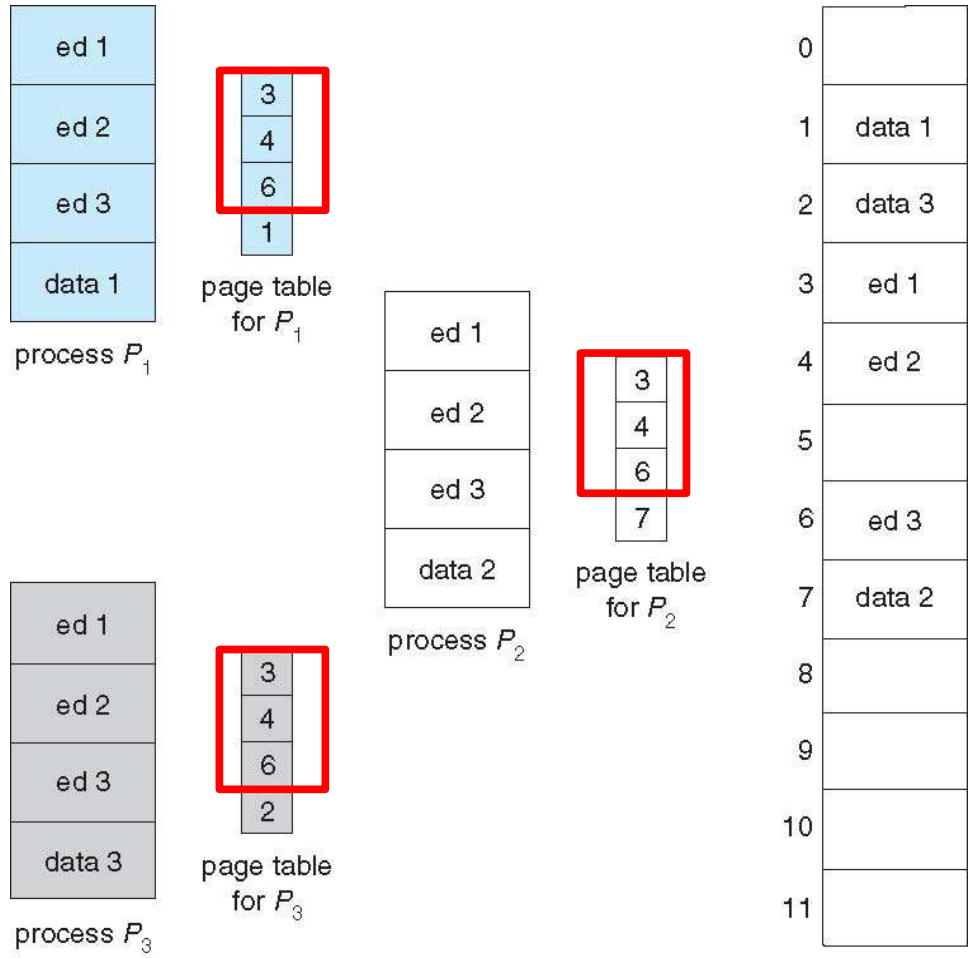
- Virtual memory is how we stuff large programs into small physical memories.
- We perform this magic by using demand paging, to bring in pages only when they are needed.
- But to bring pages into memory, means kicking other pages out, so we need to worry about paging algorithms.

Chapter 9: Virtual-Memory Management

- Appendix
  - Sharing and Memory-Mapped Files

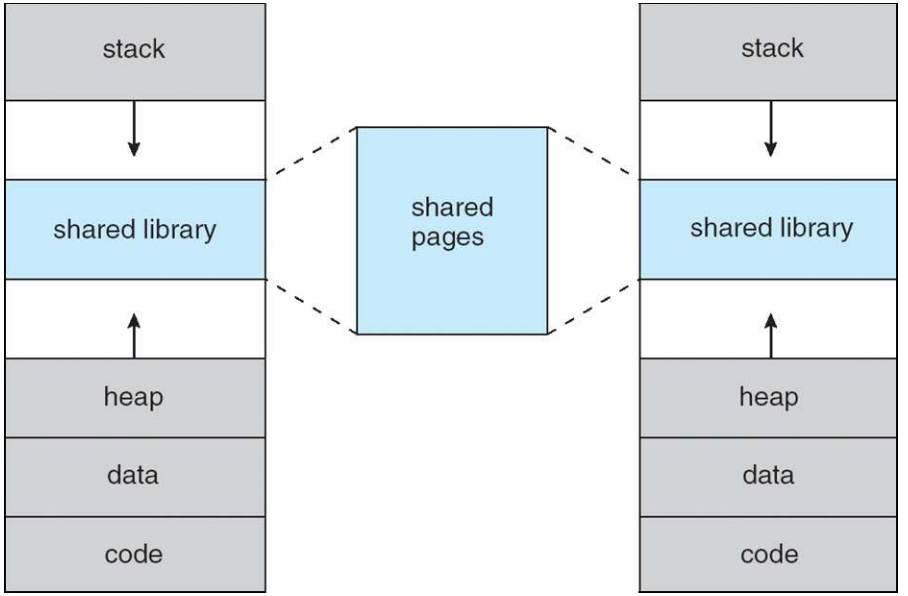
Shared Page

- Shared code
  - One copy of code shared among processes
    - Example: 40 users run same program using 150KB code and 50KB data
      - need total 200 x 40 = 8,000KB physical memory
    - Share the code!
      - need total 150 + 50x40 = 2,150KB physical memory
  - IPC, Multiple threads sharing the same memory space



Virtual Address Space

- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
  - System libraries can be shared
    - part of its virtual address space, but actual physical memory are shared
  - Shared memory by processes (IPC)
    - each process considers the shared memory as part of their virtual address space



fork(): 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
  - when the stack or heap for a process must expand
  - or when there are copy-on-write pages