CSC 4320/6320: Operating Systems

Georgia State University

Chapter 10: Virtual Memory-II

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Chapter 10: Virtual Memory

Georgia State University

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- ~~Thrashing~~
- ~~Memory-Mapped Files~~
- ~~Allocating Kernel Memory~~
- ~~Other Considerations~~
- ~~Operating-System Examples~~

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- ~~Describe how Linux, Windows 10, and Solaris manage virtual memory.~~
- ~~Design a virtual memory manager simulation in the C programming language.~~

Review: Performance of Demand Paging

Georgia State University

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- 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) x memory access
  + p (page fault overhead
  + swap page out
  + swap page in)

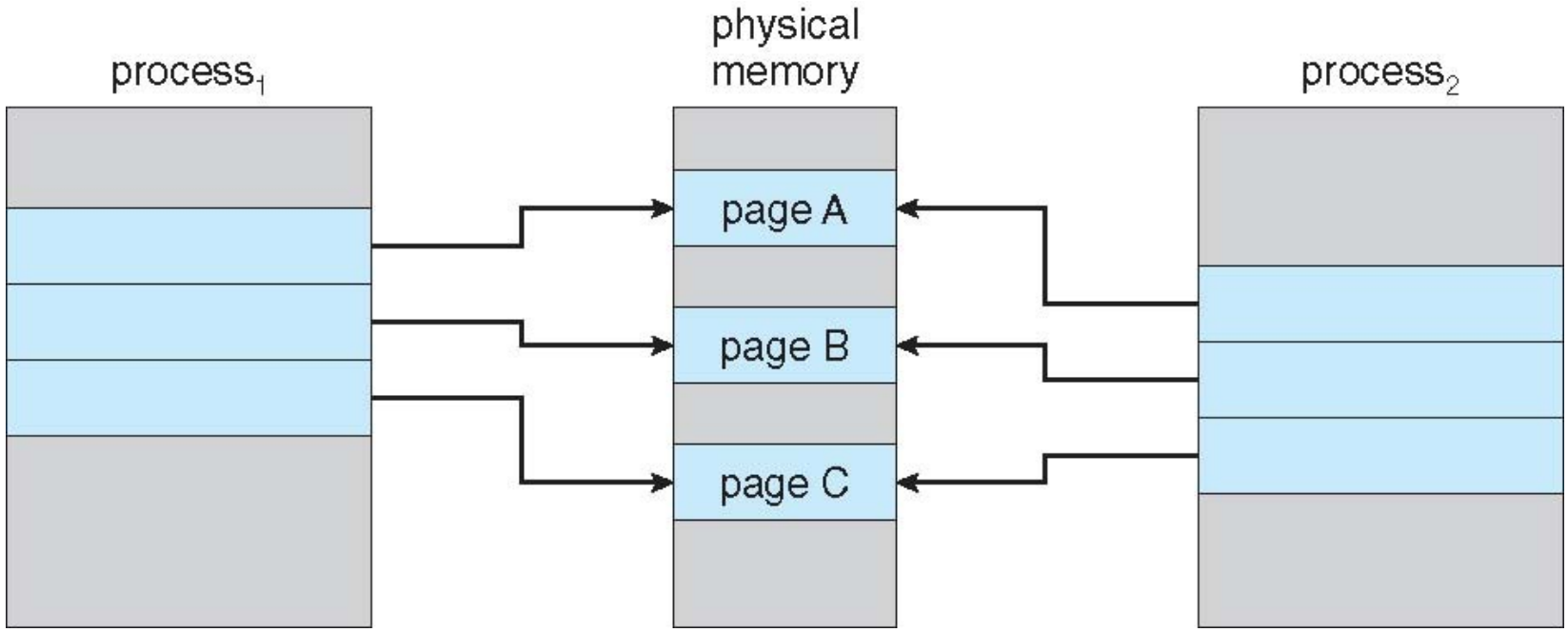
Review: Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

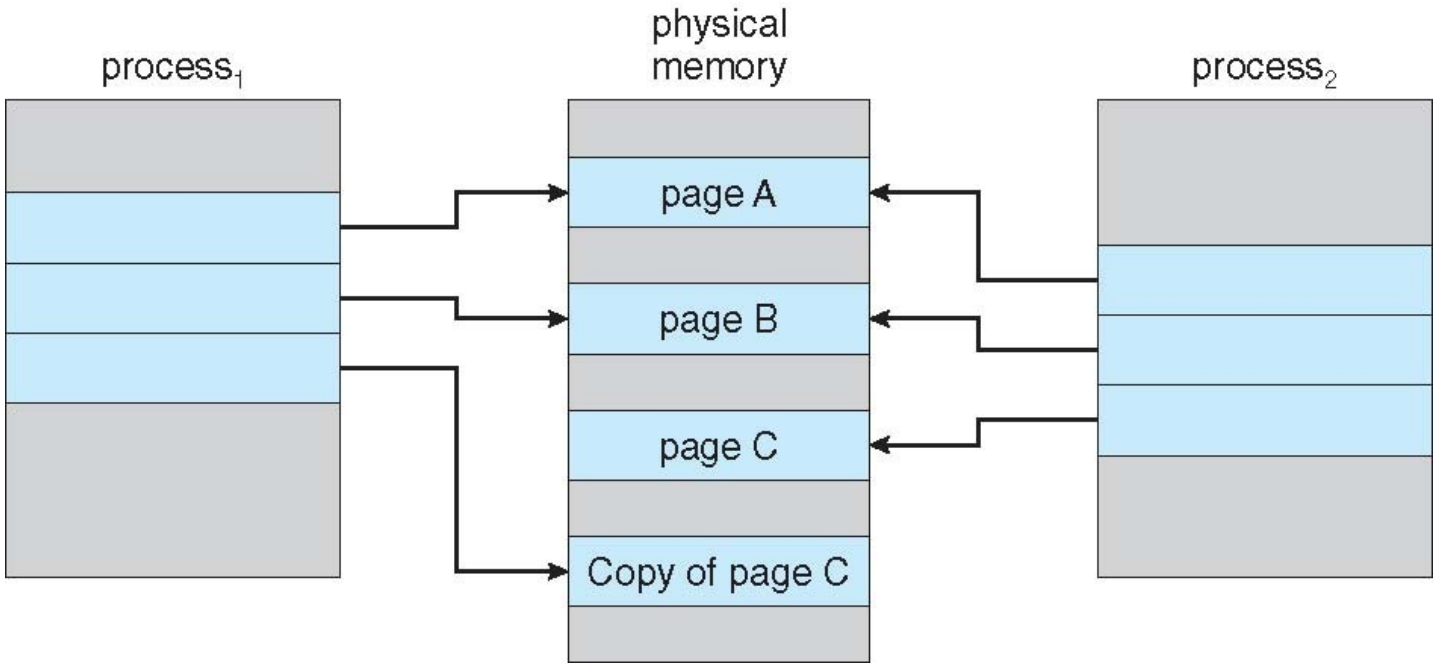
Copy-on-Write

- **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
  - When a child attempts to modify a page, OS will obtain a frame from the free-frame list and create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process.
- 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
  - Pool should always have free frames for fast demand page execution
  - Why zero-out a page before allocating it?
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec() (sharing happens only for a brief period)
  - Very efficient
  - Do not modify any variable, stack, or heap. It will affect the parent's space!!

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Questions

1. The vfork() system call in UNIX _____.

A) allows the child process to use the address space of the parent ✓

B) uses copy-on-write with the fork() call

C) is not intended to be used when the child process calls exec() immediately after creation

D) duplicates all pages that are modified by the child process

2. _____ allows the parent and child processes to initially share the same pages, but when either process modifies a page, a copy of the shared page is created.

A) copy-on-write ✓

B) zero-fill-on-demand

C) memory-mapped

D) virtual memory fork

3. On systems that provide it, vfork() should always be used instead of fork().

True

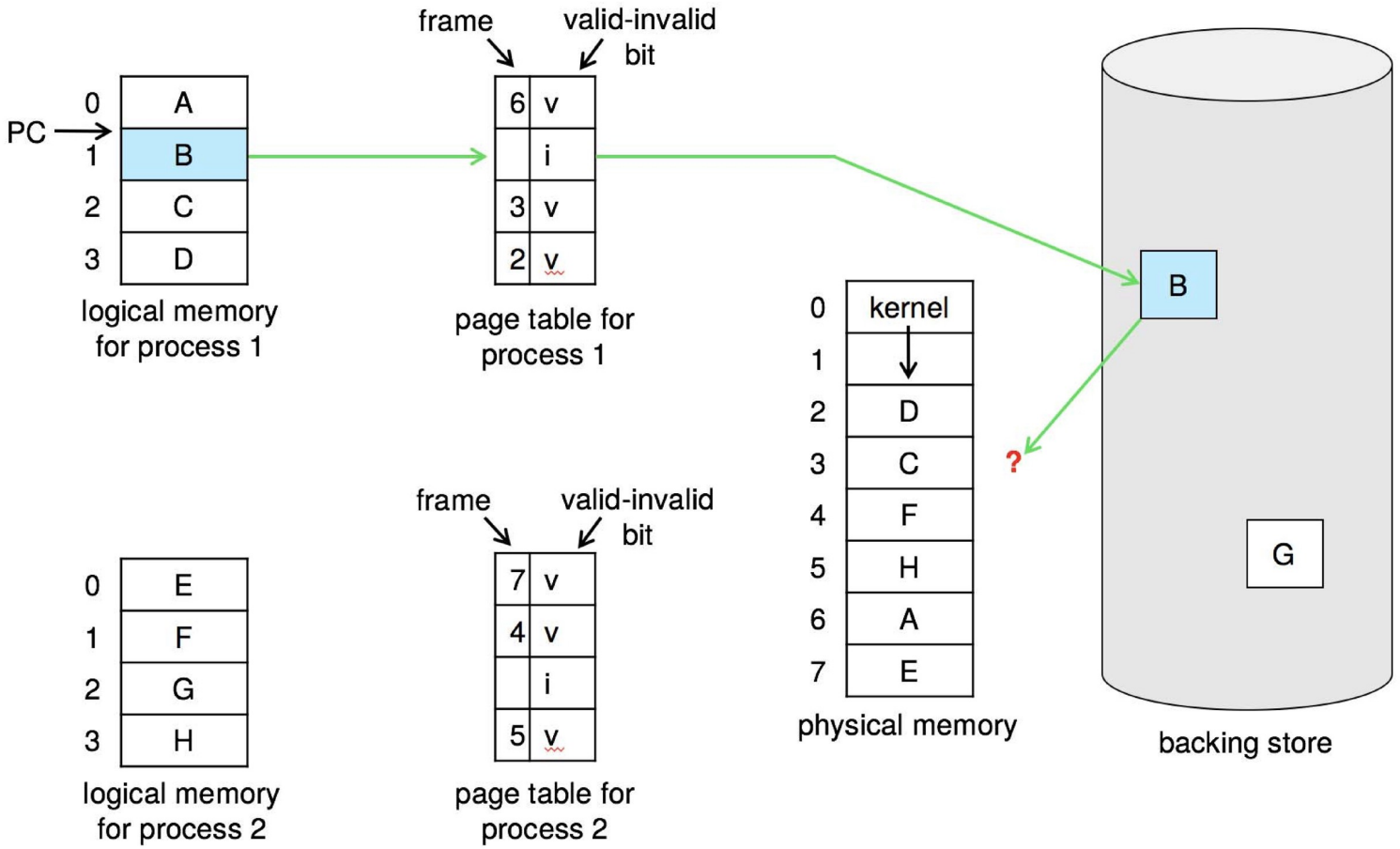
False ✓

Let's do some background discussion

Georgia State University

- 40 frames available in physical memory
- 10 pages per process (only used 5)
- 8 processes can run, rather than the 4 that could run if each required ten frames (5 of 10 were never used).
- we are over-allocating memory to increase degree of multiprogramming
- What if suddenly all processes (let's say 6) try to use all ten of its pages, resulting in a need for sixty frames?
- how much memory to allocate to I/O and how much to program pages is a significant challenge.

Need For Page Replacement



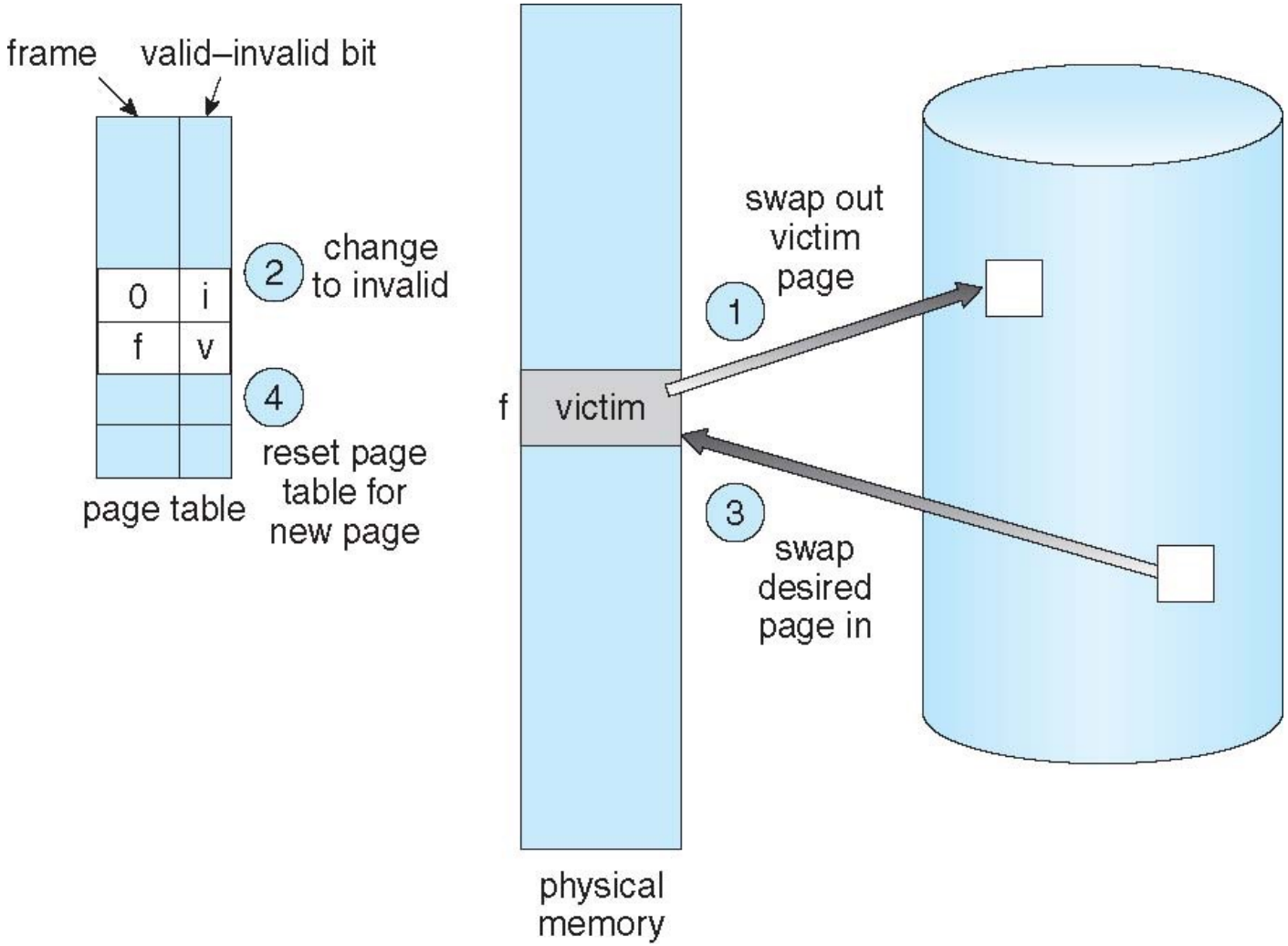
a page fault occurs but there are no free frames on the free-frame list; all memory is in use.

Basic Page Replacement

1. Find the location of the desired page on disk
2. 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
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory. That is, page replacement is basic to demand paging.

Page and Frame Replacement Algorithms

Georgia State University

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm**
  - Which frames to replace (when replacement is required)
  - 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
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

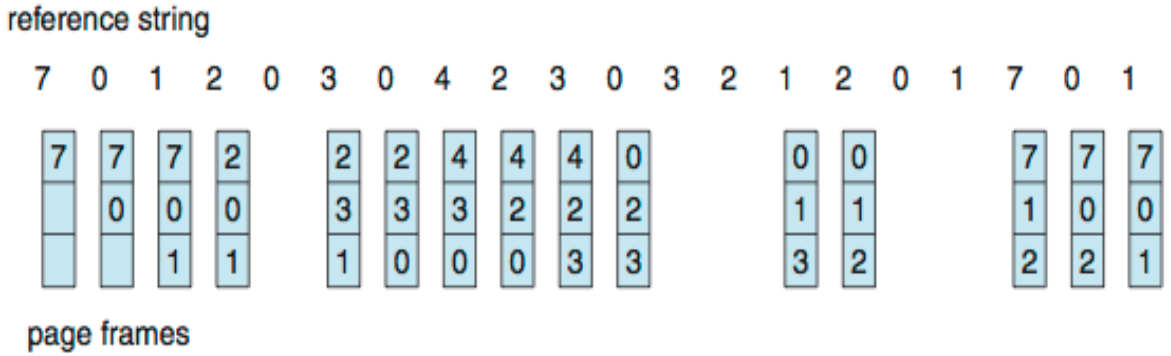
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Calculate page faults

- If we have this reference string 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1. How many page faults with 1 physical frame? **11 faults**
- if we had three or more frames, we would have only three faults—one fault for the first reference to each page. **3 faults**

First-In-First-Out (FIFO) Algorithm

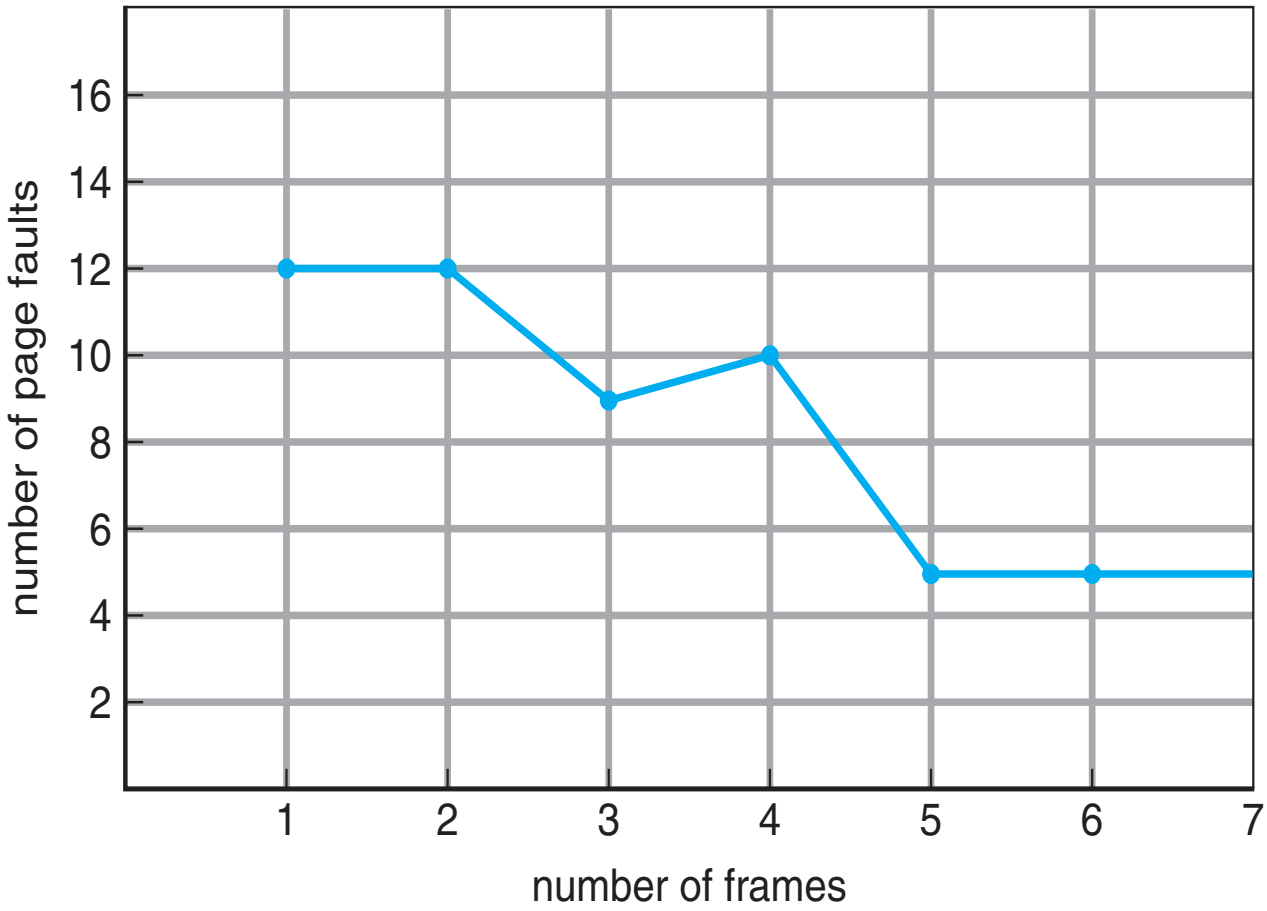
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - a bad replacement choice increases the page-fault rate and slows process execution
  - Adding more frames can cause more page faults! (Not always!)
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly

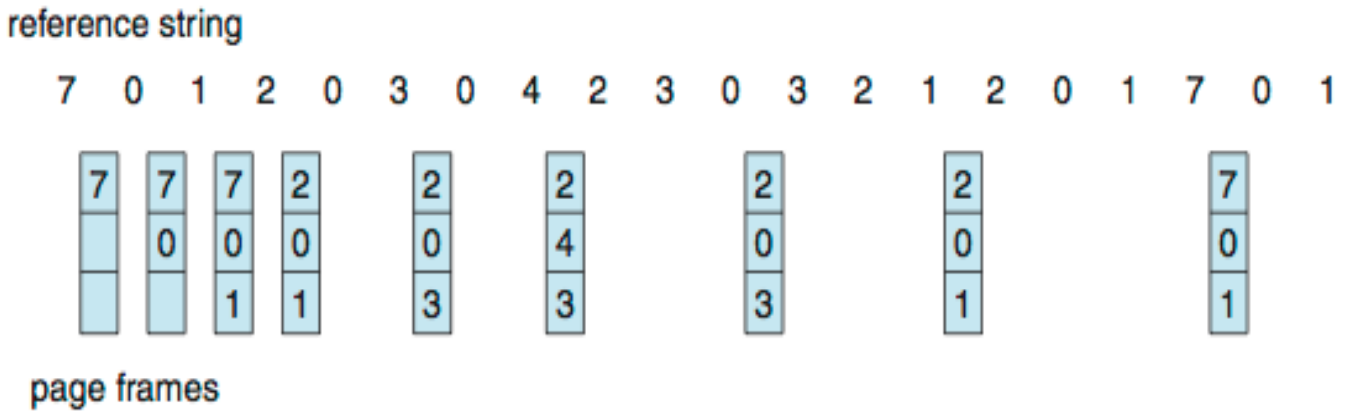


Reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)

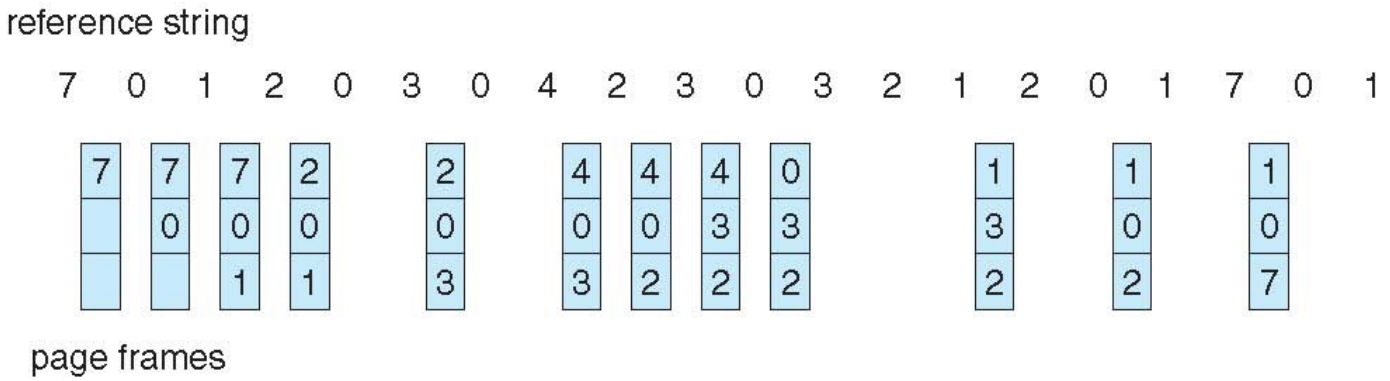
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - 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



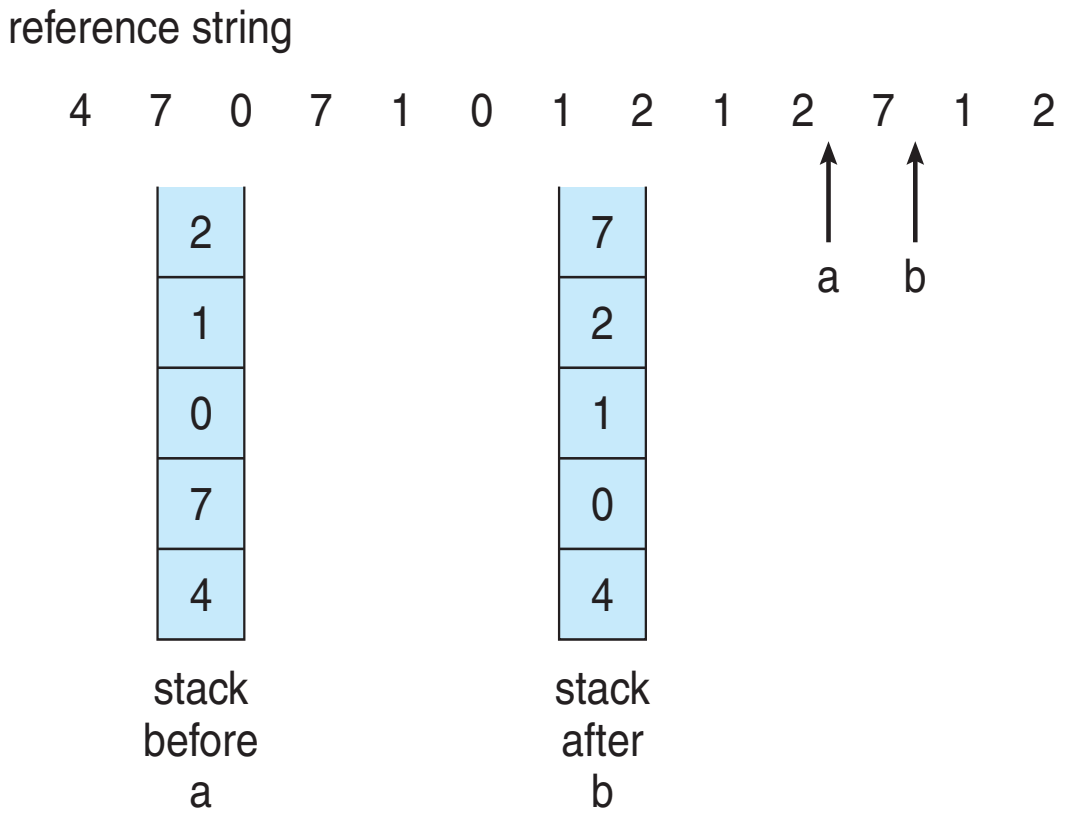
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - each page-table entry has a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference.
  - Whenever a reference to a page is made, the time-of-use field in the page-table entry is updated.
  - 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 doubly linked list:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use Of A Stack to Record Most Recent Page References



Questions

1. Suppose we have the following page accesses: 1 2 3 4 2 3 4 1 2 1 1 3 1 4 and that there are three frames within our system. Using the LRU replacement algorithm, what is the number of page faults for the given reference string?

A) 14
B) 13
C) 8 ✓
D) 10

2. Belady's anomaly states that _____.

A) giving more memory to a process will improve its performance
B) as the number of allocated frames increases, the page-fault rate may decrease for all page replacement algorithms
C) for some page replacement algorithms, the page-fault rate may decrease as the number of allocated frames increases
D) for some page replacement algorithms, the page-fault rate may increase as the number of allocated frames increases ✓

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

- s_i = size of process p_i
- $S = \sum s_i$
- m = total number of frames
- a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Global vs. Local Allocation

Georgia State University

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory