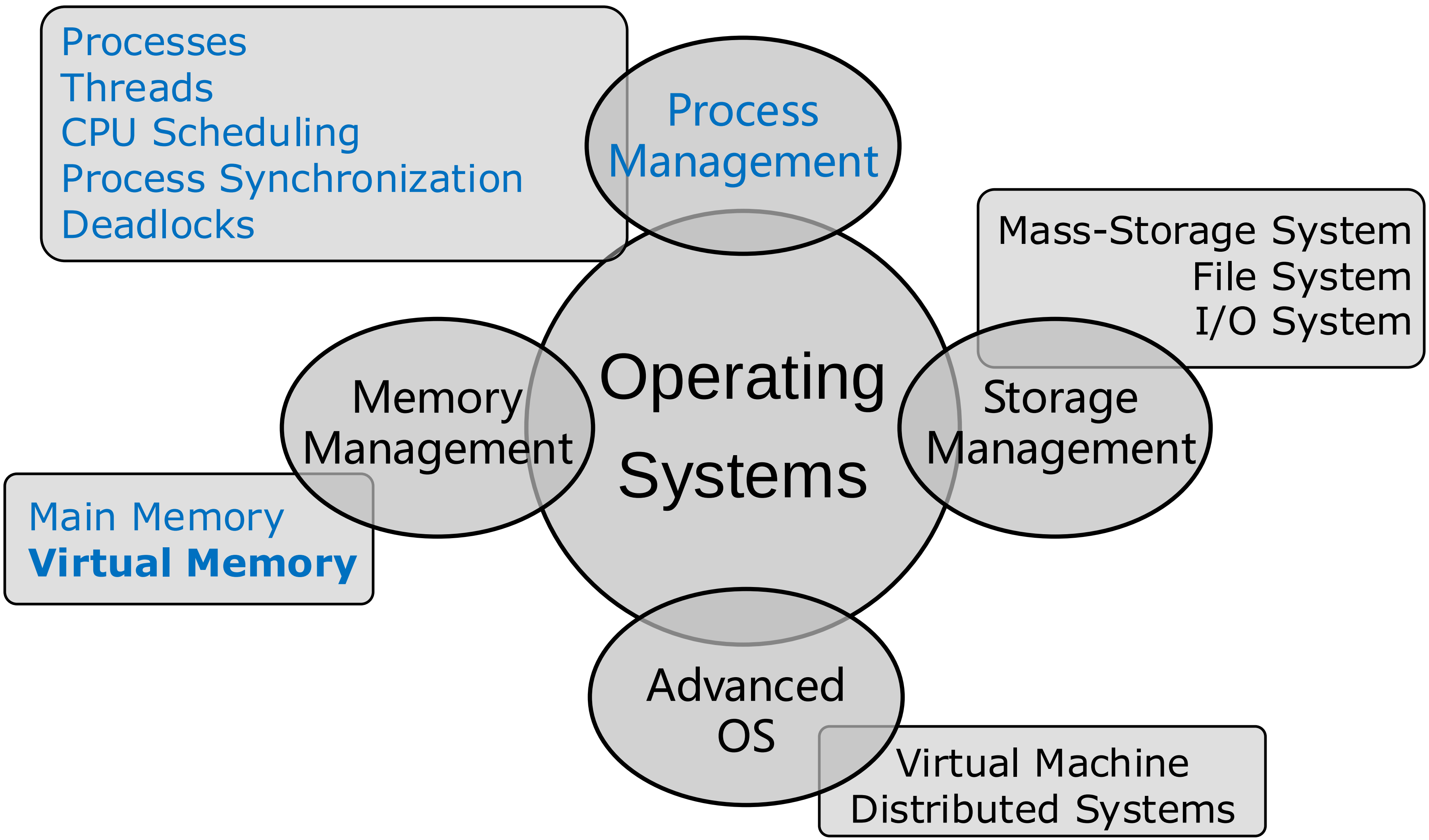
Virtual Memories

Shengzhong Liu
Department of Computer Science and Engineering
Shanghai Jiao Tong University

Operating System Topics



Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames

Background

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 the same time
- Consider ability to **execute partially-loaded program**
  - Program no longer constrained by limits of physical memory

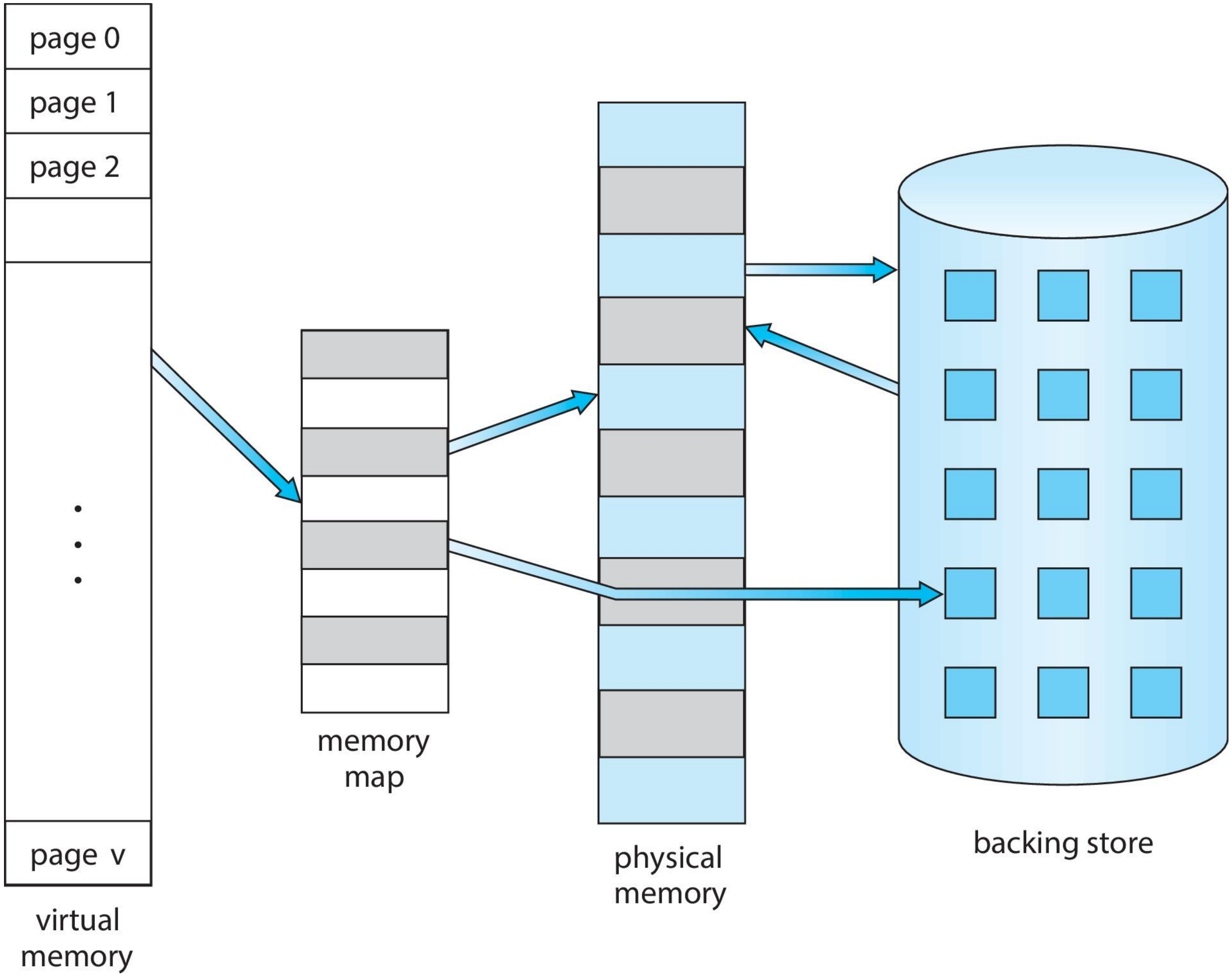
Virtual Memory

- **Virtual Memory** – separation of **user logical memory** from **physical memory**
  - Only part of the program needs to be in memory for execution

- Benefits of virtual memory:
  - Address space:
    - Logical address space can be much larger than physical address space
    - Allows memory address spaces to be shared by several processes
  - Efficiency:
    - More programs running concurrently
    - Allows for more efficient process creation
    - Less I/O needed to load or swap processes

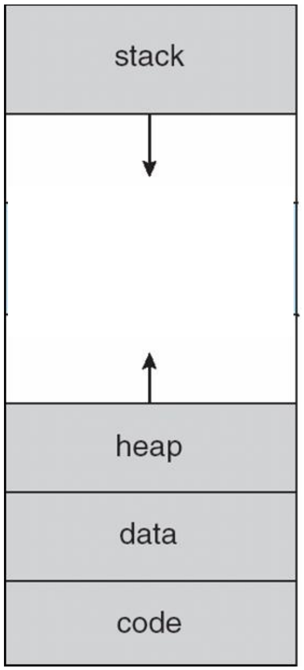
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger Than Physical Memory



Virtual Address Space

- **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
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
  - System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during `fork()`, speeding process creation



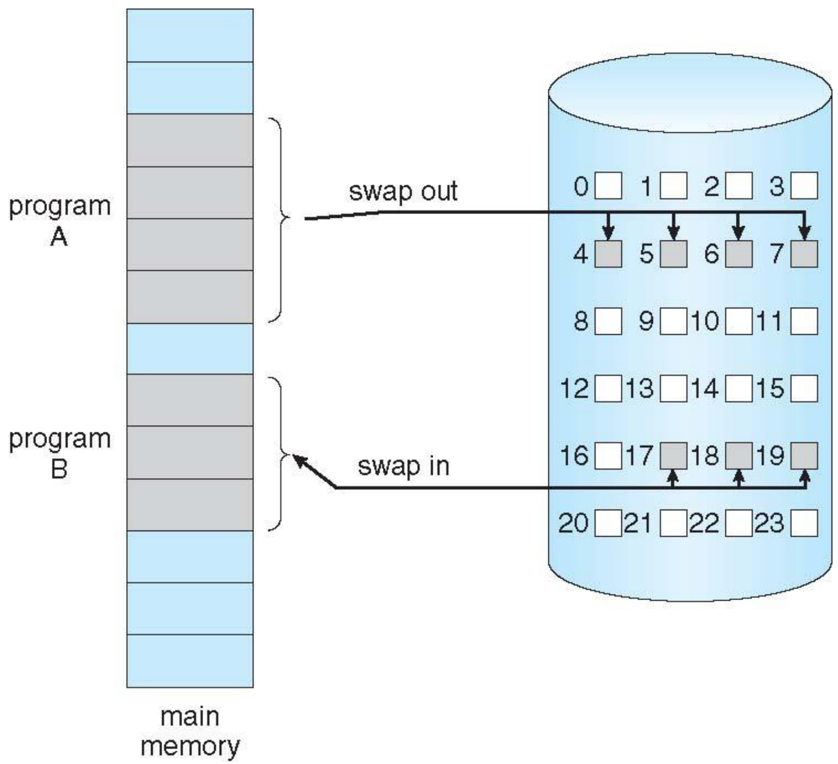
Demand Paging

Demand Paging

- Instead of bringing the entire process into memory at load time, 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
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper (pager)** – never swaps a page into memory unless page will be needed

Swap Paged Memory to Disk Space

- With swapping, pager guesses which pages will be used before swapping out
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are **memory resident**
  - No difference from non demand-paging
- If pages needed are **not memory resident**
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer changes



Valid-Invalid Bit

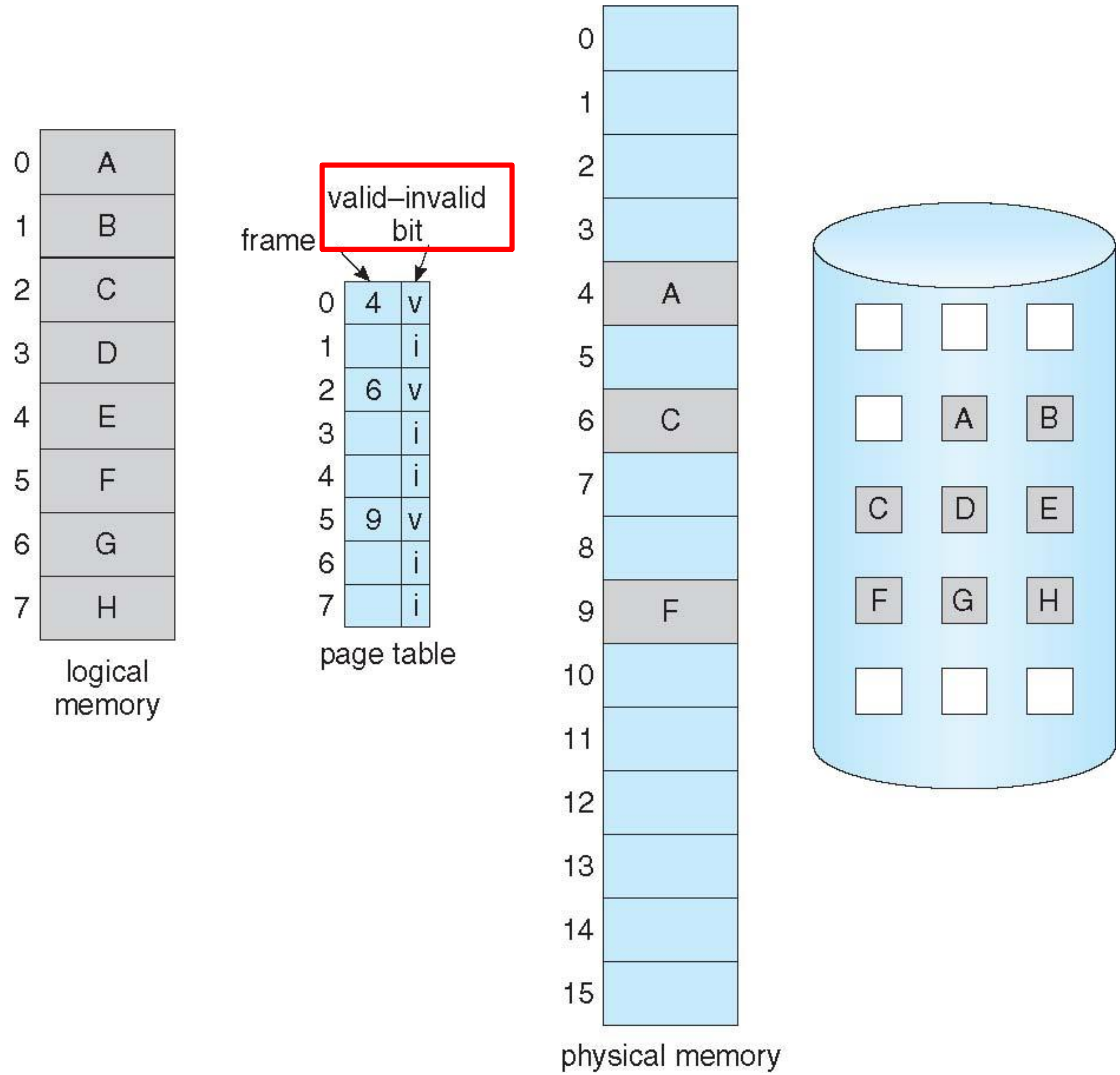
- With each page table entry a **valid–invalid** bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially, valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in the page table entry
  is **i** ⇒ **page fault**

Page Table with Pages Not in Main Memory

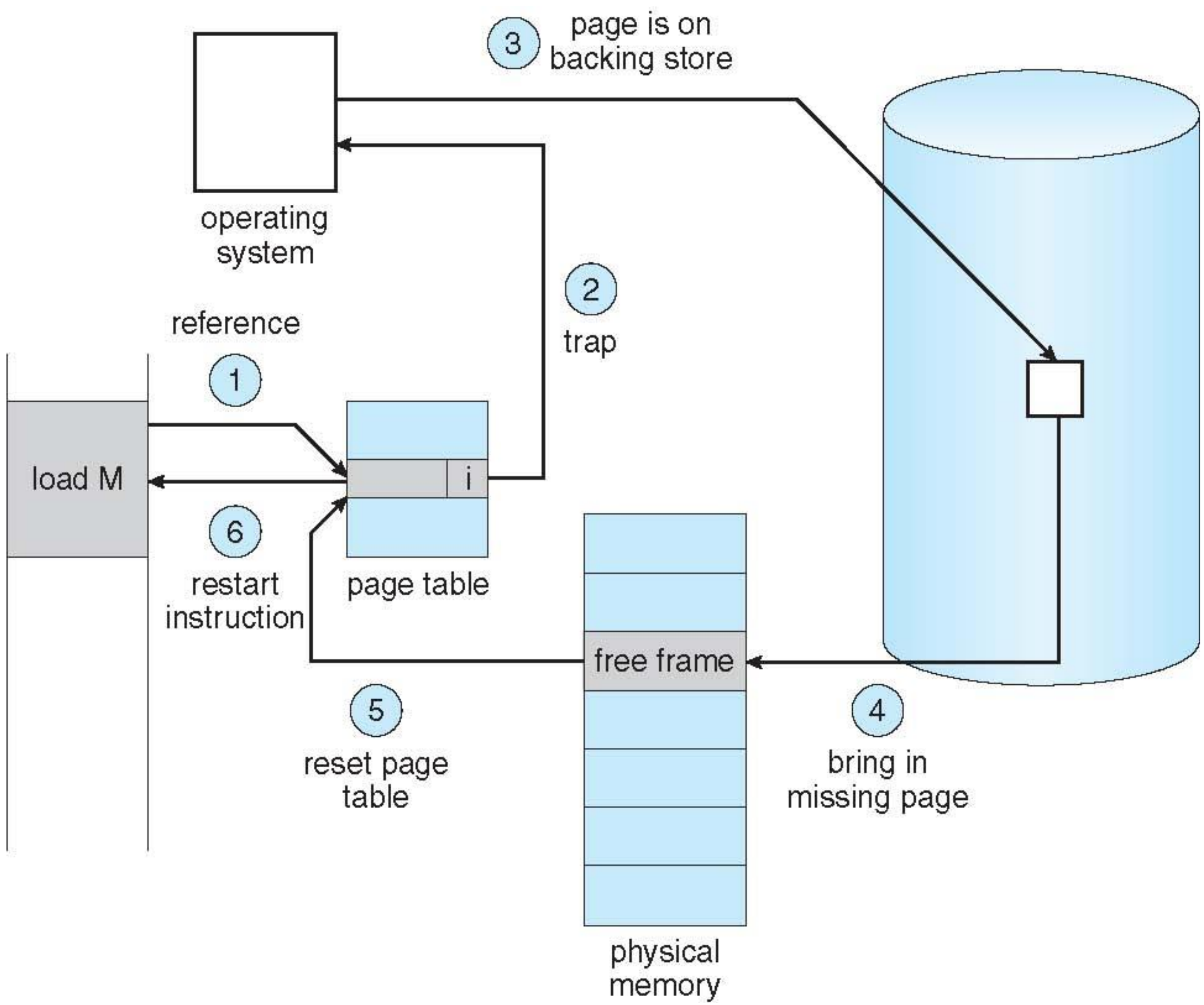


Page Fault

- If there is a reference to a page and the page is not in memory, the reference will trap to operating system: **Page fault!**

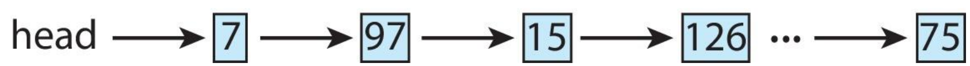
1. Operating system looks at the page table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list**
  - A pool of free frames for satisfying such requests.



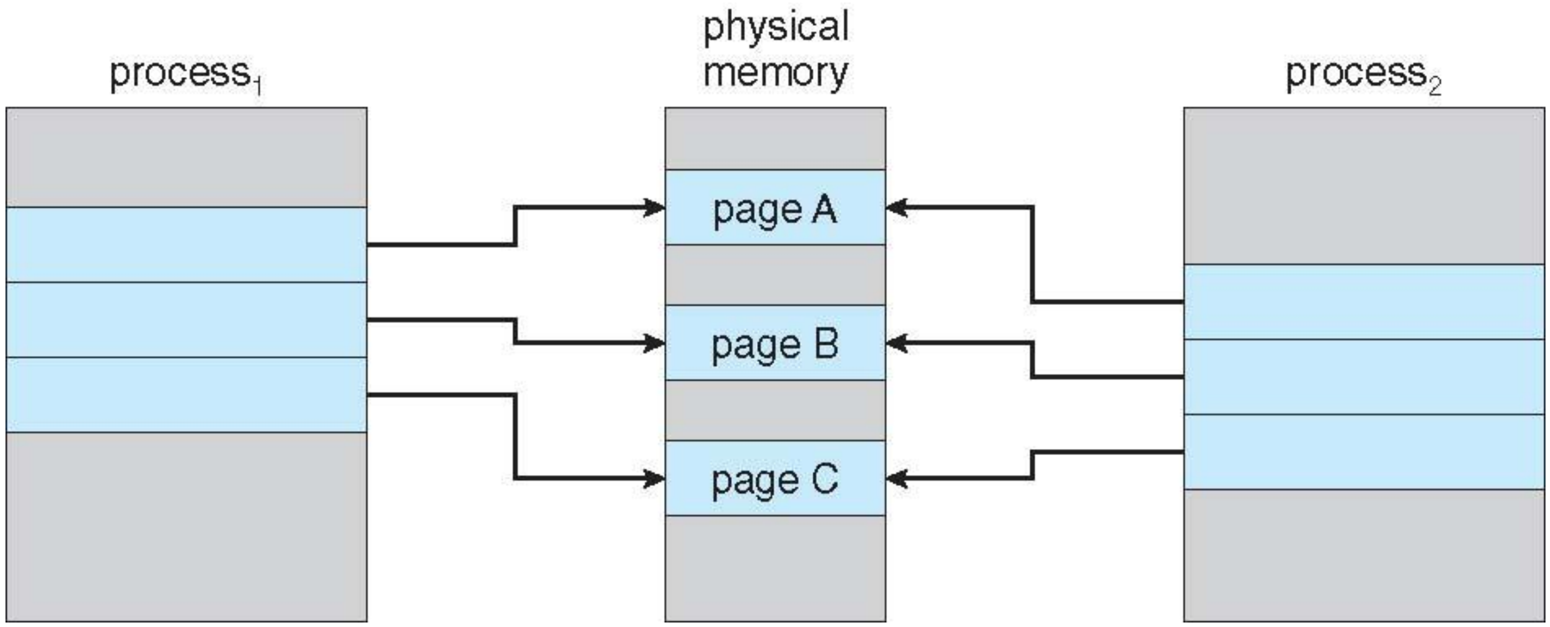
- OS typically allocates free frames using a technique known as **zero-fill-on-demand**
  - The content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Copy-on-Write

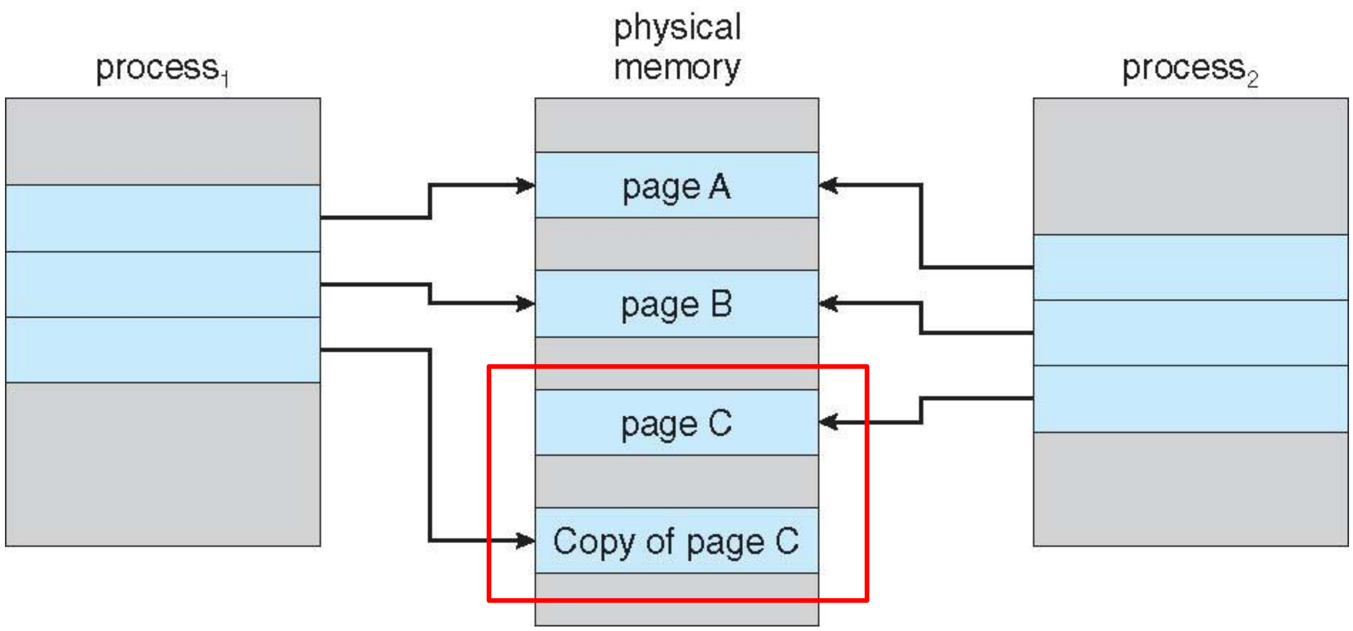
Copy-on-Write

- **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
- Free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
- `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()`
  - Very efficient
  - Sometimes used to implement UNIX command-line shell interfaces

Before Process 1 Modifies Page C



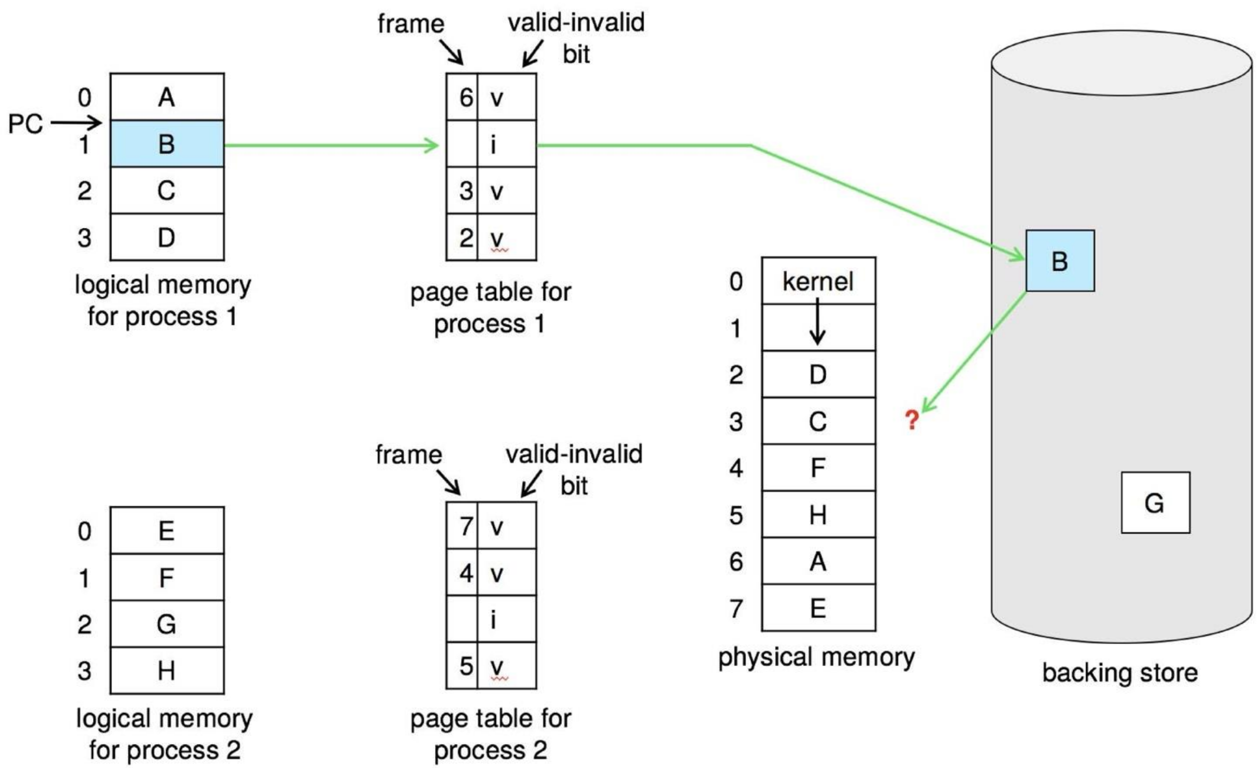
After Process 1 Modifies Page C



Page Replacement

What Happens if There is no Free Frame?

- **Page replacement** – find some page in memory, but not in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm with **minimum # page faults**



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 secondary backing storage (**if necessary**)

3. Bring the new page into the free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now can have 2 page transfers for page fault – increasing EAT

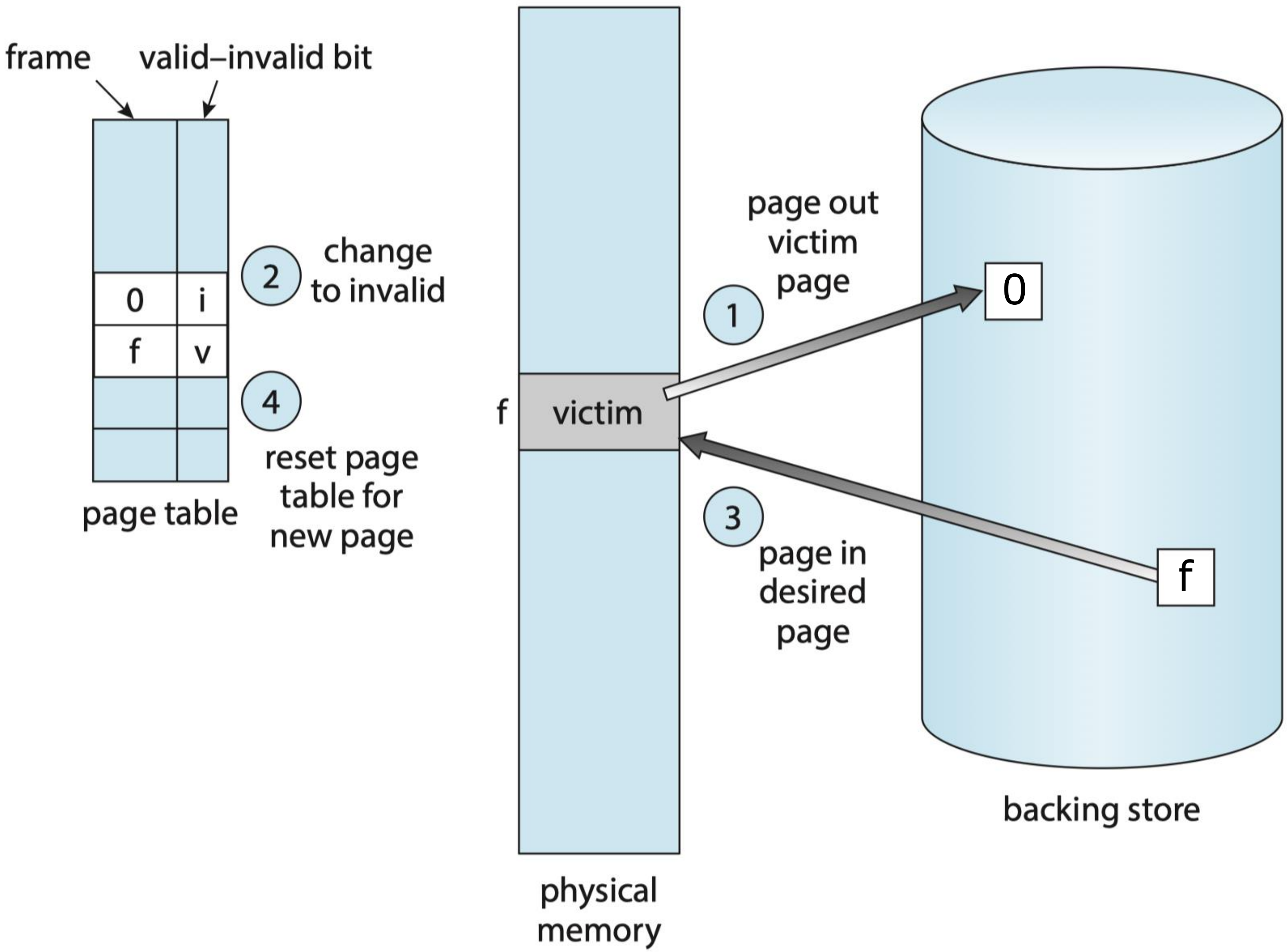
Page Replacement

- 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

Page Replacement

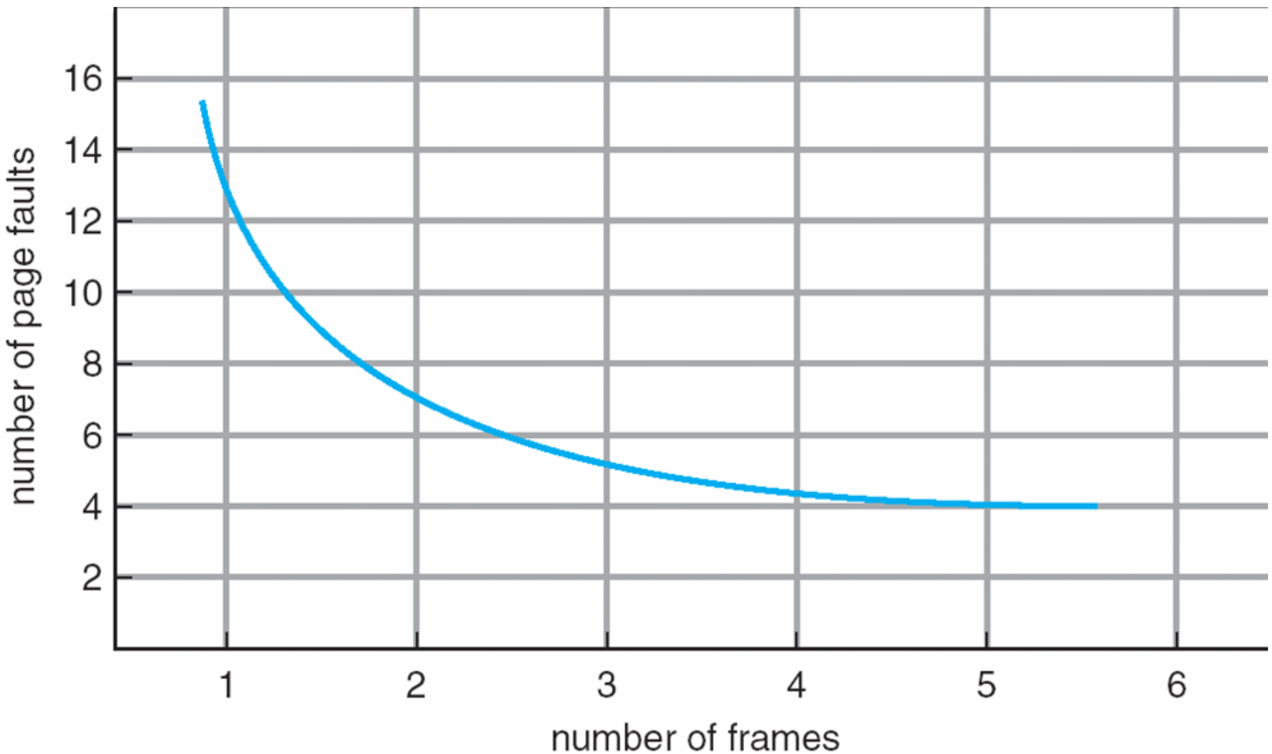


Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **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 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

Graph of Page Faults Versus the Number of Frames



Page-Replacement Algorithms

- First-In-First-Out (FIFO) Page Replacement
- Optimal Page Replacement
- Least Recently Used (LRU) Page Replacement
- LRU Approximation Page Replacement
- Counting Page Replacement

Page Replacement: (1) FIFO

- When a page must be replaced, the oldest page is chosen.
- 3 frames (3 pages can be in memory at a time per process)

- Page faults: 15
- Consider the following reference string:

    0 1 2 3 0 1 2 3 0 1 2 3 ……

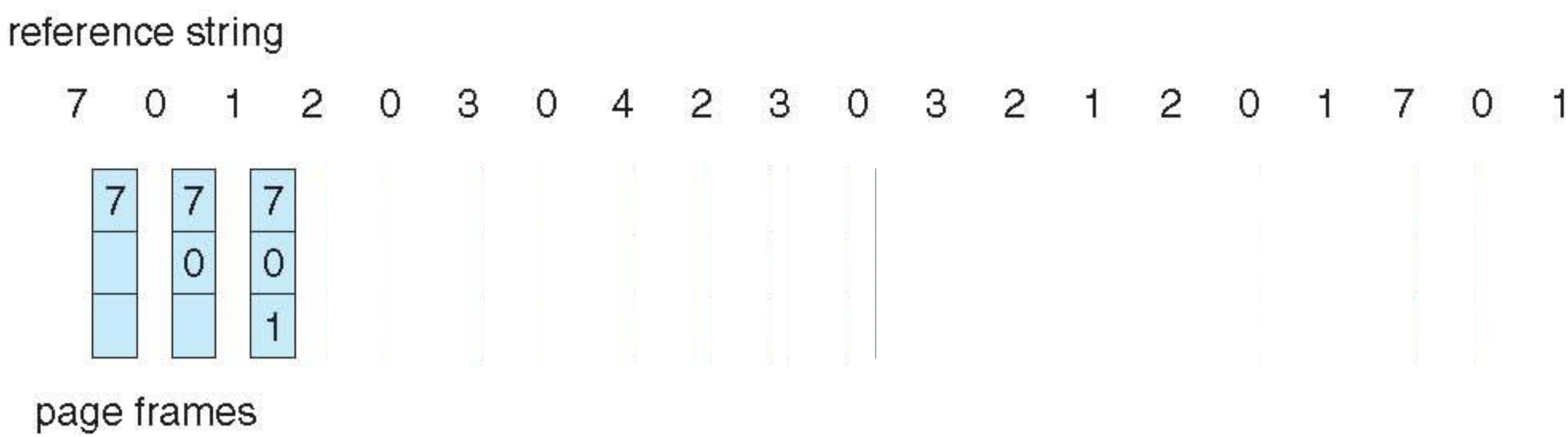
Page Replacement: (2) Optimal

- Replace page that **will not be used for the longest period of time**

- Page faults: 9
- How do you know this?
  - Can’t read the future
- Only an upper bound for measuring how well your algorithm performs

Page Replacement: (3) Least Recently Used (LRU)

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



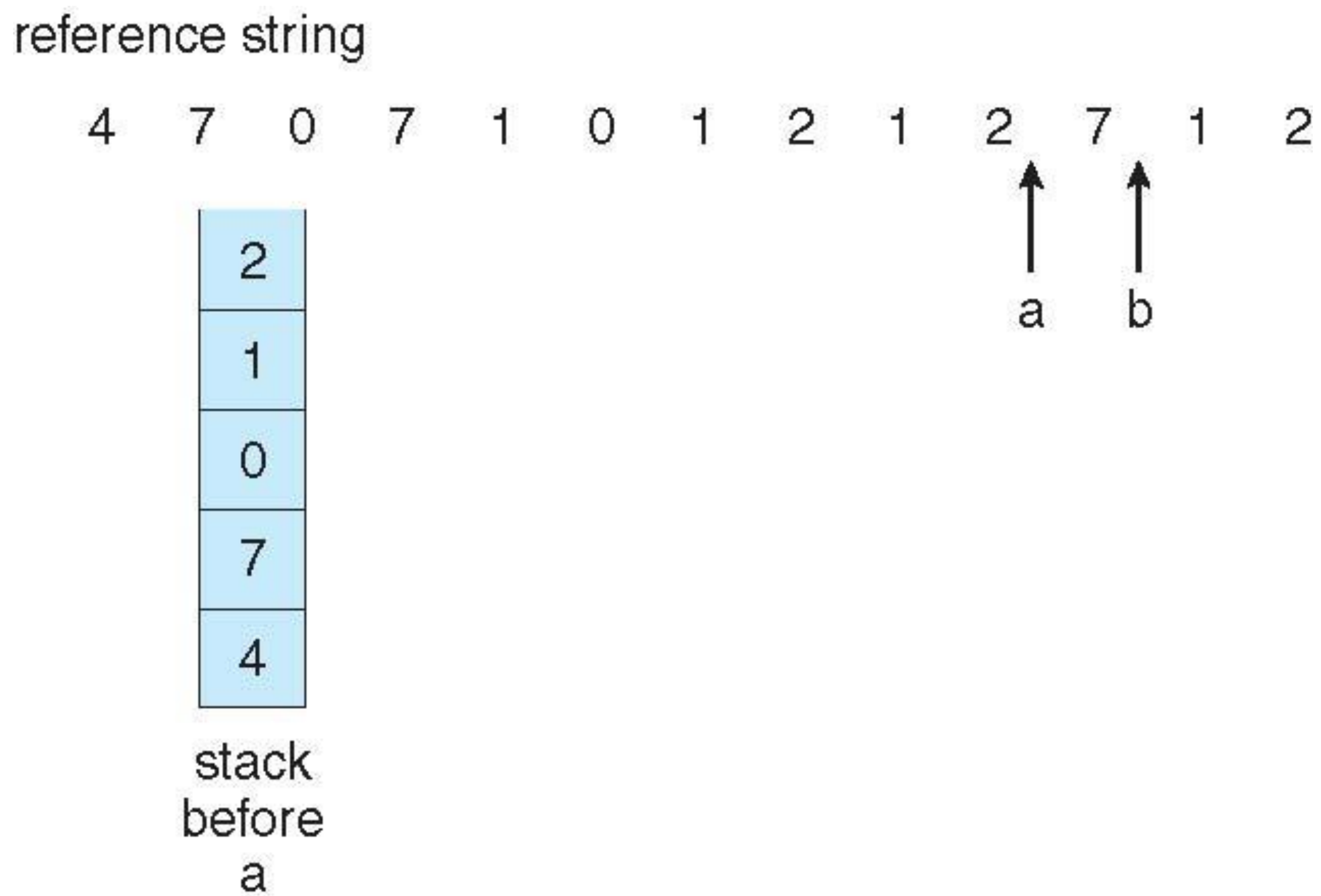
- 12 faults – better than FIFO but worse than OPT
- Generally a good algorithm and is frequently used

Page Replacement: (3) Least Recently Used (LRU)

- LRU implementation with **counter**
  - **O(1) Update**: Every page entry has a counter; every time page is referenced through this entry, **copy the clock** into the counter
  - **O(N) Replacement**: When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed

- LRU implementation with **stack**
  - Keep a stack of page numbers in a double link form
  - **O(N) Update**: Page referenced
    - Move it to the top
    - Each update is more expensive
  - **O(1) Replacement**: No search for replacement

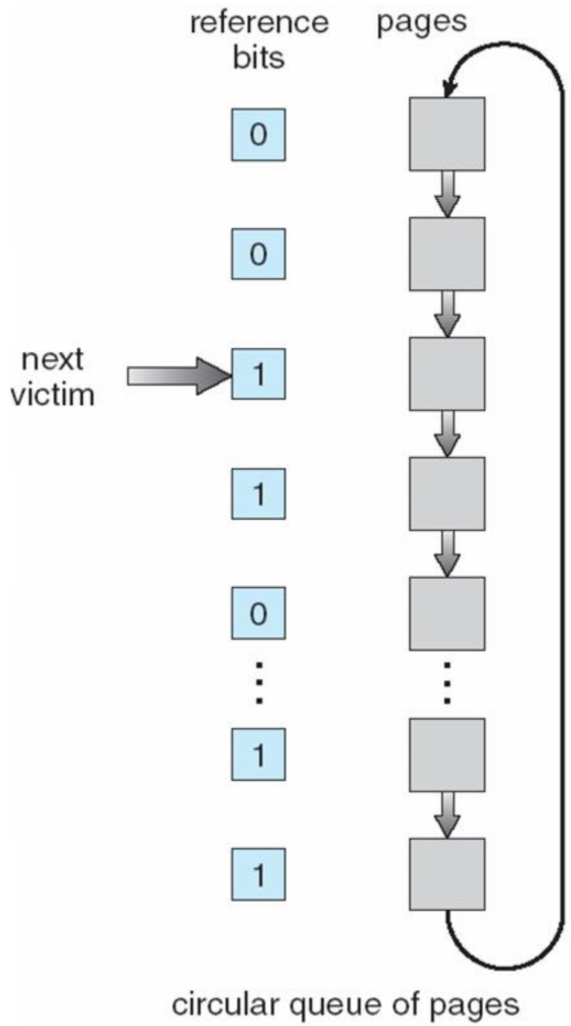
Page Replacement: (3) Least Recently Used (LRU)



Page Replacement: (4) LRU Approximations

- **Reference bit / byte**
  - 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)**
    - We do not specify the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Circular replacement
  - 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
  - Page reference reset the reference bit to 1.

Page Replacement: (4) Second-Chance Algorithm



(a)

Page Replacement: (5) Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least 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
- Not commonly used

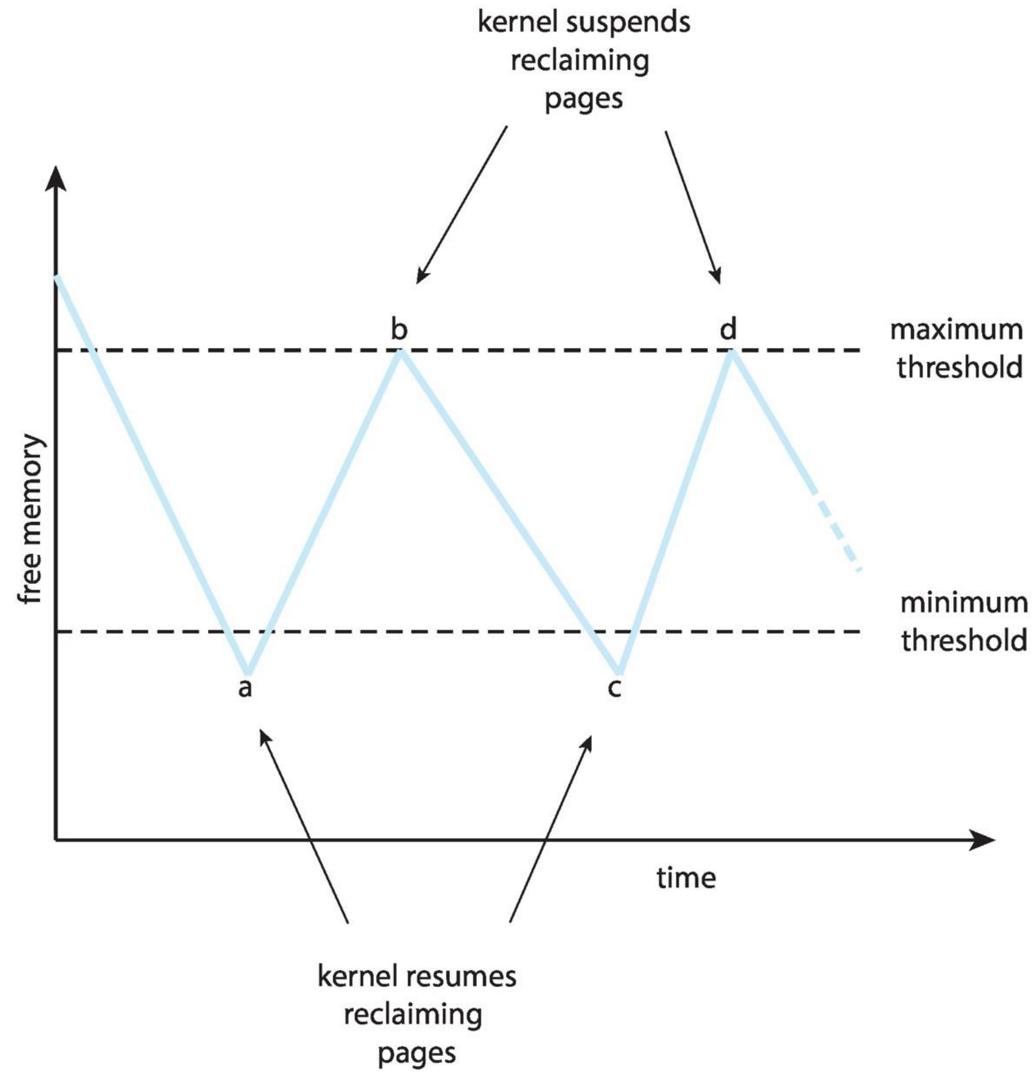
Global vs. Local Replacement

- **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

Global Replacement: Reclaiming Pages

- A strategy to implement a global page-replacement policy
- **All memory requests are satisfied from the free-frame list**
- Page replacement is triggered when the list falls below a certain threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests



Allocation of Frames

Allocation of Frames

- Each process needs **minimum** number of frames
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle *from*
    - 2 pages to handle *to*
- **Maximum** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

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 $p_i = \dfrac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Frame Allocation Example: Windows

- Uses demand paging with **clustering**.
  - Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
  - A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold**, automatic working set trimming (修整)** is performed to restore the amount of free memory
  - Working set trimming removes pages from processes that have pages in excess of their working set minimum

Summary

- Virtual memory abstracts physical memory into an extremely large uniform array of storage
- Benefits of virtual memory: (1) a program can be larger than physical memory, (2) a program does not need to be entirely in memory, (3) processes can share memory, and (4) processes can be created more efficiently.
- Demand paging loads pages only when they are demanded during program execution
- A page fault occurs when a page currently not in memory is accessed
- Copy-on-write allows the child process to share its parent’s address, and only make a copy when one of them modifies a page
- Page replacement algorithms: FIFO, optimal, LRU, LRU-approximations.
- Discussed frame allocation among processes

Homework

- Reading
  - Chapter 10

- Please check Canvas for HW3 release, due on **Mar. 28, 23:59**!