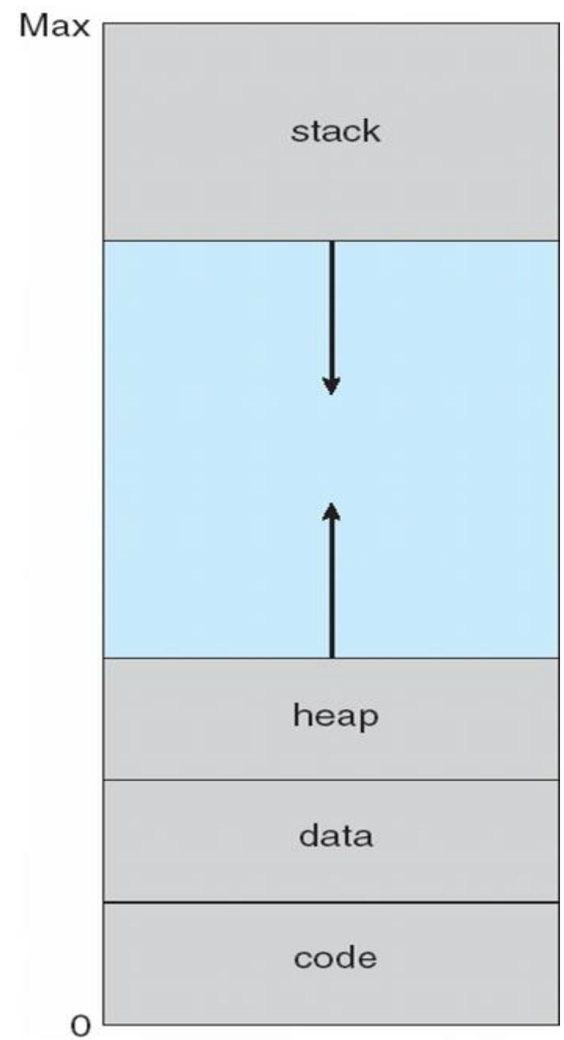
# Virtual Memory I

October 11, 2017

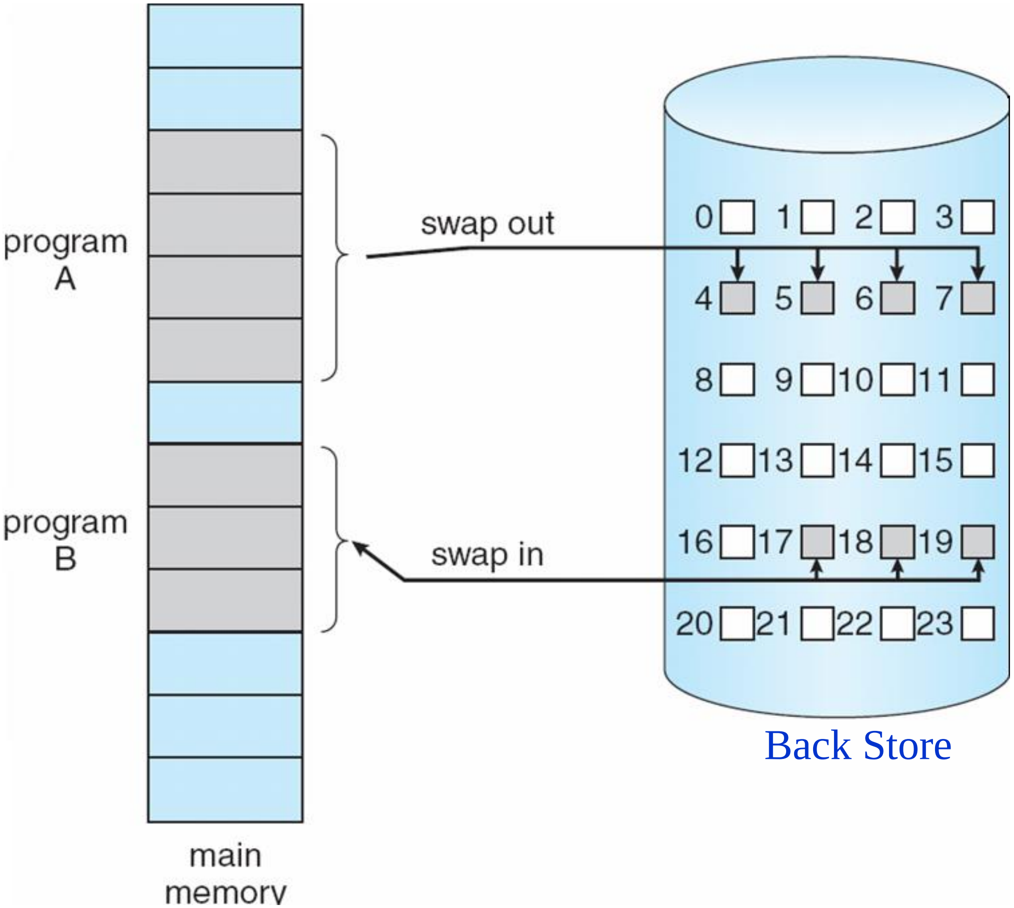
# Virtual Memory: Benefits

- Only part of a process needs to be in memory for execution
  - Part of the image of a running process can be in the back store (disk)
- Logical address space can therefore be much larger than physical address space
  - It is possible to run a 4G program on a computer with 1G memory
- Allows for more efficient process creation
  - Do not need to load in full image



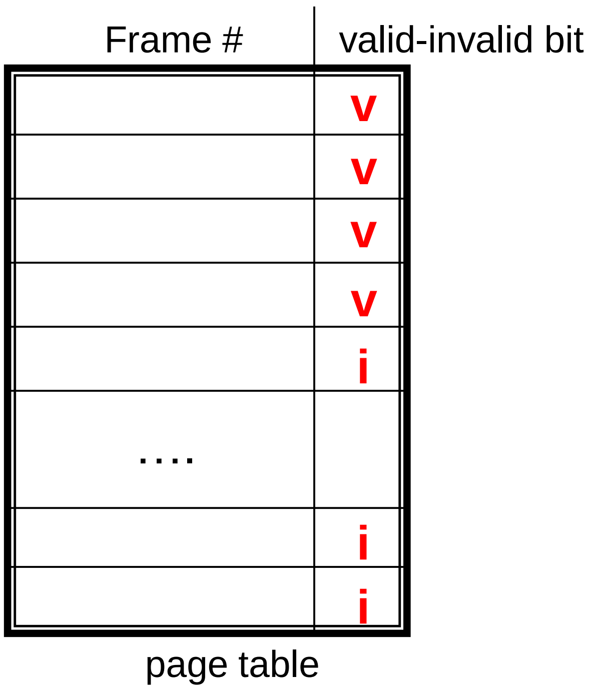
# Implementing Virtual Memory: Demand Paging

- Key idea:
  - Bring a page into memory only when it is needed
- Pager:
  - Swapper that deals with pages is also called **pager**

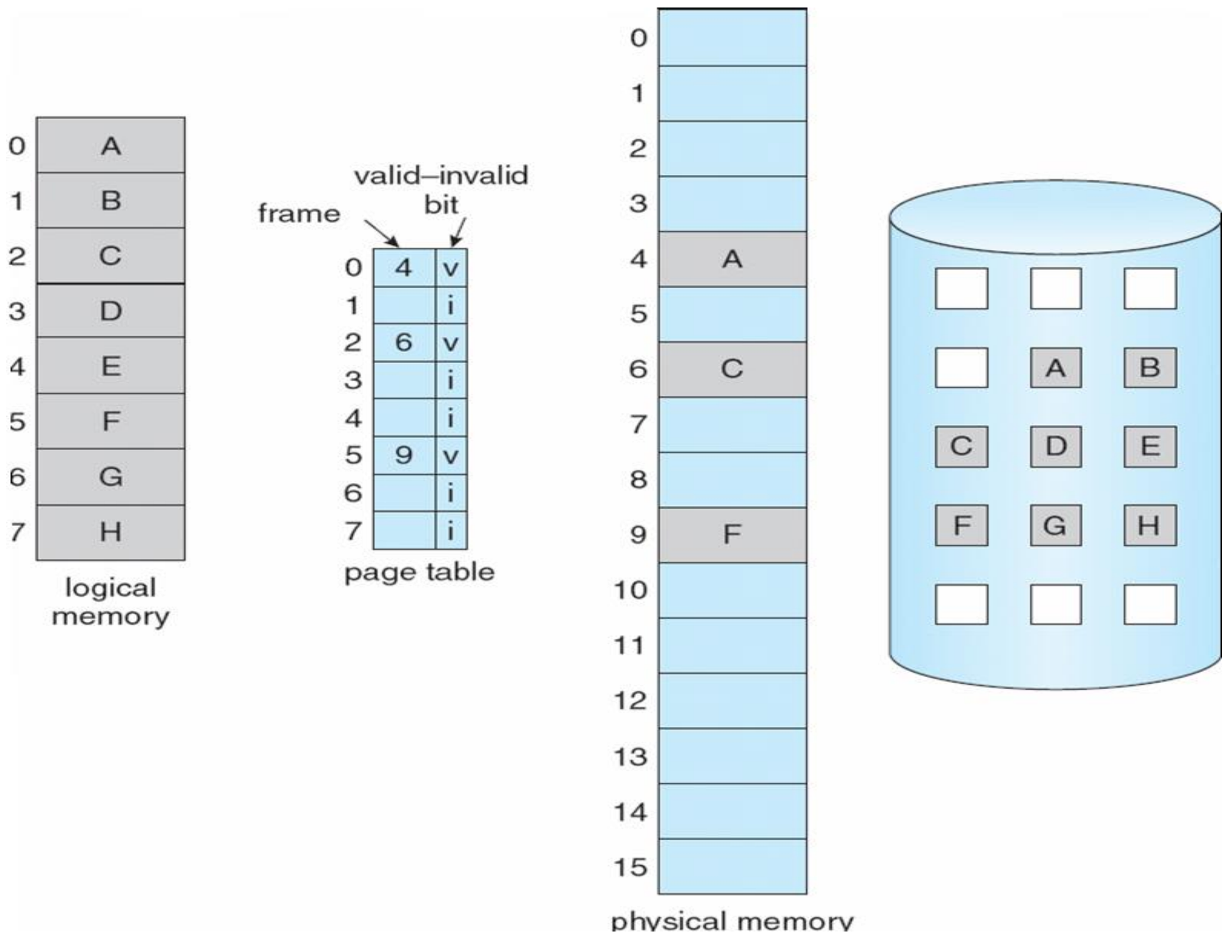


# Page Table: Valid-Invalid Bit

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



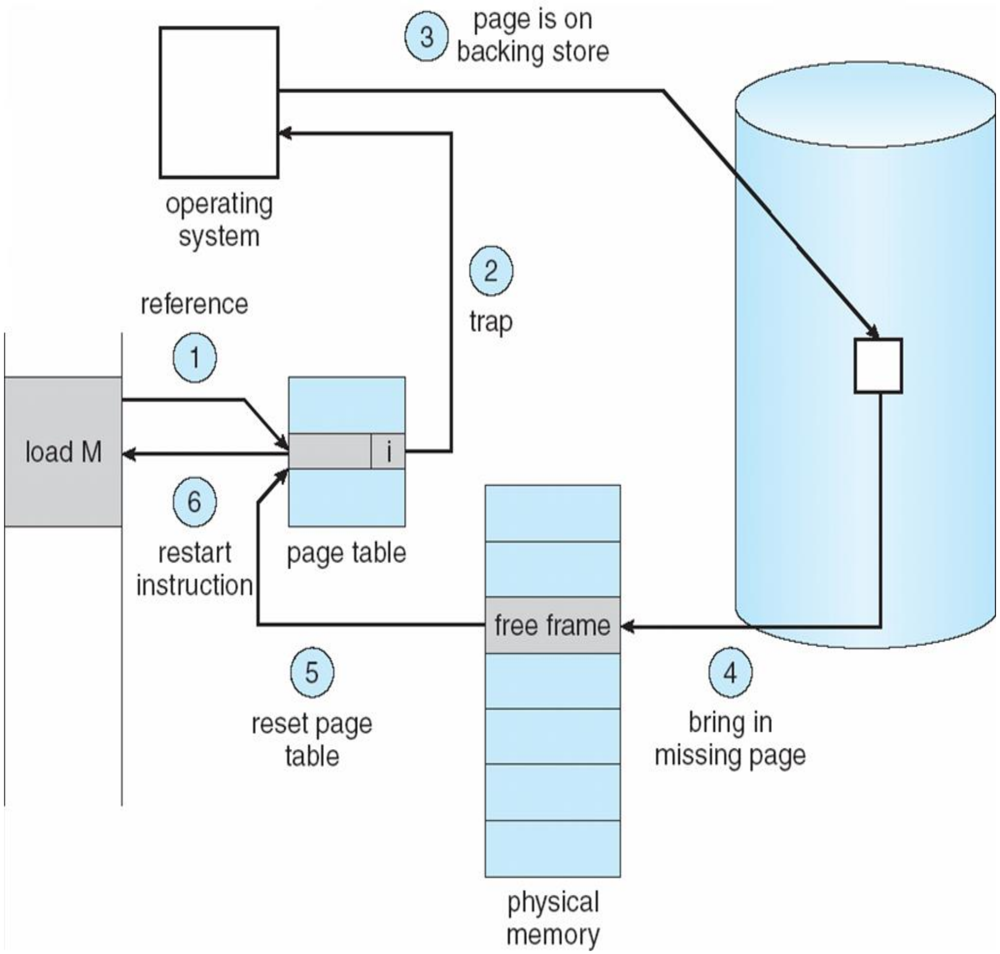
# Page Table When Some Pages Are Not in Main Memory



# Page Fault

- First reference to a page that is not in memory will trap to operating system: **page fault**

# Steps in Handling a Page Fault

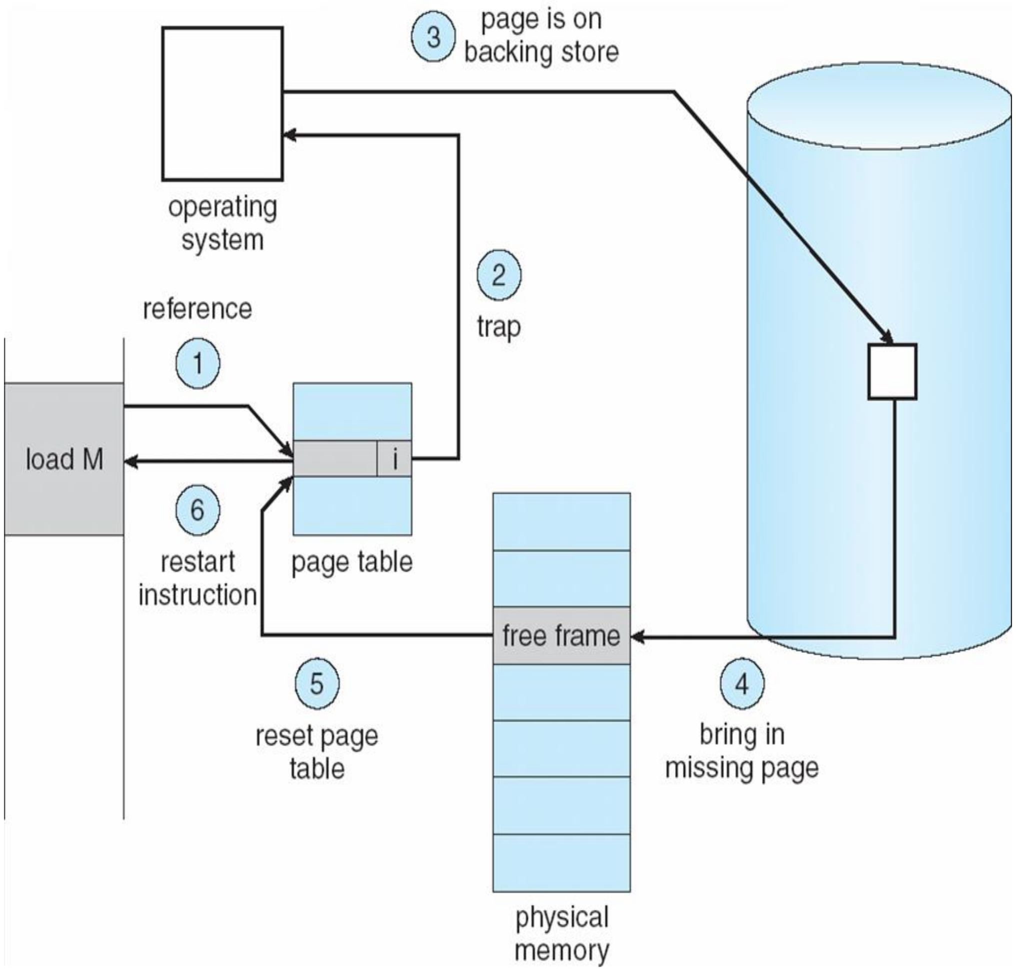



1&2. Operating system determine: If invalid reference → abort; if just not in memory → Proceed

3. Identify the desired page in the back store. Get an empty frame in the physical memory

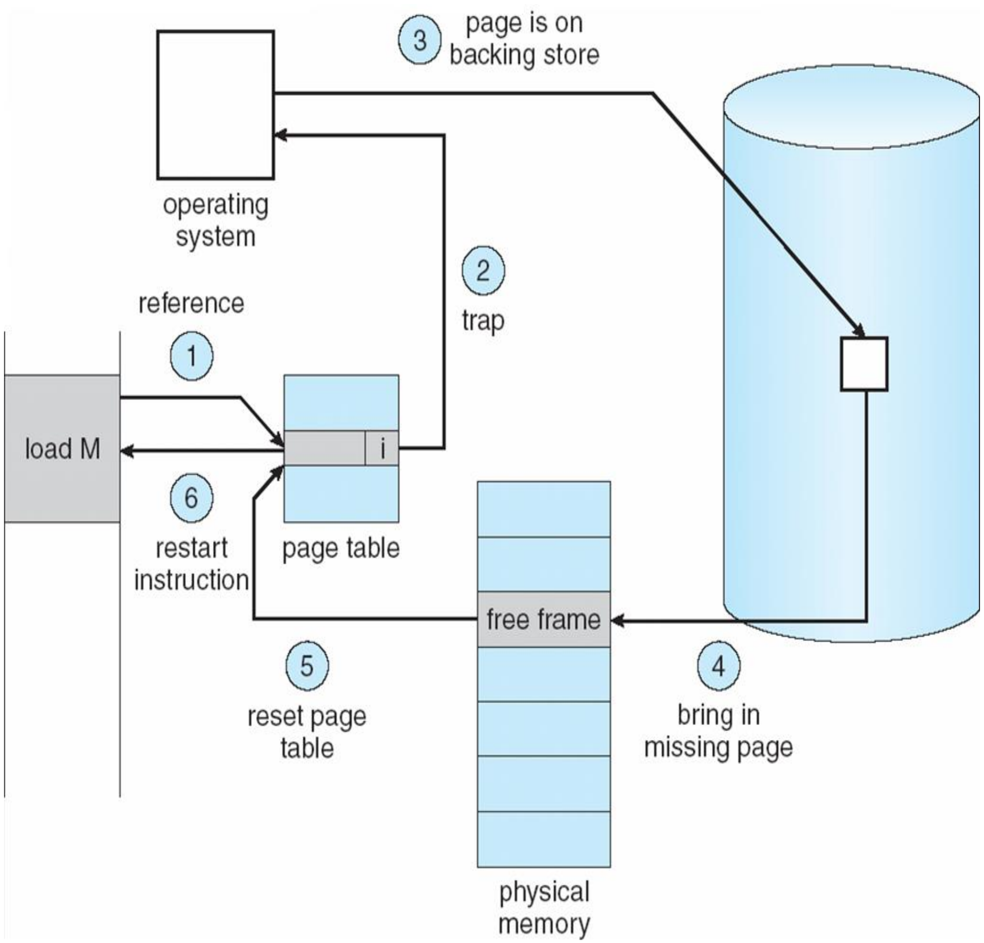
(if no empty, swap out one)

# Steps in Handling a Page Fault




4. Swap page into frame: schedule a disk operation to read the desired page into the newly allocated frame. (While waiting for I/O operation to complete, CPU may be allocated to another process; in this case, after I/O operation completes, this process should wait for the CPU to be allocated to it again)

# Steps in Handling a Page Fault




5. Modify the page table.
   Set validation bit of the newly swapped-in page to **v**
6. Restart the instruction that caused the page fault

# Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = $(1 – p)$ * memory access
+ $p$ * (page fault overhead:
page fault interrupt service
+ swap page in
+ resume interrupted process
+ memory access
)

# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault overhead = 8 milliseconds
- EAT = (1 – p) * 200 + p * (8 milliseconds)

  = (1 – p) * 200 + p * 8,000,000

  = 200 + p * 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!!
- To improve performance:
  - Reduce page-fault rate: predict pages to access; not swap out pages that will be accessed soon; …
  - Reduce page-fault processing time: faster I/O operations

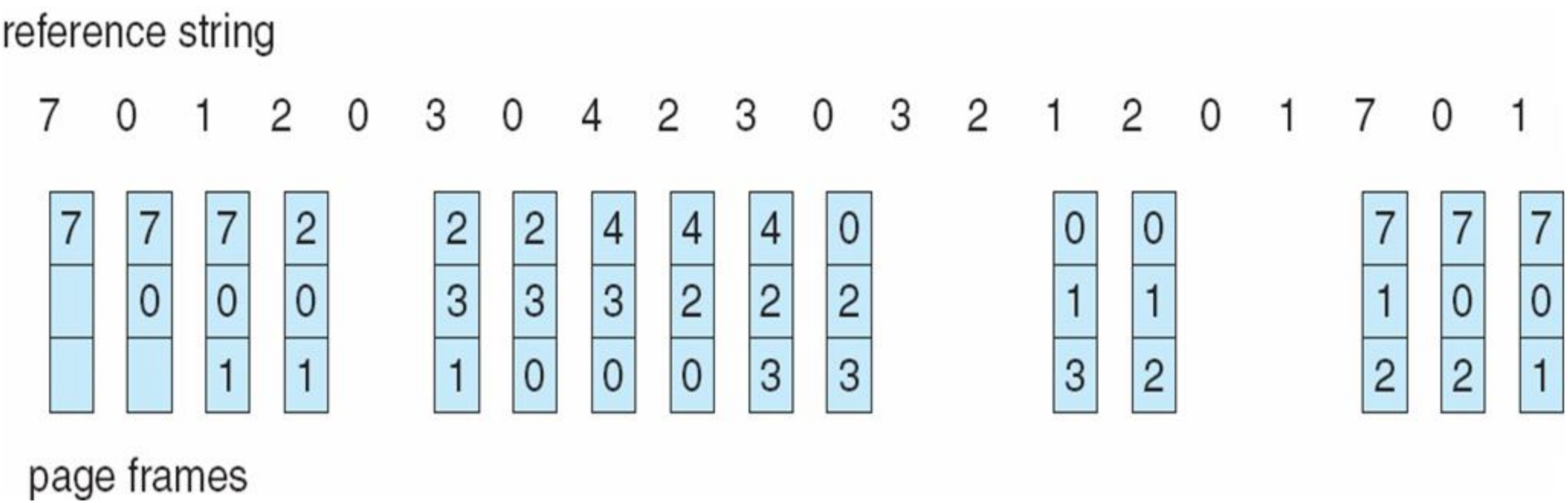
# Page Fault Handling: Basic Framework

1. Find the desired page on disk (backing store)
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
3. Bring the desired page into the free frame; update the page tables
4. Restart the process

# Page Replacement

- Requirement: an ideal page replacement algorithm should result in minimum number of page faults
- When a read-only page is to be swapped (replaced), do not need to write it back.

# FIFO Page Replacement



# First-In-First-Out (FIFO) Algorithm

- 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 |
| 3 | 3 | 2 | 4 |

9 page faults

- 4 frames

| | | | |
|---|---|---|---|
| 1 | 1 | 5 | 4 |
| 2 | 2 | 1 | 5 |
| 3 | 3 | 2 | |
| 4 | 4 | 3 | |

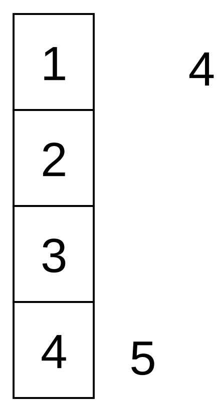
10 page faults

- Belady’s Anomaly: more frames $\Rightarrow$ more page faults

# Optimal (Ideal) Algorithm

- Replace page that will not be used for the longest period of time
- 4 frames example

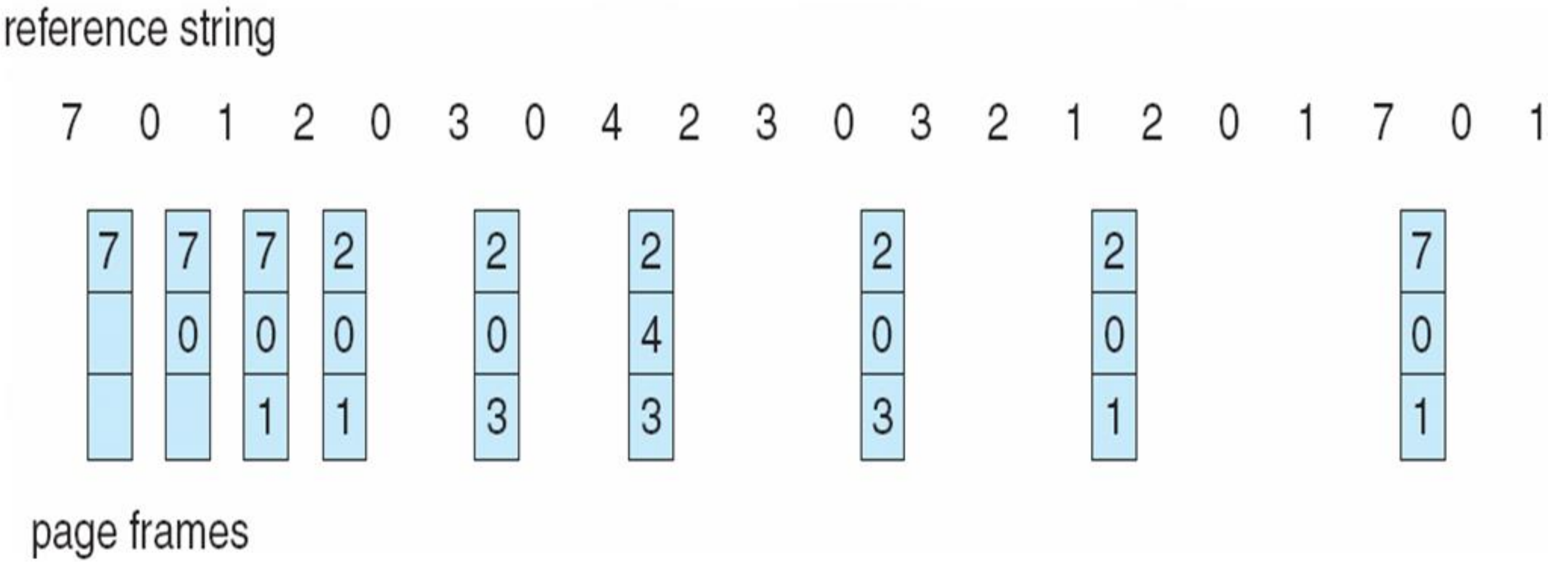
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5




6 page faults

- How do you know this? No.
- Why do we care this algorithm? Used for measuring how well your algorithm performs

# Optimal Page Replacement



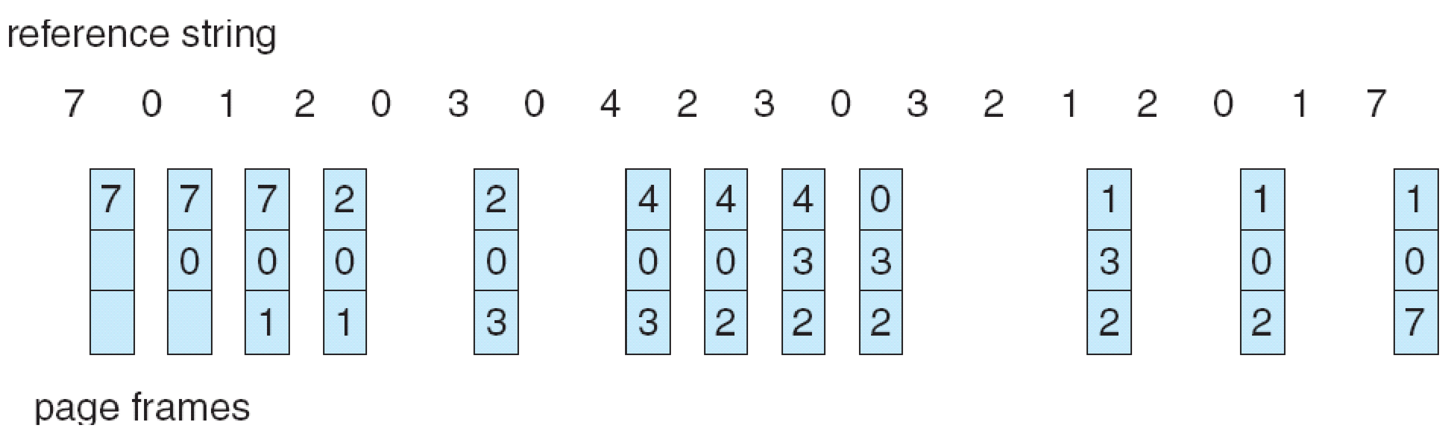
# Least Recently Used (LRU) Algorithm

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

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 5 |
| 2 | 2 | 2 | 2 | 2 |
| 3 | 5 | 5 | 4 | 4 |
| 4 | 4 | 3 | 3 | 3 |

# LRU Page Replacement

- Counter implementation
  - Every frame has a counter; every time the page in a frame is referenced, copy the clock into the counter of the frame
  - When a page needs to be replaced, the page in the frame with the oldest counter value is to be replaced



# LRU Algorithm: Stack Implementation

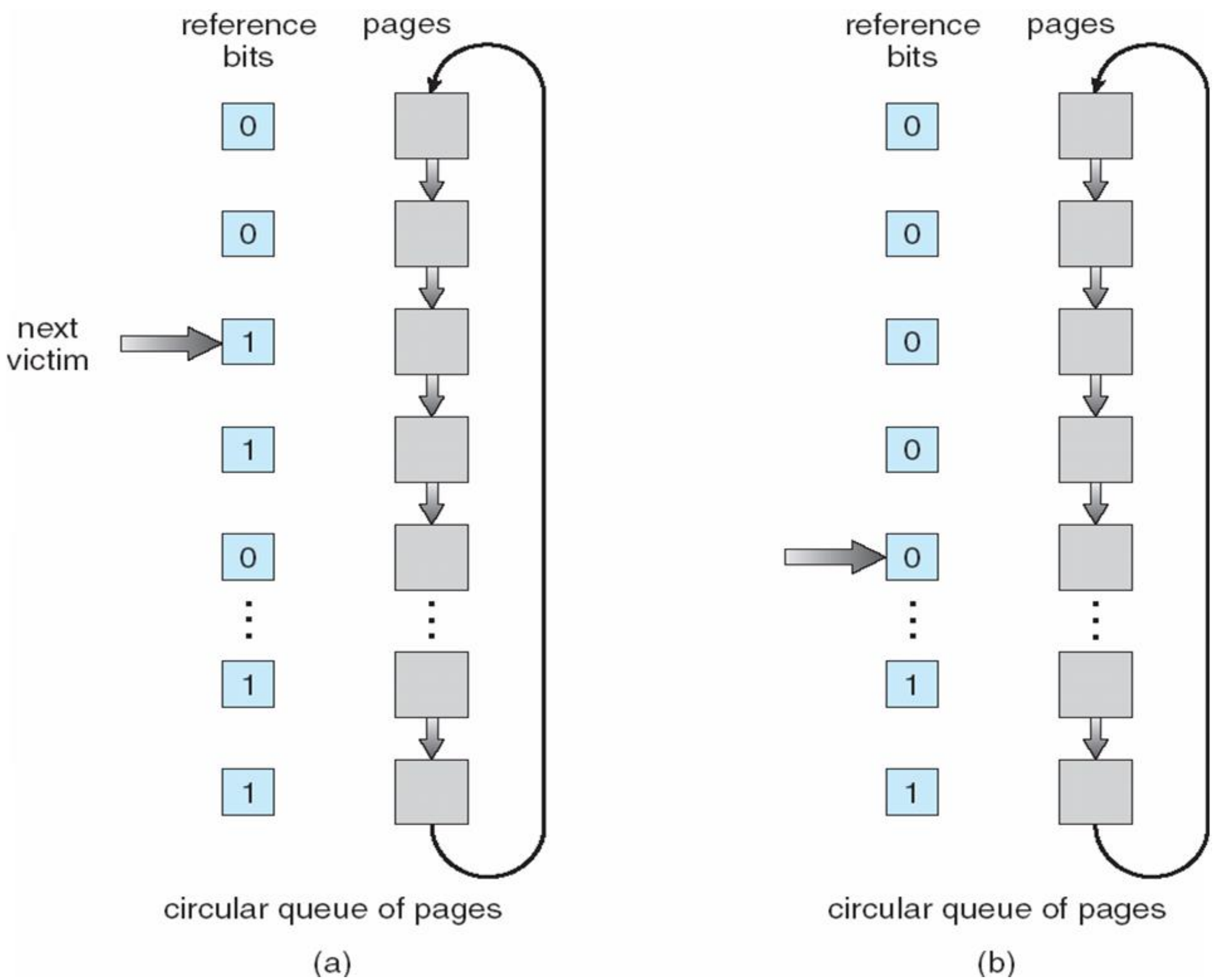
- Keep a stack of page numbers
- Page referenced is in the stack: move it to the top
- Page referenced is not in the stack:
  - If there is free frame → push the page into the stack
  - If there is no free frame → swap out the page on the stack bottom; push the new page into the stack
- Example
  - Reference string 70120304230321201701
  - #of frames=3

# Second-Chance: An Approximate LRU

- A reference bit associated with each page in physical memory
- All pages in physical memory form a circular queue; a pointer pointing to the head element
- When a page is referenced, its reference bit is set to 1.

- When a page should be replaced:
  - Step 1. Move the pointer by one step
  - Step 2. Check the page pointed by the pointer:
    - If the associated bit is 0 → replace the page
    - If the associated bit is 1 → change the bit to 0 and go to step 1.

# Second-Chance (clock) Page-Replacement Algorithm

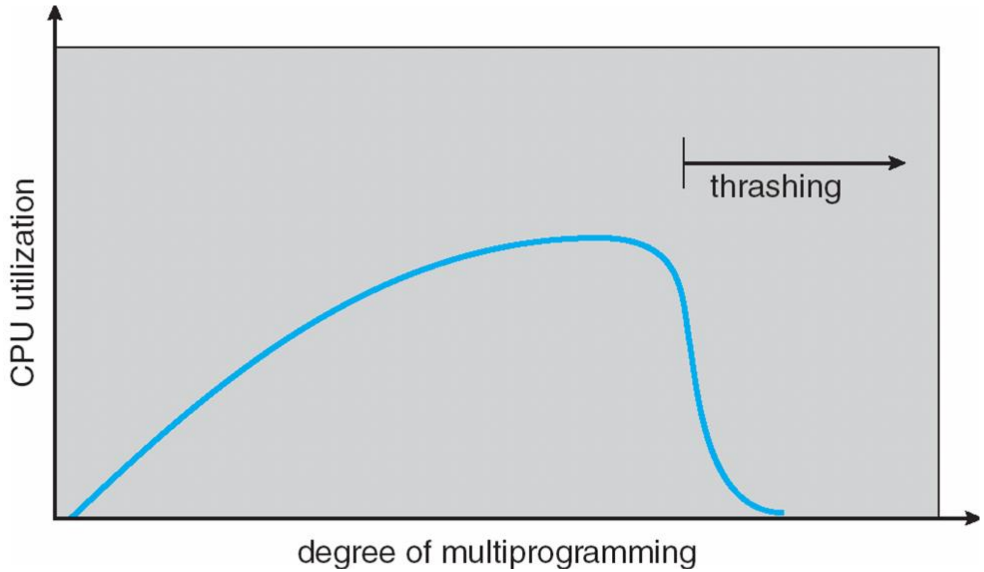


# Minimal Number of Pages Needed

- Each process needs *minimum* number of pages: determined by computer architecture
- Example:  IBM 370  needs 6 pages to handle MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*

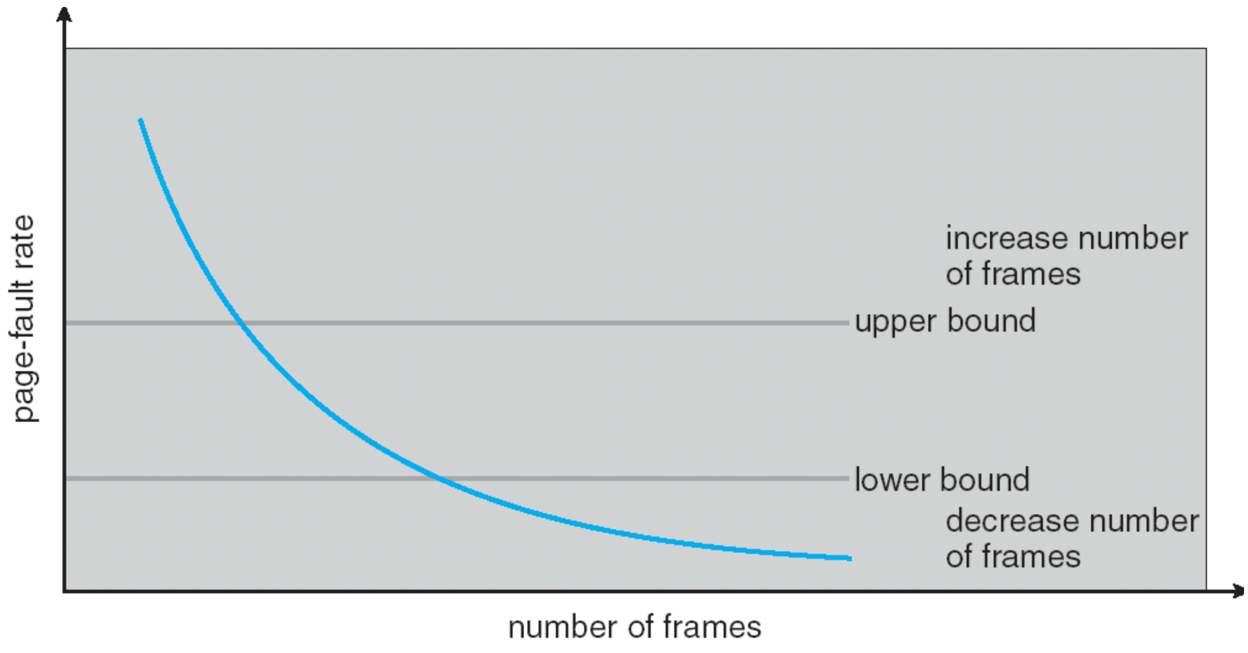
# How many pages are "enough"? Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization: handling page-fault; frequent proc scheduling
  - (because CPU is not fully used) OS thinks that it needs to increase the degree of multiprogramming: another process added to the system
- Thrashing ≡ a process is busy swapping pages in and out



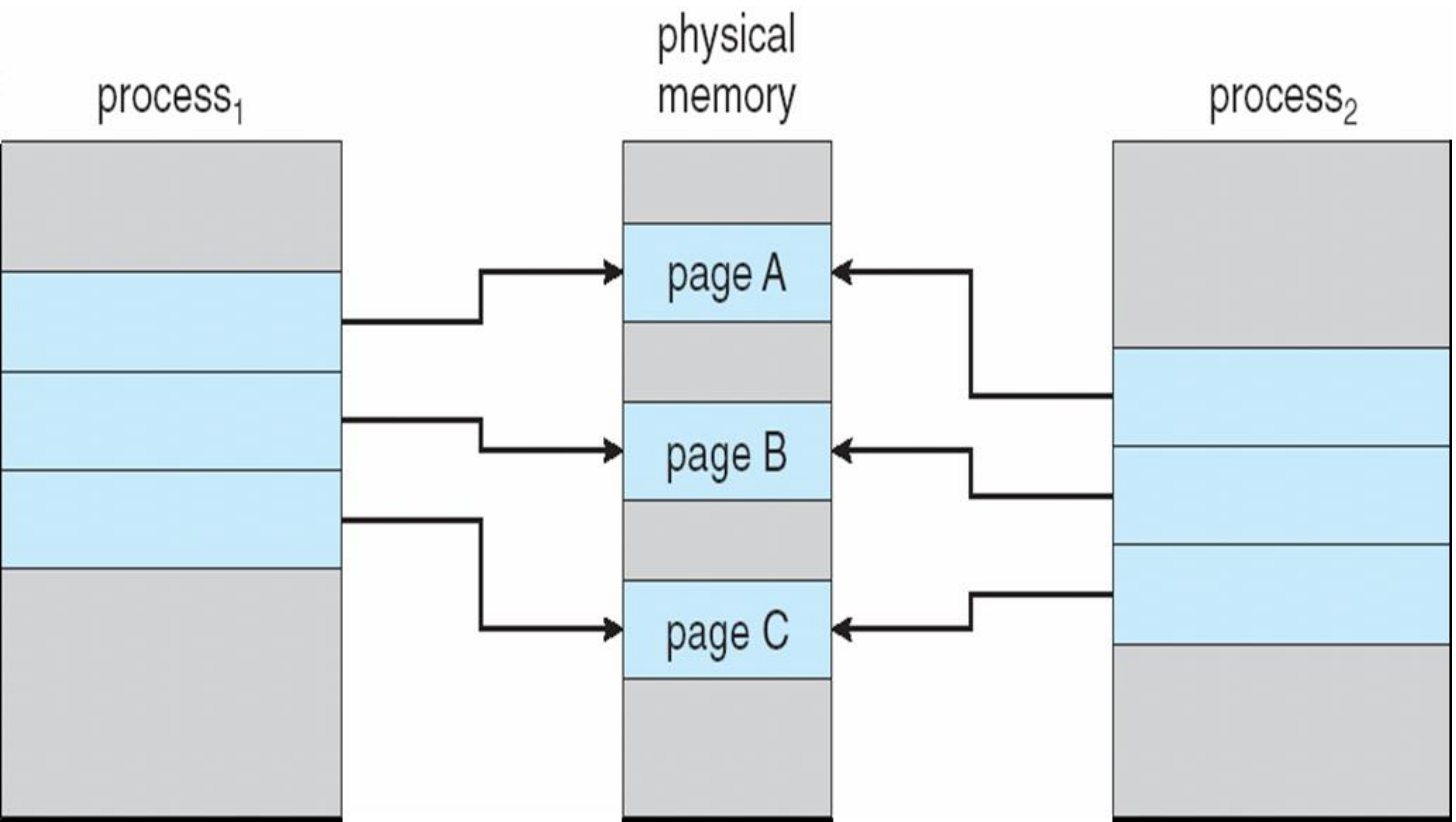
# Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate for a system
- Keep track of the actual page-fault rate for each process
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



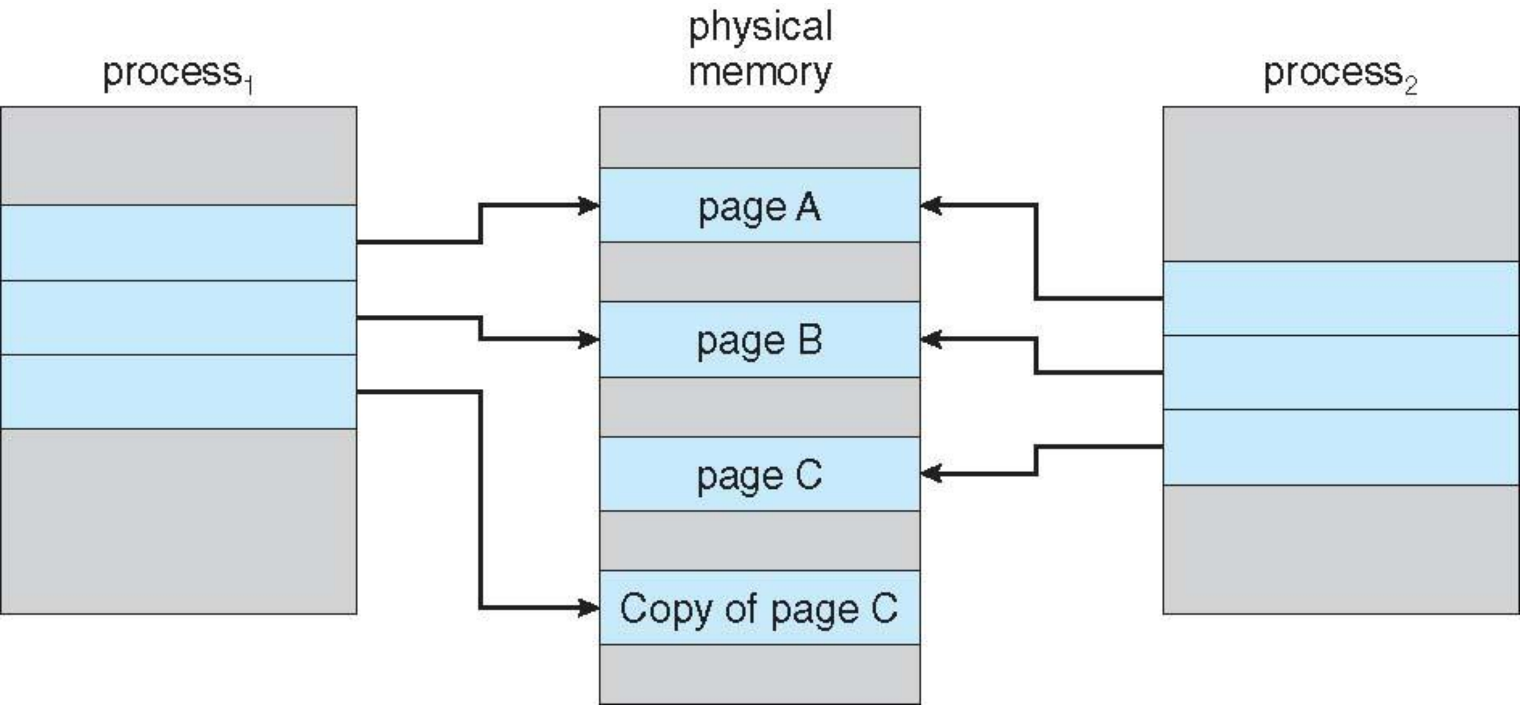
# Copy-on-Write: Speed Up Process Creation

- When a new process is created, it copies the image of its parent.
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory



# Copy-on-Write: Speed Up Process Creation

- If either process modifies a shared page, only then is the page copied



# Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls.

# Memory Mapped Files