# Lecture 16

Edited from slides for Operating System Concepts by Silberschatz, Galvin, Gagne

# Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments.
- A segment is a logical unit such as:

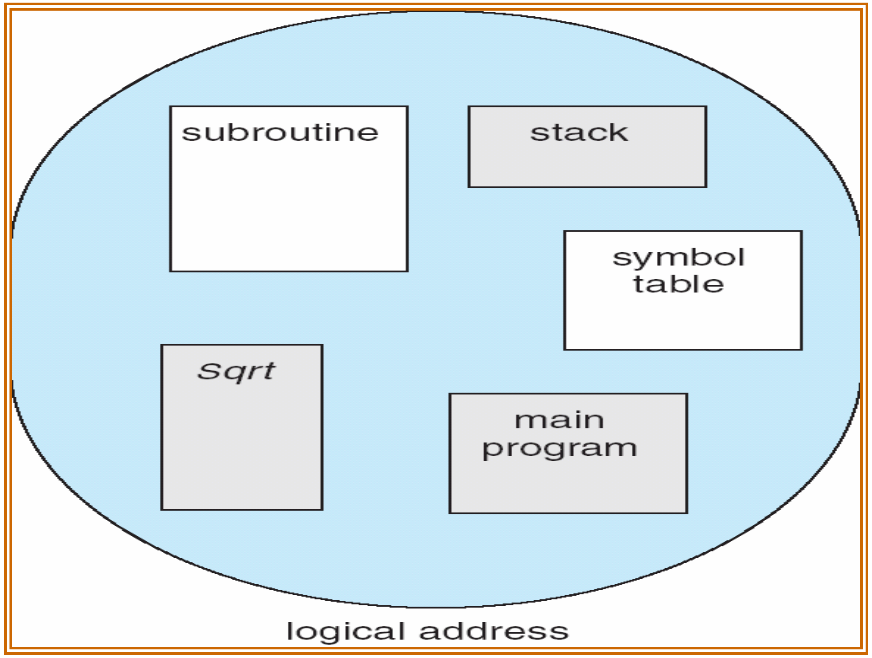
  main program, procedure,

  function, method,

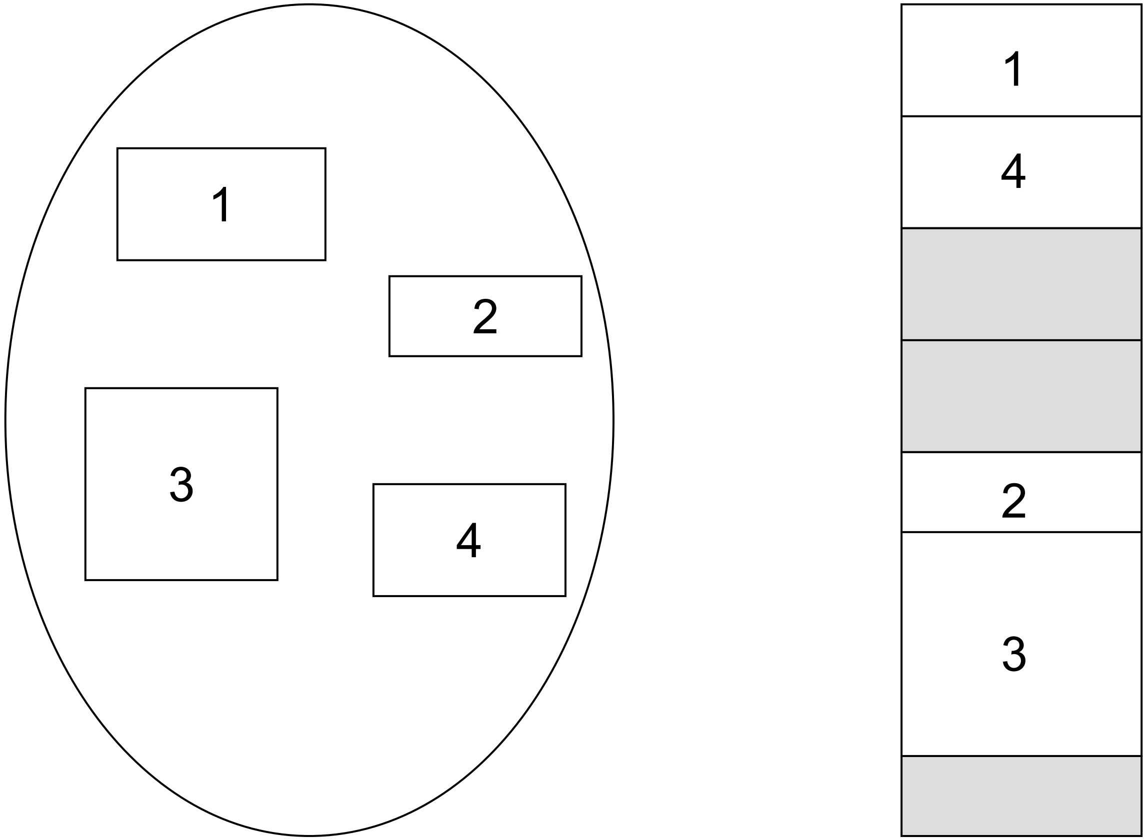
  object, local variables, global variables,

  common block, stack,

  symbol table, arrays



# Logical View of Segmentation

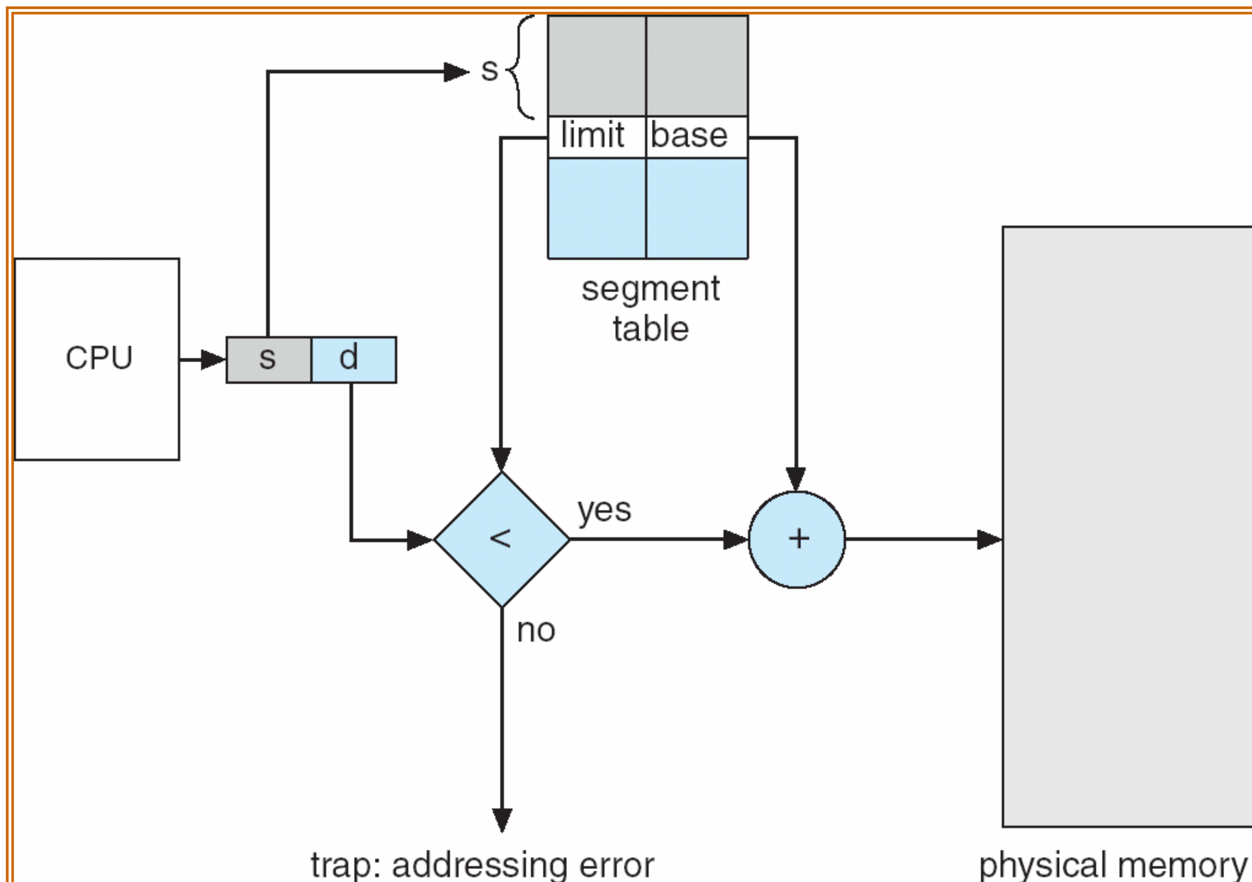



user space

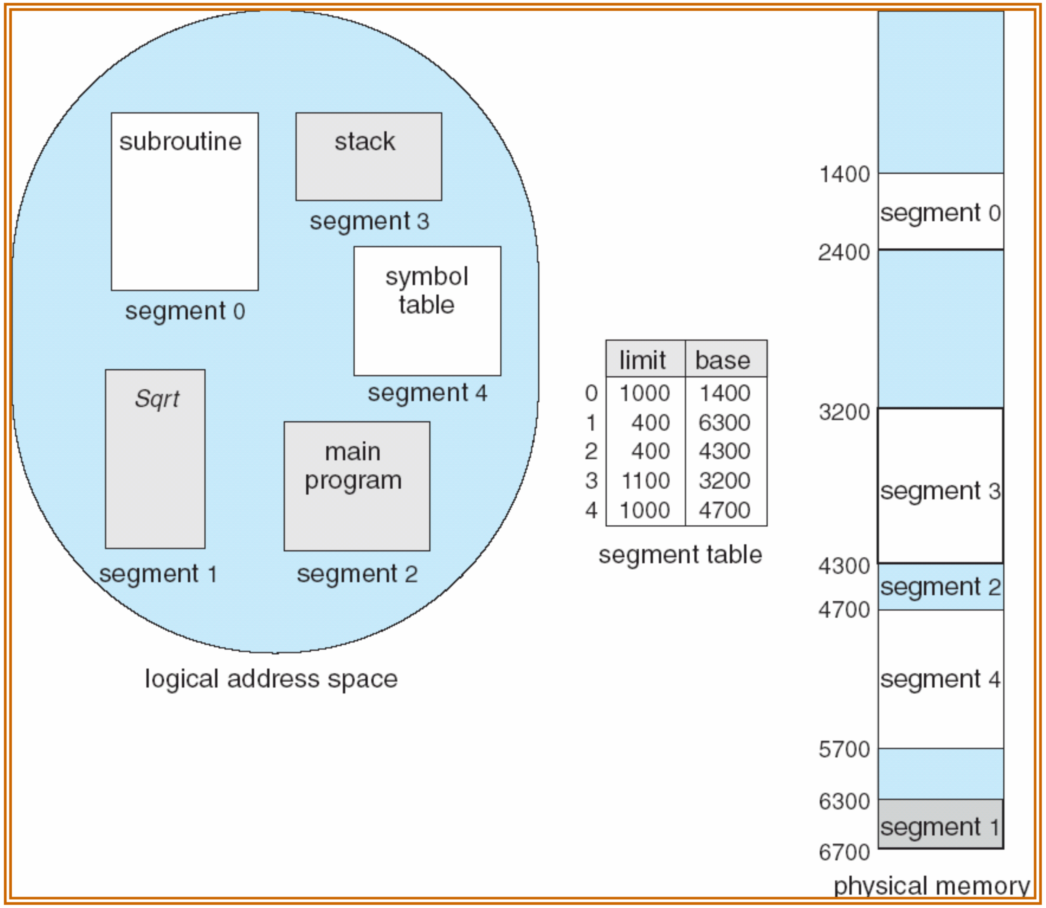
physical memory space

# Segmentation Hardware

- Logical address: <segment-number, offset>,
- **Segment table** –
  - **base** –starting physical address
  - **limit** –length of the segment



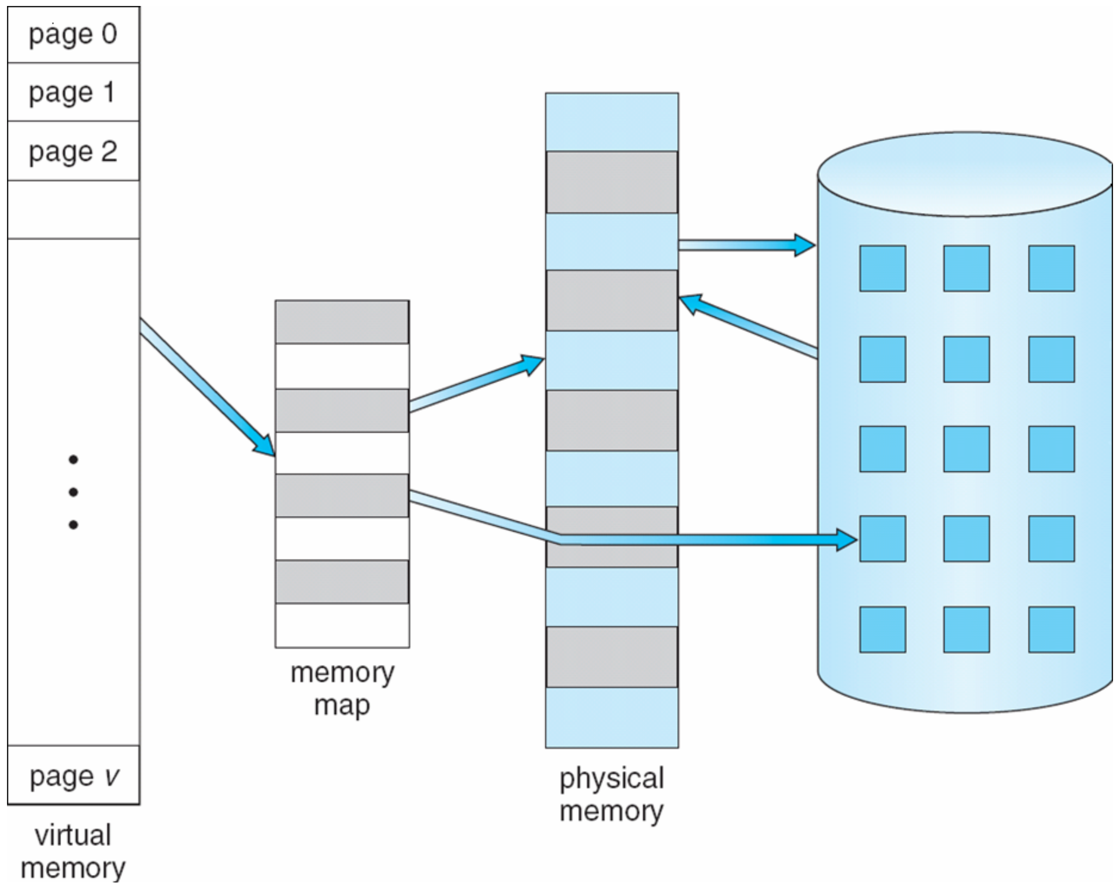
# Example of Segmentation



# Virtual Memory

- **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
  - Allows for more efficient process creation
- implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory That is Larger Than Physical Memory



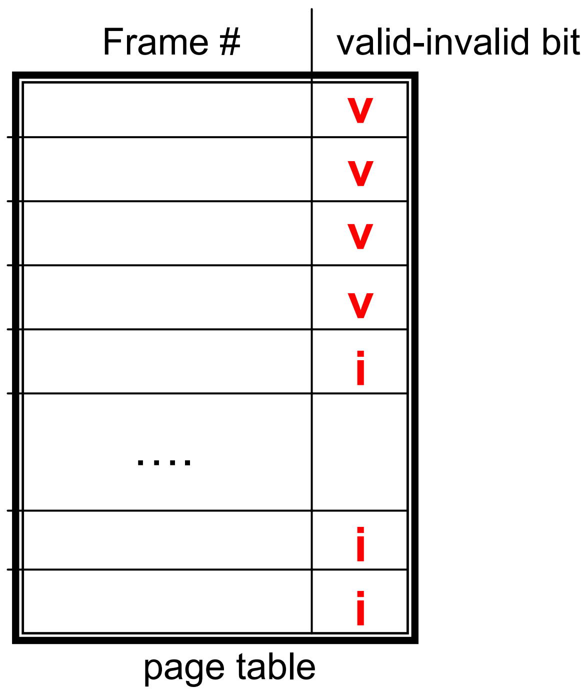
# Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - 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** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

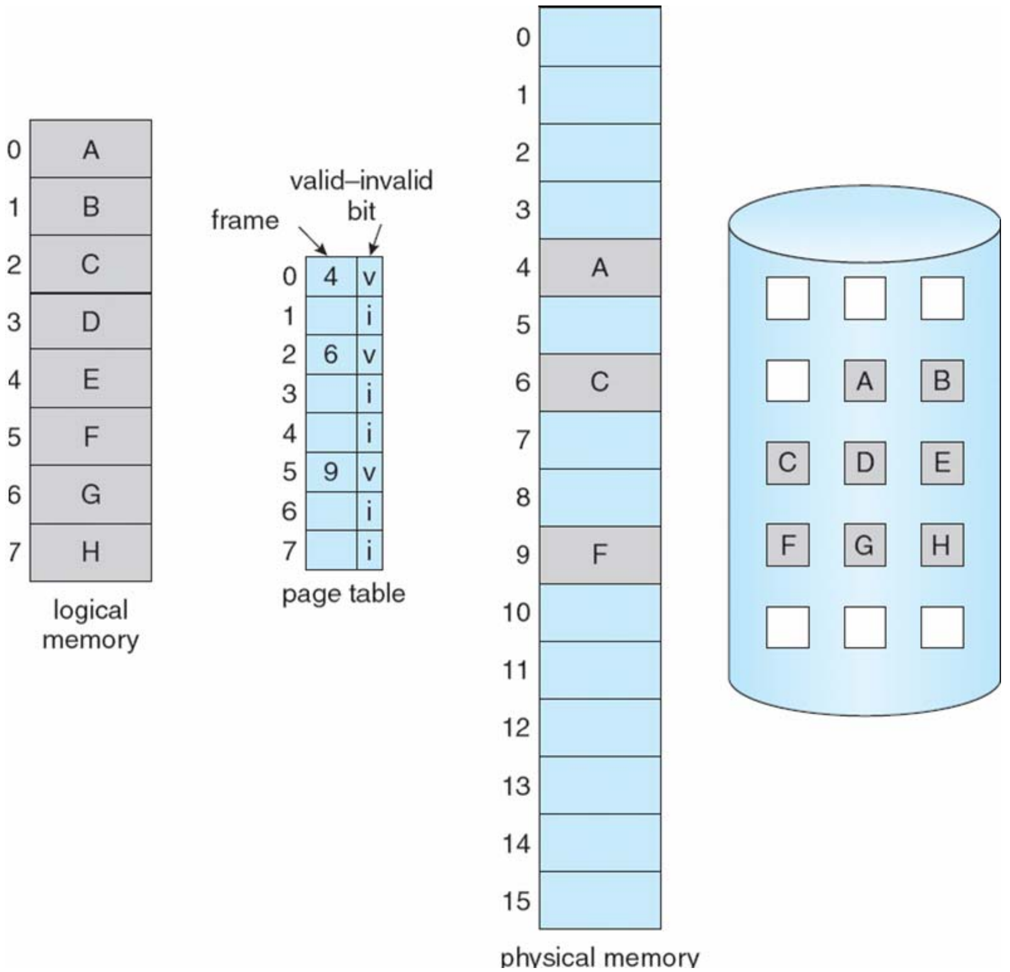
# Valid-Invalid Bit

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

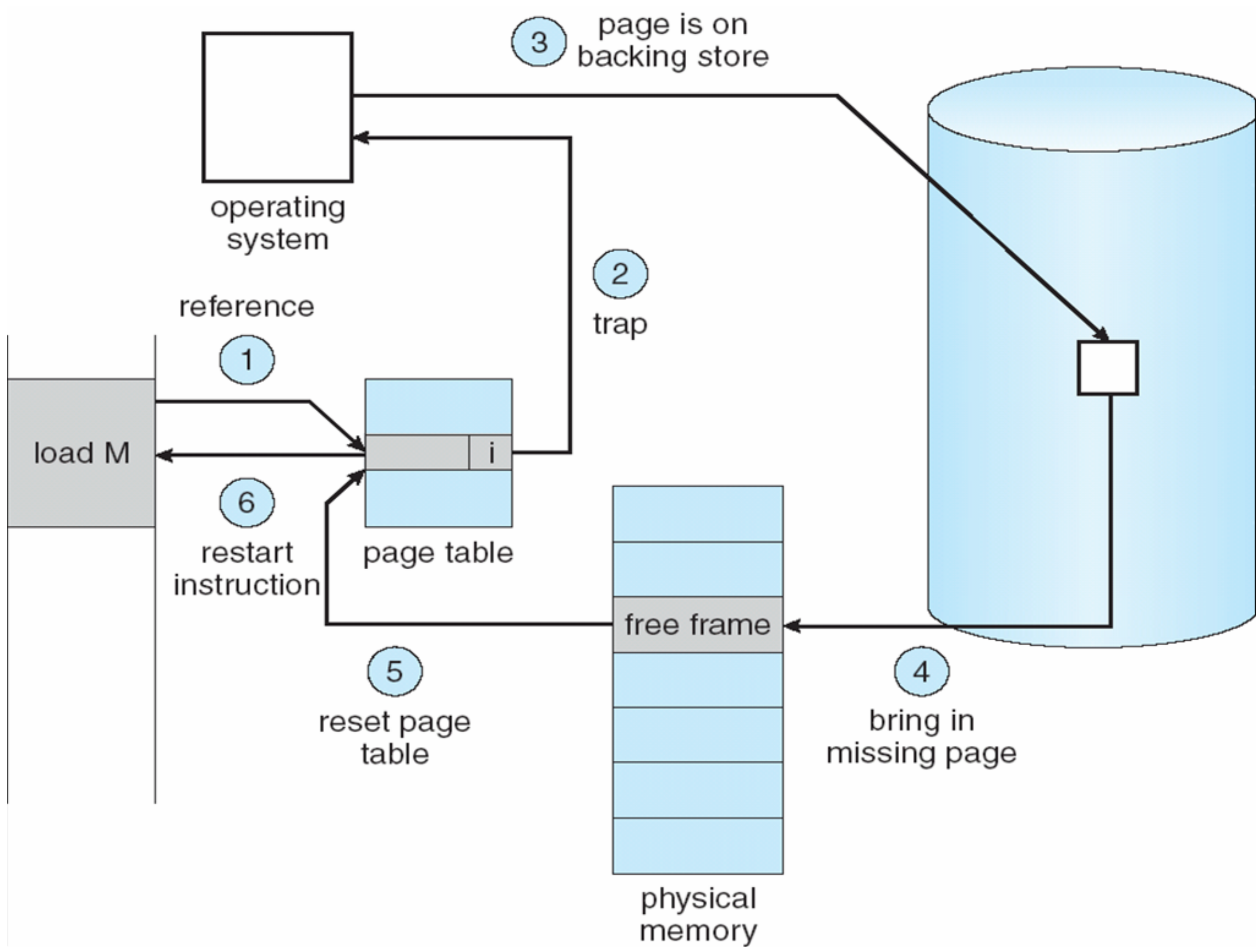



- During address translation, if valid–invalid bit in page table entry is **I** $\Rightarrow$ page fault

# Page Table When Some Pages Are Not in Main Memory



# Steps in Handling a 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 = $(1 – p)$ x memory access
+ $p$ (page fault overhead
+ swap page out
+ swap page in
+ restart overhead
)

# 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.

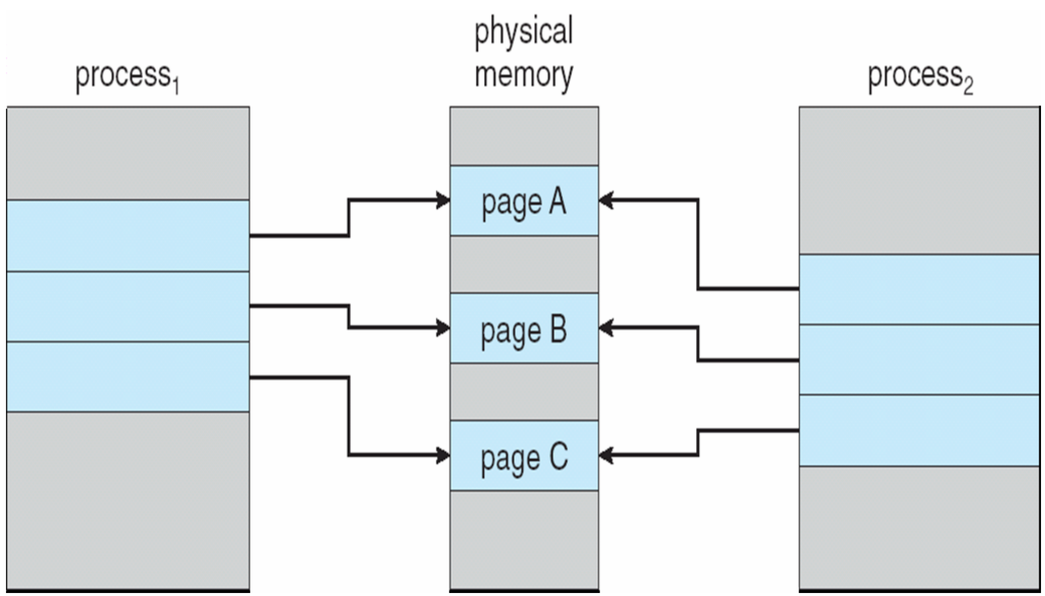
  This is a slowdown by a factor of 40!!

# Copy-on-Write: VM Advantage

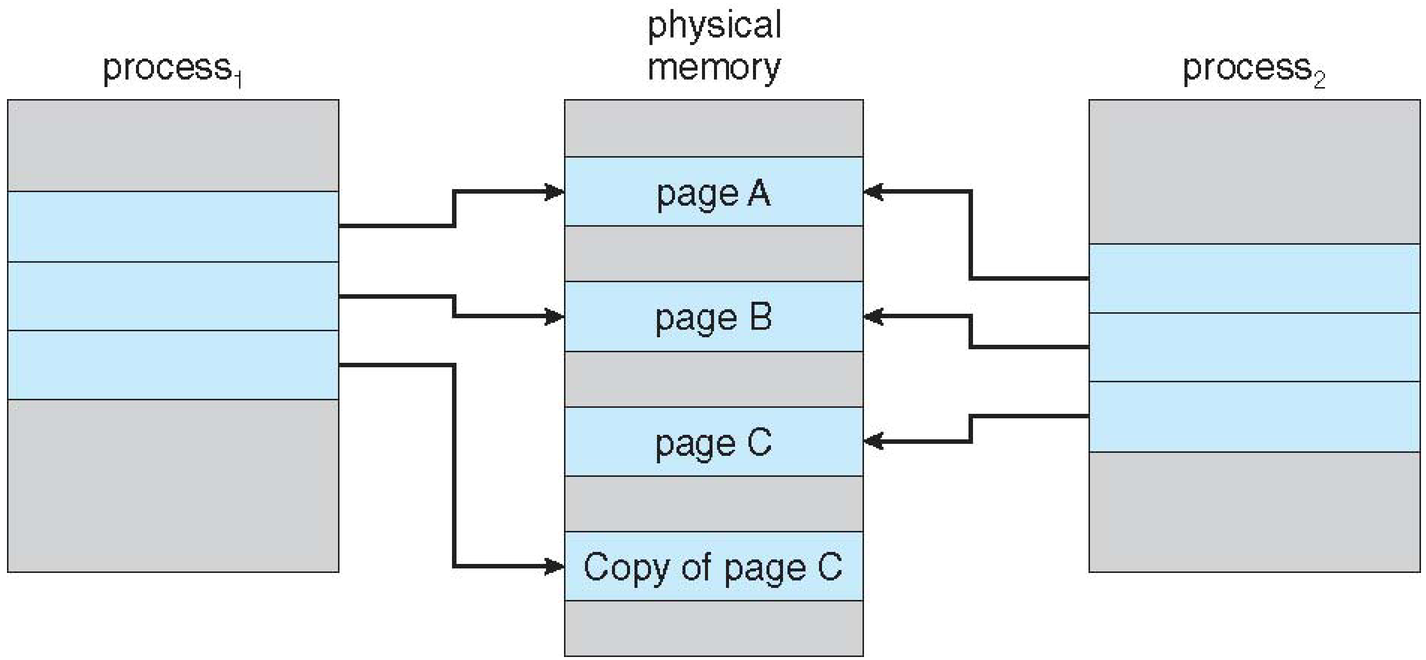
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  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



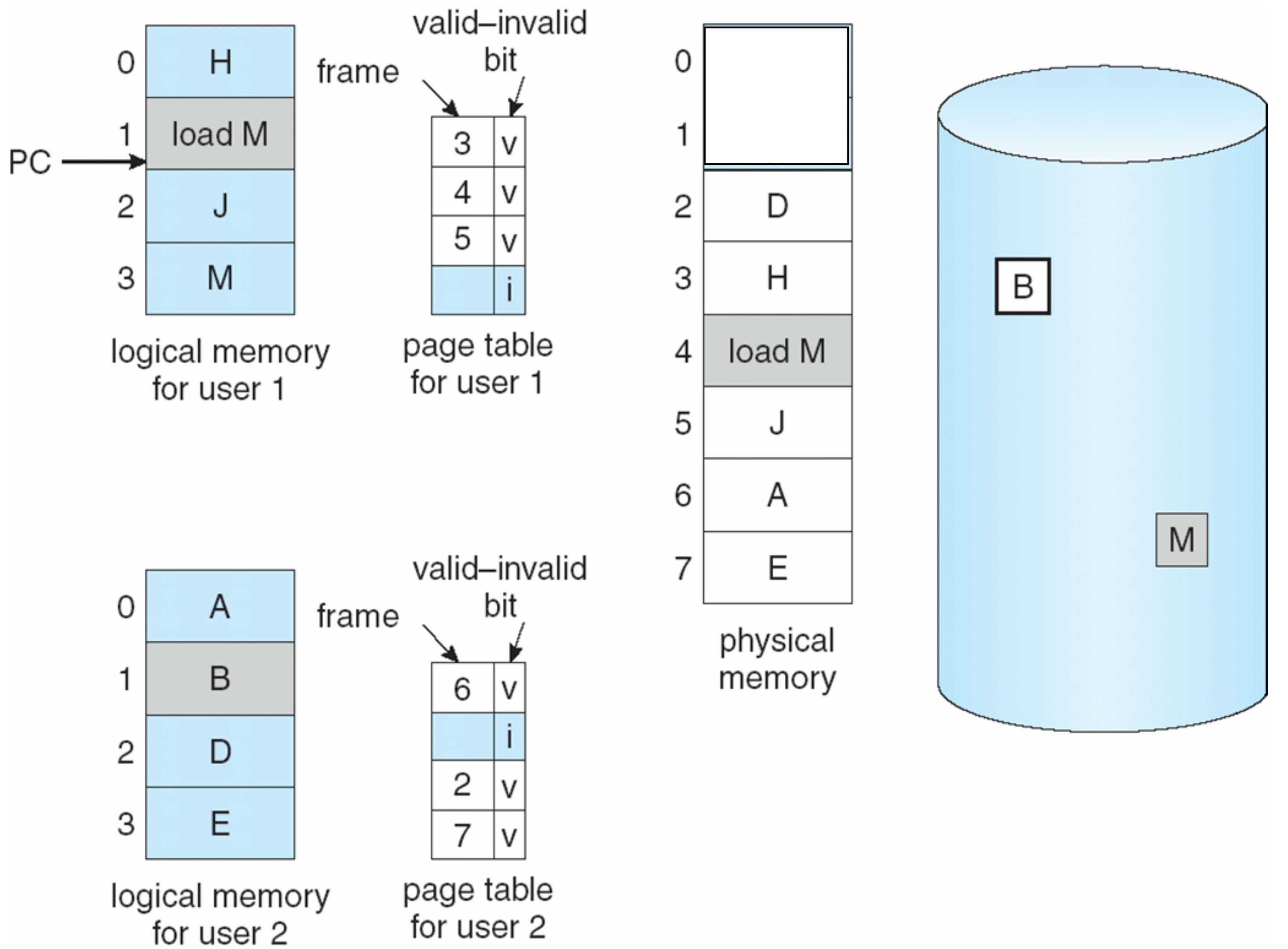
# After Process 1 Modifies Page C



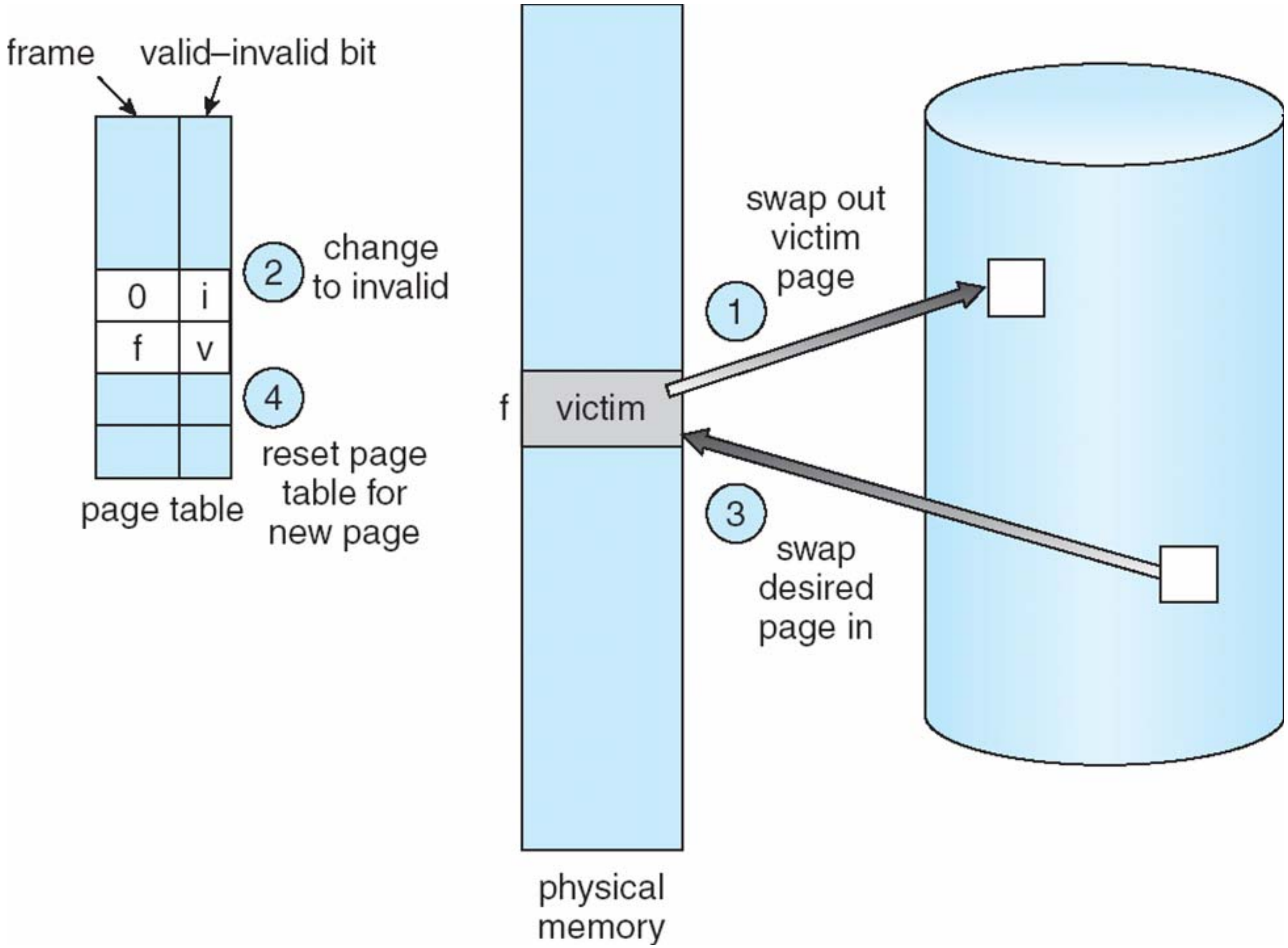
# What happens if there is no free frame?

- Page replacement
  - find some page in memory, but not really in use, swap it out
  - performance – want an algorithm which will result in minimum number of page fault
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

# Need For Page Replacement

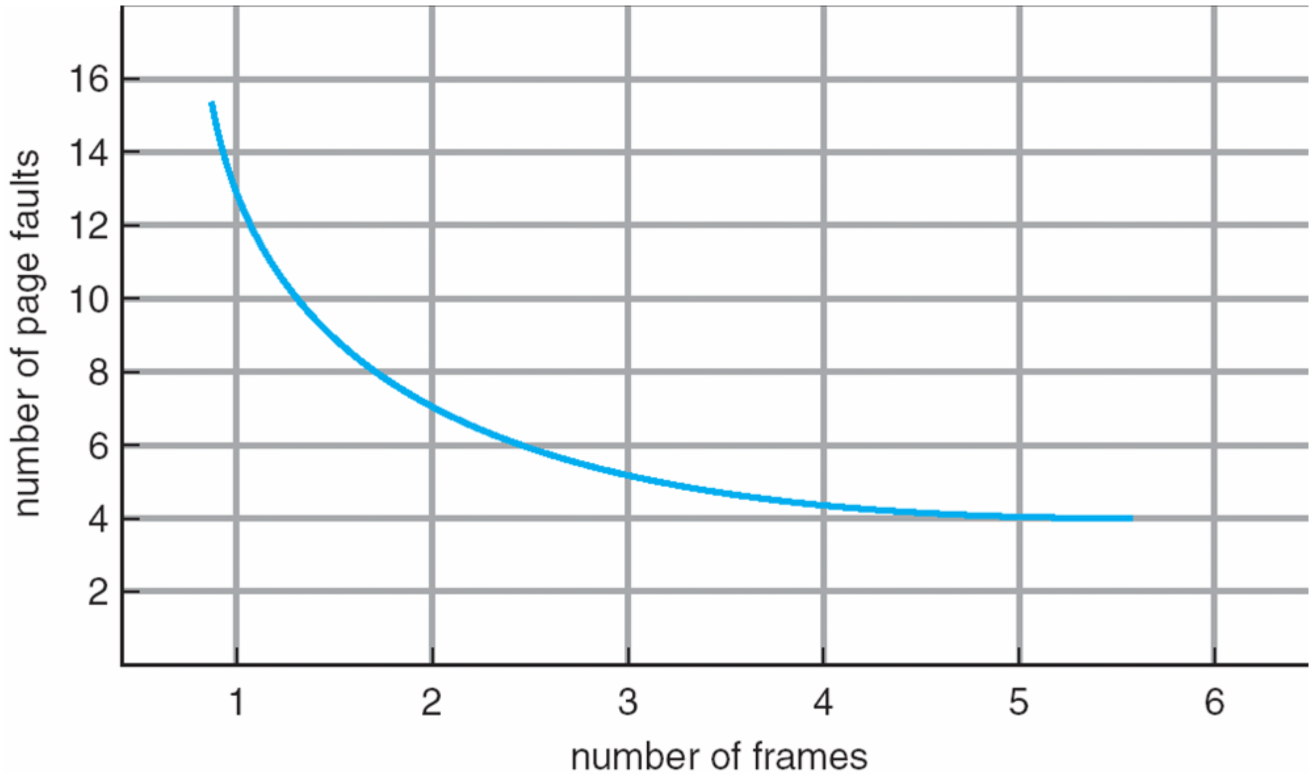


# Page Replacement



# Page Replacement Algorithms

- Minimize page-fault rate

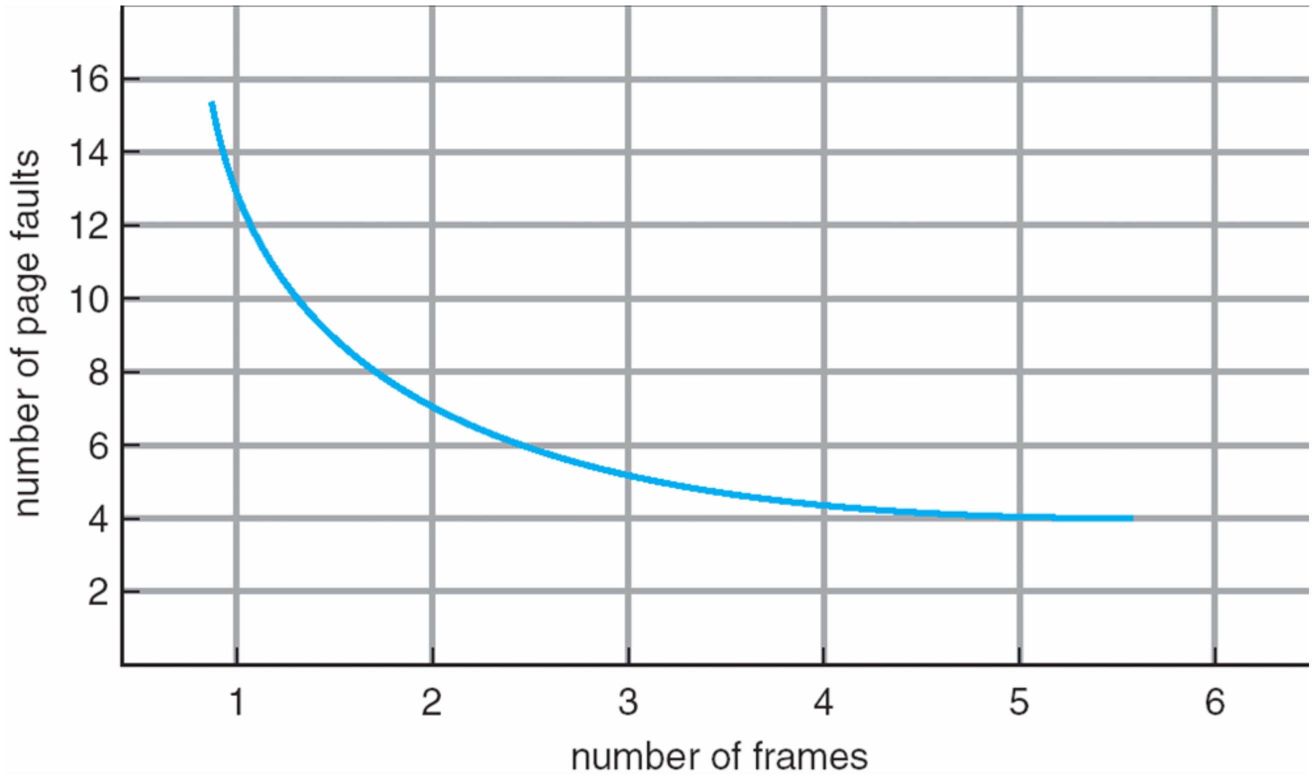


# Page Replacement Algorithms

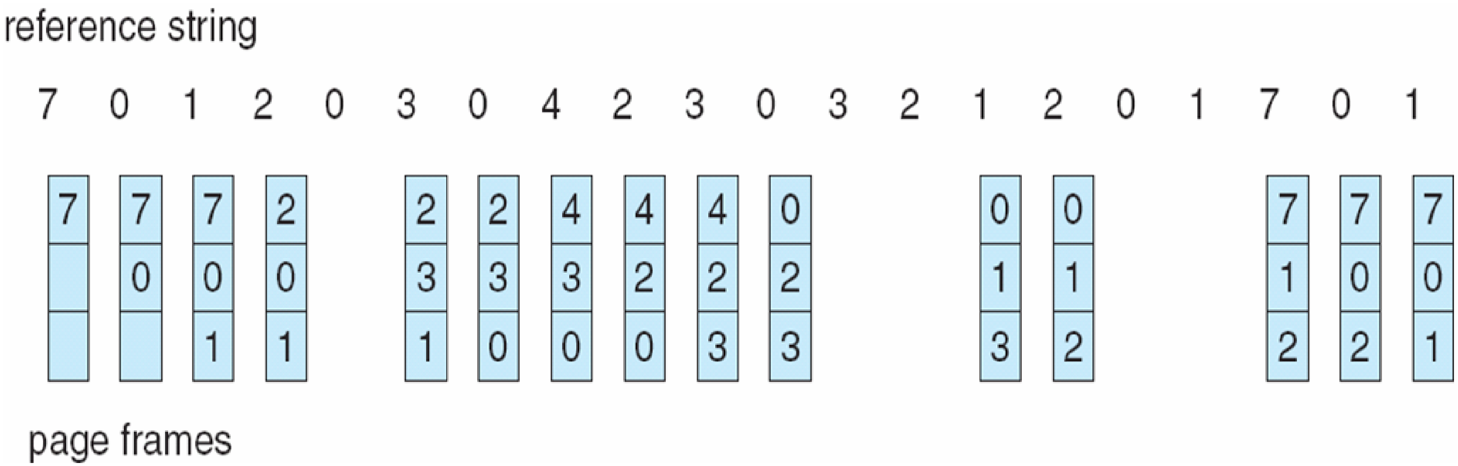
- Minimize page-fault rate

# Page Replacement Algorithms

- Minimize page-fault rate



# FIFO Page Replacement



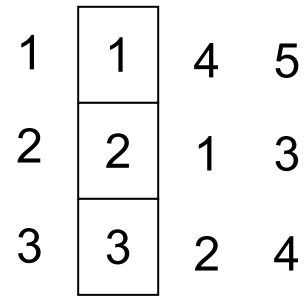
# First-In-First-Out (FIFO) Algorithm

- 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)

- 4 frames

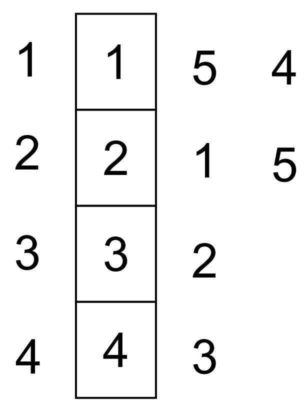
# First-In-First-Out (FIFO) Algorithm

- 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)




9 page faults

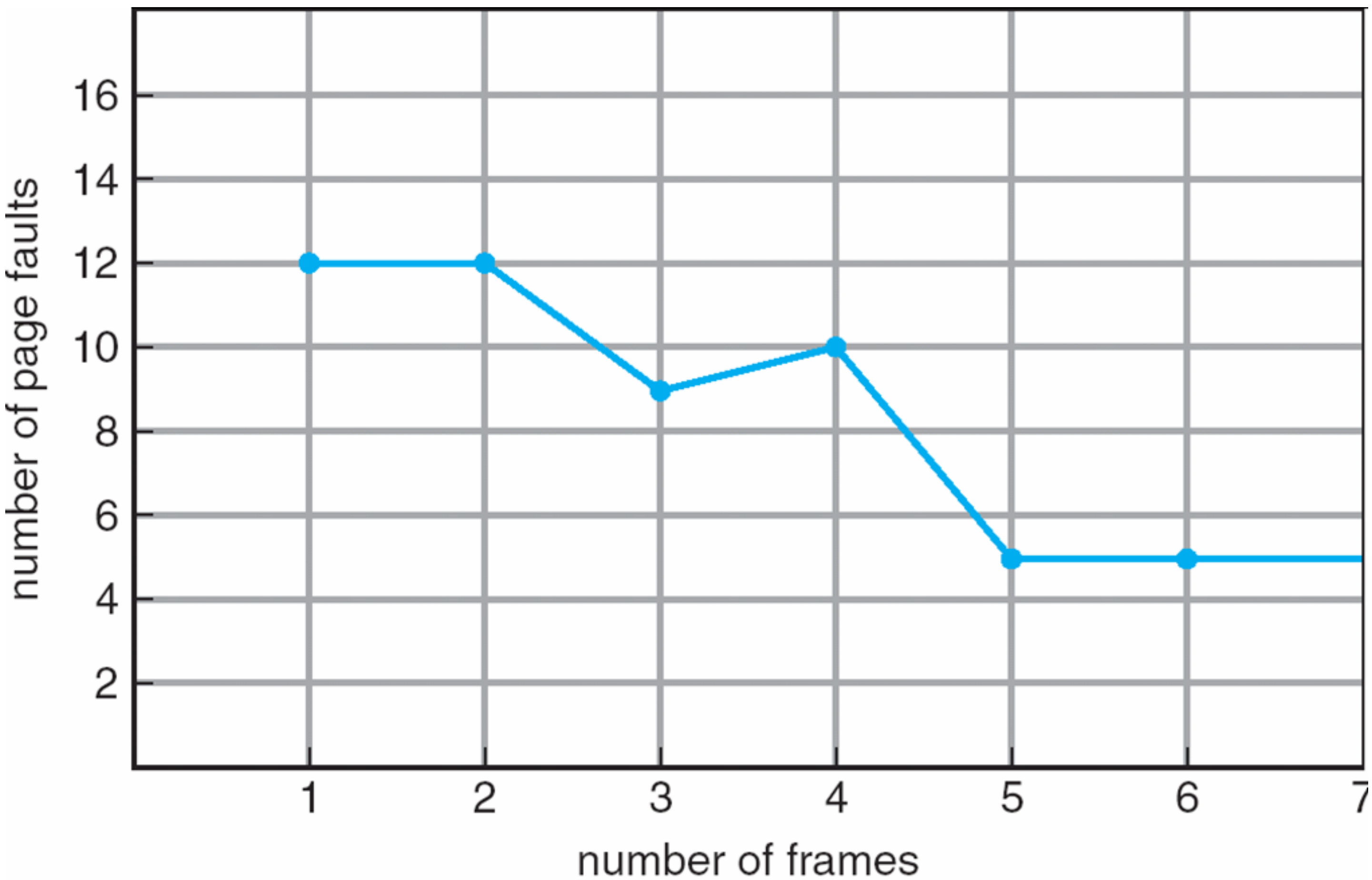
- 4 frames




10 page faults

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

# FIFO Illustrating Belady's Anomaly



# Optimal Algorithm

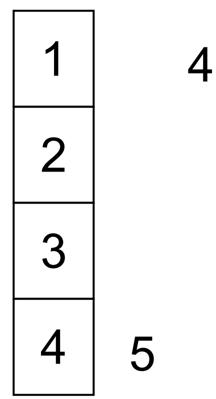
- Replace page that will not be used for longest period of time
- 4 frames example

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

# Optimal Algorithm

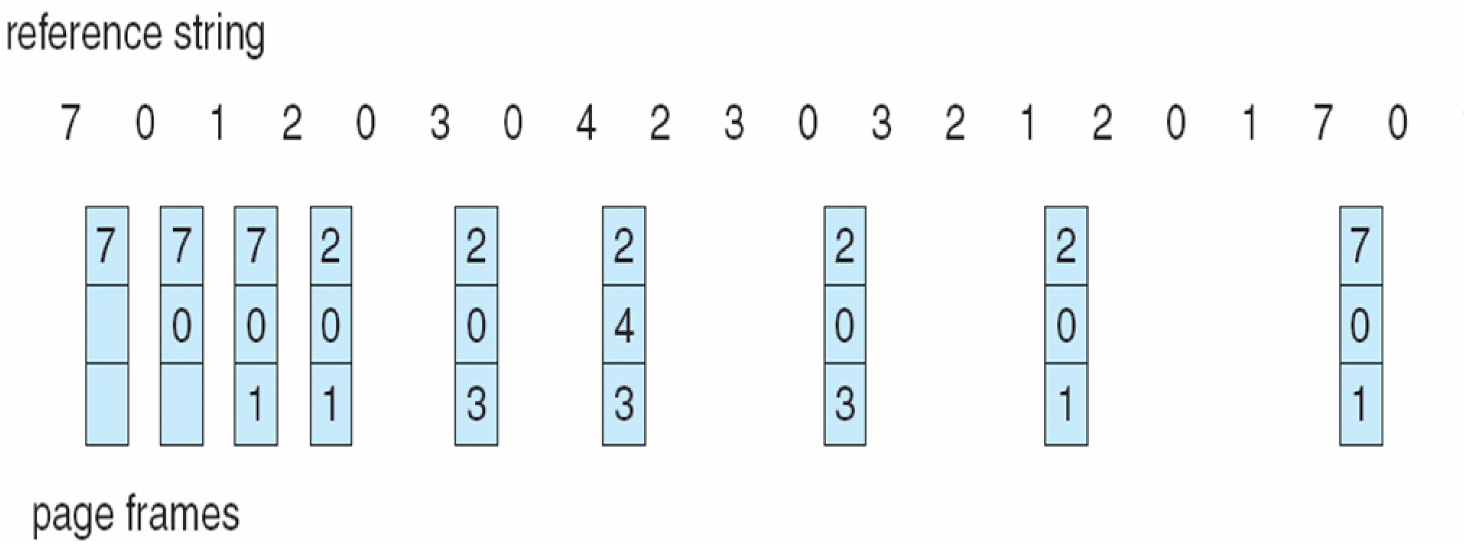
- Replace page that will not be used for longest period of time
- 4 frames example

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




6 page faults

# Optimal Page Replacement



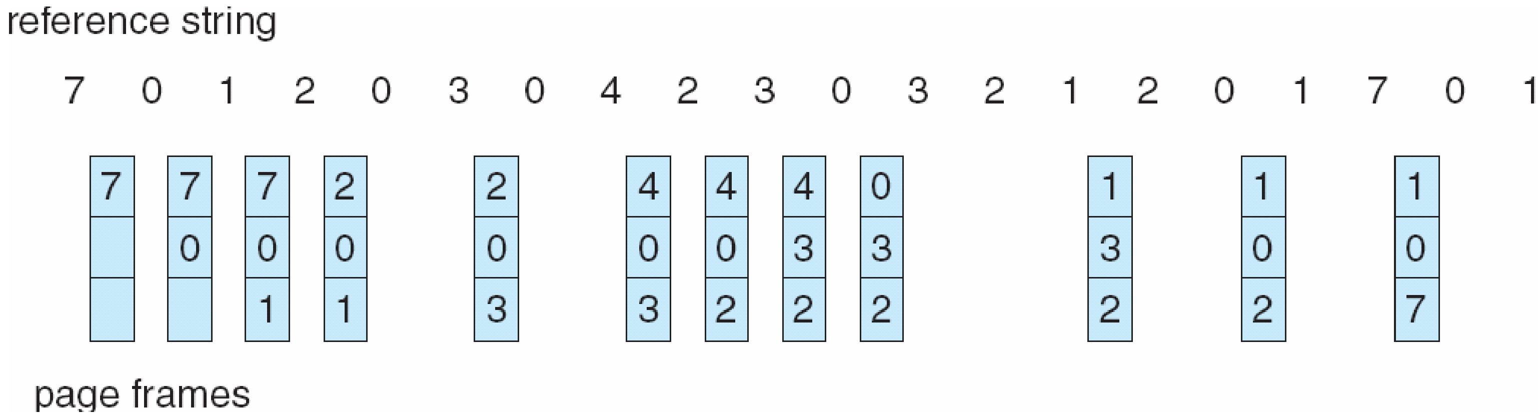
# Least Recently Used (LRU) Algorithm

- 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 |

- 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 determine which are to change

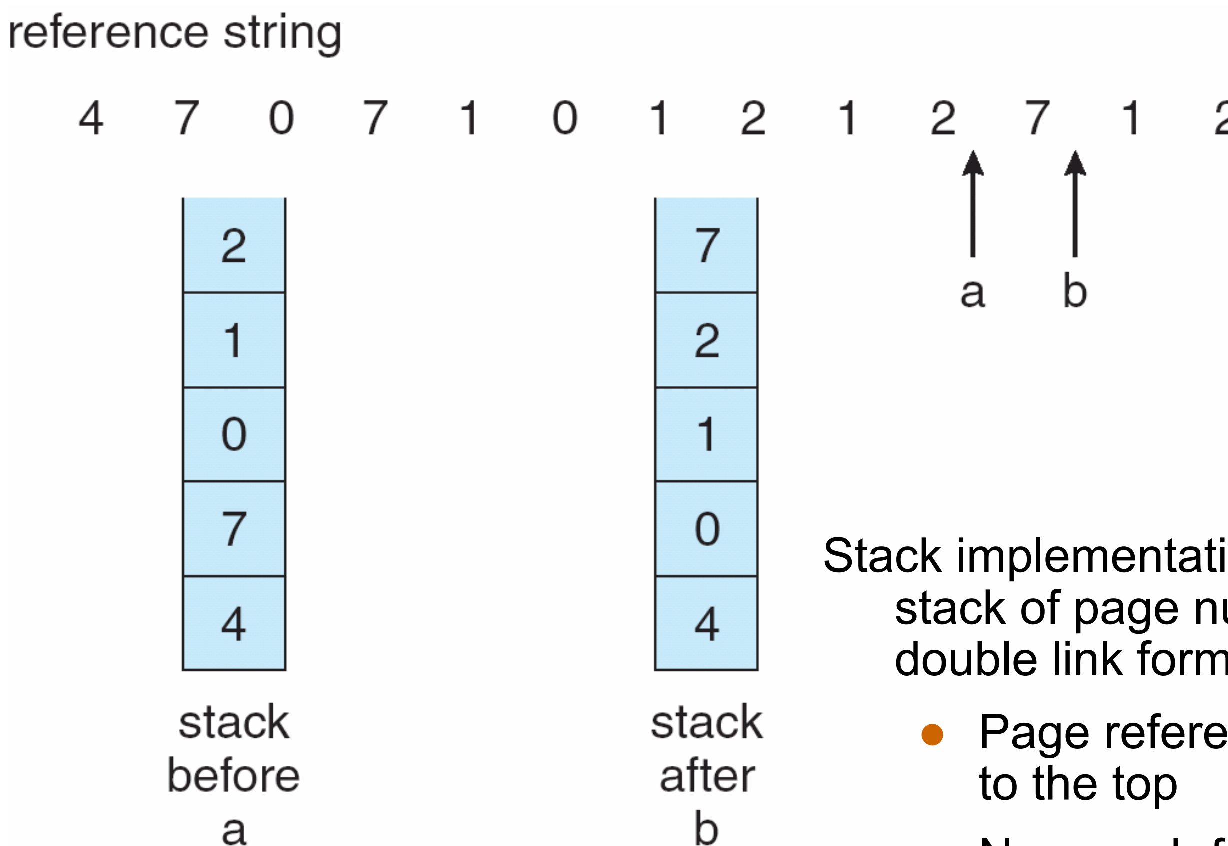
# LRU Page Replacement



# LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

# Use Of A Stack to Record The Most Recent Page References



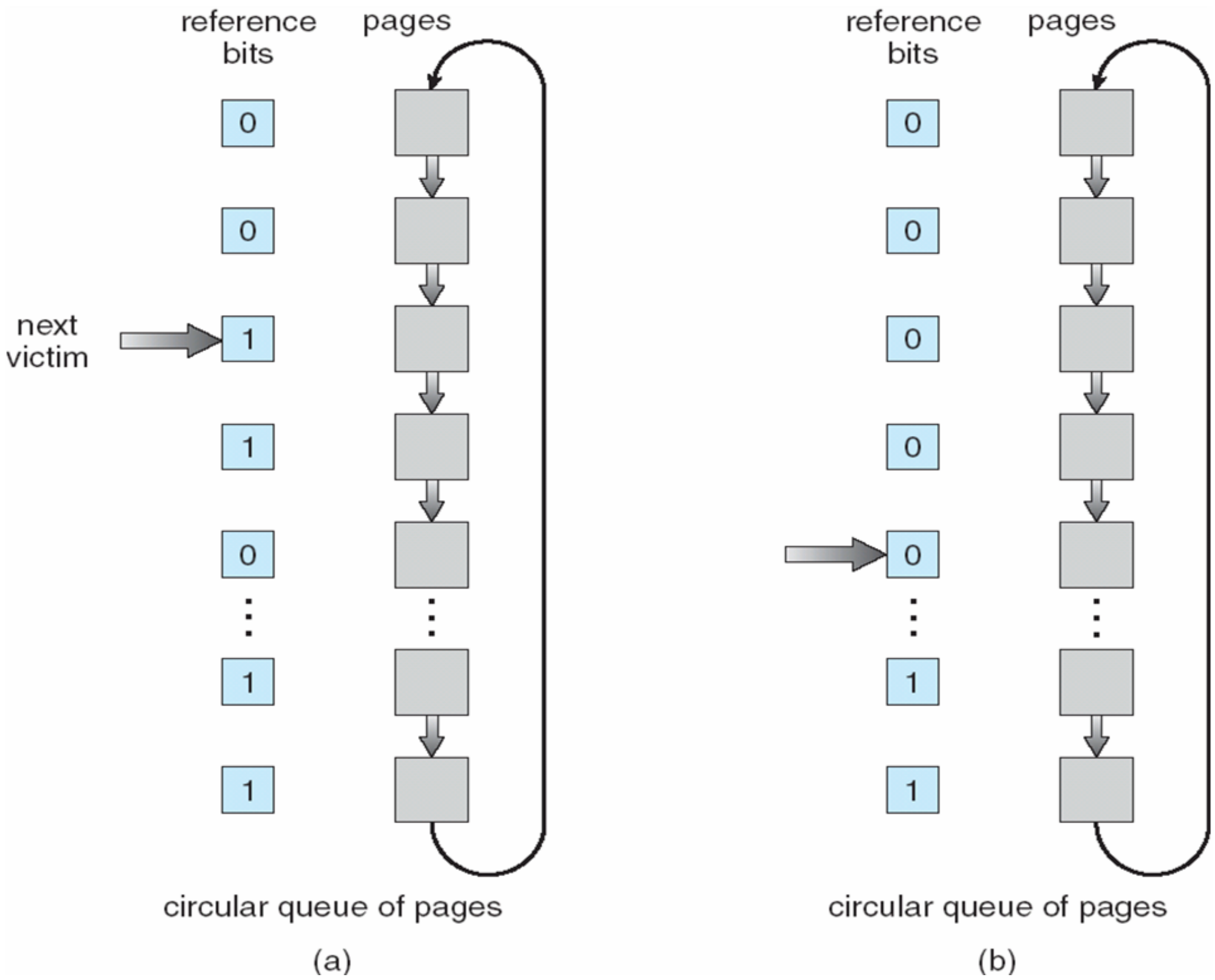

Stack implementation – keep a stack of page numbers in a double link form:

- Page referenced: move to the top
- No search for replacement

# LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

# Second-Chance (clock) Page-Replacement Algorithm



# See you next time