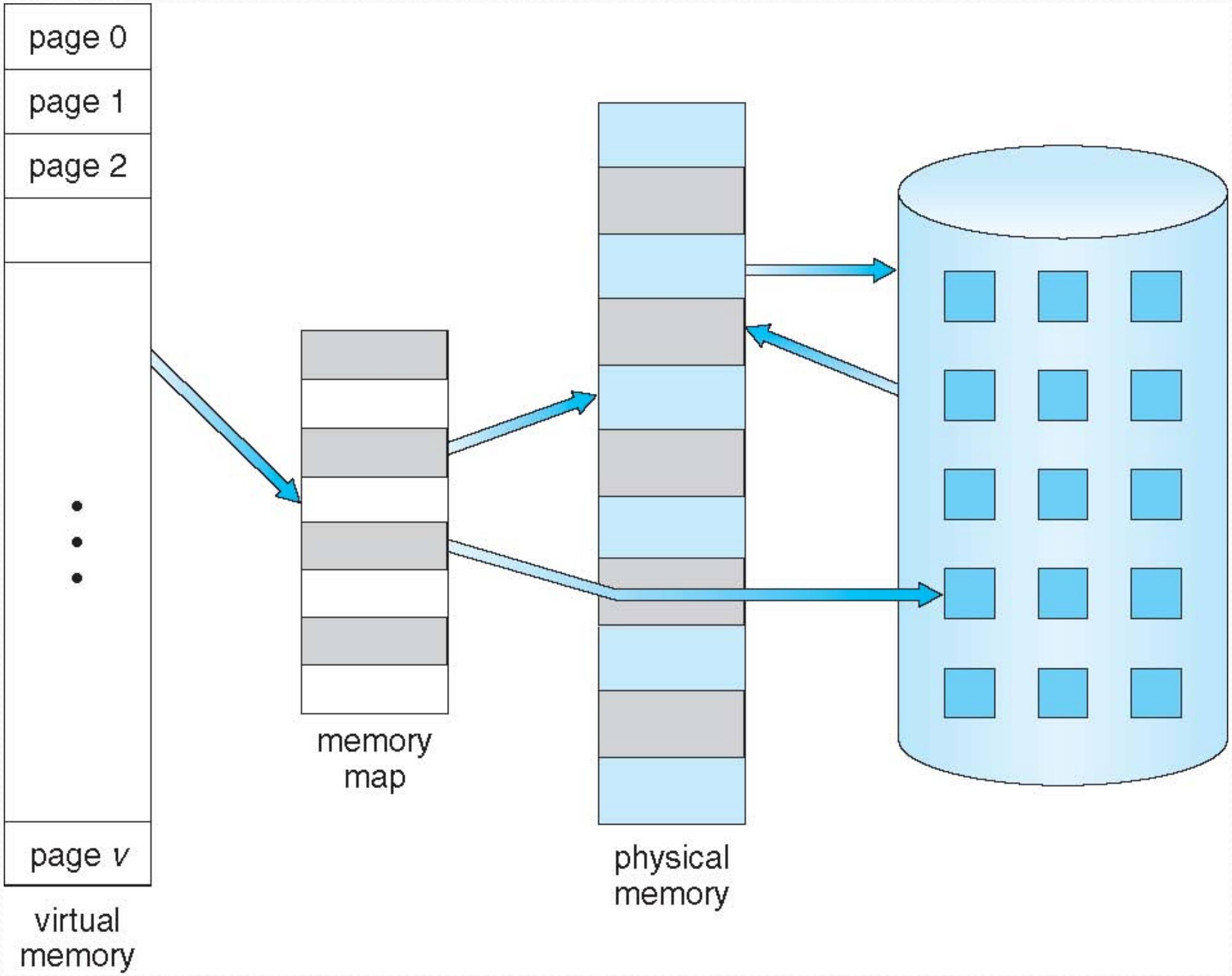
Virtual Memory

Virtual Memory

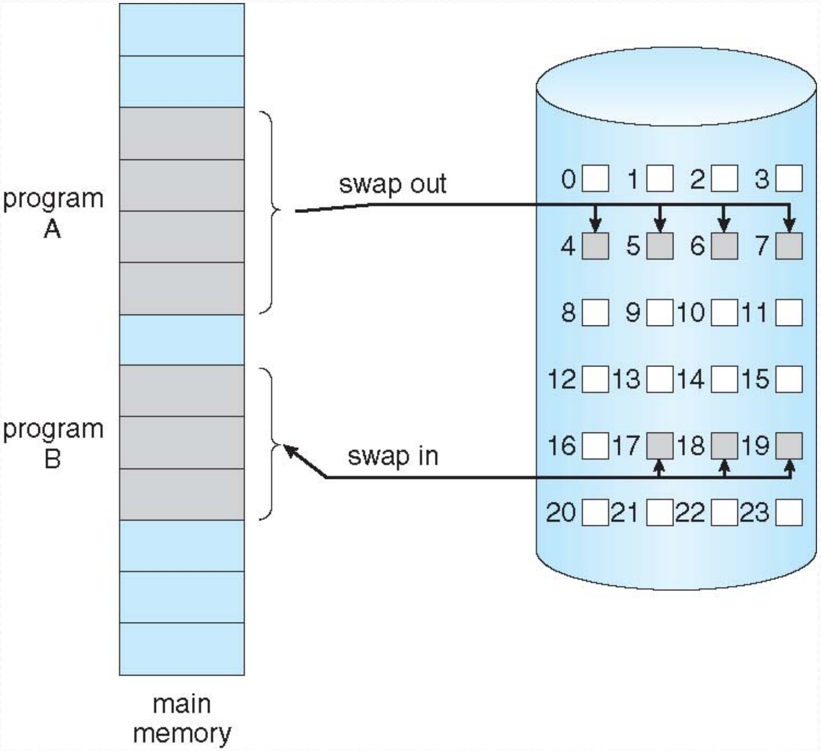
- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly

Virtual Memory That is Larger Than Physical Memory

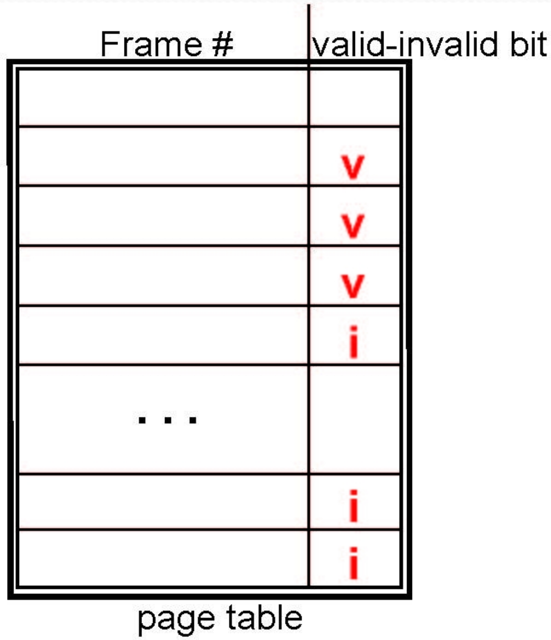


Demand Paging

- Bring a page into memory only when it is needed
- Pages that are never accessed are thus never loaded into physical memory
- Similar to paging system with swapping (diagram on right)
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

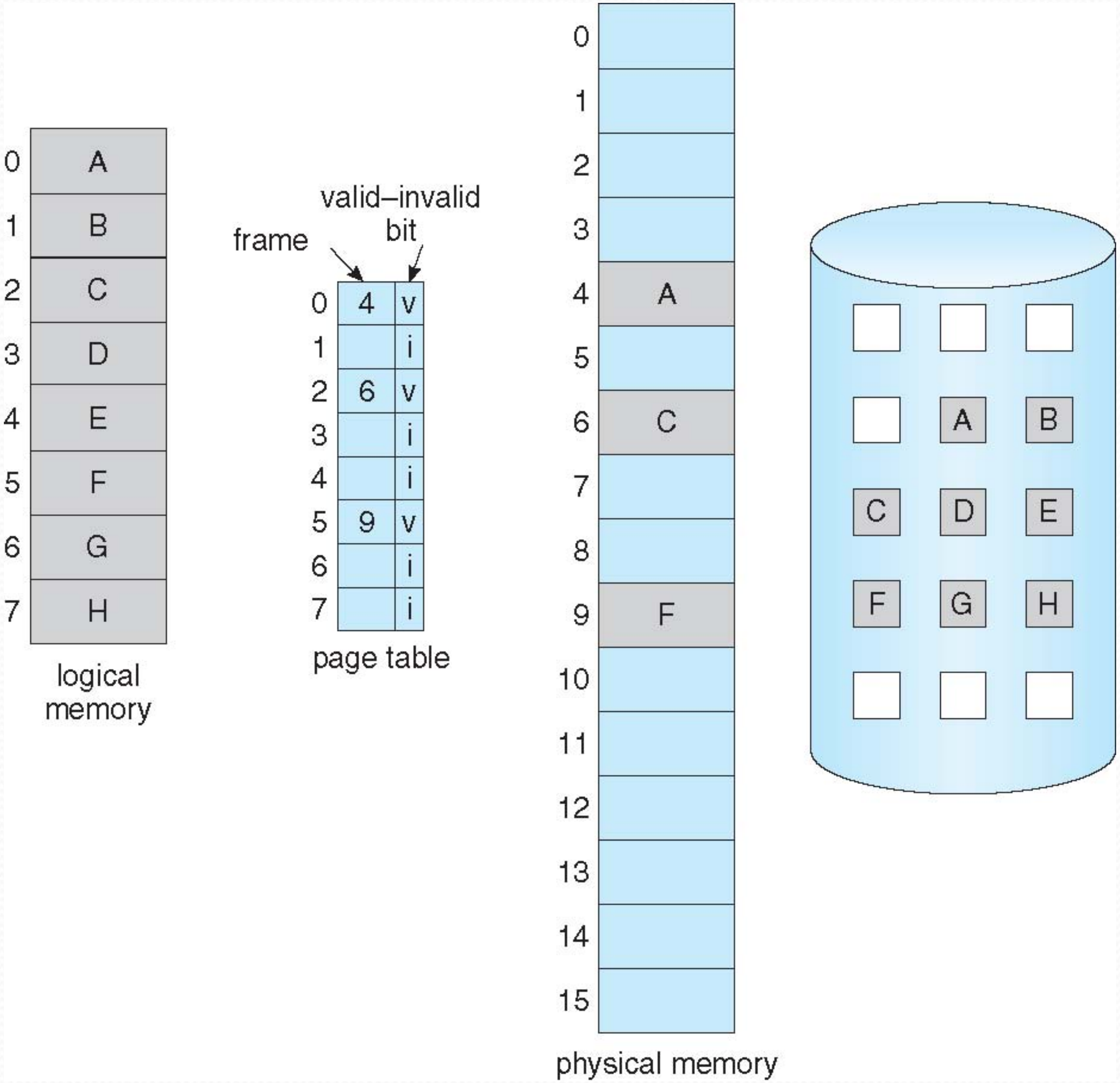


Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ 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:
- 

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

Page Table When Some Pages Are Not in Main Memory



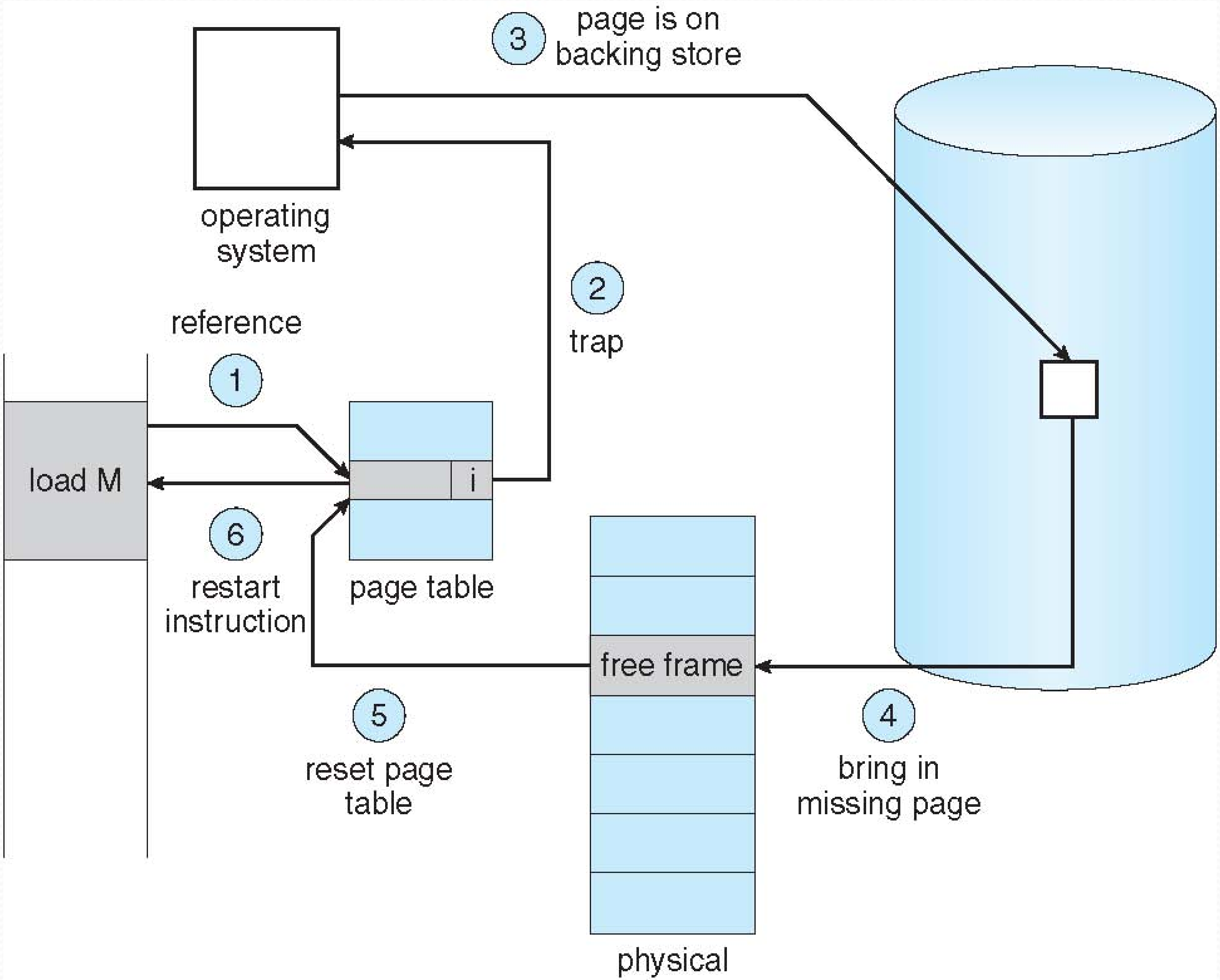
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at internal 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

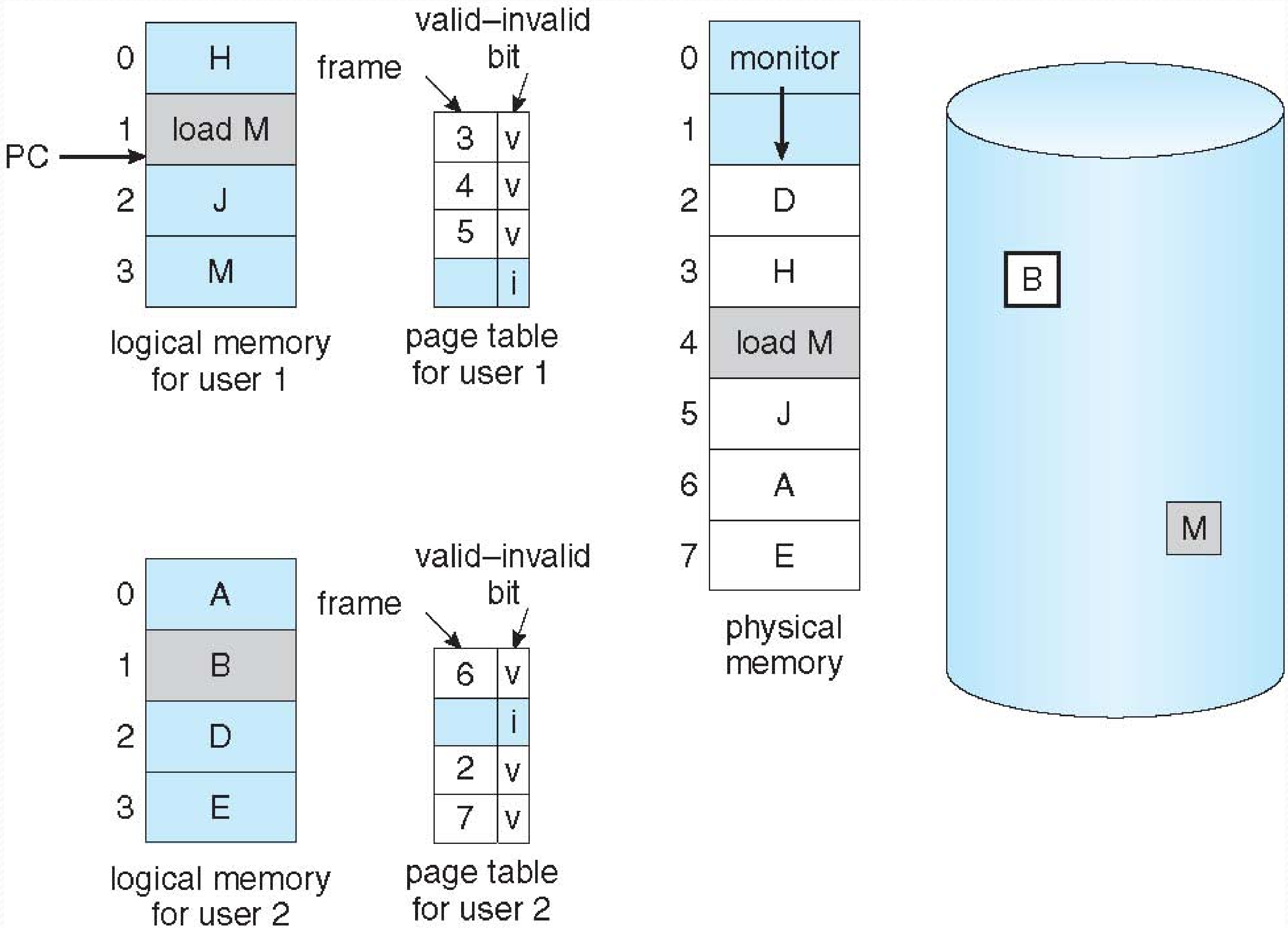
Steps in Handling a Page Fault



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access - **Pure demand paging**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
    - Usually a high-speed disk

Need For Page Replacement



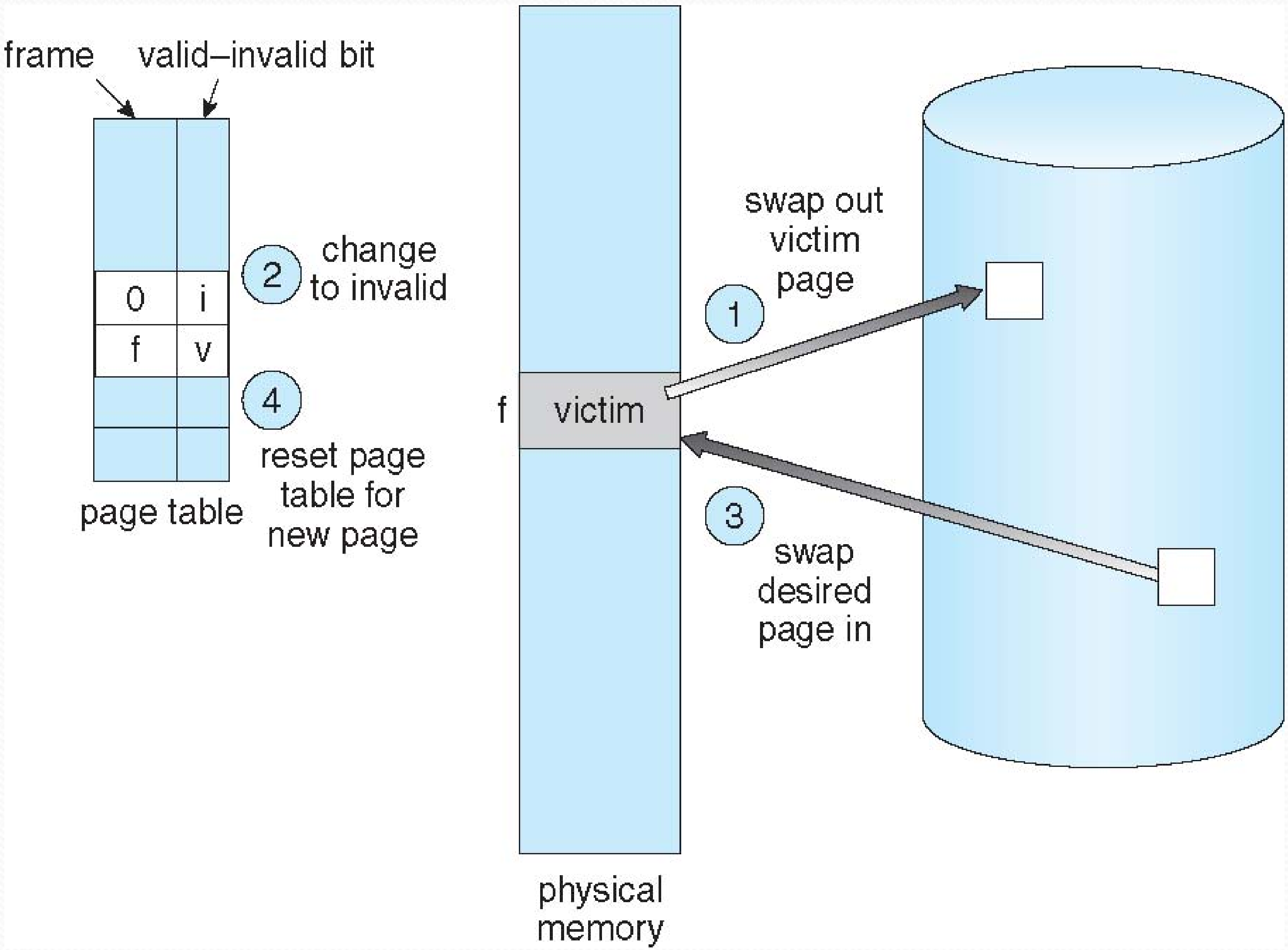
Basic Page Replacement

1. 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**
     - Write victim frame to disk if dirty

2. Bring the desired page into the (newly) free frame; update the page and frame tables

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

Page Replacement

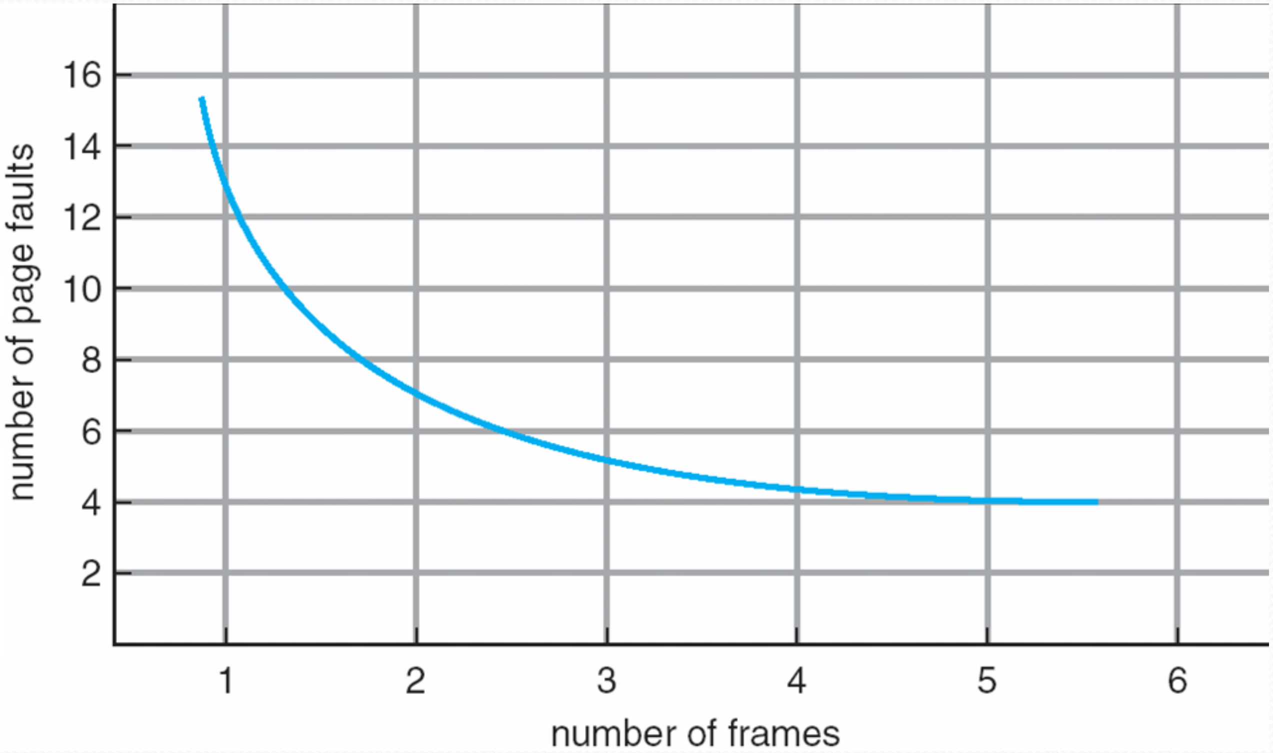


Page and Frame Replacement Algorithms

- **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 (reference string) and computing the number of page faults on that string
- In all our examples, the **reference string** of referenced page numbers is

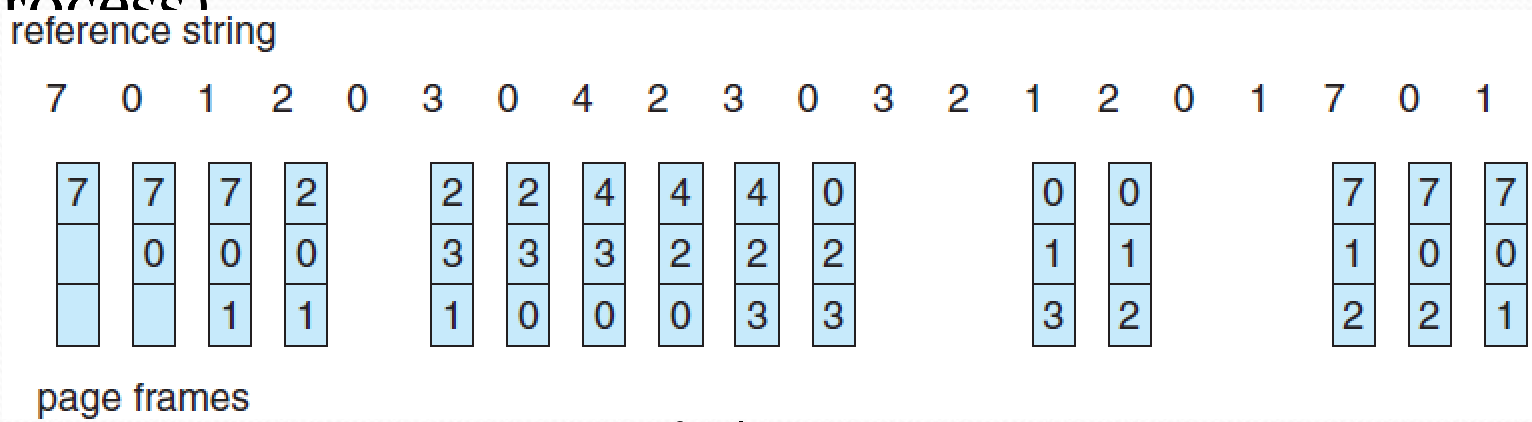
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



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)

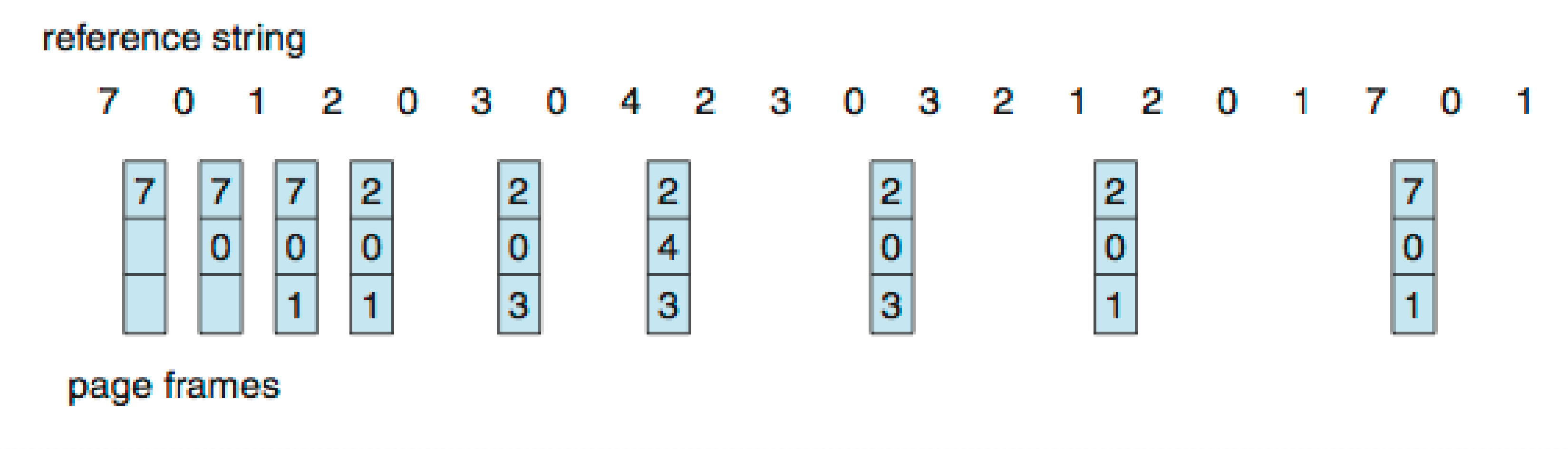


15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**

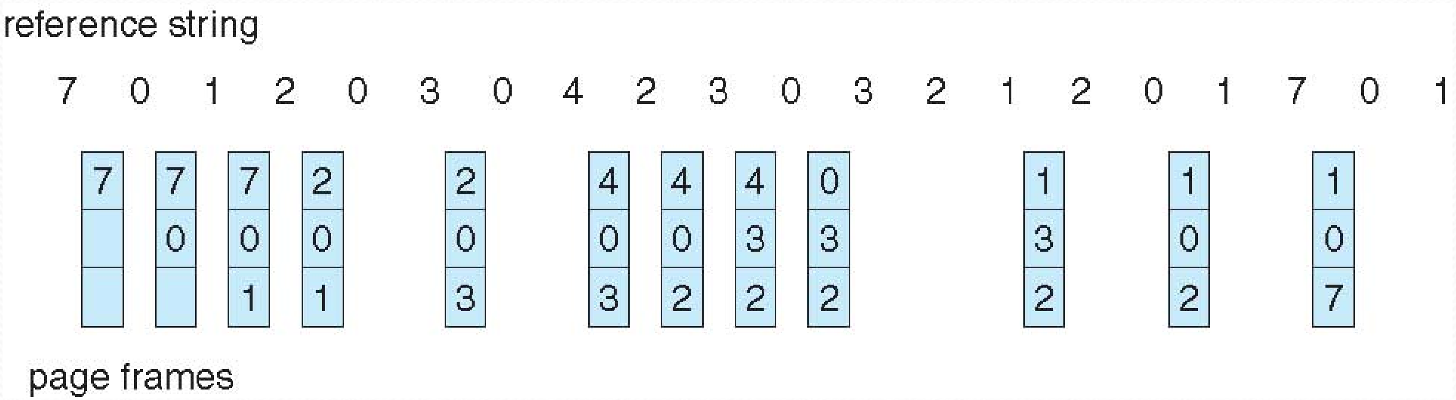
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example



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



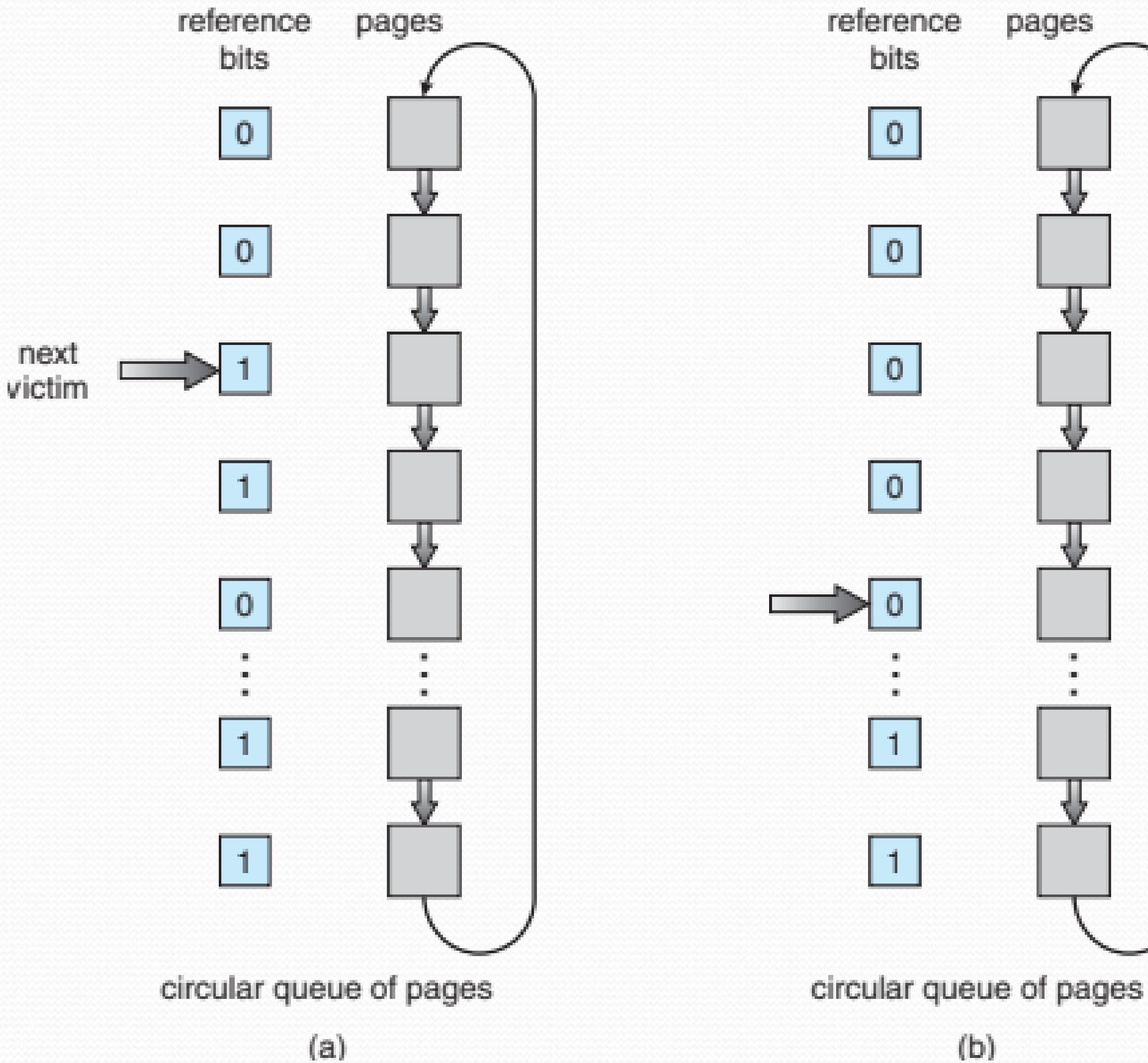
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

LRU Approximation Algorithms

- **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)

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - 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

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

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

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

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Two major allocation schemes
  - fixed allocation
  - Proportional allocation

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
- Let the size of the virtual memory for process *pi* be *si*, and define

$$S = \sum s_i .$$

- Then, if the total number of available frames is *m*, we allocate *ai* frames to process *pi*, where *ai* is approximately

$$a_i = s_i / S \times m.$$

Global vs. Local Allocation

- **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

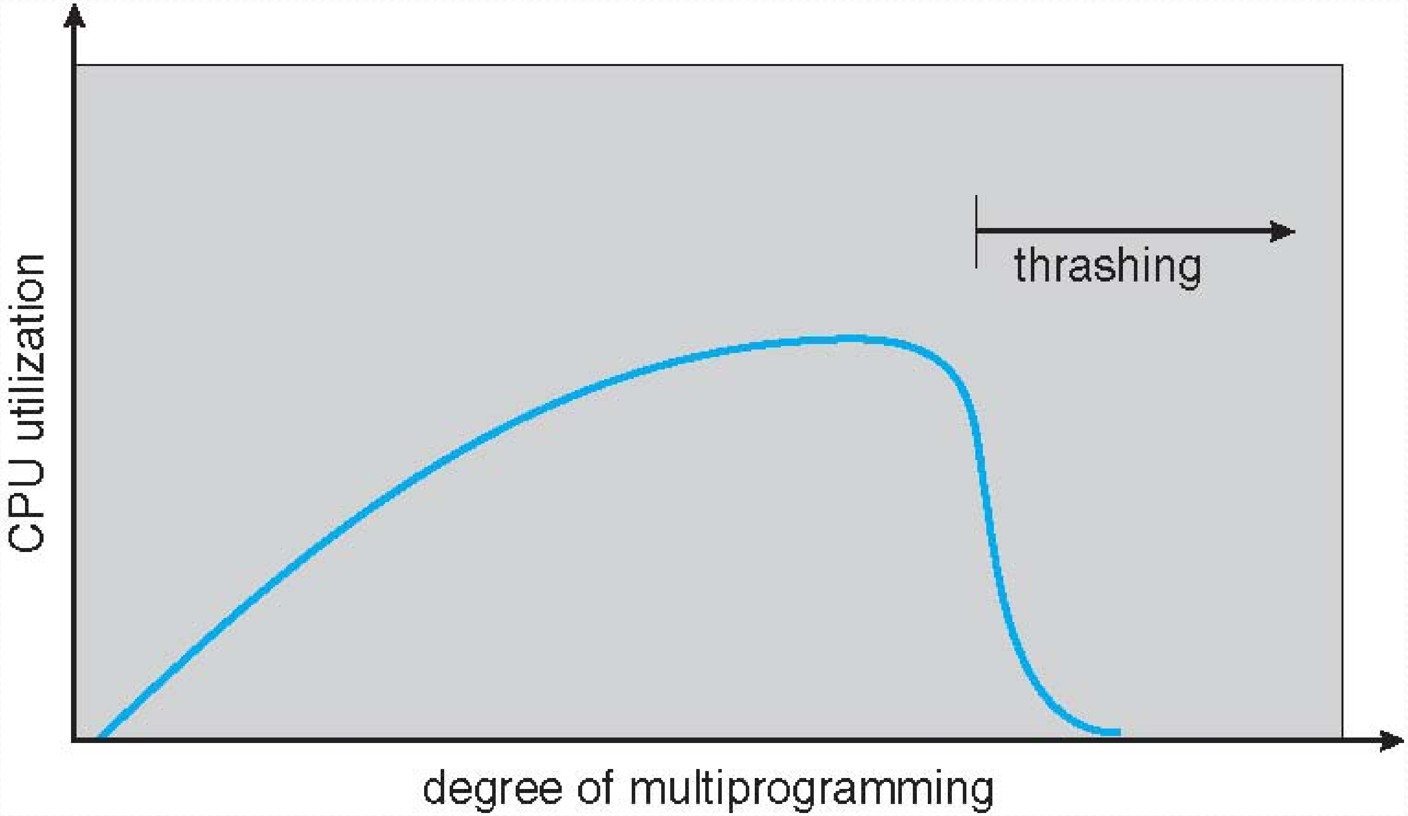
Thrashing

- A process is thrashing if it is spending more time paging than executing

- If a process does not have “enough” pages, the page-fault rate is very high
  - This leads to Low CPU utilization

- This high paging activity is called **thrashing**

Thrashing (Cont.)



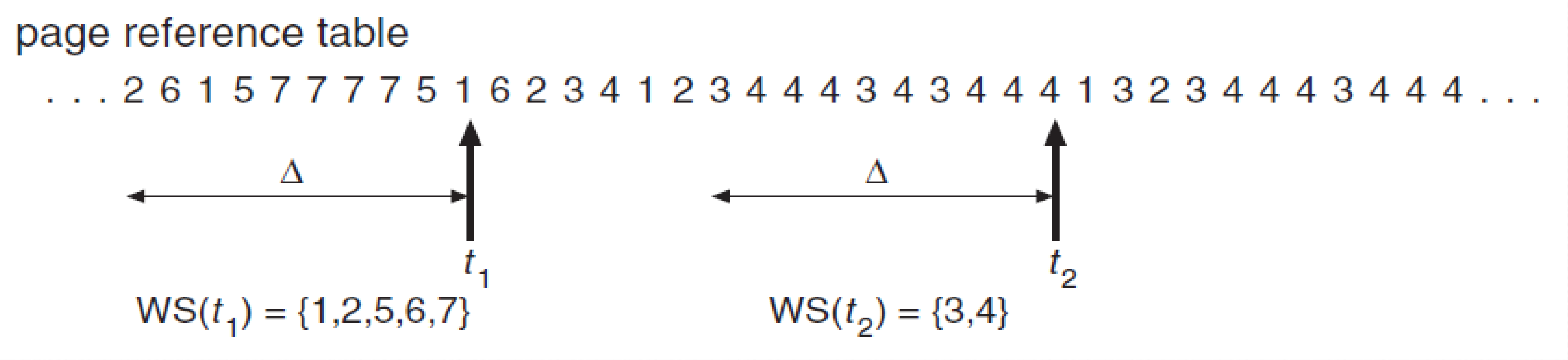
Thrashing (contd…)

- To prevent thrashing, we must provide a process with as many frames as it needs.
- But how do we know how many frames it “needs”?
- The working-set strategy is one such technique
- This approach defines the locality model of process execution
- The locality model states that, as a process executes, it moves from locality to locality
- A locality is a set of pages that are actively used together
- A program is generally composed of several different localities, which may overlap

Working-Set Model

- The working-set model is based on the assumption of locality
- This model uses a parameter, Δ to define the working-set window
- The idea is to examine the most recent Δ page references
- The set of pages in the most recent Δ page references is the working set
- If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference
- Thus, the working set is an approximation of the program's locality

Working-set model



Working-Set Model

- The accuracy of the working set depends on the selection of Δ.
  - If Δ is too small, it will not encompass the entire locality
  - if Δ is too large, it may overlap several localities
- The most important property of the working set, then, is its size. If we compute the working-set size, WSS_i, for each process in the system, we can then consider that

$$D=\sum WSS_i,$$

- whereD is the total demand for frames.
- If the total demand is greater than that of available frames(D>m),thrashing will occur

Working-Set Model

- OS monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size
- If there are enough extra frames, another process can be initiated
- If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend
- This strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it optimizes CPU utilization
- The difficulty with the working-set model is keeping track of the working set

Any queries?