Virtual Memory

Outline of this lecture

In this lecture, we will discuss the following:

- Background
- Demand Paging
- Performance of Demand Paging
- Page Replacement
- Page-Replacement Algorithms
- Thrashing

Recall from previous chapter

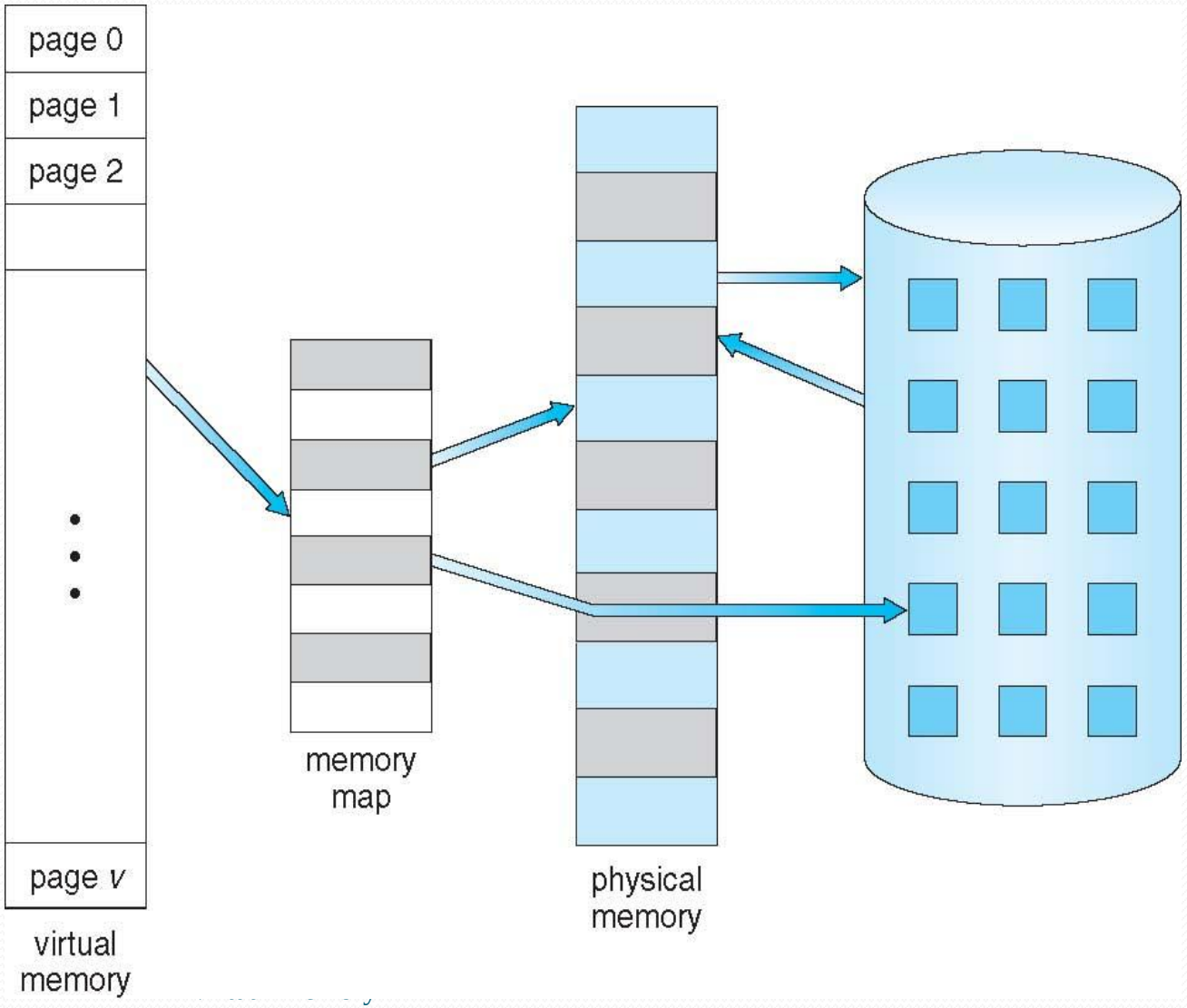
- One goal of memory management is to keep as many processes in memory simultaneously to allow multiprogramming.
- Yet, they require that an entire process be in memory before it can be executed.
- Therefore, the size of the program cannot exceed the size of physical memory.
- Usually the entire code is not needed I,e.
  - Code to handle error conditions, which seldom occur doesn't need to be in main memory.
  - Even if entire program is needed it may not be needed at the same time.
- *Virtual memory* is a technique that allows the execution of processes that are only partially in memory.

Background

- 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.
  - More programs can run at the same time - each user program takes up less physical memory. This increases CPU utilization and throughput.

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

Virtual Memory That is Larger Than Physical Memory



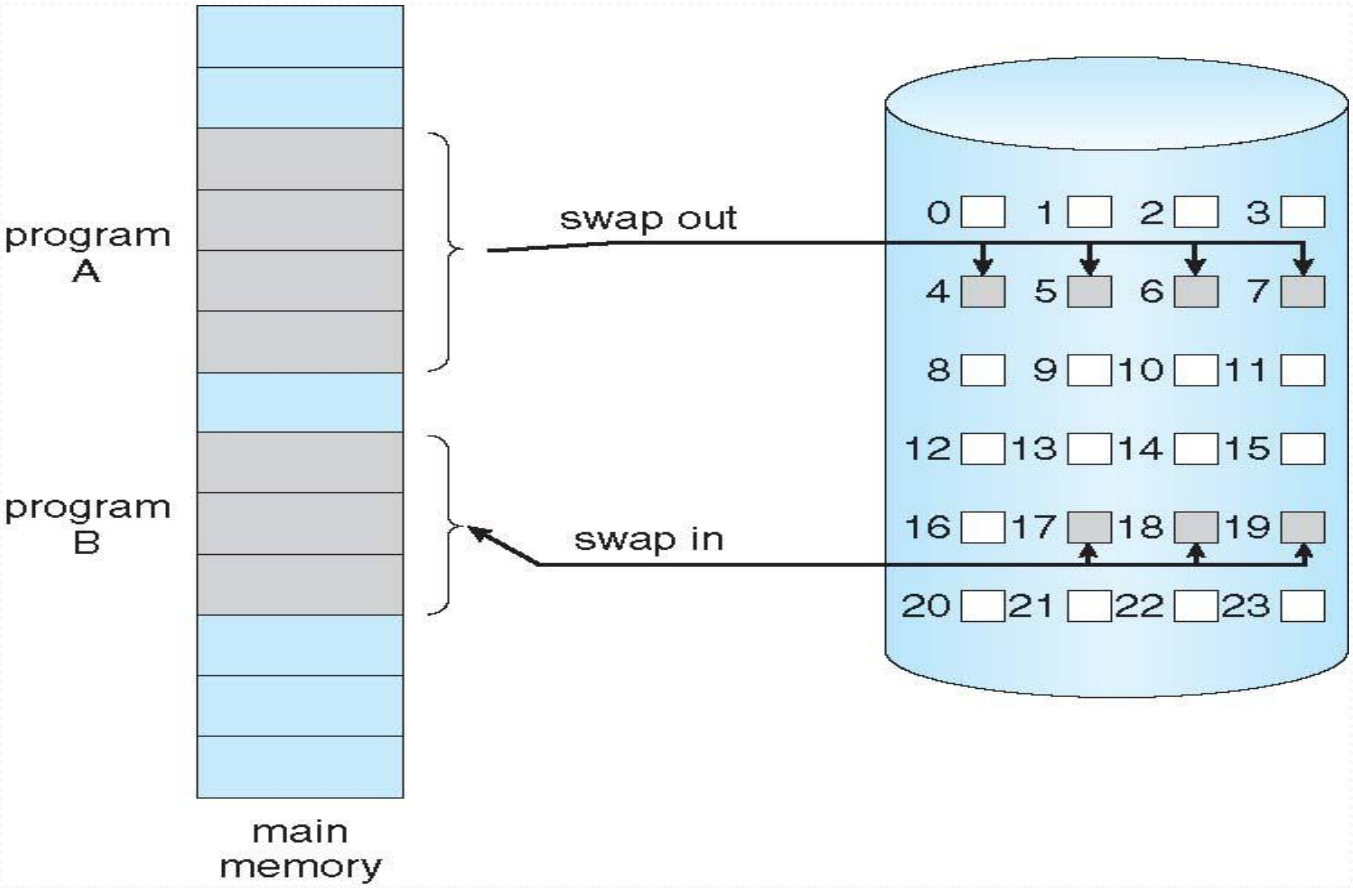
Demand paging

Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for all options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems.

Demand Paging

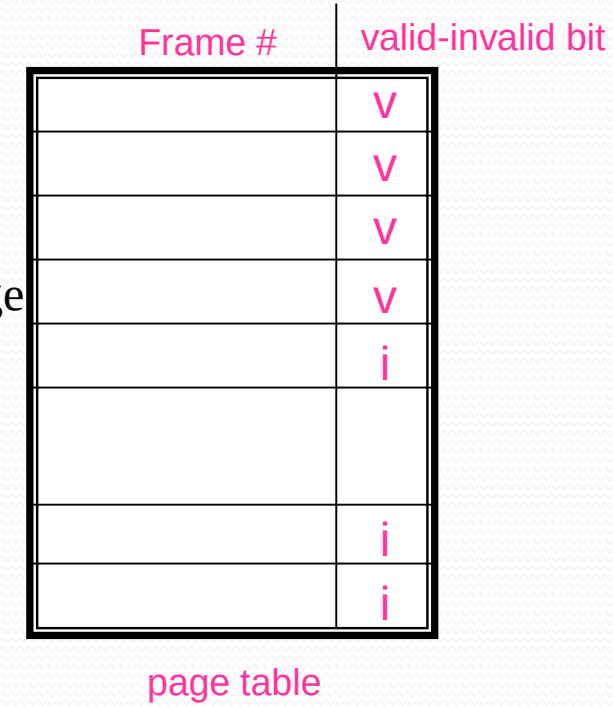
- In demand paging, processes reside on secondary storage (usually a hard disk).
- Bring a page into memory only when it is needed using a **lazy swapper** (never swaps a page into memory unless that page will be needed).
- When the process to be swapped in, the pager guesses which pages will be needed and only swaps those in.
  - Less I/O needed for loading and swapping
  - Less memory needed
  - Faster response
  - More users

Transfer of a Paged Memory to Contiguous Disk Space



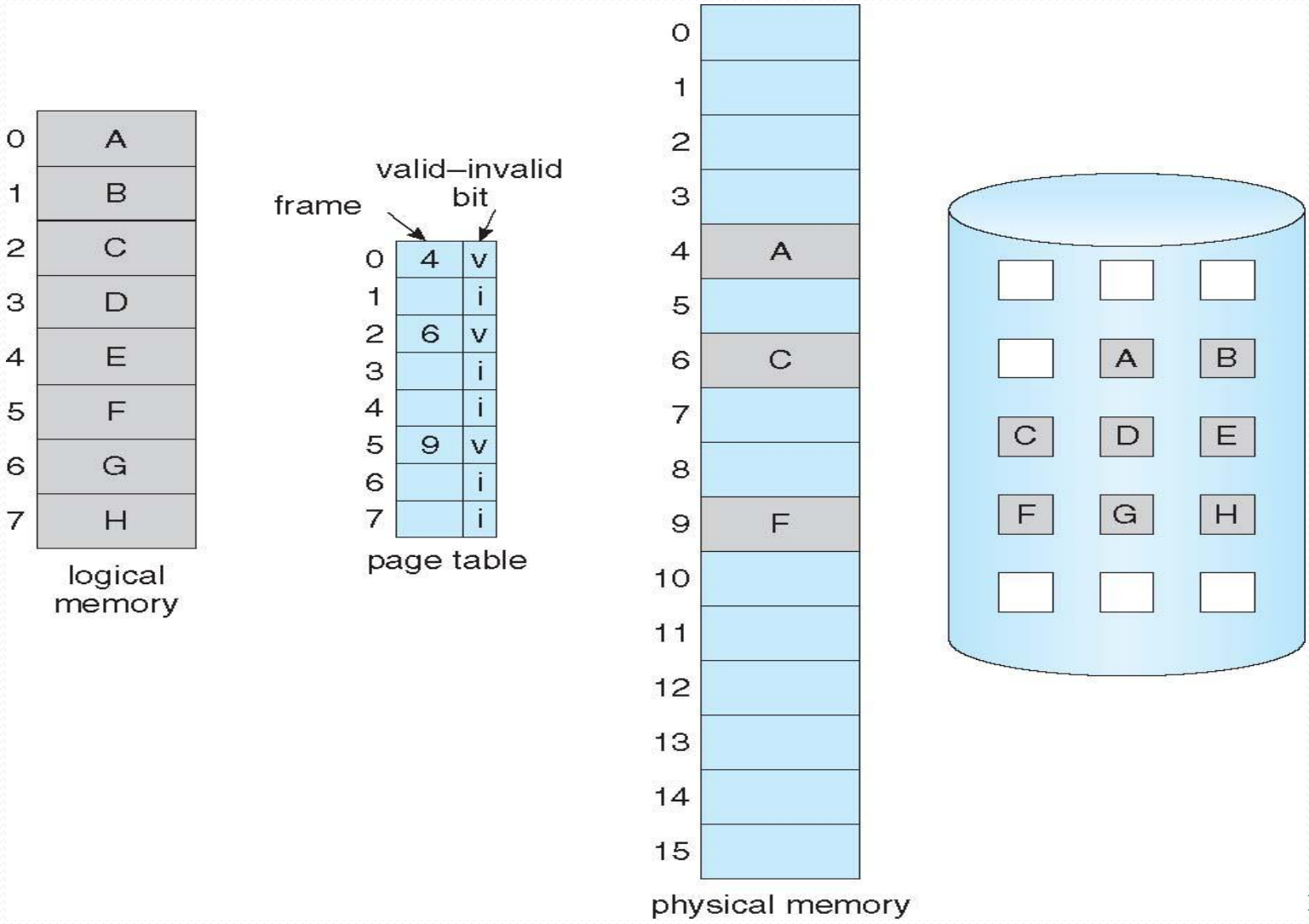
Valid-Invalid Bit

- We need to distinguish between pages in memory and pages on disk.
- With each page table entry a valid–invalid bit is associated (v ⇒ in-memory, i ⇒ not-in-memory)
- Example of page table snapshot:



- During address translation, if valid– invalid bit in page table entry is 0, it generates a page fault.

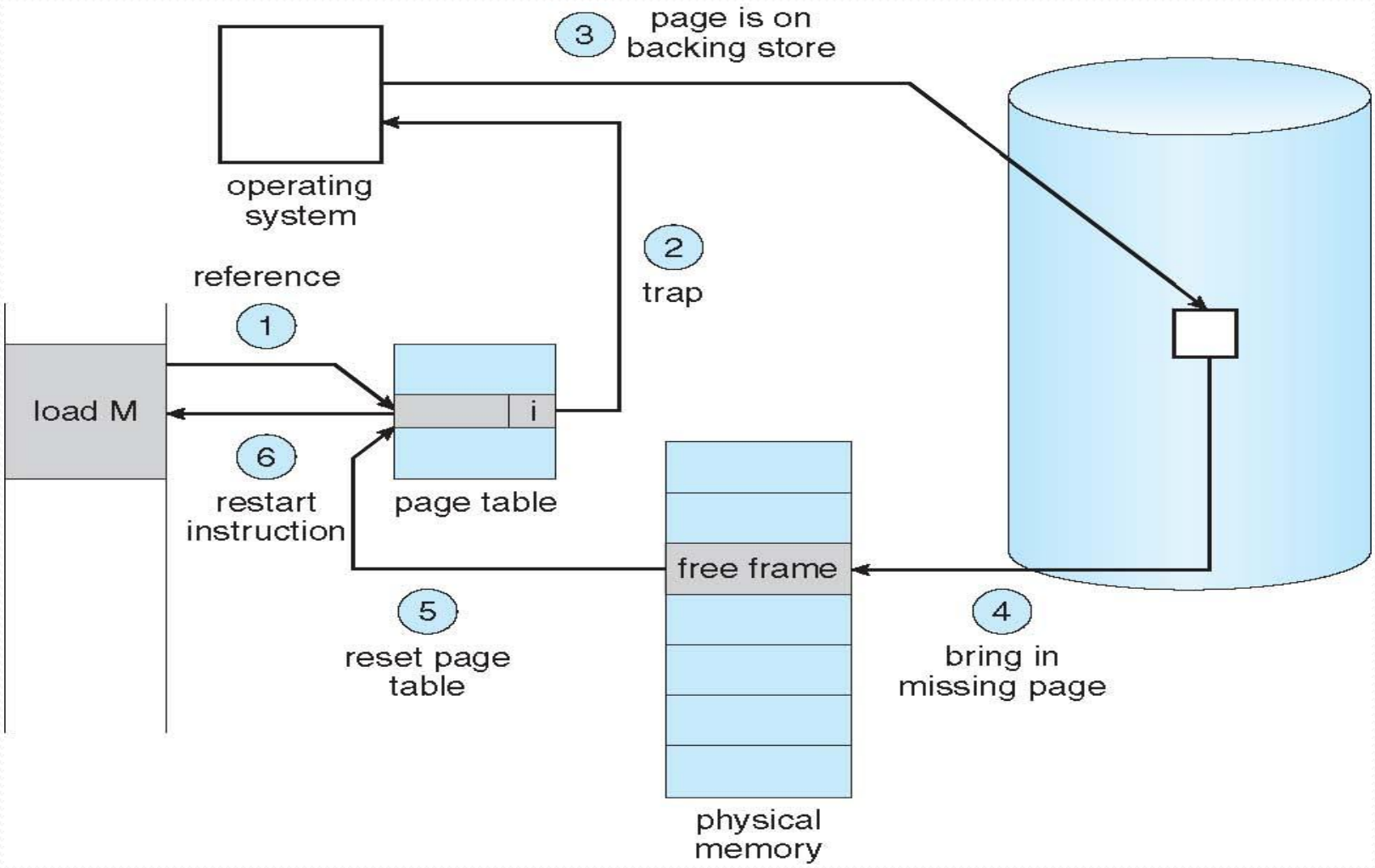
Page Table When Some Pages Are Not in MM



Page Fault

- Access to a page marked invalid ⇒ page fault trap.
- The paging HW (Page table & secondary memory) notices "i" is set ⇒ trap (the result of the OS's failure to bring the desired page into memory) to the OS.
- OS looks at an internal table (usually in the PCB) to determine whether the memory access is valid.
  - Invalid reference ⇒ abort (terminate the process).
  - Just not in memory (valid but not yet brought in, page it).
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = v.
- Restart instruction that was interrupted by the trap.

Steps in Handling a Page Fault



What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out.
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

Performance of Demand Paging

- Demand paging affect the performance of a computer system.
- Page Fault Rate $0 \leq p \leq 1.0$
  - if p = 0 no page faults --→ EAT = MAT
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – p) x ma + p * page fault service time

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 nsecs
- Page fault service time= 8 msecs
- Effective access time in nanoseconds is:

$$EAT = (1 - p) \times ma + p \times \text{page fault service time}$$
$$= (1 - p) \times 200 + p \times 8000000$$
$$= 200 + 7999800\ p \text{ (in nsec)}$$

Basic Page Replacement

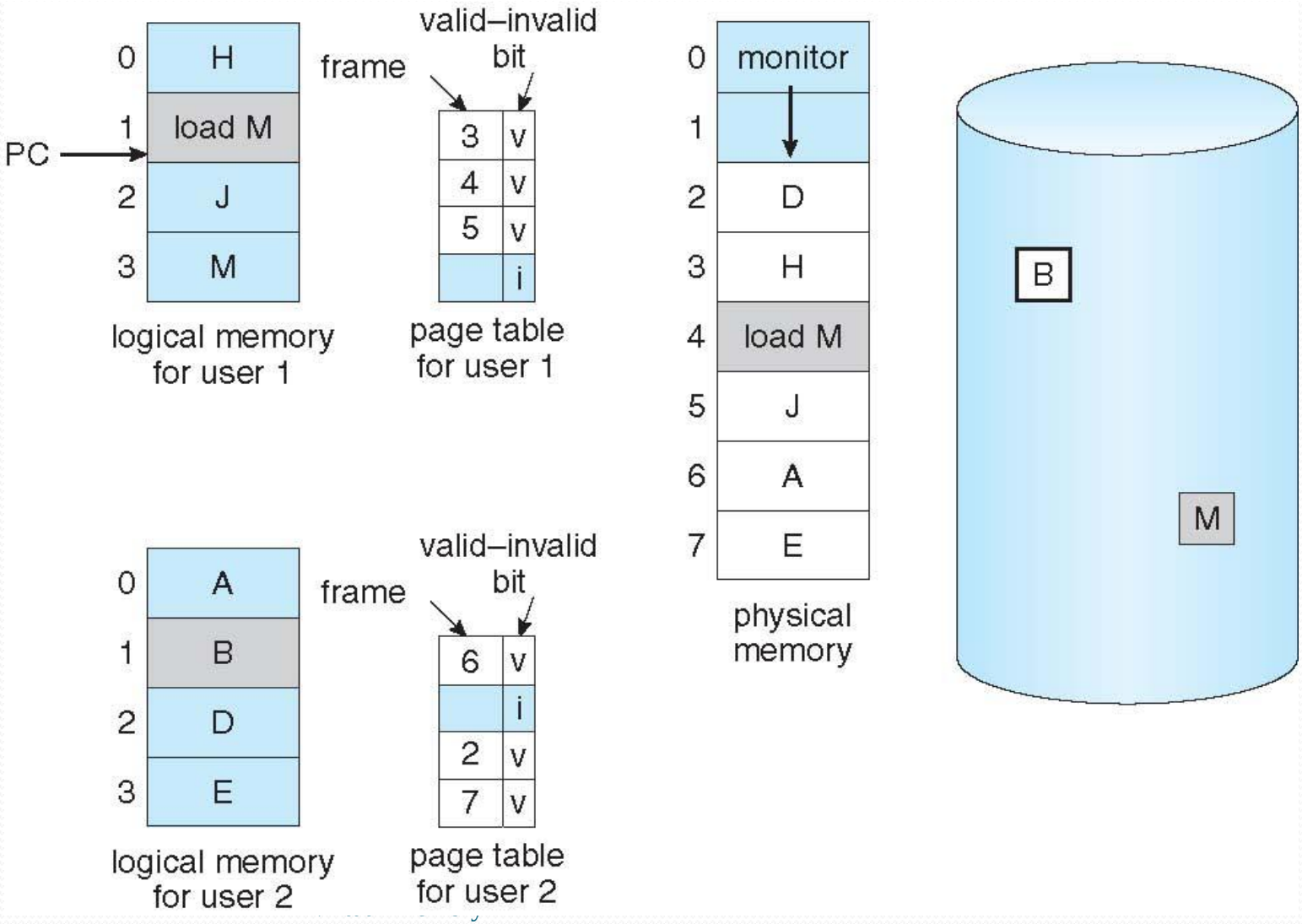
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame (all memory in use), use a page replacement (find one that is not currently being used and free it) algorithm to select a **victim** frame
   - Swap out victim frame to disk. Change the page and frame tables accordingly.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process

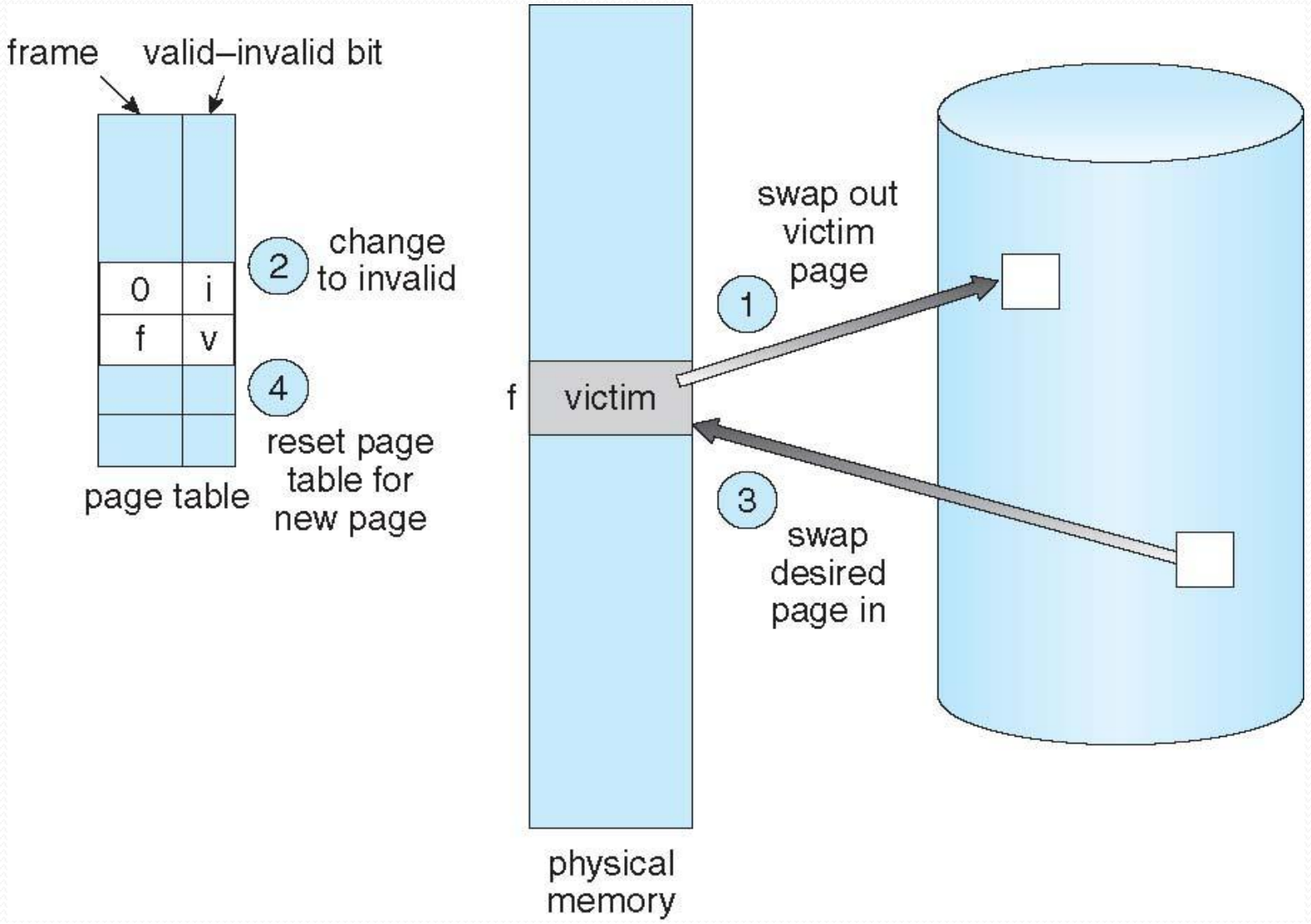
Need For Page Replacement



Basic Page Replacement (cont.)

- Two page transfers are required by Page replacement. This doubles the page fault service time and increases EAT.
- If the page to be replaced has not been modified since it was loaded from disk, there is no need to write it back to the disk.
- Use ***modify (dirty)*** *bit* to reduce overhead of page transfers – used to indicate whether or not a page has been modified.
- When the page is first read into memory - **dirty bit = 0**.
- If any word or byte is written into (page has been modified)- **dirty bit = 1**.
- The dirty bit is checked when a page is selected for replacement:
  - If the bit is set- page has been modified, must write the page to disk.
  - If the bit is not set, need not write page to disk.

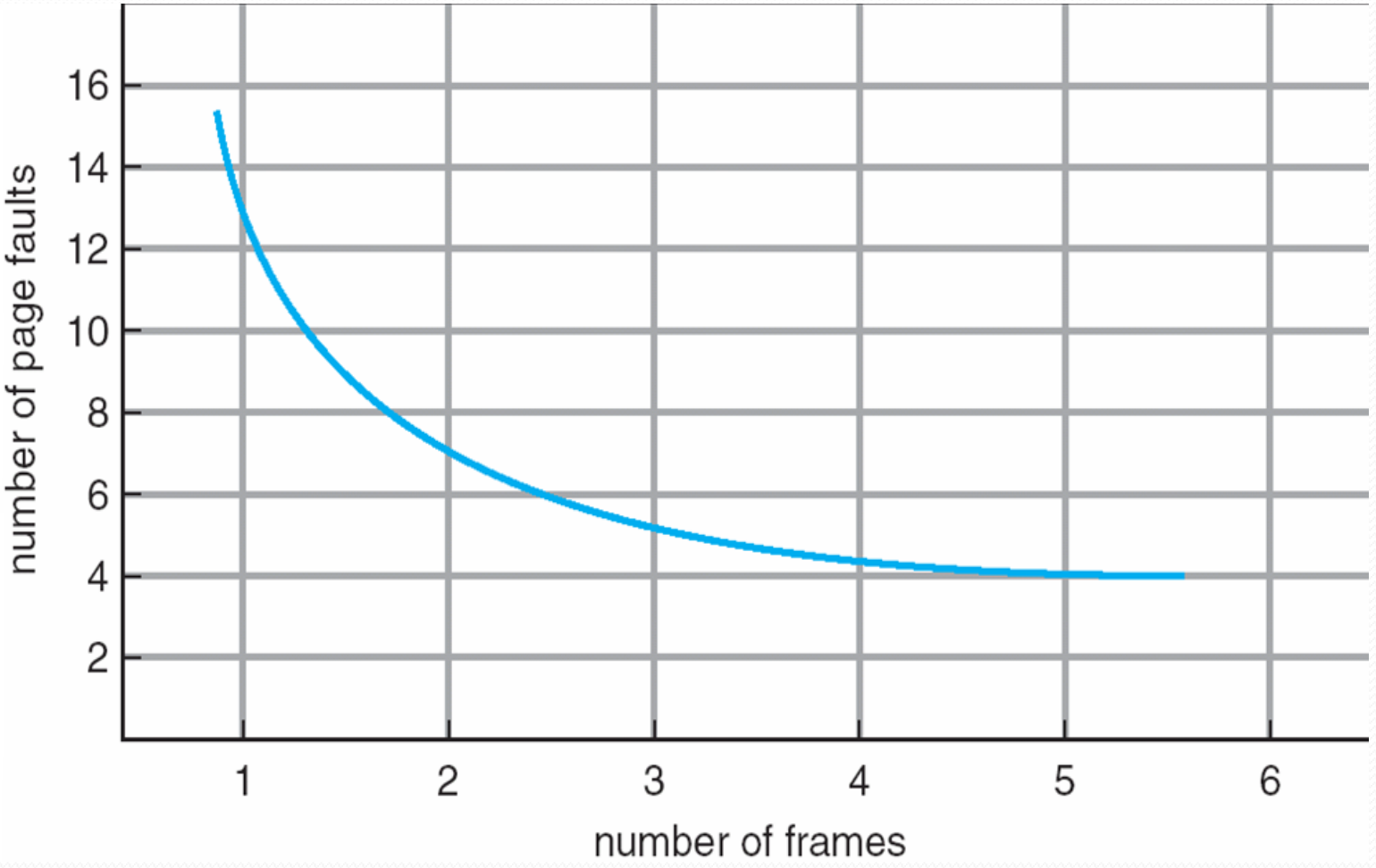
Page Replacement



Page-Replacement Algorithms

- They are many different Page-Replacement algorithms, but how do we select a particular replacement algorithms?
- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (*reference string*) and computing the number of page faults on that string.
- The more frames that are available, the fewer page faults there will be.

Graph of Page Faults Versus The Number of Frames



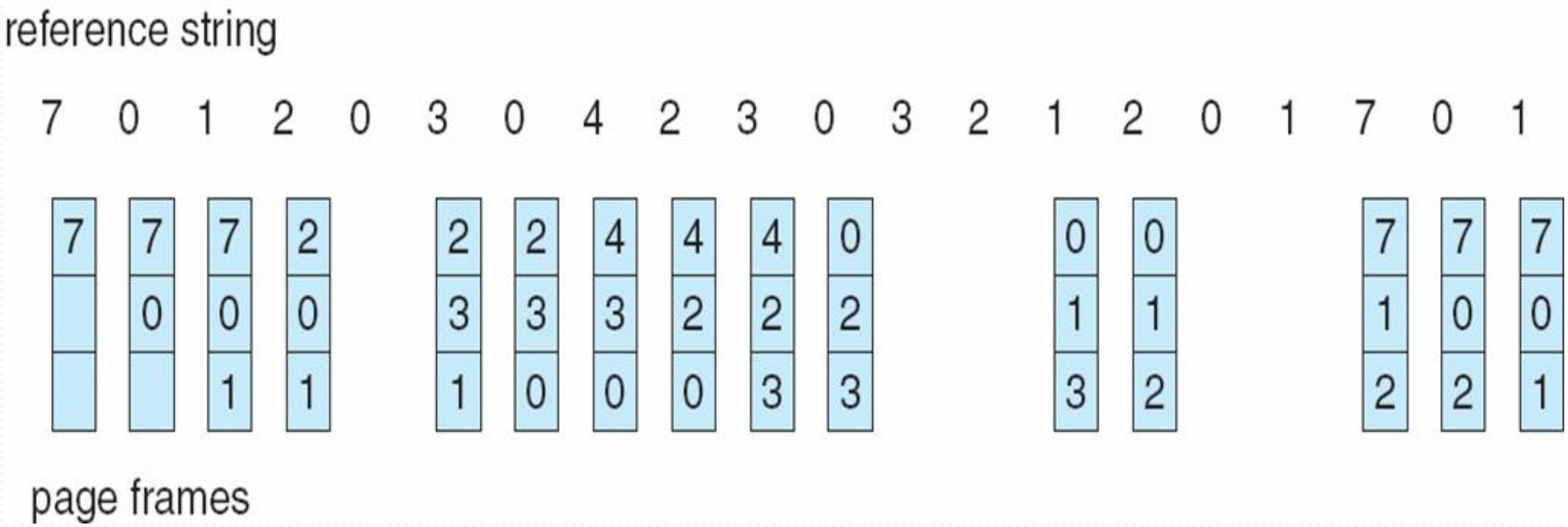
First-In-First-Out (FIFO) Algorithm

- Choose the oldest page in memory for replacement.
- **Example**: 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).

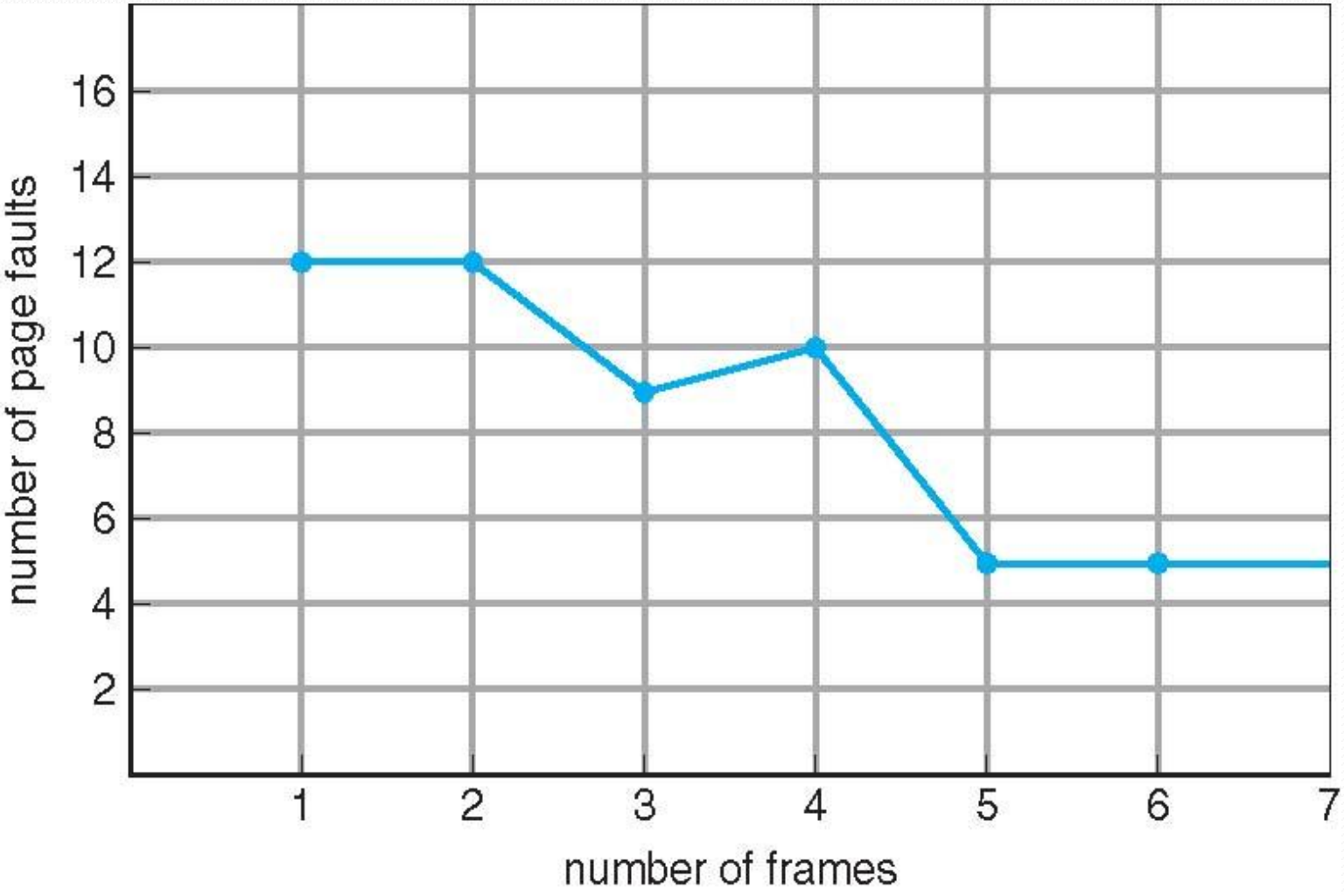
  9 page faults
- 4 frames.

  10 page faults
- Problem: Belady's Anomaly
  - more frames ⇒ less page faults

FIFO Page Replacement



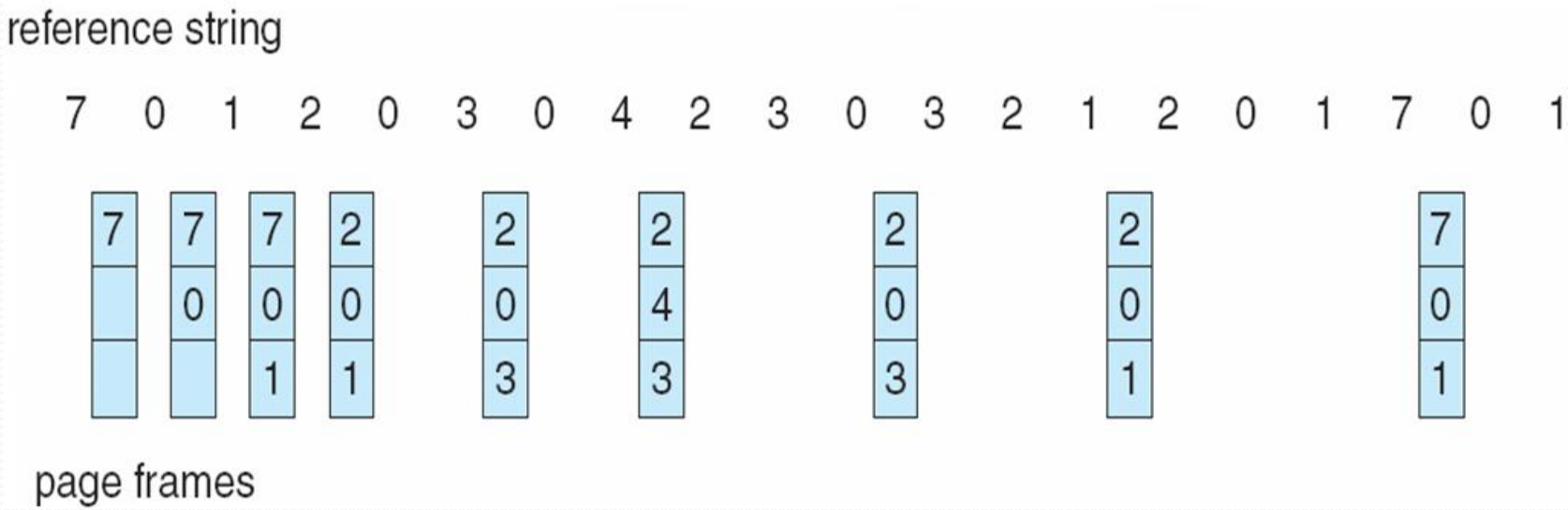
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time.
- Better than a FIFO algorithm, but difficult to implement .
- Used for measuring how well your algorithm performs.
- Difficult to implement, as it requires future knowledge of the reference string.

Optimal Page Replacement

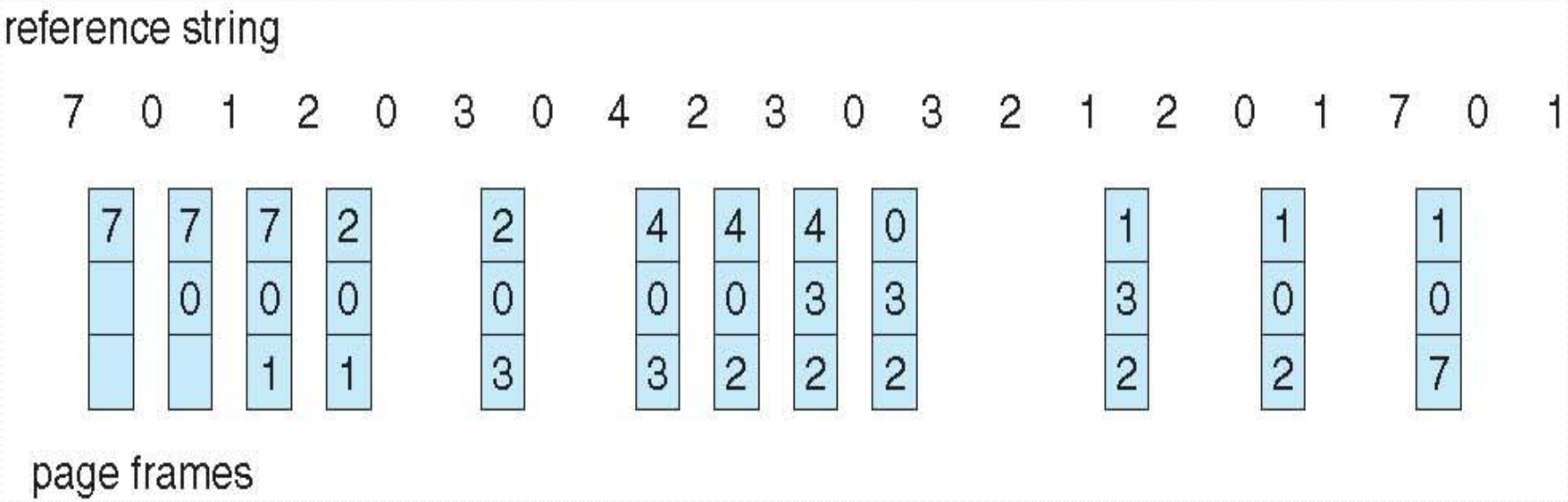


Least Recently Used (LRU) Algorithm

- Replace the page that has not been used for the longest period of time.
- The optimal page replacement algorithm looking backward in time, rather than forward.
- Considered a good replacement algorithm.
- Reference string:**7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.** Used in 3 frame memory space

  What is the total No of page faults?

LRU Page Replacement



Thrashing

- a process is busy swapping pages in and out than executing

Thrashing