Virtual Memory

Demand Paging

Page Replacement Algorithms

Thrashing

The Content is prepared with the help of existing text books mentioned below:

References:

1. Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 2009.
2. Stallings, William. *Operating Systems 5th Edition*. Pearson Education India, 2006.
3. Tannenbaum, Andrew S. "Modern Operating Systems, 2009."

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, allocation of page frames and thrashing.
- To discuss the principle of the working-set model

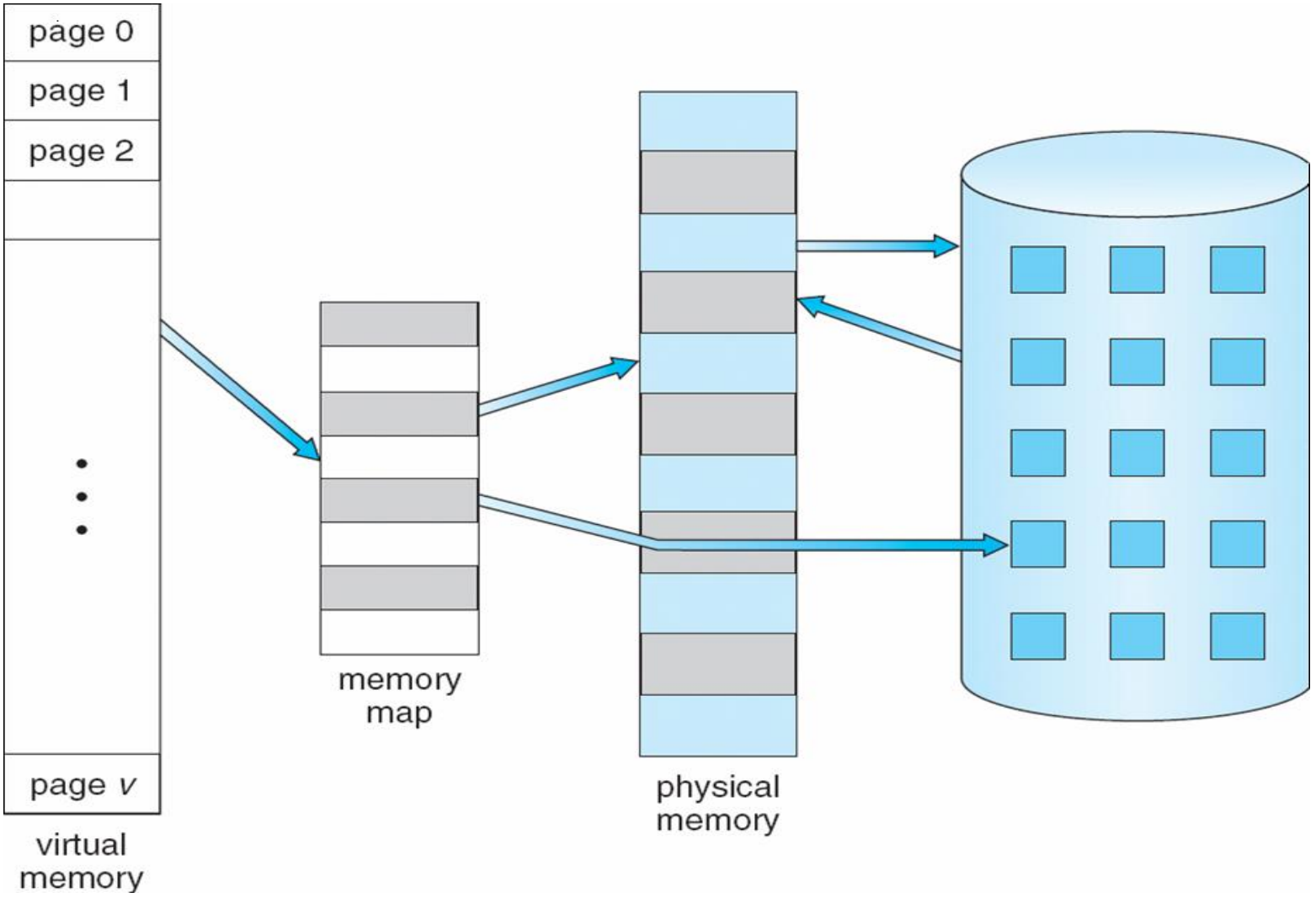
Background

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
  - **Arrays** are often over-sized
  - Certain features of **programs are rarely used.**
- The ability to load only the portions of processes that were actually needed has several benefits:
  - Programs could be written for a much **larger address space.**
  - **more memory left for other programs**, **improving CPU utilization and system throughput.**
  - **Less I/O is needed for swapping** processes in and out of RAM, speeding things up.

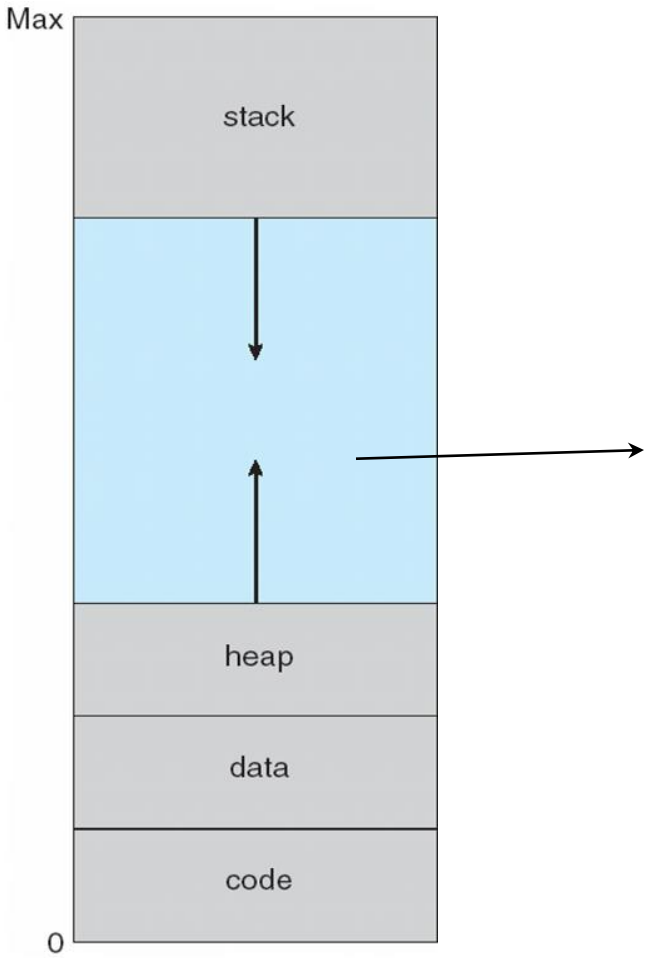
Background

- **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
  - Allows **address spaces to be shared** by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



Virtual-address Space

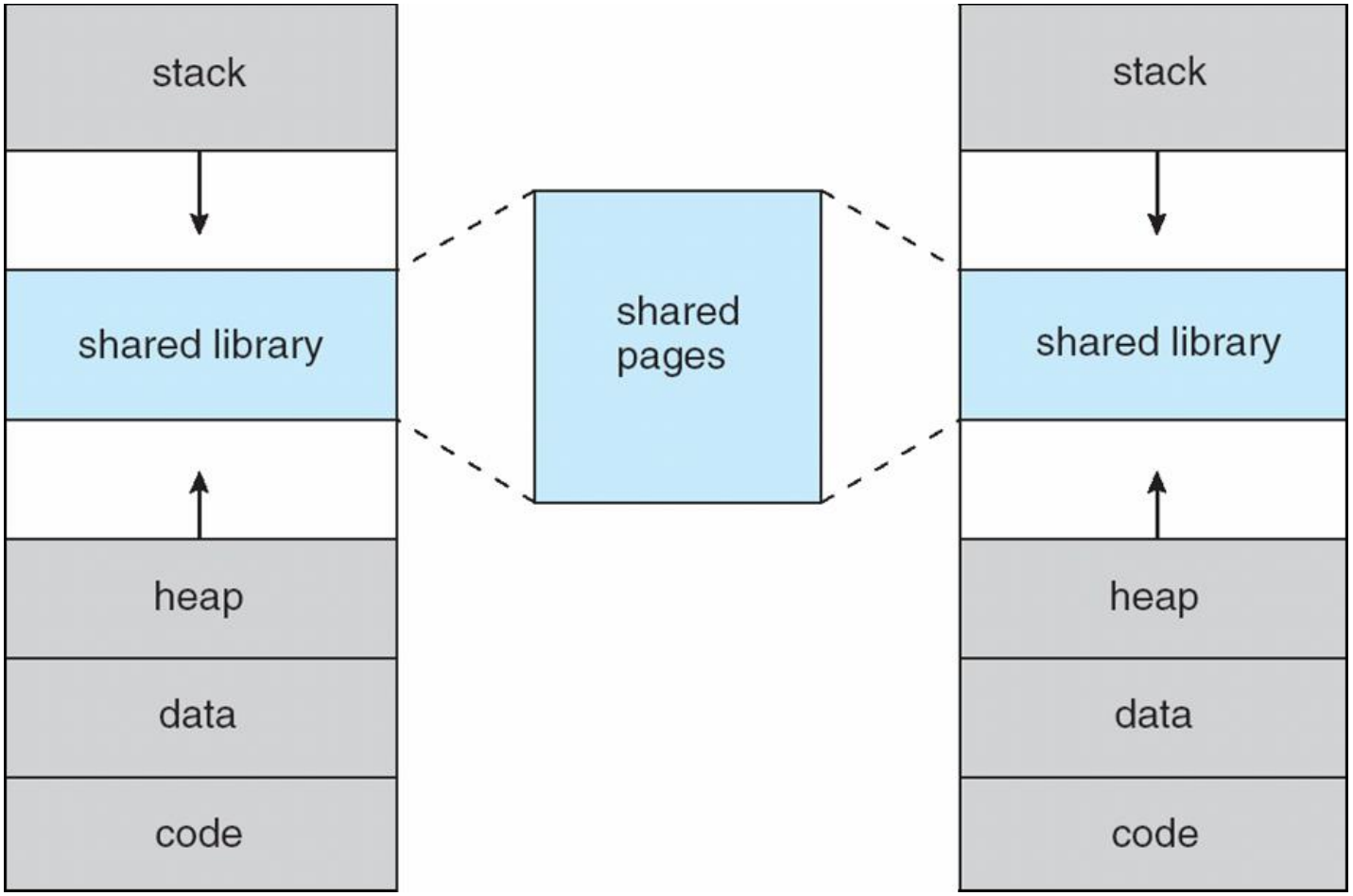


Note that the address space shown in Figure is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

Shared Library Using Virtual Memory

- Virtual memory also allows the ***sharing of files*** and **memory** by multiple processes, with several benefits:
  - System **libraries** can be **shared** by **more** than one **process**.
  - Processes can also **share virtual memory** by mapping the **same** block of **memory** to **mor**e than one **process**.
  - **Process pages** can be **shared** during a **fork( )** system call, eliminating the need to copy all of the pages of the original (parent) process.

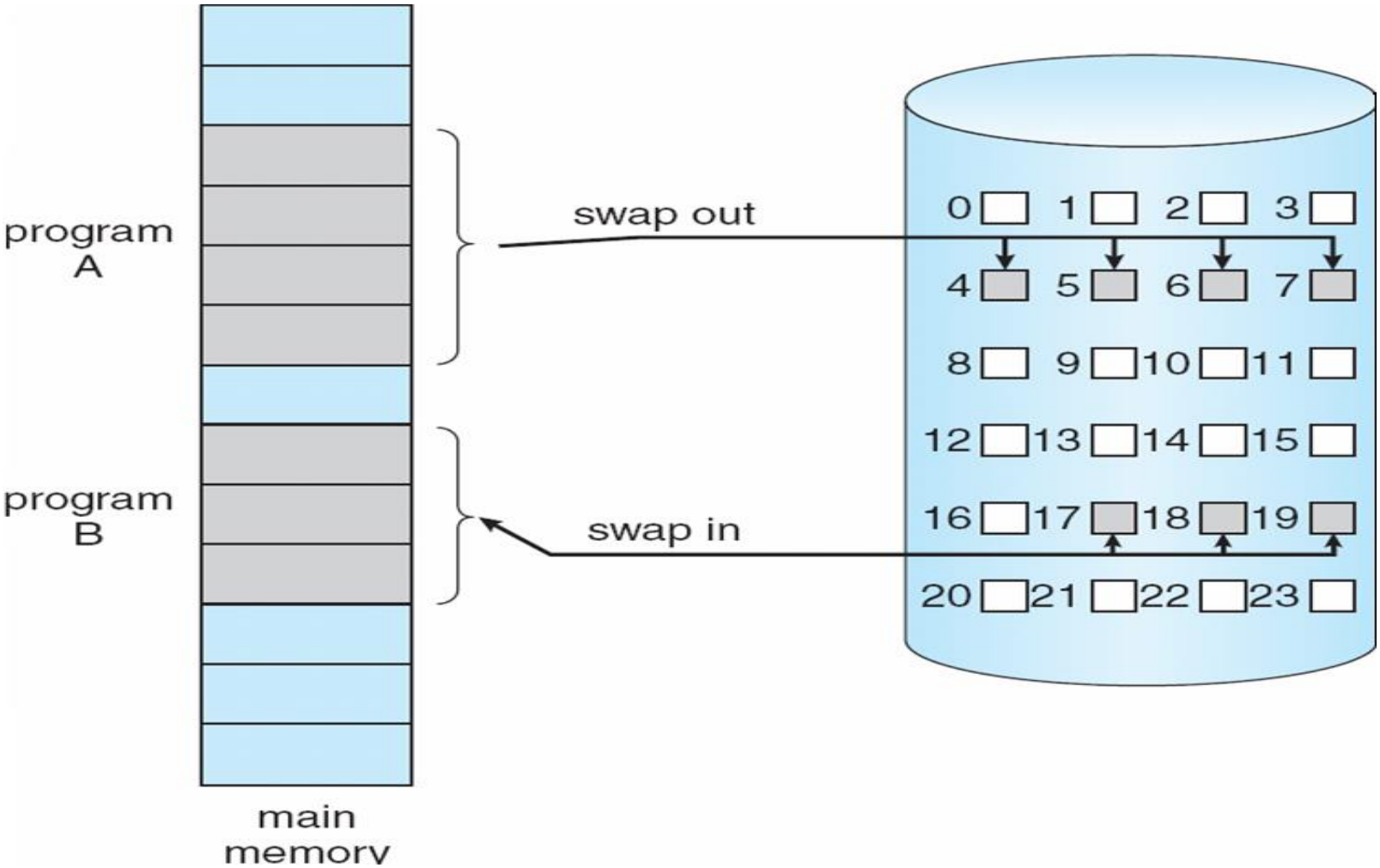
Shared Library Using Virtual Memory



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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

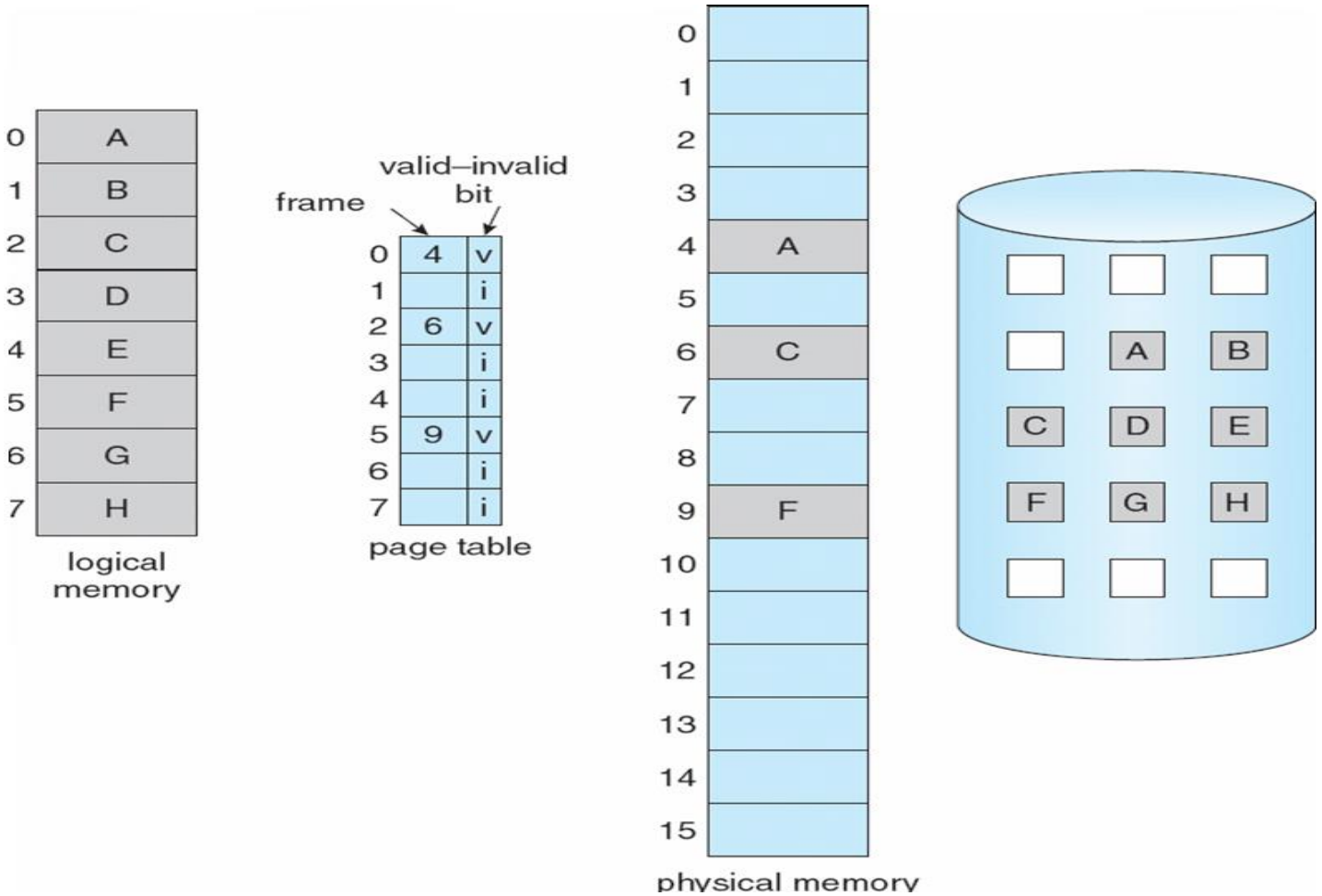
Transfer of a Paged Memory to Contiguous Disk Space



Demand Paging Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory **those pages** that it expects the process to need (**right away**)
- Pages that are **not loaded into memory** are marked as **invalid** in the page table, using the invalid bit.
- If the process only ever accesses ***memory resident*** pages, then the process runs exactly as **if all the pages were loaded in to memory**.

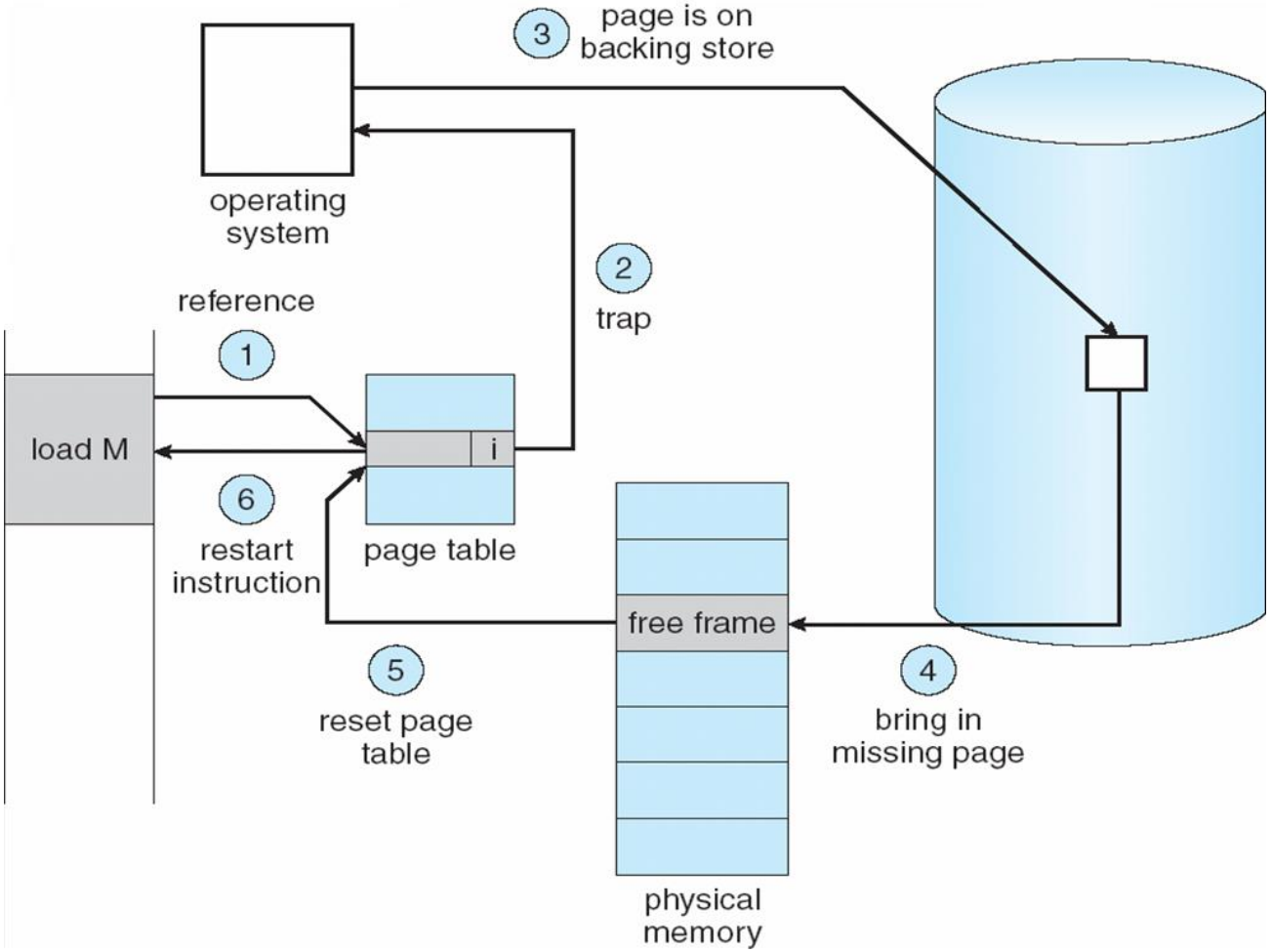
Page Table When Some Pages Are Not in Main Memory



What is Page Fault?

The memory address requested is first checked, to make sure it was a valid memory request . If the page is not in memory (invalid bit is set)than it results in page fault. Also If there is a reference to a page, first reference to that page will trap to operating system causing **page fault**

Steps in Handling a Page Fault



A page fault causes the following sequence to occur:

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced.
    b) Wait for the device seek and/ or latency time.
    c) Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/0 subsystem (I/0 completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

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) for each reference

  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!!

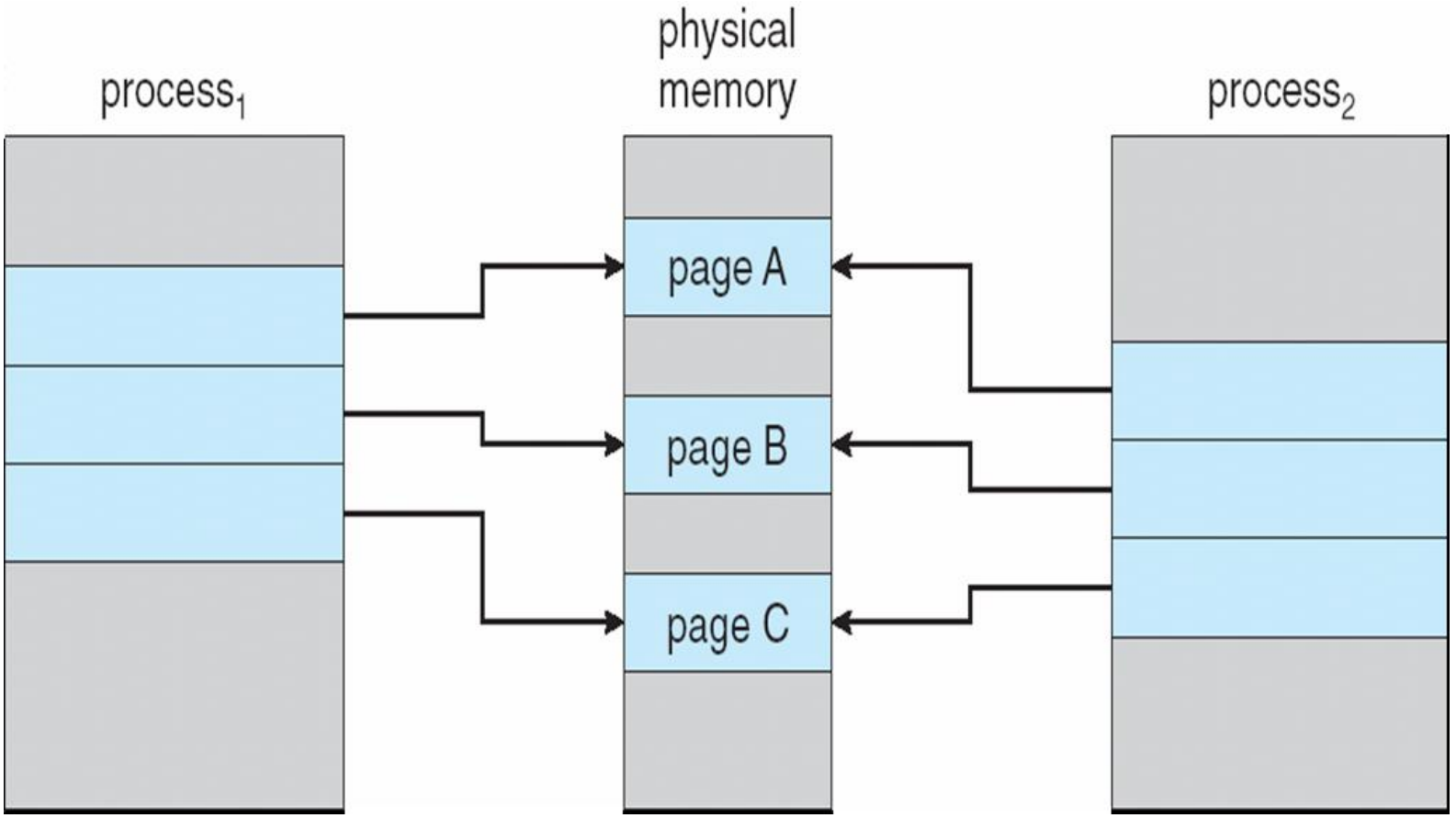
Copy-on-Write

- 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

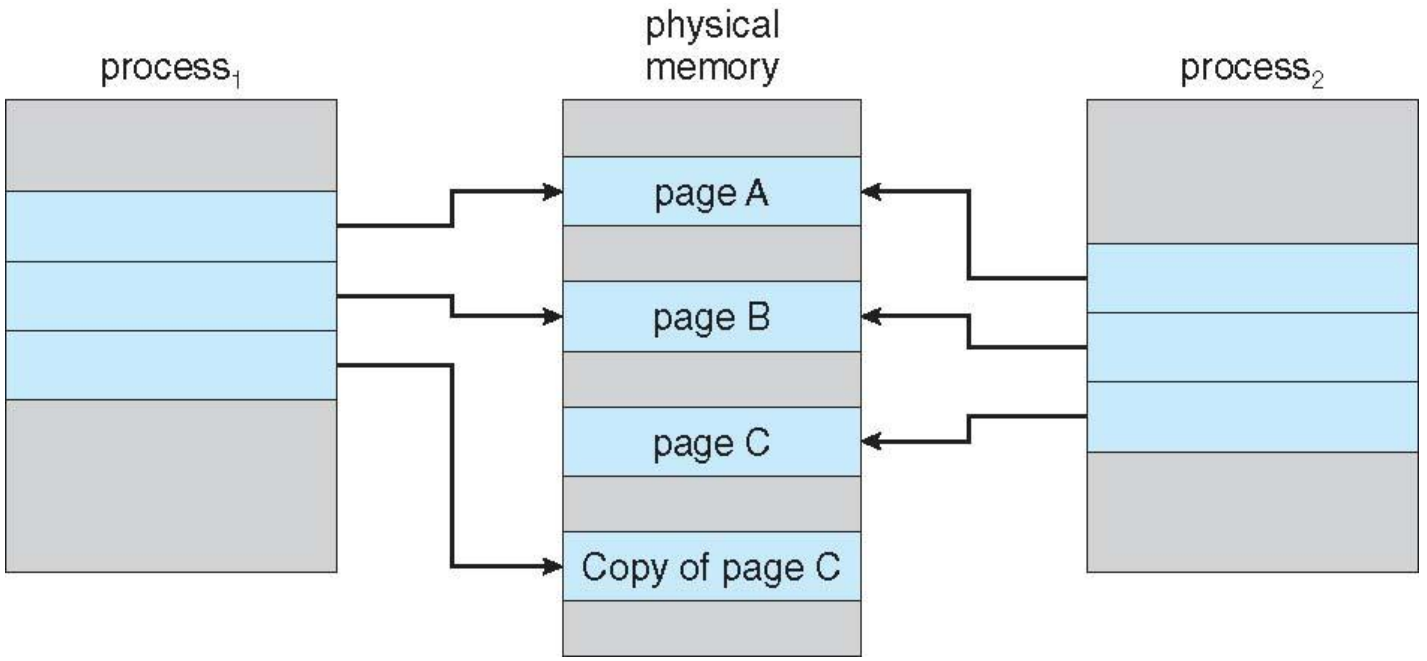
- COW allows more efficient process creation as only modified pages are copied

- **Free pages** are **allocated** from a **pool** of *zeroed-out pages*

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



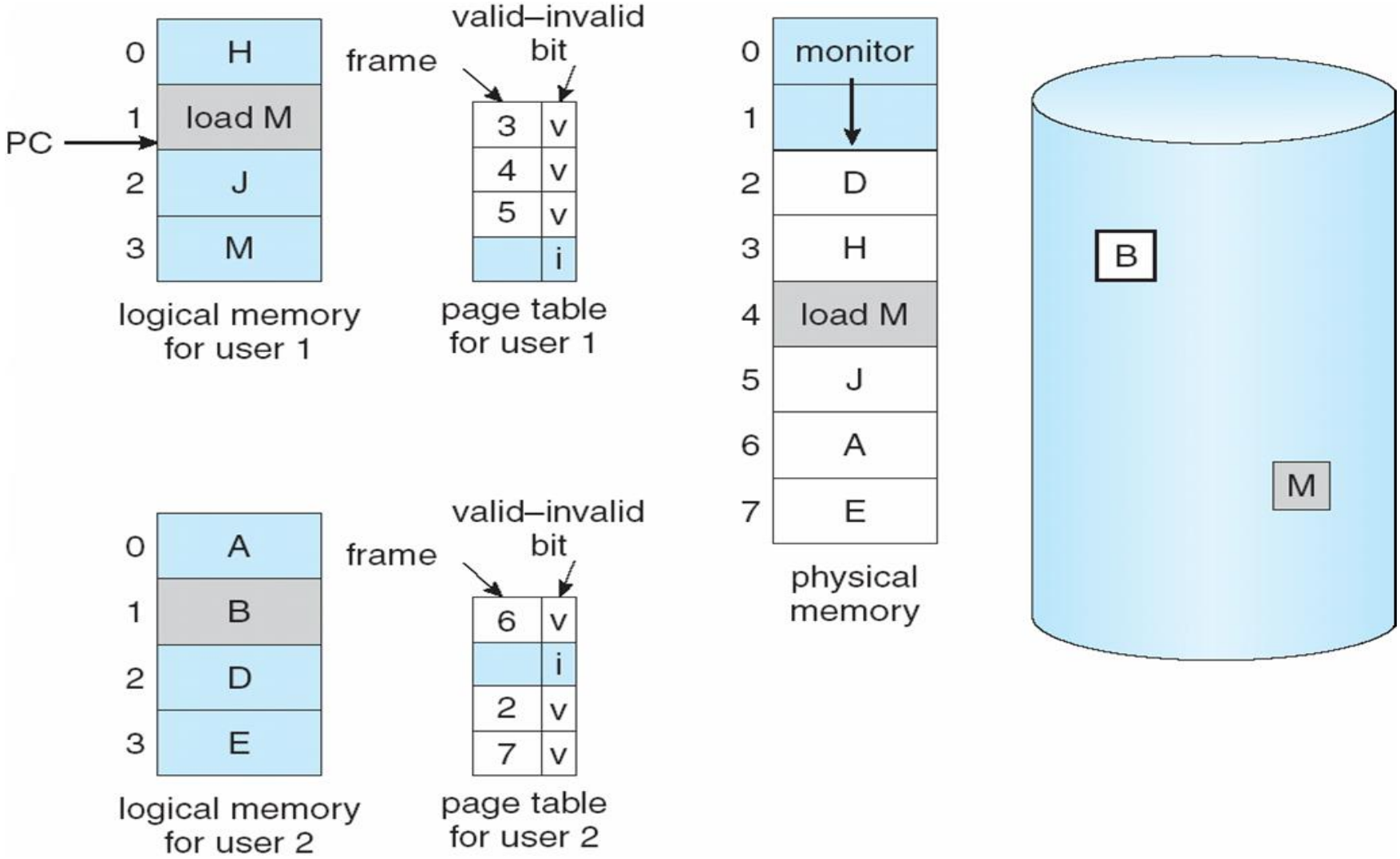
What happens if there is no free frame?

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

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

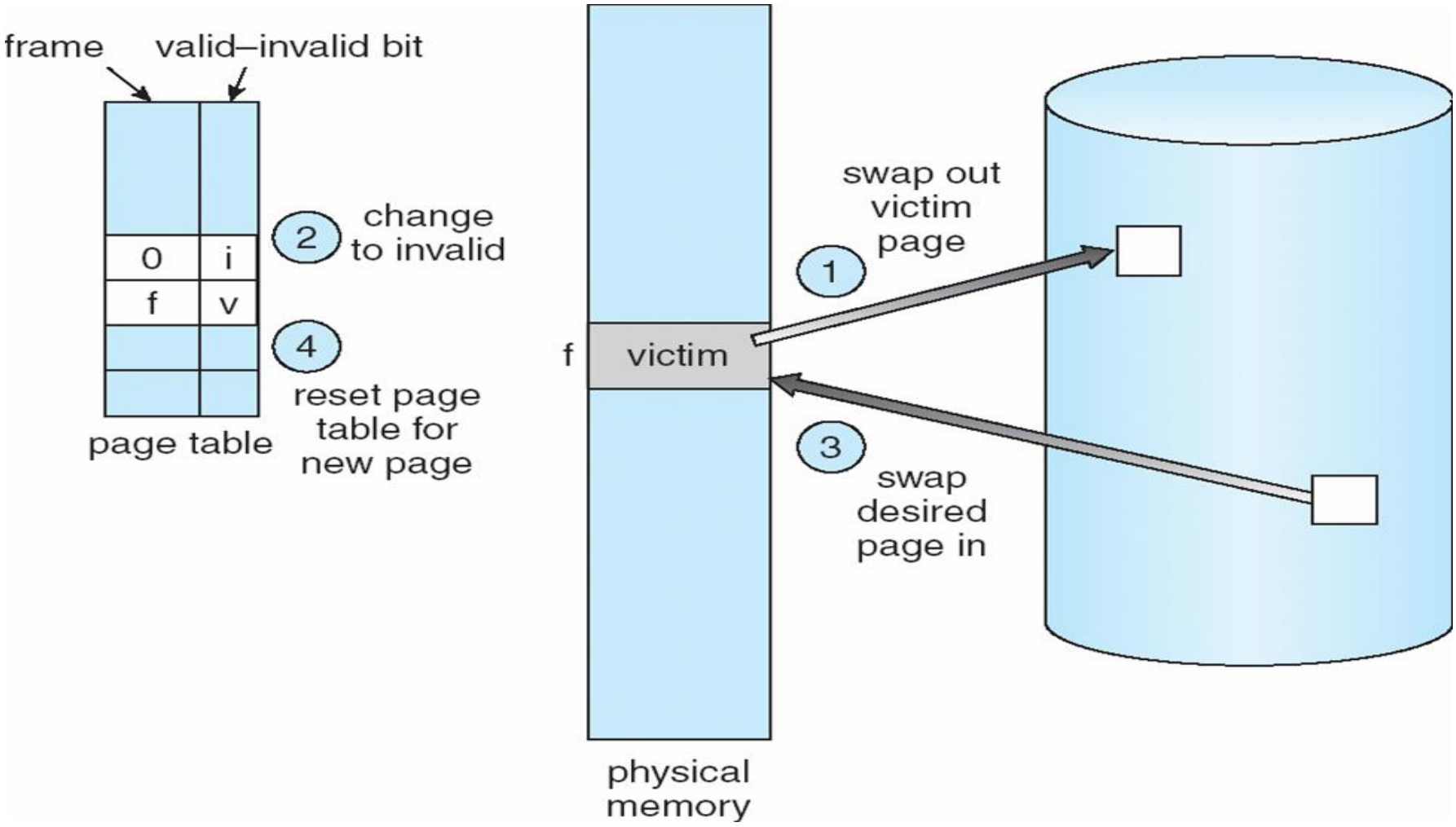
Need For Page Replacement



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
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Page Replacement



Page Replacement Algorithms

- There are two major requirements to implement a successful demand paging system. We must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm.***

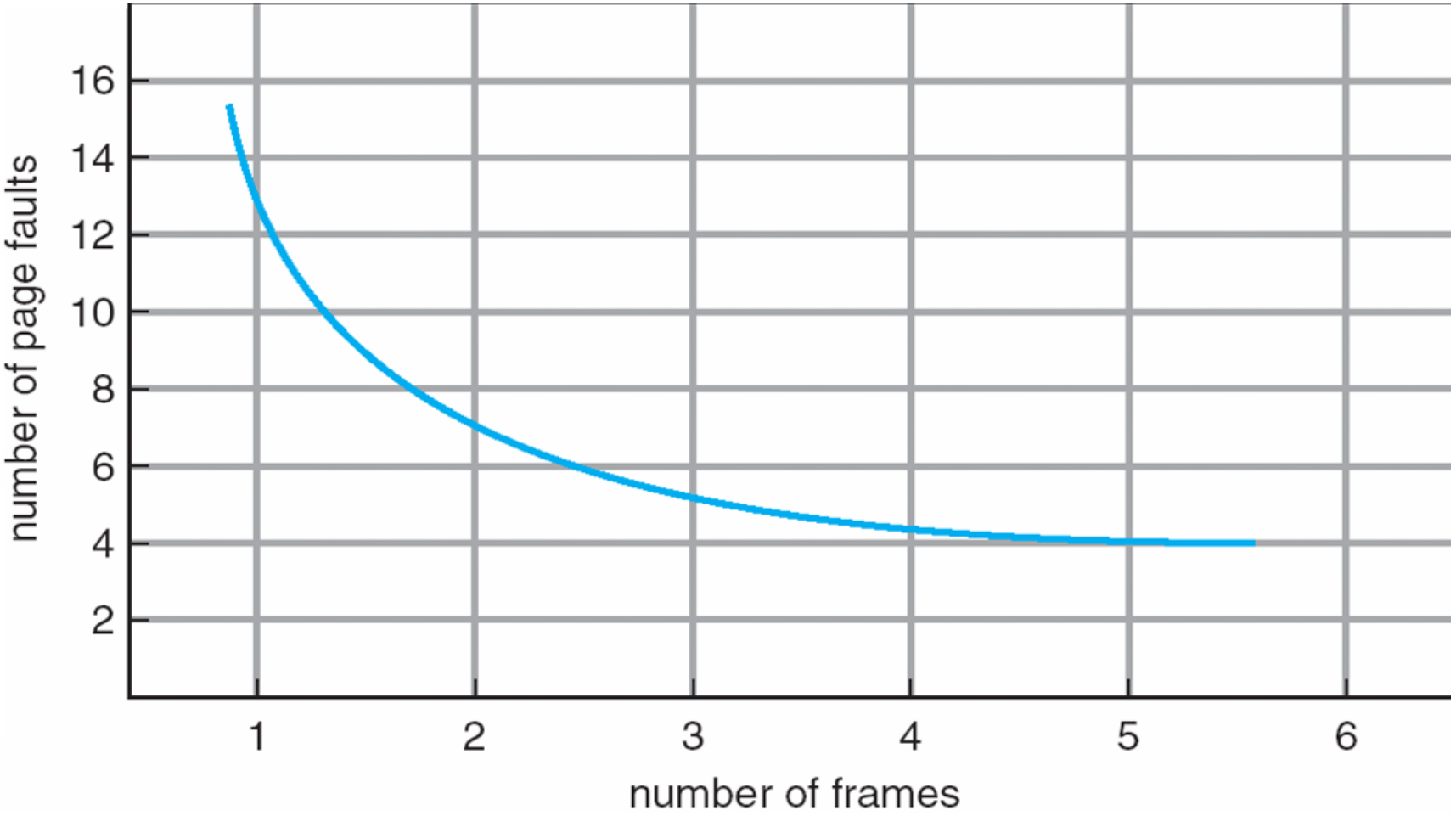
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

- Reference string can be generated as shown below.

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

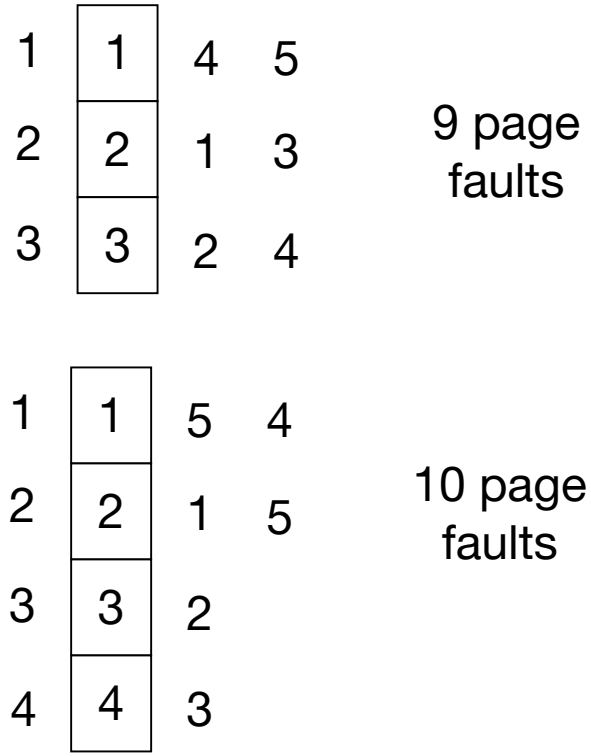
- So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4)

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

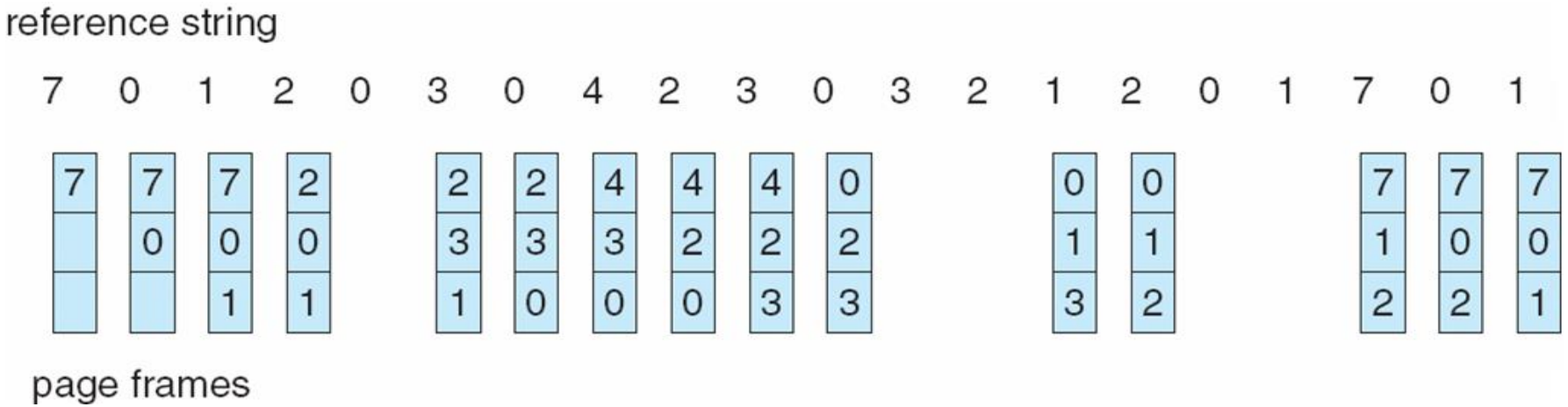
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



- 4 frames

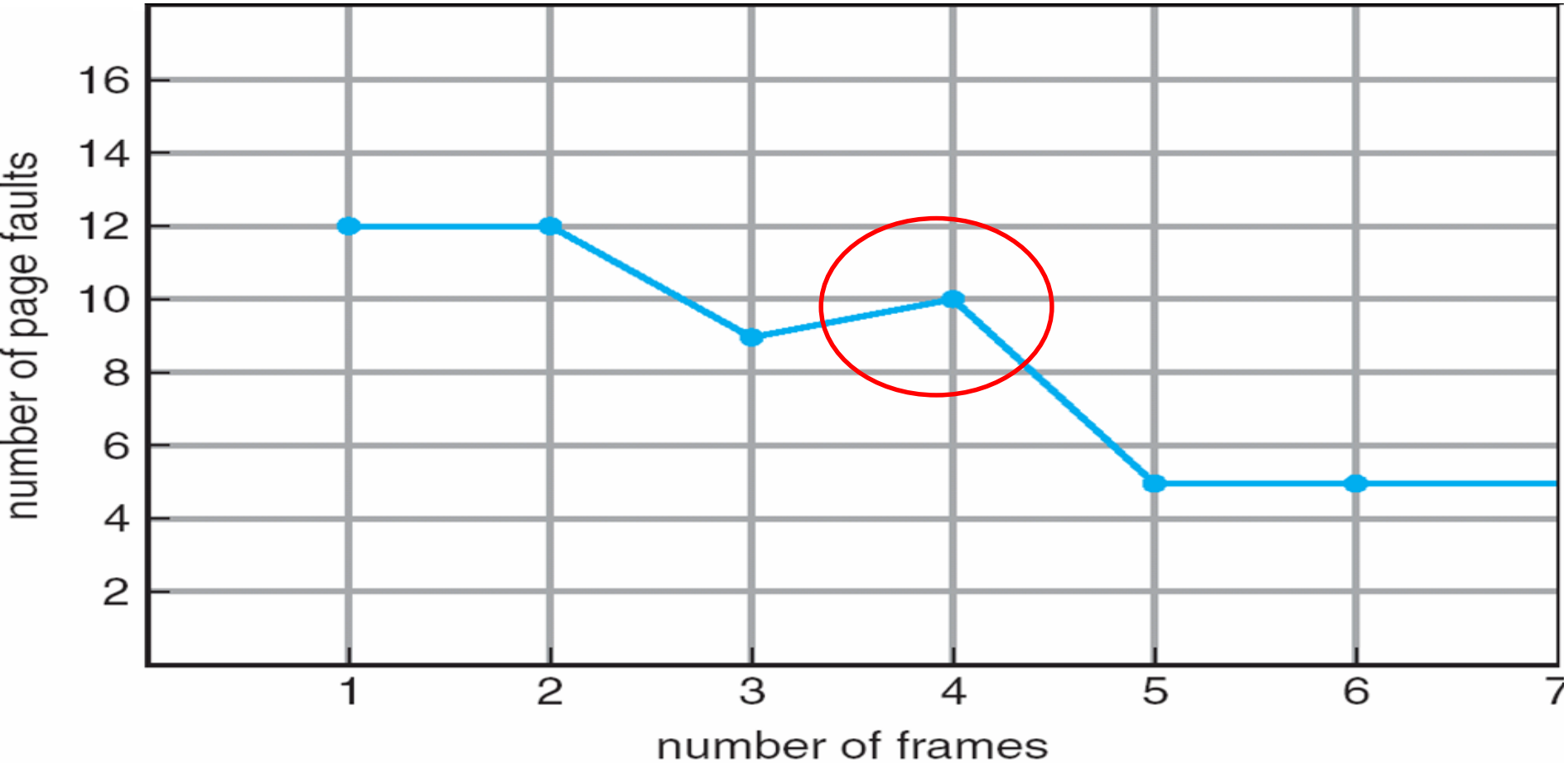
- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



In the absence of ANY hardware support, FIFO might be the best available choice.

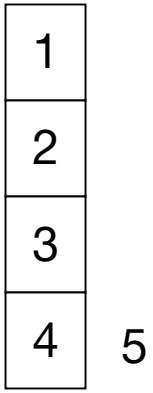
FIFO Illustrating Belady's Anomaly



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



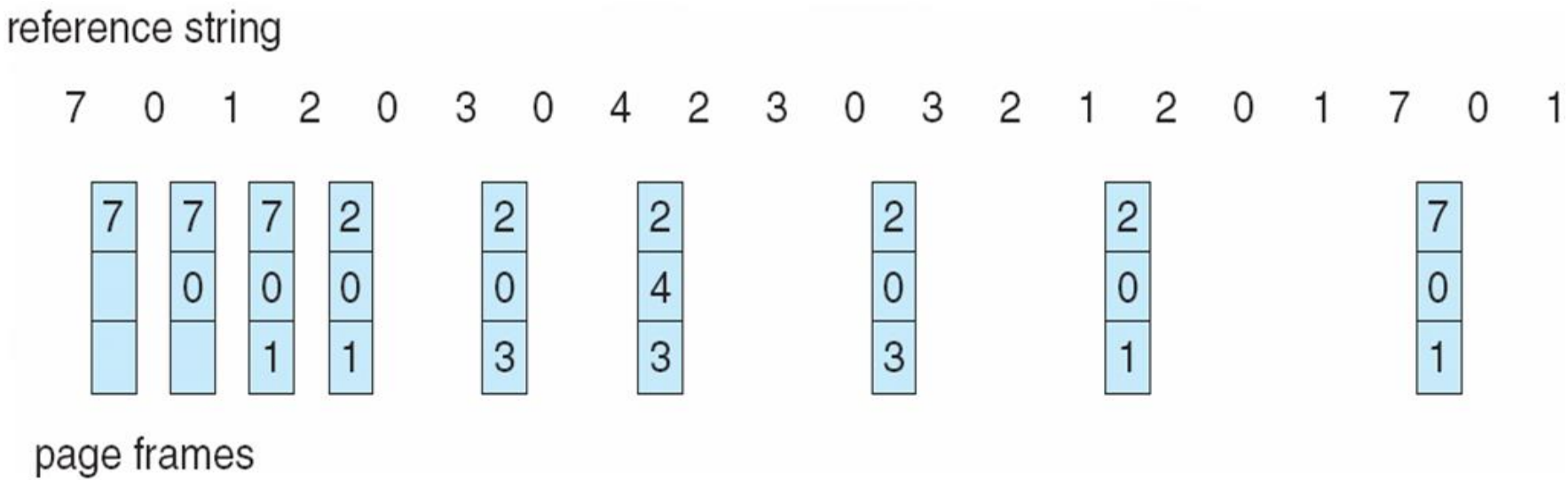
4

5

6 page faults

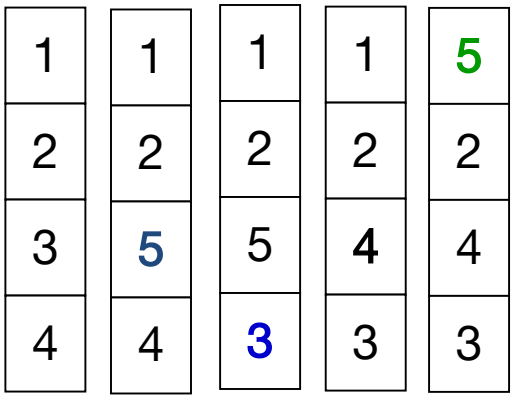
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



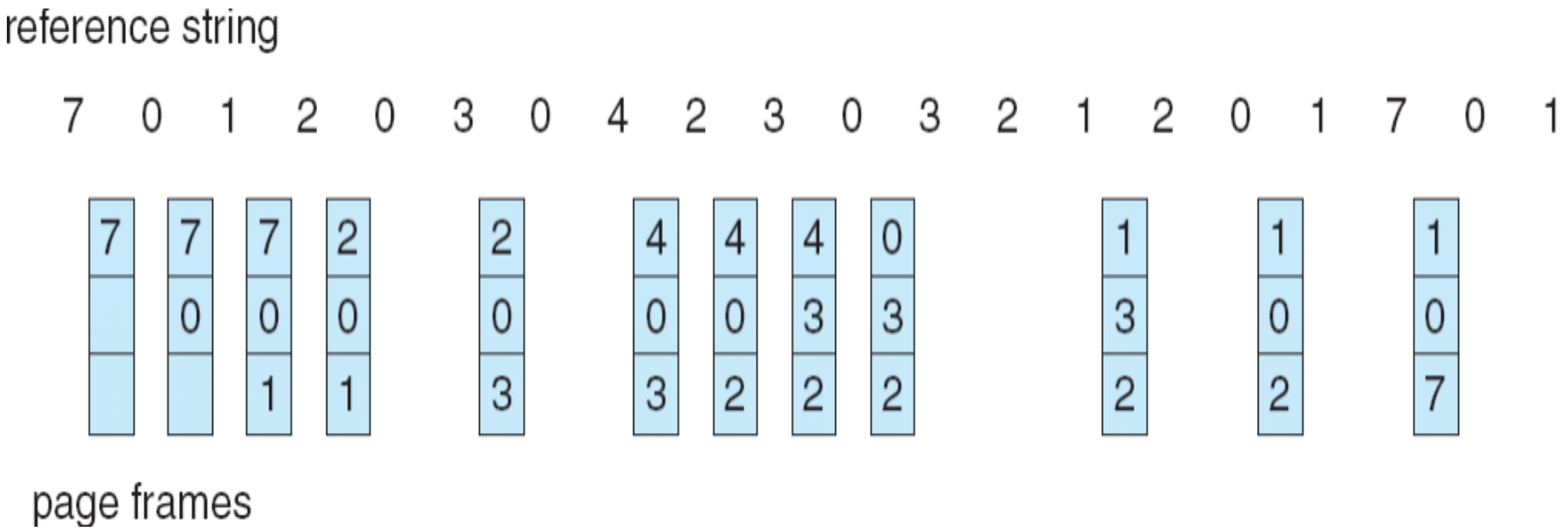
Least Recently Used (LRU) Algorithm

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

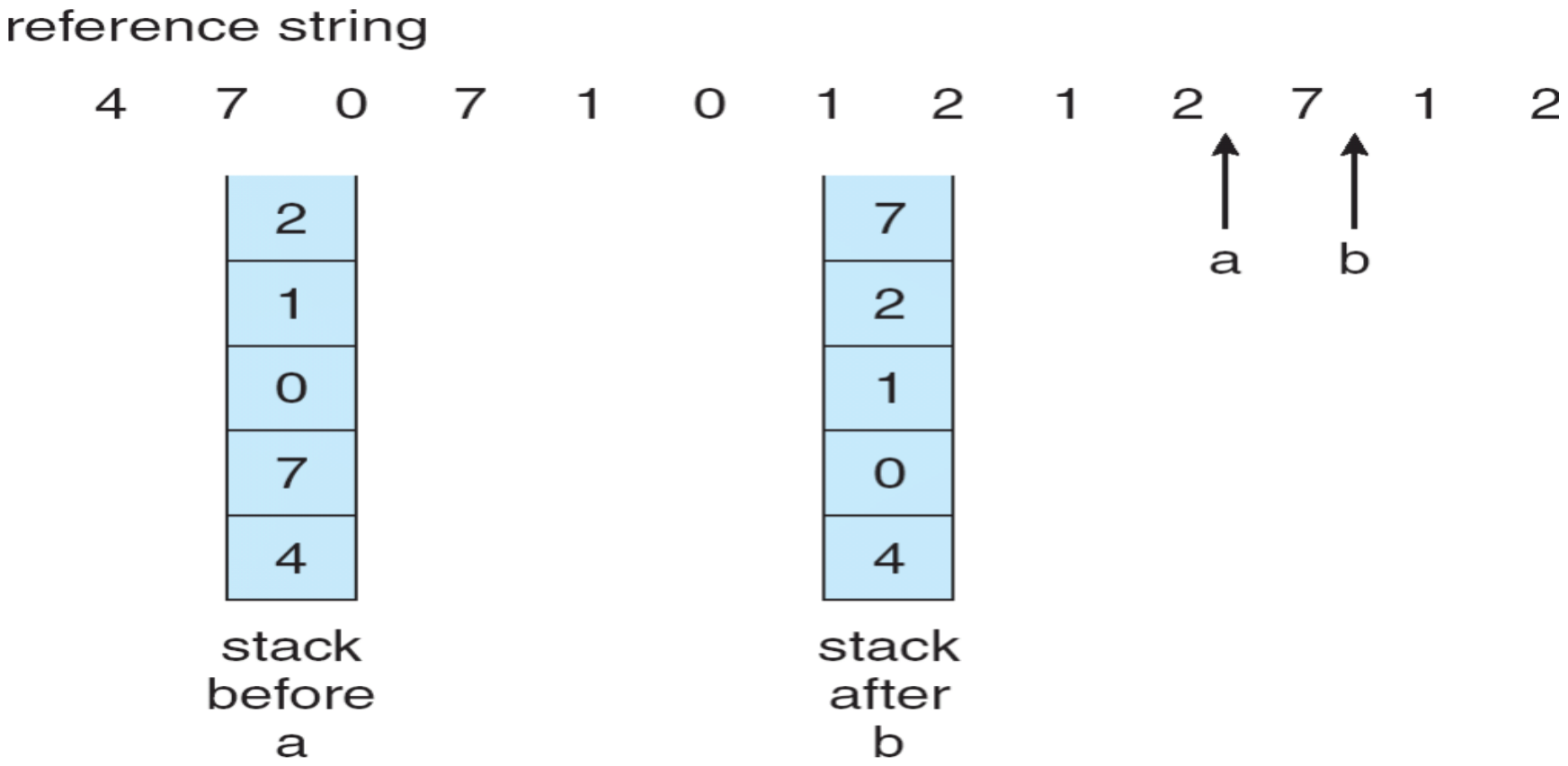


- Resent use can be implemented via Counter or Stack.
- Both **Counter** and **stack implementation** require **hardware support**, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.

LRU Page Replacement



Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time.

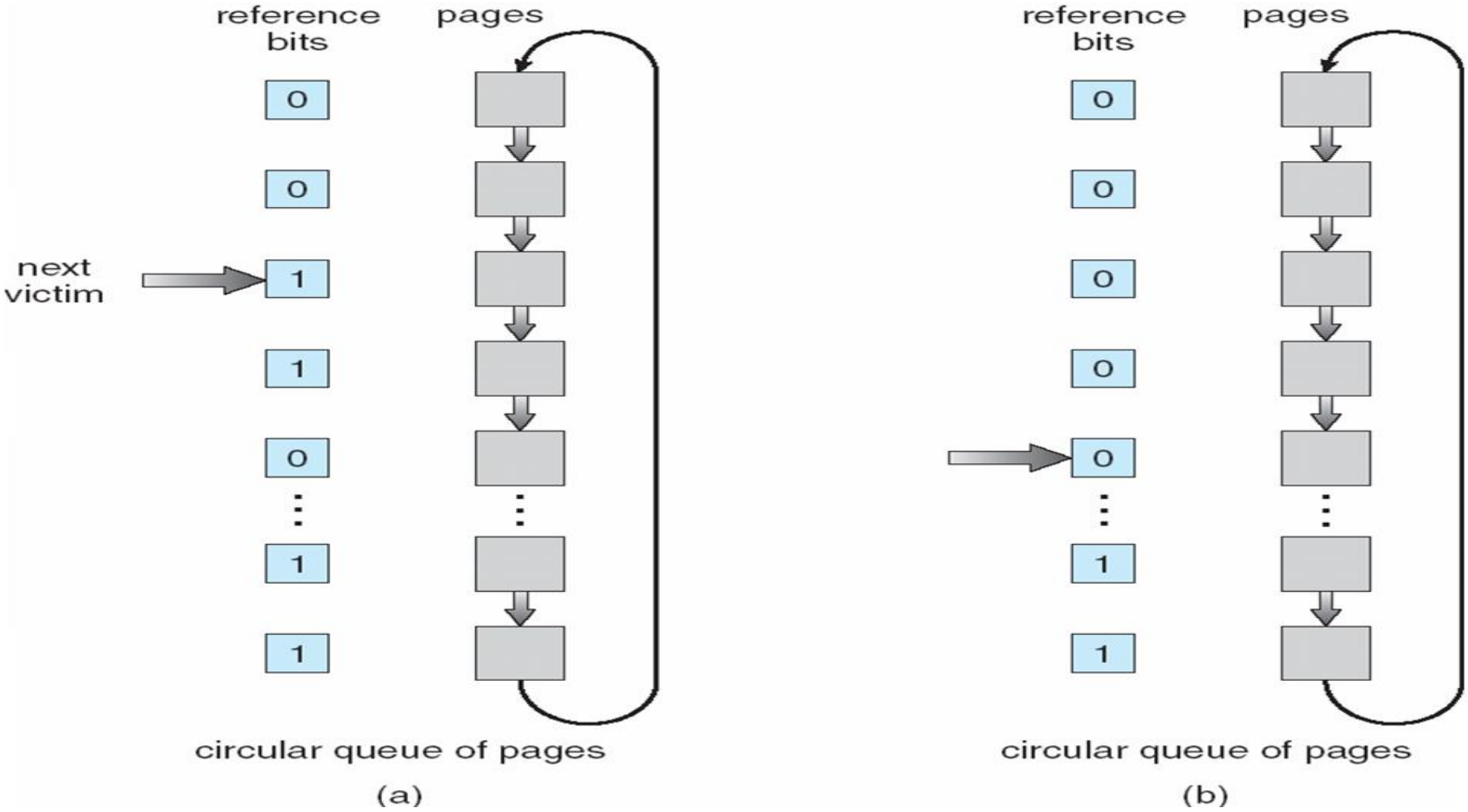
LRU Approximation : Second-Chance Algorithm

- The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
  - When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
  - If a page is found with its reference bit not set, then that page is selected as the next victim.
  - If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:

LRU Approximation : Second-Chance Algorithm

- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- This algorithm is also known as the ***clock*** algorithm, from the hands of the clock moving around the circular queue.

Second-Chance (clock) Page-Replacement Algorithm



LRU Approximation : Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit), and classifies pages into one of four classes:
    - (0, 0) - Neither recently used nor modified.
    - (0, 1) - Not recently used, but modified.
    - (1, 0) - Recently used, but clean.
    - (1, 1) - Recently used and modified.
- This algorithm searches for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.

Counting Algorithms

- There are several algorithms based on counting the number of references that have been made to a given page, such as:

  - ***Least Frequently Used, LFU:*** Replace the page with the lowest reference count.
  - ***Most Frequently Used, MFU:*** Replace the page with the highest reference count.

Allocation of Frames

- Each process needs *minimum* number of pages
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

-

- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

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

- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

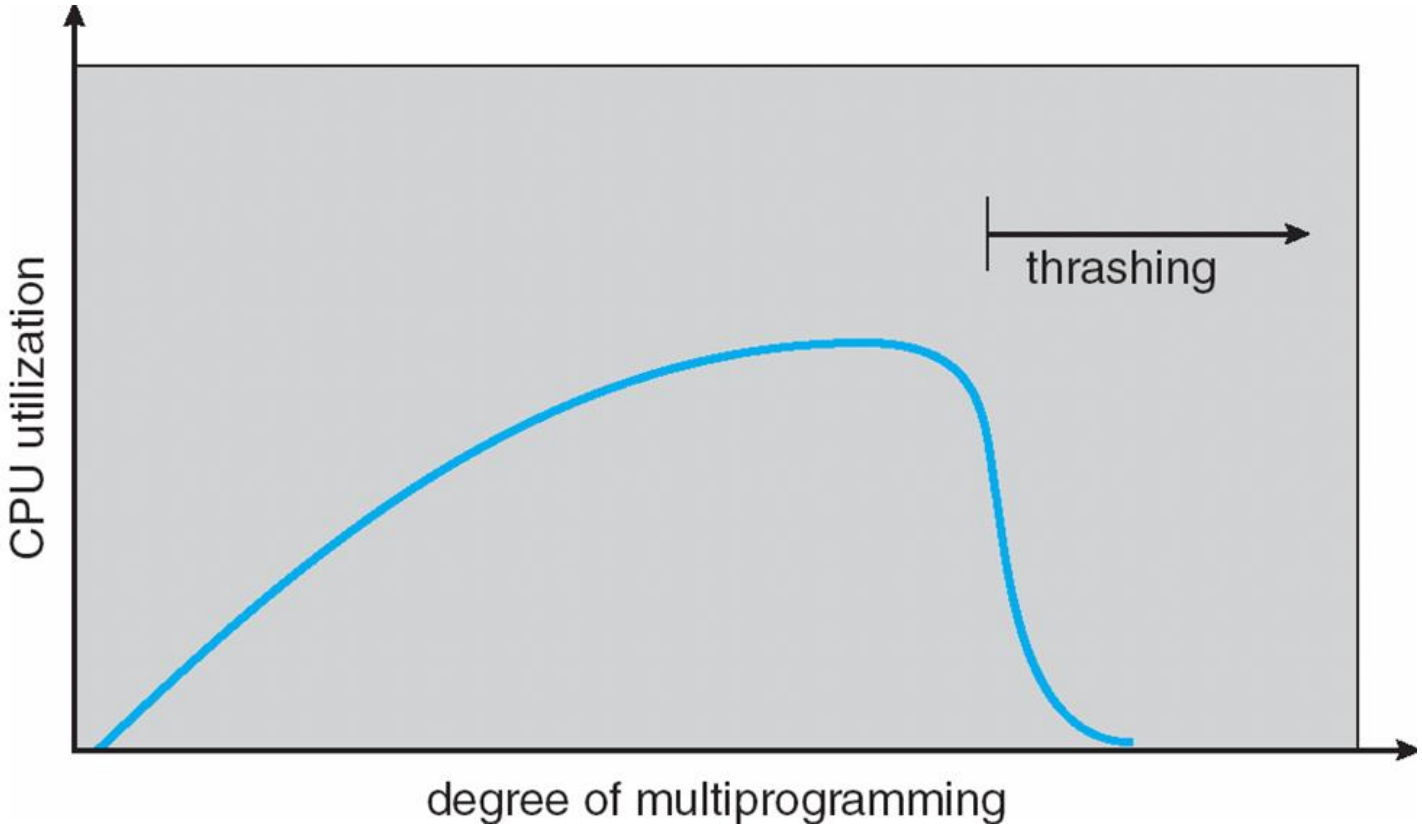
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** ≡ If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

Cause of Thrashing

- The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming.
- Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.
- These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.
- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
- As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time in paging.



Solution to Thrashing

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm). With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. However, the problem is not entirely solved.

Solution to Thrashing

Provide a process with as many frames as it needs. But how do we know how many frames it “needs”?

Starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

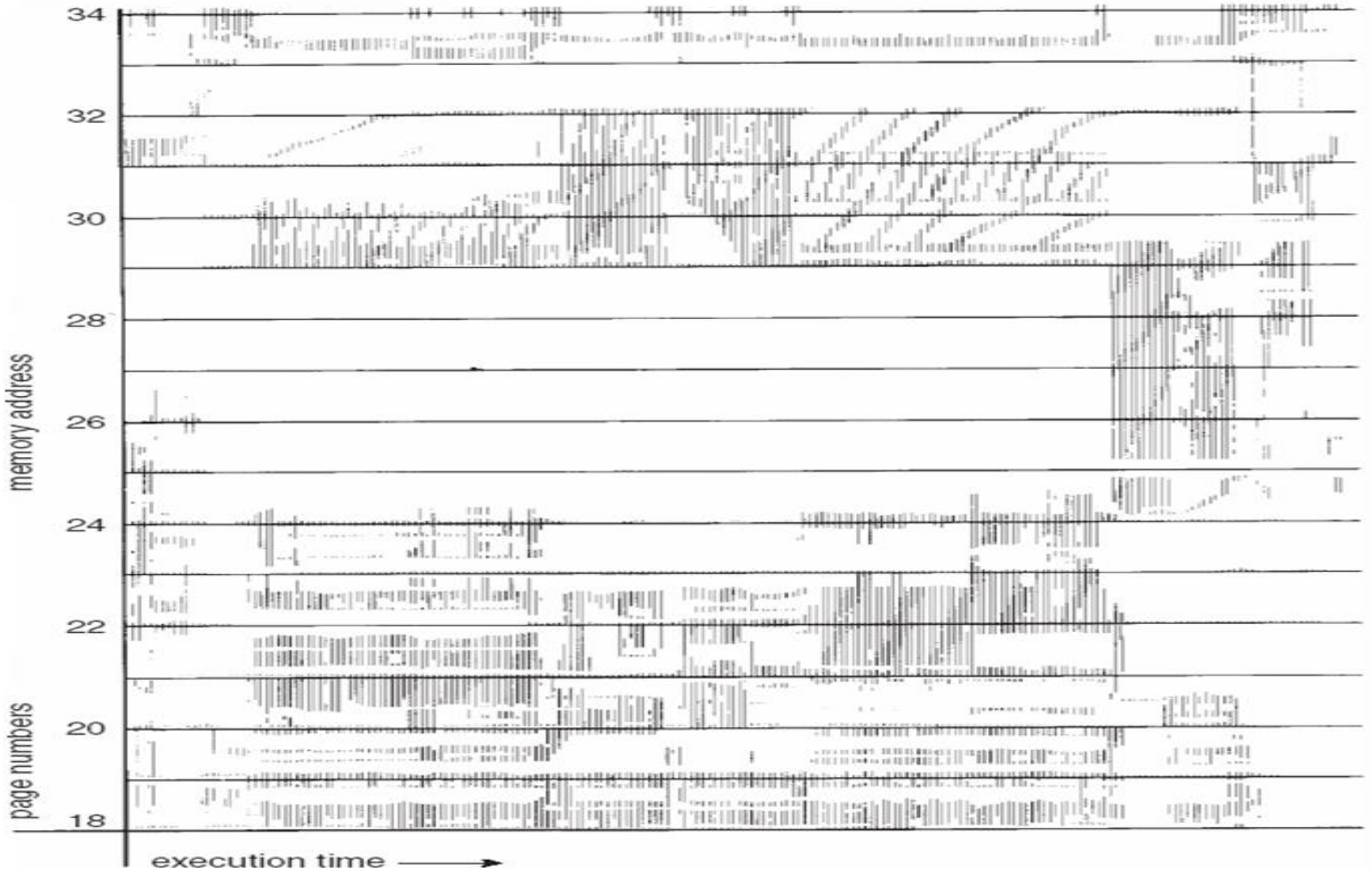
The locality model states that, as a process executes, it moves from locality to locality. For example, when a function is called, it defines a new locality with its local and global variables.

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  Σ size of locality > Total Memory Size M

Locality In A Memory-Reference Pattern

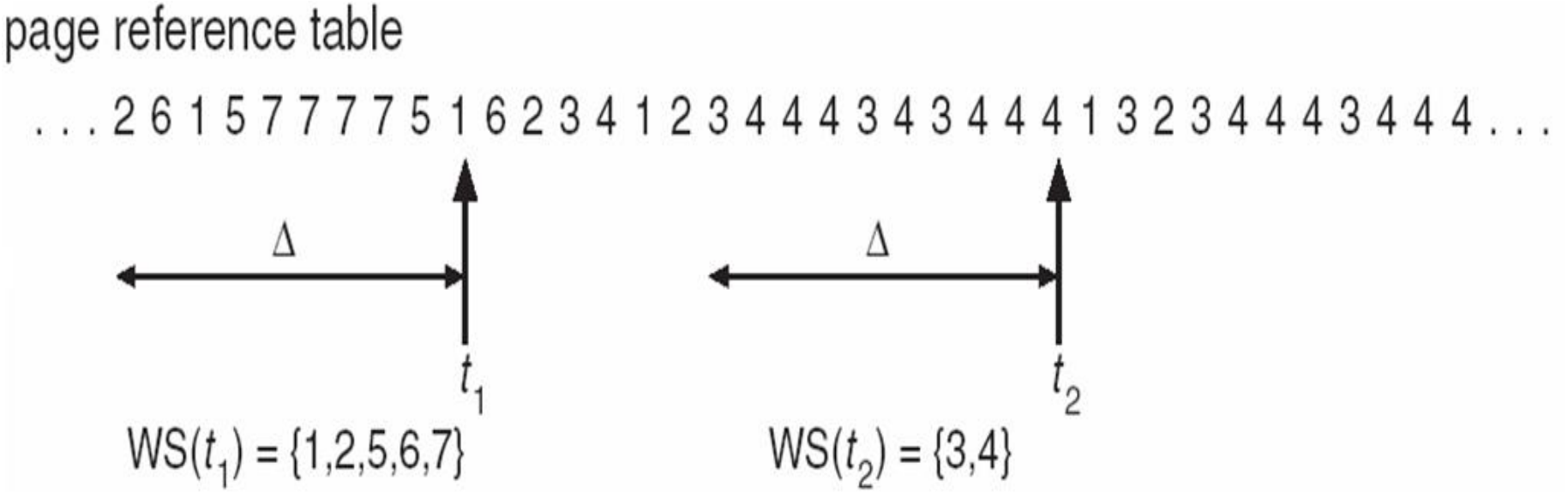


Working-Set Model

- Δ ≡ working-set window ≡ a fixed number of page references

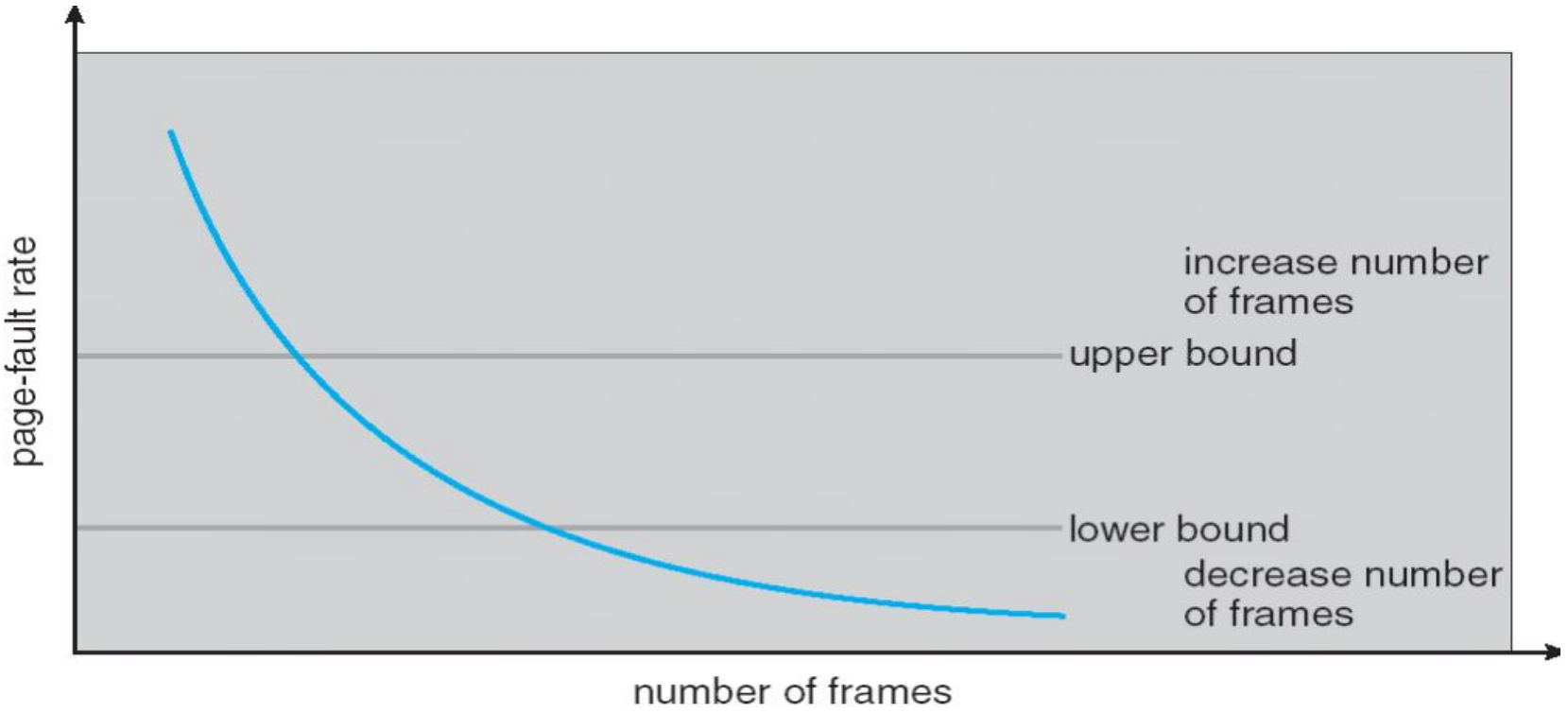
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma\ WSS_i$ ≡ total demand frames
- if $D > M$ ⇒ Thrashing
- Policy if D > M, then suspend one of the processes

Working-set model



Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Example-1: Suppose R=3,2,4,3,4,2,2,3,4,5,6,7,7,6,5,4,5,6,7,2,1 is a page reference stream, if the window size is 6 and assuming pure demand paging. How may page faults will it cause under the working set algorithm.

Solution: Working set here at any point contains the previous 6 page references (non-distinct).

3 - Working Set = [3], Page fault count = 1
2 - Working Set = [3 2], Page fault count = 2
4 - Working Set = [3 2 4], Page fault count = 3
3 - Working Set = [2 4 3], Page fault count = 3
4 - Working Set = [2 3 4], Page fault count = 3
2 - Working Set = [3 4 2], Page fault count = 3
2 - Working Set = [3 4 2], Page fault count = 3
3 - Working Set = [4 2 3], Page fault count = 3
4 - Working Set = [2 3 4], Page fault count = 3
5 - Working Set = [2 3 4 5], Page fault count = 4
6 - Working Set = [2 3 4 5 6], Page fault count = 5
7 - Working Set = [2 3 4 5 6 7], Page fault count = 6
7 - Working Set = [3 4 5 6 7], Page fault count = 6
6 - Working Set = [4 5 7 6], Page fault count = 6
5 - Working Set = [7 6 5], Page fault count = 6
4 - Working Set = [7 6 5 4], Page fault count = 7
5 - Working Set = [7 6 4 5], Page fault count = 7
6 - Working Set = [7 4 5 6], Page fault count = 7
7 - Working Set = [4 5 6 7], Page fault count = 7
2 - Working Set = [4 5 6 7 2], Page fault count = 8
1 - Working Set = [4 5 6 7 2 1], Page fault count = 9

Example-2: Let the page reference and the working set window be c c d b c e c e a d and 4, respectively. The initial working set at time t=0 contains the pages {a,d,e}, where a was referenced at time t=0, d was referenced at time t=−1, and e was referenced at time t=−2. Determine the total number of page faults and the average number of page frames used by computing the working set at each reference.

Answer: 5 page faults

Avg frame requirement=3.2