Chapter 9: Virtual Memory

Virtual Memory

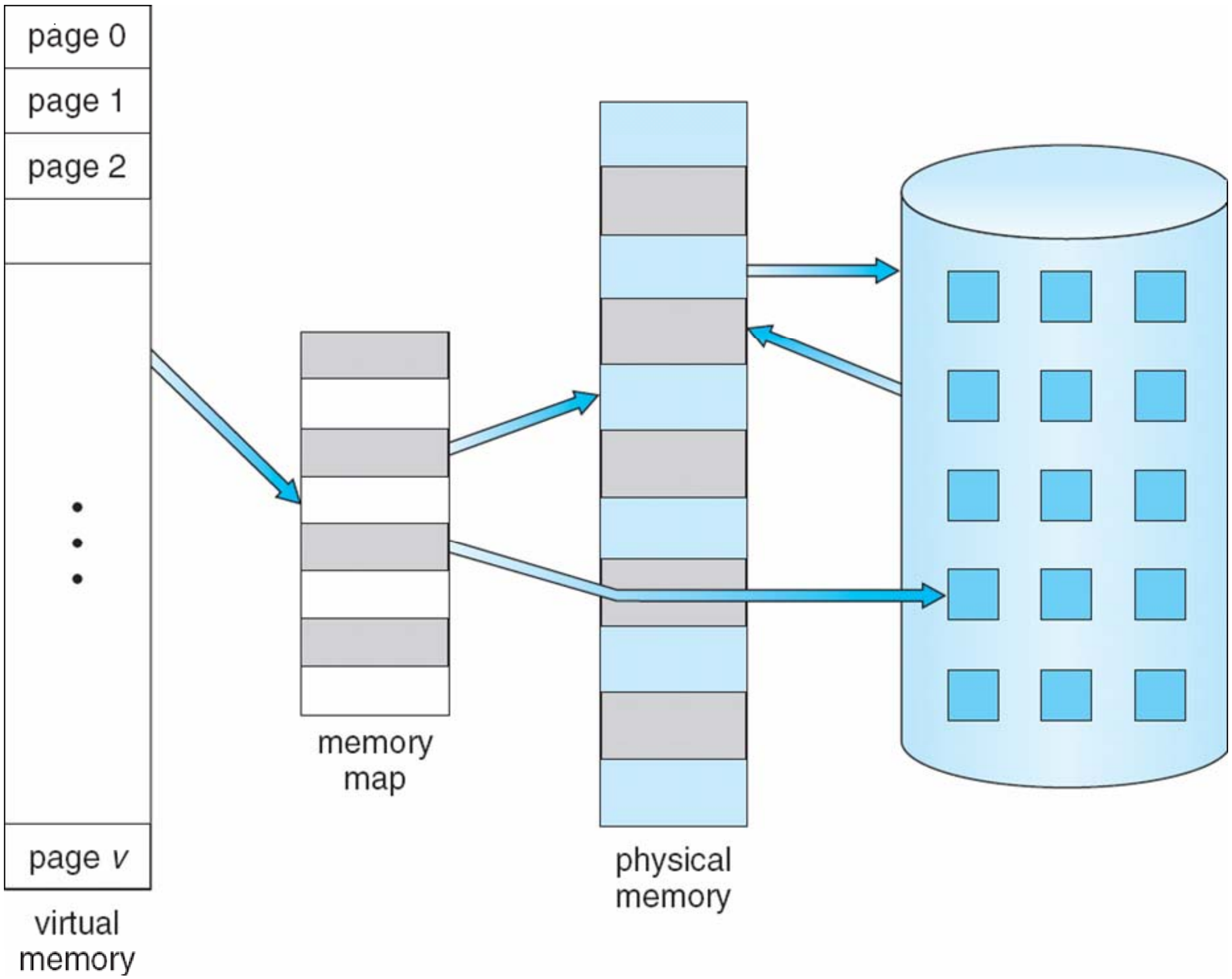
- Objectives
  - To describe the benefits of a virtual memory system
  - To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
  - To discuss the principles of the working-set model
- Topics
  - Background
  - Demand Paging
  - Process Creation: Copy-on-Write and Memory-mapped Files
  - Page Replacement
  - Allocation of Frames
  - Thrashing
  - Memory-Mapped Files
  - Other Considerations

Background

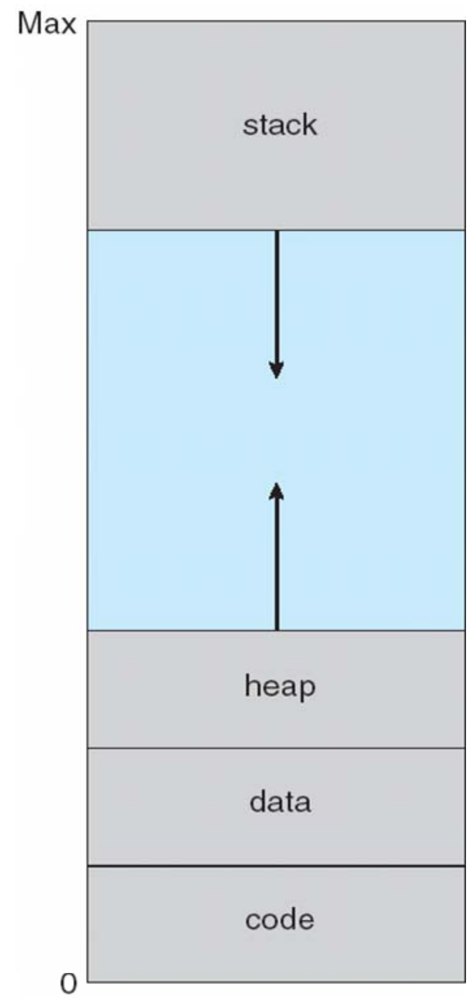
- Virtual memory
  - Separation of user logical memory from physical memory
    - Only part of the program needs to be in memory for execution
    - Logical address space can therefore be much larger than physical address space
    - Allows address spaces to be shared by several processes
    - Allows for more efficient process creation

- Virtual memory can be implemented via
  - Demand paging
  - Demand segmentation

Virtual Memory and Address Space

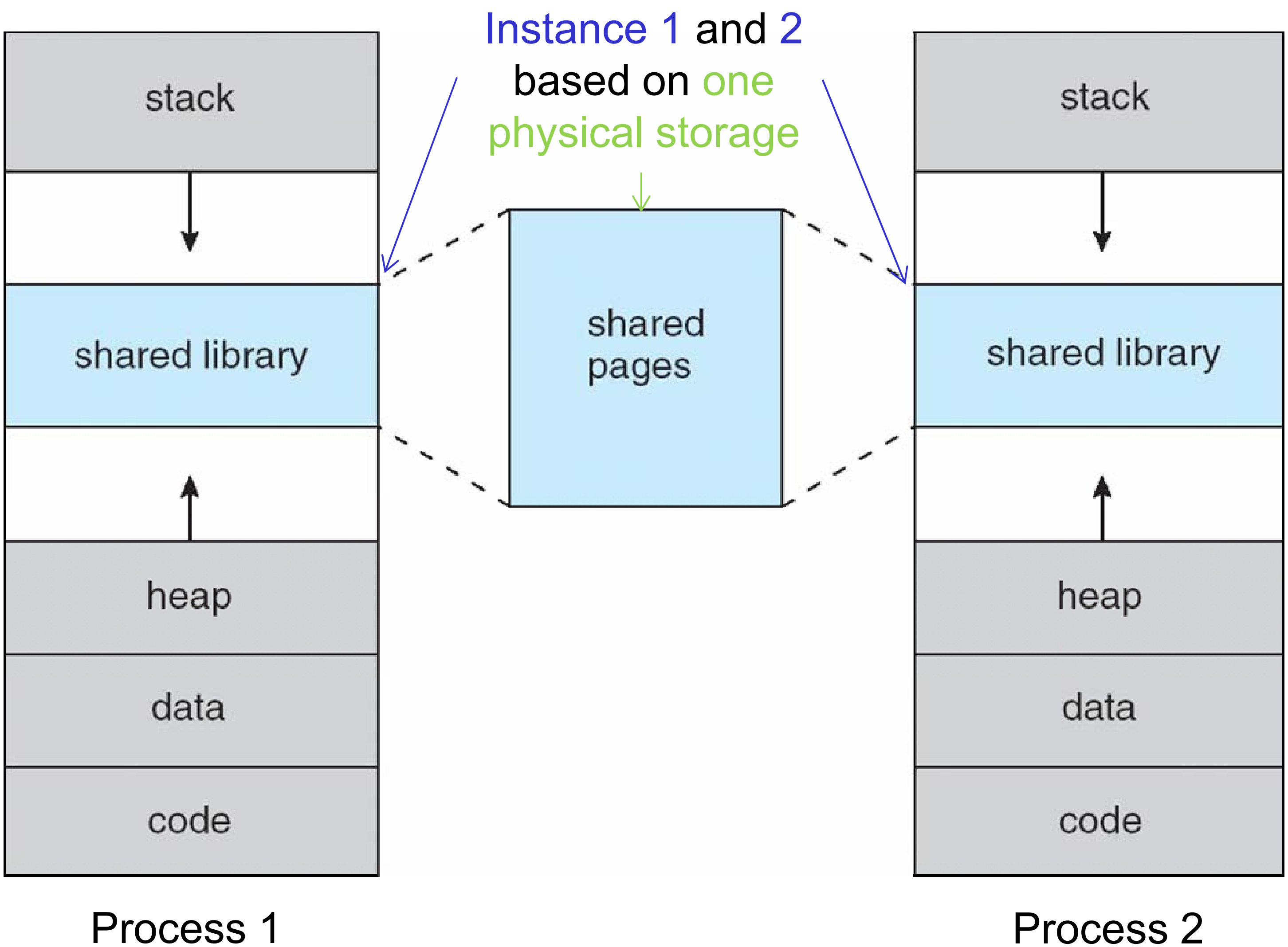


Virtual Memory (larger than Physical Memory)



Virtual Address Space

Shared Library Using Virtual Memory



Demand Paging

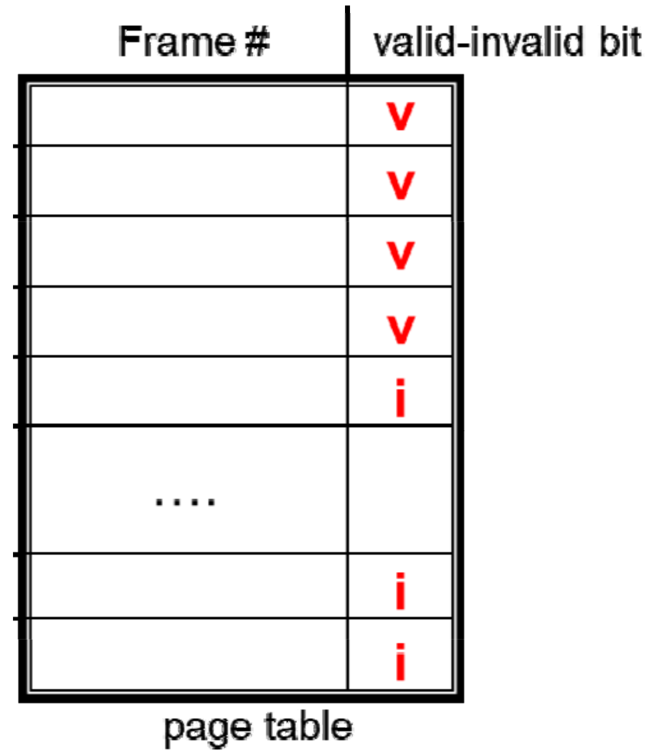
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - Invalid reference $\Rightarrow$ abort
  - Not-in-memory $\Rightarrow$ bring to memory
- Lazy swapper: never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

Valid-Invalid Bit

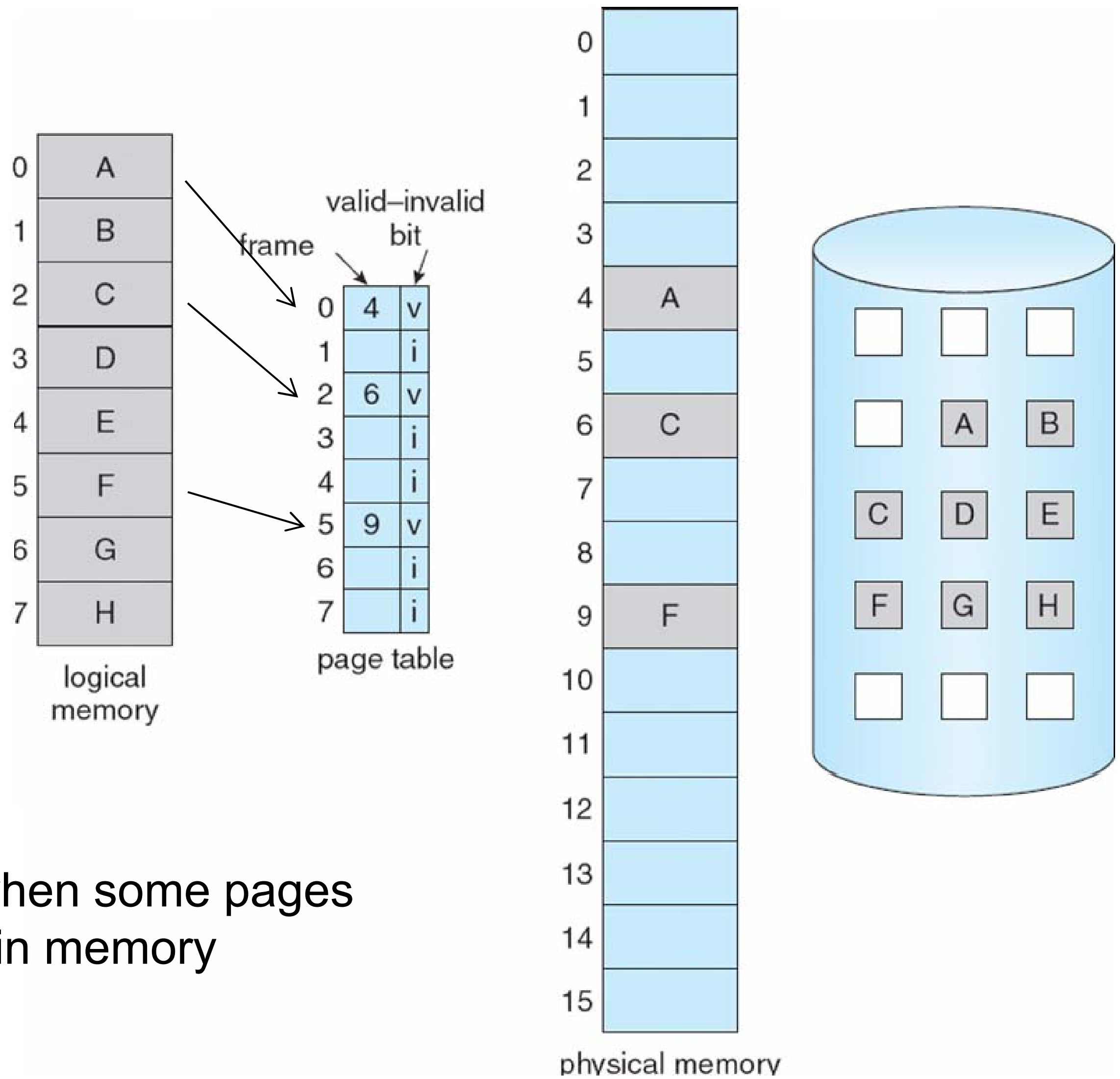
- With each page table entry a valid–invalid bit is associated
  - **v** ⇒ in-memory
  - **i** ⇒ not-in-memory

- Initially valid–invalid bit is set to **i** on all entries

- Example of a page table snapshot: During address translation, if valid–invalid bit in page table entry is **I** ⇒ page fault



Pages Missing in Main Memory



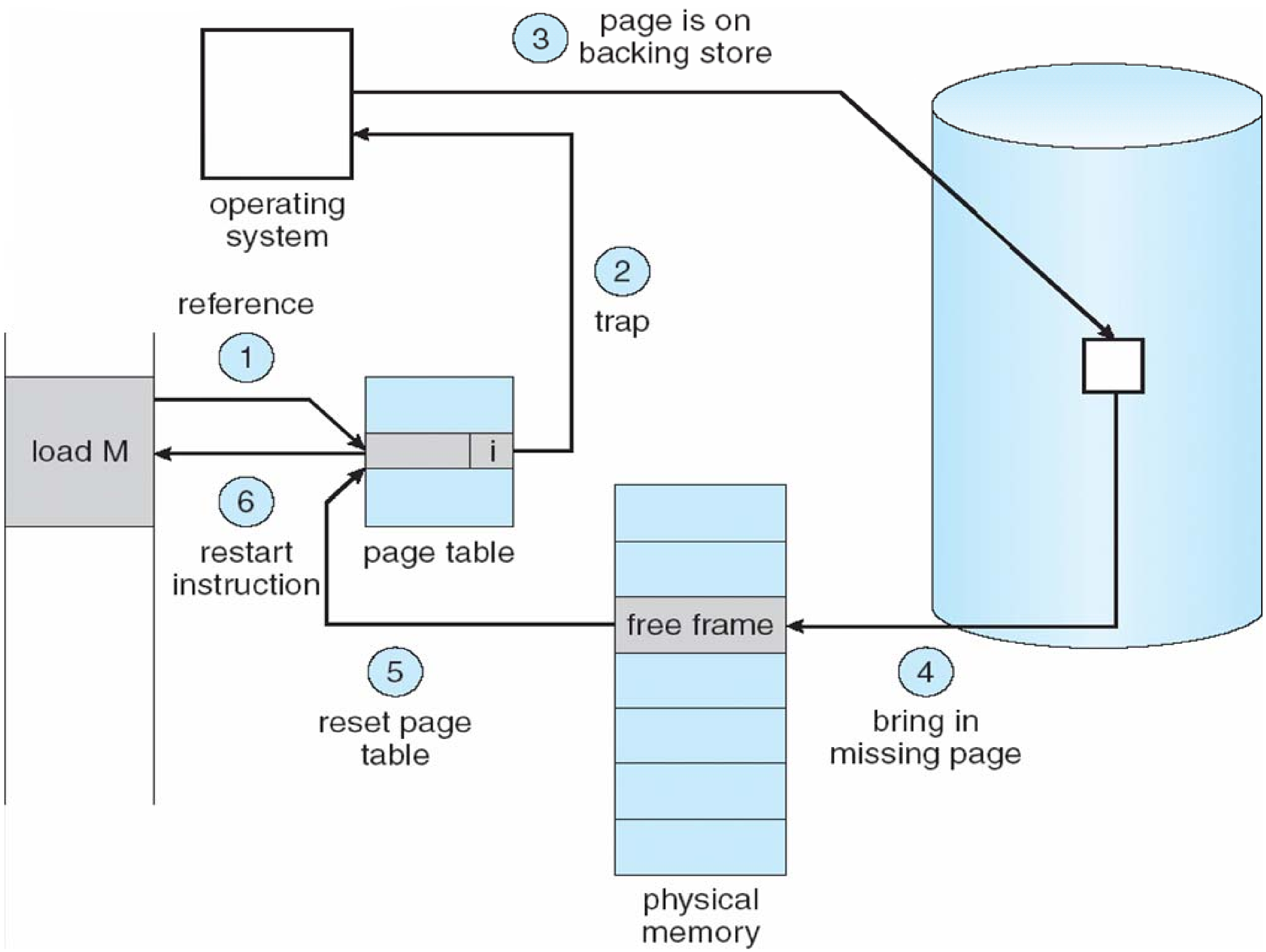
Page Table when some pages are not in main memory

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 another table to decide:
   – Invalid reference ⇒ abort
   – Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Process Creation

- Virtual memory allows for other benefits during the process creation (such as `fork()`):
  - Copy-on-Write
  - Memory-Mapped Files

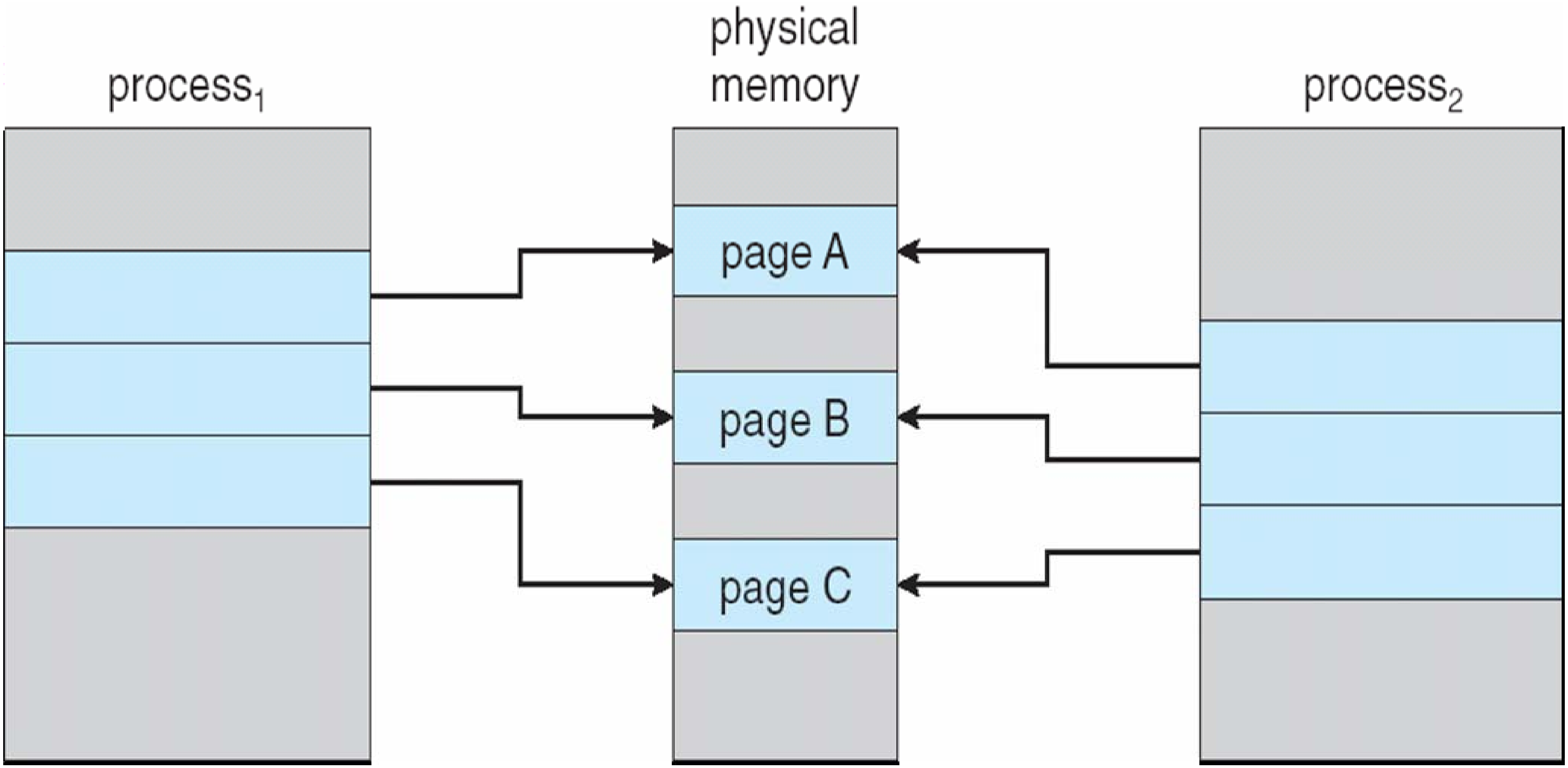
Copy-on-Write

- 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

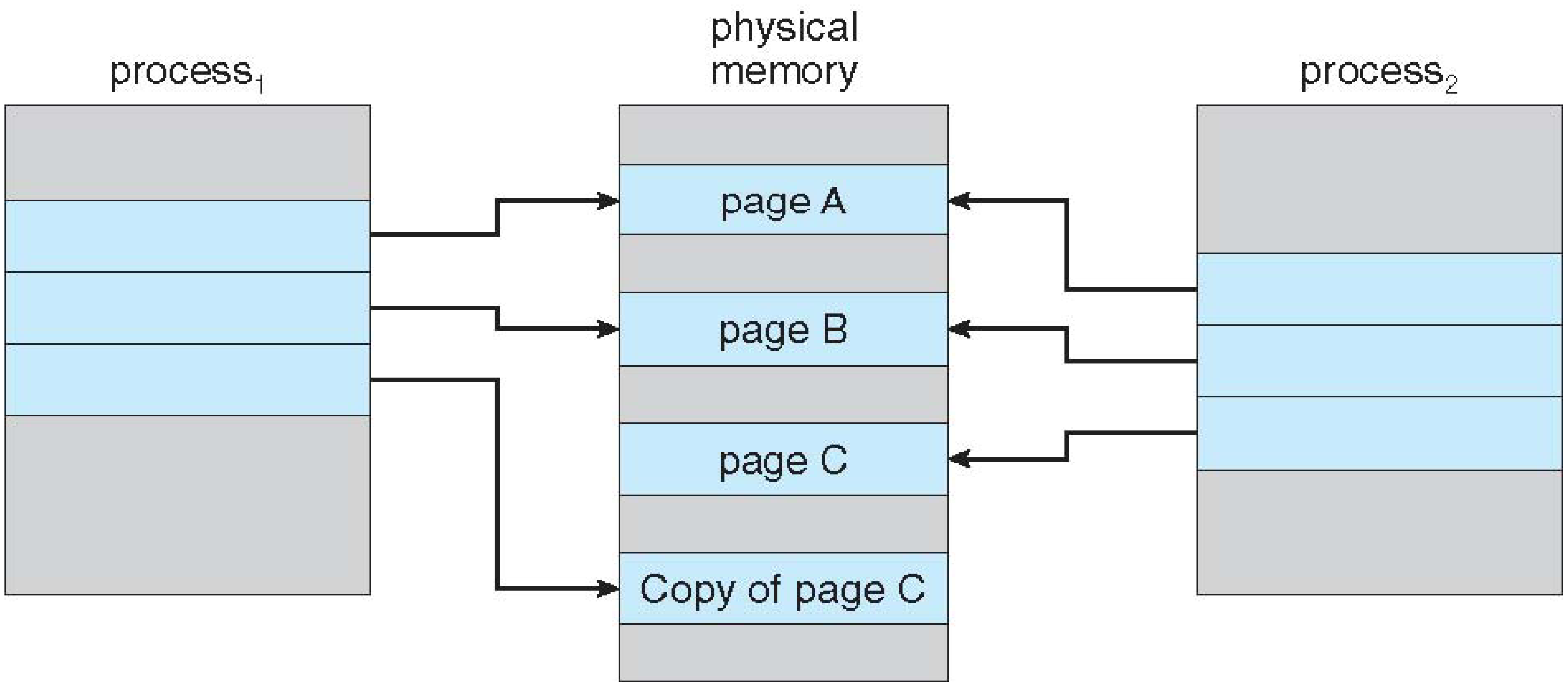
- COW allows more efficient process creation as only modified pages are copied

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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Only pages that can be modified need be marked as copy-on-write!

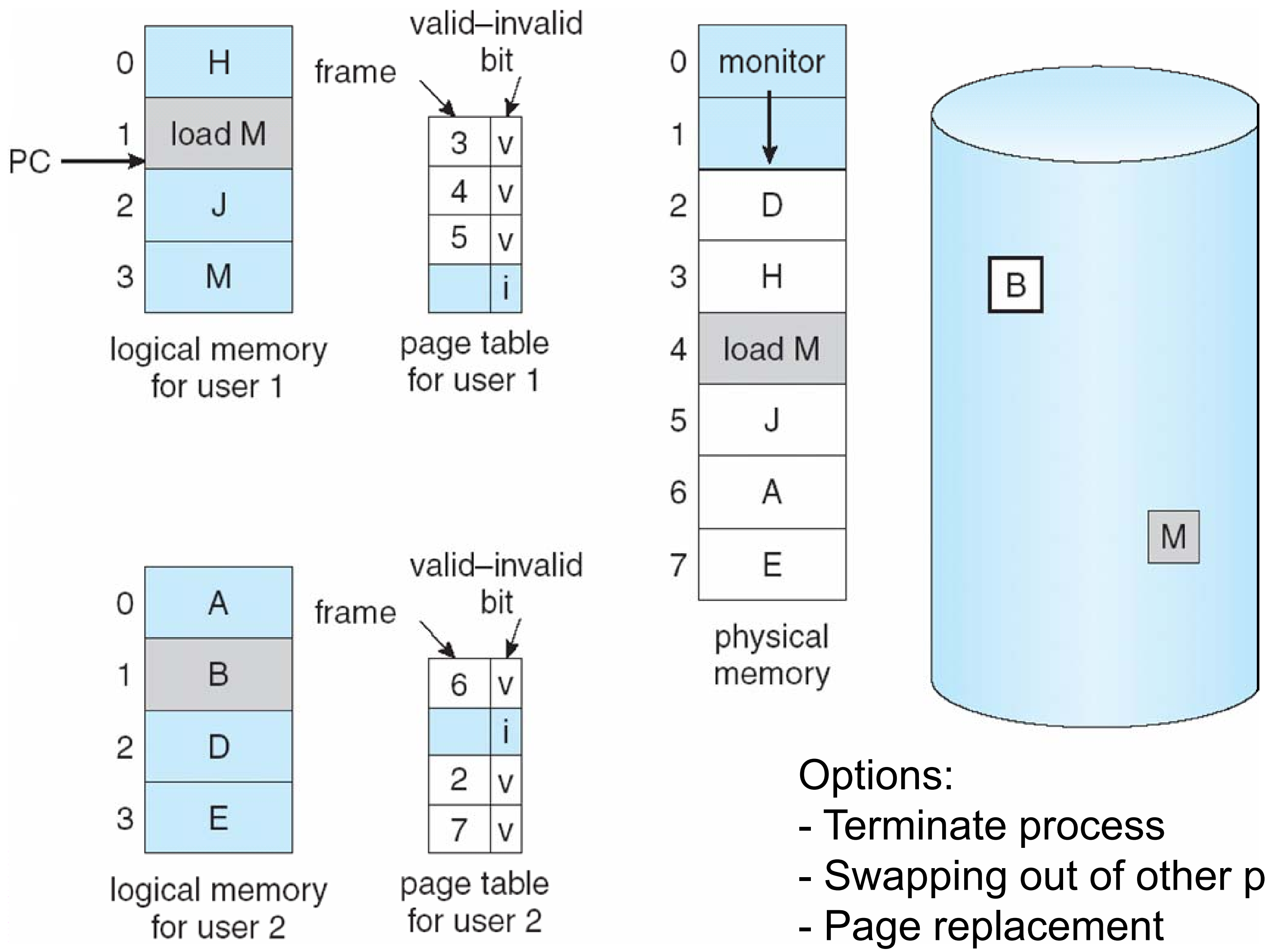
Lack of Free Frames

- Page replacement
  - Find some page in memory, but not really in use, swap it out
    - Algorithm
    - Performance demands
      - In search of an algorithm which will result in a 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



Options:
- Terminate process
- Swapping out of other process
- Page replacement

Basic Page Replacement

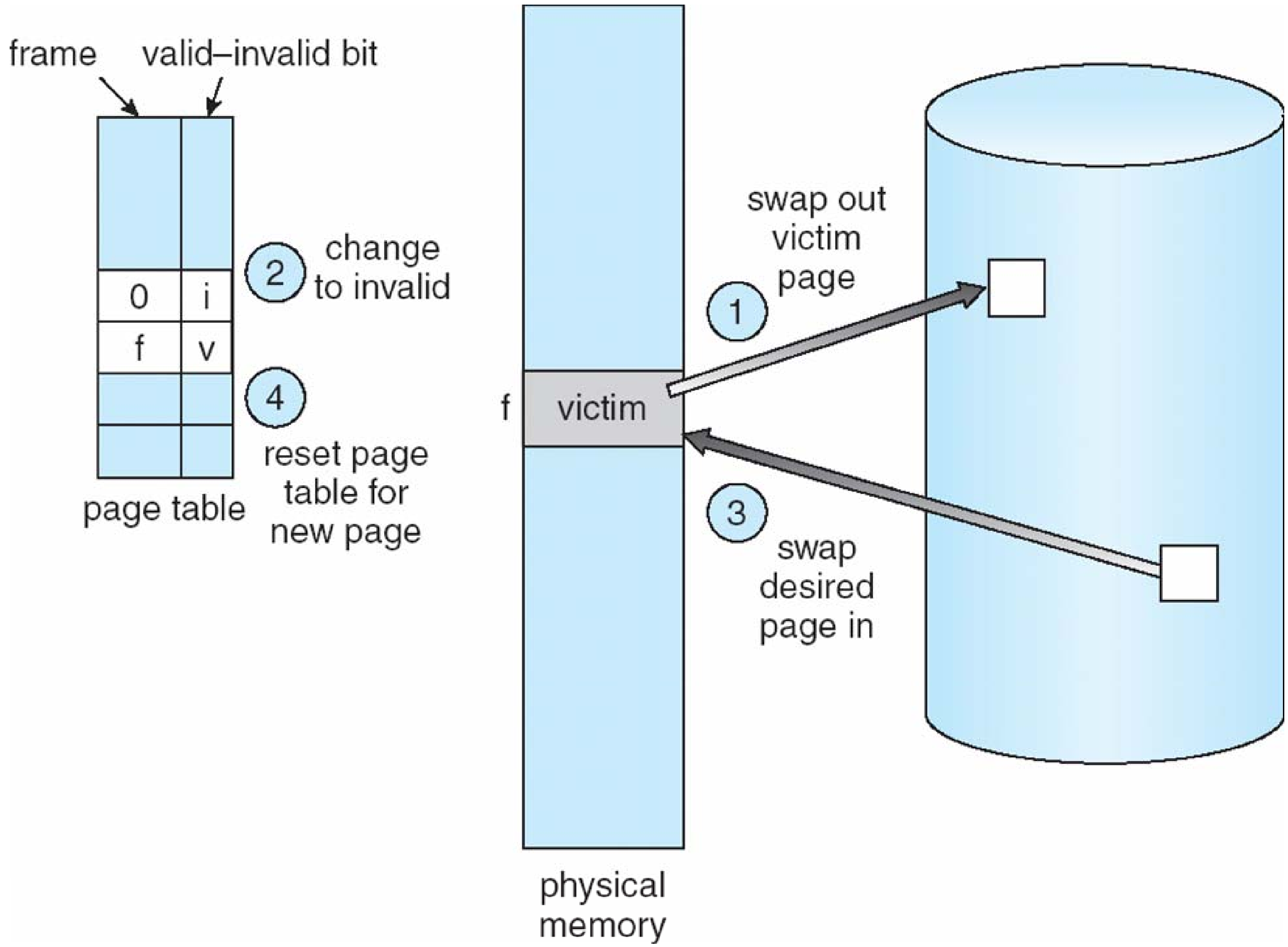
1. Find the location of the desired page on disk

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

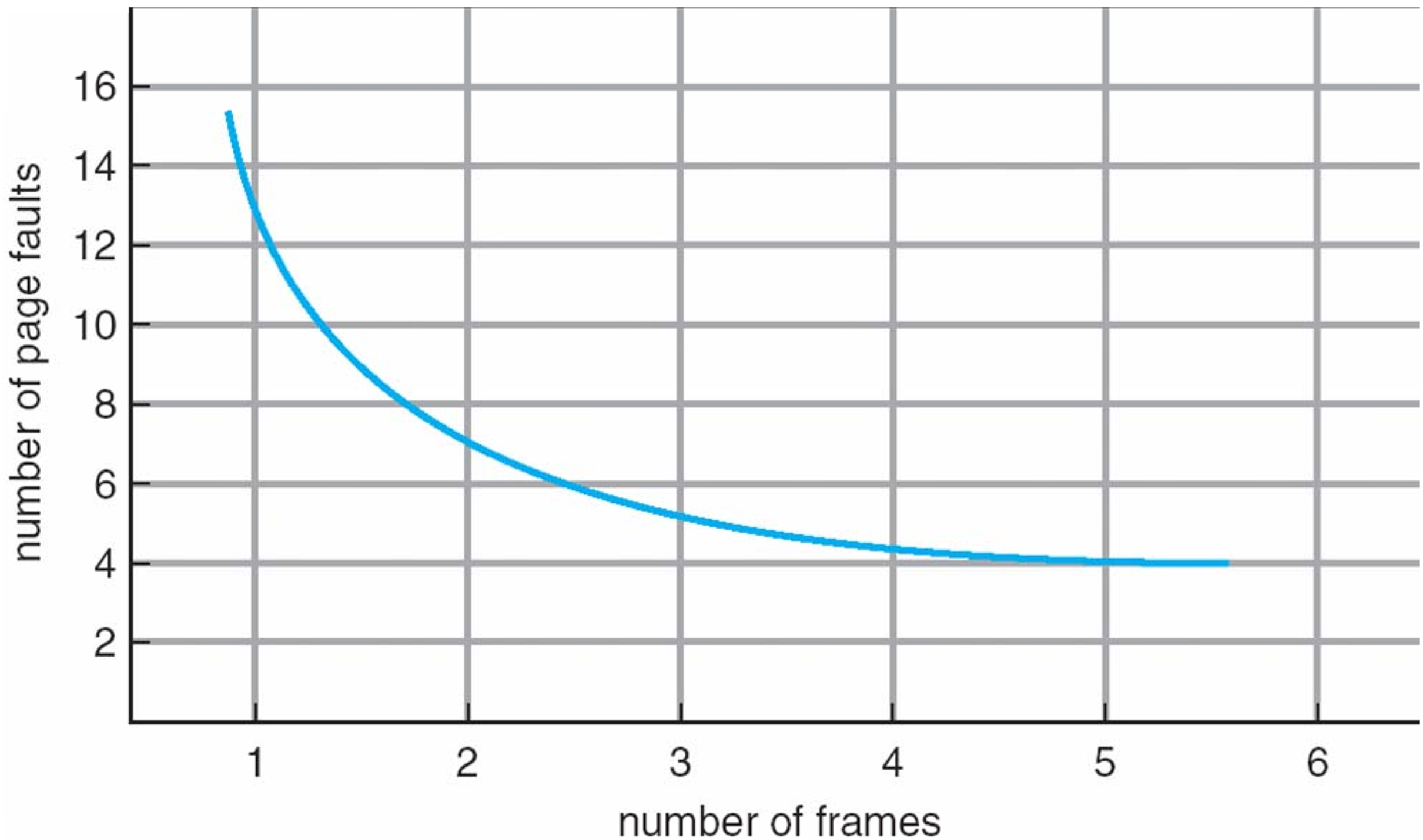


Page Replacement Algorithms

- 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
- In all of the examples the reference string is:

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

Page Faults vs. Number of Frames

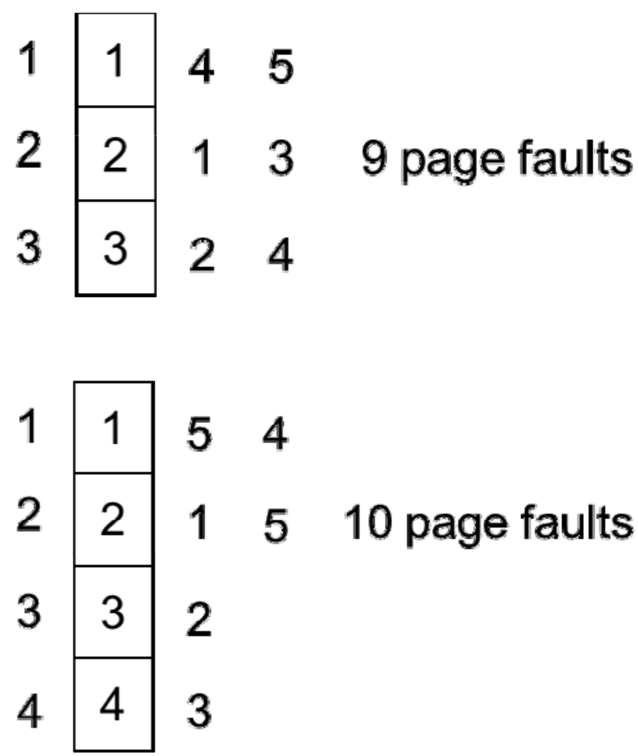


Expectation: #frames increases, #page faults drops, or adding physical memory

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):

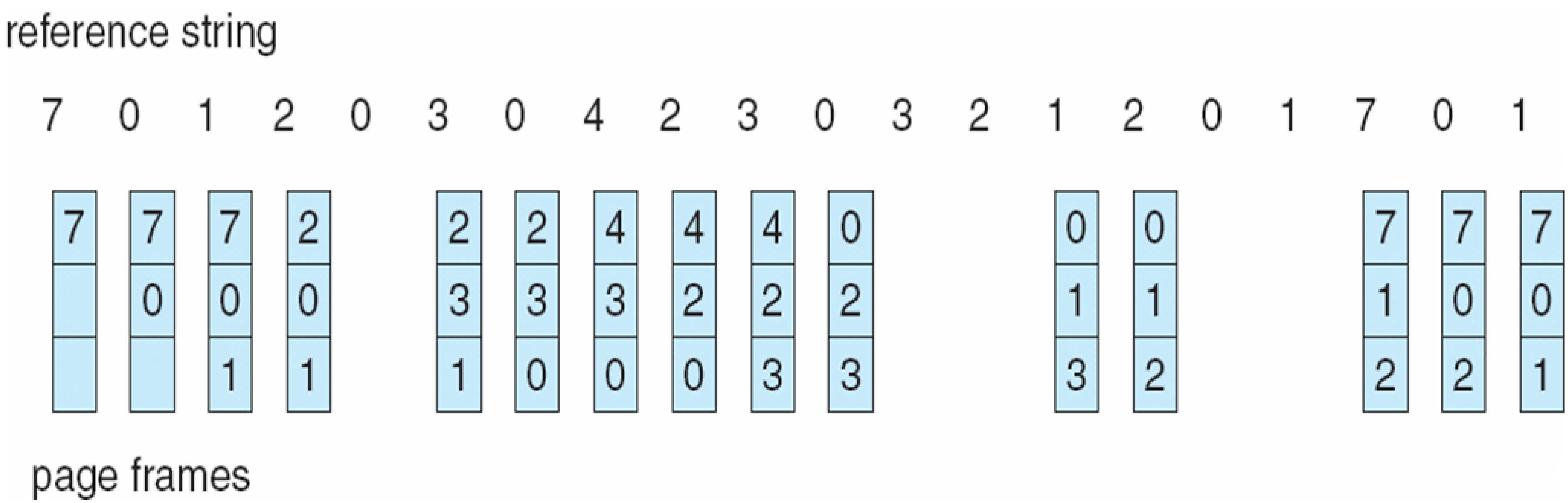
4 frames:



FIFO queue:
- All page requests queued
- Replace page at head of queue
- Page brought into memory is added at tail of queue

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

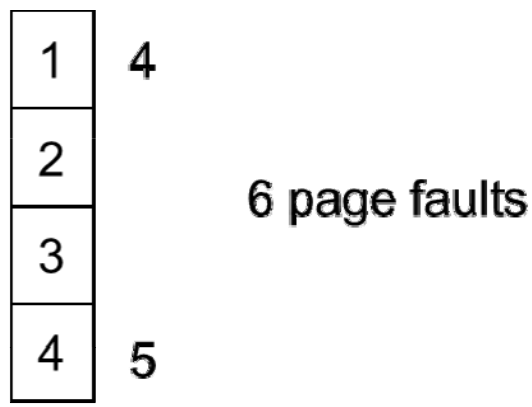
FIFO Page Replacement



Optimal Algorithm

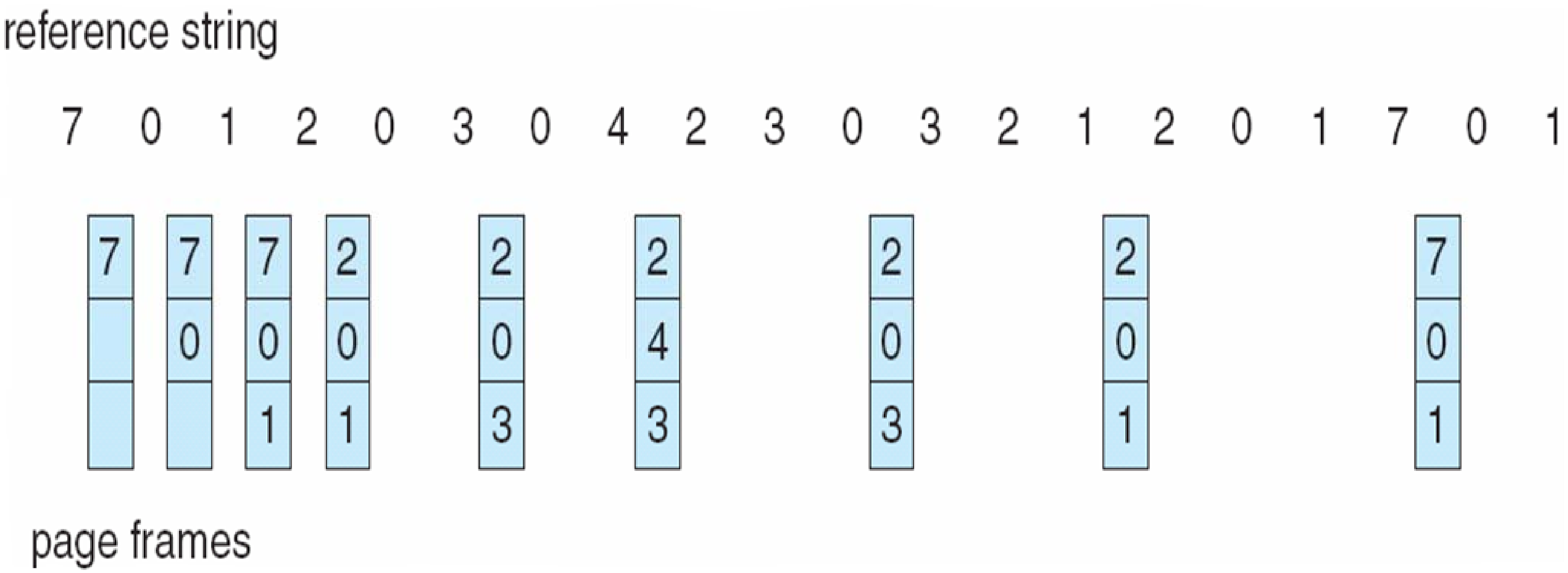
- Replace page that will not be used for longest period of time
- 4 frames example:

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



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

Optimal Page Replacement



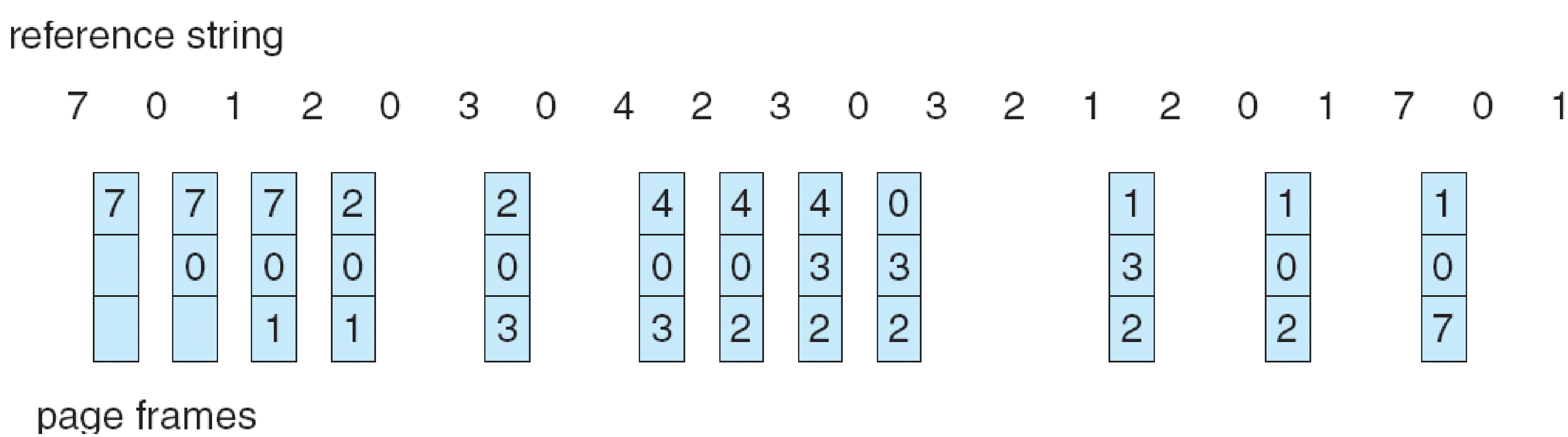
Least Recently Used (LRU) Algorithm

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

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



Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- LFU Algorithm
  - Replaces page with smallest count

- MFU Algorithm
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
  – How to allocate the fixed amount of free memory among P_i?

- Example: IBM 370
  – 6 pages to handle SS MOVE instruction:
    • Instruction is 6 byte, might span 2 pages
    • 2 pages to handle *from*
    • 2 pages to handle *to*

- Two major allocation schemes
  – Fixed allocation
  – Priority allocation

Fixed Allocation

- Equal allocation
  - *E.g.*, 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

- Use a proportional allocation scheme using priorities rather than size

- 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

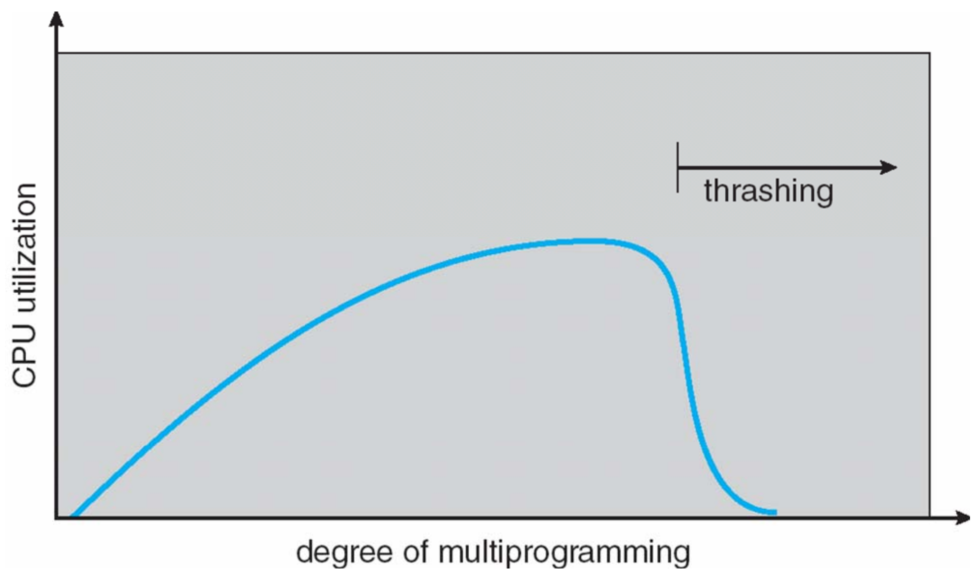
Global vs. Local Allocation

- 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

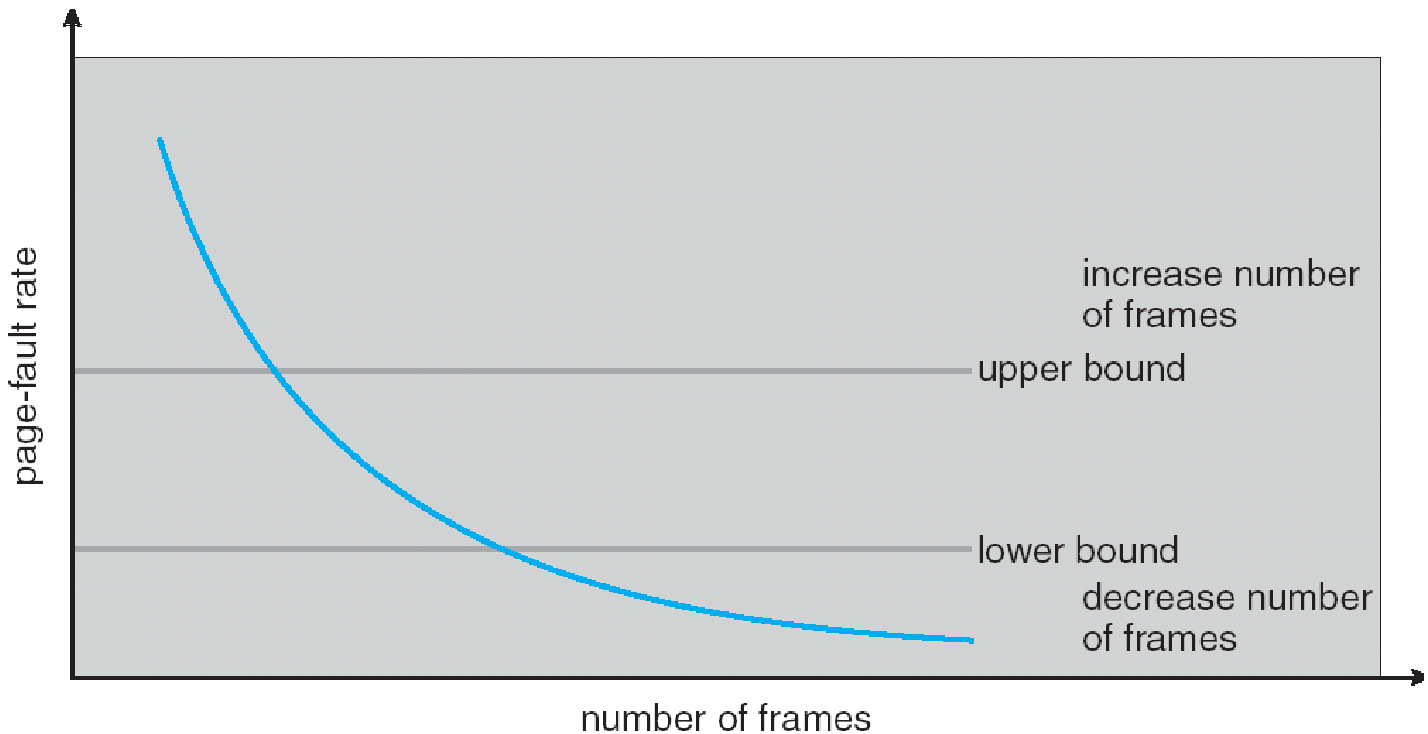
- If a process does not have “enough” pages, the page-fault rate is very high
- This leads to
  - Low CPU utilization
  - Operating system thinks that it needs to increase the degree of multiprogramming
  - Another process added to the system



- Thrashing ≡ a process is busy swapping pages in and out

Page-Fault Frequency Scheme

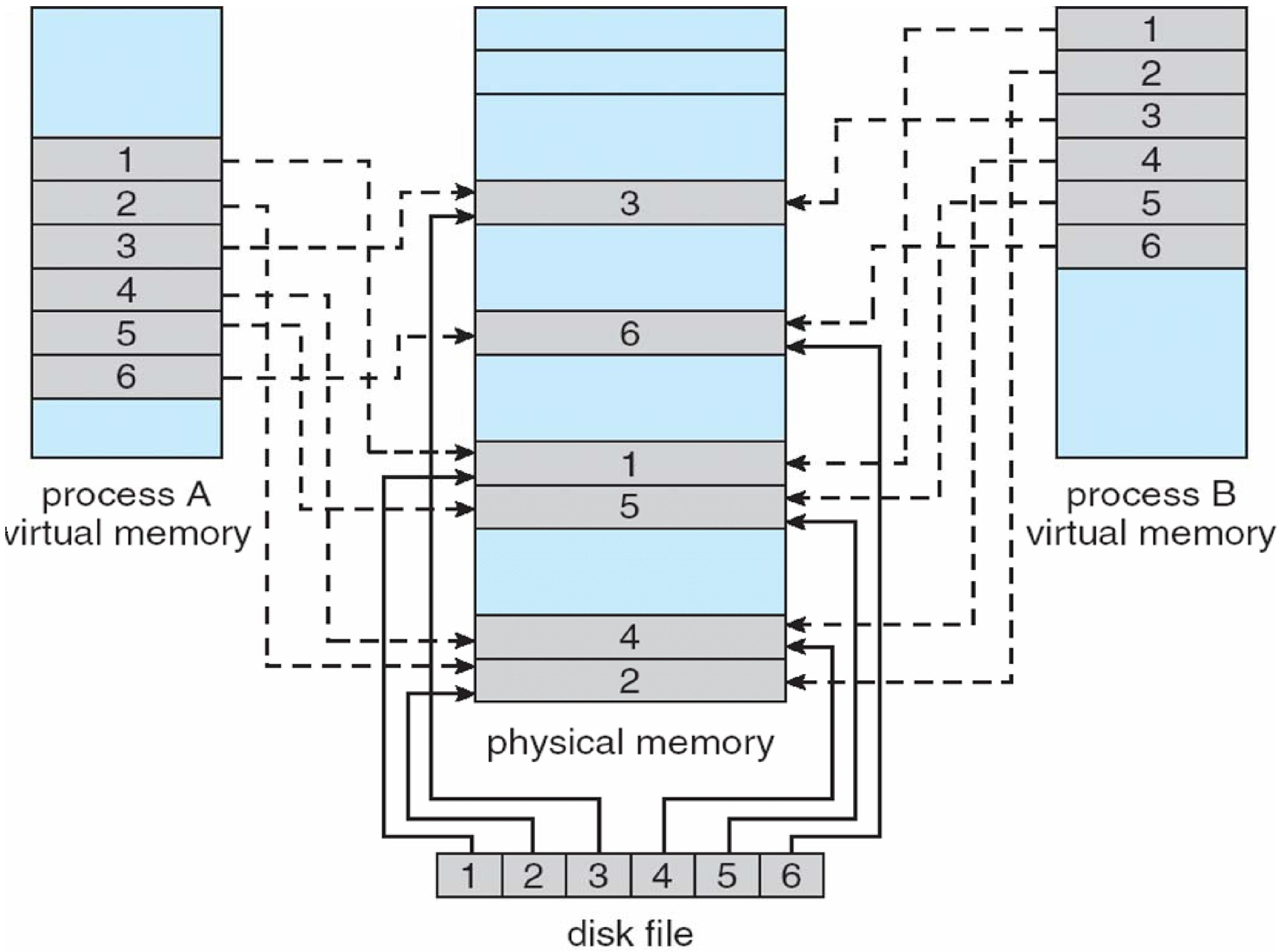
- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Memory Mapped Files (1)

- 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
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files (2)



Other Issues (1)

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues (2)

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - Locality

- I/O Interlock
  - Pages must sometimes be locked into memory
    - Consider I/O: Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm