CS420: Operating Systems

Virtual Memory

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

Based on Operating System Concepts, 9th Edition by Silberschatz, Galvin, Gagne

Background

- **Code needs to be in memory to execute, but entire program rarely used**
  - Error code, unusual routines, large data structures

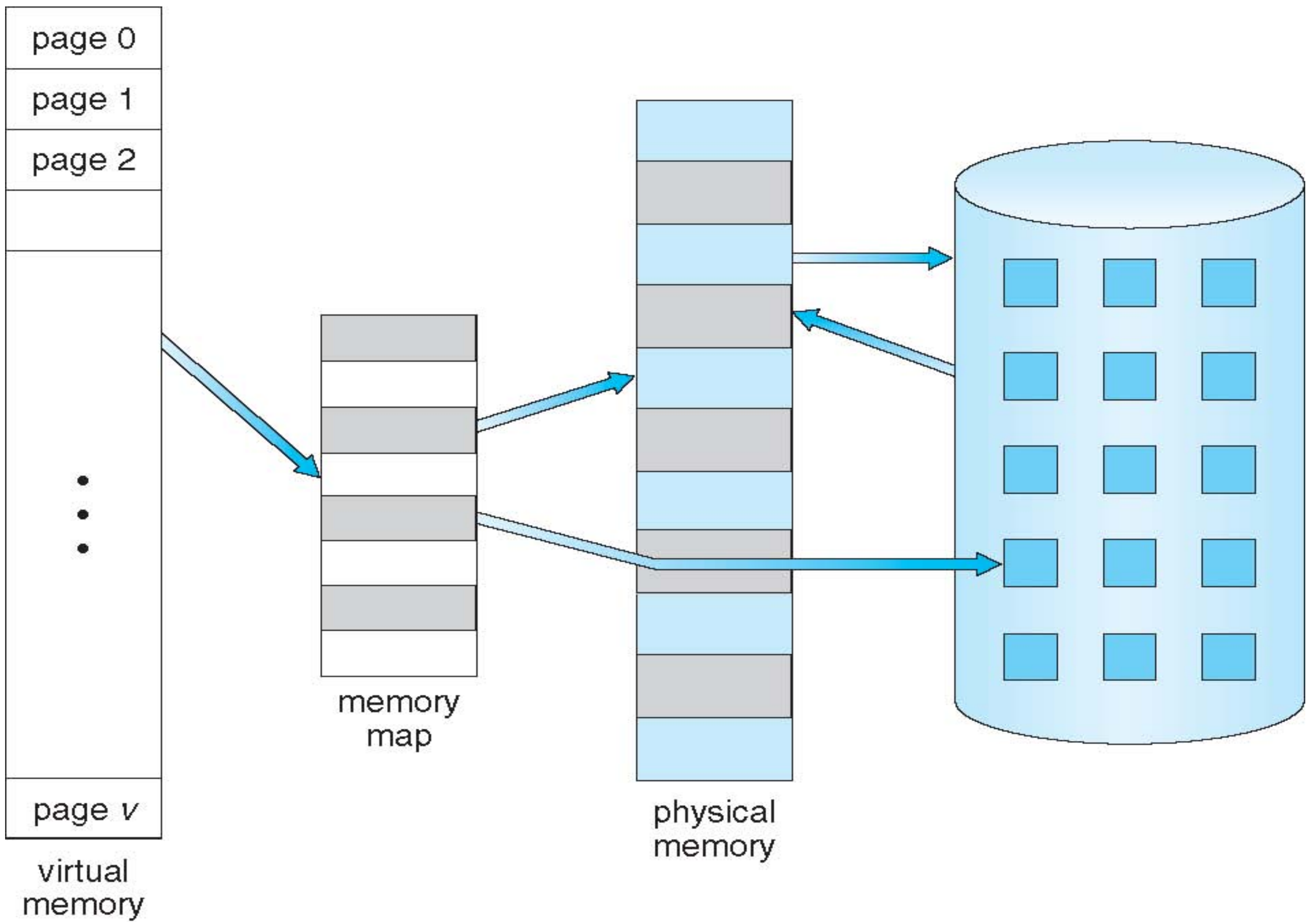
- **Want the ability to execute partially-loaded program**
  - Programs no longer constrained by limits of physical memory
  - Program data no longer constrained by limits of physical memory

Background - 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 physical address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

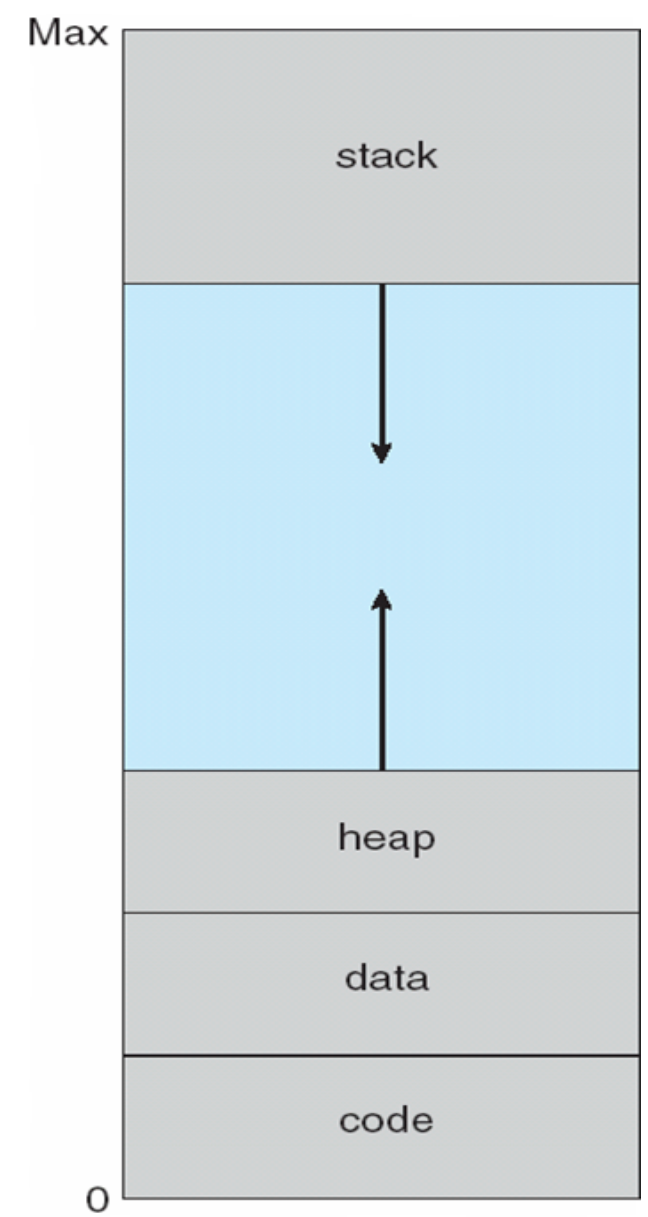
- **In contrast to dynamic loading, virtual memory does not require programmer to do anything extra**

Virtual Memory that is Larger than Physical Memory



Virtual Address Space

- **Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc.**
  - e.g. don't waste physical memory with empty space that is intended for the growth of the stack/heap

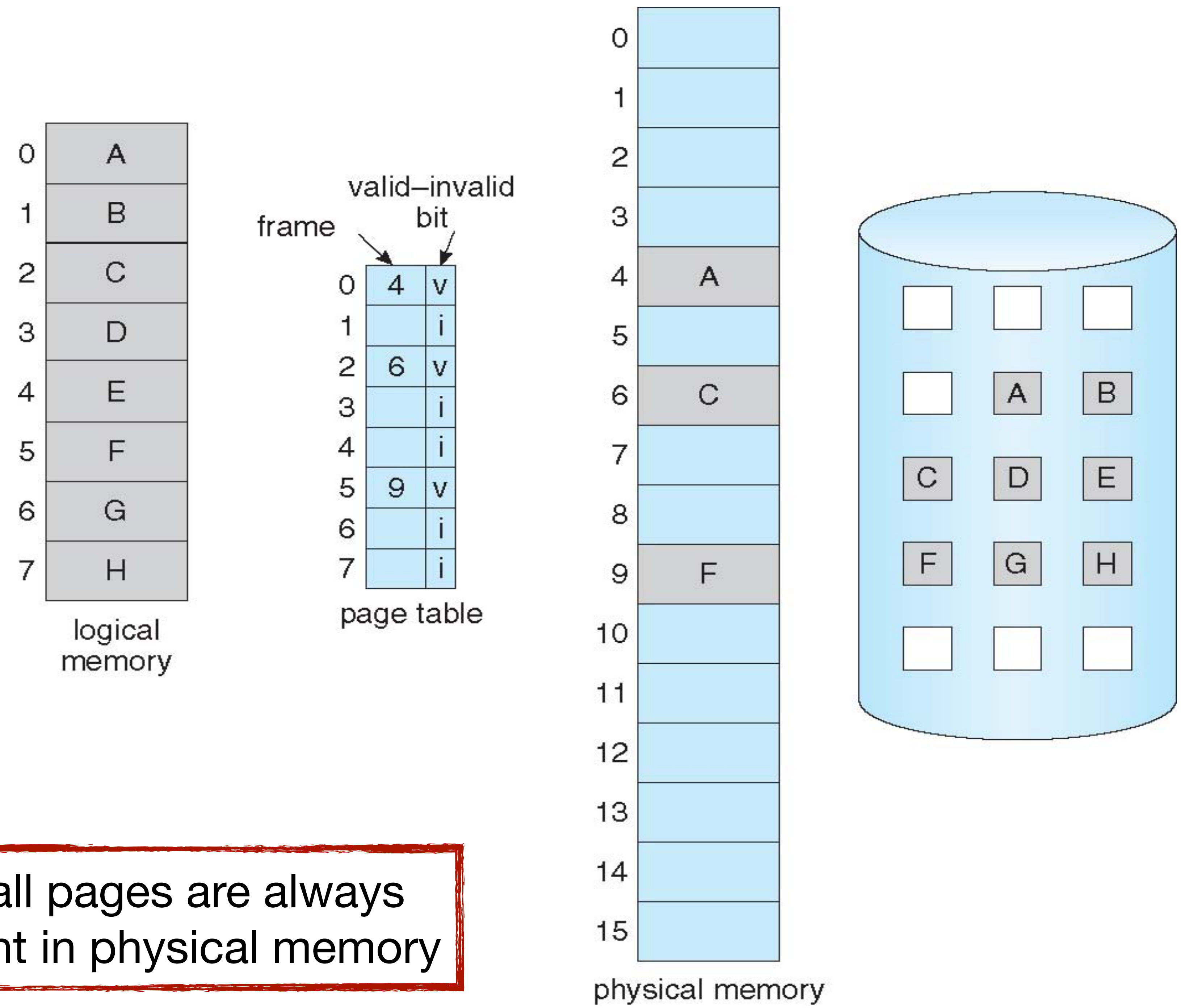


- **System libraries can be shared by mapping them into virtual address space**
- **Can create shared memory by mapping pages into virtual address space**
- **Virtual memory allows pages to be shared during fork(), speeding up process creation**

Demand Paging

- **Could bring entire process into memory at load time**
- **Or load a page into memory only when it is needed, called demand paging**
  - Pages that are never used are never loaded
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
  - Similar to page table system with swapping, but demand paging doesn't swap entire process into memory, instead uses a lazy swapper
- **Lazy swapper – never swaps a page into memory unless it will be needed**
  - A swapper that deals with pages is called a pager

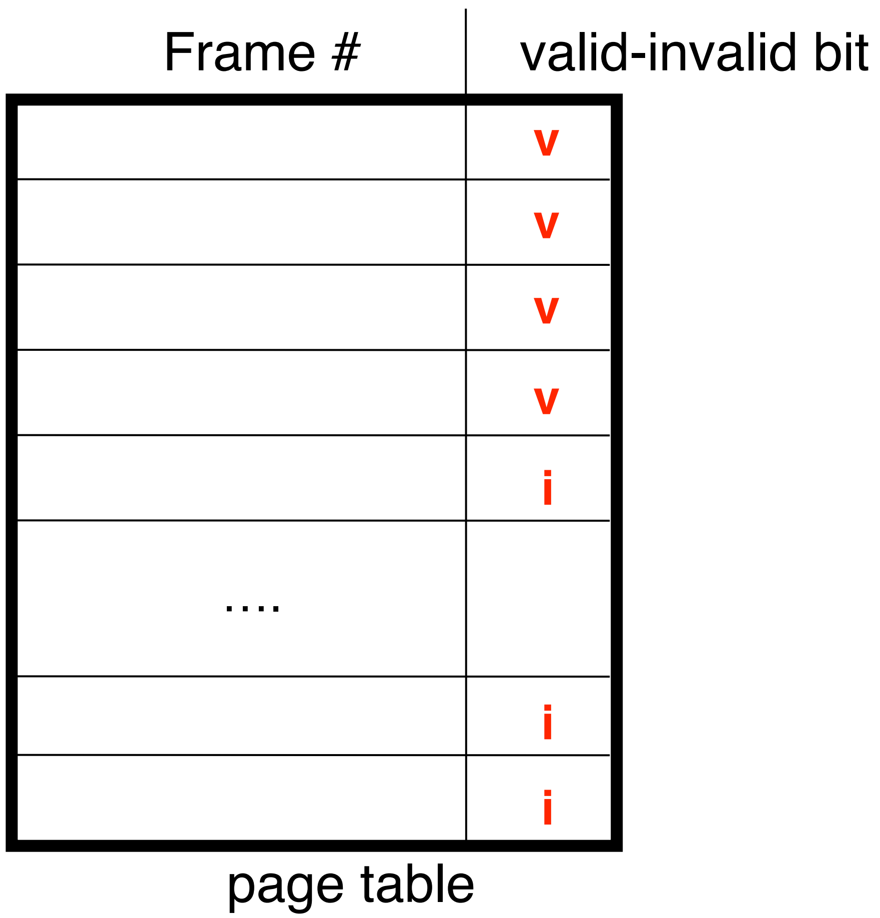
Page Table Example



Not all pages are always resident in physical memory

Valid-Invalid Bit

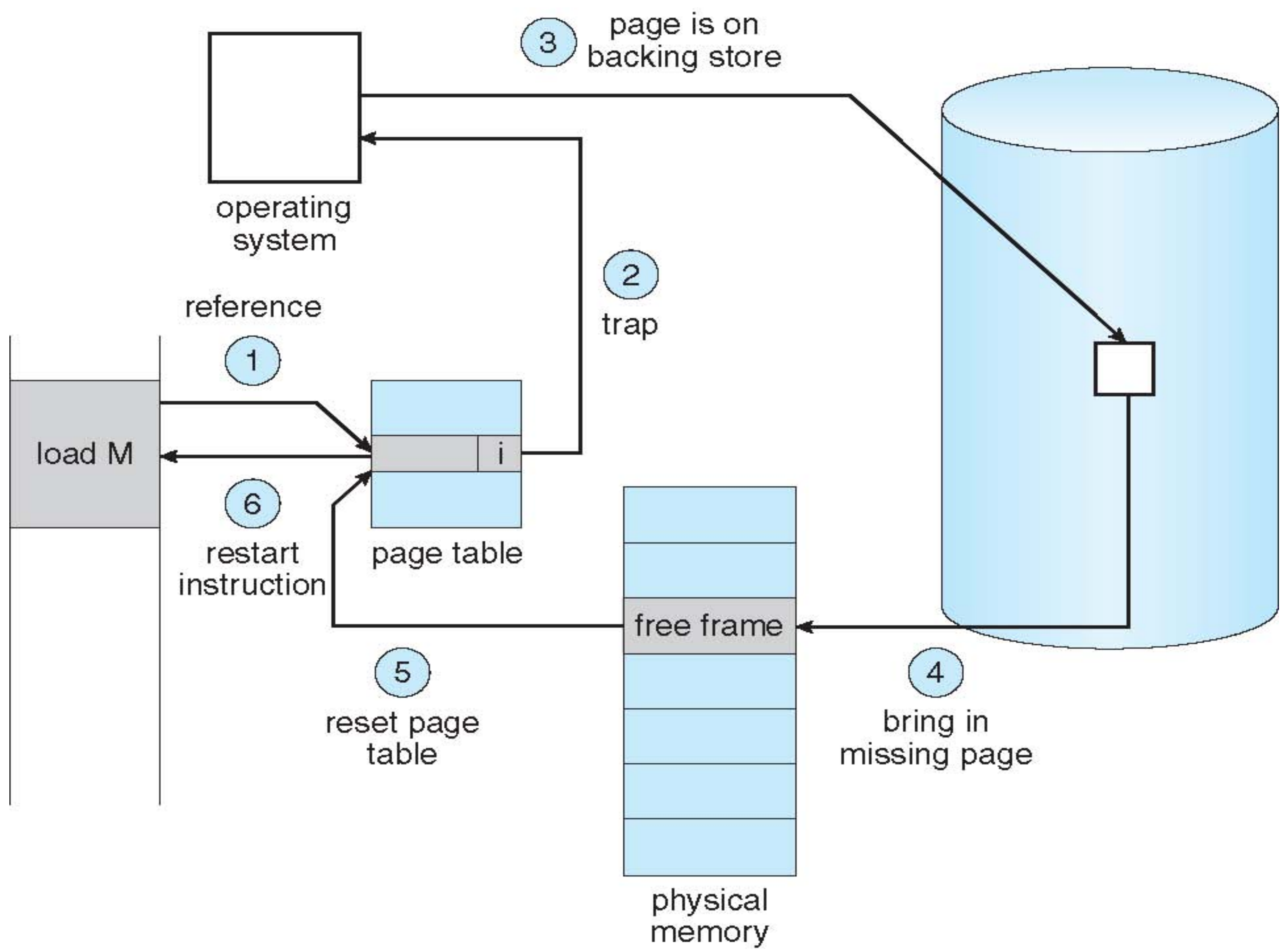
- **Each page table entry includes a valid–invalid bit**
  - If valid, then process is allowed to access that page, AND it is in physical memory
  - If invalid, process may not be allowed to access the requested page, or the page may not yet be in physical memory
- **Initially valid–invalid bit is set to i on all entries**



Page Fault

- **If requested page is memory resident, then process can operate normally**
- **If requested page not in physical memory (i.e. page is invalid in page table), then a page fault occurs**
- **If page fault occurs, first check to see if the requested page was an illegal request or if it just isn't in physical memory**
  - If an illegal reference, the process terminates (seg fault, bus error, ...)
  - If simply not in memory:
    - Get a free frame of physical memory from the free-list
    - Swap page from disk into the frame via a scheduled disk operation
    - Update tables to indicate that the page is now in physical memory (i.e. set valid-invalid bit to 'v')
    - Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

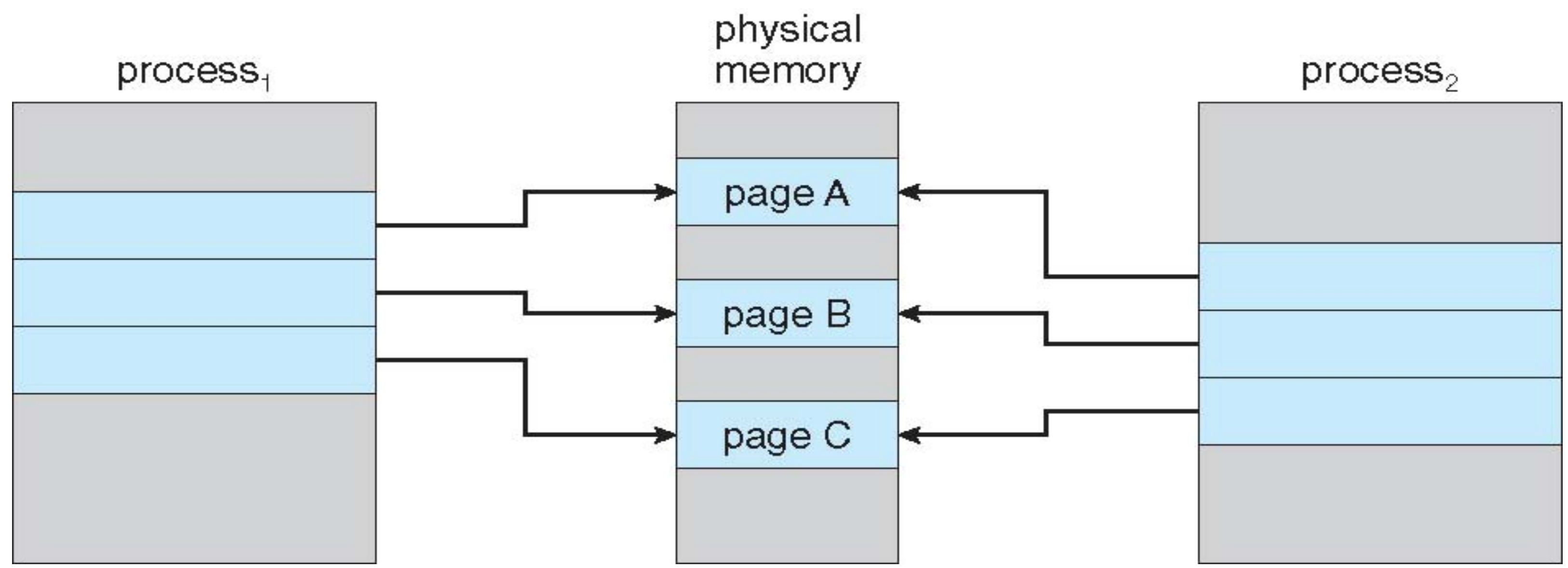
- **Extreme case of demand paging – start a process with no pages in memory**
  - Never bring a page into physical memory until it is needed
  - A page fault will occur each time a new page is requested, including on the very first page
  - The scheme is called pure demand paging
- **Demand paging (and pure demand paging) require hardware support**
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Ability to restart an instruction
- **Performance of system can be severely impacted if too much page swapping is required (see example in OSC9 section 9.2.2, 40x slowdown!)**

Page Sharing - Copy-on-Write

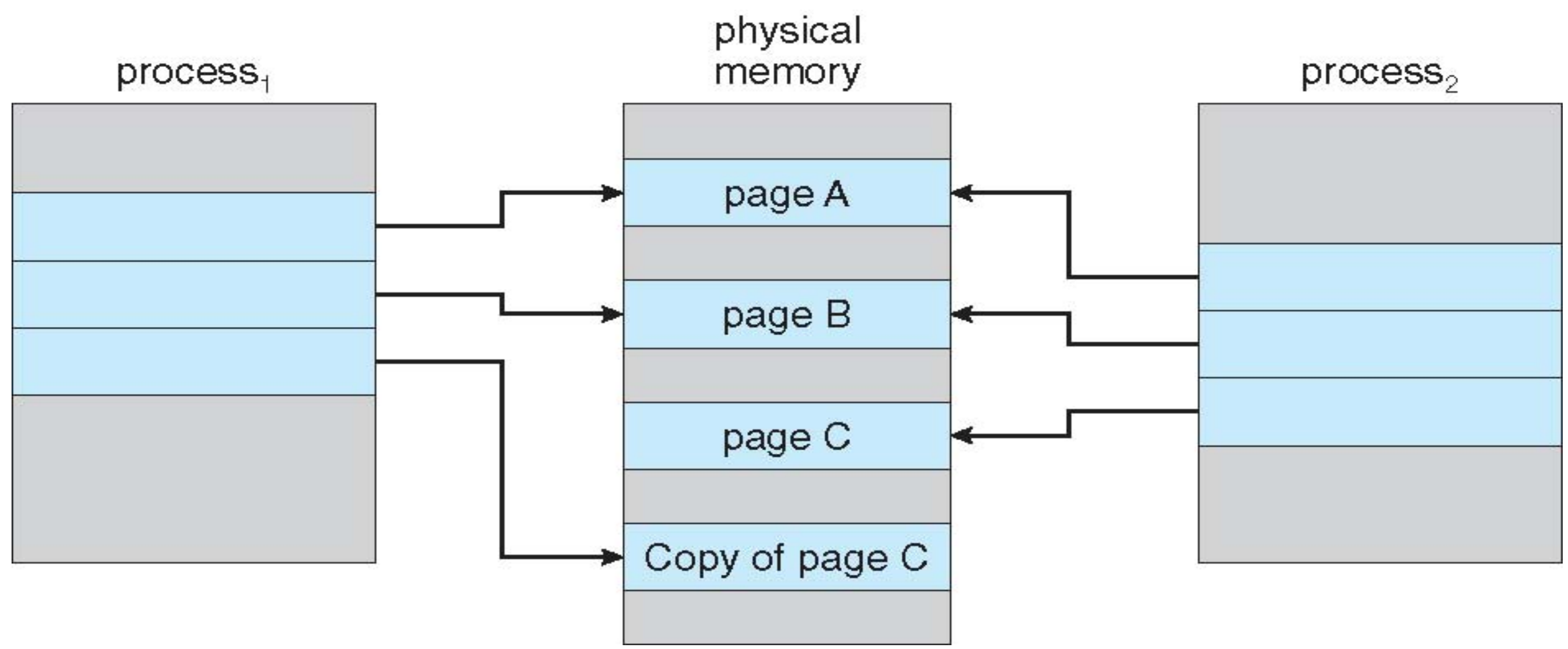
- **During process creation Copy-On-Write (COW) allows both parent and child processes to initially share the same pages in memory**
  - If either process modifies a shared page, only then is the page copied
- **COW allows more efficient process creation as only modified pages are copied**
- **Free pages are typically allocated from a pool of zero-fill-on-demand pages**
- **A variation of `fork()` exists called `vfork()` (virtual memory fork)**
  - Parent suspends so child can use its memory address space
  - If child modifies parent's address space, those changes will be visible to parent (i.e. it does NOT use copy-on-write)
  - Designed to be used when child calls exec() immediately after fork()
  - Very efficient since no pages are copied during creation of child

Example of Copy-on-Write

Memory before either process tries to write to shared pages



Memory after process$_1$ writes to page C



What Happens if There is no Free Frame?

- **All frames used up by process pages and I/O buffers**
  - How much memory should be allocated to I/O and how much to processes?

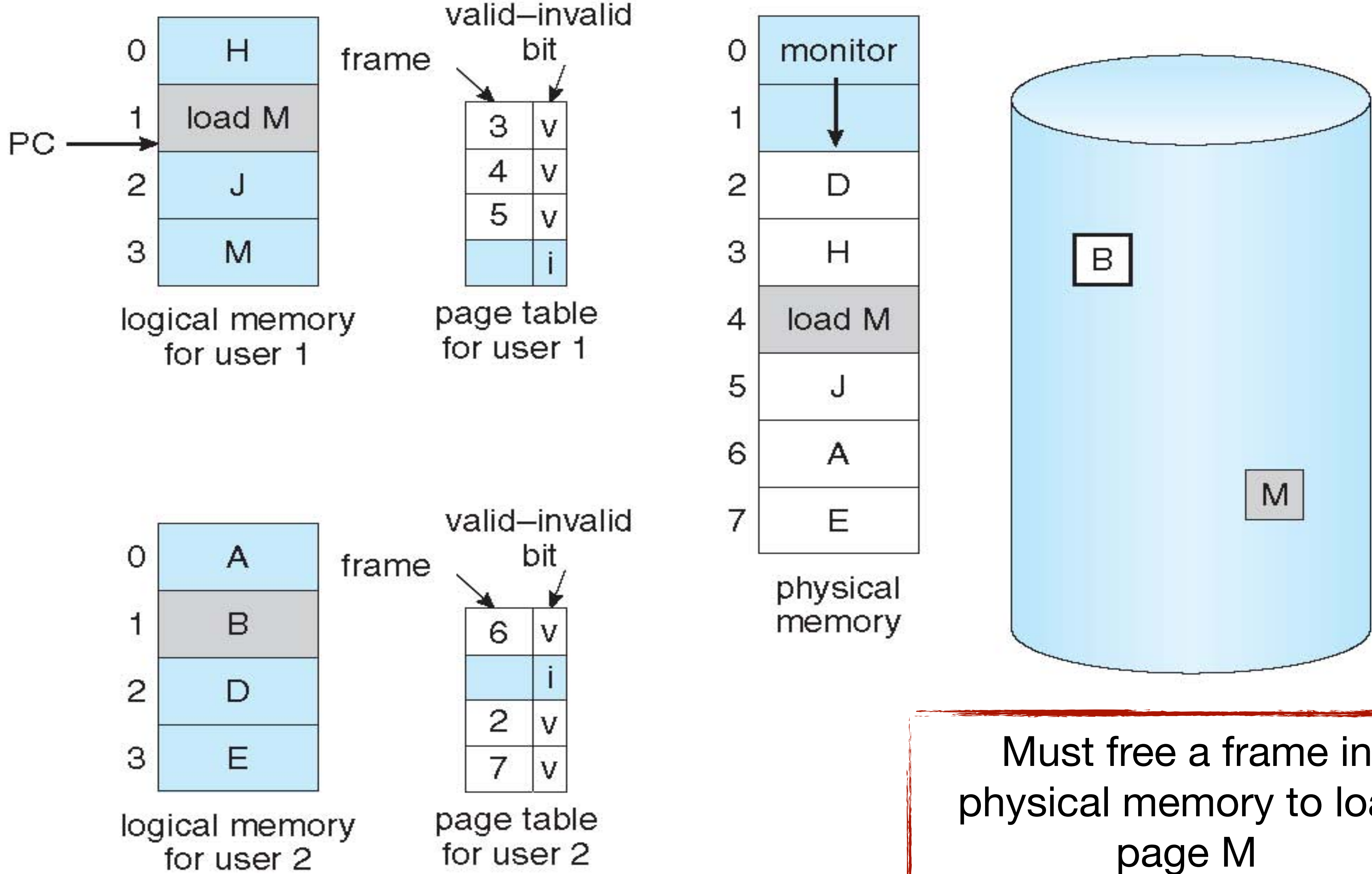
- **Page replacement – find some page in physical memory, but not really in use, page it out to swap space**
  - Once a page is in physical memory, that doesn't mean it will always be in physical memory
  - Same page may be brought into physical memory several times
  - For performance reasons, want a page replacement algorithm which will result in minimum number of page faults

Page Replacement

- **Must prevent over-allocation of physical memory by modifying page-fault service routine to include page replacement**
  - If no free frame exists, must select a victim frame and write the page that it contains off to swap space
  - After writing page to swap space, must update pages tables accordingly
  - Required page can then be read into newly freed frame
  - Continue process by restarting instruction that caused the page-fault

- **Use modify (dirty) bit to reduce overhead of page transfers**
  - Only modified pages are written back to disk from physical memory
  - Without modify bit, unmodified pages may be unnecessarily written back to swap space

Need For Page Replacement



Must free a frame in physical memory to load page M

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm determines**
  - How many frames to give each process
- **Page-replacement algorithm determines**
  - Which frames to replace
  - Want lowest page-fault rate on both first access and re-access
- **Evaluate algorithms by running them on a particular sequence of memory references (reference string) and computing the number of page faults on that sequence**
  - Reference string is just page numbers, not full addresses
    - Example reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
  - Repeated access to the same page does not cause a page fault
  - The more physical memory a system has, the more frames it has ... resulting in fewer page-faults

Page Replacement Algorithms

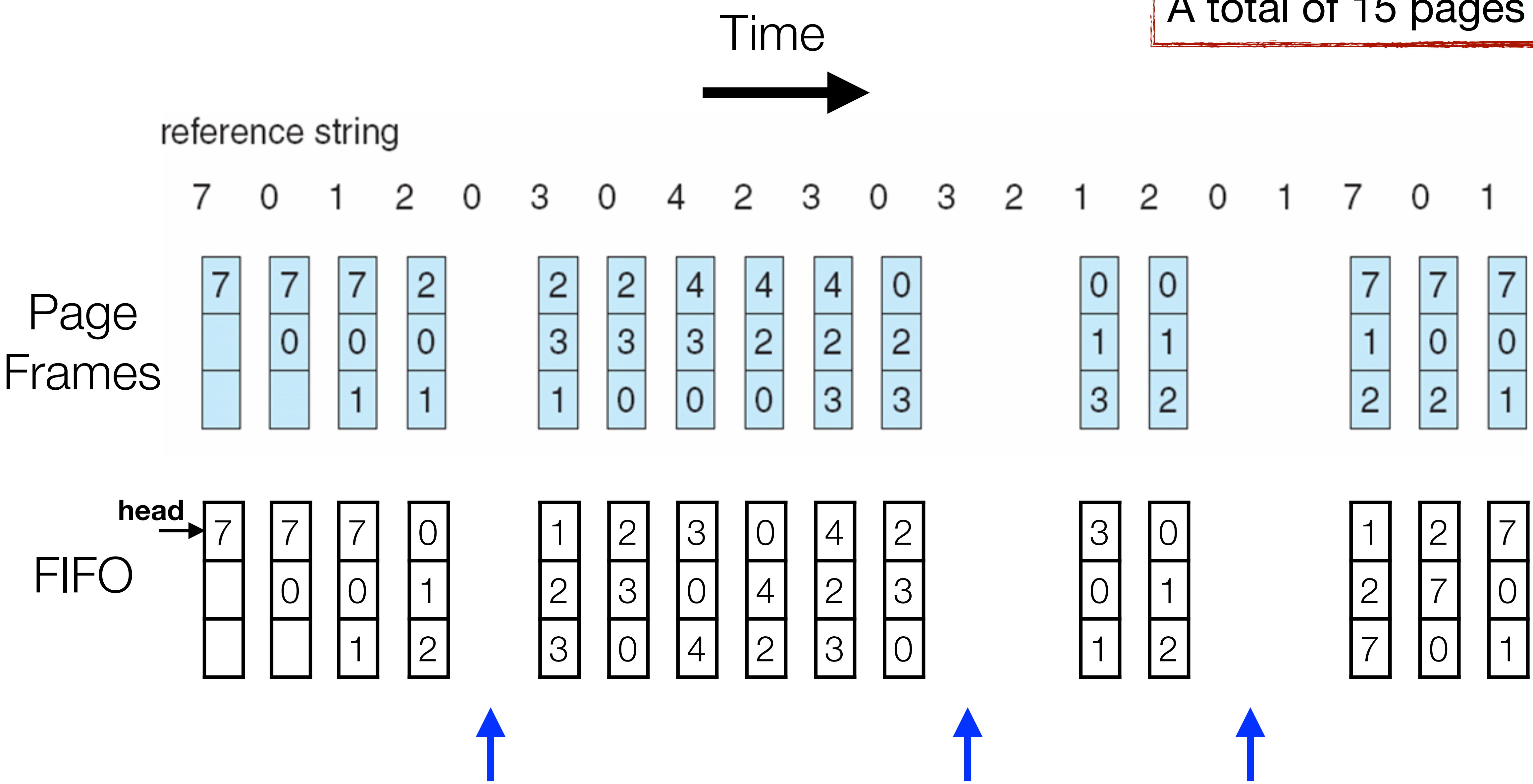
- **Many different page replacement algorithms exist**
  - FIFO Page Replacement
  - Optimal Page Replacement (theoretical best case)
  - Least-Recently Used (LRU) Page Replacement
  - Counting-Based Page Replacement

FIFO Page Replacement

- **A very simple page replacement algorithm**
  - When a page is brought into physical memory, insert a reference to that page into a FIFO
  - When a page must be replaced, replace the oldest page first (i.e. the page at the head of the FIFO)
- **Pros:**
  - Very easy to understand and program
- **Cons:**
  - Performance may not be very good
  - May result in a high number of page faults

FIFO Page Replacement Example

A total of 15 pages faults



no page fault when accessing page 0

no page fault when accessing pages 3 or 2

no page fault when accessing pages 0 or 1

Optimal Page Replacement

- **An optimal page replacement algorithm can be described very simply:**
  - *Replace the page that will not be used for longest period of time*

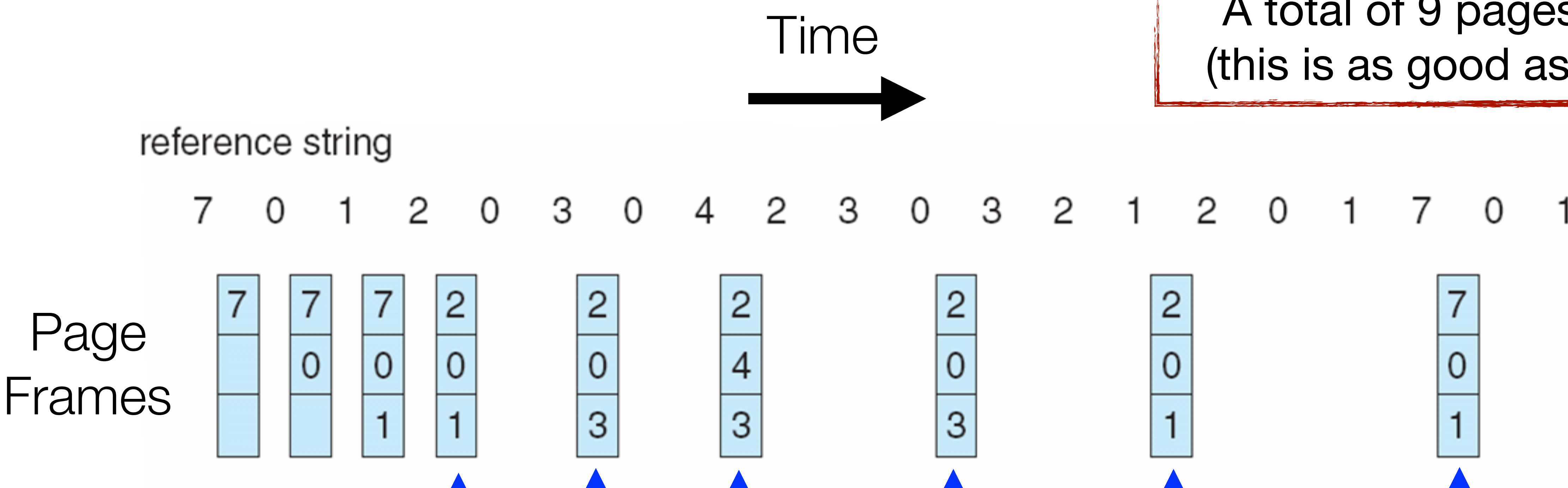
- **Guarantees the lowest possible page-fault rate for a fixed number of frames**

- **Sadly, it is not possible to know which page won't be used for the longest period of time (can't see the future)**

- **Optimal algorithm is still very useful**
  - Used for measuring how well other algorithm performs
  - How close are other algorithms to optimal?

Optimal Page Replacement Example

A total of 9 pages faults
(this is as good as it gets)



replaced page 7 as it will not be used again for longest time

replaced page 1 as it will not be used again for longest time

replaced page 0 as it will not be used again for longest time

replaced page 4, it will never be used again

replaced page 3, it will never be used again

replaced page 2, it will never be used again

Least Recently Used (LRU) Algorithm

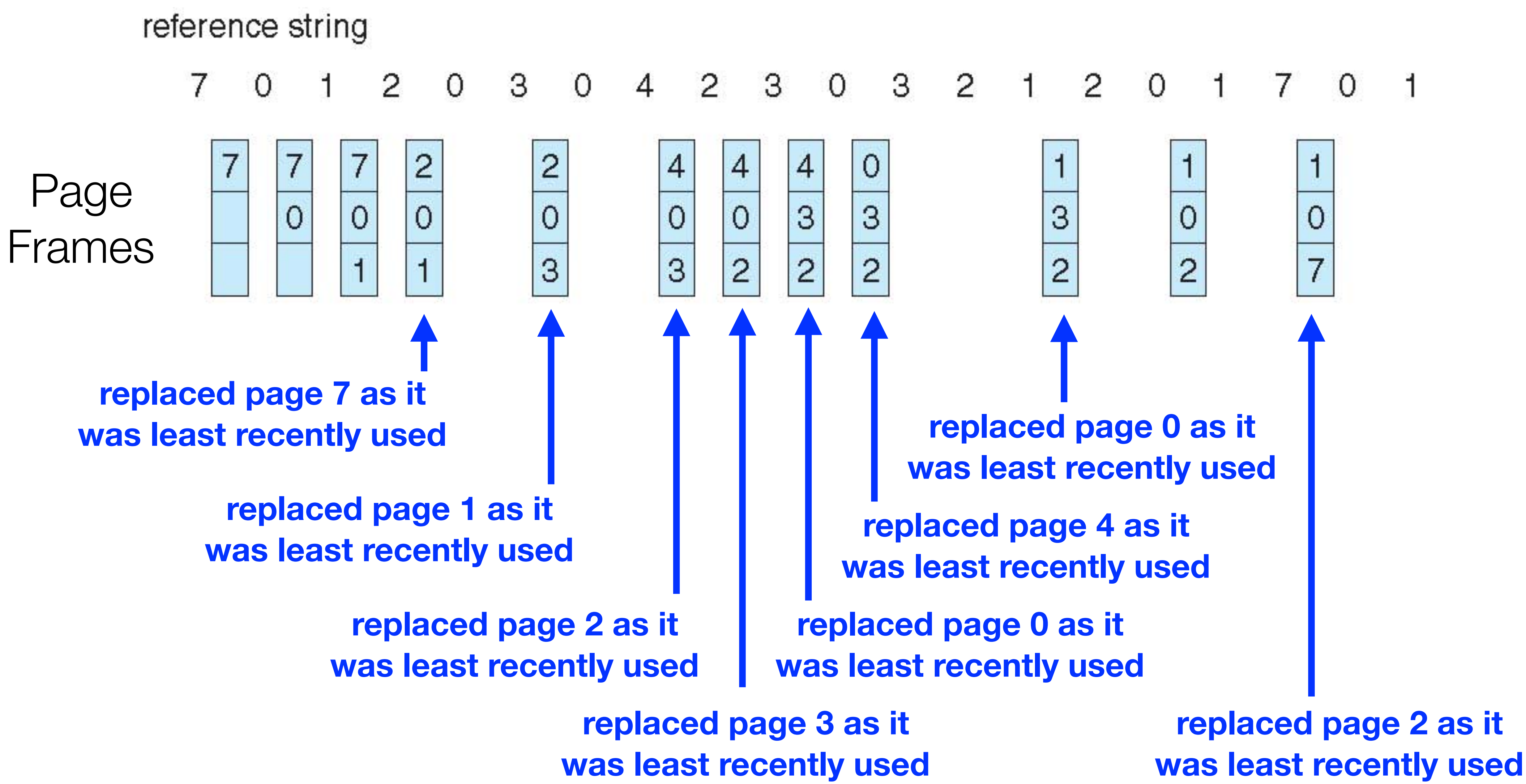
- **Interestingly, the same number of page-faults occur if a reference string is read either forwards or backwards and used to access pages**
  - Because of this property, it is possible to use past knowledge of page use rather than future knowledge

- **Least Recently Used (LRU) Replacement Algorithm:**
  - Replace page that has not been used in the most amount of time (i.e. replace the page that is the least recently used page)
    - Associate time of last use with each page

- **Generally good algorithm and used frequently**

Least Recently Used (LRU) Algorithm Example

A total of 12 pages faults;
better than FIFO, but not optimal

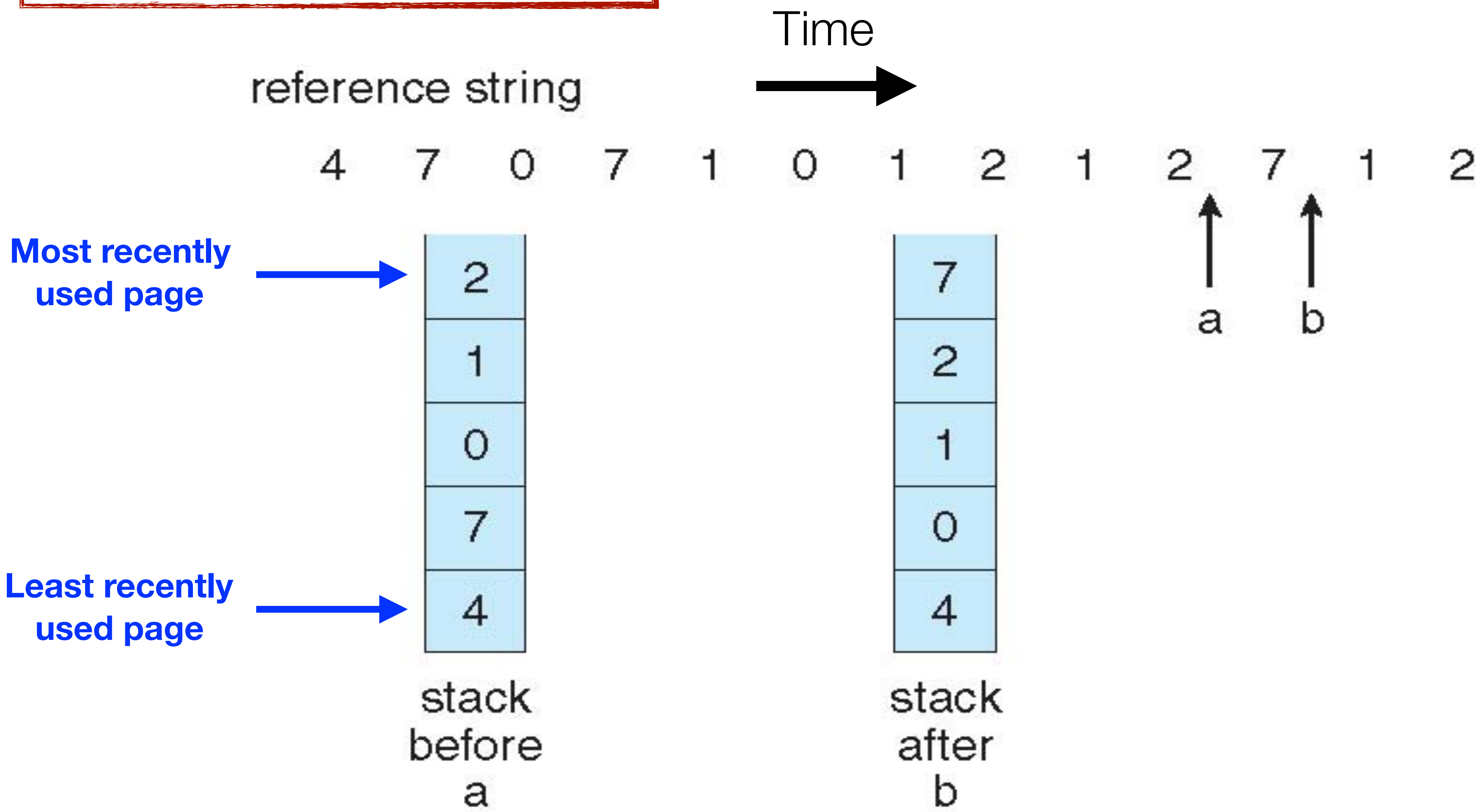


Least Recently Used (LRU) Algorithm

- **How should a least-recently-used algorithm be implemented?**
  - Option #1 - 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 replaced, look at the counters to find smallest value
      - Requires a potentially lengthy search through the page table (SLOW)
  - Option #2 - Stack implementation
    - Keep a stack of page numbers in a doubly linked list
    - If a page is referenced, move it to the top of the stack (not a pure stack implementation)
      - Most recently used page is always at the top of the stack; the least recently used page is always at the bottom of the stack
    - Finding and moving page numbers in the stack takes time (updates to the stack must be done on *every* memory reference) (SLOW)

Stack Implementation

Using a Stack to Record the Most Recent Page References



LRU Approximation Algorithms

- **The LRU algorithm as described previous is too slow, even with specialized hardware**
- **Instead of finding exactly which page was least recently used, an approximation will do**
- **One possible approach for approximating the least recently used page is to use a reference bit**
  - Associate a reference bit with each page, initially = 0
  - When page is referenced, set bit to 1
  - When it is time to replace a page, replace any that have reference bit = 0 (if one exists)
  - Can also use multiple reference bits to maintain a longer history and thus more closely approximate a true LRU algorithm

Counting-Based Page Replacement

- **Keep a counter of the number of references that have been made to each page**
  - Not common

- **Least-frequently-used (LFU) algorithm: replaces the page with smallest count**

- **Most-frequently-used (MFU) algorithm: replaces the page with the largest count**
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Speeding Up Page Replacement

- **Always maintain a pool of free frames**
  - No need to search for a free frame when the page-fault occurs
  - Load page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
    - No longer have to wait for victim frame to get paged out before new page can get paged in
- **Possibly, keep list of modified pages**
  - When backing store is otherwise idle, write pages and set to non-dirty
- **Possibly, keep free frame contents intact and note what is in them**
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Allocation of Frames

- **Each process needs some minimum number of frames that it cannot run without**
  - Even a single instruction may require more than a single frame of physical memory
- **Example: IBM 370 – 6 pages to handle SS MOVE instruction:**
  - Instruction is 6 bytes, and therefore might span 2 pages
  - 2 pages to handle for *from*
  - 2 pages to handle for *to*
- **Maximum number of frames that can be allocated is the total number of frames in the system**
- **Two major allocation frame schemes**
  - Fixed allocation
    - Equal allocation
    - Proportional allocation
  - Priority allocation

Types of Fixed Allocation

- **Equal allocation – every process in the system is allocated an equal share of the available frames**
  - If the number of frames is *m* and the number of processes is *n*, each process is allocated *m/n* frames
  - For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 100/5 = 20 frames
    - Maybe keep some as free frame buffer pool for faster paging
  - Pros: Easy to implement
  - Cons:
    - Why allocate 20 frames to a process that might actually only need 5?
    - Can be very wasteful
    - A higher priority process doesn't get any more frames than a lower priority process

Types of Fixed Allocation (Cont.)

- **Proportional allocation – Allocate available frames according to the size of a process**
  - Larger processes are allocated more frames than smaller processes
  - Dynamic as degree of multiprogramming change

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

$-S = \sum s_i$

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

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

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

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

  - Pros: Better allocation scheme than equal allocation
  - Cons: A higher priority process doesn’t get any more frames than a lower priority process

Priority Allocation

- **Use a proportional allocation scheme using process priorities rather than process size**

- **If process P_i generates a page fault,**
  - Select for replacement one of its frames (local replacement)
  - Select for replacement a frame from a process with lower priority number (global replacement)

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**
  - Process execution time can vary greatly due to another process stealing frames
  - Tends to have greater throughput (the number of processes that complete execution per unit time)
  - More common than local replacement

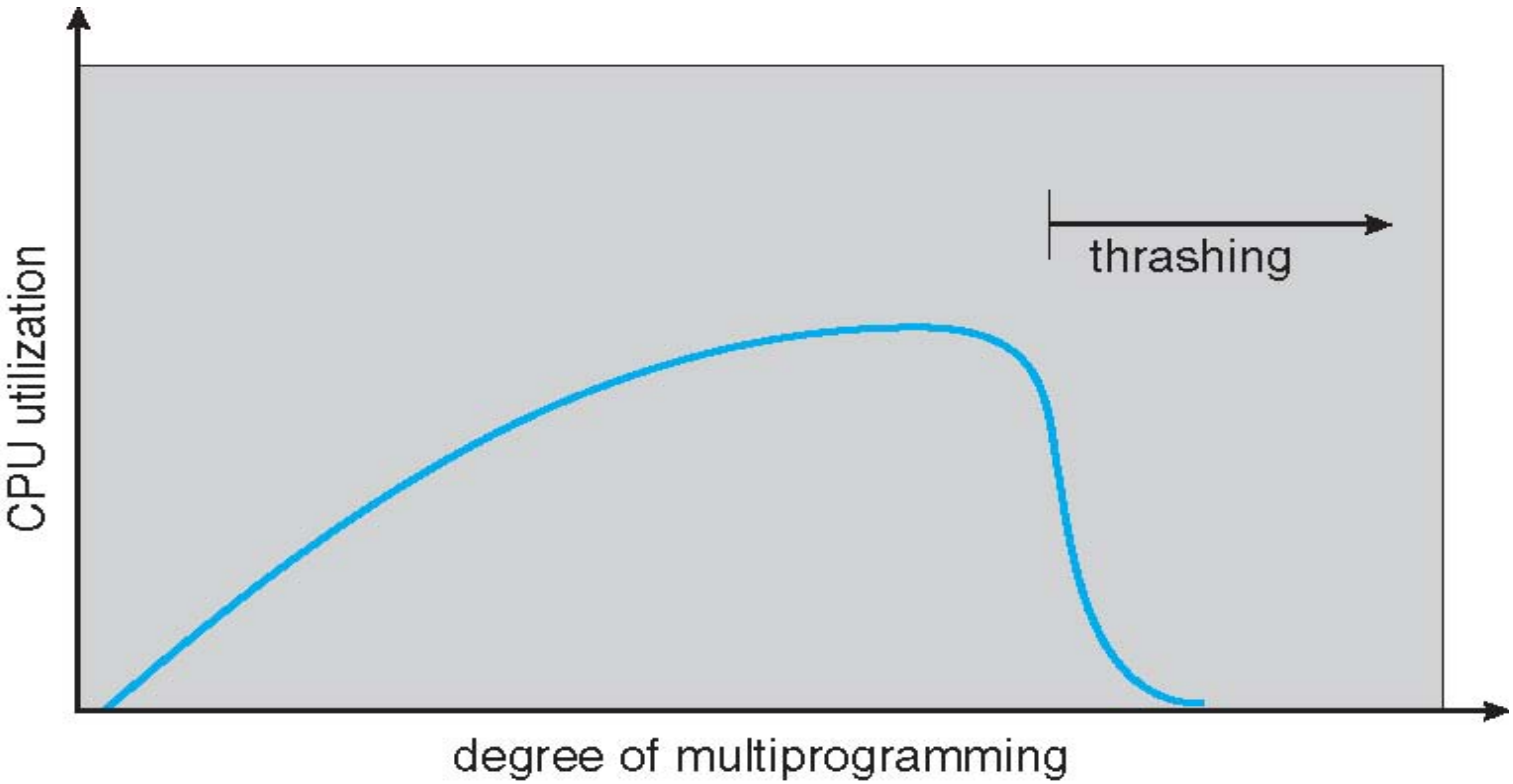
- **Local replacement – each process selects a replacement frame from only its own set of allocated frames**
  - Per-process performance is more consistent
  - But possibly underutilized memory

Thrashing

- **If a process does not have “enough” pages, the page-fault rate is very high**
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking 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 (spends more time paging than executing)**

Thrashing (Cont.)



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
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern