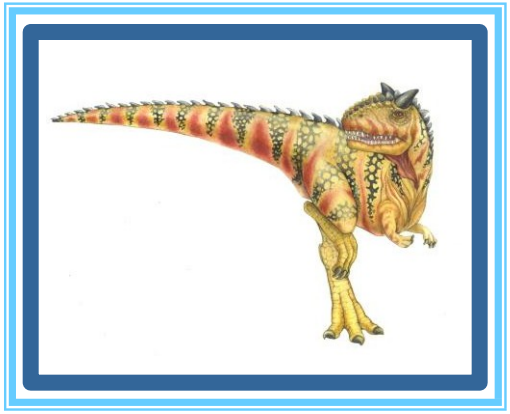
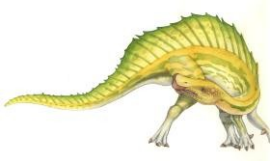
Chapter 9: Virtual Memory



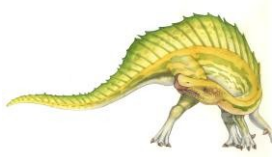
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



Objectives

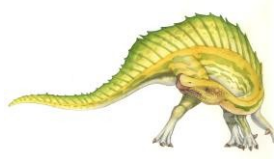
- To describe the benefits of a virtual memory system
  - **Goal of memory-management strategies**: keep many processes in main memory to allow multi-programming; see Chap-8
    - **Problem**: Entire processes must be in memory before they can execute
  - **Virtual Memory** technique: running process need not be in memory entirely
    - Programs can be larger than physical memory
    - Abstraction of main memory; need not concern with storage limitations
    - Allows easy sharing of files and memory
    - Provide efficient mechanism for process creation
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed



Background

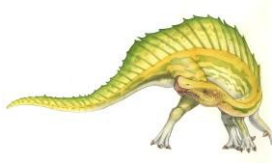
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures; are all seldom used
    - Ex: declared array of size 100 cells but only 10 cells are used

  Entire program code not needed (in main memory) at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running
    - Thus, more [partially-loaded] programs can run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time; more multi-programming and time-sharing
  - Less I/O needed to load or swap programs into/from memory
    - Thus, each user program would run faster



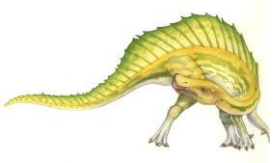
Background

- **Virtual memory** – separation of user logical memory from physical memory
  - As perceived by users; **that programs exist in contiguous memory**
  - Abstracts physical memory: need not worry about memory requirements
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
    - Programmers can work as if memory is an unlimited resource
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
    More programs running concurrently; increased multi-programming and/or time-sharing
  - Less I/O needed to load or swap processes; hence, faster program execution

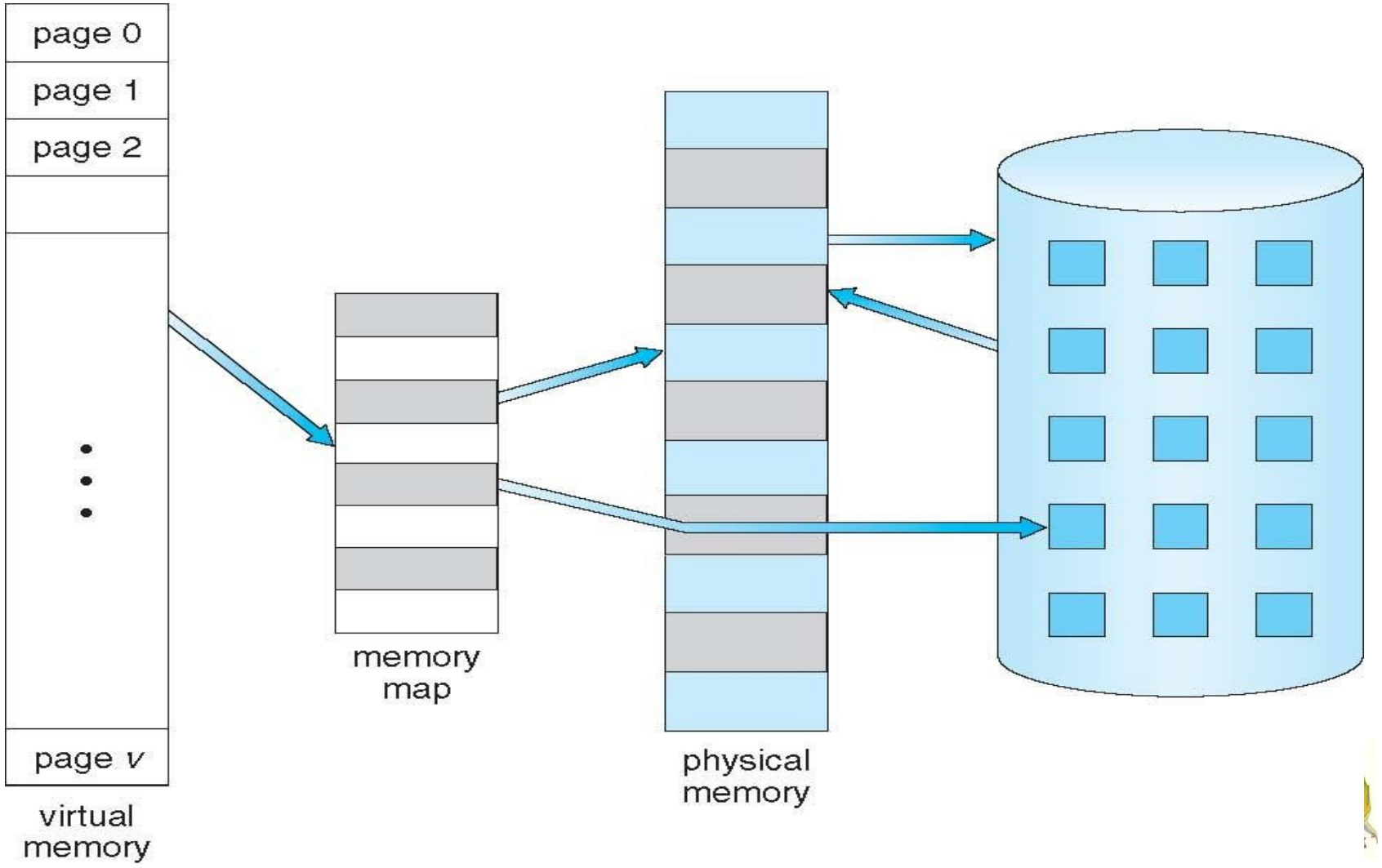


Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Process starts at address 0 with contiguous addresses until end of its address space
  - Meanwhile, physical memory organized in page frames; not contiguous (see Chap-8)
  - MMU maps logical pages to physical pages (i.e., frames) in memory
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

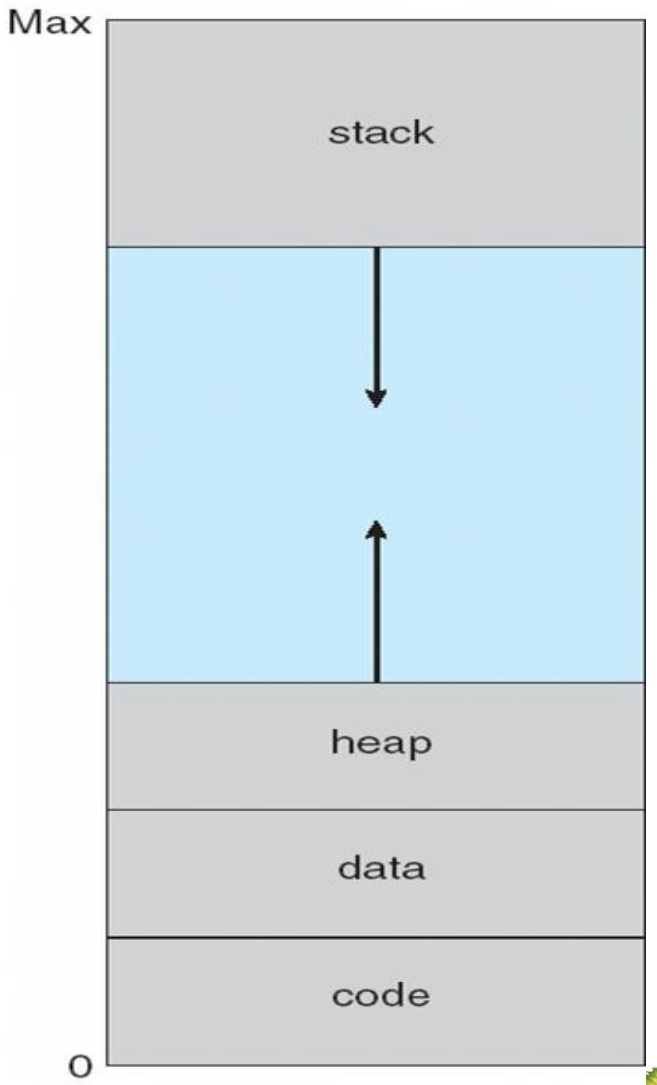


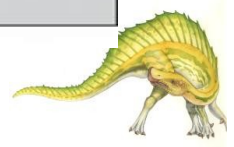
Virtual Memory That is Larger Than Physical Memory



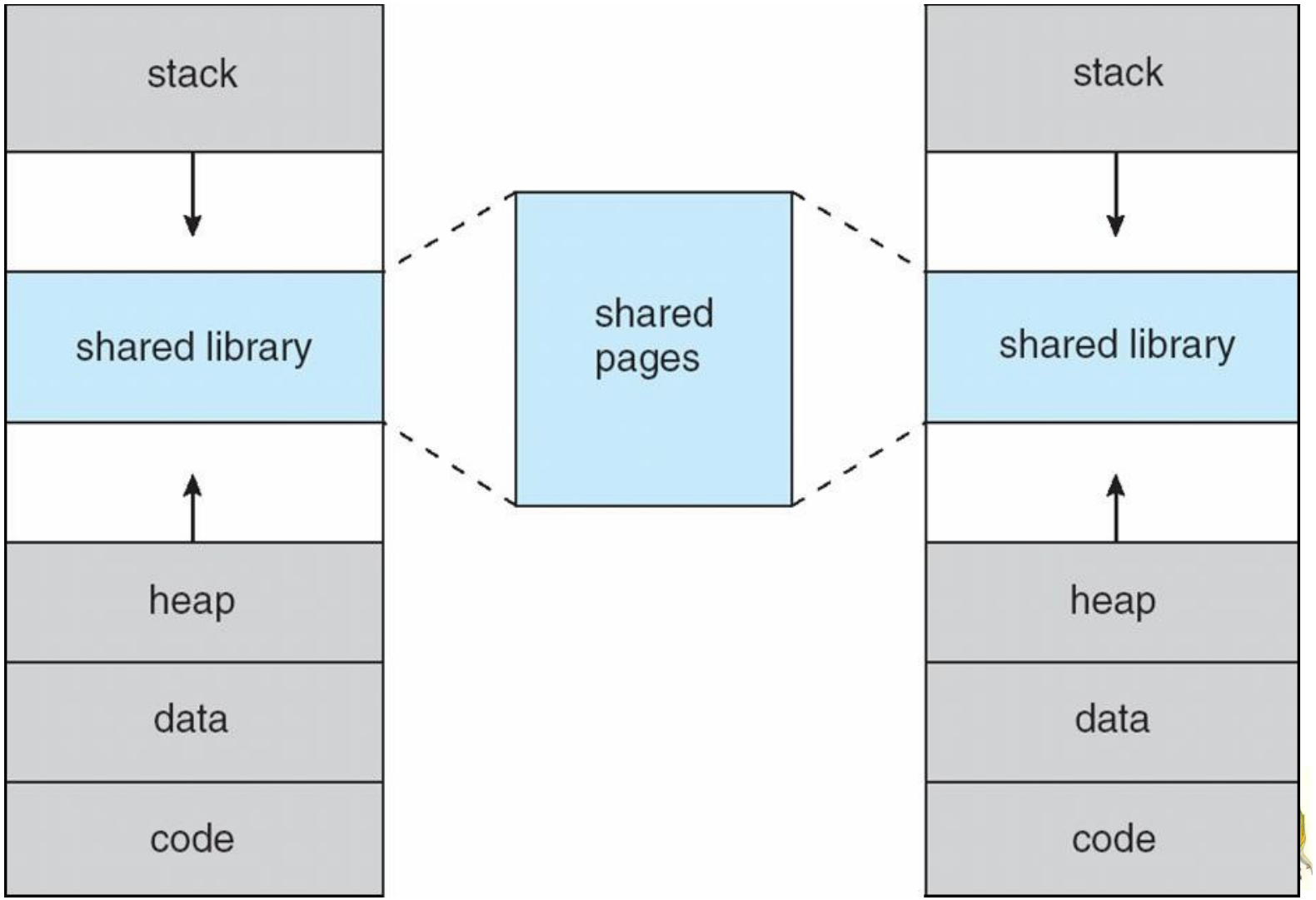
Virtual-Address Space of a Process

- For a process: heap grows upward while stack grows downward in memory; in process's space
  - Unused address space between the two is a hole; part of **virtual-address space**
    - Require actual physical pages only if the heap or the stack grow
  - Maximizes address space use
- Enables **sparse** address spaces with holes left for growth, or to dynamically link libraries, etc
- System libraries can be shared by many processes through mapping of the shared objects into virtual address space
- Processes can share memory by mapping read-write pages into virtual address space
- Pages can be shared during process creation with the `fork()`; speeding up process creation



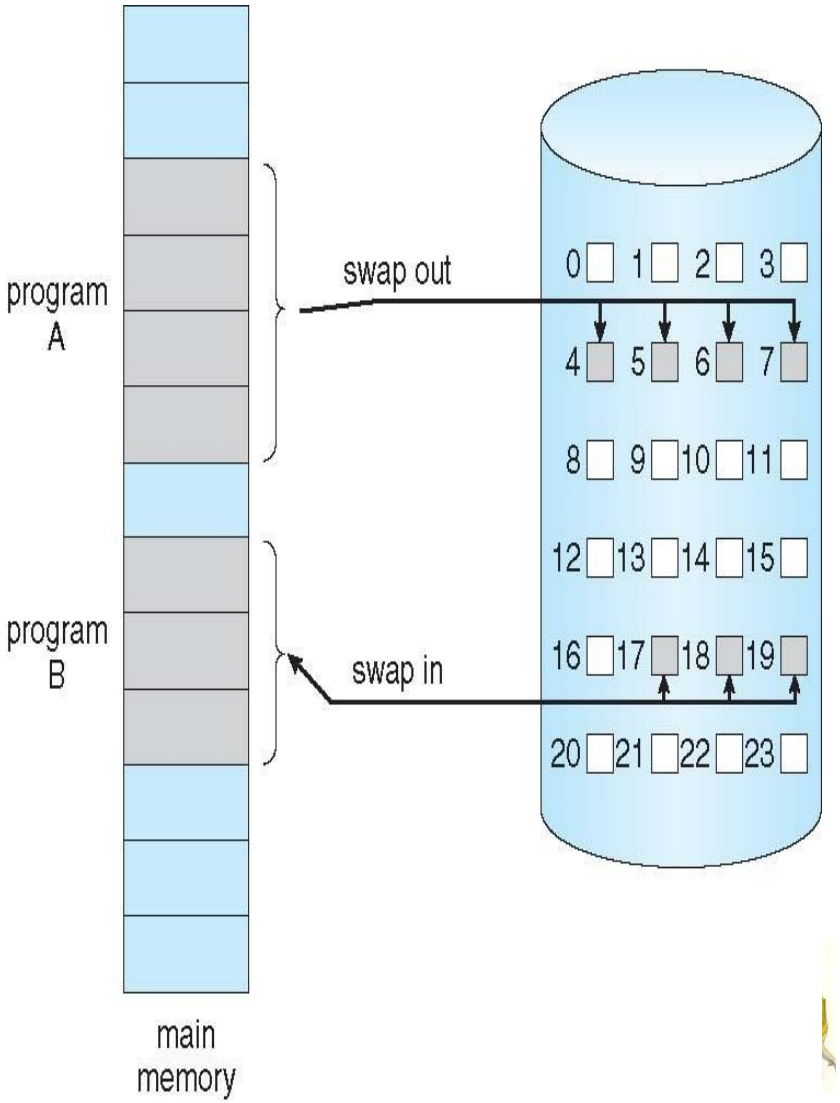


Shared Library Using Virtual Memory



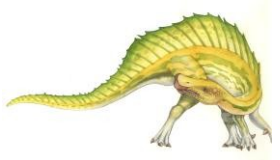
Demand Paging

- Consider how an executable program might be loaded from disk into memory.
- Could bring an entire process into memory at load time.
- Or bring a process's page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to a 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**
- Page is needed ⇒ reference it; see Slide-14
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory



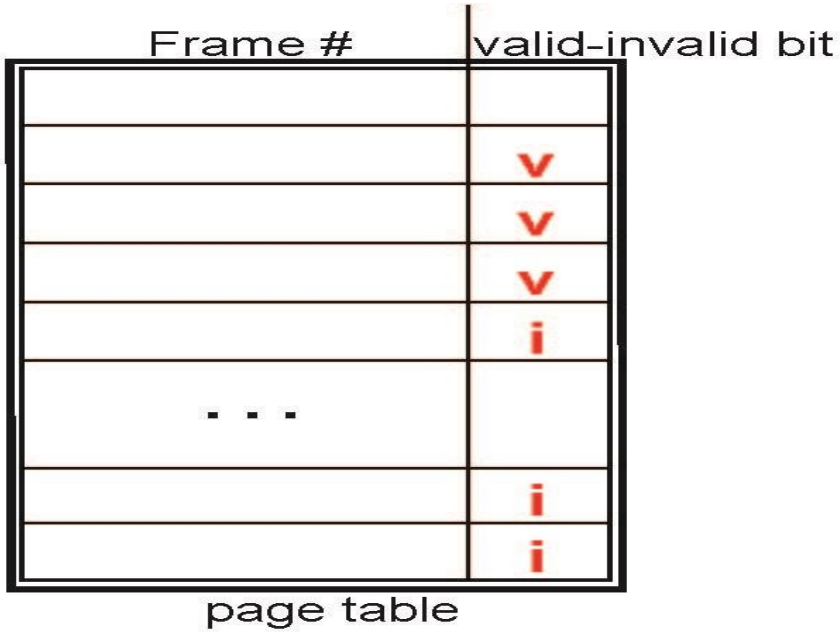
Basic Concepts

- When swapping in a process, the pager guesses which pages will be used before swapping it out again
  - The pager brings in only those **needed** pages into memory
    - Thus, decreases swap time and amount of needed physical memory
- Need new MMU hardware support to implement demand paging; see Slide-15
  - To distinguish between **in-memory** pages and **on-disk** pages
    - Uses the **valid—invalid scheme** of Slide-40 Chap-8
- If pages needed are already **memory resident**
  - Execution proceeds normally
- If page needed and is not memory resident; see Slide-14
  - Need to find the needed page from the disk and load it into memory
    - Without changing program behavior
    - Without programmer needing to change code

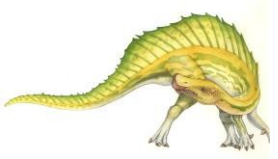


Valid-Invalid Bit

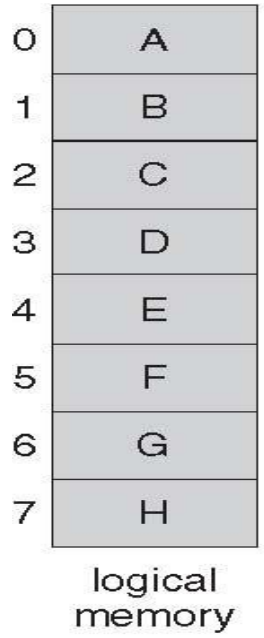
- A valid–invalid bit is associated with each page-table entry; see Chap-8, Slide-40
  (**v** ⇒ in-memory – **memory resident** ; **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

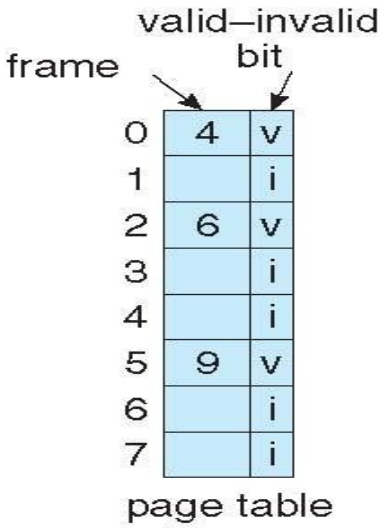


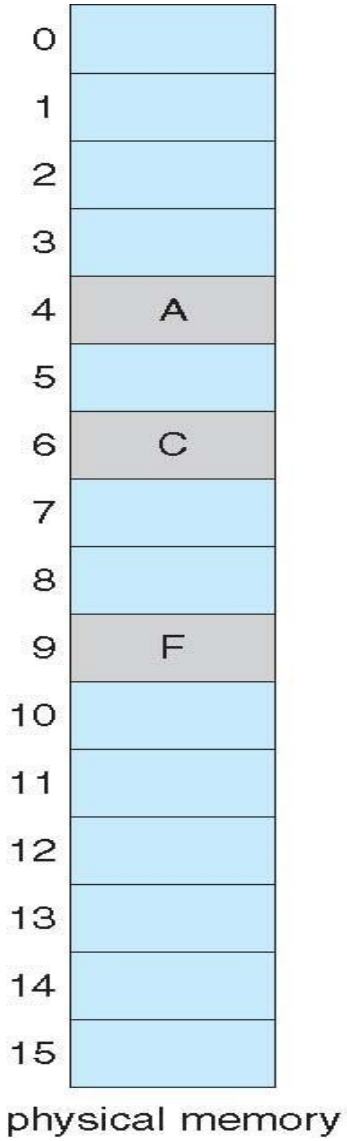
- During MMU address translation:
  - if valid–invalid bit in page-table entry is **i** ⇒ there is a page fault

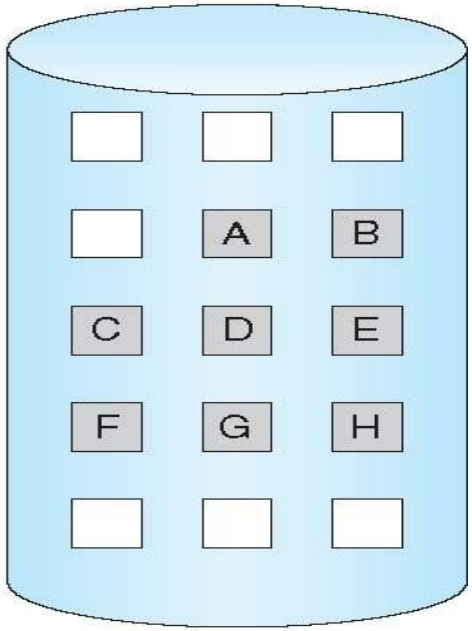


Page Table When Some Pages Are Not in Main Memory







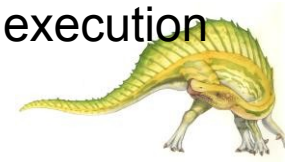


Page Fault

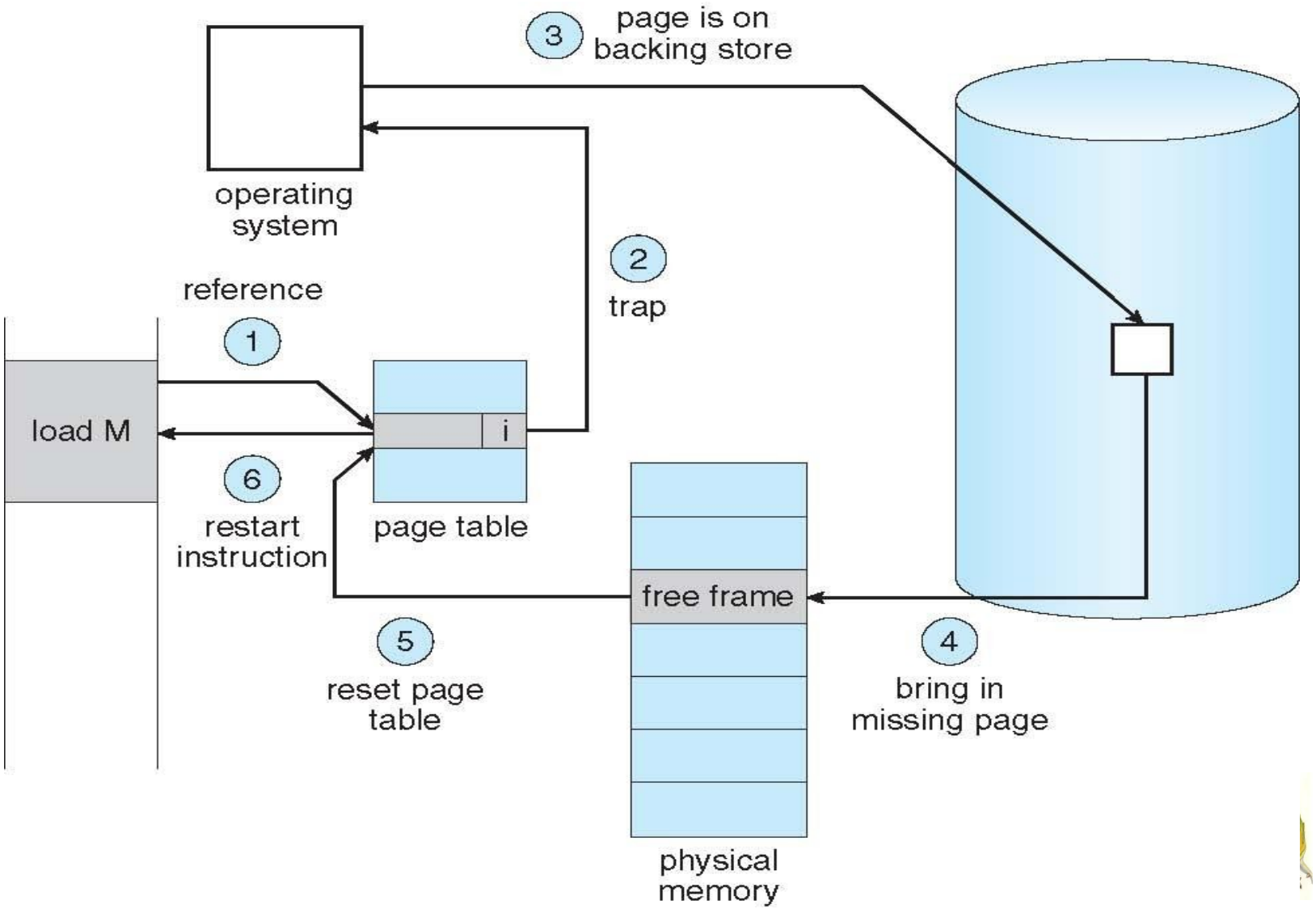
- What if the process refers to (i.e., tries to access) a page not in-memory ?
  - The [first] reference (i.e., address) to that **invalid** page will trap to operating system and causes a **page fault**

- **Procedure for handling a page fault**

1. OS checks an internal table to see if reference is valid or invalid memory access
2. If
   - Invalid reference ⇒ abort the process
     - address is not in logical address space of process
   - Just not in memory ⇒ page in the referred page from the disk
     - logical address is valid but page is simply not in-memory
3. Find a free frame; see Chap-8
4. Read the referred page into this allocated frame via scheduled disk operation
5. Update both internal table and page-table by setting validation bit = **v**
6. Restart the instruction that caused the page fault and resume process execution



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; logical address = (*p*, *d*)
    - Since page *p* is non-memory-resident then a page fault is issued
    - Page *p* is loaded and… same for all other process pages on first reference
  - This scheme is **pure demand paging**: load a page only when it is needed
- A given instruction may refer to multiple distinct pages; thus, multiple page faults
  - Consider fetching the instruction "`ADD A, B`" and fetching the values of data
    - `A` and `B` from memory and then storing the result back to memory
    - Addresses of "**`ADD A, B`**", "**`A`**", and "**`B`**" may all be in three different pages Multiple page fault per instruction results in unacceptable performance
    - Very unlikely, fortunately, due to **locality of reference**; see Slide-51
- Hardware support needed for demand paging; same as hardware for paging and swapping
  - **Page table** with valid / invalid bit, or special protection bits

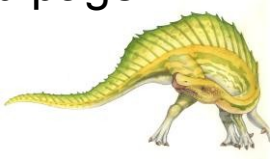
Performance of Demand Paging

- What is the **E**ffective **A**ccess **T**ime in demand paging? (worst case number of steps)

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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. 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/O subsystem (I/O 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 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

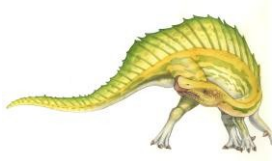
Performance of Demand Paging

- Not all steps (in Slide-18) are necessary in every case; **e.g., Step-6**
- **Three major components** of the page-fault service-time
  1. Service the interrupt; between 1 to 100 microseconds
     - Careful coding of the ISR means just several hundred instructions needed
  2. Read in the page – lots of time; **at least 8 milliseconds** + time in device-queue **+ …**
  3. Restart the process; between 1 to 100 microseconds – again, careful coding…
- Page Fault Rate $0 \le p \le 1$; p = probability of a page-fault and we expect $p \approx 0$
  - if $p = 0$ then there is no page faults
  - if $p = 1$ then every memory reference causes a page-fault
- **Effective Access Time (EAT)**
  - **EAT = [(1 – *p*) × memory_access_time] + [*p* ×page_fault_time]**
    - **page_fault_time =** page fault overhead + swap page out + swap page in



Demand Paging Example

- Memory access time = 200 nanoseconds; between 10 to 200ns in most computers
- Average page-fault service time = 8 milliseconds
- EAT = [(1 – *p*) × (200 nanoseconds)] + [*p* × (8 milliseconds)]
  = [(1 – *p*) × 200] + [*p* × 8,000,000] nanoseconds
  = 200 + 7,999,800*p* nanoseconds; thus, EAT is directly proportional to *p*
- If one access out of 1,000 causes a page fault, then
  - EAT = 8,199.8 nanoseconds = 8.2 microseconds.
    - This is a slowdown by a factor of 40!!;Because of demand paging
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 × *p*
  - 20 > 7,999,800 × *p*
  - Thus, we must have *p* < .0000025
  - That is, to keep slowdown due to demand paging
    - *p* < one page fault in every 399,990 memory accesses



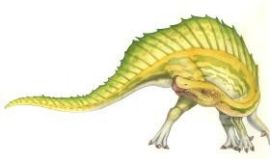
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

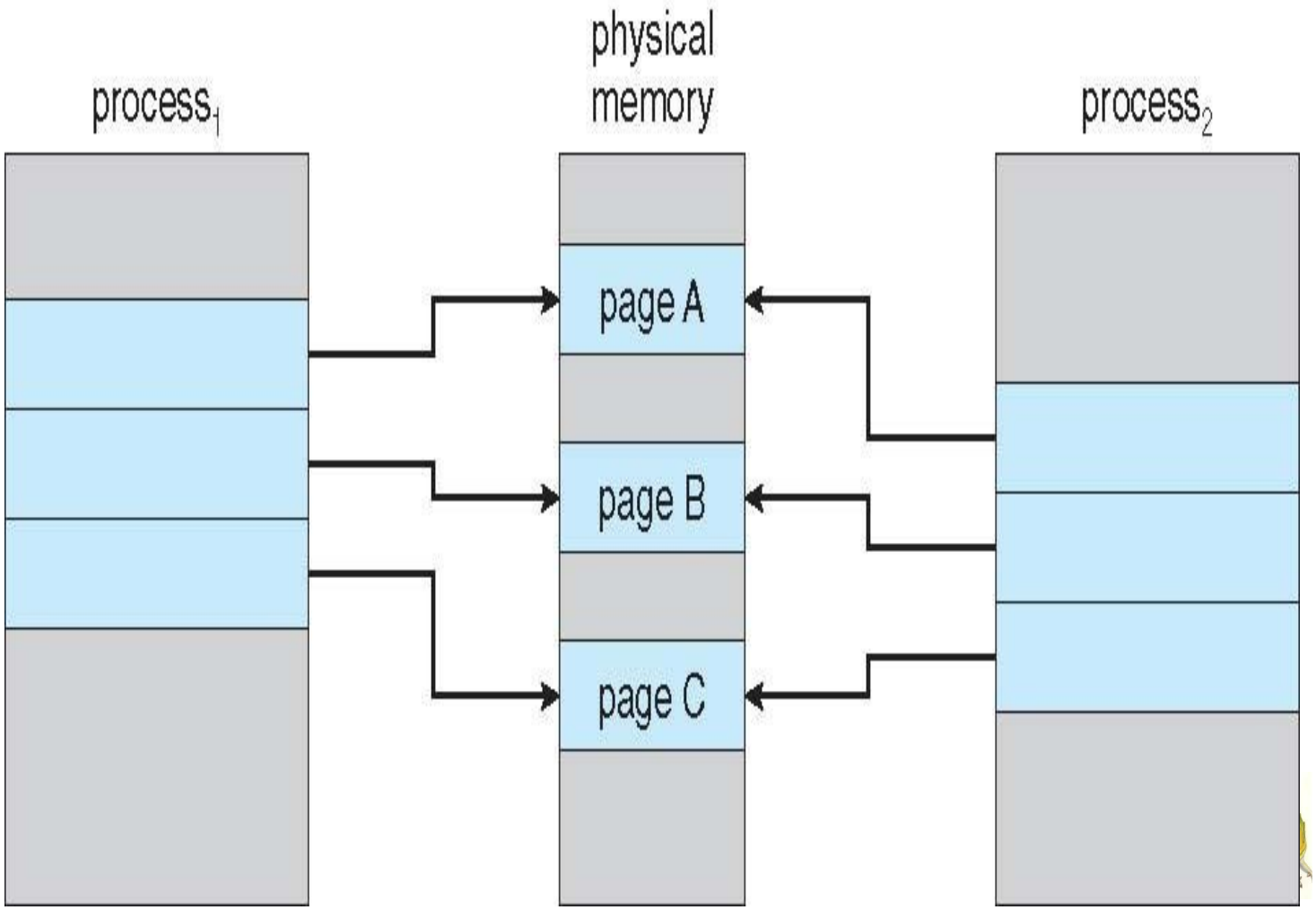
Copy-on-Write

- **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

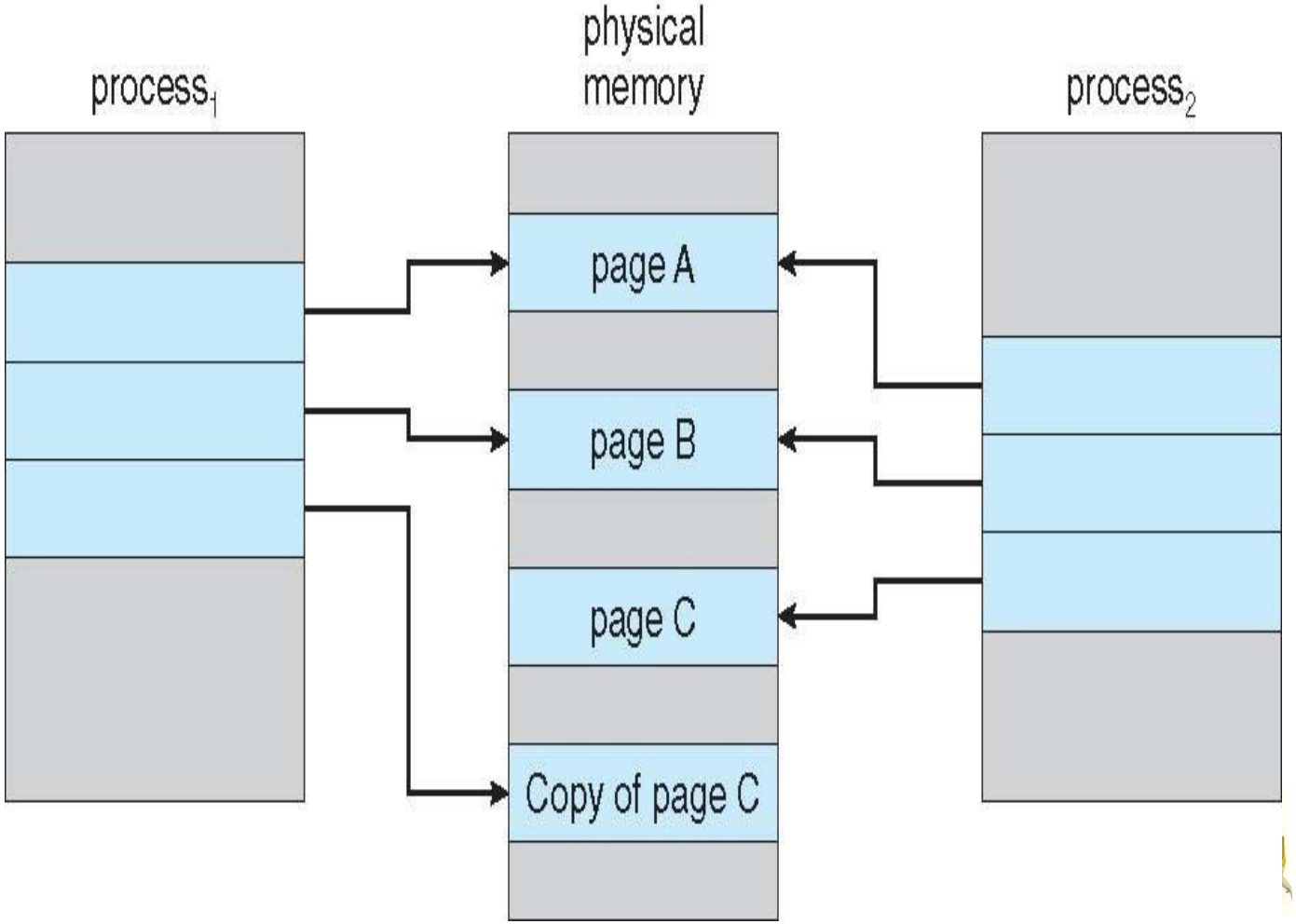
  COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspended and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient



Before Process 1 Modifies Page C

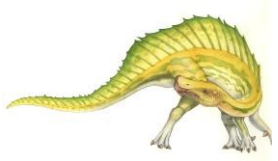


After Process 1 Modifies Page C

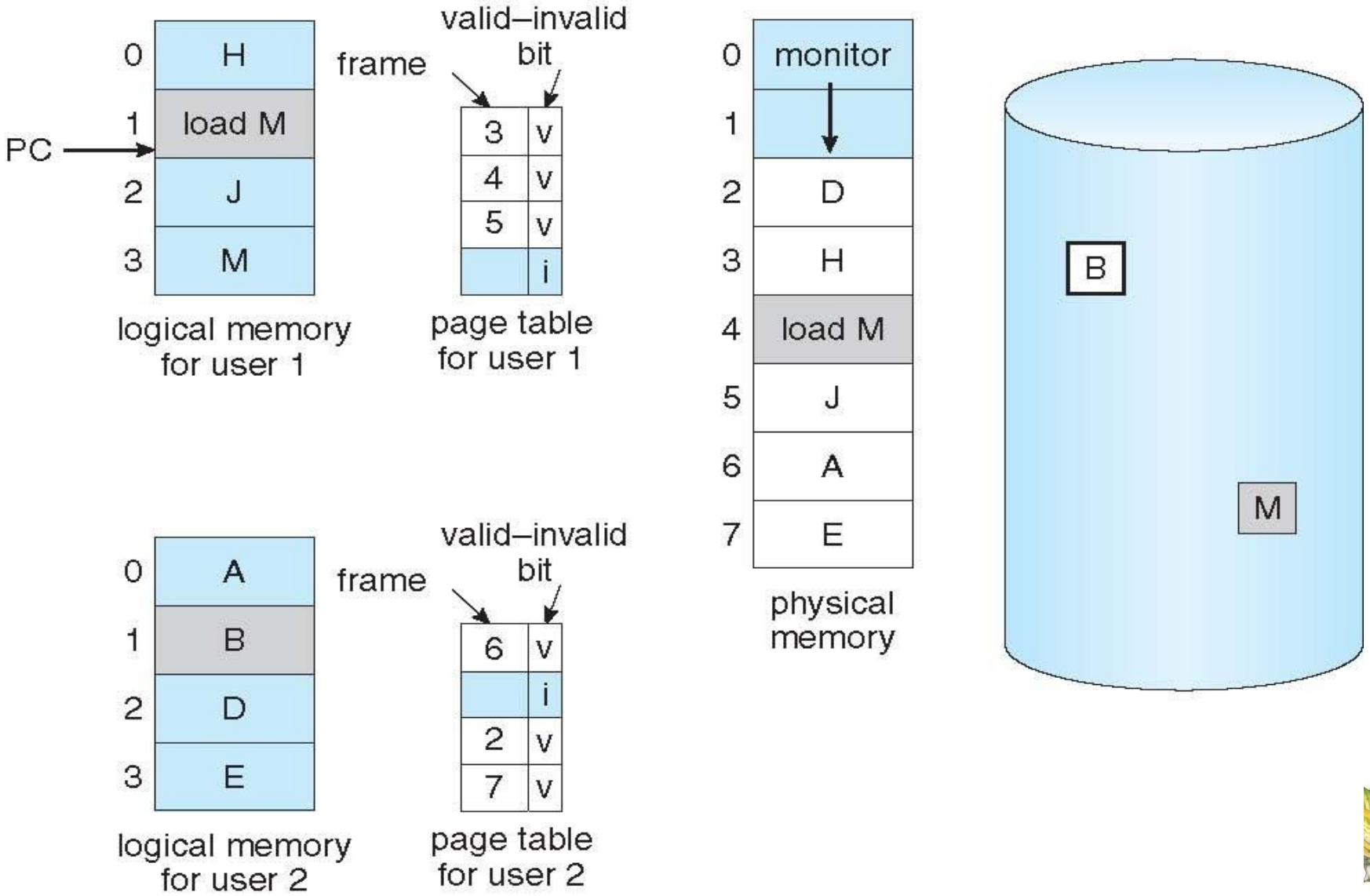


What Happens if There is no Free Frame?

- Many pages need to be loaded but not enough free frames available fro them
  - Memory is being used up by process pages; both, user and kernel processes
  - Also memory is in demand from the kernel, I/O buffers, etc
- How much memory to allocate to I/O buffers, kernel, processes, … , etc
- **Solution**: Page replacement; when paging in pages of a process but no free frames
  - Terminate the process? Big fat **no**
  - Swap out some process? **Yes**, but not always a good option
  - Find currently un-used frame to free it; Page it out and page in process page
    - Replacing the un-used memory page with the new page
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times



Need For Page Replacement



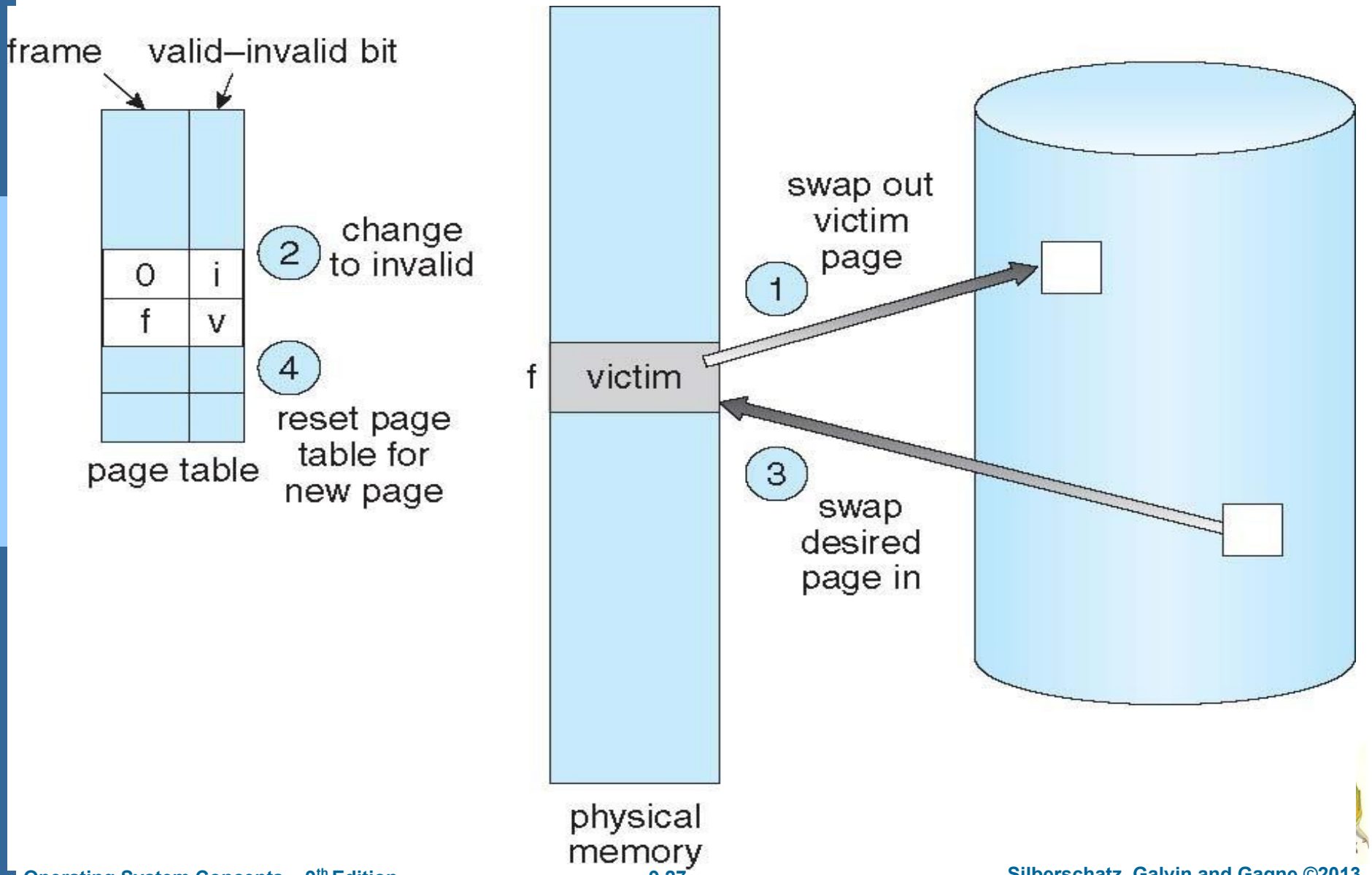
Basic Page Replacement Algorithm

- The page-fault service routine is modified to include page replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   1. If there is a free frame, use it
   2. If there is no free frame, use a page-replacement algorithm to select a **victim frame**
   3. Write the victim frame to the disk [if dirty]; change the page and the frame tables accordingly
4. Read the desired page into the newly freed frame; change the page and frame tables
5. Continue the user process from where the page fault occurred

- We have potentially ***two*** page transfers to do – increasing EAT
  - Only if no frames are free; one page in required and one page out required

Page Replacement



Page Replacement …

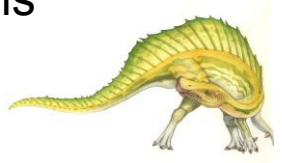
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk ; **see Slide-28**
  - Each page or frame is associated with a **modify bit**
  - Set by the hardware whenever a page is modified
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
  - **A user process of 20 pages can be executed in 10 frames** simply by using demand-paging and using a page-replacement algorithm to find a free frame whenever necessary

Page- Replacement and Frame-Allocation Algorithms

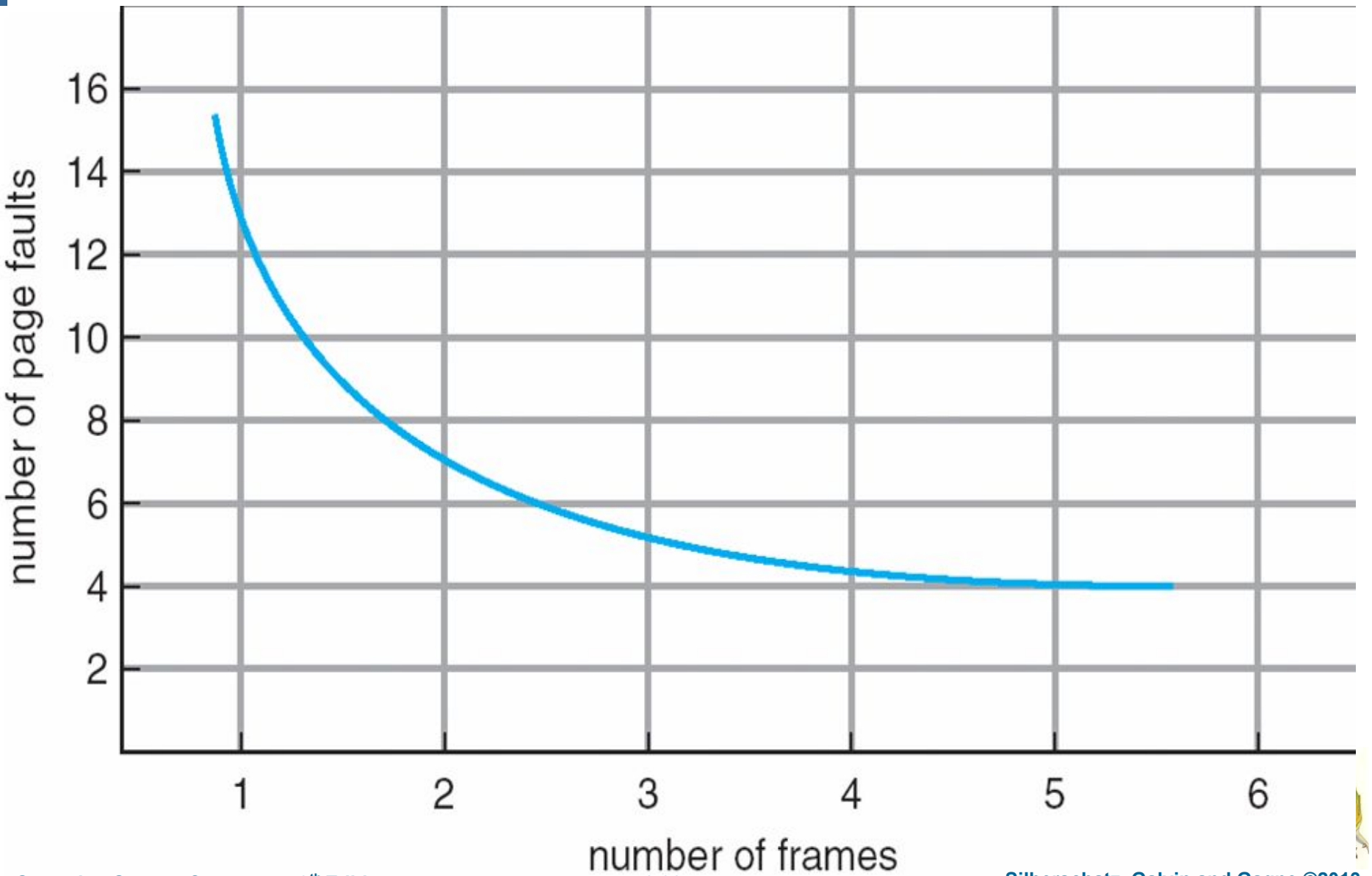
- Two major demand-paging problems : **frame allocation** and **page replacement**
- **Frame-allocation algorithm** determines
  - How many frames to allocate to each process
  - Which frames to replace; when page replacement is required
- **Page-replacement algorithm**
  - We want an algorithm which yields the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (the **reference string**) and computing the number of page faults on that string
  - String is just page numbers ***p***, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- 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

  for a memory with three frames



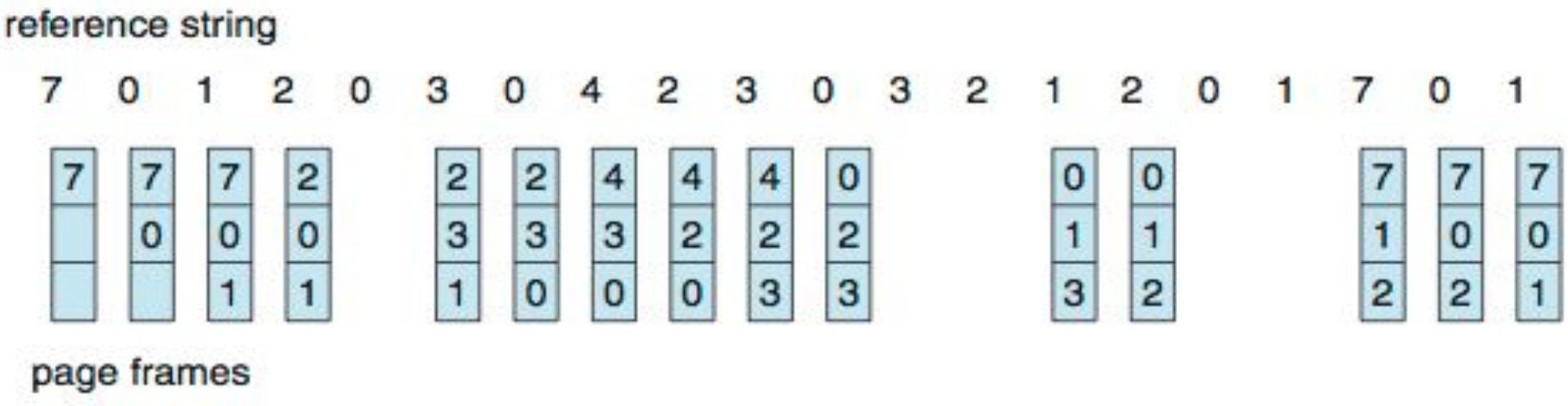
Graph of Page Faults Versus The Number of Frames



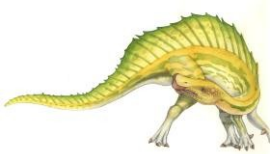
FIFO Page Replacement Algorithm

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Each page brought into memory is also inserted into a **first-in first-out queue**
  - Page to be replaced is the **oldest** page; the one at the head of the queue
  - Our example yields 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**

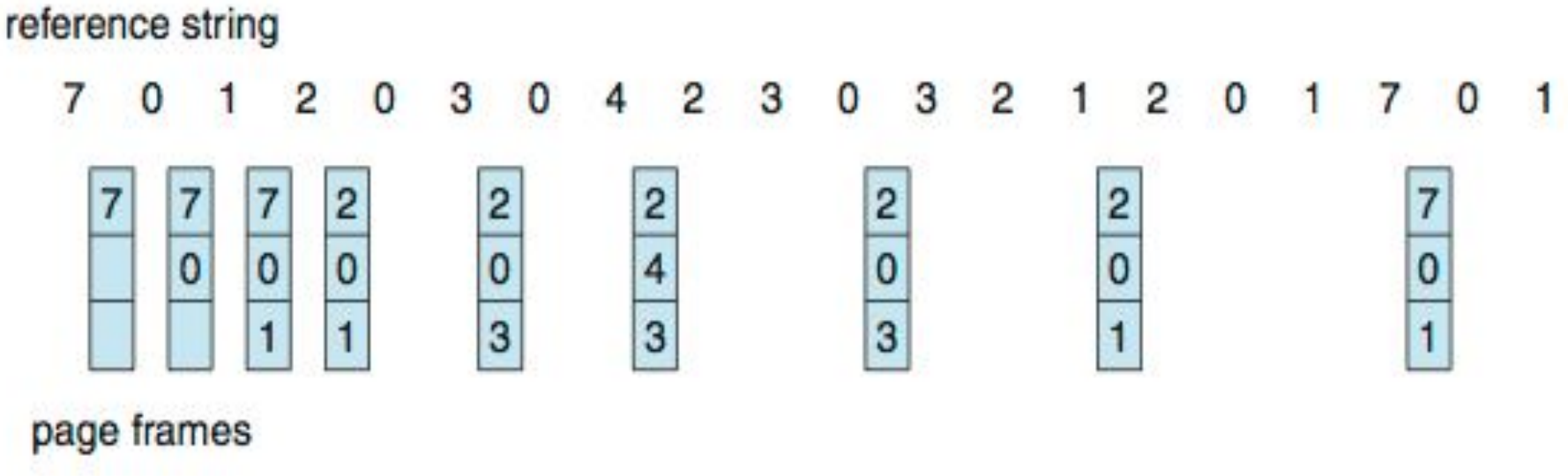


FIFO Illustrating Belady's Anomaly

Optimal Page Replacement 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 and Memory = 3 frames]

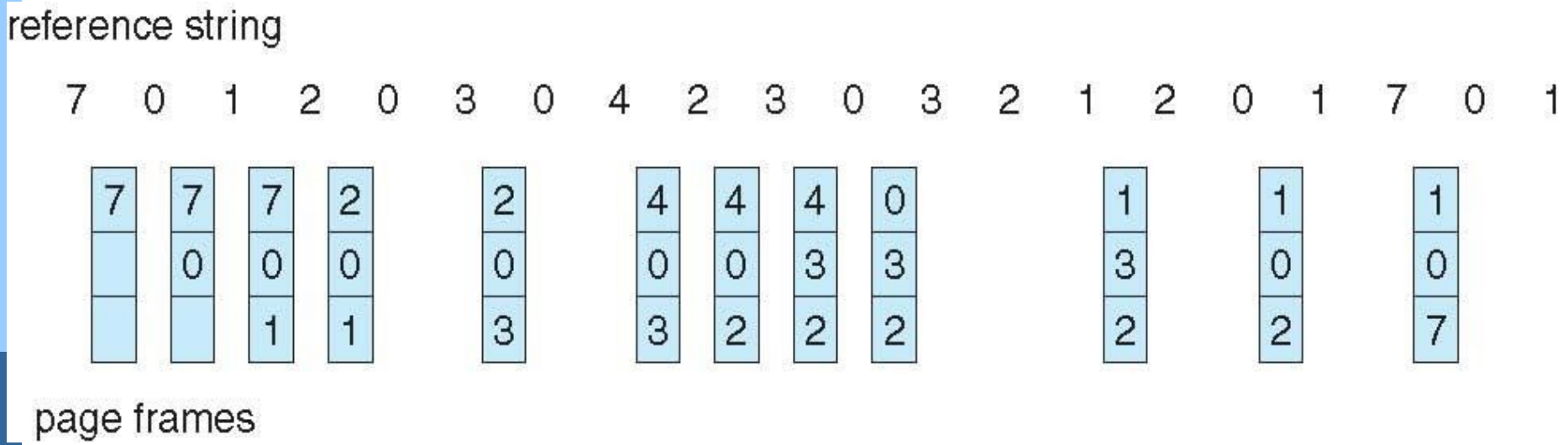
- Replace the page that will not be used for longest period of time
- Our example yields 9 page faults
- Unfortunately, OPR is **not feasible** to implement
  - Because: we can't know the future; i.e., what is the next page?
    - We have assumed that we know the reference string. No, we don't
- OPR is used only for comparing with new algorithms; **how close to the optimal?**



LRU Page Replacement 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 and Memory = 3 frames]

- Use the recent past as an approximation of the near future
- Replace the page that *has not been used* for the longest period of time
  - That is, the **least recently used** page
- Associate time of last use with each page



- Our example yields 12 page faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- Algorithm is feasible but not easy to implement.
  - LRU algorithm may require substantial hardware support

LRU Algorithm

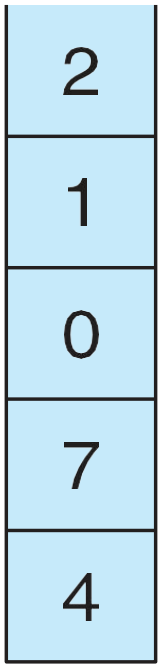
- **Counter implementation**
  - Each page-table entry has a counter; every time the page is referenced through this entry, copy the current clock value into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through the page-table needed; to find the LRU page
- **Stack implementation**
  - Keep a stack of page numbers in a double link form, with head and tail pointers:
  - Whenever a page is referenced:
    - move it to the top; most recently used page is always at the top of stack
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement; as LRU page is always at the bottom of the stack
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use of A Stack to Record Most Recent Page References

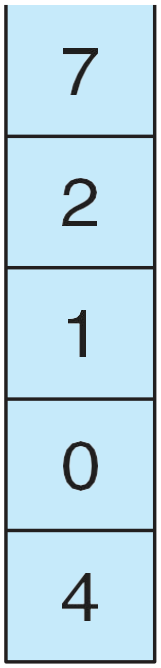
reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

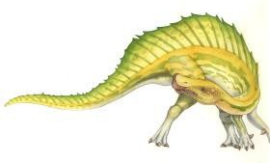
a b



stack before a

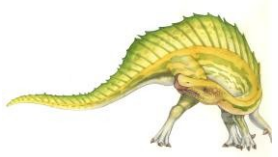


stack after b

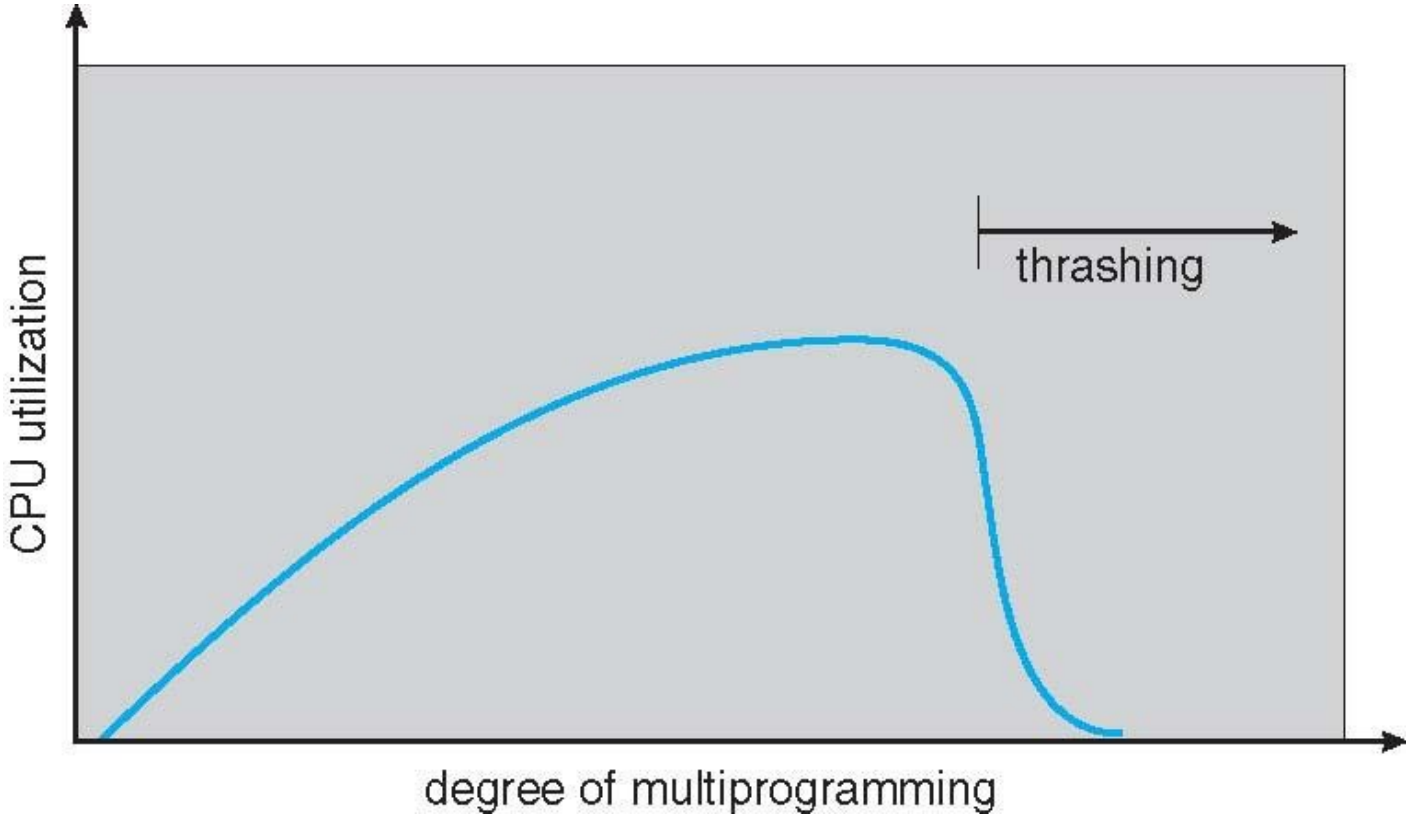


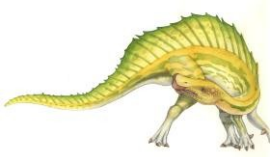
Thrashing

- 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



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

End of Chapter 9