OPERATING SYSTEMS

Memory Management - 8

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

Course Syllabus - Unit 3

Unit-3:Unit 3: Memory Management: Main Memory

Hardware and control structures, OS support, Address translation, Swapping, Memory Allocation (Partitioning, relocation), Fragmentation, Segmentation, Paging, TLBs context switches

Virtual Memory - Demand Paging, Copy-on-Write, Page replacement policy - LRU (in comparison with FIFO & Optimal), Thrashing, design alternatives - inverted page tables, bigger pages.

Case Study: Linux/Windows Memory

Course Outline

25	Main Memory: Hardware and control structures, OS support, Address translation	8.1	64.2
26	Dynamic linking, Swapping	8.2	
27	Memory Allocation (Partitioning, relocation), Fragmentation	8.3	
28	Segmentation	8.4	
29	Paging: OS Support, TLBs, Address Translation	8.5	
30	Structure of the Page Table	8.6	
31	Design Alternatives - Inverted Page Tables, Bigger Pages	8.7-8.8	
32	Virtual Memory: Demand Paging, Copy-OnWrite	9.1-9.3	
33	Page replacement policy - LRU	9.4	
34	FIFO & Optimal	9.5	
35	Thrashing	9.6	
36	Case Study: Linux/ Windows Memory Management	9.10	

Topic Outline

- Virtual Memory - Background
- Virtual Memory that is Larger Than Physical Memory
- Virtual Address Space
- Shared Library Using Virtual Memory
- Demand Paging

Topic Outline

- Swapping - Basic Concepts
- Valid-Invalid Bit
- Page Table When Some Pages Are Not in Main Memory
- Page Fault
- Steps in Handling a Page Fault

Topic Outline

- Aspects of Demand Paging
- Performance of Demand Paging
- Demand Paging Example
- Demand Paging Optimizations
- Copy-on-Write

Virtual Memory - Background

- Code needs to be in memory to execute, but entire program rarely used
    - Error code, unusual routines, large data structures
- Entire program code not needed at 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 -> more programs run at the same time
        - Increased CPU utilization and throughput with no increase in response time or turnaround time

Virtual 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
  - Allows 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

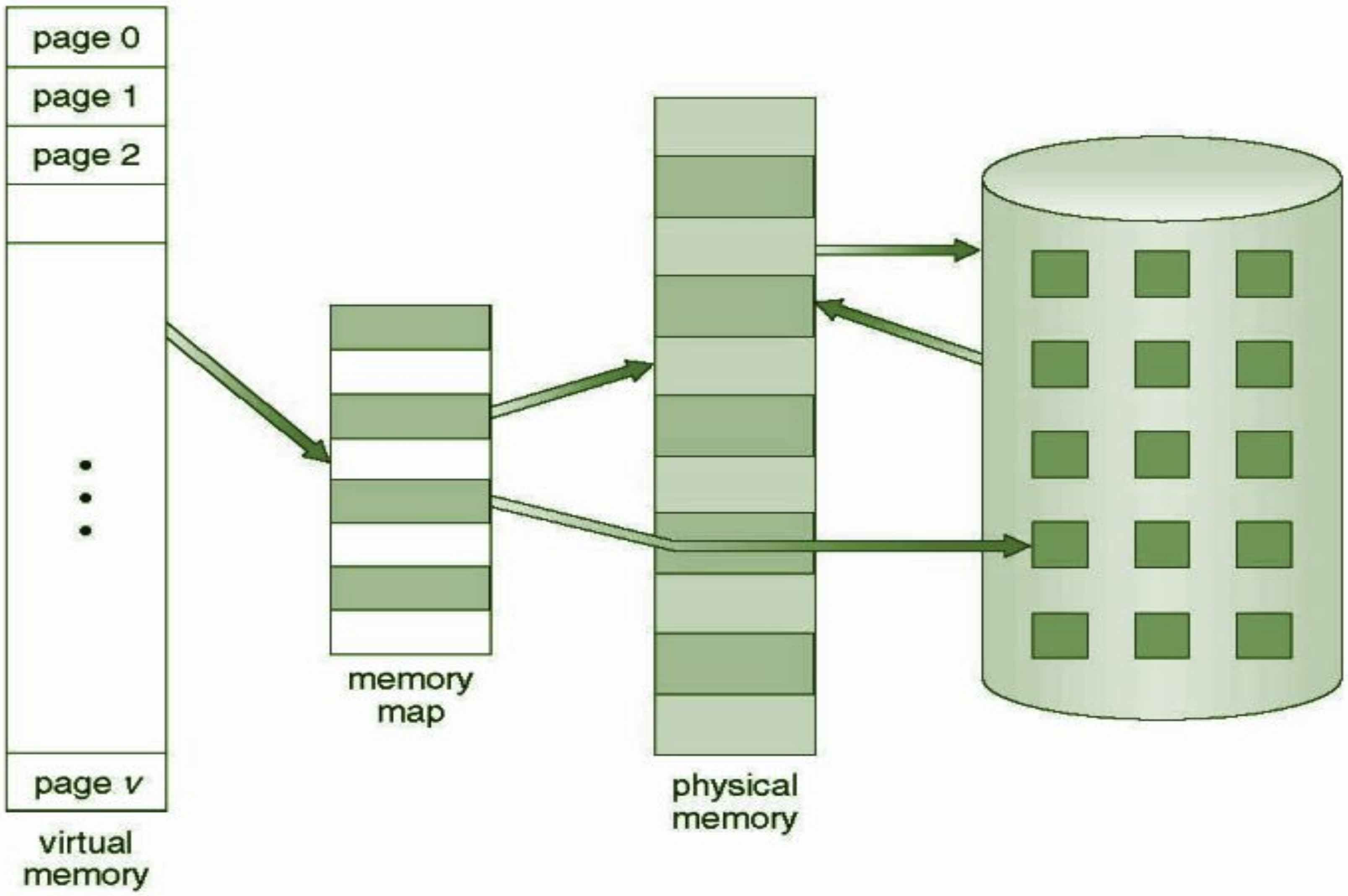
Virtual Memory - Background

- Virtual address space => logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

Virtual Memory - Background

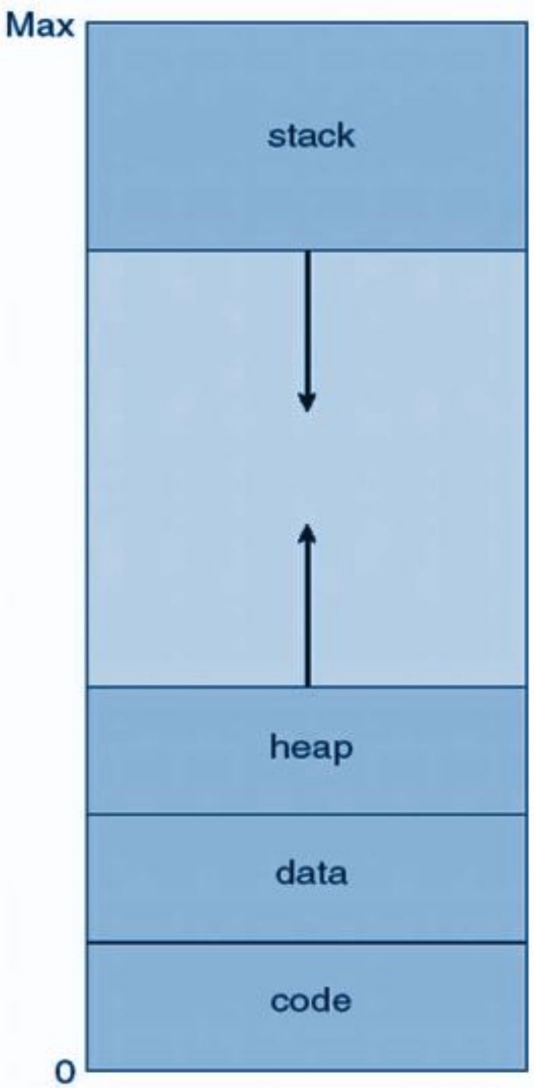
- Virtual memory can be implemented using:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



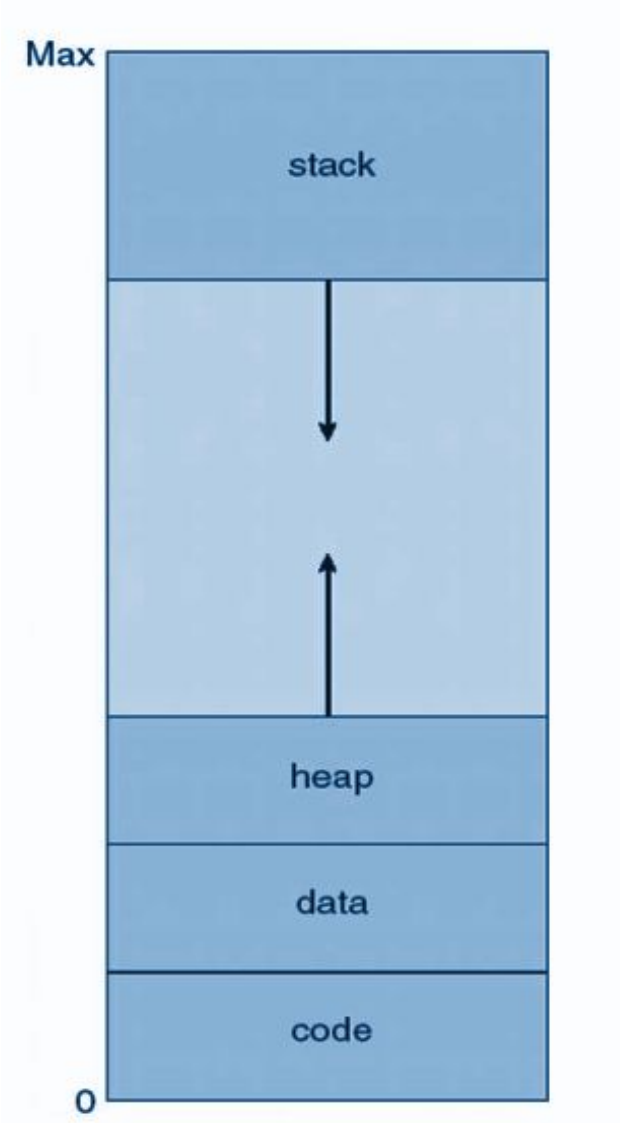
Virtual Address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
    - Maximizes address space use
    - Unused address space between the two is hole or CC
    - No physical memory needed until heap or stack grows to a given new page

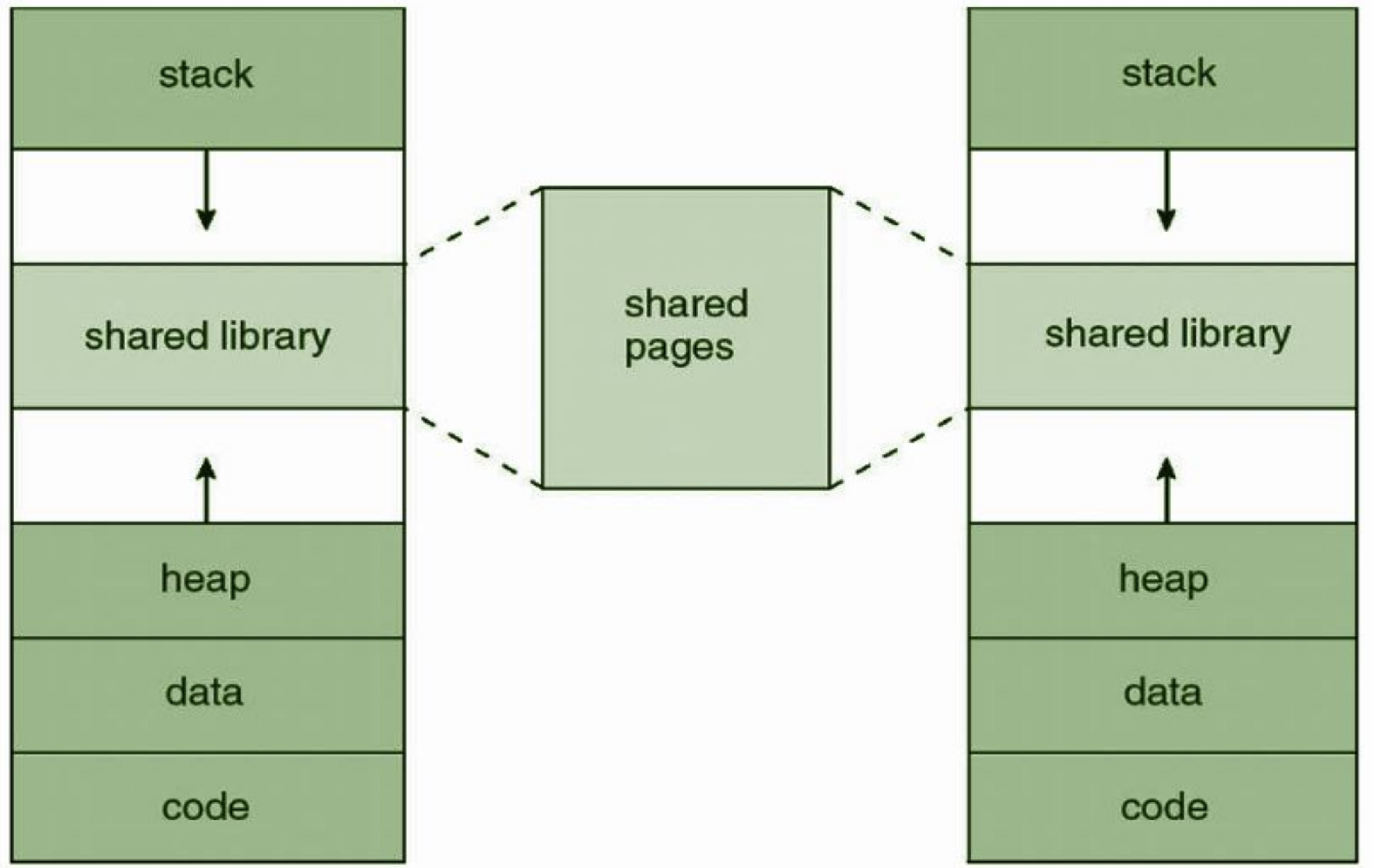


Virtual Address Space

- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during fork(), speeding process creation

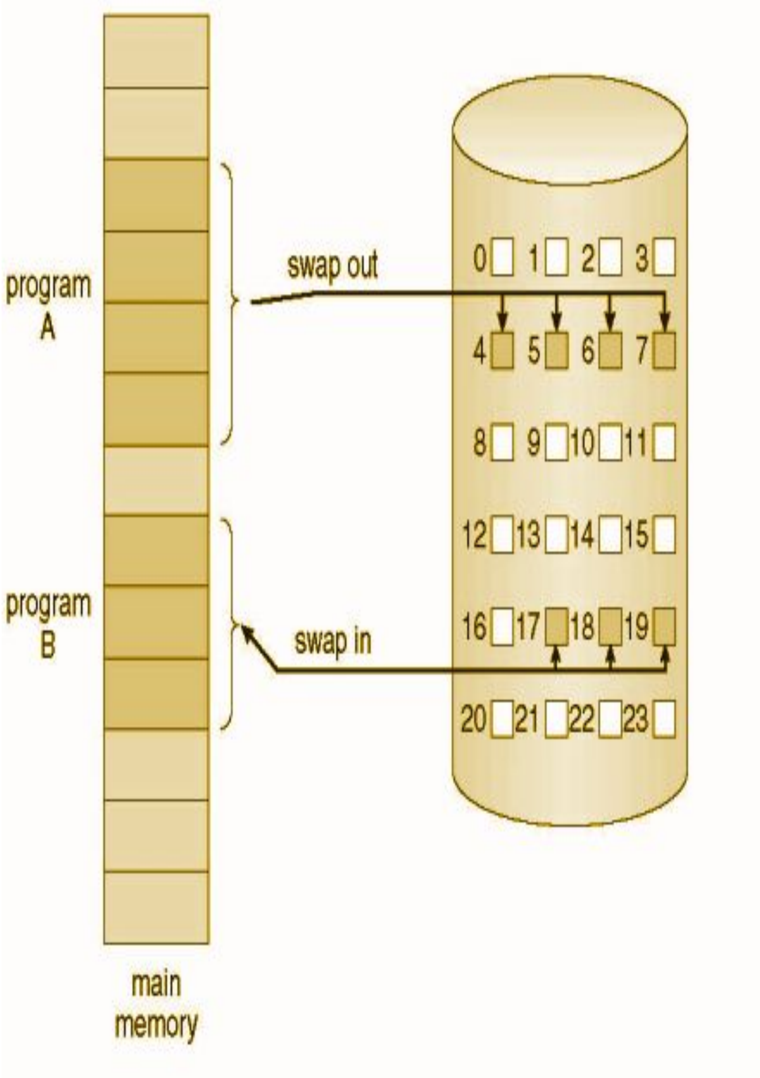


Shared Library Using Virtual Memory



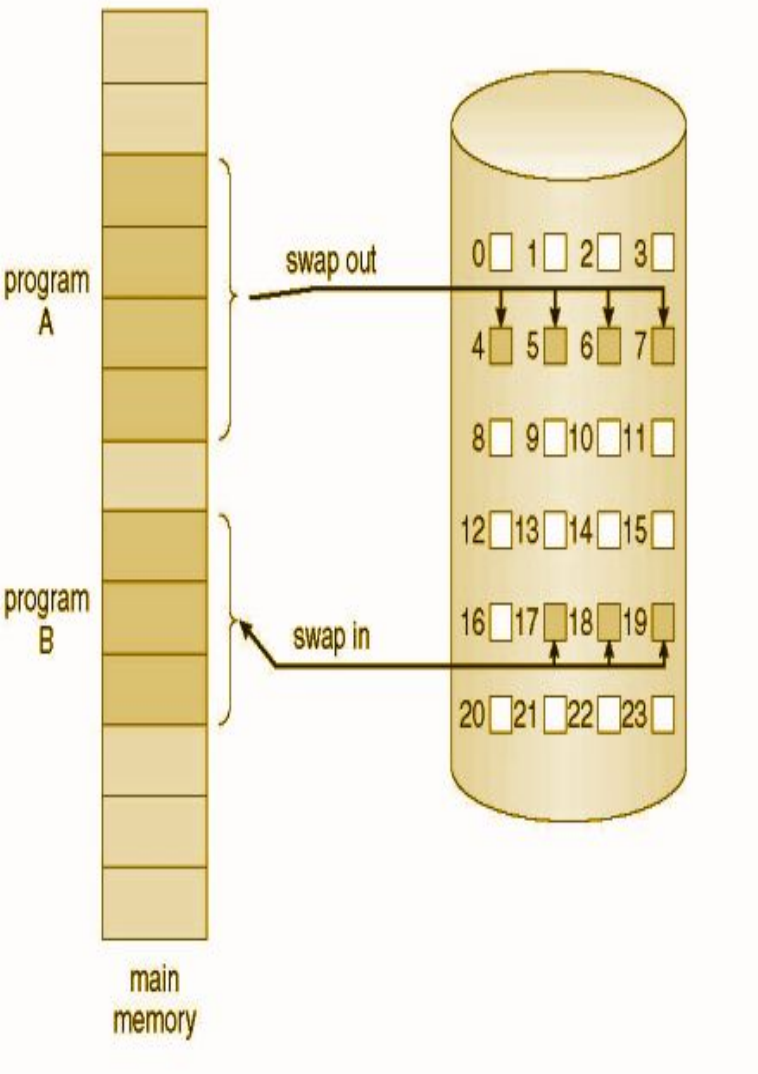
Demand Paging

- Could bring entire process into memory at load time
- Or bring a 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 paging system with swapping



Demand Paging

- 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**



Swapping - Basic Concepts

- With swapping, **Pager** guesses which pages will be used before swapping out again
- Instead, **Pager** brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory resident
  - No difference from non demand-paging

Swapping - Basic Concepts

- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code

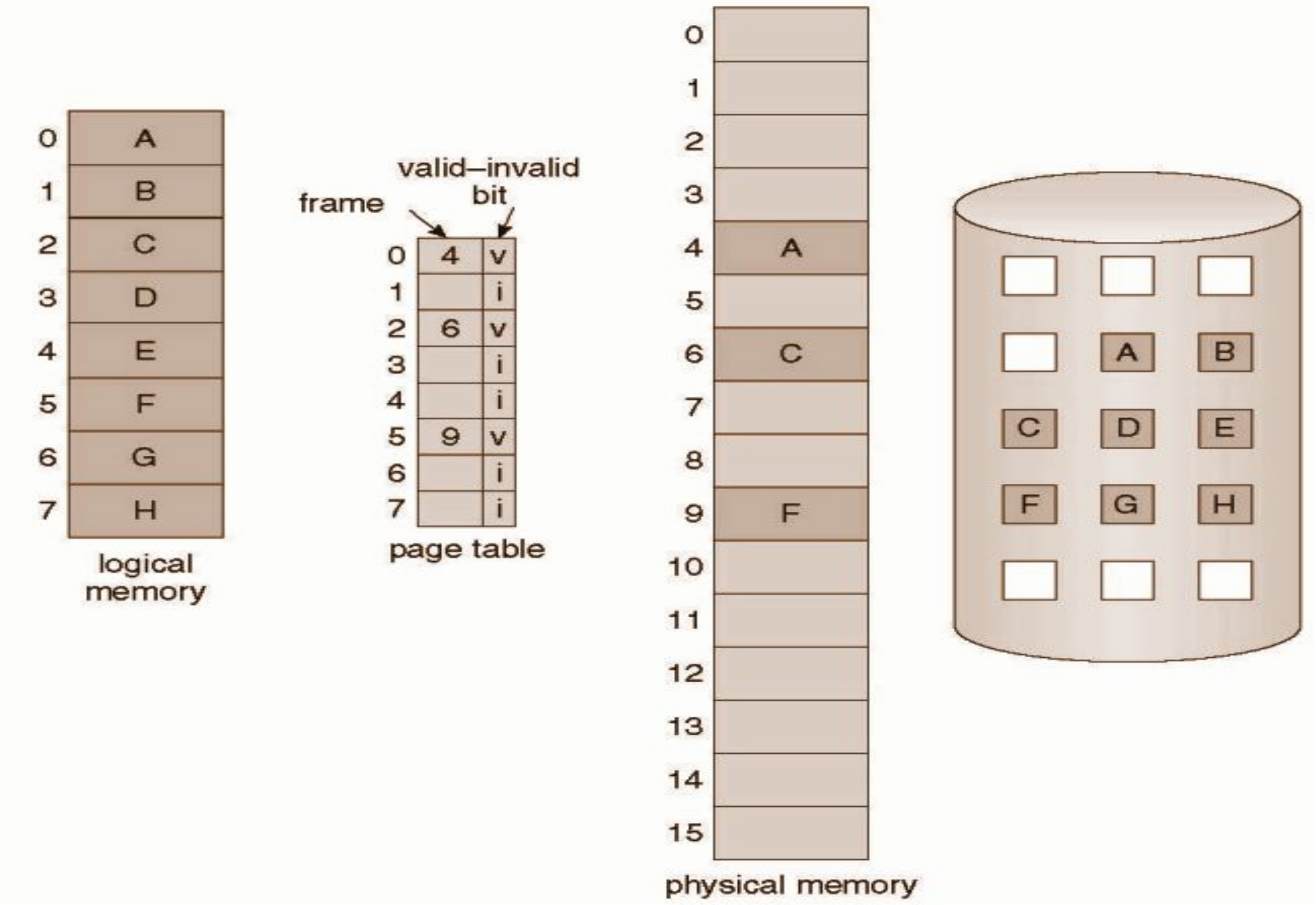
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - **v** in-memory – memory resident,
  - **i** not-in-memory
- Initially valid–invalid bit is set to **i** on all entries
- During MMU address translation, if valid–invalid bit in page table entry is **i** => **Page Fault**

Page #	Frame #	Valid - Invalid Bit
0		i
1		i
2	23	v
3		i
4	9	v
5	11	v
6		i

Page Table

Page Table When Some Pages Are Not in Main Memory



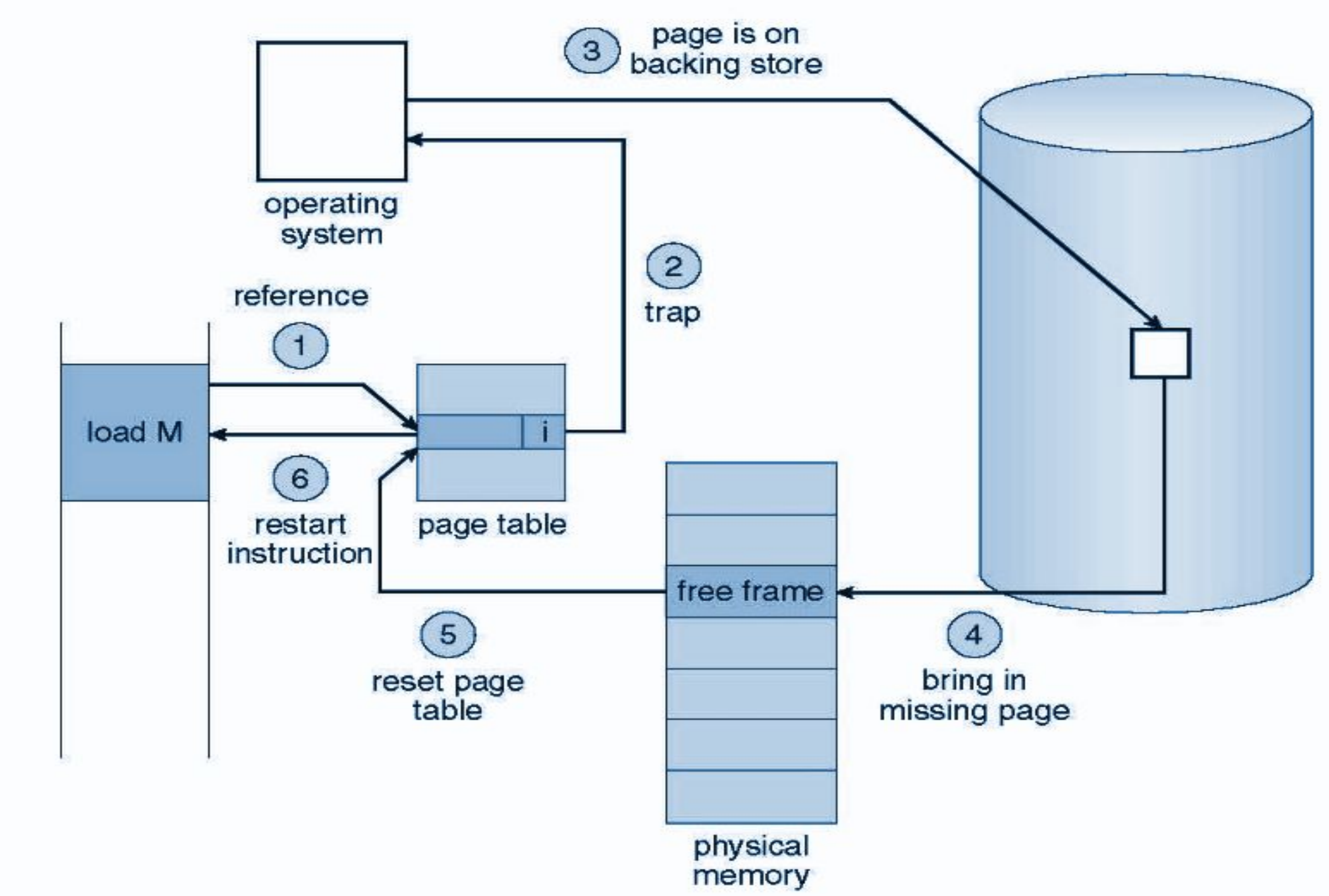
Page fault

- If there is a reference to a page, first reference to that page will trap to operating system => **Page Fault**

Page fault

1. Operating system looks at another table to decide:
   i. **Invalid reference abort**
   ii. **Just not in memory**
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
5. Set validation bit => **v**
6. Restart the instruction that caused the **Page Fault**

Page fault



3
page is on backing store
operating system
2
trap
reference
1
load M
i
page table
6
restart instruction
free frame
physical memory
5
reset page table
4
bring in missing page

Aspects of Demand Paging

- Extreme case => start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident => **Page Fault**
- For every other process pages on first access **Pure Demand Paging**

Aspects of Demand Paging

- Actually, a given instruction could access multiple pages => multiple page faults

- Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory

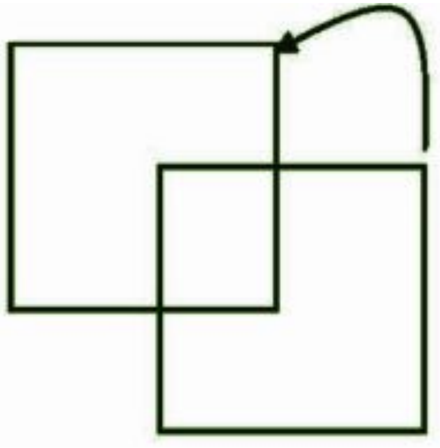
- Pain decreased because of **Locality of Reference**

Aspects of Demand Paging

- Hardware support needed for Demand Paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

Aspects of Demand Paging

- Consider an instruction that could access several different locations
  - block move
  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?



Performance of Demand Paging - Worst Case

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:
    i. Wait in a queue for this device until the read request is serviced
    ii. Wait for the device seek and/or latency time
    iii. Begin the transfer of the page to a free frame

Performance of Demand Paging - Worst Case

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
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 - Worst Case

- Three major activities
  - Service the interrupt => careful coding means just several hundred instructions needed
  - Read the page => lots of time
  - Restart the process => again just a small amount of time

Performance of Demand Paging - Worst Case

- Page Fault Rate 0 <= p <= 1
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 – p) x memory access
    + p (page fault overhead=> + [swap
    page **out]** + swap page **in** +
    Restart Overhead)

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if No Page Fault i.e p=>0 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 0) * 200 + 0 * PFST

  => 200ns

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if Page Fault i.e p=>1 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 1) * 200 + 1 * PFST

  => PFST

Demand Paging - Performance Calculation

- 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!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 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 Paging Optimizations

- 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

Demand Paging Optimizations

- Mobile systems
  - Typically don’t support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

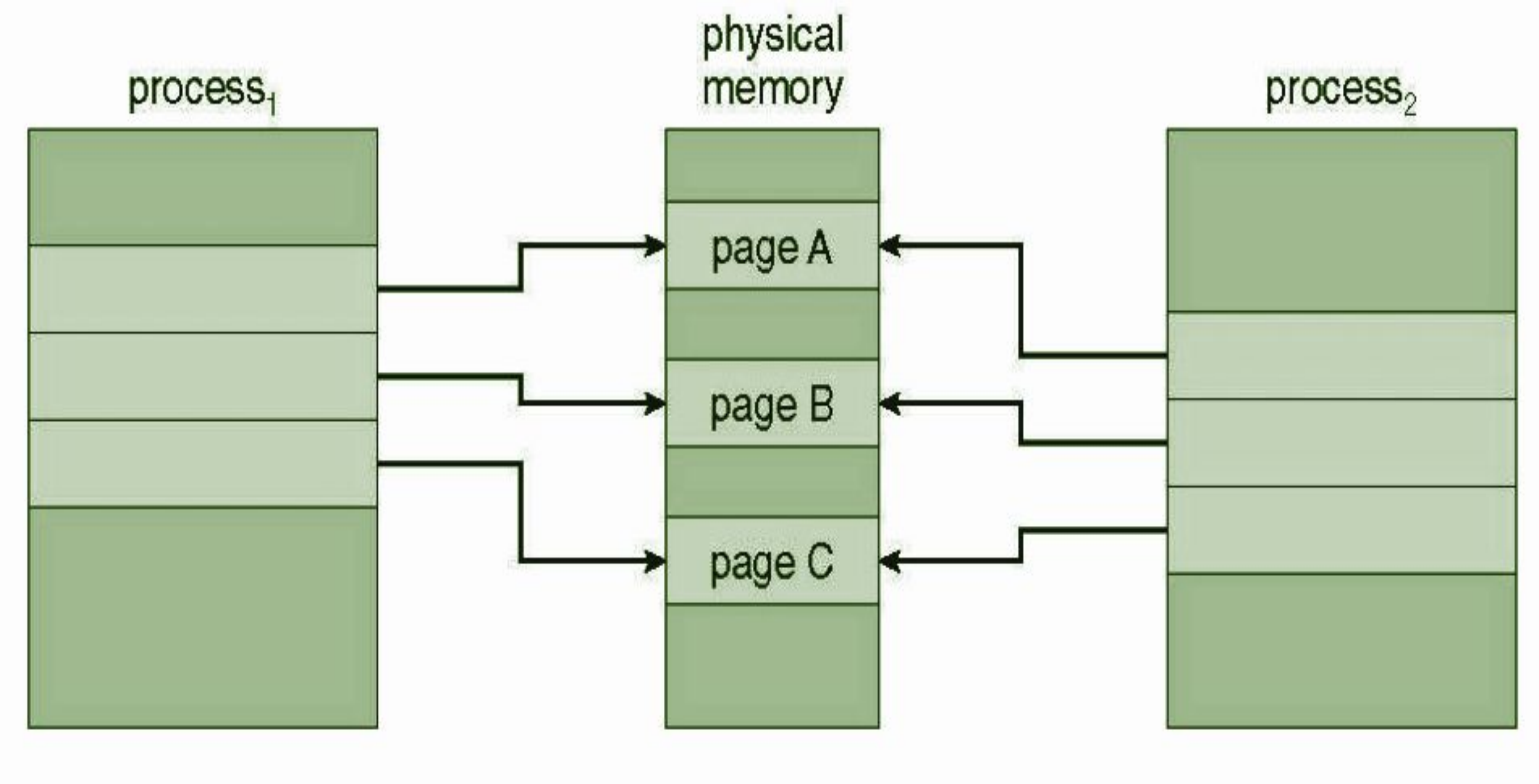
Copy-on-Write

- 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 ?

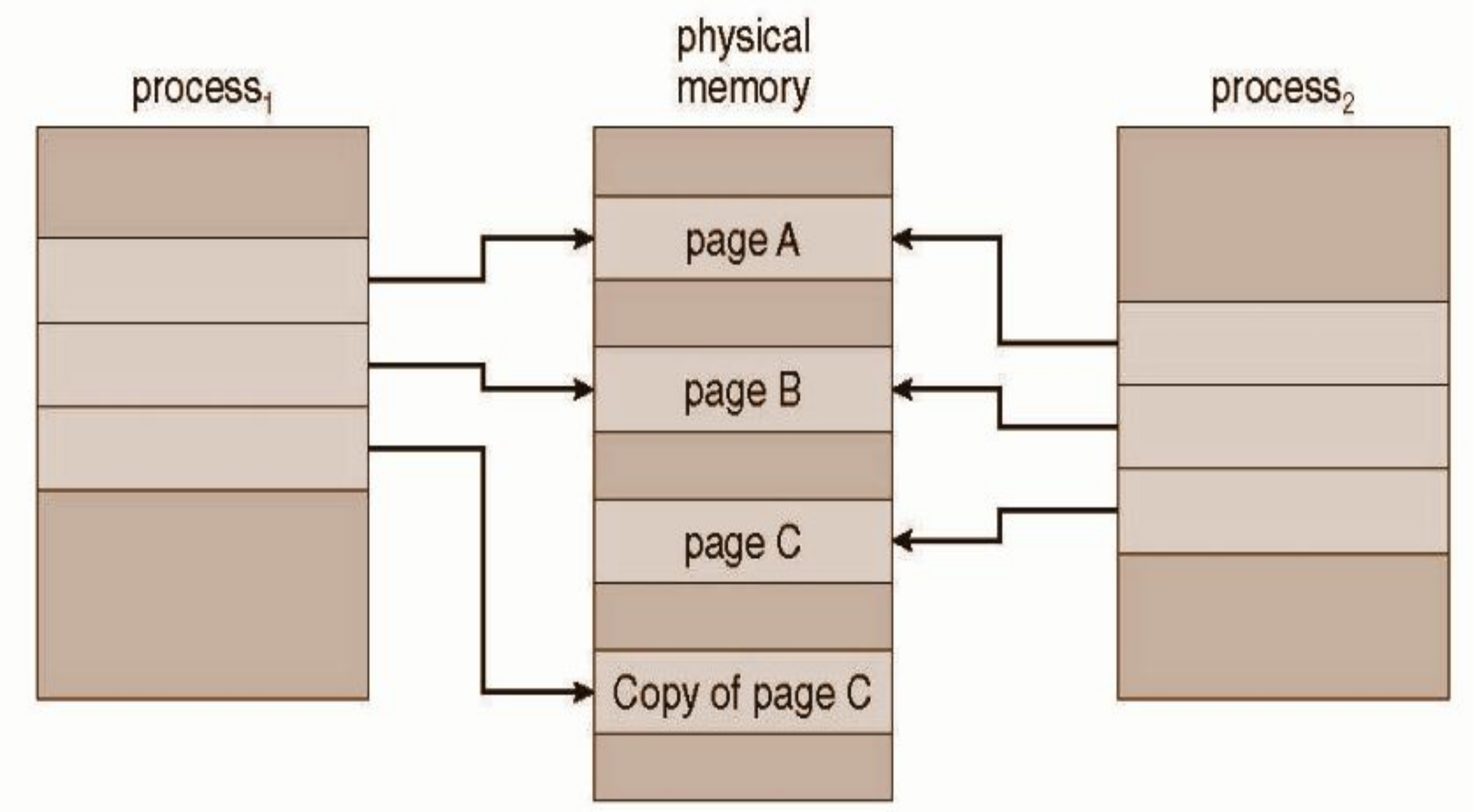
Copy-on-Write

- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient

Copy-on-Write: Before Process 1 Modifies Page C



Copy-on-Write: Before Process 1 Modifies Page C



THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com and complete reading assignments provided by the Anchor on Edmodo