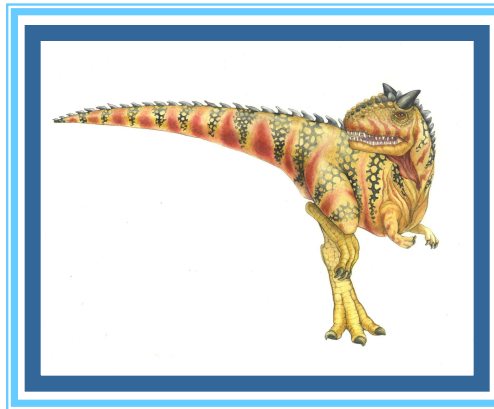


# Chapter 9: Virtual Memory

## Part 1

---





# Objectives

---

- To understand the virtual memory concept
- Benefits of virtual memory
- Demand Paging
- Copy on Write





# Virtual Memory Benefits

---

- Code needs to be **in memory to execute**, but entire program rarely used
- 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
  - 4 Increased CPU utilization and throughput with no increase in response time or turnaround time
- Less I/O needed to load or swap programs into memory -> each user program runs faster





# Virtual Memory Benefits

- **Virtual memory** – separation of user logical memory from physical memory
- **Virtual Memory** is a storage mechanism which offers user an illusion of having a very big main memory. It is done by treating a part of secondary memory as the main memory. In Virtual memory, the user can store processes with a bigger size than the available main 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

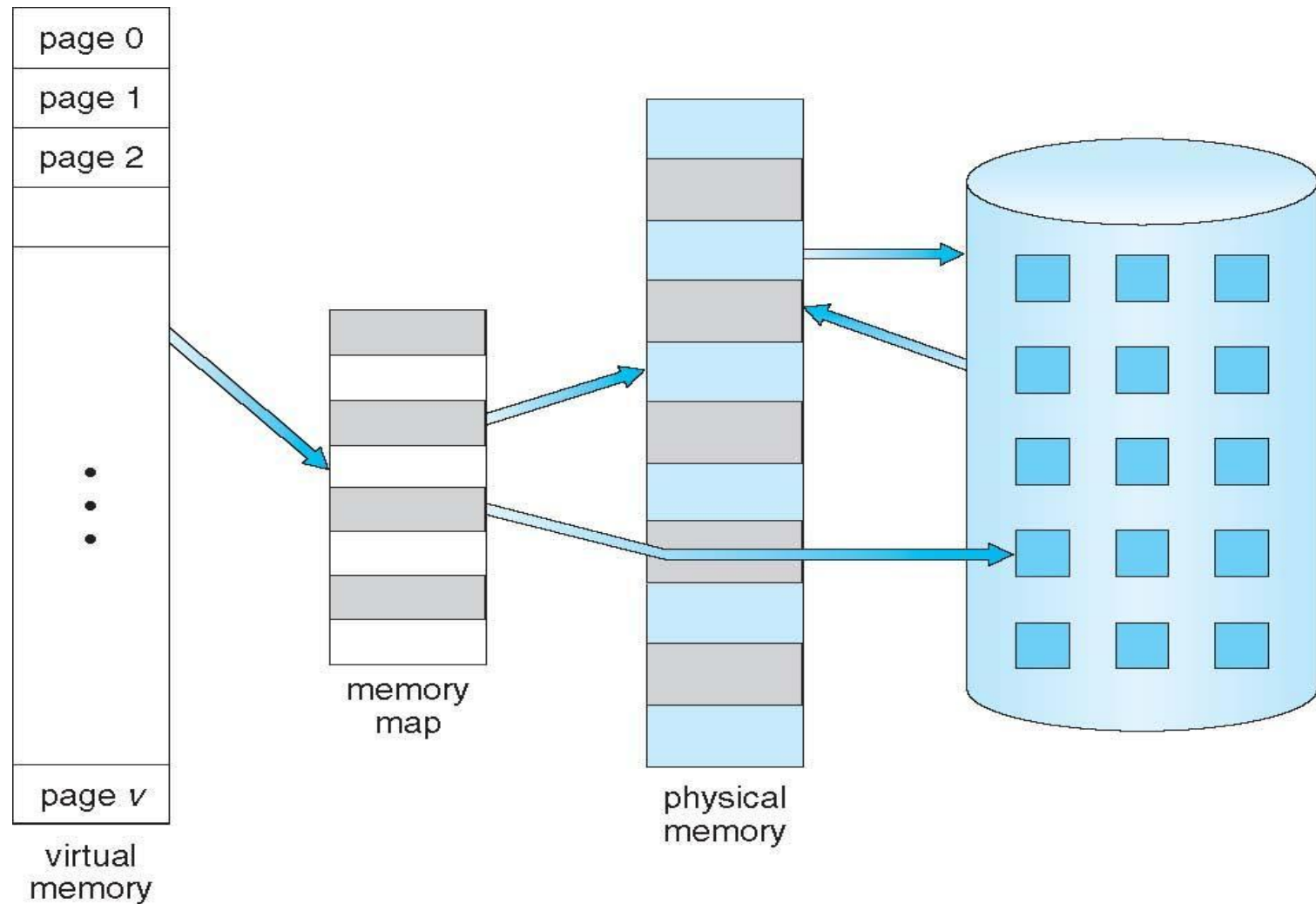
---

- **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 can be implemented via:
  - Demand paging
  - Demand segmentation





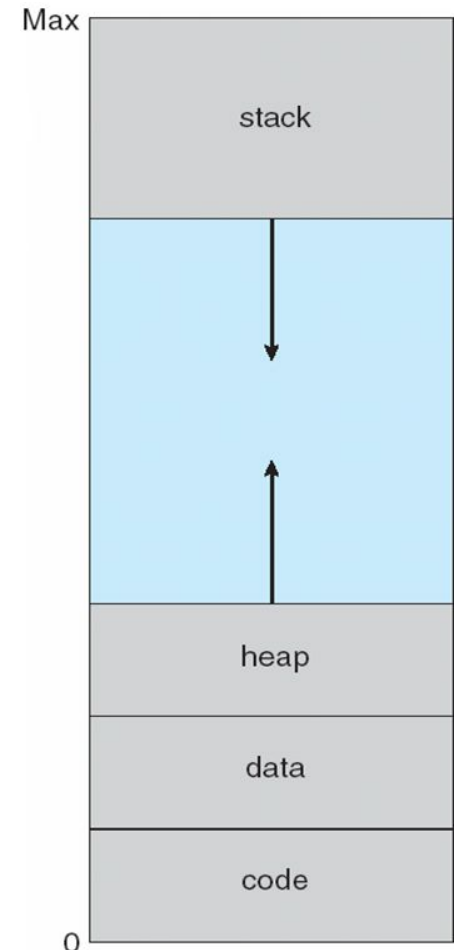
# Virtual Memory That is Larger Than Physical Memory





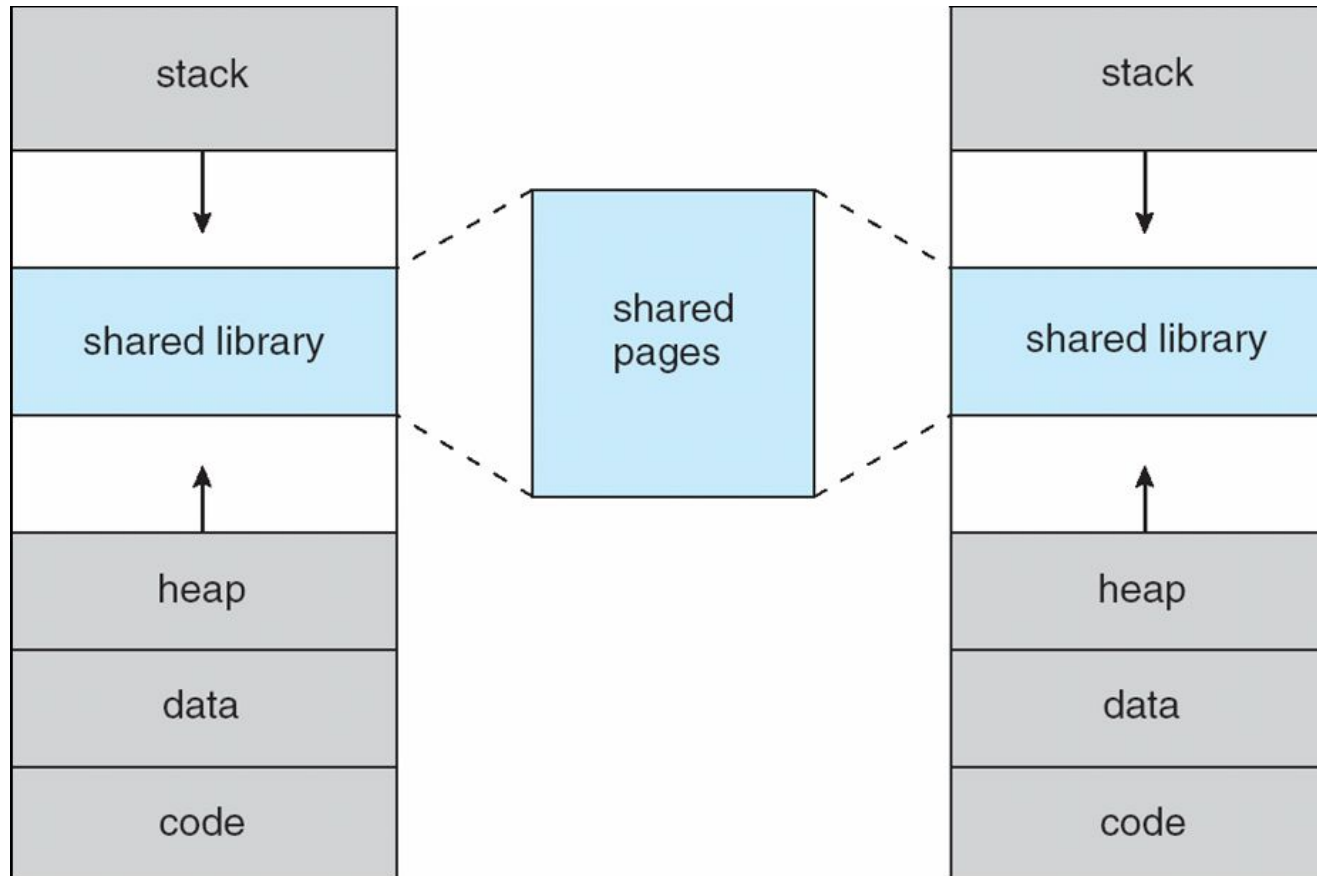
# Virtual-address Space

- 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
    - 4 No physical memory needed until heap or stack grows to a given new page
- 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





# Shared Library Using Virtual Memory

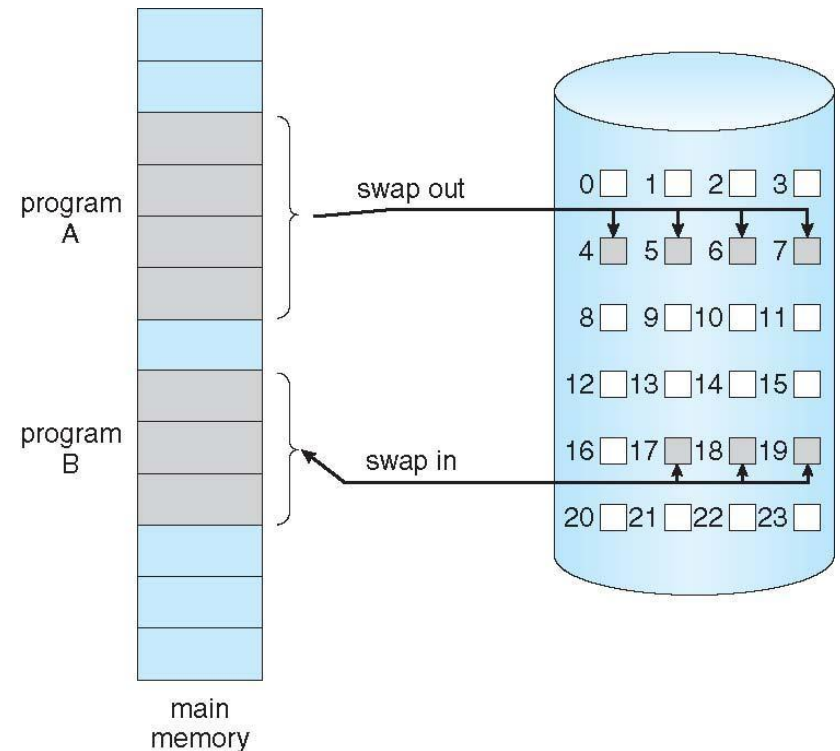






# Demand Paging

- 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 (diagram on right)
- 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**





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





# Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v**  $\Rightarrow$  in-memory – **memory resident**, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid–ir
	v
	v
	v
	i
...	
	i
	i

page table

- During MMU address translation, if valid–invalid bit in page table entry is **i**  $\Rightarrow$  page fault





# Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

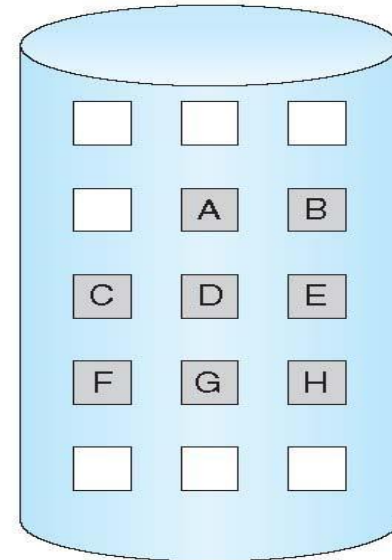
logical  
memory

valid-invalid frame      bit		
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

physical memory





# Page Fault

---

- If there is a reference to a page, first reference to that page will trap to operating system:

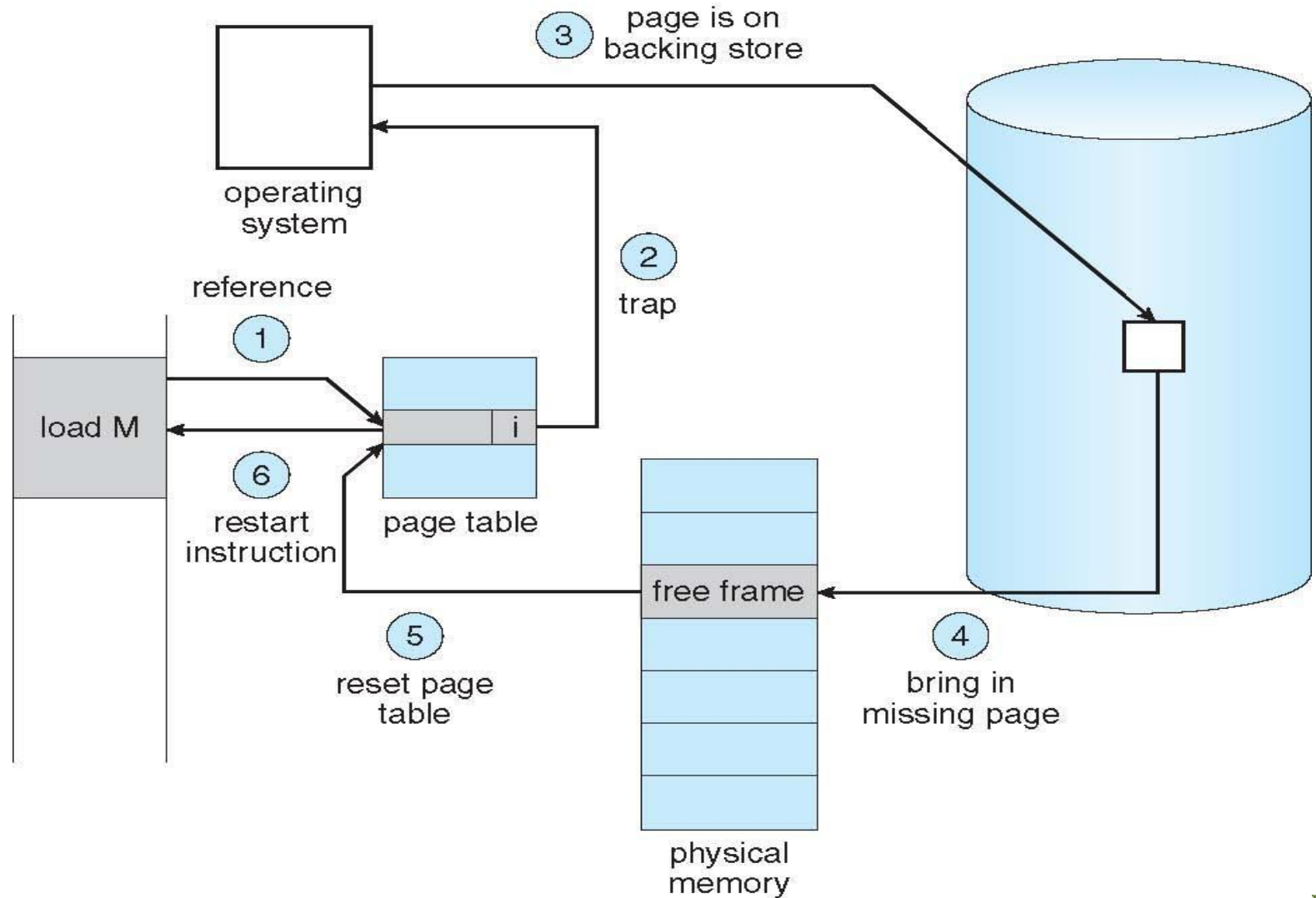
## page fault

1. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - 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  
Set validation bit = **v**
5. Restart the instruction that caused the page fault





# Steps in Handling a Page Fault





# Performance of Demand Paging

- Stages in Demand Paging (worse 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:
    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
  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 (Cont.)

- 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
- Page Fault Rate  $0 \leq p \leq 1$ 
  - 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 )







# Demand Paging Example

---

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$   
 $= (1 - p) \times 200 + p \times 8,000,000$   
 $= 200 + p \times 7,999,800$
- If one access out of 1,000 causes a page fault, then  
 $EAT = 8.2 \text{ microseconds.}$
- The CPU has to access the missed page from the secondary memory. If the number of page fault is very high, then the effective access time of the system will become very high.





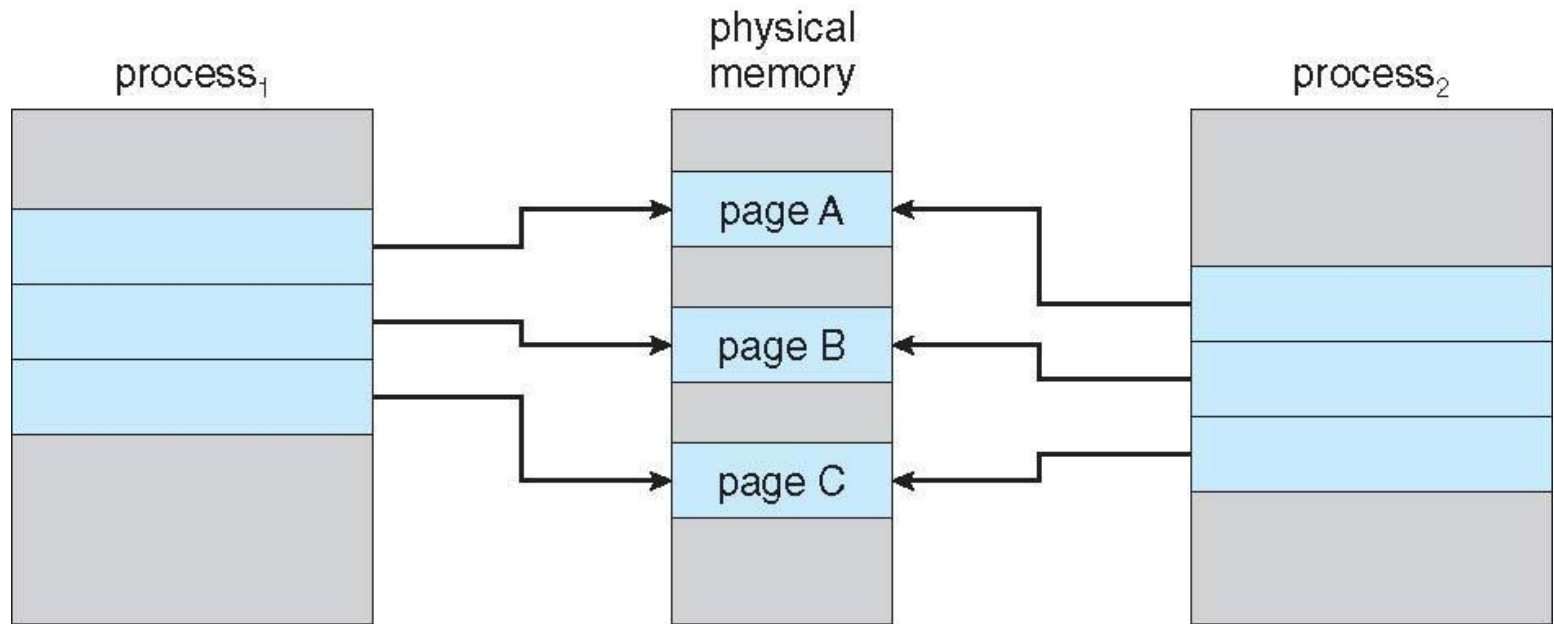
# Copy-on-Write

- **Copy on Write** or simply COW is a resource management technique. One of its main use is in the implementation of the fork system call in which it shares the virtual memory(pages) of the OS.
- **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
    - 4 Don't want to have to free a frame as well as other processing on page fault





# Before Process 1 Modifies Page C





# After Process 1 Modifies Page C

