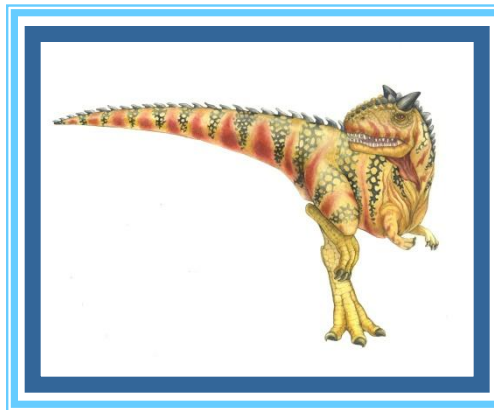


# Chapter 9: Virtual Memory



# Chapter 9: Virtual Memory

- Background
- Demand Paging

# Objectives

- To describe the definition and benefits of a virtual memory system
- To explain the concepts of demand paging,

# 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
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

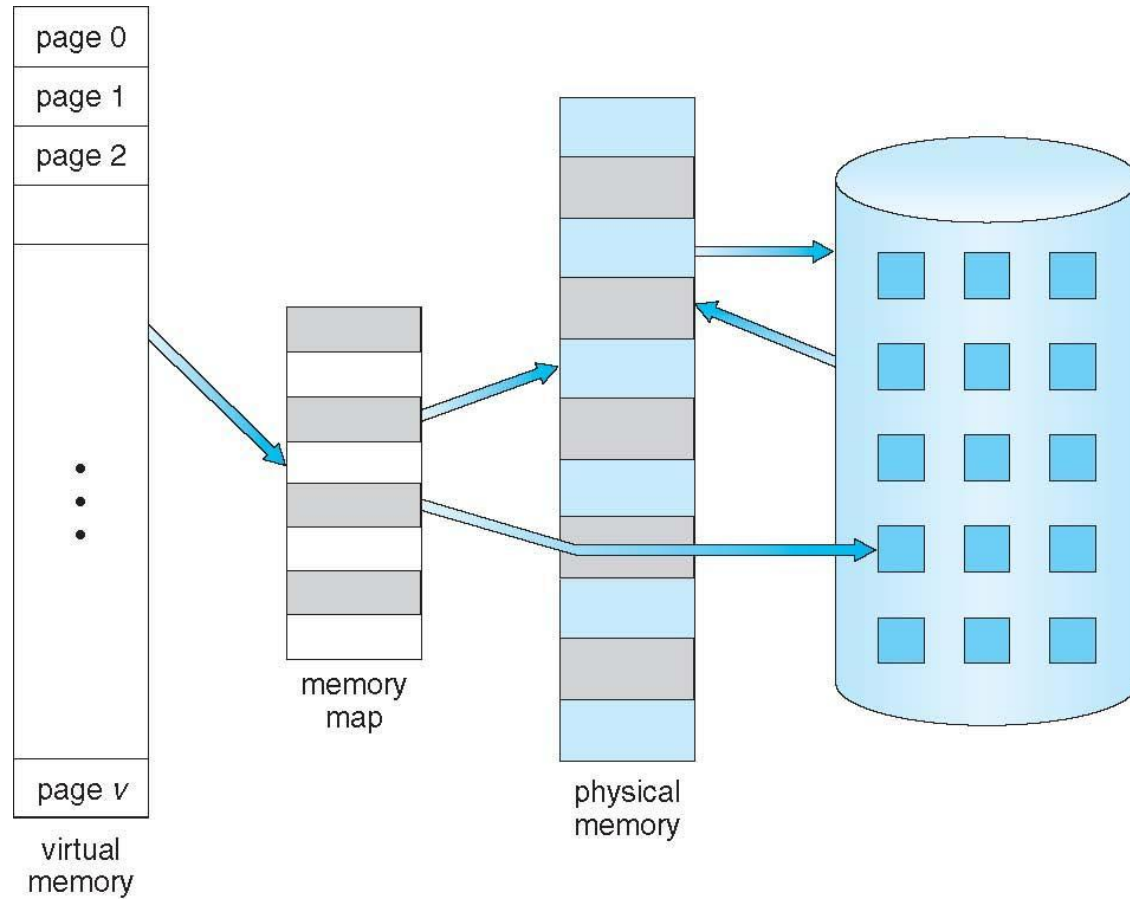
# Background (Cont.)

- **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

# Background (Cont.)

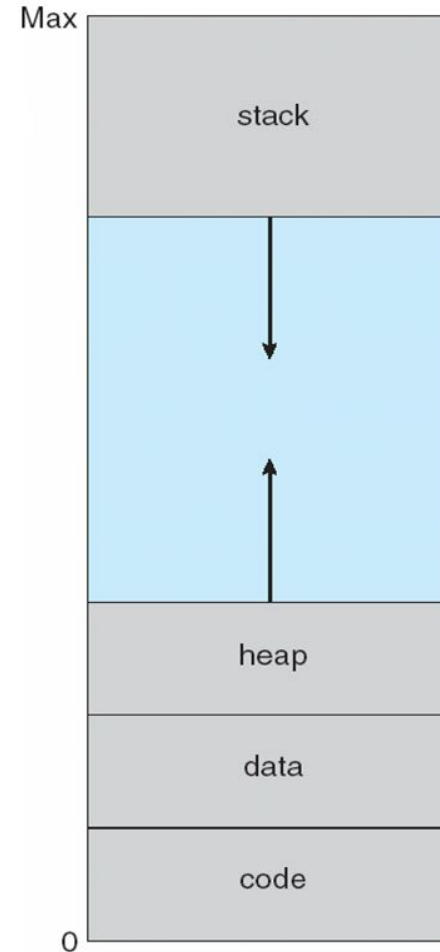
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory That is Larger Than Physical Memory



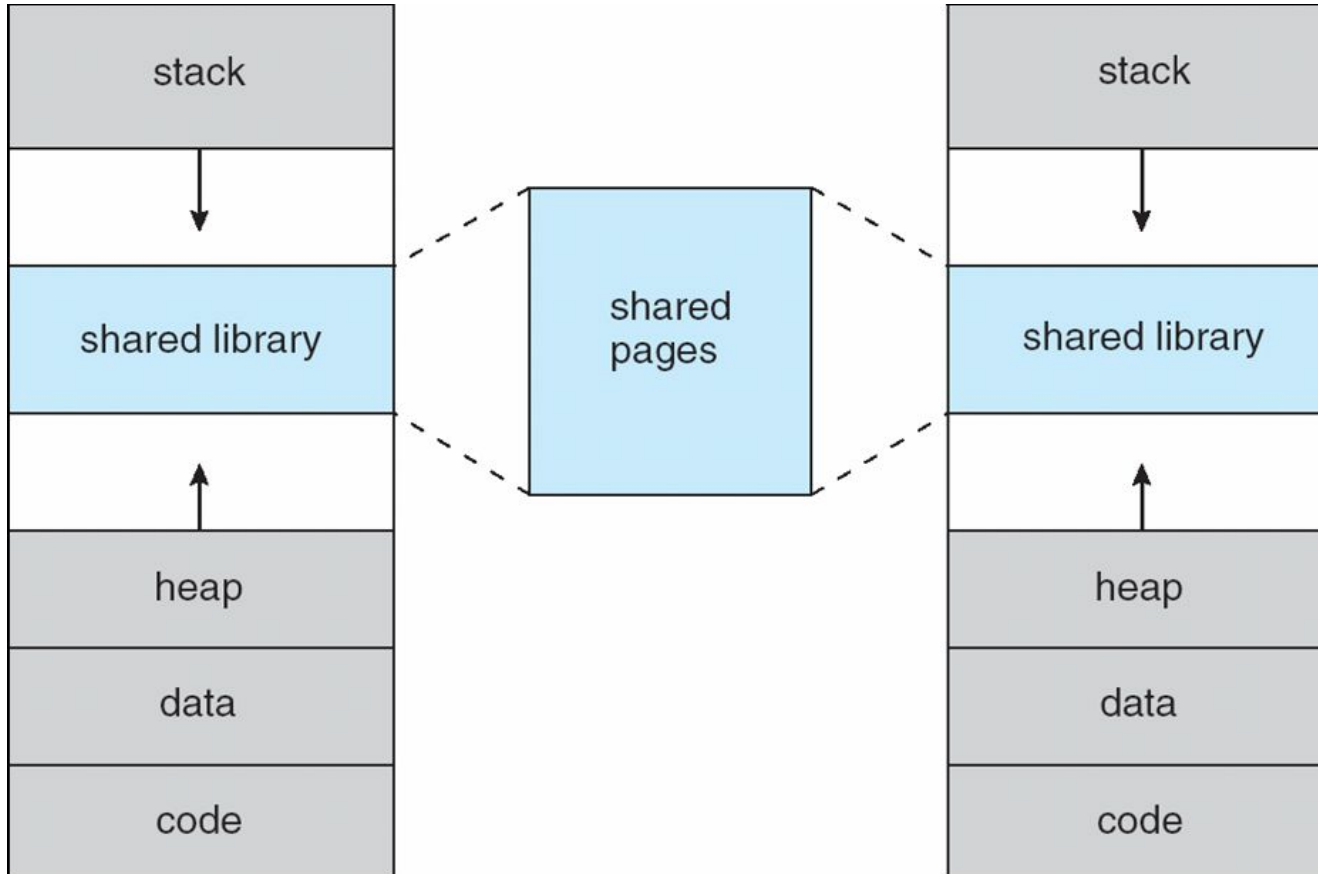
# Virtual-address Space

- 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
- Pages can be shared during `fork()`, speeding process creation



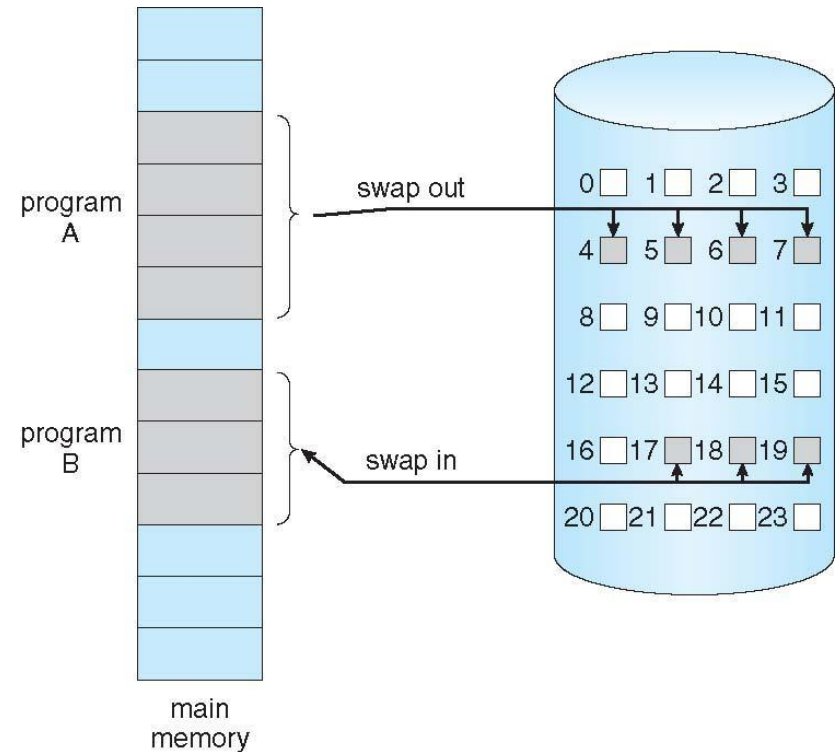


# Shared Library Using Virtual Memory



# 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 (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
    - 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**  $\Rightarrow$  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-invalid bit |
|---------|-------------------|
|         |                   |
|         | <b>v</b>          |
|         | <b>v</b>          |
|         | <b>v</b>          |
|         | <b>i</b>          |
| ...     |                   |
|         | <b>i</b>          |
|         | <b>i</b>          |

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 |

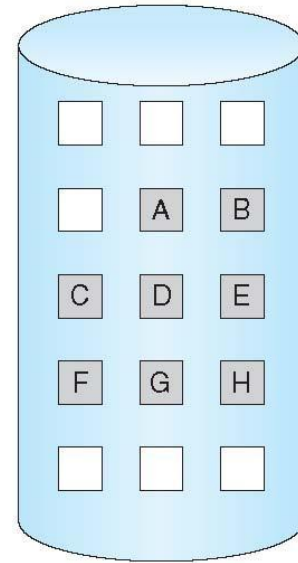
logical  
memory

|   | frame | valid-invalid<br>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  |
| 5  |
| 6  |
| 7  |
| 8  |
| 9  |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |
| 15 |

physical memory



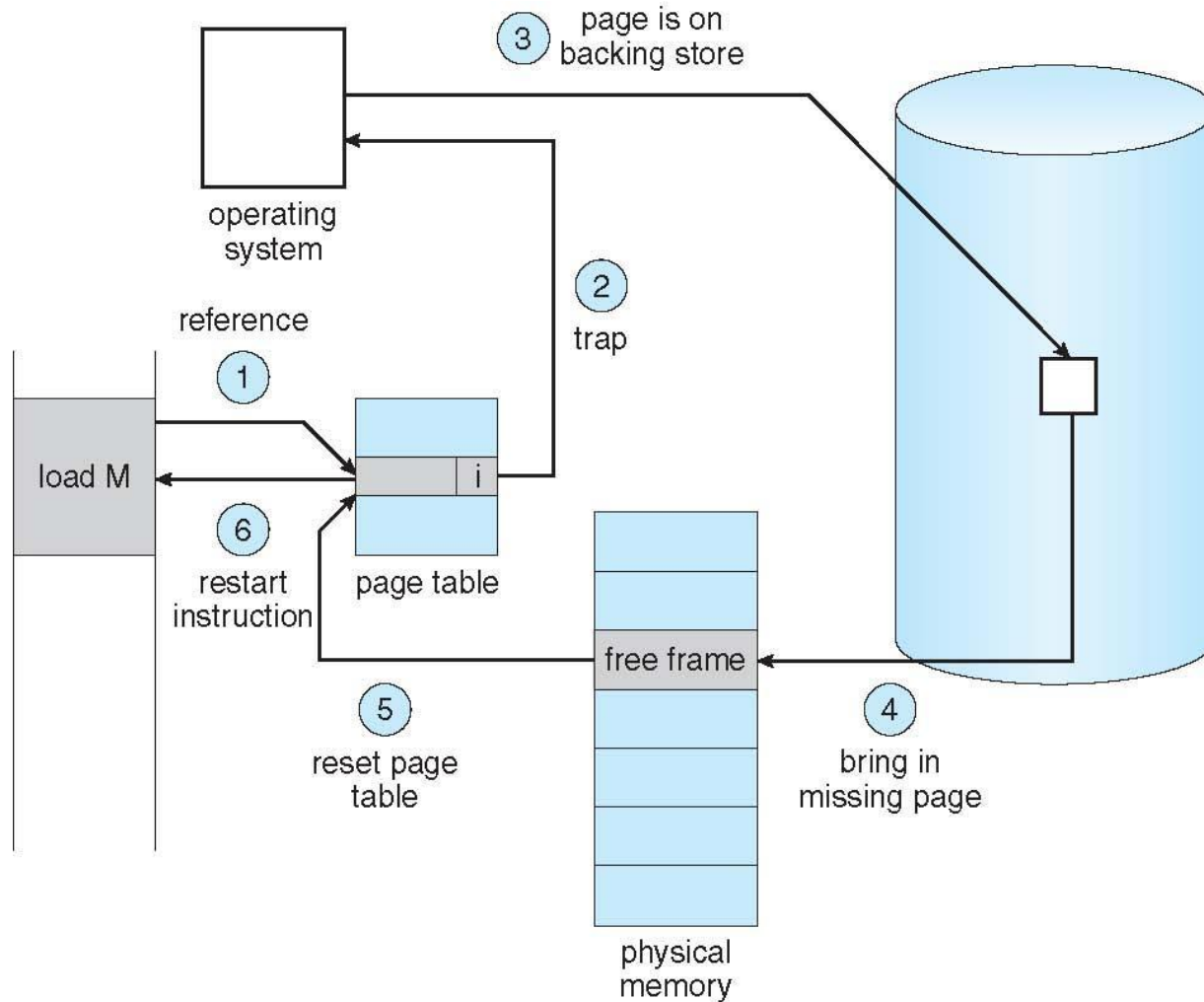
# Page Fault

- 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



# 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
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults