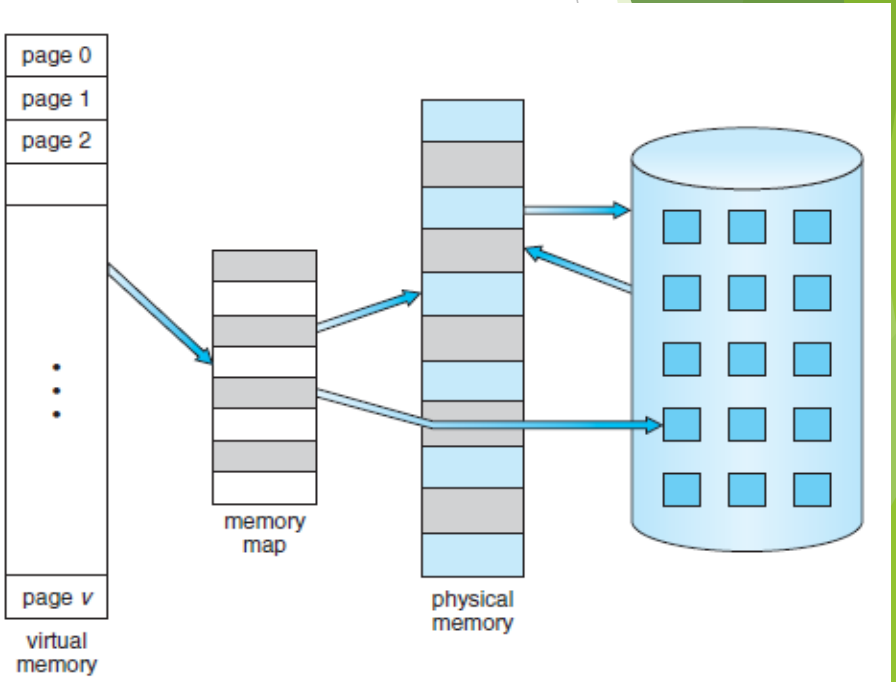
Virtual Memory

Concept

- It gives the illusion to the programmer that the programs of larger size than actual physical memory can be executed.
- Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.
- In many cases, the entire program is not needed. For instance, consider the following:
  - Programs often have code to handle unusual error conditions.
  - Arrays, lists, and tables are often allocated more memory than they actually need.
- The ability to execute a program that is only partially in memory would confer many benefits:
  - Users would be able to write programs for an extremely large ***virtual*** address space, simplifying the programming task.
  - Because each user program could take less physical memory, more programs could be run at the same time.
  - Less I/O would be needed to load or swap user programs into memory

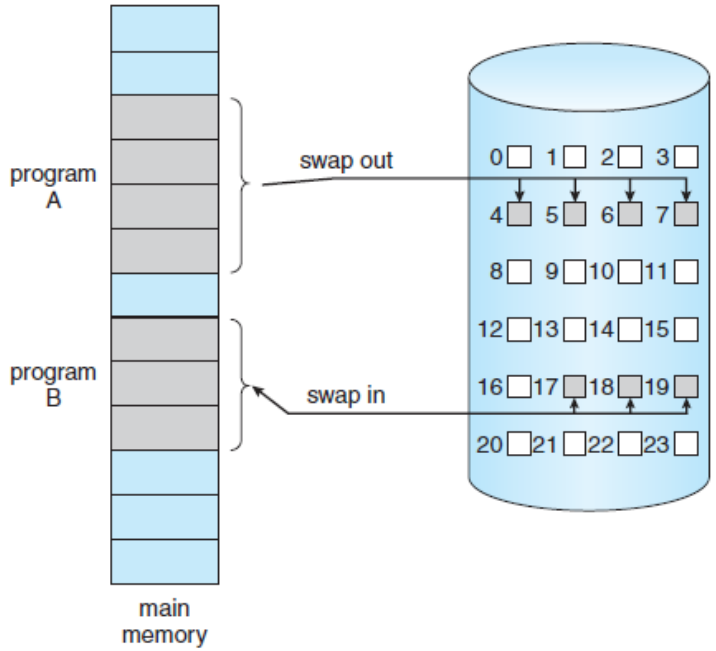
- **Virtual memory** involves the separation of logical memory as perceived by users from physical memory.
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.
- The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory



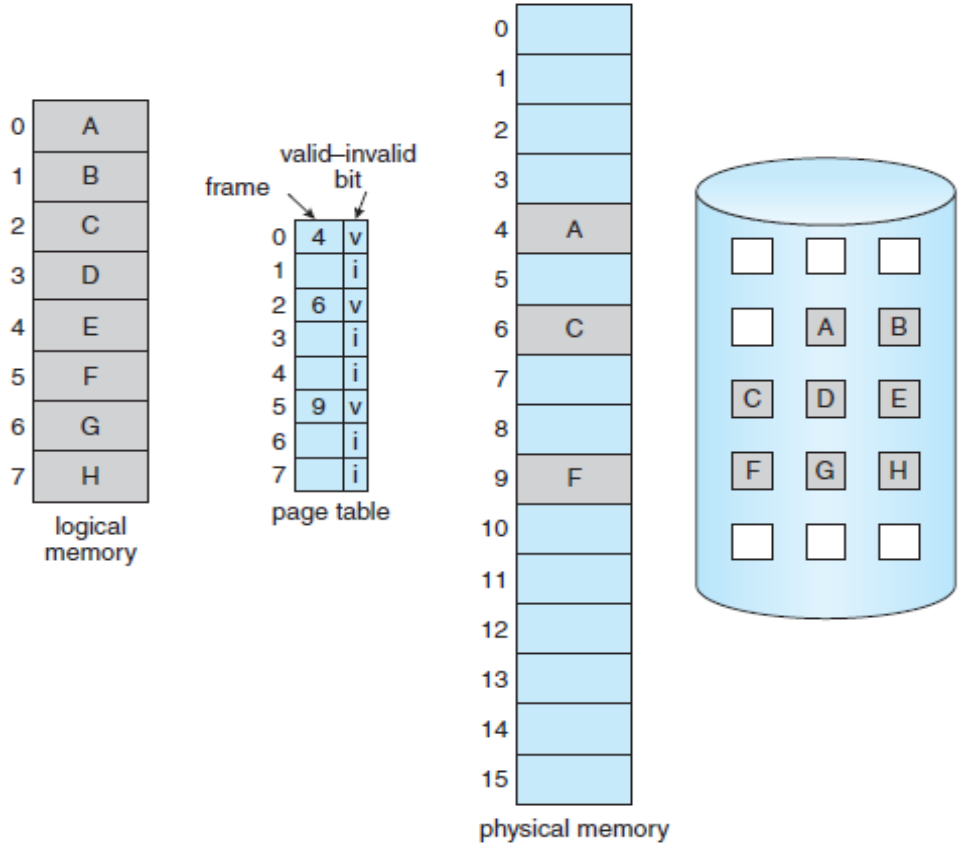
Demand Paging

- An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory.
- When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process into memory, though, we use a **Pager**. A Pager never swaps a page into memory unless that page will be needed.

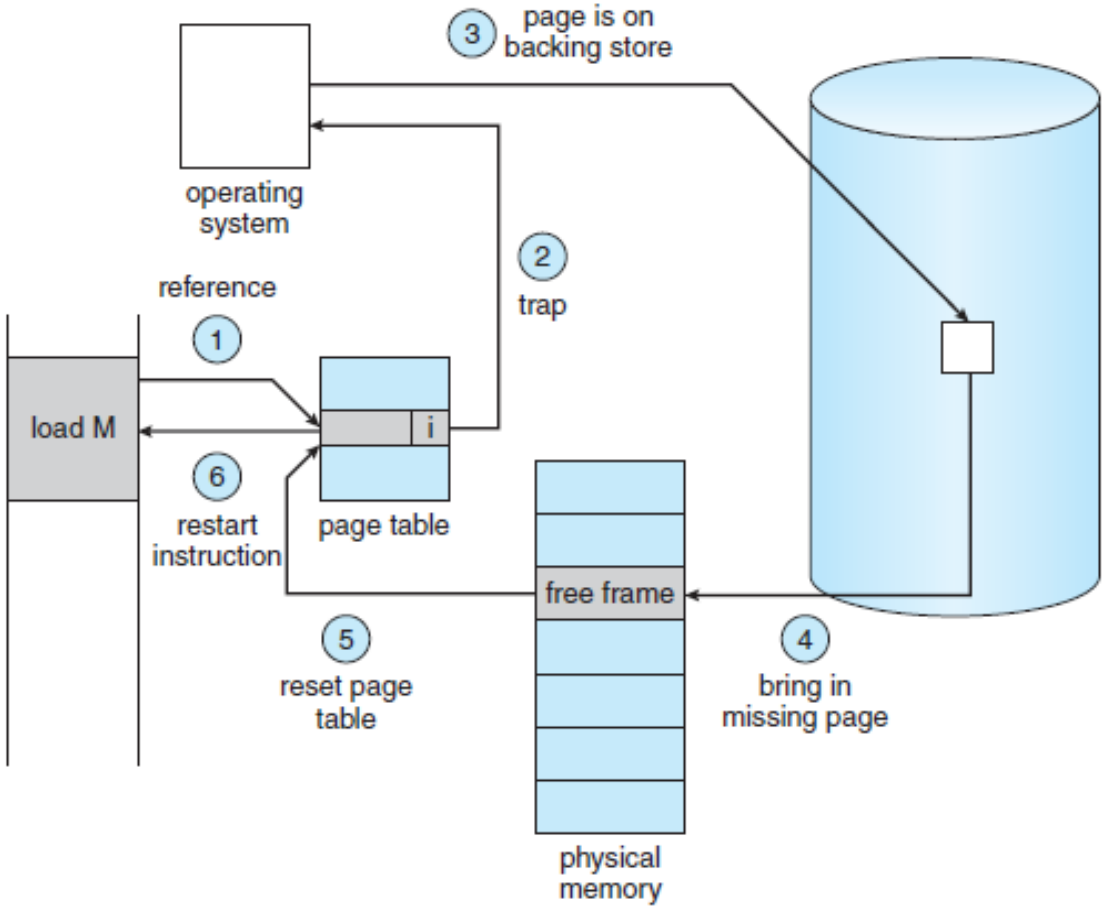
Demand Paging



Valid and Invalid bit



Steps in handling a page fault



Performance of Demand Paging

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}$$

- With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is:

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p\ (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p.\end{aligned}$$

- Consider a system with mm access time =1ns and page fault service time =100ns and a page hit ratio is 99%. Then what is EMAT?

Page Replacement

- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory.
- We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement

1. Find the location of the desired page on the disk.

2. Find a free frame:

a. If there is a free frame, use it.

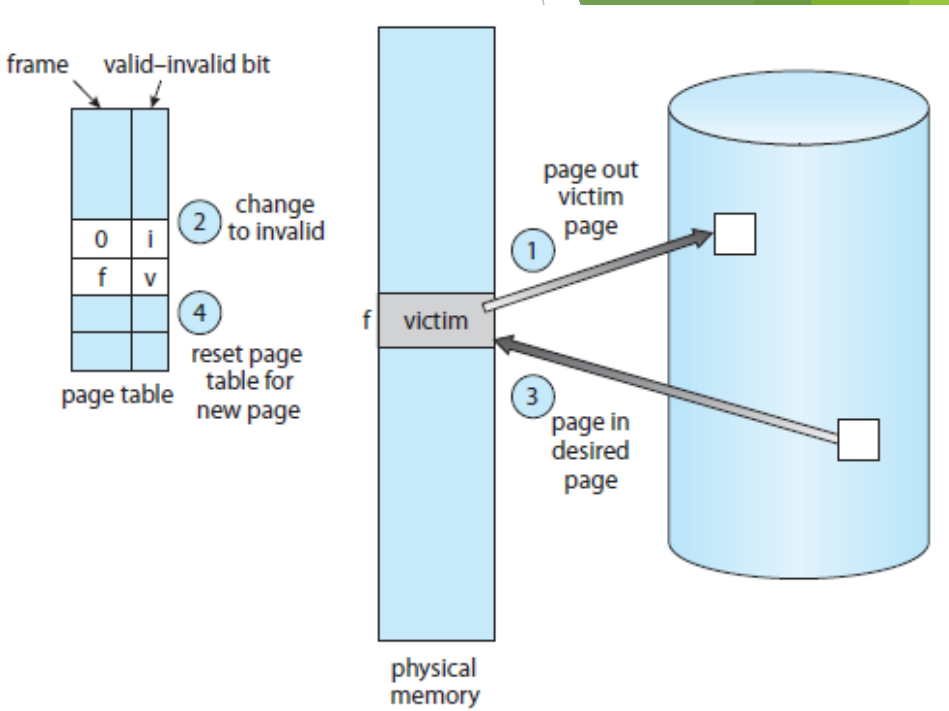
b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.

c. Write the victim frame to the disk; change the page and frame tables accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.

4. Continue the user process from where the page fault occurred.

- Notice that, if no frames are free, ***two*** page transfers (one out and one in) are required.



Read only Pages

- The **modify bit** for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.
- When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
- If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code).

Page Replacement Algorithm

- We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is,
  - if we have multiple processes in memory, we must decide how many frames to allocate to each process; and
  - when page replacement is required, we must select the frames that are to be replaced.
- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

Page Replacement Algorithm

- FIFO
  - Reference String

    7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

    3 frames
  - Reference String

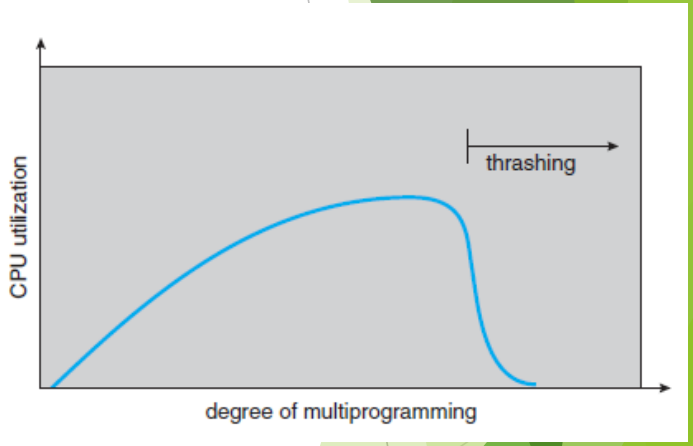
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

    3 frames and 4 frames
  - Belady's Anomaly
- Optimal Page Replacement
- LRU Page Replacement- implementation- hardware support, stack
- **LRU-Approximation Page Replacement-** reference bit is used
  - **Additional-Reference**- Bits Algorithm- 8byte –shift operation
- Counting-Based Page Replacement
  - least frequently used (LFU)
  - most frequently used (MFU)

FIFO-15
OPT-9
LRU-12

Thrashing

- This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.
- Initially Degree of multi-programming is very high, if we further increase the degree of multi-programing after some threshold. The CPU utilization will drastically fall down .



- Causes:
  - High degree of multiprogramming
  - Lack of frame
- Recovery
  - Not bring the process into memory after the threshold
  - Suspend some of the processes