Chap 10: Virtual Memory Management

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

KIIT DEEMED TO BE UNIVERSITY

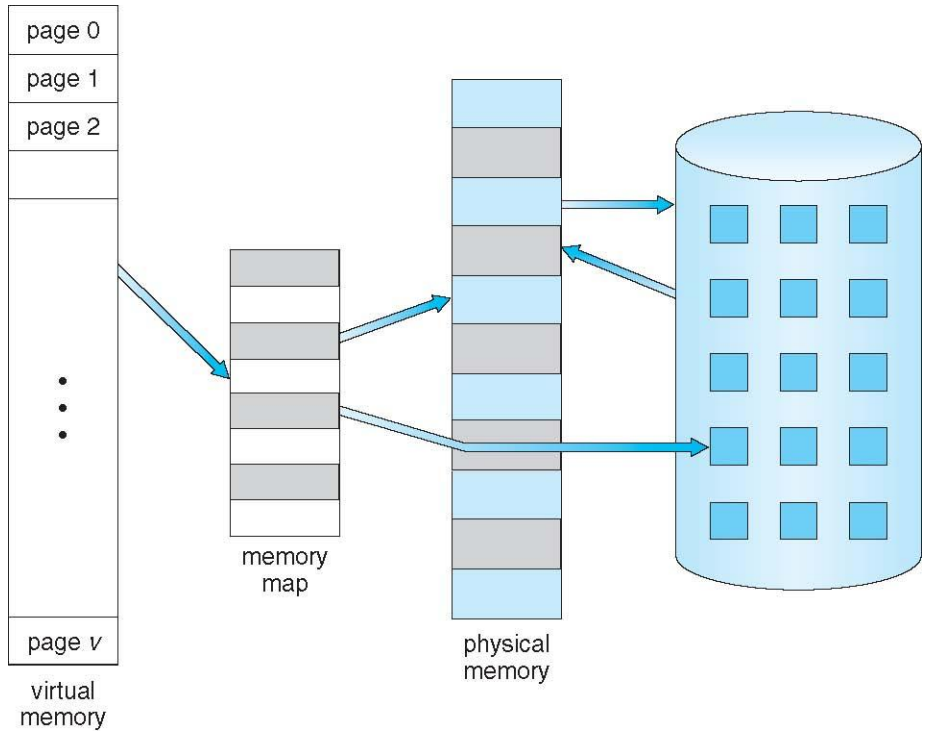
Background

- Basic requirement of memory management strategies : instruction being executed must be in physical memory.
- To achieve this, one obtion is place entire LAS into physical memory through dynamic loading.
- **But this requirement also limits the size of the program to the size of the physical memory.**
- **Real life instances show that in many cases, the entire code is not needed or even if the entire program is needed, it is not required all at the same time.**
- **So we need schemes to execute a program that is only partially in main memory.**

Benefits of program that is only partially in main memory

The benefits it will lead to both system and user are:

- A program is no longer constrained by the amount of physical memory and use a large virtual address space.So easy programming by user.
- Each program takes less physical memory, more programs can be run simultaneously so CPU utilization, multiprogramming, throughput increases but with no increase in RT or TAT.
- Less I/O is needed to load or swap a user program into memory so each program runs faster.



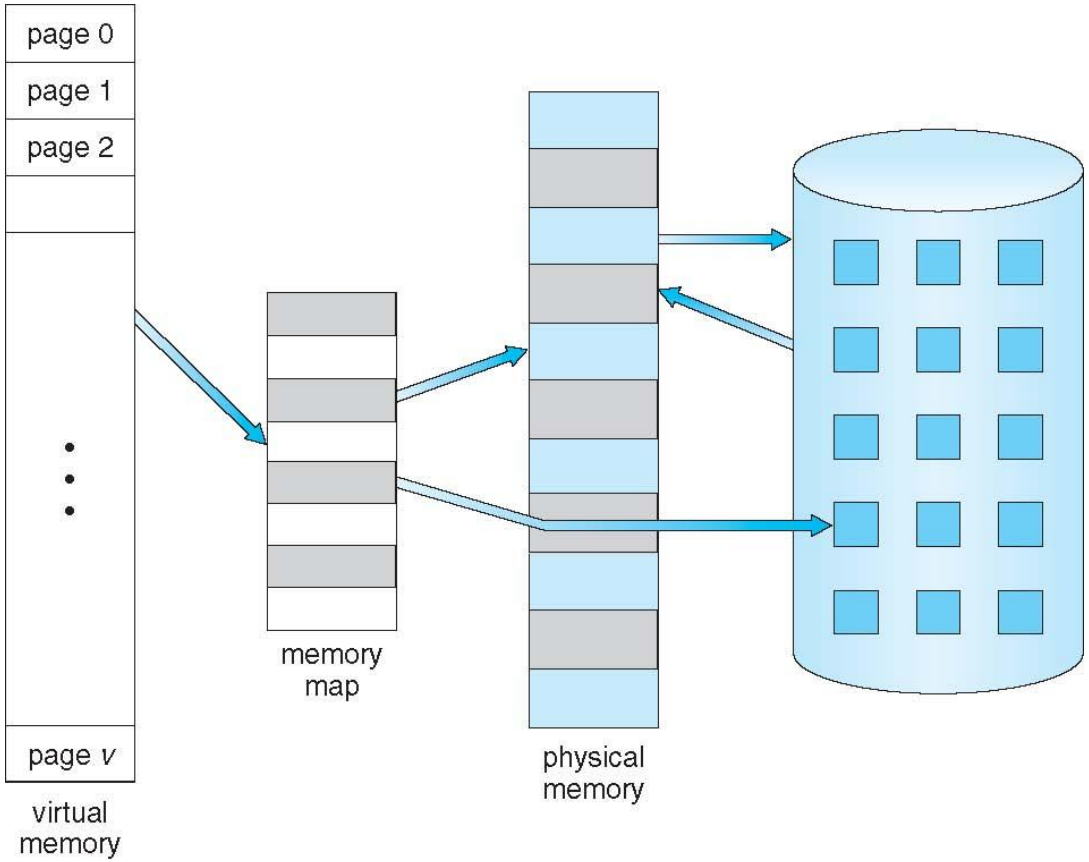
Virtual Memory

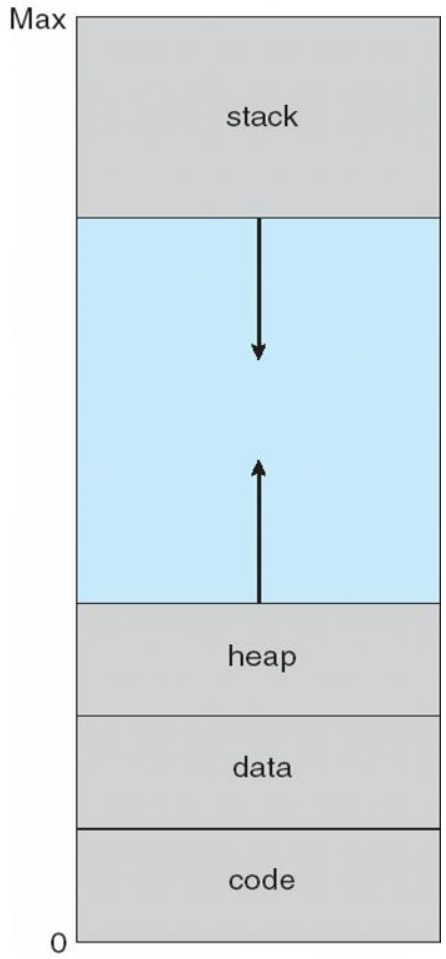
- Involves separation of user perceived logical memory from physical memory.
- Only part of the program needs to be in memory for execution.
- Logical address space can therefore be much larger than physical address space so proframmer no longer needs to worry about the amount of memory available.
- Allows address spaces to be shared by several processes.
- Allows for more efficient process creation.

Virtual Address Space

- It refers to the 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 That is Larger Than Physical Memory





The usual design logical address space suggests for stack to start at Max logical address and grow “down” while the heap grows “up”

It helps to maximizes address space use

Unused address space between the two is called a hole which is part of the LAS.

No real physical page is needed until heap or stack grows to a given new page

Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc

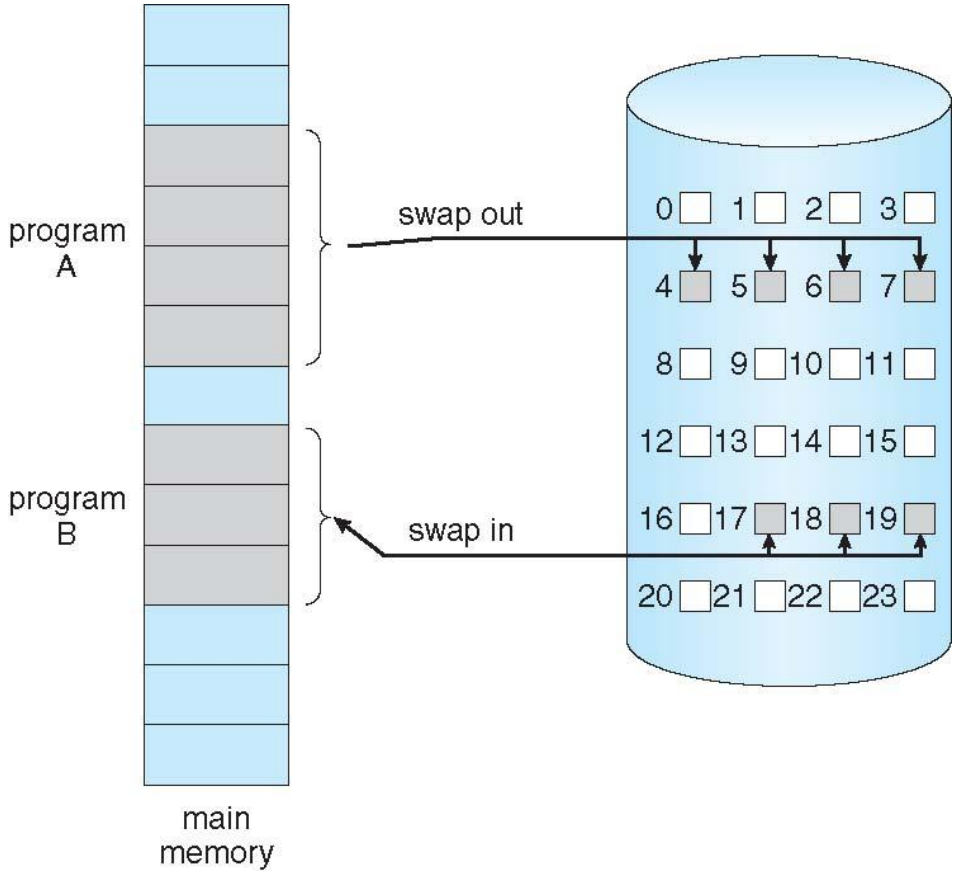
System libraries shared via mapping into virtual address space

Shared memory by mapping pages read-write into virtual address space

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

Demand Paging

- Loading of pages only when needed or demanded during execution is known as Demand Paging.
- Pages that are never accessed are thus never loaded into physical memory.
- It is similar to paging system with swapping where initially processes reside in secondary memory. When we want to execute a process, we swap it into memory.
- But rather than swapping the entire process into memory, we use a **lazy swapper**, which never swaps a page into memory unless needed.
- Since here we are interested in pages, we use the term **pager** henceforth instead of the term swapper.



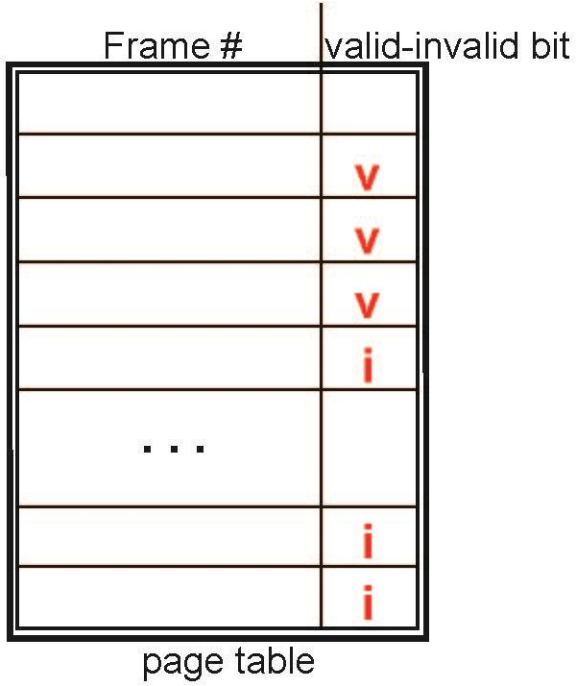
Working in Demand Paging

- When a page is needed, reference to it. If it is an invalid reference , abort it. If it is not-in memory then bring to memory
- Thus we need new MMU functionality to implement demand paging
- If pages needed are already memory resident, then no difference from non demand-paging
- If page needed and not memory resident, we need to detect and load the page into memory from storage
- But all this must be done without changing program behavior or without programmer needing to change code .
- **Advantages of Demand Paging** : Less I/O needed, Less memory needed, Faster response, More users

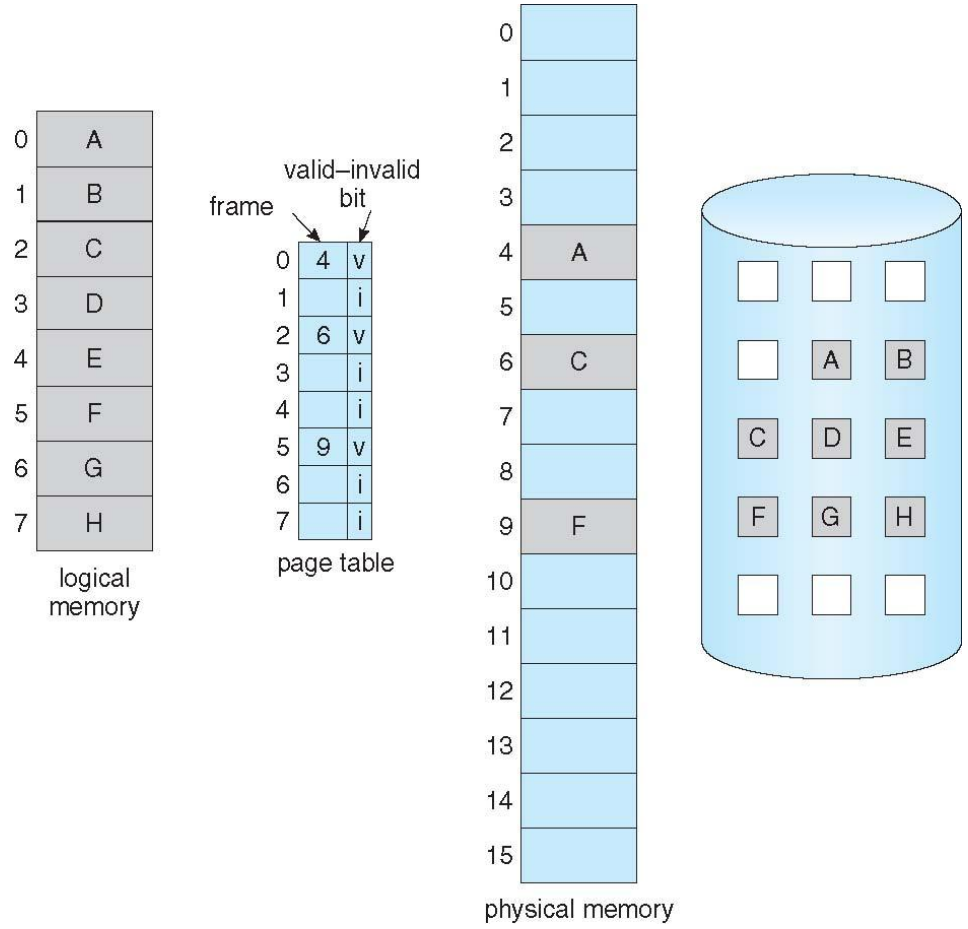
Valid-Invalid Bit in Demand Paging

- With each page table entry a valid–invalid bit is associated
- v means page in-memory – memory resident.
- i means not-in-memory

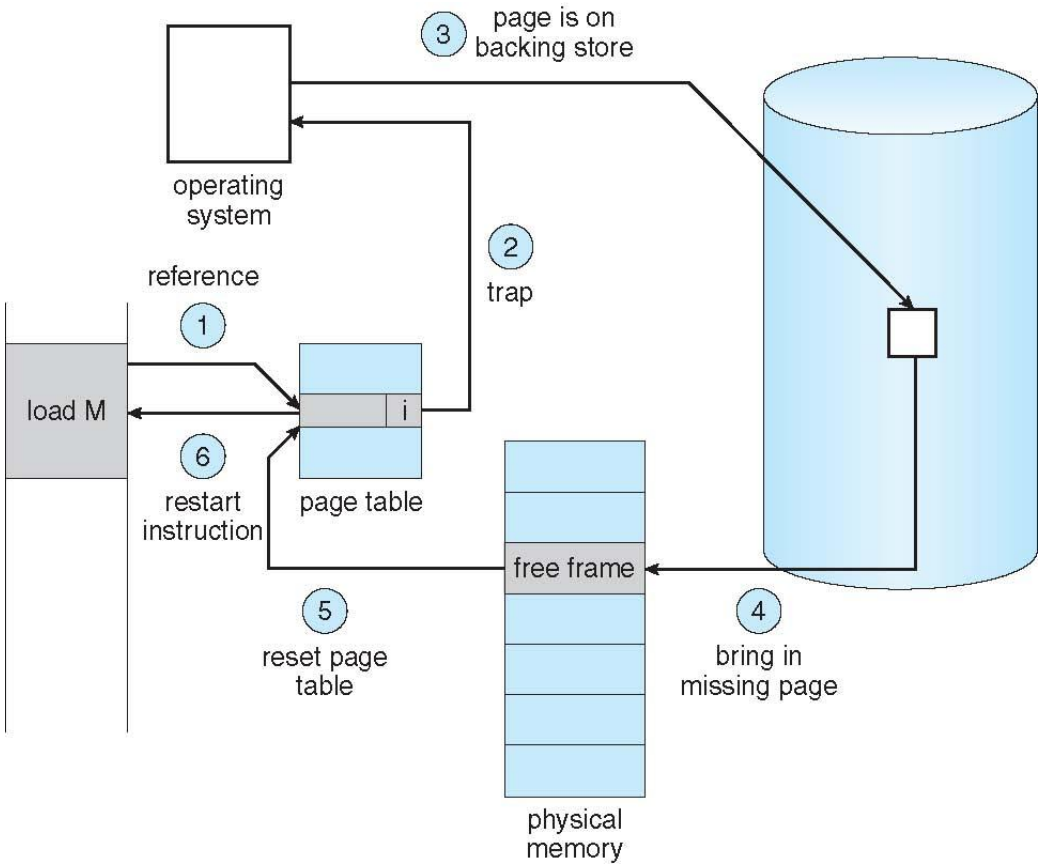
- Initially valid–invalid bit is set to i on all entries
- During MMU address translation, if valid–invalid bit in page table entry is i then we call it a **page fault.**
- **Means we are trying to access a page that is not brought in memory**



Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault



- If an invalid bit is set, a trap is caused to the operating system. This is because of the OS's failure to bring the desired page into memory.
- So initially, we check an internal page table stored in the PCB for a process to determine whether the reference was a valid or invalid memory access.
- If reference is invalid, terminate the process.
- If reference is valid but is not yet brought into memory a trap is invoked. So we got to bring it from disk store.
- To do this, we have to find a free frame from the free frame list.
- We then schedule a disk operation to read the desired page into the newly allocated frame.
- When disk read is complete, we modify the internal page table kept in PCB to indicate that page is now in memory.
- We again restart the instruction that was interrupted by the trap.
- Now, the process can access the page as though it was always in the memory.

Pure Demand Paging

- In extreme case, we can start executing a process with no pages in memory.
- When the OS sets the IP to first instruction of the process which is a non resident page so the process there is an immediate page fault.
- After this page is brought into memory, the process continues to execute, faulting as long as every page that it needs is in memory.
- After this we will have no more faults.
- This scheme is called pure demand paging which means never bring a page into memory until needed.

Locality of Reference

- Some programs could access several new pages of memory with each instruction execution possibly causing multiple page faults per instruction.
- This badly affects the system performance.
- locality of reference, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.
- So better bring the non resident pages to such location repetedly to enhance performance through demand paging.

Hardware to support Demand Paging

We as usual need

- **Page Table**: It enables to mark an entry invalid through valid - invalid bit or similar protection bits.
- **Secondary memory**: It holds those pages that are not present in main memory. It is usually a high speed disk. Section of the disk used for this purpose is called swap space.

Instruction Restart

After a page fault, it is crucial for demand paging to restart any instruction.

Since we save the state of the interrupted process when page fault occurs, we can restart it from exactly the same location and state.

If a page fault occurs during instruction fetch, we can restart by fetching the instruction afain.

If a page fault occurs during operand fetch, to restart, we must fetch and decode the instruction again and then fetch the operand again.

If page fault occurs while storing the result of some operation, to restart, we must fetch the instruction again, decode it again, fetch the operands again, perform the operation again and then store it again.

Major problems arise when one instruction modifies several location like MV. Basically it invols moving upto 256 bytes from source to destination. Suppose a fault occurs when the move is partially done, we cannot restart the instruction. **(solution??)**

Stages in 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:
    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

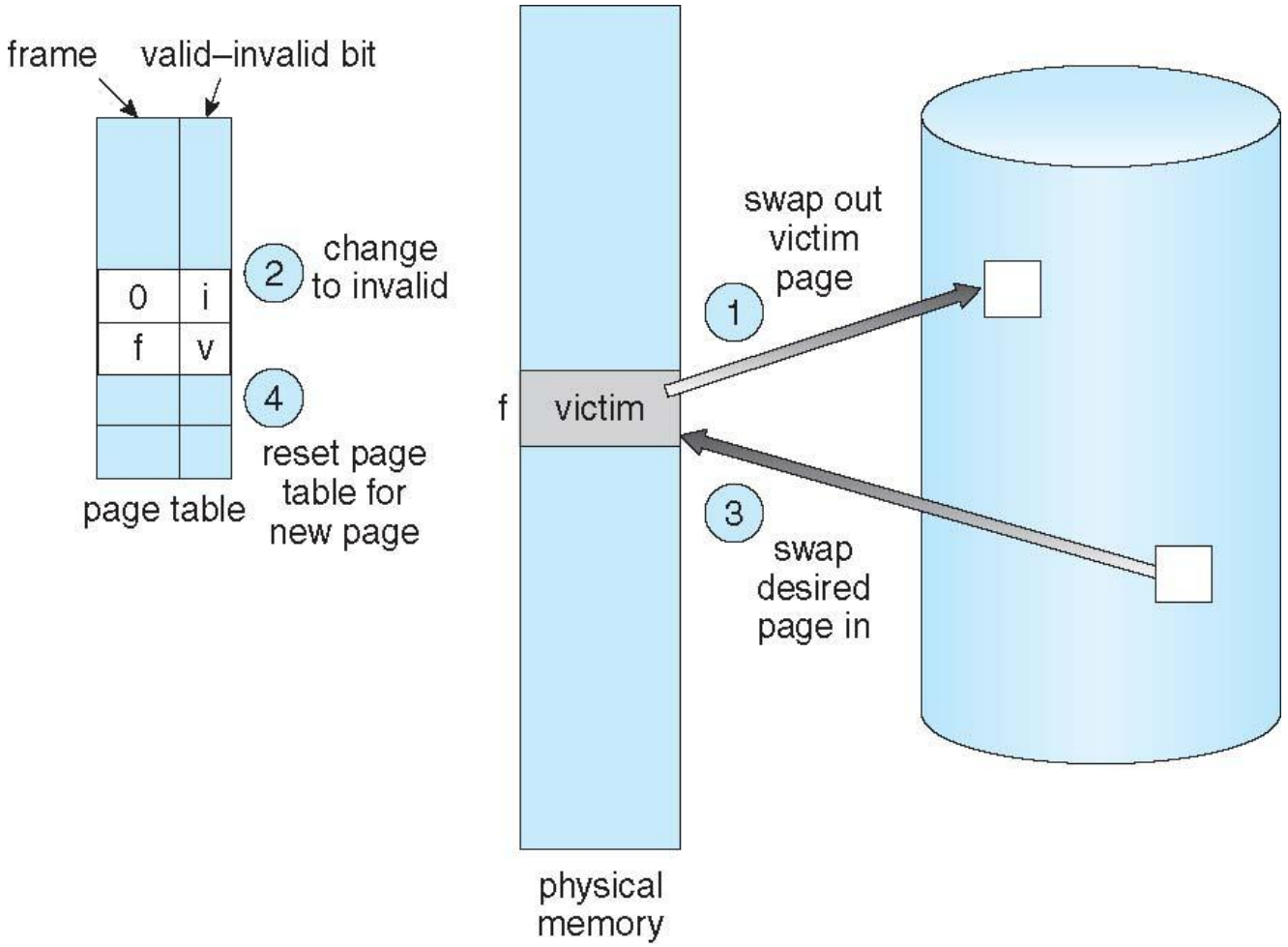
- Three major activities performed during Demand Paging
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- If no page fault, EAT = Time to access memory
- If page fault occurs, we have to read the page from disk and then access the desired location.
- Let Page Fault Rate be $0 \leq p \leq 1$ where if $p = 0$, no page faults and if $p = 1$, every reference is a faulty
- Effective Access Time (EAT) = $(1 - p)$ x memory access + p (page fault overhead + swap page out + swap page in)

Numerical discussed on whiteboard, Read Anonymous memory and Copy on Write from Book

Page Replacement

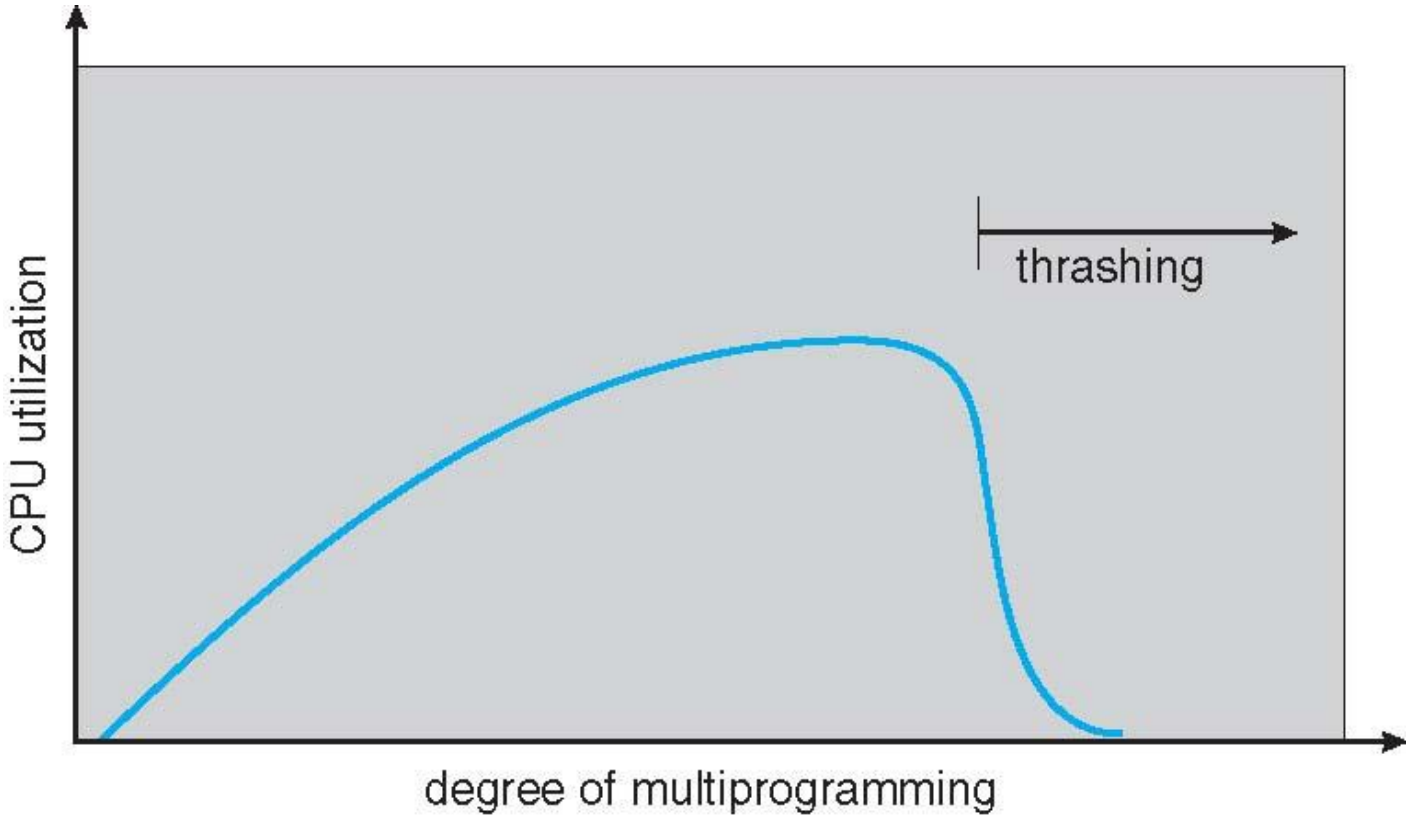
- If a page fault occurs and there are no free frames then we have to terminate the process. But this will reduce the system utilization and throughput.
- The second option is to swap out a process that is currently not being used, freeing all its frames and replace these with concerned frames.
- The basic steps will be
  - Find the location of the desired page on disk
  - Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a page replacement algorithm to select a victim frame
      - Write victim frame to disk if dirty
  - Bring the desired page into the (newly) free frame; update the page and frame tables
  - Continue the process by restarting the instruction that caused the trap

Numerical discussed on whiteboard on 3 Page Replacement Policies


frame
valid–invalid bit
0 i
f v
page table
2
change to invalid
4
reset page table for new page
f
victim
1
swap out victim page
3
swap desired page in
physical memory

Thrashing

- Whenever the OS sees that CPU utilization is less, it tries to increase the degree of multi programming.
- But the newly introduced processes start causing more page faults.
- Because of this, the CPU utilization further drops. However, the CPU scheduler keeps trying to increase the degree of multiprogramming.
- This phenomenon is called thrashing were no work is done because the processes spend all their time paging.
- It lowers the throughput and increases the page fault tremendously.



- As the degree of multiprogramming increases, CPU utilization increases slowly until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in and CPU utilization drops sharply.
- At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.