SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

# 18CSC205J-Operating Systems

## Unit- IV

# Syllabus

- Basics- Virtual Address Space
- Demand Paging
  - Page table, Handling Page faults, Performance of demand paging, Copy on write
- Page Replacement Algorithm
  - Pros and Cons of Page Replacement
  - FIFO, Optimal, LRU, LRU Approximation ,Counting based Page replacement algorithms.
- Allocation of Frames
  - Global Vs Local Allocation
  - MFC ,MVC
- Thrashing
  - Cause of Thrashing
  - Working Set Model

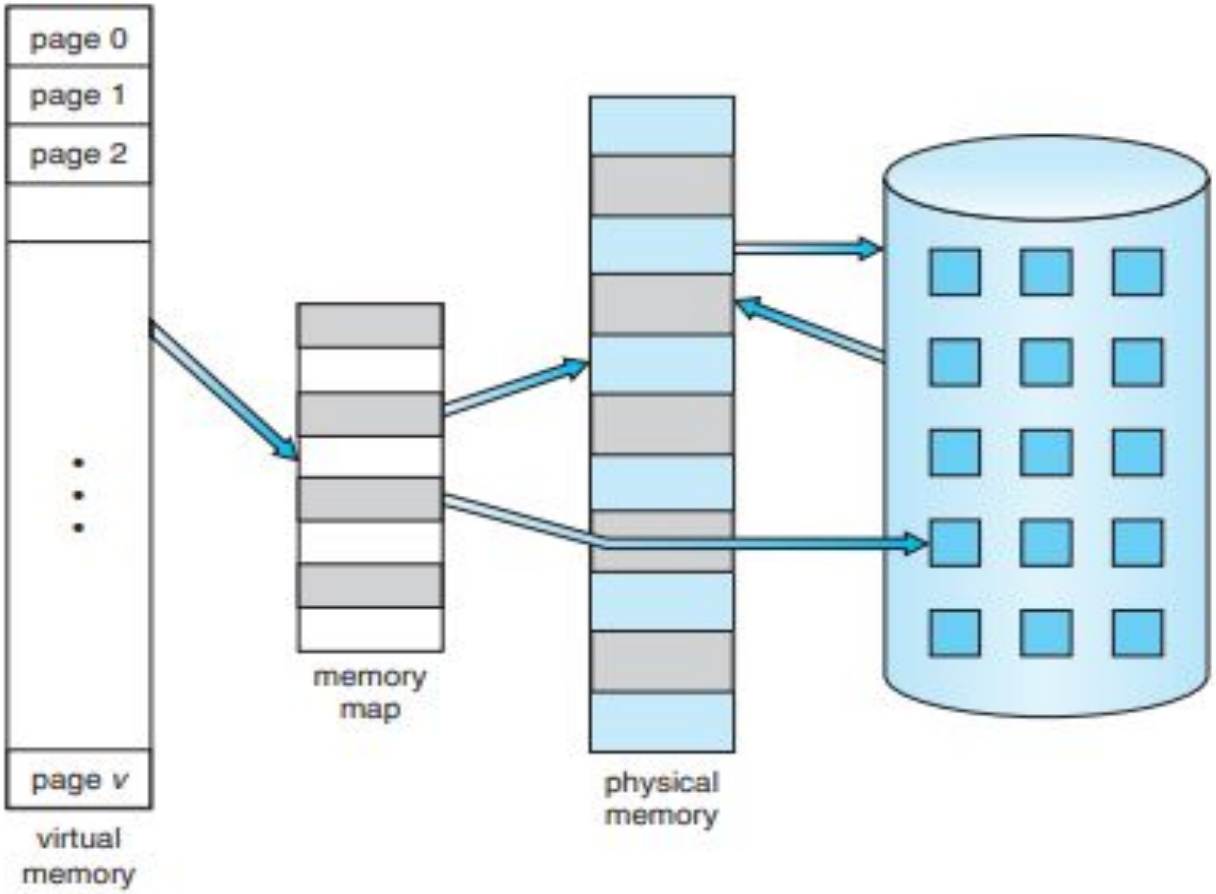
# Virtual Memory

- **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

- 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.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- In addition, it provides an efficient mechanism for process creation.

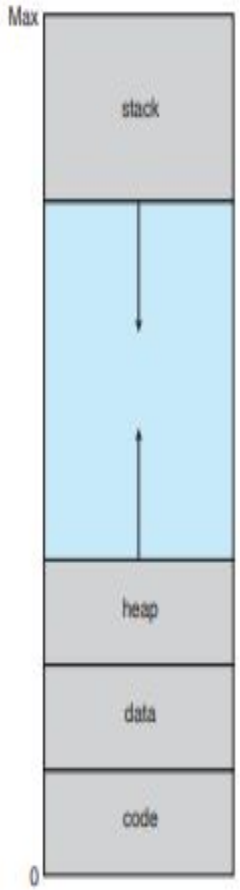
# Diagram Showing Virtual Memory Larger than Physical Memory



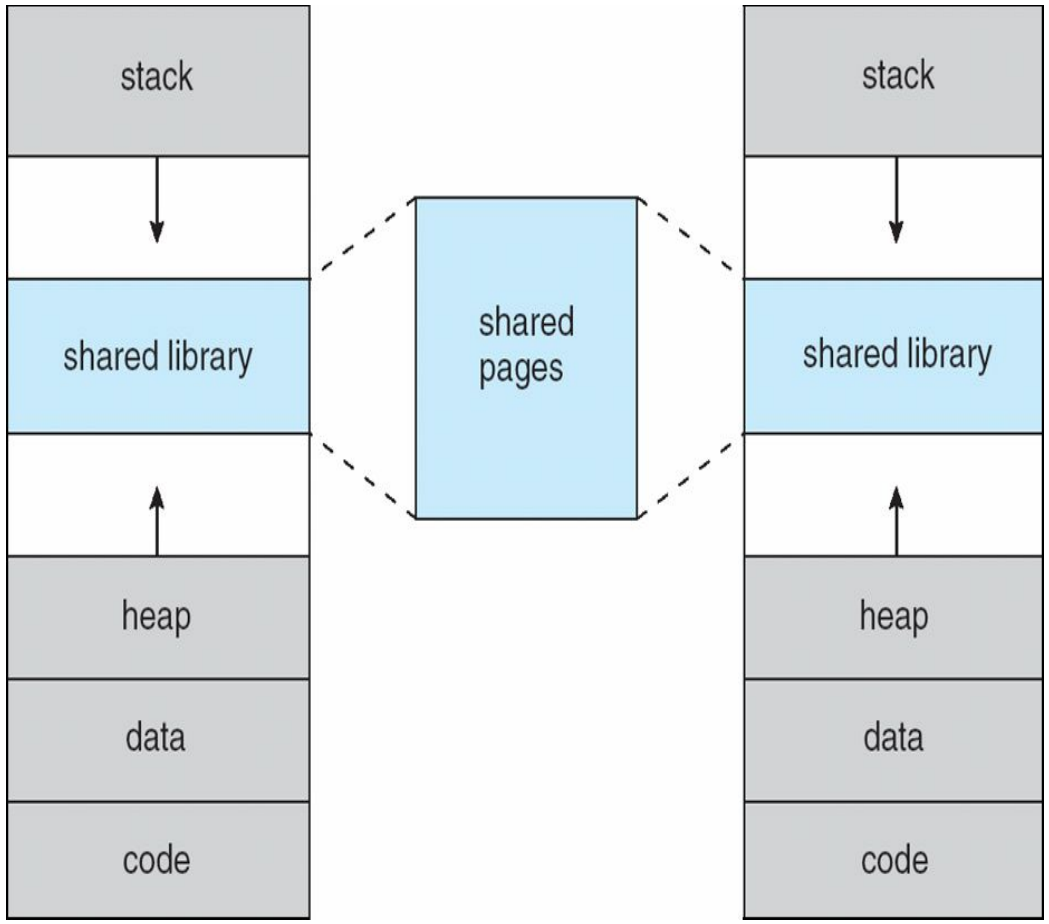
# Virtual address space

- **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 Address Space

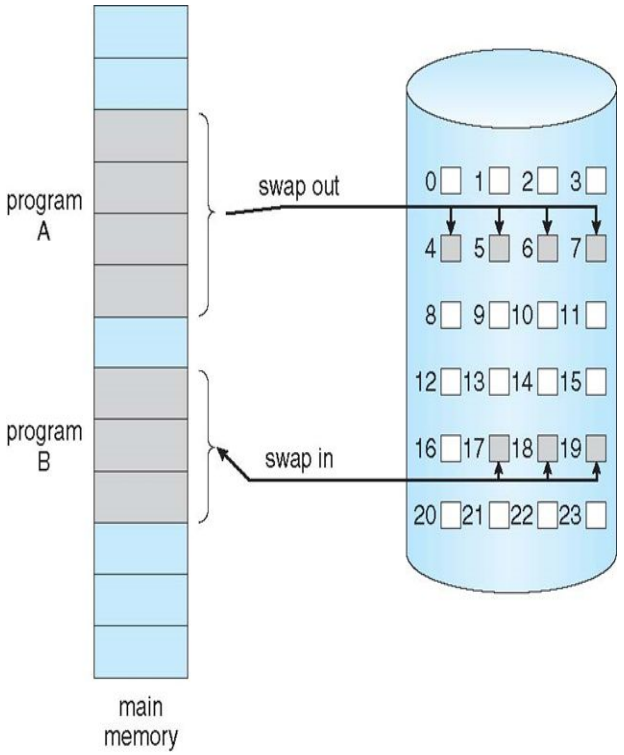


# Shared Library Using Virtual Memory



# 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 ⇒ 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**

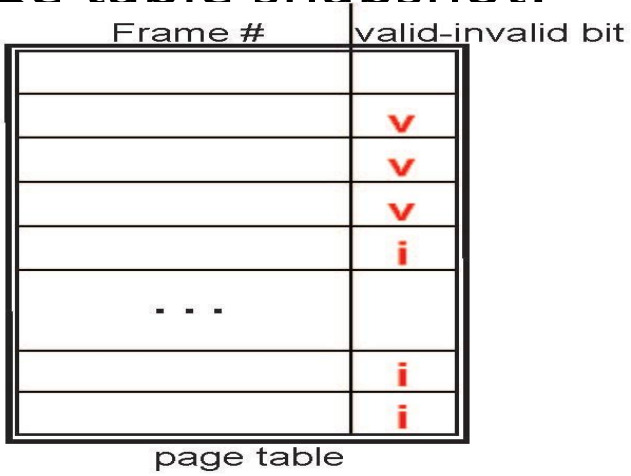


# Basic Concepts

- 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

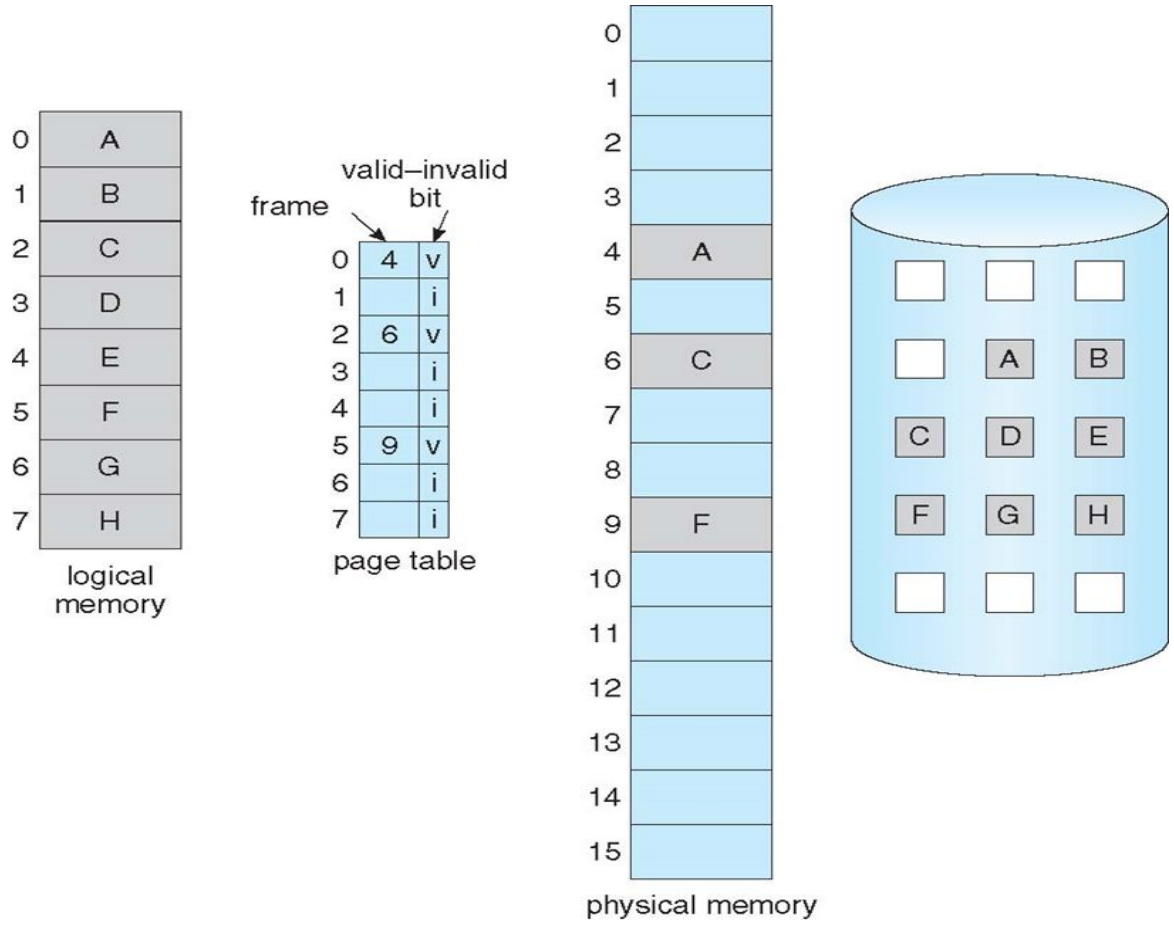
# Valid-Invalid Bit

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




During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

# Page Table When Some Pages Are Not in Main Memory



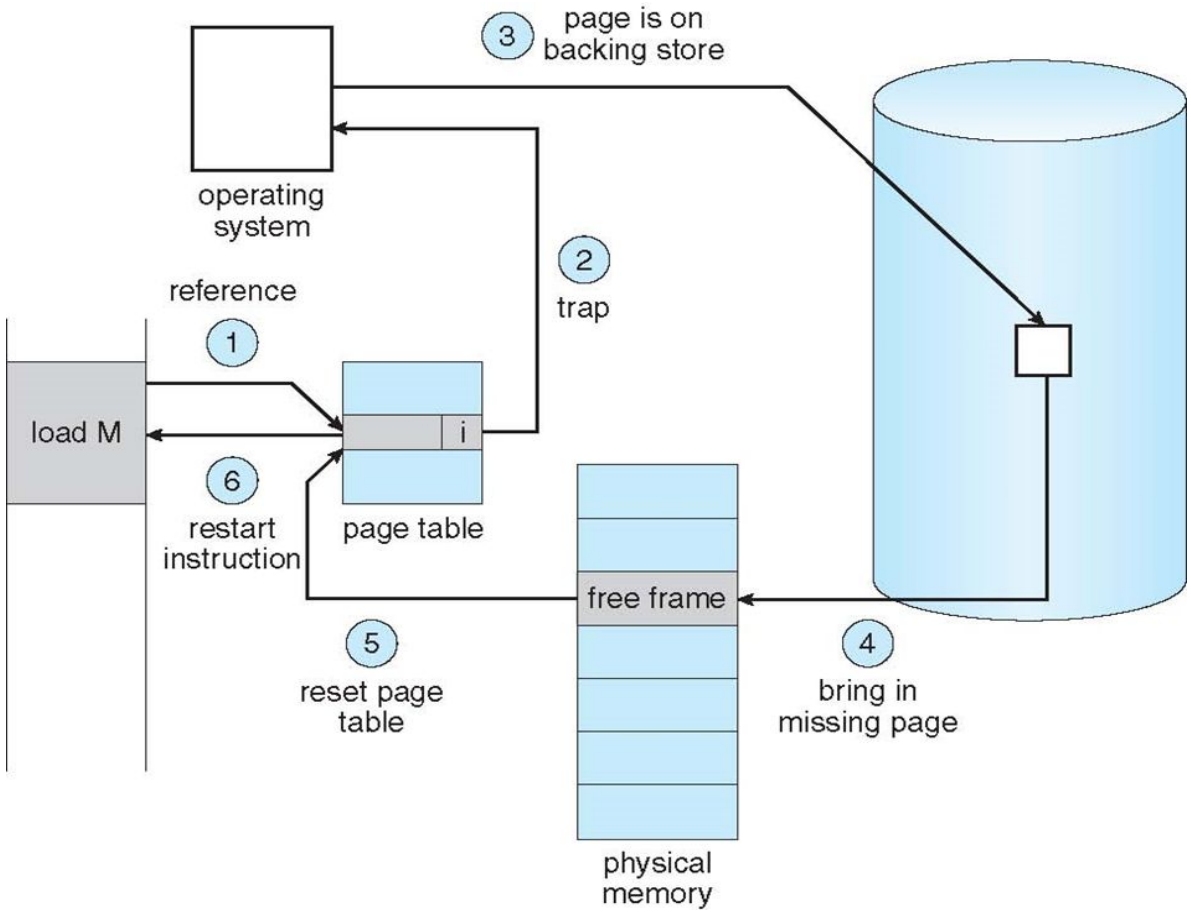
# Page Fault

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

  **page fault**

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ 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
  - 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**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

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

# Page replacement Algorithms

# Page Fault in OS

- A page fault occurs when a page referenced by the CPU is not found in the main memory.
- The required page has to be brought from the secondary memory into the main memory.
- A page has to be replaced if all the frames of main memory are already occupied.

# Page replacement

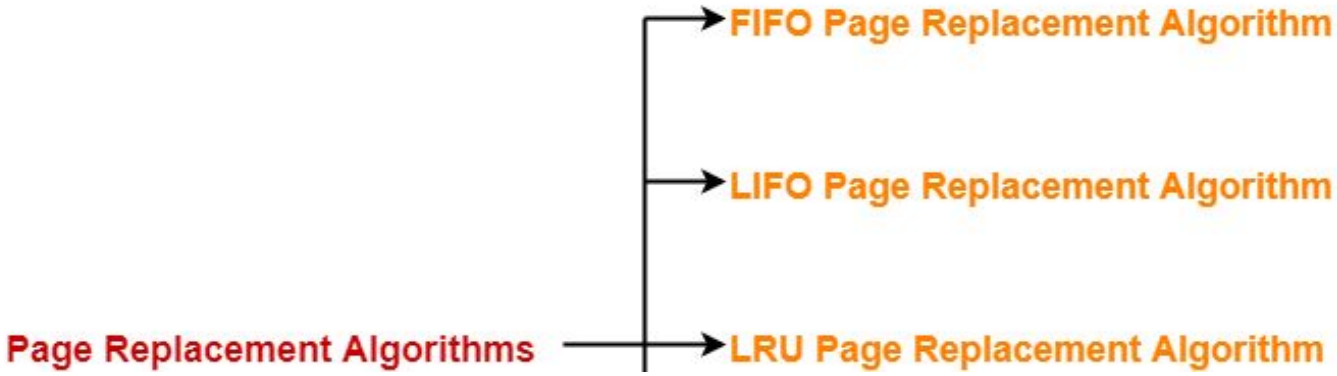
- Page replacement is a process of swapping out an existing page from the frame of a main memory and replacing it with the required page.

Page replacement is required when-

- All the frames of main memory are already occupied.
- Thus, a page has to be replaced to create a room for the required page.

# Page Replacement Algorithms

- Page replacement algorithms help to decide which page must be swapped out from the main memory to create a room for the incoming page.
- Various page replacement algorithms are-



# FIFO Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of "**First in First out**".
- It replaces the oldest page that has been present in the main memory for the longest time.
- It is implemented by keeping track of all the pages in a queue.

# LIFO Page Replacement Algorithm

- As the name suggests, this algorithm works on the principle of "**Least Recently Used**".
- It replaces the page that has not been referred by the CPU for the longest time.

# Optimal Page Replacement Algorithm

- This algorithm replaces the page that will not be referred by the CPU in future for the longest time.
- It is practically impossible to implement this algorithm.
- This is because the pages that will not be used in future for the longest time can not be predicted.
- However, it is the best known algorithm and gives the least number of page faults.
- Hence, it is used as a performance measure criterion for other algorithms.

# Example

**Consider a reference string: 4, 7, 6, 1, 7, 6, 1, 2, 7, 2. the number of frames in the memory is 3. Find out the number of page faults respective to:**

- FIFO Page Replacement Algorithm
- LRU Page Replacement Algorithm
- Optimal Page Replacement Algorithm

# FIFO Page replacement algorithms

## FIFO Page Replacement Algorithm

| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Miss | Hit |

**Number of Page Faults in FIFO = 6**

# LRU Page replacement algorithms

## LRU Page Replacement Algorithm

| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 7 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 2 | 2 | 2 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Miss | Hit |

**Number of Page Faults in LRU = 6**

# Optimal Page replacement algorithm

## Optimal Page Replacement Algorithm

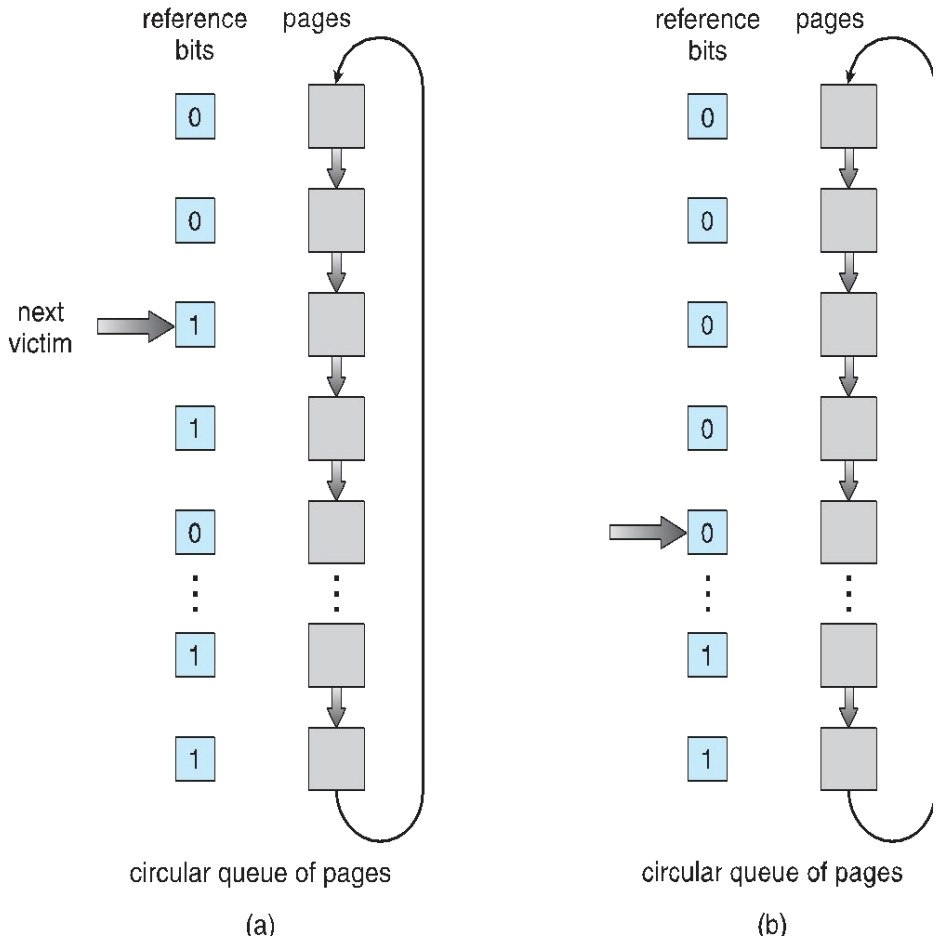
| Request | 4 | 7 | 6 | 1 | 7 | 6 | 1 | 2 | 7 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 3 | | | 6 | 6 | 6 | 6 | 6 | 2 | 2 | 2 |
| Frame 2 | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| Frame 1 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Miss/Hit | Miss | Miss | Miss | Miss | Hit | Hit | Hit | Miss | Hit | Hit |

**Number of Page Faults in Optimal Page Replacement Algorithm = 5**

# LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

# Second-Chance (clock) Page-Replacement Algorithm



# Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common

- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count

- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

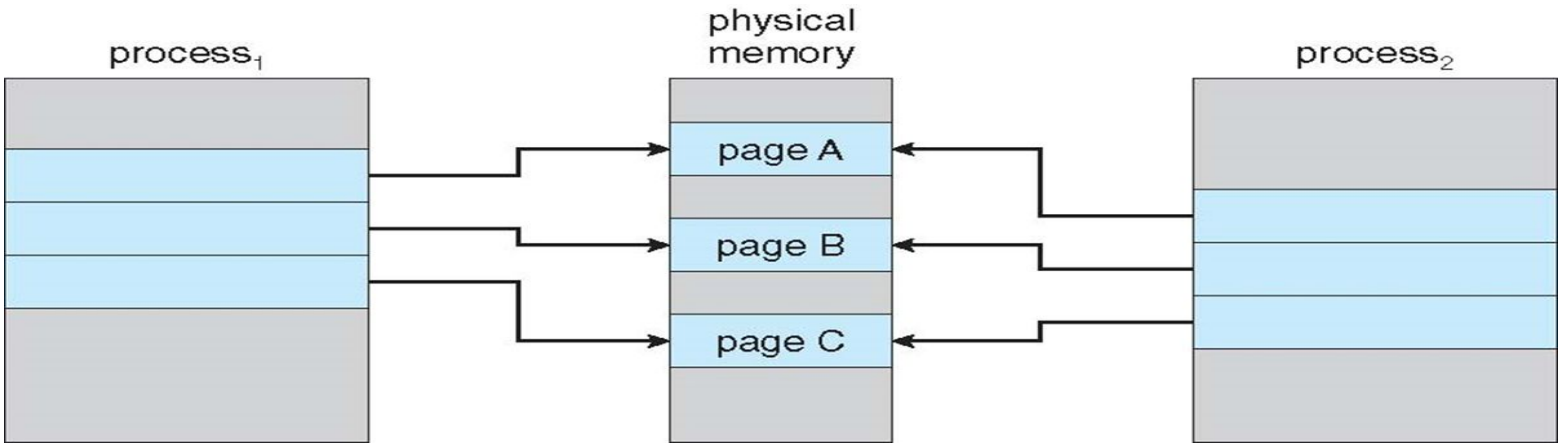
# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

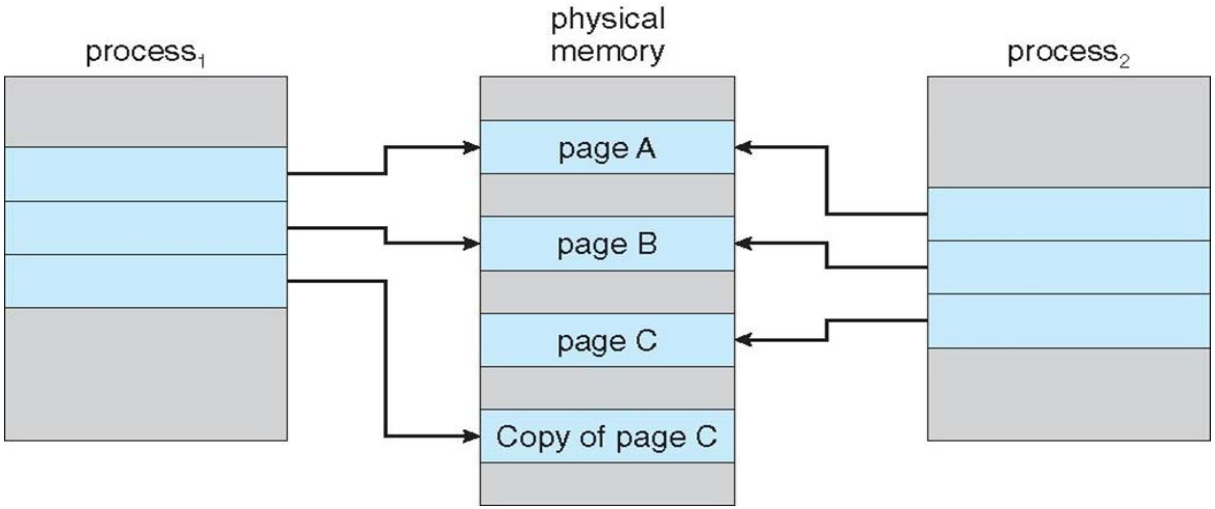
# Copy-on-Write

- **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?
- 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

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

# Frames

- When main memory is divided in to small fragments for paging purpose , the small fragments are called frames.
- whereas if the process is divided into small fragments for paging it is called page. ( Page_size = Frame_size )
- A frame refers to a storage frame or central storage frame.
- In terms of physical memory, it is a fixed sized block in physical memory space, or a block of central storage.
- In computer architecture, frames are analogous to logical address space pages.

# Frame Allocation Algorithm

- An important aspect of operating systems, virtual memory is implemented using demand paging. Demand paging necessitates the development of a page replacement algorithm and a **frame allocation algorithm**.
- Frame allocation algorithms are used if you have multiple processes; it helps decide how many frames to allocate to each process.
- There are various constraints to the strategies for the allocation of frames:
- You cannot allocate more than the total number of available frames.
- At least a minimum number of frames should be allocated to each process.
- This constraint is supported by two reasons. The first reason is, as less number of frames are allocated, there is an increase in the page fault ratio, decreasing the performance of the execution of the process. Secondly, there should be enough frames to hold all the different pages that any single instruction can reference.

# Types of Frame Allocation Algorithm

- The algorithms that are commonly used to allocate frames to a process are:
- **Equal allocation:** In a system with x frames and y processes, each process gets equal number of frames, i.e. x/y. For instance, if the system has 48 frames and 9 processes, each process will get 5 frames. The three frames which are not allocated to any process can be used as a free-frame buffer pool.
  - **Disadvantage:** In systems with processes of varying sizes, it does not make much sense to give each process equal frames. Allocation of a large number of frames to a small process will eventually lead to the wastage of a large number of allocated unused frames.

# Proportional Allocation

- **Proportional allocation:** Frames are allocated to each process according to the process size.
  For a process $p_i$ of size $s_i$, the number of allocated frames is $\mathbf{a_i = (s_i/S)*m}$, where S is the sum of the sizes of all the processes and m is the number of frames in the system.
- For instance, in a system with 62 frames, if there is a process of 10KB and another process of 127KB, then the first process will be allocated (10/137)*62 = 4 frames and the other process will get (127/137)*62 = 57 frames.
  - **Advantage:** All the processes share the available frames according to their needs, rather than equally.

# Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $P_i$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

# Comparison between Local and Global Replacement

- **Local replacement**: When a process needs a page which is not in the memory, it can bring in the new page and allocate it a frame from its own set of allocated frames only.
- **Advantage:** The pages in memory for a particular process and the page fault ratio is affected by the paging behavior of only that process.
- **Disadvantage:** A low priority process may hinder a high priority process by not making its frames available to the high priority process.

- **Global Replacement:**
- When a process needs a page which is not in the memory, it can bring in the new page and allocate it a frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
- **Advantage:** Does not hinder the performance of processes and hence results in greater system throughput.
- **Disadvantage:** The page fault ratio of a process can not be solely controlled by the process itself. The pages in memory for a process depends on the paging behavior of other processes as well.

# MFT-Multiprogramming with a Fixed number of Tasks

- MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition.
- The memory assigned to a partition does not change.

# MVT-Multiprogramming with Variable number of Tasks

- MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which each job gets just the amount of memory it needs.
- That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. **MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.**

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- Thrashing ≡ a process is busy swapping pages in and out.

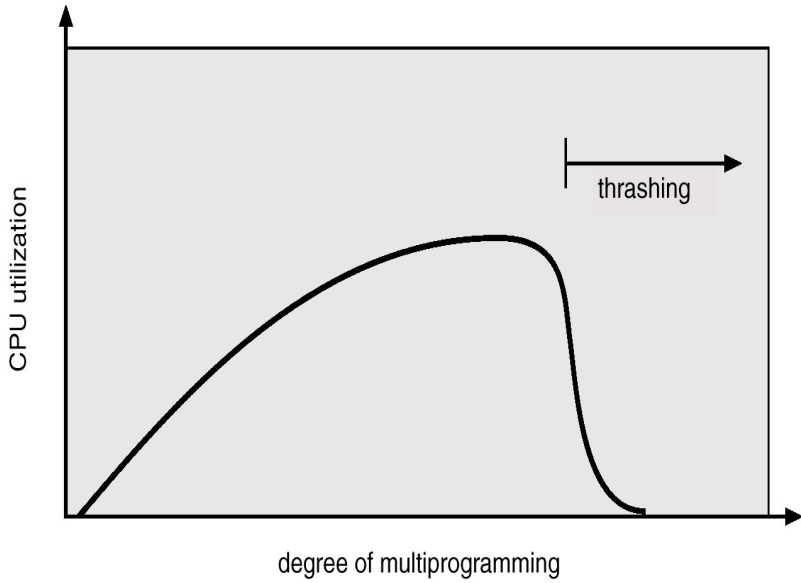
This high paging activity is called spending more time paging than executing.

A process is thrashing if it is

# Thrashing Diagram

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```




- Why does paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

# Thrashing

We can limit the effects of thrashing by using a **local replacement algorithm**(or **priority replacement algorithm**).

**local replacement** - if one process starts thrashing, it cannot steal frames from another process to thrash.

This problem is not entirely solved. If processes are thrashing, they will be in the queue for the paging device most of the time.

The average service time for a page fault will increase Thus, the effective access time will increase even for a process that is not thrashing.

# Locality Model

To prevent thrashing, we must provide a process with as many frames as it needs.

But how do we know how many frames it “needs”?
**Technique -** working-set strategy starts by looking at how many frames a process is actually using. This approach defines the

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together

A program is generally composed of several different localities that may overlap.

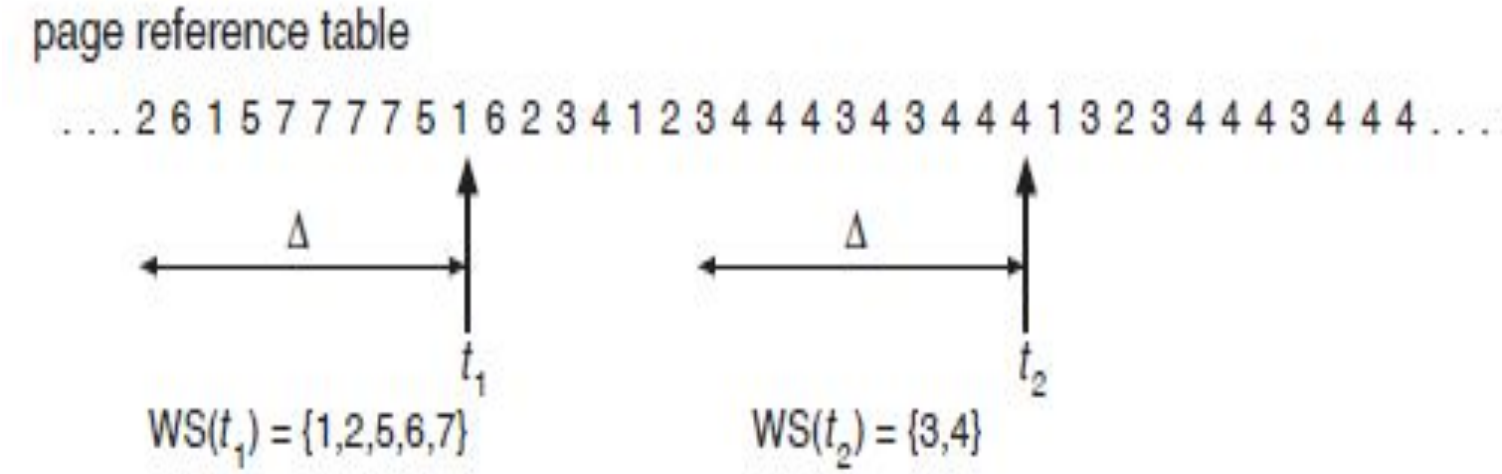
# Locality Model

- Why does paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size

# Working Set Model

- Δ ≡ working-set window ≡ a fixed number of page references
  Example: 10,000 instruction
- $WSS_i$ (working set of Process $P_i$) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality.
  - if Δ too large will encompass several localities.
  - if Δ = ∞ ⇒ will encompass entire program.
- $D = \Sigma\ WSS_i$ ≡ total demand frames
- if $D > m$ ⇒ Thrashing
- Policy if $D >$ m, then suspend one of the processes.

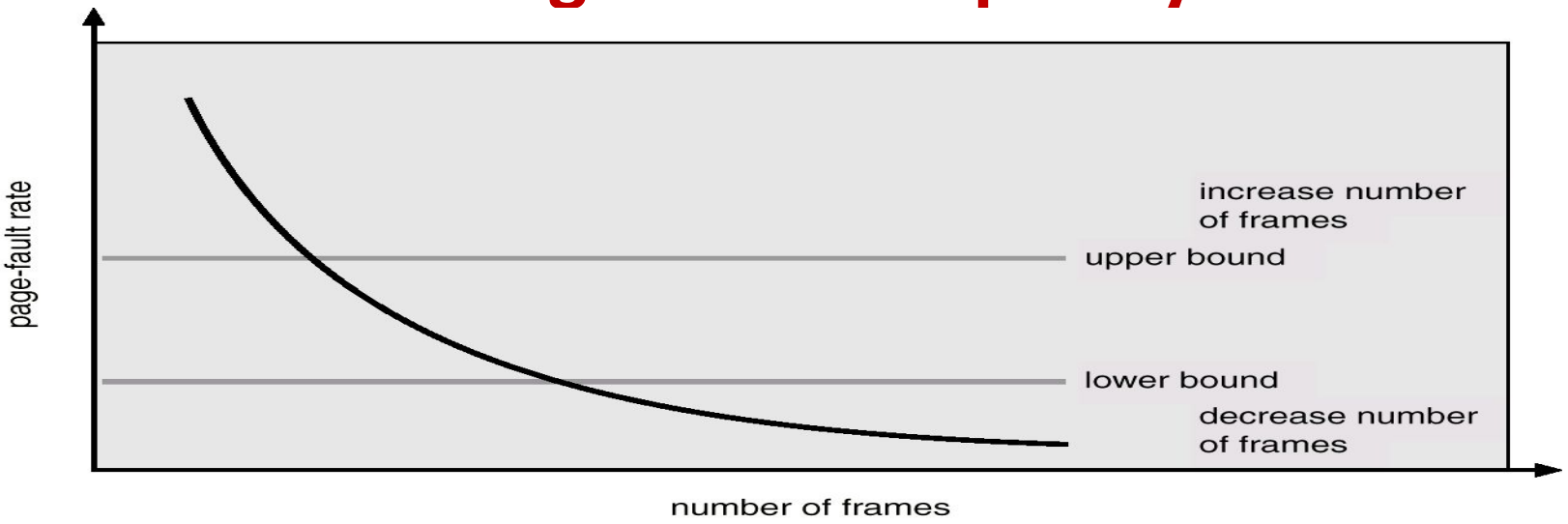
# Working Set Model



# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 ⇒ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units.

# Page-Fault Frequency




- Establish “acceptable” page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.