Unit IV: to Memory Management

Introduction of Memory management

Memory management is a critical function of a computer system, primarily handled by the operating system. It involves the **process of controlling and coordinating computer memory** , assigning blocks to various running programs to optimize overall system performance.

In simpler terms, memory management ensures that each process has enough memory to execute while maximizing the efficiency of the available memory resources.

Memory management in an opera□ng system (OS) is the process of e□ciently alloca□ ng, tracking, and op□ mizing the main memory (RAM) for various running programs and processes, ensuring smooth mul□tasking, system stability, and e□cient resource u□liza□on. It involves techniques like memory alloca□on, dealloca□on, protec□on, and sharing, using methods such as paging and segmenta□on to manage memory e□ec□vely and prevent issues like memory leaks and fragmenta□on

What is Memory Management?

Function: It's a fundamental OS function responsible for the organization and control of computer memory.

Objective: To provide processes with the memory they need to execute, and to do so efficiently, securely, and in an organized manner.

Scope: Memory management handles allocating and deallocating space, tracking free and occupied memory locations, and moving data between main memory and secondary storage (like hard drives)

Why is Memory Management Important?

- **Efficient Resource Utilization** : Prevents memory wastage and ensures optimal use.
- **Multitasking Support** : Enables multiple programs to run simultaneously without interfering with each other.
- **System Stability** : Prevents memory leaks and unauthorized memory access, protecting the system from crashes or security breaches.
- **Faster Execution** : Reduces overhead and delays by managing memory access smartly.

Key Functions of Memory Management

1. **Allocation and Deallocation**
   - Allocates memory to processes when requested.
   - Frees memory when it's no longer needed.
2. **Address Mapping**
   - Translates logical addresses (used by programs) to physical addresses (actual RAM locations).
3. **Memory Protection**
   - Ensures that one process cannot access memory allocated to another process.
4. **Swapping and Paging**
   - Moves data between RAM and disk storage to manage memory efficiently, especially when RAM is full.
5. **Garbage Collection (in high-level languages)**
   - Automatically reclaims memory that is no longer in use.

Key Responsibilities

- **Memory Allocation:** Assigning memory space to processes and applications when they need it.
- **Memory Deallocation:** Reclaiming memory that is no longer in use by a process.
- **Memory Tracking:** Keeping a record of which memory locations are free and which are occupied.
- **Memory Protection:** Ensuring that one process cannot access or corrupt the memory space of another process.
- **Efficient Utilization:** Maximizing the use of the limited main memory available to the system

Why is it Important?

- **Multitasking:** Enables a computer to run multiple applications and processes simultaneously by dividing memory among them.
- **System Performance:** Optimizes system performance by ensuring fast and seamless memory access for processes.
- **Stability:** Prevents crashes and ensures the overall stability of the system by managing memory resources effectively.

- **Security:** Protects processes from interfering with each other, contributing to the security of the system

Common Techniques

- **Paging:** Divides memory into fixed-sized blocks (pages) and maps them to memory, solving the problem of external fragmentation.
- **Segmentation:** Divides memory into variable-sized blocks (segments) based on logical program units.
- **Swapping:** Moves entire processes or pages between main memory and secondary storage to free up space in RAM

Types of Memory

- **Primary Memory (RAM)**
- **Secondary Memory (Hard disk, SSD)**
- **Cache Memory**
- **Virtual Memory**

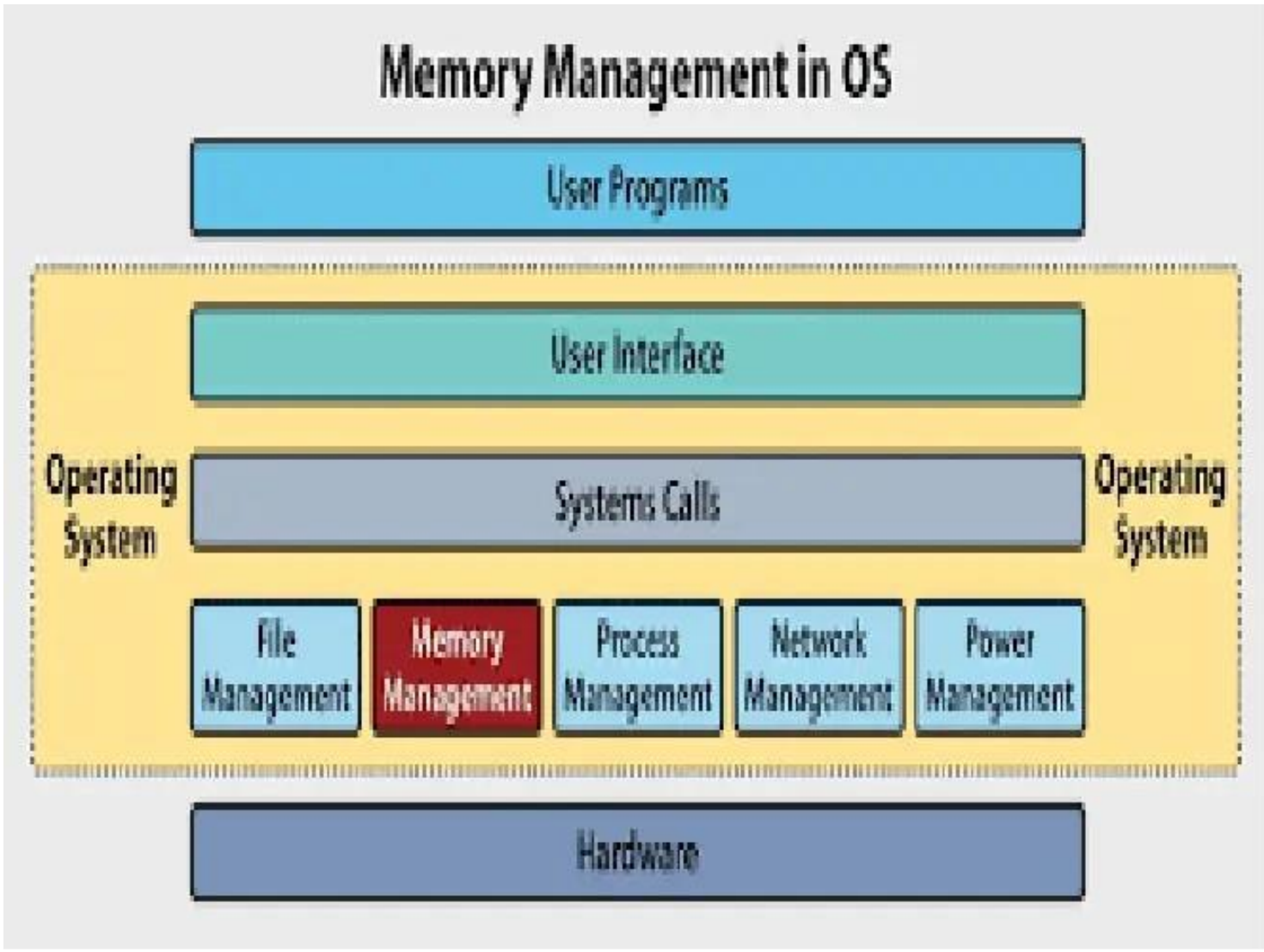


Fig: Memory Management

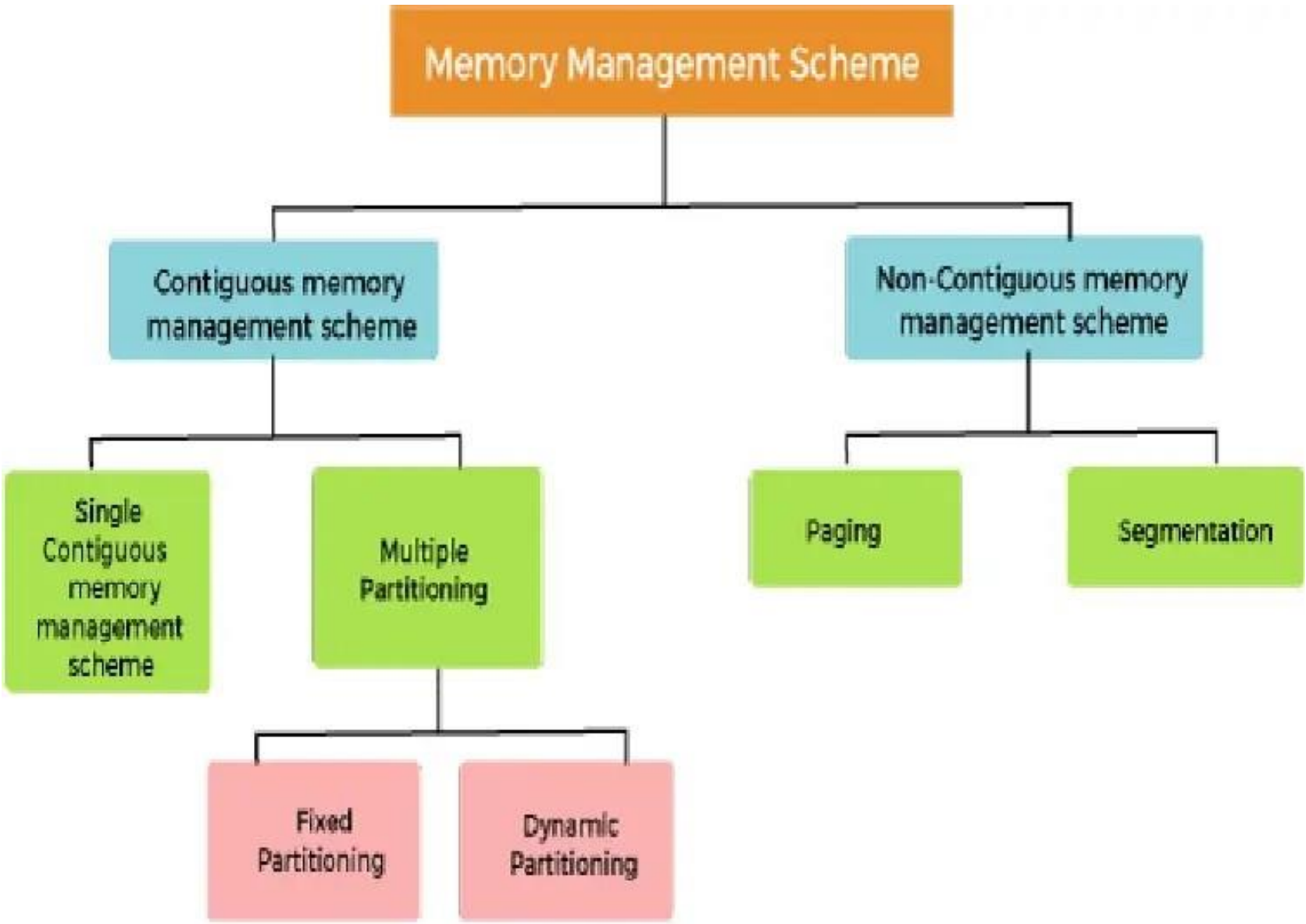
Memory management is a cri□cal aspect of opera□ng systems that ensures e□cient use of the computer's memory resources. It controls how memory is allocated and deallocated to processes, which is key to both performance and stability. Below is a detailed overview of the various components and techniques involved in memory management.

Memory allocaton s□ ra□ egies

Memory alloca□on strategies in an opera□ng system manage how processes are assigned memory, broadly categorized into con□guous memory alloca□on (assigning con□nuous memory blocks) and non-con□guous memory alloca□on

(assigning non-adjacent blocks). Con□guous methods include □xed par□□onsand

dynamic par□□ons, using placement algorithms like First-Fit, Best-Fit, and Worst-Fit to select memory blocks. Non-con□guous methods like paging and segmenta□on provide greater □exibility by dividing memory into □xed-size pages or logical segmentsFF Fi



Classification of memory management schemes

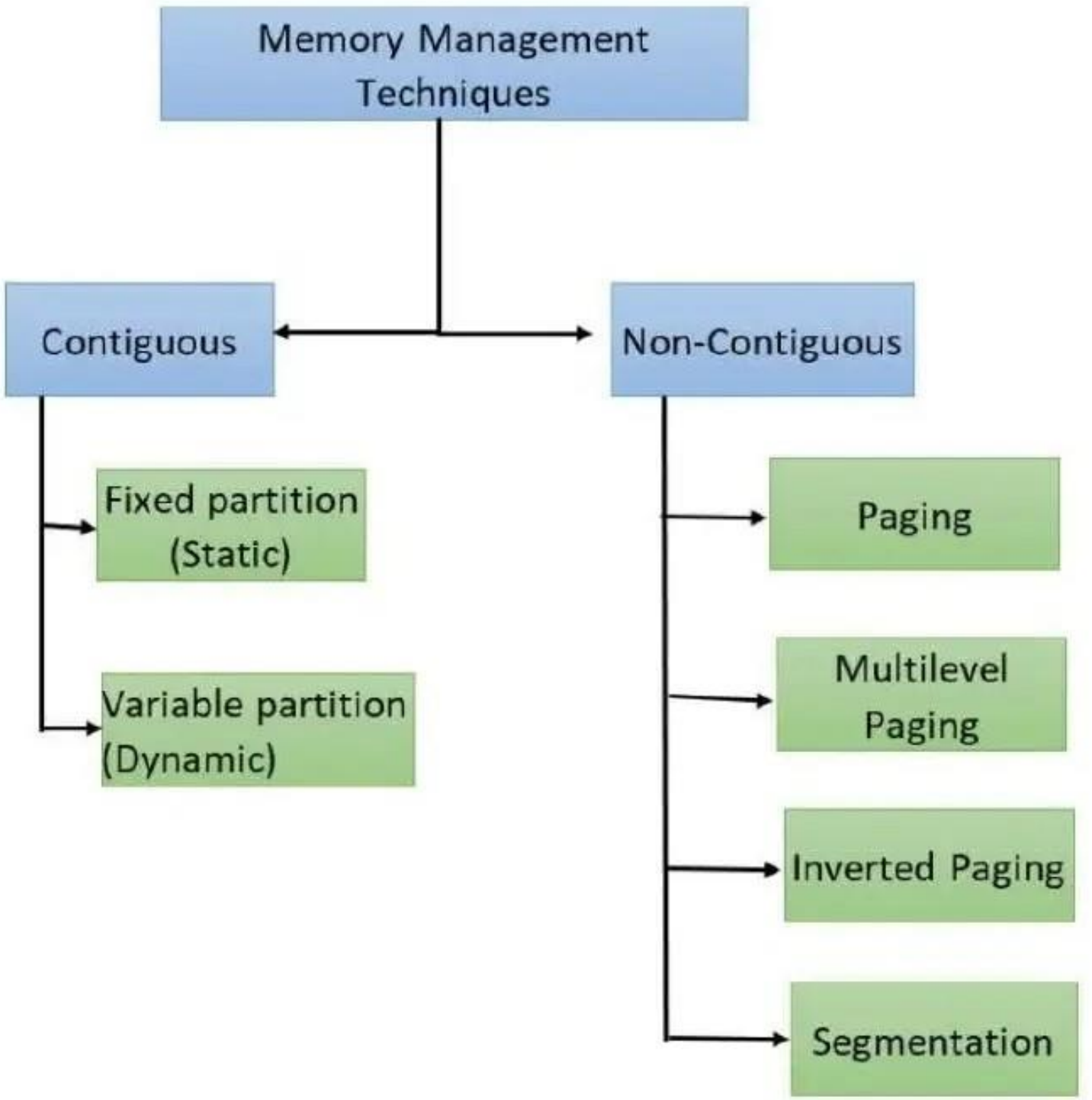


Fig: Memory Managemen Techniques

Contiguous Memory Allocation

This strategy allocates a single, continuous block of memory to a process.

- **Fixed Partition (or Static Partition) Allocation:**

Memory is divided into a fixed number of partitions of predetermined sizes.

Placement Algorithms: When a process needs memory, the OS uses algorithms to decide which free partition to assign.

- **First-Fit:** Allocates the first free partition found that is large enough for the process.
- **Best-Fit:** Allocates the smallest free partition that can accommodate the process.
- **Worst-Fit:** Allocates the largest available free partition, which can leave a useful remaining block.

- **Dynamic (Variable) Partition Allocation:** Memory is divided into variable-sized partitions based on the process's actual memory needs. This can reduce internal fragmentation (unused memory within an allocated block) but can lead to external fragmentation (free memory scattered in small, unusable blocks).

Non-Contiguous Memory Allocation This approach allows a process to occupy non-adjacent blocks of memory.

- **Paging:** Memory is divided into fixed-size blocks called pages for processes and fixed-size blocks called page frames in main memory. Paging is crucial for virtual memory, allowing processes larger than physical RAM to run by swapping pages between RAM and secondary storage.
- **Segmentation:** Memory is divided into logical units called segments, which correspond to logical parts of a process (like code, data, or stack). Segments don't have to be in contiguous memory locations.

Summery table :

Strategy/Partitioning	Description	Advantages	Disadvantages
First Fit	Allocate first suitable block	Fast allocation	External fragmentation
Best Fit	Allocate smallest suitable block	Minimizes leftover space	Slow search, fragmentation
Worst Fit	Allocate largest block	Leaves large free blocks	Inefficient memory use
Fixed Partitions	Fixed-size memory blocks	Simple, fast	Internal fragmentation
Dynamic Partitions	Variable-size blocks	Better memory utilization	External fragmentation,

Paging

Paging is an opera□ng system memory management technique that divides a process into □xed-size units called pages and physical memory into equally sized blocks called frames. Pages from secondary storage are loaded into memory frames, allowing for non-con□guous alloca□on and elimina□ng external fragmenta□on. This process enables more processes to run, even those larger than physical memory, and is a crucial part of virtual memory management in modern opera□ng system

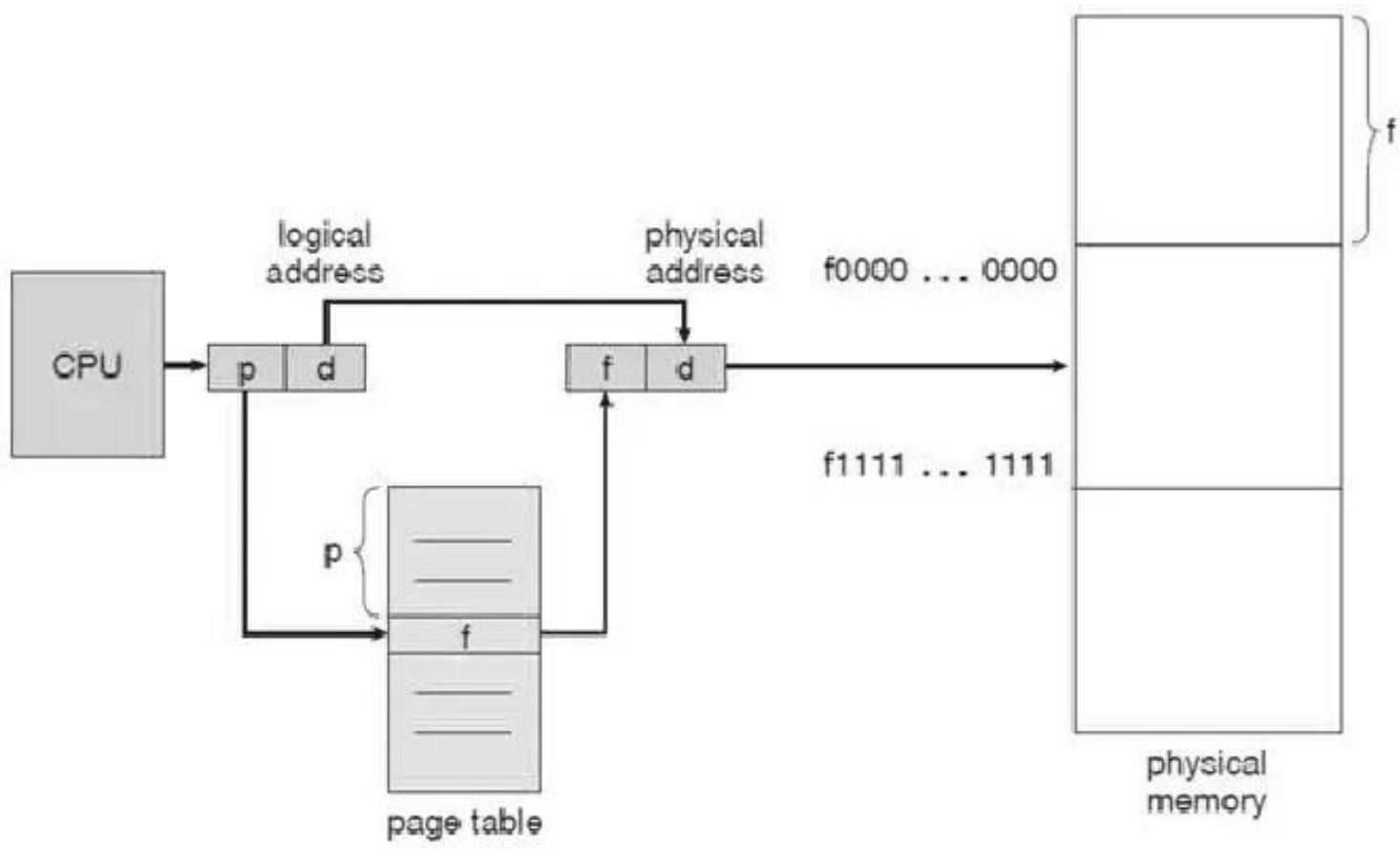


Fig: Example of paging

Paging divides a process's memory into □xed-size blocks (pages) and maps them to □xed-size frames in physical memory using a page table, which contains entries mapping virtual page numbers to physical frame numbers. Segmenta□on divides a process into variable-size logical units called segments, providing a user-centric view of memory. Virtual memory, enabled by demand paging, loads pages into physical memory only when they are needed, reducing I/O and allowing processes to be larger than available RAM by using secondary storage as an extension of main memory

Paging

- **Structure of a Page Table:** A page table is a data structure used by the operating system to translate virtual addresses to physical addresses.
  - **Entries:** Each entry in a page table typically corresponds to a virtual page and contains the frame number where that page is stored in physical memory, along with other control bits like protection and valid/invalid bits.
  - **Virtual Address Components:** A virtual address is divided into a page number and an offset.
  - **Translation Process:** The page number is used to find the corresponding entry in the page table, which provides the frame number. The offset is then added to this frame number to produce the final physical address.

  - **Multi-level Page Tables:** For large address spaces, multi-level page tables are used to avoid large, monolithic tables that consume too much memory.
  - **Inverted Page Tables:** An alternative approach where there is one entry for each physical frame, storing the virtual page number and process owner of the page.

Segmentation

- **User-Centric Division:** Segmentation organizes a program into logical units called segments, which are variable in size.
- **Logical Structure:** Segments provide a natural way for programmers to

structure their programs, with each segment often representing a logical part of the program (e.g., code, data, stack).

- **Protection:** Segmentation offers improved protection by isolating each segment, preventing it from affecting other parts of the process's memory

Segmentation is a memory management technique in operating systems that divides a program into logically related parts called segments of varying sizes, such as code, data, and stack. The OS maintains a segment table for each process, which stores the base address and size (or limit) for each segment, allowing the CPU to map a logical address (segment number and offset) to a physical memory location. This approach provides a user's-view of memory, enabling protection and more efficient allocation for different program components.

How it Works

Division: A process is divided into logical units (segments) like code, data, stack, etc., each of a different, variable size.

Segment Table: The operating system maintains a segment table for each process, which contains information about each segment.

Logical Address: When a program needs to access a memory location, it generates a logical address consisting of a segment number and an offset within that segment.

Translation: The segment table is used to find the base address (starting point) of the corresponding segment in physical memory.

Physical Address: The CPU calculates the physical address by adding the base address to the offset.

Fig: Segmentation

Key Characteristics

- **Logical Structure:** Segmentation reflects the logical structure of a program, making it more intuitive for users and programmers.
- **Variable-Sized Segments:** Unlike paging (which uses fixed-size pages), segments are of varying sizes, adapting to the program's needs.
- **User-Centric:** It aligns memory management with how the user sees the program.
- **Protection:** Segments can be protected, allowing only specific processes to access them, enhancing security.

Advantages

- **Improved CPU Utilization:** By loading entire, logically related modules, memory is used more effectively, and internal fragmentation is avoided.

- **User Protection:** Each segment can be granted different access permissions, ensuring that one part of a program cannot interfere with another.

Disadvantages

- **External Fragmentation:** Because segments are of unequal, variable sizes, free memory can become scattered into small, unusable chunks (external fragmentation).
- **Overhead:** Maintaining segment tables for each process requires extra memory and management time.
- **Slower Memory Access:** Accessing a memory location requires two lookups: first in the segment table and then in the main memor

Virtual Memory: Background

- **Purpose:** Virtual memory creates an illusion of a much larger main memory than what is physically available, allowing programs to be larger than RAM and multiple processes to run concurrently.
- **Mechanism:** It achieves this by using secondary storage (like a hard drive) as an extension of RAM to store parts of programs that are not currently in use.

Virtual memory uses a por□on of secondary storage (like a hard drive) to extend the physical RAM (main memory), allowing the system to run larger programs or more programs than its RAM could normally hold. A hardware component, the Memory Management Unit (MMU) , along with the Opera□ng System (OS), translates program-generated virtual addresses into the physical addresses in RAM or secondary storage, crea□ng the illusion of a larger, con□guous memory space.

How it works :

- **Memory Illusion:** The OS creates the illusion of a much larger memory space than the physical RAM available.
- **Address Translation:** When a program needs to access memory, it uses a virtual address. The MMU, a hardware component, translates this virtual address into a physical address.
- **Paging:** This translation typically happens using a technique called paging, where memory is divided into fixed-size blocks called pages. The OS uses a page table to map virtual page numbers to physical memory frames.

- **Page Faults:** If the required page is not in RAM (a page fault occurs), the OS retrieves it from secondary storage (swap space or the page file) and loads it into a RAM frame, potentially swapping out an unused page to make space.
- **Translation Lookaside Buffer (TLB):** To speed up this process, the MMU uses a cache called a TLB that stores frequently used virtual-to-physical address translations.

Fig: Virtual Memory

Benefits

- **Larger Programs:** Allows users to run applications that are larger than the available physical RAM.
- **Increased Concurrency:** Enables the system to run more applications simultaneously by efficiently managing memory usage between them.
- **Improved Performance:** By effectively managing memory, the system can operate more quickly and efficiently

Demand Paging

- **Lazy Loading:** Demand paging is a specific virtual memory technique where pages are loaded from secondary storage into main memory only when they are actually needed by the program.
- **Page Fault:** When the CPU accesses a virtual address whose corresponding page is not in physical memory, a "page fault" occurs.
- **Page Fault Handling:** The valid-invalid bit in the page table entry indicates the page is not present. The operating system intercepts the fault, loads the required page from disk into an available physical frame. The page table is updated to reflect the new location, and the instruction that caused the fault is restarted

Demand paging is a virtual memory management technique where pages of a process are loaded into main memory (RAM) only when they are needed by the CPU during execu□on, rather than loading the en□re program at once. This "lazy swapping" method improves memory e□ciency and allows for larger applica□ons to run on limited hardware by avoiding the loading of unnecessary pages. When a needed page is not in memory, a "page fault" occurs, triggering the opera□ng system to fetch the page from secondary storage, load it into RAM, and update the page table before the process can con□nue.

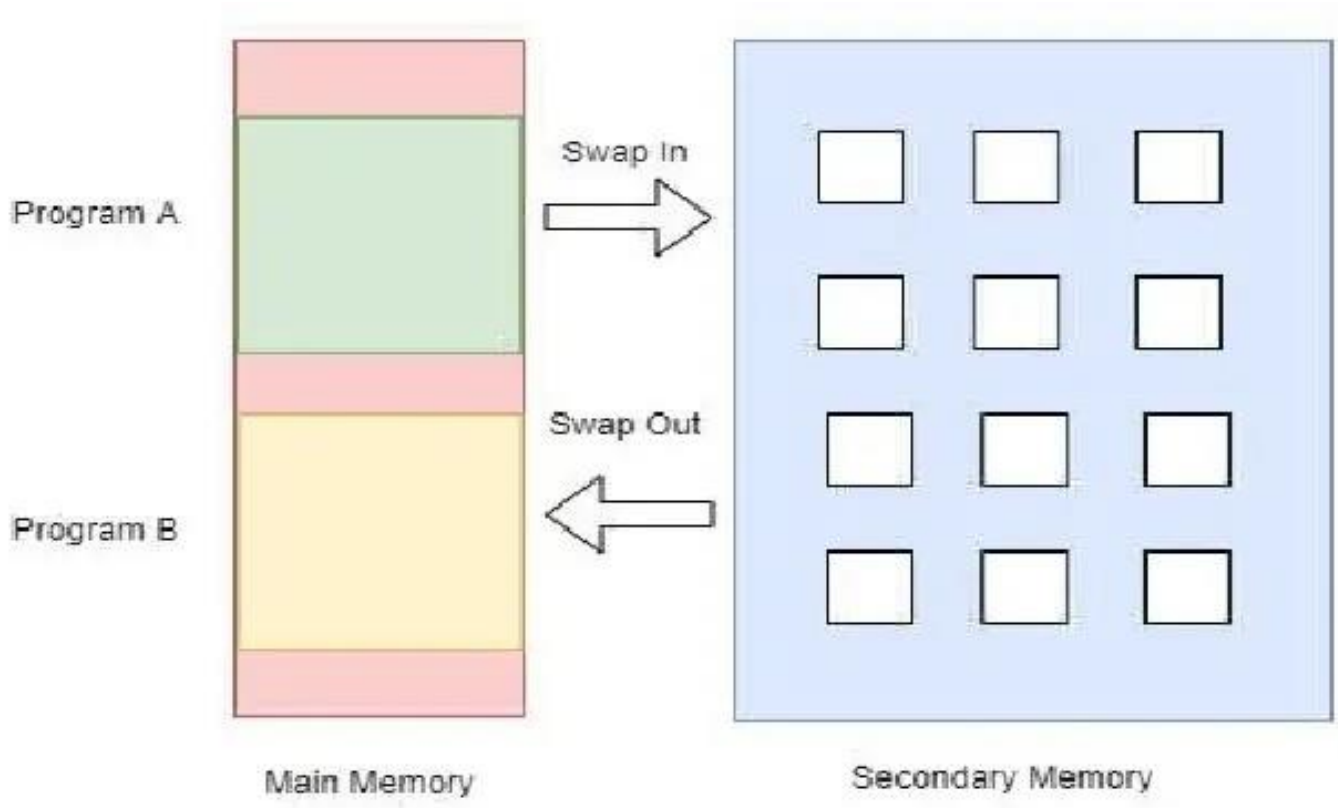


Fig: Demand paging

How Demand Paging Works

- **No Initial Load:** When a process starts, no pages are loaded into memory by default.
- **Page Reference & Fault:** When the CPU requests an address in a page that is not in RAM, a page fault occurs.
- **Fault Handling:** The operating system intervenes and locates the required page on the disk.
- **Page Fetch:** The necessary page is loaded from the secondary storage (disk)

into a free frame in the physical memory.

- **Page Table Update:** The page table is updated to reflect the new location of the page in RAM.
- **Process Resume:** The process is resumed, and execution continues with the page now available in memory.

Key Characteristics

- **Lazy Swapping:** Also known as a "lazy swapper" because pages are only swapped into memory when the CPU explicitly requests them.
- **Virtual Memory:** Demand paging is a cornerstone of virtual memory systems, enabling the illusion of having more memory than physically available.
- **Page Faults:** The primary mechanism for bringing pages into memory, occurring when a page is not found in RAM.
- **Page Tables:** Essential for mapping logical addresses to physical addresses and tracking the validity of pages in memory.

Benefits of Demand Paging

- **Improved Memory Utilization:** Only brings necessary pages into RAM, conserving memory resources.
- **Faster Startup Times:** Processes can start quickly without waiting for the entire program to be loaded.
- **Support for Large Applications:** Allows for the execution of programs that are larger than the physical memory capacity.

Challenges

- **Page Fault Overhead:** Each page fault introduces latency as the system must pause the process to handle the fault and load the page.
- **Potential for Thrashing:** If the system spends too much time paging (moving pages in and out of memory), it can degrade performance significantly

Page replacemen

Page replacement algorithms like FIFO and LRU manage memory by deciding

which page to remove when new pages are needed, with FIFO removing the oldest page and LRU removing the least recently used one. The Op□mal algorithm, theore□cally best but imprac□cal, removes the page not used for the longest future □me. Thrashing is a system performance issue where it spends excessive □me swapping pages due to frequent page faults, leading to a signi□cant slowdown.

Page replacement is a concept in **Operatng Sys□ ems**used in **vir□ual memory managemen□**. It occurs when a **page of memory** needs to be loaded into **RAM**, but **RAM is full**, so the system must **replace** an exis□ng page in RAM with the new one.

Why is Page Replacement Needed?

- Physical RAM is limited.
- Virtual memory allows programs to use more memory than physically available.
- When a program accesses a page that is **not in RAM** , a **page fault** occurs.
- If there's no free space in RAM, the OS uses a **page replacement algorithm** to decide which page to remove.

Page Replacement Process (Steps)

1. **Page fault** occurs (required page is not in RAM).
2. **Check for free frame** in RAM:
   - If available, load the page into it.
   - If **not available** , use a **page replacement algorithm** .
3. Select a **victim page** (page to remove).
4. If victim page is modified, **write it back** to disk.
5. Load the new page into RAM.
6. Update the **page table** .

Common Page Replacemen□ Algori□hms

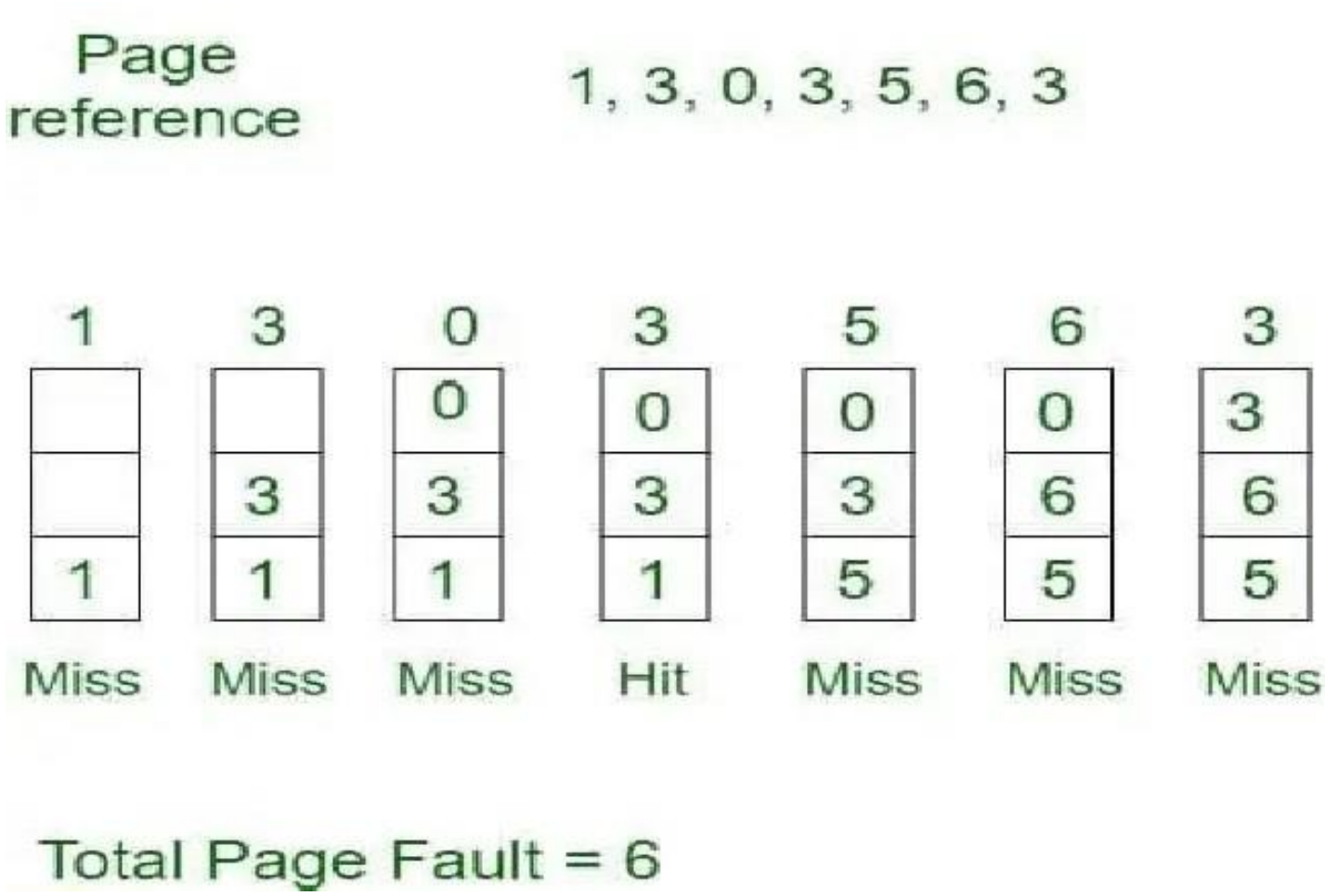
Algorithm	Description
FIFO (First-In, First-Out)	
LRU (Least Recently Used)	

Oldest page is replaced first.	Page not used for the longest time is replaced.
Optimal	Replaces the page that won't be used for the longest time in the future (used for comparison only).
Clock	Circular queue with a reference bit (efficient approximation

Algorithm	Description
	of LRU).

FIFO

Page replacement algorithms like FIFO and LRU select a page to remove from memory when a page fault occurs, aiming to minimize future page faults. FIFO removes the oldest page, while LRU removes the page least recently used. The Op□mal algorithm o□ers perfect performance by removing the page that won't be used for the longest □me but isn't prac□cal. Thrashing is a performance issue where the system spends too much □me swapping pages between main memory and disk due to excessive page faults, hindering overall performance.



How FIFO Works

- **Queue Management:** The operating system maintains a queue of pages currently in physical memory.
- **Page Fault:** When a program requests a page that isn't in memory (a page fault), the algorithm needs to find a page to remove to load the new one.
- **Replacement:** The FIFO algorithm identifies the page at the front of the queue. This is the page that was brought into memory first and has been there the longest.
- **Swap:** The oldest page (the one at the front of the queue) is replaced by the newly requested page from secondary storage

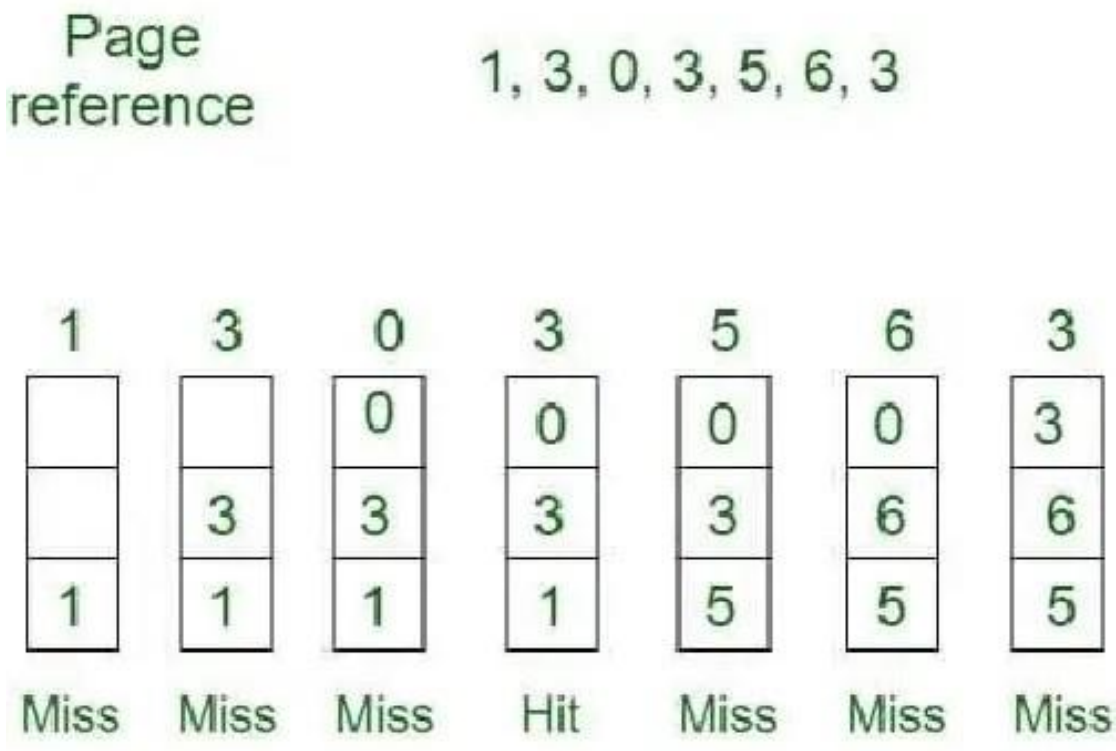
Key Characteristics

- **Simplicity:** FIFO is one of the easiest page replacement algorithms to understand and implement.
- **Queue-Based:** It relies on a basic queue data structure to keep track of page insertion order.
- **No Consideration of Usage:** Unlike other algorithms, FIFO doesn't consider how recently or frequently a page was used; it only cares about the order in which pages entered the system

The First-In, First-Out (FIFO) page replacement algorithm works like a queue, removing the oldest page to make space for a new one when memory is full. For example, with page frames of size 3 and a page request sequence of 7, 0, 1, 2, 0,

3, 0, 4, the FIFO algorithm results in 6 page faults. The algorithm □lls the frames, then replaces the page that has been in memory the longest.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3-page frames. Find the number of page faults using FIFO Page Replacement Algorithm.



Total Page Fault = 6

FIFO - Page Replacement

- Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots ---> **3 Page Faults.**
- When 3 comes, it is already in memory so ---> 0 **Page Faults.**
- Then 5 comes, it is not available in memory, so it replaces the oldest page slot i.e 1. ---> 1 **Page Fault.**
- 6 comes, it is also not available in memory, so it replaces the oldest page slot i.e 3 ---> 1 **Page Fault.**

- Finally, when 3 come it is not available, so it replaces 0 1-page **fault.**

Example:

- **Page Frames:** 3
- **Page Request Sequence:** 7, 0, 1, 2, 0, 3, 0, 4

Step-by-step Execution:

1. **7:** Page 7 is requested. Memory is empty.
   - Page Fault. Load 7.
   - Memory: [7 | - | -] (Faults: 1)
2. **0:** Page 0 is requested. Memory is not full.
   - Page Fault. Load 0.
   - Memory: [7 | 0 | -] (Faults: 2)
3. **1:** Page 1 is requested. Memory is not full.
   - Page Fault. Load 1.
   - Memory: [7 | 0 | 1] (Faults: 3)
4. **2:** Page 2 is requested. Memory is full. 7 was the first in.
   - Page Fault. Replace 7 with 2.
   - Memory: [2 | 0 | 1] (Faults: 4)
5. **0:** Page 0 is requested. It is already in memory.
   - Page Hit. No fault.
   - Memory: [2 | 0 | 1] (Faults: 4)
6. **3:** Page 3 is requested. Memory is full. 2 was the first in.
   - Page Fault. Replace 2 with 3.
   - Memory: [3 | 0 | 1] (Faults: 5)
7. **0:** Page 0 is requested. It is already in memory.
   - Page Hit. No fault.
   - Memory: [3 | 0 | 1] (Faults: 5)
8. **4:** Page 4 is requested. Memory is full. 3 was the first in.
   - Page Fault. Replace 3 with 4.
   - Memory: [4 | 0 | 1] (Faults: 6)

Total Page Faults: 6

LRU:

In Opera□ng Systems, the Least Recently Used (LRU) algorithm is a page

replacement strategy that evicts the page in memory that has gone the longest without being accessed, based on the principle of temporal locality. When a page fault occurs (a requested page isn't in memory) and the memory is full, the OS iden□□es the page that hasn't been used for the longest □me and replaces it with the new page. This is done to op□mize performance by keeping recently used pages in memory, as they are likely to be needed again soon.

How LRU Works

- **Locali□ y of Reference:** The algorithm relies on temporal locality , the tendency for a program to re-reference the same memory loca□ons within a short period.
- **Page Access Tracking:** Each □me a page is accessed, its access □me is updated, or it's moved to the "most recently used" posi□on in a data structure.
- **Page Faul□ Handling:** If a requested page is already in memory (a page hit), no replacement occurs, and the page's "used" status is updated. If a page fault occurs and memory is full, the algorithm looks for the page with the oldest access □me (the least recently used page). This page is then replaced with the new, requested page.

Implementation

LRU can be implemented using various data structures to manage page usage order:

- **Linked List:** A linked list can store pages in order of their last use, with the most recently used page at the head and the least recently used at the tail.
- **Hash Table and Linked List:** A combination of a hash table for quick page lookups and a linked list for maintaining the order of usage is also a common approach.
- **Counters/Timestamps:** Each page has a timestamp or counter indicating when it was last used. When a replacement is needed, the page with the lowest timestamp is chosen.

Pros and Cons

- **Pros:**

  LRU generally performs well by minimizing page faults and is considered a good strategy in practice, especially for applications exhibiting strong temporal locality.

- **Cons:**

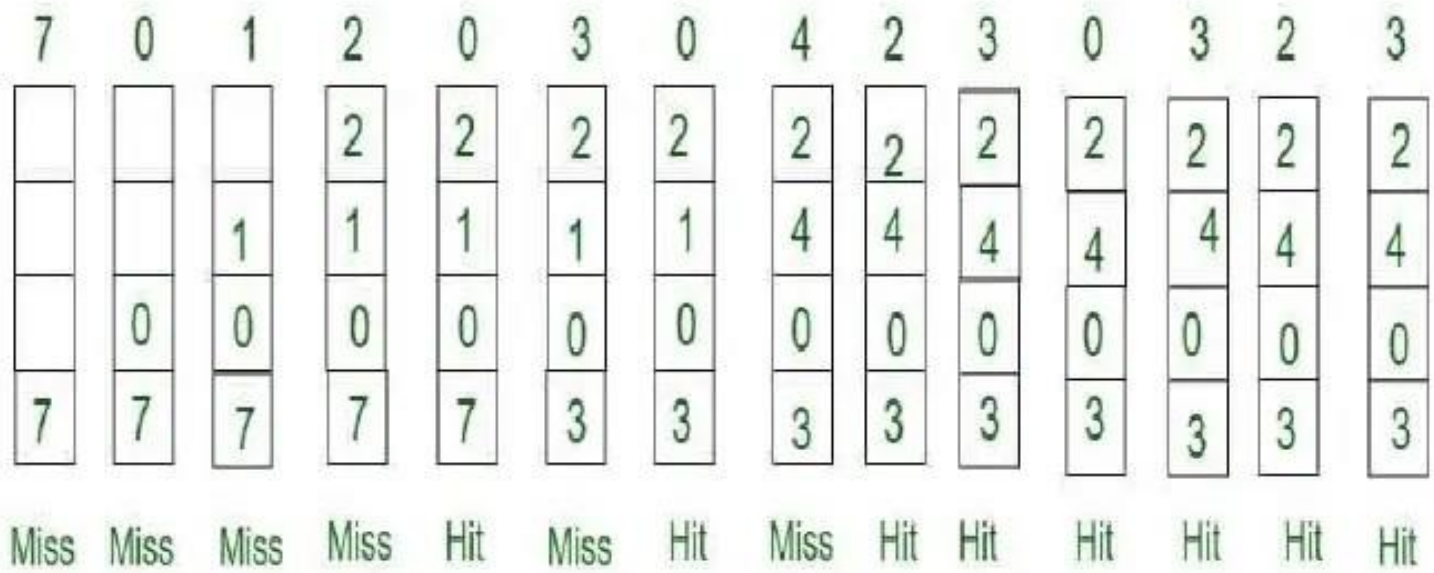
  The primary drawback is the complexity of implementation and the overhead required by the operating system to maintain accurate access history for every page, often requiring hardware assistance. this algorithm, page will be replaced which is least recently used.

Example

Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4-page frames. Find number of page faults using LRU Page Replacement Algorithm.

- Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots ---> 4 Page faults.
- 0 is already there so ---> 0 Page fault. when 3 came it will take the place of 7 because it is least recently used ---> 1 Page fault.
- 0 is already in memory so ---> 0 Page fault.
- 4 will takes place of 1 ---> 1 Page Fault.
- Now for the further page reference string ---> 0 Page fault because they are already available in the memory.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Optimal Page Replacement is a theoretical algorithm where a process that will not be used for the longest period in the future is replaced during a page fault, aiming to minimize page faults and serve as a benchmark for other algorithms. Thrashing is a system performance issue characterized by excessive page swapping between memory and disk due to insufficient physical memory, leading to low system productivity, high disk activity, and a high page fault rate. While Optimal Page Replacement seeks to solve memory management problems, thrashing is a state the system enters when it cannot effectively manage its memory.
Optimal Page Replacement

Concept : The core idea is to swap out the page that is required the farthest in the future of the reference string, thereby minimizing future page faults.

- **How it Works** :

When a page fault occurs, the operating system looks ahead in the page reference string and replaces the page that won't be needed for the longest time.

- **Significance** : This algorithm is considered optimal because it provides the lowest possible number of page faults for a given reference string and number of memory frames.
- **Limiations** : It is a theoretical concept because it requires future knowledge of the entire page reference string, which is impossible for a real-time operating system to possess.

Thrashing in Operating Systems

Definition : Thrashing is a pathological state in an operating system where the system spends too much time swapping pages between primary memory (RAM) and secondary storage (disk).

Causes : It happens when the total memory required by processes exceeds the available physical memory (RAM), leading to constant page faulting and swapping.

Symptoms :

- **High Page Fault Rate** : The system is constantly experiencing page faults because the needed pages are not in memory.
- **Increased Disk Activity** : Heavy disk I/O occurs as pages are swapped in and out continuously.
- **Low Productivity** : The CPU spends more time on swapping and less time on actual user processes and computations, leading to significantly reduced system performance.

Relationship Between Optimal Replacement and Thrashing

Optimal Page Replacement is a Solution (Theoretical) : The Optimal Page Replacement algorithm is designed to prevent thrashing by making the "best" possible choice of which page to replace, thus minimizing page faults.

Thrashing is the Problem : Thrashing is the outcome of inefficient page management, particularly when the system has too few frames for the workload.
No Direct Link in Practice : While Optimal Page Replacement aims to achieve efficiency, thrashing occurs when the system's available memory is simply insufficient for the active processes.

Optimal Page Replacement Algorithm Example
Let's use the page reference string (7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2) with 3 page frames:
Total Page Faults: 3 + 1 + 0 + 1 + 0 + 1 + 1 + 0 = 7

Thrashing in OS Example
A system experiences thrashing when it doesn't have enough physical RAM to hold all the active pages of the running processes.

- **Scenario:** Imagine a system with a limited amount of RAM that is running multiple memory-intensive applications.
- **Process:** When a new process needs a page, the operating system might need to swap out a page belonging to another process to make room.
- **Result:** This leads to a constant cycle of page swapping between RAM and the much slower hard disk. The CPU spends most of its time managing these swaps rather than executing application code.
- **Effect:** This creates a severe performance degradation, slow application response times, and can even lead to system crashes, indicating a state of thrashing