UNIT-IV

MEMORY MANAGEMENT

MAIN MEMORY

The main job of a computer is to run programs (like games, applications, or tools).

- When a program is running, the program itself and the data it uses must be stored in the computer's **main memory** (also called **RAM**).
- Think of memory as a long row of storage boxes. Each box stores 1 byte of data and has its own unique number called an **address**.
- The CPU (Central Processing Unit – the brain of the computer) follows a **Program Counter (PC)** which tells it which instruction to run next.
- Based on this PC value, the CPU fetches instructions (gets the next command) from the memory and runs them.

BASIC HARDWARE

The CPU can **directly access** information only from:

1. **Main Memory (RAM)**
2. **Processor Registers** (small storage spaces inside the CPU)

These two are known as **Direct Access Storage Devices** because the CPU can use them **directly** without needing help from other components.

How Instructions and Data are Used

- Whenever the CPU is running an instruction, it must:
  - Have the **instruction** itself and
  - Any **data** the instruction uses
  - Both must be already in **registers or main memory**
- If that data is **not in main memory**, the computer must **first bring the data into main memory** from some other place (like hard disk or SSD) **before** the CPU can use it.

Registers vs Main Memory Access Time

- **Registers** (inside the CPU) are very fast. The CPU can **get data from registers almost instantly** — in **one CPU clock cycle**.
- But **main memory is slower**. Accessing it might take **multiple clock cycles**. The data is transferred via the **memory bus** (a special connection between CPU and RAM).

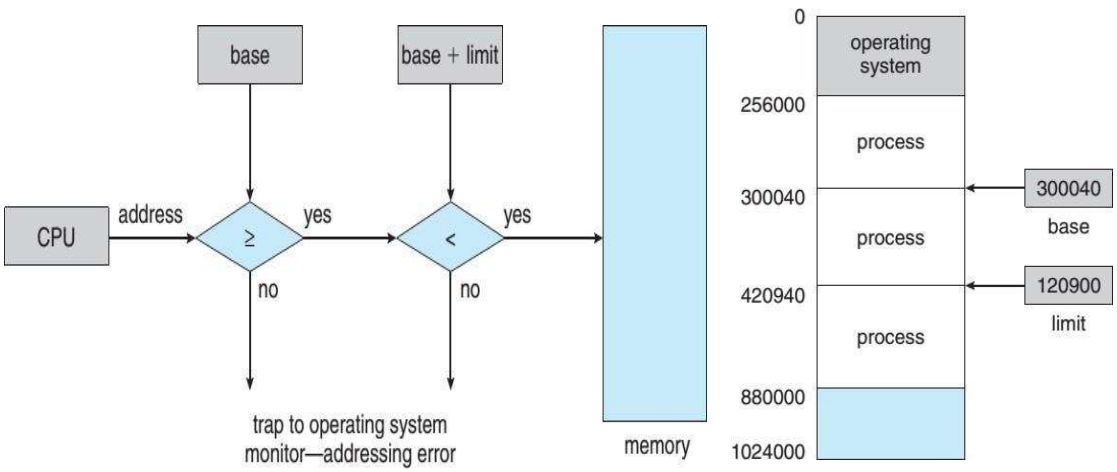
Memory Stall:

- If the CPU is waiting too long to get the data from main memory, it **stops temporarily**. This wait is called a **memory stall**.
- Memory stalls are bad because they **slow down the performance** of the computer.

Cache Memory:

- To reduce waiting time, a **special fast memory** called **Cache Memory** is used.
- **Cache is placed between the CPU and Main Memory**.
- Cache keeps a **copy of frequently used data and instructions**, so the CPU can get them quickly **without waiting for main memory**.

BASE REGISTER & LIMIT REGISTER



In a computer, **many programs (processes)** can run at the same time. To do this safely, **each process must be kept separate** in memory so that **one program doesn't disturb another**. To ensure this protection, two special hardware registers are used:

1. Base Register

- It stores the **starting address** (lowest memory location) where the process is allowed to access memory.
- This is the **smallest legal physical address** for that particular process.

2. Limit Register

- It defines the **size** of the memory the process can use.
- This tells how **far from the base address** the process is allowed to go (i.e., its memory range).

Example:

- Suppose the **Base Register** has: 300040
- And the **Limit Register** has: 120900

Then the process can **only access memory addresses from** 300040 to 420939 (because 300040 + 120900 - 1 = 420939).

It **cannot access memory** outside this range.

Memory Protection Using Registers

- Only the **Operating System (OS)** can set or change the values of the base and limit registers.
- This is done using **special privileged instructions** that can be executed **only in kernel mode** (i.e., when the OS is running).

Memory Protection:

- Whenever a user program tries to access a memory location:
  - The **CPU checks** whether the address is within the **Base and Limit** range.
- If the program tries to access an **invalid address** (outside its allowed range), the CPU **generates a trap** (a type of error).
- The OS treats this as a **Fatal Error**, and the program is stopped.

This system **protects the OS and other programs** from being accidentally or intentionally changed by a user program.

OS Access in Kernel Mode

- When the OS is running in **kernel mode**, it has **full access** to:
  - Its own memory
  - All user program memory

This is important because the OS must do things like:

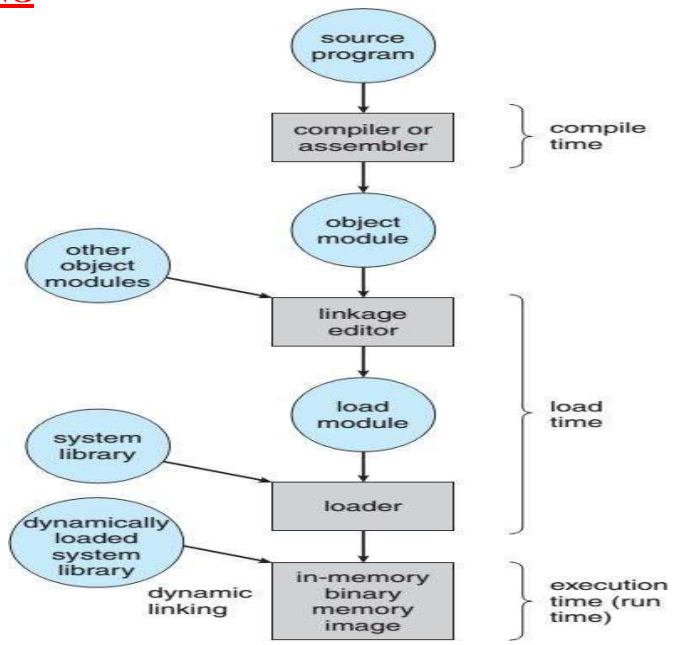
- Load user programs into memory
- Remove programs from memory if they crash
- Access and update values used in **system calls**
- Perform **input/output** using user memory
- Handle **context switches** between programs

Example: Context Switch

- In a **multiprogramming OS**, multiple programs run one after the other.
- Before switching from one program to another, the OS must:
    1. **Save** the current program's state (registers) into main memory.
    2. **Load** the next program's state from memory into the registers.

This whole process is called a **context switch**, and the base and limit registers help make sure that **each process stays in its own safe memory area**.

ADDRESS BINDING



A program is initially stored on the disk as a binary executable file. Before execution, the program must be brought into main memory and assigned proper memory addresses. This process is known as **address binding**, where different types of addresses are mapped to each other during different stages of a program's life cycle.

While running, the operating system may move the process between disk and memory. All programs that are waiting to be brought into memory for execution are kept in the **input queue**.

During program development and execution, addresses appear in different forms:

- **Symbolic addresses** are used in source code (e.g., variable names like x, count).
- The **compiler** converts symbolic addresses into **relocatable addresses** (e.g., "14 bytes from start of program").
- The **loader** or **linkage editor** finally converts relocatable addresses into **absolute addresses** (e.g., 74014, which means 74000 base + 14 offset).

Each step in the program setup process maps one address space to another. This is the essence of address binding.

Address binding can occur at different times:

Compile-time Binding

When the exact memory location where the program will be loaded is known during compilation, the compiler directly generates **absolute code**. If the program's starting location changes later, recompilation is needed.

For example, MS-DOS .COM format programs are compiled using this method. These programs are always loaded at the same fixed location in memory.

Load-time Binding

If the memory location is not known at compile time, the compiler generates **relocatable code**. During loading, the loader assigns the actual physical address and updates all addresses in the program. If the program needs to be loaded at a different location later, it just needs to be reloaded, not recompiled.

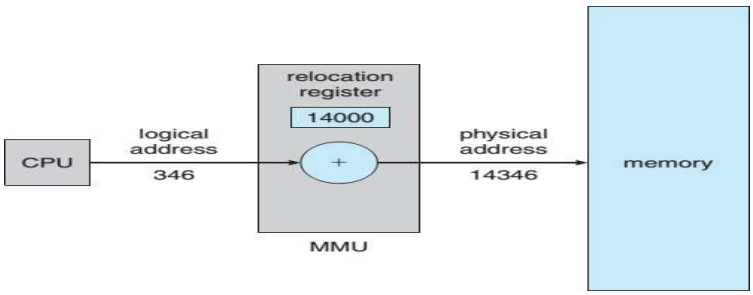
Execution-time Binding

In advanced systems where a program may move in memory during its execution, address binding is done **at run time**. This requires hardware support through the **Memory Management Unit (MMU)**. Most modern operating systems like Windows and Linux use this method, as it provides high flexibility and efficient memory management.

This binding method is especially important in systems using techniques like virtual memory and demand paging, where the actual physical location of a program or data can change dynamically while the program is running.

Binding Time	Type of Code Generated	Flexibility	Example
Compile-Time	Absolute Code	Low	MS-DOS .COM programs
Load-Time	Relocatable Code	Medium	Reload if location changes
Execution-Time	Relocatable + MMU	High	Modern general-purpose OS

LOGICAL VERSUS PHYSICAL ADDRESS SPACE



Logical Address:

- A **logical address** is the address generated by the **CPU** during the execution of a program.
- It is also called a **virtual address**, especially when it differs from the actual physical memory address.
- The user program always works with **logical addresses**. It never knows where exactly in physical memory its instructions and data are stored.

Physical Address:

- A **physical address** is the actual location in the **main memory (RAM)**.
- This is the address that is finally used by the **Memory Address Register (MAR)** to fetch or store data in memory.

Address Spaces

- **Logical Address Space (LAS):**
  - This is the **set of all logical addresses** generated by a program.
  - Example: If a program generates addresses from 0 to 999, then LAS = 0 to 999.
- **Physical Address Space (PAS):**
  - This is the **set of all physical addresses** where the program is actually loaded in memory.
  - Example: If the program is loaded from physical location 14000, then PAS = 14000 to 14999.

Binding Methods and Address Types

i. Compile-Time and Load-Time Binding:

- The **logical and physical addresses are the same**.
- The program is always loaded at a fixed address in memory.
- The address generated by the CPU is directly used to access memory.

ii. Execution-Time Binding:

- The **logical and physical addresses are different**.
- The logical address is also called a **virtual address** here.
- During execution, logical addresses must be converted to physical addresses.
- This conversion is done **dynamically at run-time** using a hardware component called **MMU (Memory Management Unit)**.

Memory Management Unit (MMU)

- The **MMU** is a hardware device that maps logical addresses (generated by the program) to physical addresses (actual RAM locations). This allows programs to be moved in memory during execution without changing the program's instructions.

Role of Base Register (Relocation Register)

- In execution-time binding, the **Base Register** is used to **relocate** addresses.
- It is also called the **Relocation Register**.

Every address generated by a user program is **added to the value in the base register** before it is sent to the memory.

Example: Suppose:

  - The **Base Register** (Relocation Register) holds the value **14000**
  - The program tries to access **logical address 0**

Then:

- Actual **physical address** = 14000 + 0 = **14000**
- If it tries to access **logical address 346**, then:
  - Physical address = 14000 + 346 = **14346**

Behaviour of the User Program

- The user program works **only with logical addresses**.
- It can:
  - Create a pointer to address 346
  - Store it in a variable
  - Compare it with another address

- Use it in expressions

But it always treats that number as **346**.

- Only when that logical address is actually used to access memory, the **MMU translates** it to the corresponding physical address using the base register.

Final Address Resolution

- The actual physical address is **not known at compile time or load time** in execution-time binding.
- The final physical address is calculated **only at the moment when the memory access happens**, using:

Physical Address = Logical Address + Base Register (Relocation Register)

Another Example: Let's assume:

- Logical Address Range = 0 to max (say, 999)
- Base Register Value = R = 14000

Then:

Logical Address	Physical Address
0	14000
1	14001
346	14346
999	14999

So, the program thinks it's using addresses 0 to 999, but the actual memory used is from 14000 to 14999.

LOADING

When a program is prepared for execution, it needs to be **loaded into memory**. There are two types of loading:

1. Static Loading

- In **static loading**, all the program modules and routines are **loaded into memory at once**, **before** the program starts executing.
- Every part of the program is present in memory during its entire execution.
- No additional code is brought into memory during runtime.

2. Dynamic Loading

- In **dynamic loading**, a **routine (part of the program)** is **not loaded** into memory **until it is actually called** during execution.
- Initially, only the **main program** is loaded into memory.
- Other routines are stored on disk in a **relocatable load format**.

How it works:

1. The main program runs in memory.
2. When it needs to call another routine:
    - It first checks if that routine is already in memory.
    - If **not**, the **relocatable linking loader** is invoked to:
        - Load the required routine into memory.
        - Update the **program's address tables** to reflect the new memory location.
3. Then, control is passed to the newly loaded routine.

Advantage:

- Very useful when large code is needed for **rarely used tasks**, like **error handling**.
- Even if the total program is large, only the **used portions** occupy memory, saving space.

Note:

- It is the **responsibility of the programmer** to design the program in a way that supports dynamic loading.
- Operating systems may offer **library routines** to help implement dynamic loading.

LINKING

Linking is the process of **connecting multiple modules and libraries** to create a single executable. It can be done either **before execution** or **during execution**.

1. Static Linking

- In static linking, all **required libraries and modules** (including system libraries) are **linked** and combined into a **single binary executable** at **compile time or load time**.
- The entire code becomes part of the final program.

Drawback:

- Each program will have its own **separate copy** of system libraries.
- This leads to **waste of disk space and main memory**, especially if multiple programs use the same library.

2. Dynamic Linking

- In dynamic linking, the required **libraries are not combined** with the program during compile time.
- Instead, the linking is **postponed until execution time**.

Commonly used with:

- **System libraries**, such as **language subroutine libraries**.

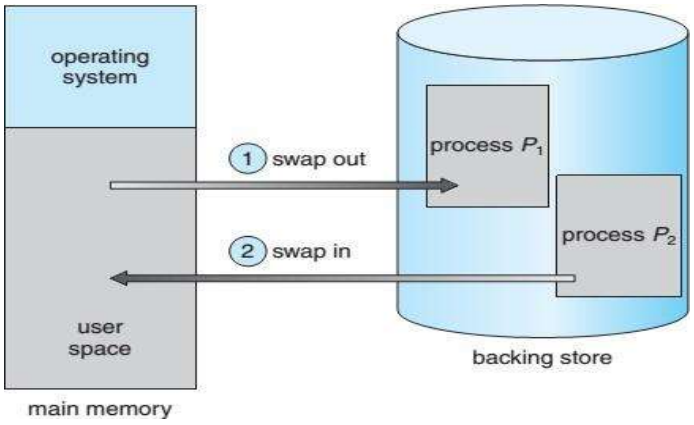
How Dynamic Linking Works:

1. A **stub** is placed in the executable for each dynamically linked routine.
    - A **stub** is a small piece of code used to:
        - Locate the required routine in memory.
        - Or load the routine if it's not already present.
2. When the stub executes:
    - It checks if the required routine is already in memory.
    - If not, the **routine is loaded dynamically**.
    - The stub then **replaces itself with the memory address** of the routine.
    - The routine is then executed.
3. On subsequent calls, the routine is directly executed with **no extra cost** for linking.

Advantage:

- Only **one copy** of the library code is stored in memory.
- All processes using that library **share the same copy**, saving both memory and disk space.

SWAPPING



- A **process must be in main memory (RAM)** to be executed by the CPU.
- If there isn't enough space in memory, the OS **temporarily moves (swaps out)** a process to the hard disk (called **backing store**).
- Later, when needed, the process is **brought back (swapped in)** to continue execution.
- This technique allows **more processes to be handled**, even if RAM is limited — this increases **multiprogramming**.

Standard Swapping (Old Method)

- The **entire process** is swapped between RAM and the disk.
- The **backing store** is usually a **hard disk** which:
  - Is large enough to hold all processes.
  - Supports **direct access** (to quickly locate and load any process).

How it works:

1. **Ready Queue** contains all ready-to-run processes (some in RAM, some on disk).
2. CPU picks the next process to run.
3. If that process is **not in RAM**:
   - A running or idle process is **swapped out**.
   - The required process is **swapped in** from disk.
4. CPU registers are reloaded, and the new process starts.

Time Taken:

- **Swapping time is high** because entire process memory must be transferred.
- Only **idle processes** (e.g. those waiting for I/O) can be swapped.

Disadvantages:

- **Very slow** and **wastes CPU time**.
- So, **modern OS do not use this** full-process swapping method anymore.

Swapping in Modern Operating Systems

Modern OS like **Windows, Linux, UNIX** use an **improved version** of swapping:

- **Swapping is done only when necessary**:
  - Enabled when memory is **low**.
  - Disabled when memory is **available**.
- **Only parts of a process** (not the whole process) are swapped.
- Works together with **Virtual Memory** to manage RAM efficiently.

Swapping in Mobile Operating Systems (iOS, Android)

Mobile OS do **not use traditional swapping**. The **reasons is:**

- They use **flash memory**, not hard disks.
- Flash memory has:
  - **Limited write cycles** (gets damaged with too many writes).
  - **Slower speed** between flash and RAM.
- Also, mobile devices have **less memory space**.

What do Mobile OS do instead?

iOS: When memory is low:

  - OS asks apps to **release unused memory**.
  - **Code sections** (read-only) may be removed and **reloaded later**.
  - **Changed data** (like stack, heap) is **never removed**.
  - If an app doesn't free enough memory, it may be **closed (killed)**.

Android: If memory is full:

  - Android may **kill background apps**.
  - Before closing, it **saves the app state** to flash memory.
  - So, next time, the app can be **resumed quickly**.

CONTIGUOUS MEMORY ALLOCATION

In an Operating System, **memory management** is one of the most important tasks. Every time a program (or process) is run, it needs some amount of **main memory (RAM)** to execute. The way this memory is allocated to processes plays a key role in:

- **System performance**
- **CPU utilization**
- **Multiprogramming (running multiple processes simultaneously)**

One of the oldest and simplest memory allocation techniques is **Contiguous Memory Allocation**, where each process is given a **single continuous block of memory**.

Contiguous Memory Allocation:

In this technique:

- Each process is stored in **a single block** of memory.
- That block must be **contiguous (unbroken)** in physical memory.
- Memory is allocated either by dividing into fixed-size partitions or by adjusting sizes dynamically.

There are **two main schemes** of Contiguous Memory Allocation:

1. Fixed Partition Scheme (MFT – Multiprogramming with Fixed Number of Tasks)

- The entire memory is **divided into fixed-size blocks (partitions)** at the beginning.
- Each partition can hold **exactly one process**.
- If a process is smaller than the partition, **unused memory is wasted** (called internal fragmentation).
- If a process is larger than the partition size, it **cannot be allocated**.

Working:

- When a partition becomes free, a process from the **input queue** is loaded into that partition.
- Once the process completes, the partition becomes **available again**.

Example:

If there are 4 fixed partitions of 256 KB each, and you try to load a 300 KB process — it will be rejected, even if memory is available, because the partition is not big enough.

This method was used in early systems like **IBM OS/360 (MFT)** but is **no longer used** due to its rigid nature.

2. Variable Partition Scheme (MVT – Multiprogramming with Variable Number of Tasks)

- Here, memory is **not pre-divided**.
- Instead, memory is **allocated dynamically** based on the size of each process.
- As processes come and go, they leave **free memory blocks (called holes)**.

- When a process arrives, the system looks for a **free hole** large enough to hold the process.
- If the hole is larger than required, it is **split** — one part is given to the process, the other remains free.
- When a process completes, its memory is **freed** and added back to the list of holes.
- **Adjacent holes are merged (coalesced)** to form larger holes when possible.

Important Terms:

- **Hole** – A block of free memory.
- **External Fragmentation** – When free memory is scattered across many small holes, making it hard to allocate large processes.

Dynamic Storage Allocation Problem

In variable partitioning, memory gets divided into holes of different sizes. When a new process arrives, the OS must decide **which hole to use**. This leads to the **Dynamic Storage Allocation Problem**. There are **4 popular solutions** to this:

1. First Fit

- Scans memory from the **beginning**.
- Allocates the **first hole** that is big enough.
- **Fastest** method.
- May leave **small unusable holes** behind.

2. Best Fit

- Searches **entire memory**.
- Allocates the **smallest possible hole** that fits the process.
- **Minimizes leftover space**.
- Can cause **many small holes**, leading to fragmentation.

3. Worst Fit

- Searches for the **largest available hole**.
- Allocates the process in that hole.
- Leaves behind a **larger leftover hole** that might be more useful later.
- Can still lead to waste.

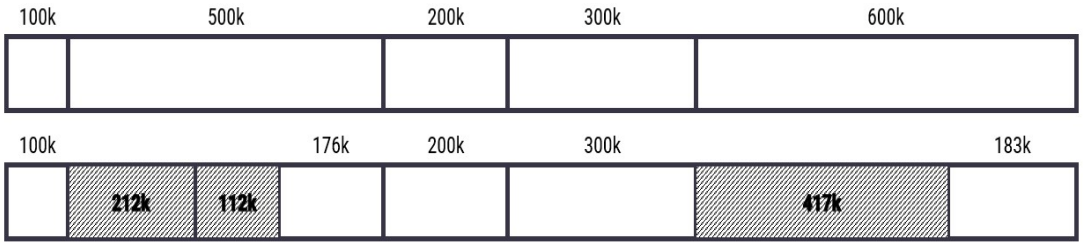
4. Next Fit

- Similar to First Fit, but search continues from where the last allocation left off (not from the beginning).
- Slightly faster in some cases compared to First Fit.
- Avoids clustering at the beginning of memory.

Example: Apply First Fit, Best Fit, Worst Fit, Next Fit

- Processes of **212K, 417K, 112K** and **426K** arrives in order.

First Fit:

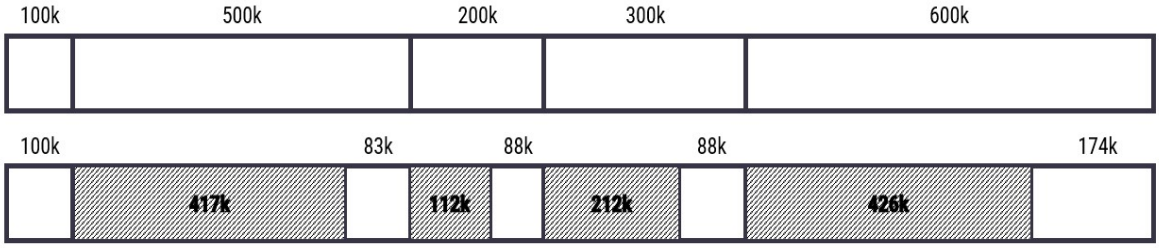


- **Here process of size 426k will not get any partition for allocation.**

Next Fit:

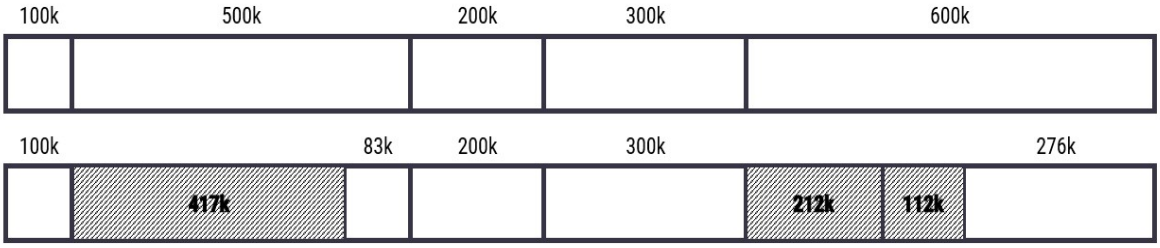
- **Here process of size 426k will not get any partition for allocation.**

Best Fit:



- **Memory is allocated to all the processes**

Worst Fit:



- **Here process of size 426k will not get any partition for allocation.**

FRAGMENTATION

When memory is allocated to processes, it is not always used efficiently. In some cases, a portion of the memory goes unused. This wastage of memory is known as **fragmentation**. It affects system performance and memory utilization.

Fragmentation is mainly of two types:

1. Internal Fragmentation

Internal fragmentation occurs when a process is allocated more memory than it actually needs. The extra space inside the allocated memory block remains unused, leading to wastage.

Example:

Consider a memory block of **18,464 bytes**.

If a process requests **18,462 bytes**, the entire block is allocated.

This leaves **2 bytes** unused within the block.

This small unused portion is considered **internal fragmentation** because it is inside the allocated block and cannot be used by other processes.

Key Points:

- Occurs **inside** the allocated memory.
- Usually happens when memory is divided into **fixed-size blocks**.
- The unused space is **part of the allocated block**, so it cannot be reassigned.
- Common in systems using **fixed partitioning**.

2. External Fragmentation

External fragmentation occurs when the total free memory is enough to satisfy a process's request, but the memory is split into small, non-contiguous holes. Since the required memory is not available in a single continuous block, the process cannot be loaded.

Example:

Suppose a process needs **300 KB** of memory.

There is free memory in three blocks — 100 KB, 80 KB, and 120 KB — all scattered in different locations.
Although the total free memory is 300 KB, it is **not continuous**, so the process cannot be accommodated.
This situation is known as **external fragmentation**.

Key Points:

- Occurs **between** allocated blocks.
- Free memory is split into many **small non-contiguous chunks**.
- Prevents large processes from being loaded, even if total memory is sufficient.

Solution to External Fragmentation

To reduce or eliminate external fragmentation, the following techniques are used:

1. Compaction

- Compaction involves moving all allocated memory blocks together and combining all the free space into one large block.
- This helps in creating a **single large hole** for future allocations.
- Compaction is only possible if the system supports **dynamic relocation** during execution.
- Note: It is a time-consuming process because it involves **shifting memory contents**.

2. Paging and Segmentation

- These are non-contiguous memory allocation techniques.
- **Paging** divides memory and processes into fixed-size pages and frames, allowing them to be placed anywhere in physical memory.
- **Segmentation** divides processes into variable-sized logical segments (like code, data, stack), which can also be placed in any available space.

SEGMENTATION

Segmentation is a memory management technique where the **logical address space** of a process is divided into **multiple segments**, and each segment is stored separately in physical memory.

Need for Segmentation

In real-time applications and programming, we naturally organize programs into sections like **code, data, heap, stack**, and **libraries**. Segmentation supports this structure, making **memory allocation logical, efficient**, and **modular**.

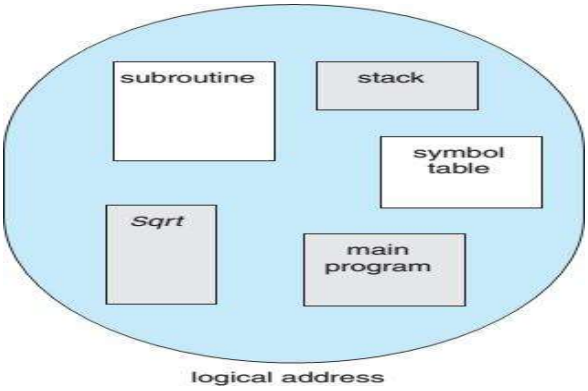
Structure and Role of Segments

A **segment** is a meaningful unit of a program such as:

- Code (program instructions)
- Data (global/static variables)
- Heap (dynamically allocated memory)
- Stack (function calls and local variables)
- Libraries (standard or user-defined functions)

Each segment has:

- A **Segment Number** (unique identifier)
- A **Length** (number of bytes it occupies)



Logical Address Representation

In segmentation, the **logical address** is expressed using **two components**:

Logical Address = ⟨ Segment Number, Offset ⟩

- **Segment Number (s)** → Identifies which segment
- **Offset (d)** → Position within the segment

Example Segment Allocation in C Program

Segment Name	Contents	Example Segment Number
Code	Program instructions	0
Global Data	Global/static variables	1
Heap	Dynamic memory (malloc, new, etc.)	2
Stack	Function calls, local variables	3
Library	Standard C library functions	4

Mechanism of Address Translation

The **Memory Management Unit (MMU)** and a **Segment Table** are used to convert logical addresses into physical addresses.

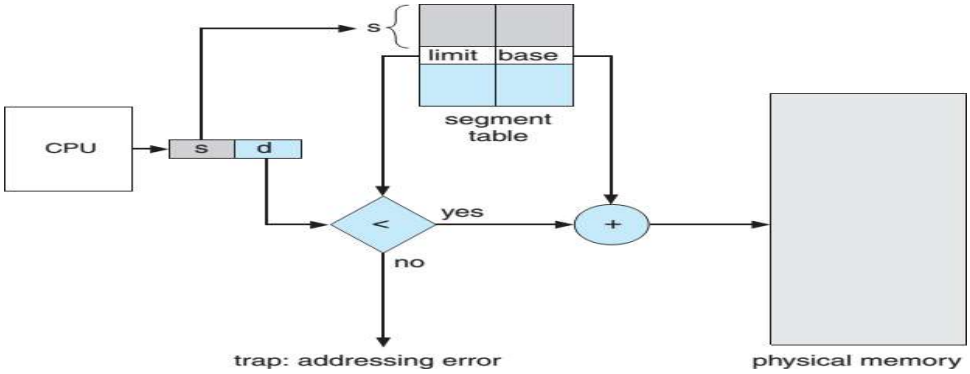
Segment Table Entry Contains:

- **Base** → Starting physical address of the segment
- **Limit** → Maximum size of the segment in bytes

Steps in Address Translation:

1. Use **segment number (s)** to look up the segment table.
2. Retrieve the **base** and **limit** values for that segment.
3. Check the **offset (d)**:
    - If d < limit, it's a **valid** address.
    - If d ≥ limit, it's **invalid**, and the system triggers a **trap (error)**.
4. For valid addresses:

Physical Address = Base + Offset

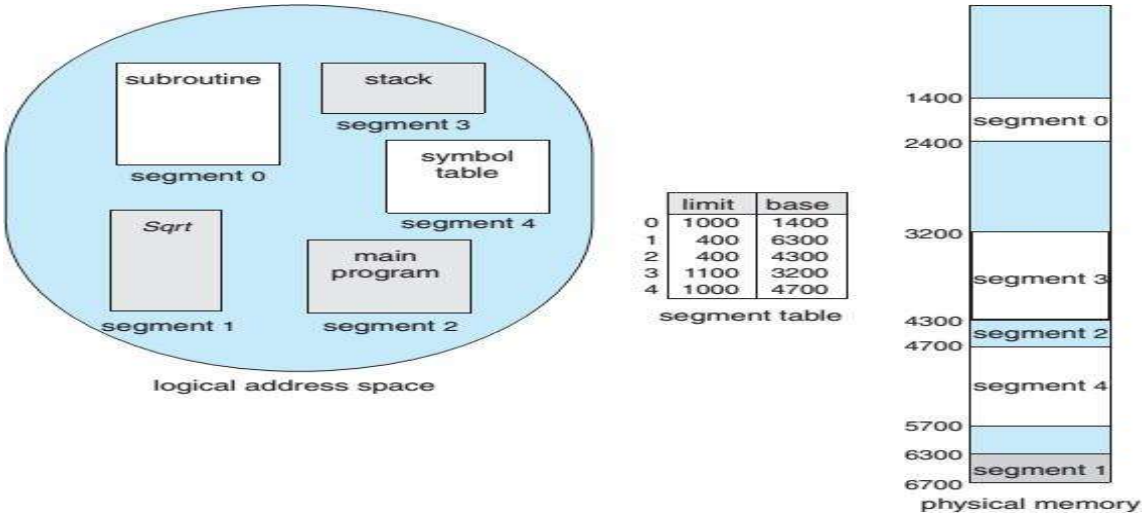


Segment Table Example:

Segment No.	Base	Limit
0	2190	1000
1	2300	400
2	4300	400
3	3200	1100
4	4700	300

Sample Address Translations:

- Segment 2, Offset 53
  → Limit = 400 → Valid
  → Physical Address = 4300 + 53 = **4353**
- Segment 3, Offset 852
  → Limit = 1100 → Valid
  → Physical Address = 3200 + 852 = **4052**
- Segment 0, Offset 1222
  → Limit = 1000 → **Invalid**
  → **Trap (Error):** Access beyond segment boundary



Advantages of Segmentation

- **Logical Division of Programs:** Segmentation follows how a programmer naturally thinks — dividing the program into code, data, stack, etc.
- **Modularity and Ease of Management:** Each segment can be managed, protected, and grown independently. Helps in isolating bugs and organizing code better.
- **Efficient Use of Memory:** Since only the required segments are loaded, it saves memory and speeds up execution.
- **Support for Protection:** Each segment can have different access rights (read, write, execute), offering better security and protection.
- **Ease of Sharing:** Code or library segments can be shared among processes without sharing their data or stack.
- **Dynamic Segment Growth:** Segments like stack and heap can grow or shrink independently, depending on the need.

Disadvantages of Segmentation

- **External Fragmentation:** Even though logical division is clear, free memory blocks may get scattered in physical memory — leading to fragmentation.
- **Complex Memory Management:** Keeping track of base and limit for each segment and performing protection checks increases hardware and OS complexity.
- **Overhead of Segment Table:** Each process needs its own segment table, which consumes memory and must be maintained properly.
- **Difficult Compaction:** Merging free memory blocks (compaction) is harder due to scattered segments — and may affect performance.

PAGING

Paging is a **memory management technique** used in Operating Systems that allows processes to be stored in **non-contiguous memory locations**. It avoids **external fragmentation** by using a technique where **both the process and memory are divided into fixed-size blocks**.

- **Pages**: The process is divided into fixed-size blocks called **pages**.
- **Frames**: The main memory is also divided into blocks of the same size, called **frames**.
- One page of a process is stored in one frame of memory.
- Pages can be stored **anywhere** in memory, but the system prefers storing them in **contiguous frames** when available.

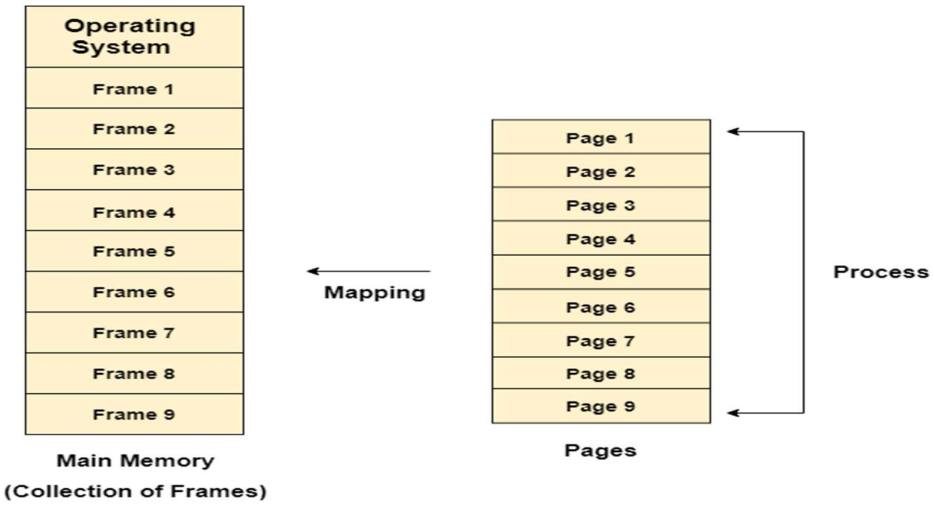
This method ensures efficient memory usage and simplifies memory allocation.

Working Mechanism of Paging

When a process is to be executed:

- It is divided into pages.
- These pages are loaded into available memory frames.
- Pages of a process do not need to be stored in contiguous frames.

Pages are brought into the main memory **only when required**. The rest of the pages stay in **secondary storage** (like hard disk). The size of pages and frames must be **equal**.

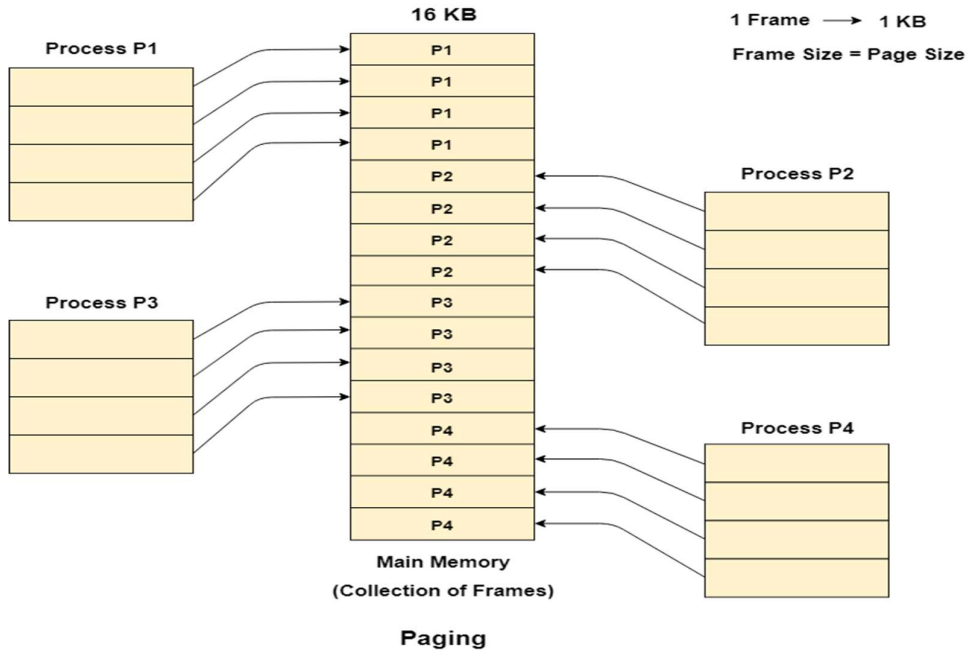


Example

Let us consider the main memory size is **16 KB**, and each frame size is **1 KB**.

- Memory is divided into **16 frames** of 1 KB each.
- There are **4 processes**: P1, P2, P3, and P4, each of size 4 KB.
- Each process is divided into **4 pages** of 1 KB each.
- Initially, as all frames are empty, the pages will be stored in **contiguous frames**.

Now, assume **P2 and P4** move to the waiting state. Their memory frames become free (8 frames). A new process **P5** of 8 KB (8 pages) is ready to execute. These **non-contiguous 8 free frames** can now store the pages of P5, thanks to the flexibility provided by paging.



Paging

Page Table

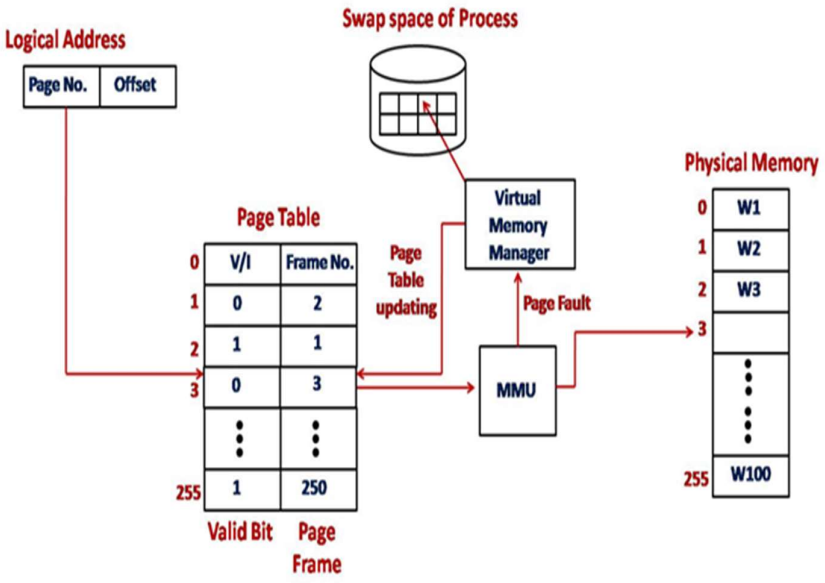
The **Page Table** is a key data structure used in paging. It **maps each page number to a frame number**.

- Each process has its own page table.
- When a logical address is generated by the CPU, the **page number is used to index into the page table**.
- The page table returns the **frame number** where that page is stored.

Each page table entry may include:

- **Frame number**
- **Present/absent bit** (Is the page in memory?)
- **Protection bit** (Access rights)
- **Modified bit** (Has the page been changed?)
- **Reference bit** (For replacement algorithms)

Since the page table is stored in memory, **Translation Lookaside Buffer (TLB)** is used to speed up the address translation by caching frequent page table entries.



Logical to Physical Address Translation

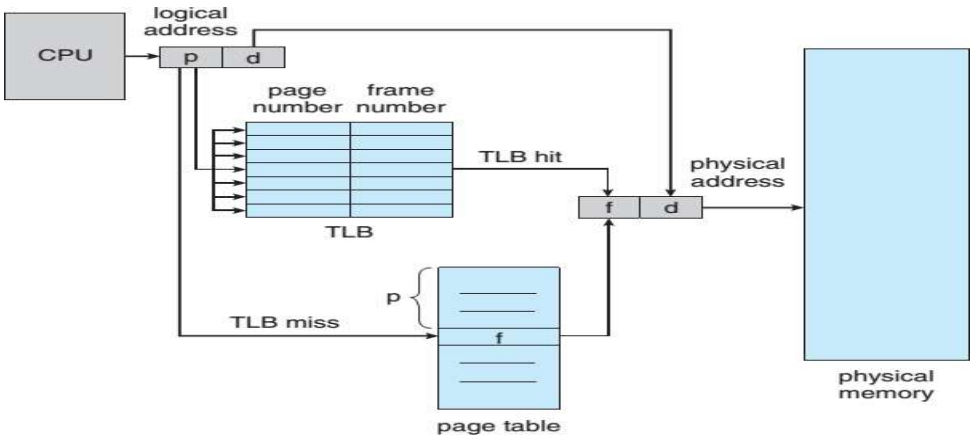
The **Memory Management Unit (MMU)** is responsible for translating **logical addresses into physical addresses**.

- A logical address consists of:
  - **Page Number**
  - **Offset**

The MMU uses the page number to **access the frame number from the page table** and combines it with the offset to get the physical address.

Example:

If the CPU requests the **10th word of 4th page of process P3**, and page 4 of P3 is stored in **frame 9**, then the **10th word of frame 9** will be accessed in physical memory.



VIRTUAL MEMORY AND DEMAND PAGING

Virtual Memory allows a system to run programs that require more memory than physically available RAM.

- It uses **part of the hard disk as an extension of RAM**.
- The operating system **divides the logical memory into pages** and only loads the required pages into RAM.
- This technique provides **efficient memory usage** and supports **multitasking**.

Think of virtual memory like your study table and drawers. If your table (RAM) is full, you keep some books (pages) in drawers (hard disk) and bring them up only when needed.

Paging and Swapping in Virtual Memory

- Memory is divided into **pages**.
- Only some pages are loaded into RAM.
- When more pages are needed:
    - Unused pages in RAM are **swapped** to the hard disk.
    - Needed pages are **loaded from disk to RAM**.

The OS uses a **page table** to maintain the mapping between **virtual addresses** and **physical locations** (RAM or disk).

Advantages of Virtual Memory

- Allows **more programs** to run simultaneously (increased multiprogramming).
- Supports **large applications** even with limited RAM.
- Reduces hardware cost (no need for large RAM).

Disadvantages of Virtual Memory

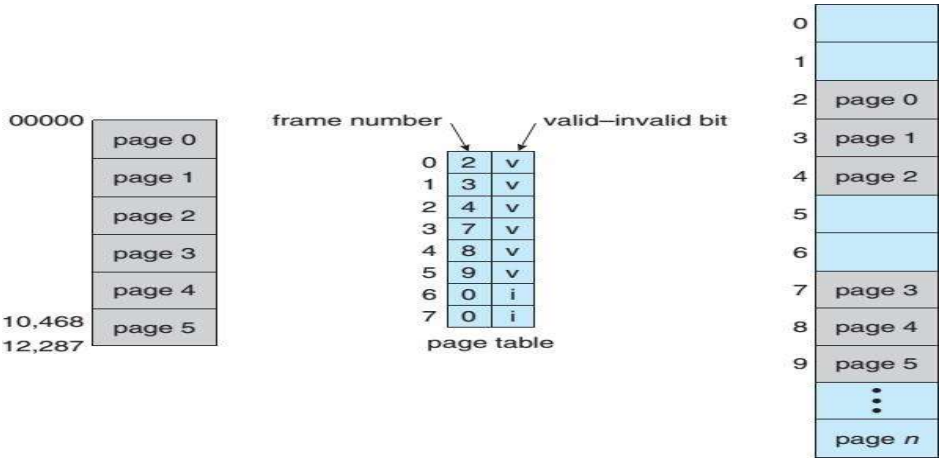
- Slower performance when frequent swapping happens.
- Switching between applications can take more time.
- Occupies space on the hard disk.

Demand Paging and Page Faults

In **Demand Paging**, pages are **loaded into RAM only when needed**, not in advance.

- All pages are initially stored in **secondary memory**.
- When a page is needed, the OS checks if it is in RAM:
    - If **yes**: continues execution.
    - If **no**: a **Page Fault** occurs.

The missing page is then fetched from the hard disk into RAM.



Page Fault

A **page fault** occurs when the process tries to access a page **not currently in RAM**.

- The OS pauses the process.
- Fetches the page from secondary memory.
- Loads it into a free frame in RAM.
- Resumes the process.

Too many page faults slow down the system.

Thrashing

Thrashing occurs when there are **excessive page faults**.

- The CPU spends more time **loading and unloading pages** instead of executing instructions.
- This leads to **very poor system performance**.

It happens when the system **doesn't have enough RAM** for all active processes.

Effective Access Time (EAT)

Effective Access Time is the average time taken to access memory, considering both normal memory access and page faults.

- Let:
  - **PF** = Page fault rate (e.g., 0.1 = 10%)
  - **S** = Time to service a page fault (includes reading from disk and updating tables)
  - **ma** = Memory access time (in RAM)

Formula:

$$EAT = PF \times S + (1 - PF) \times ma$$

If page fault rate is high, EAT becomes high, making memory access slow.

PAGE REPLACEMENT ALGORITHMS

When there is no free frame in memory and a page fault occurs, the system must replace one of the existing pages in memory with the required page from the disk. This process is known as **Page Replacement**.

To handle this situation, the system uses **page replacement algorithms** to decide **which page to remove**.

Page Replacement Process

1. The location of the required page on the disk is identified.
2. The system checks for a free frame:
    - If a free frame exists, it is used directly.
    - If no free frame is available, a page replacement algorithm is used to select a victim frame.
    - If the victim page has been modified, it is written to disk before replacement.
3. The required page is read into the newly freed frame.
4. The page table and frame table are updated accordingly.
5. The process resumes from where it was interrupted due to the page fault.

Modify Bit (Dirty Bit)

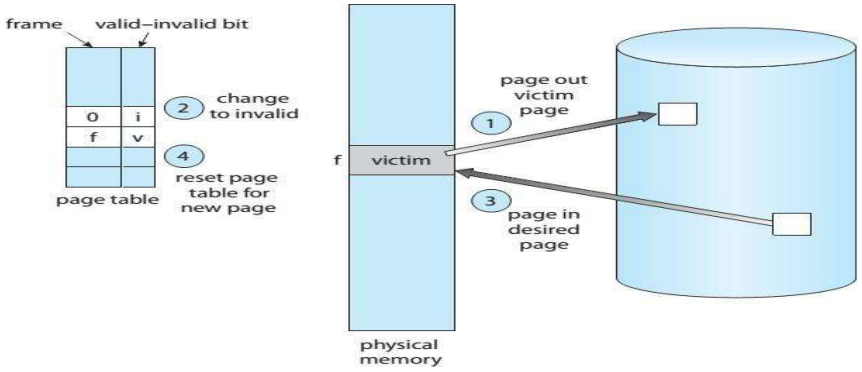
Each page or frame is associated with a **modify bit**, which indicates whether the page has been changed since it was loaded into memory.

- If the modify bit is **set**, the page has been modified and must be written to disk before being replaced.
- If the modify bit is **not set**, the page remains unchanged and does not need to be saved again to disk.

This mechanism helps avoid unnecessary writing to the disk.

Relationship Between Frames and Page Faults

The number of page faults depends on the number of frames available in memory. In general, increasing the number of frames reduces the number of page faults. However, in some cases, increasing frames can result in **more** page faults, which is known as **Belady's Anomaly**.



Page Replacement Algorithms

The various algorithms are:

1. FIFO (First – in – First – Out)
2. LRU (Least Recently Used)
3. Optimal (OPT)

First-In-First-Out Page Replacement Algorithm

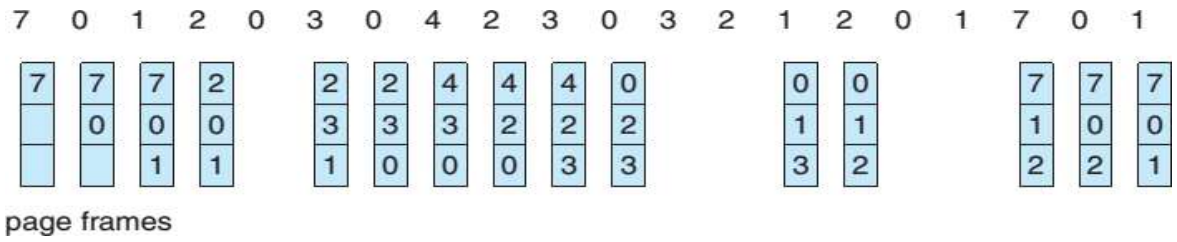
This is the **simplest** page replacement algorithm.

- It associates time with each page when the page was brought into memory.
- When a page must be replaced, the **oldest page** (that came first) is chosen.
- A queue is maintained to track the order of pages.

Example: Reference String:

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

Frames: 3 → **Page Faults:** 15



Advantages:

- Very easy to understand and implement.
- Requires minimal bookkeeping (just a simple queue to track the order of insertion).

Disadvantages:

- Can result in **Belady's Anomaly**, where increasing the number of frames leads to more page faults.
- Does **not consider page usage patterns**, so important pages might be removed too early.

Optimal Page Replacement Algorithm (OPT Algorithm)

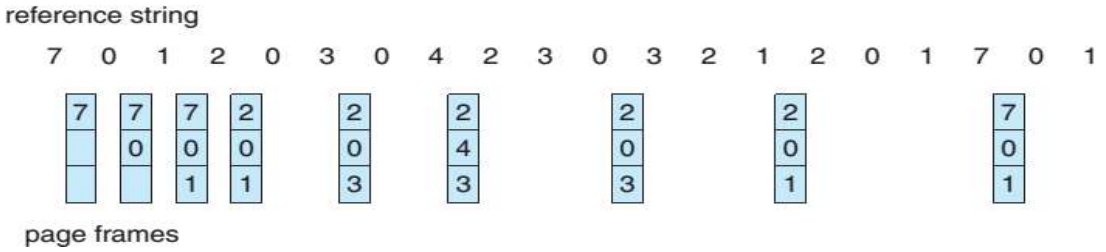
This algorithm gives the **minimum number of page faults**.

- It replaces the page that will **not be used for the longest period of time in the future**.
- It is used mainly for **theoretical comparison**, as future knowledge is required.

Example: Reference String:

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

Frames: 3 → **Page Faults:** 9



Advantages:

- **Lowest possible page fault rate**.
- Serves as a **benchmark** for comparing the performance of other algorithms.

Disadvantages:

- **Not implementable in practice**, because it needs knowledge of **future page references**, which is not possible at runtime.
- Only suitable for simulations and performance analysis

LRU Page Replacement Algorithm

This algorithm replaces the page that has **not been used for the longest time**.

- It associates with each page the **time of its last use**.
- When a page must be replaced, LRU selects the page that hasn't been used recently.

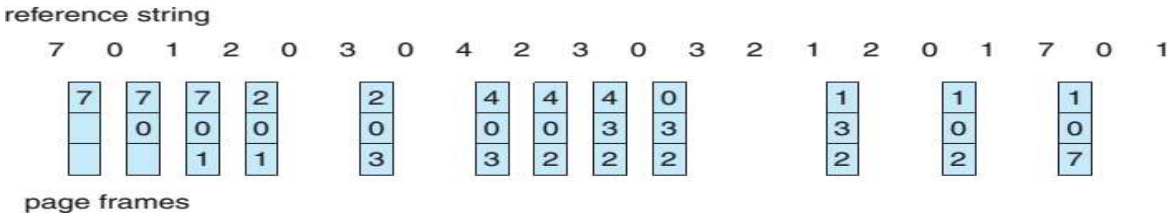
Implementation:

- Using a **stack** or counters to keep track of recent usage.
- More practical and performs better than FIFO.

Example: Reference String:

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

Frames: 3 → **Page Faults:** 12



Advantages:

- More accurate than FIFO, as it is based on recent usage.
- Generally, produces **fewer page faults** than FIFO.
- Good performance in practical scenarios.

Disadvantages:

- **Needs extra hardware or software support** to track recent usage of pages.
- Can be **costly to implement** because maintaining usage history is complex.
- Searching and updating usage data adds **computational overhead.**

SYSTEM PROTECTION

Definition:

*Protection in an OS means **controlling access** to files, programs, devices, and other system resources. It makes sure that only the **right users** and **processes** can use or change them.*

Protection in an operating system refers to the **techniques and mechanisms** that are used to **control access** to resources like files, memory, CPU, and devices.

It ensures that:

- Only **authorized users** or processes can access specific resources.
- Resources are used **safely and according to rules or policies**.

Just like how you lock your phone or laptop to keep others from accessing your personal data, an operating system uses protection mechanisms to **safeguard important parts of the system**.

Think of protection as a helper for multiprogramming operating systems, where multiple users and programs are active at the same time. It makes sure that one user or process does not disturb another user's data or resources.

Need for Protection

Protection is necessary to ensure the system works **correctly and securely**. Here are the main reasons why:

1. **Prevent Unauthorized Access:**
    - Protection stops **unauthorized users** (like hackers or other users) from entering or making changes to the system.
    - Example: A student shouldn't access the exam question paper file meant only for the teacher.
2. **Monitor Active Processes:**
    - The system must keep track of which programs are currently running.
    - Only **trusted and safe programs** should be allowed to run to avoid viruses or harmful operations.
    - Example: If a malware app tries to run, protection should block it.
3. **Improve System Reliability:**
    - Protection helps in **detecting mistakes**, blocking harmful actions, and keeping the system running without crashes.
    - It **prevents one user's actions** from affecting the entire system.
    - Example: If one student deletes a file, it should not delete files of other students.

Role of Protection in a System

The **main role of protection** is to **enforce policies**—rules that say **who can do what** with different system resources.

These policies:

- Might be defined by the **OS designers** (when the system is built),
- Might be defined by the **system administrators** (like IT staff managing users and security),
- Or might be set by **individual users** (like protecting their own files or folders).

This means protection is **not only the responsibility of the OS designers**, but also:

- The **system administrators** who maintain it, and
- The **application developers** who must make sure their apps are safe and don't allow misuse.

Goals of Protection

These are the **main aims** of protection in an OS:

1. **Prevent Misuse of Resources:**
    - Users or programs should not use more resources than allowed.
    - Example: A normal user shouldn't be able to format the hard disk.
2. **Enforce Rules for Shared Resources:**
    - Resources like **memory, CPU time, printers** are shared. Protection makes sure they are used properly and fairly.
    - Example: Two users sending print jobs should not disturb each other.
3. **Minimize Damage in Case of Failure:**
    - If a program crashes or misbehaves, it should not damage the entire system.
    - Protection **isolates** the problem and prevents it from spreading.
    - Example: If a game crashes, it shouldn't affect your running Excel sheet.
4. **Support a Stable System:**
    - Protection adds **discipline** to how users and programs behave.
    - A well-protected system is more **reliable, secure, and efficient** in its operations.

Principles of Protection

These are the **rules and best practices** for designing a protected system:

1. **Least Privilege Principle:**
    - Give only **minimum access** to users or programs that is required to perform their task.
    - This reduces the risk of misuse.
    - Example: A student needs only read access to notes, not delete access.
2. **Minimize Harm from Failures:**
    - By giving limited access, if a mistake happens, the **damage will also be limited**.
    - Example: If a low-level user clicks on a virus, it can't spread system-wide because they don't have permission.
3. **Special Privileges Only When Needed:**
    - Some roles like the **backup team, system maintenance team, or network admins** need special rights.
    - These privileges should be given **carefully** and only when required.
4. **Use Individual User Accounts:**
    - Every user should have their **own username and password**.
    - This helps in **tracking activity** and limiting access.
    - Example: In a college lab, every student logs in with their own ID.
5. **Limit Use of Root (Admin) Account:**
    - The root or admin account has full control over the system.
    - It should be used **only when absolutely necessary** like software installation, system updates, etc.
    - Misuse of this account can **cause major damage**, so it must be protected and rarely used.

DOMAIN OF PROTECTION

In a computer system, processes must be restricted in their access to system resources to prevent misuse, whether intentional or accidental. This is where the concept of protection becomes important. Protection ensures that each process uses only the resources it is allowed to and does so according to clearly defined policies. The protection mechanism is a foundation for a secure and stable multiprogramming environment.

Understanding Protection and Domains

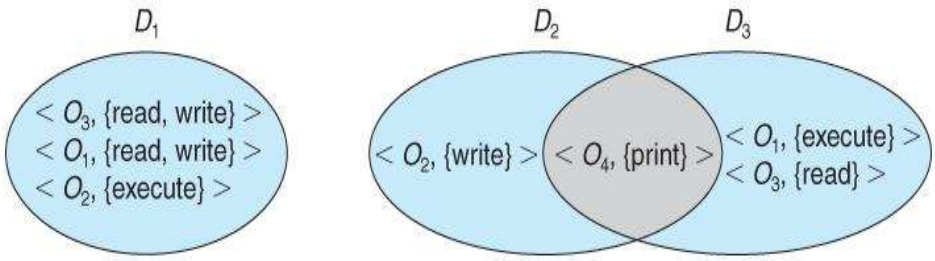
Each process in the system operates within a protection domain. A domain defines a set of objects and the operations that are allowed on those objects by processes operating in the domain. Objects can include files, memory segments, I/O devices, etc., while operations include read, write, execute,

etc. By associating different domains with different processes, the system can restrict access to sensitive resources and ensure isolation among processes.

Objects and processes are treated as abstract data types, with each object having a set of valid operations. The association of a domain with a process controls the set of operations that process can perform.

Domain Structure

The structure of a domain is characterized by a list of <object, access rights> pairs. Each domain contains a collection of objects and a list of access rights or operations that are allowed on those objects.



- A domain defines the resources a process may access and the types of operations allowed on each resource.
- An access right is the permission to execute a specific operation on a specific object.
- A domain can be represented as:

  Domain D = {(O1, [read, write]), (O2, [execute]), ...}

- Domains can be disjoint (no shared objects) or overlapping (shared objects with possibly different permissions).

The association between a process and a domain can be either static or dynamic:

- **Static association:** The domain remains fixed during the lifetime of the process. Any change in access rights must be made by changing the contents of the domain.
- **Dynamic association:** Processes are allowed to switch from one domain to another. This requires a mechanism for domain switching.

Domains may be implemented in several ways:

- As individual users: Each user has their own domain of accessible resources.

- As processes: Each process has a specific domain.
- As procedures: Changing domains corresponds to changing the procedure ID.

Access Matrix Model

To define and enforce protection, the Access Matrix model is used. It provides a formal framework to represent the rights of each domain over each object.

- The access matrix is a conceptual representation where:
  - **Rows** represent **domains** (processes/users).
  - **Columns** represent **objects** (resources).
  - Each cell **Access (i, j)** in the matrix holds a set of rights that domain *i* has over object *j*.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

For example, if a process in domain D1 wants to read object O1, the matrix cell Access(D1, O1) must contain the read permission.

Use of Access Matrix

The access matrix not only serves as a means to enforce current access policies but can also be used to implement dynamic protection:

- Operations may be added or removed during execution.
- The matrix can support **special rights**:
  - **owner** – allows addition or removal of rights for an object.
  - **copy** – allows rights to be copied to another domain.
  - **control** – allows one domain to modify the rights of another domain.
  - **transfer** – allows switching from one domain to another.

For instance, if domain D2 has control rights over domain D4, then D2 can modify the access rights available to D4.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Copy Rights Variations

The ability to copy rights in the access matrix can be implemented in several ways:

- **Copy with asterisk (*)**: The right and the right to copy it are both transferred.
- **Copy without asterisk**: Only the access right is copied, not the ability to copy it again (limited copy).
- **Transfer**: The right is moved from one domain to another and removed from the source domain.

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Owner and Control Rights

- **Owner rights**: Provide permission to add or remove access rights from the same column (object).
- **Control rights**: Enable a domain to manage access rights of other domains (rows).

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Implementation of Access Matrix

While the access matrix is conceptually useful, its direct implementation is inefficient for large systems. Therefore, several practical implementations are used:

1. **Global Table**
   - A single table listing all <domain, object, rights> entries.
   - Simple but inefficient in terms of space and difficult to manage.
2. **Access Lists for Objects**
   - For each object, maintain a list of domains and their access rights.
   - Efficient for checking rights for a given object.
   - Default rights can be used to handle common cases.
3. **Capability Lists for Domains**
   - For each domain, maintain a list of objects it can access along with the allowed operations.
   - Capability lists are not directly accessible to user programs.
   - These are protected using hardware tagging or isolated memory segments.
4. **Lock-Key Mechanism**
   - Each object (resource) has a set of locks (bit patterns).
   - Each domain has a set of keys.
   - Access is granted if a domain's key matches the object's lock.
   - Domains cannot alter their own keys.

This mechanism is efficient and provides a clear separation between the process and its capabilities.