Logical Address

A logical address is generated by the CPU while a program runs. It represents the address from the process's perspective and does not exist physically, hence it is also called a virtual address.

- The logical address space is the set of all logical addresses a process can generate.
- Programs use logical addresses to reference memory, and the MMU translates them into physical addresses when accessing actual memory.

Physical Address

A physical address is the real location in main memory (RAM) where data or instructions are stored. The physical address space consists of all physical addresses corresponding to logical addresses.

- The MMU performs address translation using a page table, mapping each logical page to a physical frame.
- This allows processes to access memory transparently, without knowing actual memory locations.

Memory Management Unit

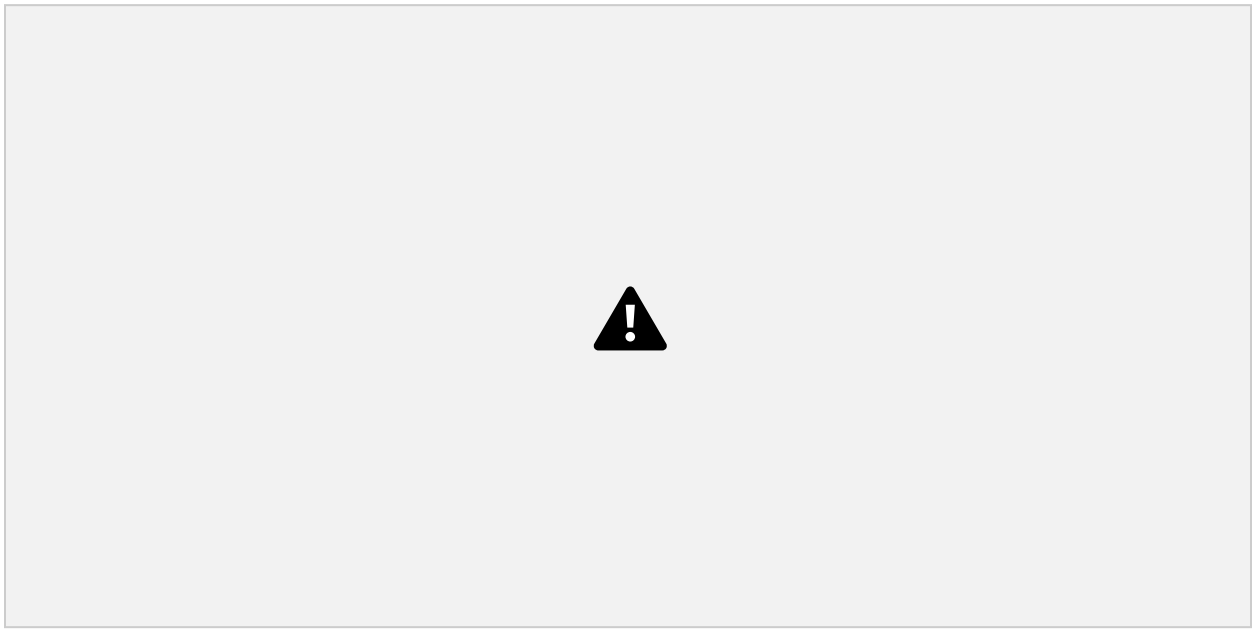
The Memory Management Unit (MMU) is a hardware component in the computer system that handles all memory and caching operations associated with the CPU. Its main function is to translate **l**ogical (virtual) addresses generated by the CPU into physical addresses in main memory.

Logical Address vs. Physical Address

Logical Address	Physical Address
Generated by the CPU during program execution	Generated by the Memory Management Unit (MMU)
Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
User can view and access the logical address of a program.	User can never view physical address of program.
Can change during program execution (due to relocation, paging, etc.)	Generally fixed once assigned in memory
The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Logical address can be change.	Physical address will not change.
Virtual address.	Real address.

Address Binding is the mapping of a logical (virtual) address to a physical address. It allocates a physical memory region to a logical pointer. It is the process of assigning physical memory addresses to program instructions and data. The OS handles this aspect of computer memory management on behalf of programs that need memory access.

- Executes at compile time, load-time, or execution-time.
- The linker and loader assign actual memory addresses.
- Allows code relocation in memory.
- Supports dynamic memory allocation at runtime.



Example: Consider a program that generates the following logical addresses during execution:

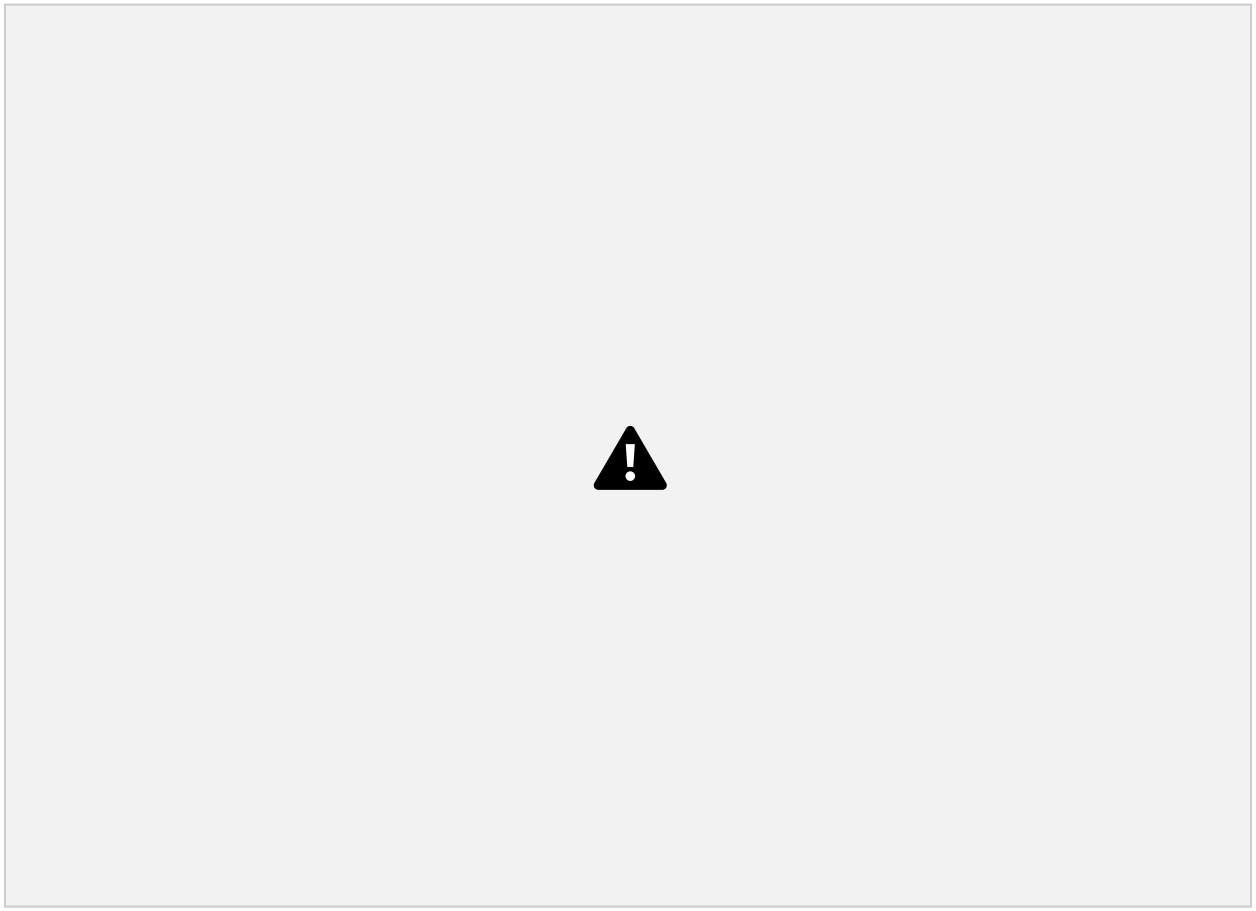
- Logical Addresses: `0, 4, 8`
- Base (physical) address: `1000`

Logical Address	Physical Address
0	1000
4	1004
8	1008

`Address Binding Rule:` `Physical Address = Base Address + Logical Address`

Types of Address Binding

The mapping of data and computer instructions to actual memory locations is known as address binding. In computer memory, logical and physical addresses are employed.



Compile-Time Address Binding

Compile-time address binding assigns fixed physical addresses to symbolic addresses during compilation. Functions and global variables occupy the same memory locations at runtime. This method is simple and efficient but cannot adapt to runtime changes.

- Done by the compiler during compilation.
- Symbolic addresses linked to fixed physical addresses.
- Addresses determined before program execution.

- Suitable for functions and global variables.
- Simple and efficient, but not adjustable at runtime.

Load Time Address Binding

Load Time Address Binding occurs when a program is loaded into memory. The loader assigns physical addresses, and the linker resolves external references. More flexible than compile-time binding, allowing memory adjustments and support for dynamic libraries.

- Take place when the program is loaded into memory.
- Performed by the OS memory manager (loader).
- Loader assigns physical memory addresses during loading.
- More flexible than compile-time binding.
- Allows memory adjustments at load time.
- Supports dynamic libraries

Execution Time or Dynamic Address Binding

Execution Time Address Binding occurs during program execution, allowing memory locations to change. Supports dynamic memory allocation, late binding, and polymorphism. Used in dynamic and object-oriented languages, with modern OSs like Windows, Linux, and Unix.

- Performs during program execution.
- Program memory locations may change at runtime.
- Memory can be allocated and deallocated dynamically.
- Most flexible type of binding.
- Common in dynamic and object-oriented languages.

Difference Between Contiguous and Non-contiguous Memory Allocation

Contiguous Memory Allocation	Non-Contiguous Memory Allocation
Contiguous memory allocation allocates consecutive blocks of memory to a file/process.	Non-Contiguous memory allocation allocates separate blocks of memory to a file/process.
Faster in Execution.	Slower in Execution.
It is easier for the OS to control.	It is difficult for the OS to control.

Contiguous Memory Allocation	Non-Contiguous Memory Allocation
Overhead is minimum as not much address translations are there while executing a process.	More Overheads are there as there are more address translations.
Both Internal fragmentation and external fragmentation occurs in Contiguous memory allocation method.	Only External fragmentation occurs in Non-Contiguous memory allocation method.
It includes single partition allocation and multi-partition allocation.	It includes paging and segmentation.
Wastage of memory is there.	No memory wastage is there.
In contiguous memory allocation, swapped-in processes are arranged in the originally allocated space.	In non-contiguous memory allocation, swapped-in processes can be arranged in any place in the memory.
It is of two types: 1. Fixed(or static) partitioning 2. Dynamic partitioning	It is of five types: 1. Paging 2. Multilevel Paging 3. Inverted Paging 4. Segmentation 5. Segmented Paging
It could be visualized and implemented using Arrays.	It could be implemented using Linked Lists.
Degree of multiprogramming is fixed as fixed partitions	Degree of multiprogramming is not fixed

Paging vs. Segmentation

Paging divides memory into fixed-size blocks called pages, which simplifies management by treating memory as a uniform structure. In contrast,

segmentation divides memory into variable-sized segments based on logical units such as functions, arrays, or data structures.

Both methods offer distinct advantages and are chosen based on the specific needs and complexities of applications and system architectures. Often, modern systems combine both techniques to leverage the benefits of each.

Paging

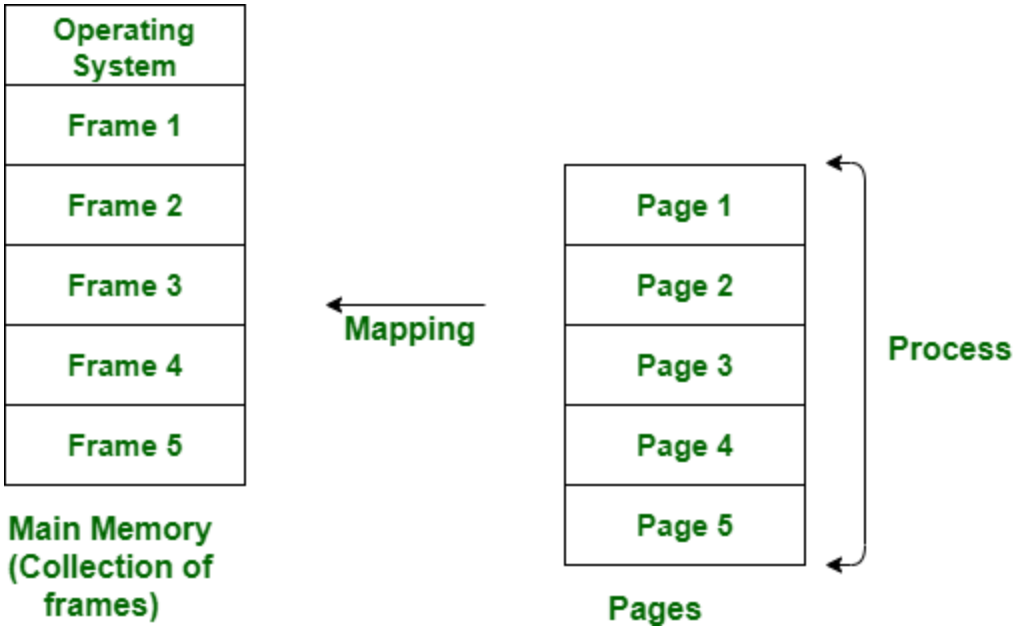
Paging is a method or technique which is used for non-contiguous memory allocation. It is a fixed-size partitioning theme (scheme). In paging, both main memory and secondary memory are divided into equal fixed-size partitions. The partitions of the secondary memory area unit and main memory area unit are known as pages and frames respectively.

Features of Paging

- **Fixed-Size Division:** Memory is divided into fixed-size pages, simplifying memory management.
- **Hardware-Defined Page Size:** Page size is set by hardware and is uniform across all pages.
- **OS-Managed:** The operating system handles paging, including maintaining page tables and free frame lists.
- **Eliminates External Fragmentation:** Paging avoids external fragmentation but can suffer from internal fragmentation.
- **Invisible to User:** Paging is transparent to programmers and users, simplifying software development.

Paging is a memory management method accustomed to fetching processes from the secondary memory into the main memory in the form of pages. in paging, each process is split into parts wherever the size of every part is the same as the page size.

The size of the last half could also be but the page size. The pages of the process area unit hold on within the frames of main memory relying upon their accessibility.



Segmentation

Segmentation is another non-contiguous memory allocation scheme, similar to paging. However, unlike paging which divides a process into fixed-size pages segmentation divides memory into variable-sized segments that correspond to logical units such as functions, arrays, or data structures.

Features of Segmentation

- **Variable-Size Division:** Memory is divided into logical segments of varying sizes based on program structure.
- **User/Programmer-Defined Sizes:** Segment sizes are defined by the programmer or compiler, reflecting logical program units.
- **Compiler-Managed:** Segmentation is primarily managed by the compiler, with OS support for memory allocation.
- **Supports Sharing and Protection:** Segmentation facilitates sharing code/data between processes and easy implementation of protection.
- **Visible to User:** Segmentation is visible to programmers, allowing better control over logical memory organization.

In segmentation, both main memory and secondary memory are not divided into equal-sized partitions. Instead, they are split into segments of varying sizes. These segments are tracked using a data structure called the segment table.

The segment table stores information about each segment, primarily:

- Base: The starting physical address of the segment in memory.

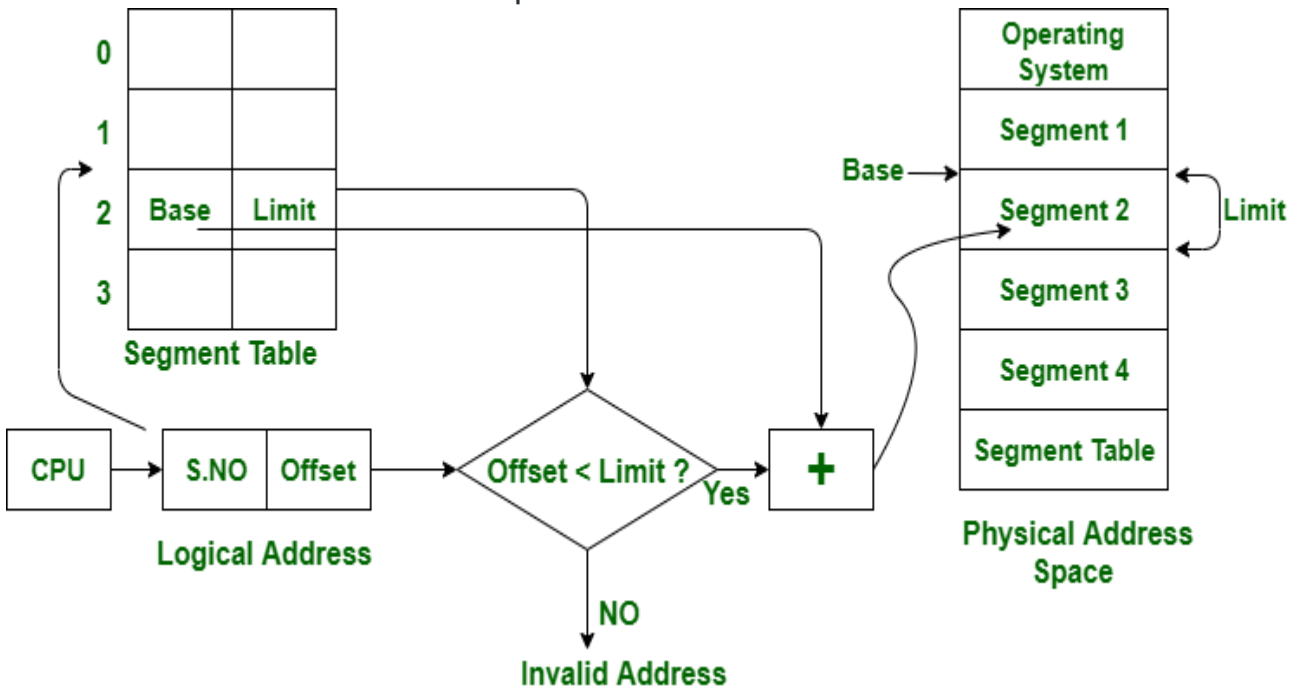
- Limit: The length (or size) of the segment.

When accessing memory, the CPU generates a logical address composed of:

- A Segment Number
- A Segment Offset

The MMU (Memory Management Unit) uses the segment number to find the corresponding base and limit in the segment table. If the offset is less than the limit, the address is considered valid, and the physical address is computed by adding the offset to the base.

If the offset exceeds the limit, an error (segmentation fault) occurs, indicating an invalid address access attempt.



The above figure shows the translation of a logical address to a physical address.

Paging vs. Segmentation

Feature	Paging	Segmentation
Division Unit	Fixed-size pages	Variable-size segments

Feature	Paging	Segmentation
Managed By	Operating system	Compiler
Unit Size Determined By	Hardware	User/programmer
Address Structure	Page number + page offset	Segment number + segment offset
Data Structure Used	Page table	Segment table
Fragmentation Type	Internal fragmentation	External fragmentation
Speed	Faster	Slower
Programmer Visibility	Invisible to the user	Visible to the user
Sharing	Difficult	Easy
Data Structure Handling	Inefficient	Efficient
Protection	Hard to implement	Easier to apply
Size Constraints	Page = Frame size	No fixed size required
Memory Unit Perspective	Physical unit	Logical unit

Feature	Paging	Segmentation
System Efficiency	Less efficient	More efficient

🔷 Fragmentation in Operating System

🔹 Definition

Fragmentation is a condition in memory management where **available memory is not used efficiently**, leading to **wastage of memory space**.

It mainly occurs when processes are allocated and deallocated dynamically.

🔷 Types of Fragmentation

🔸 1. Internal Fragmentation

✅ Definition

Internal fragmentation occurs when **allocated memory block is larger than the required memory**, resulting in unused space **inside the block**.

⚙ How it Occurs

- Memory is divided into **fixed-size partitions**
- Process is assigned a partition even if it doesn't fully use it

---

📌 Example

- Block size = 10 KB
- Process size = 7 KB
- Unused space = 3 KB (**wasted inside**)

❗ Causes

- Fixed partitioning
- Poor memory allocation strategy
- Large block sizes

🔧 Solutions

- Use **smaller block sizes**
- Use **dynamic memory allocation**
- Use **paging with smaller page size**

📊 Diagram (Conceptual)

```
| Allocated Block (10 KB) |
|-------------------------|
| Used (7 KB)             |
| Unused (3 KB) ❌        |
```

🔸 2. External Fragmentation

✅ Definition

External fragmentation occurs when **free memory is available but not in contiguous form**, so it cannot be allocated to a process.

⚙ How it Occurs

- Memory is allocated and freed dynamically
- Free space breaks into **small scattered holes**

📌 Example

Free memory blocks:

- 10 KB + 20 KB + 15 KB = 45 KB total
  Process needs = 40 KB
  ❌ Allocation fails because memory is **not continuous**

❗ Causes

- Variable-size memory allocation
- Frequent allocation and deallocation

- Process termination creating gaps

🔧 Solutions

- **Compaction** ⟶ Combine small holes into one large block
- **Paging** ⟶ Non-contiguous allocation
- **Segmentation with paging**

📊 Diagram (Conceptual)

```
| P1 | Free 10KB | P2 | Free 20KB | P3 | Free 15KB |

Total Free = 45 KB
But not continuous ❌
```

🔷 Key Differences (Detailed Comparison)

Feature	Internal Fragmentation	External Fragmentation
Location of Waste	Inside allocated block	Outside allocated blocks
Memory Allocation Type	Fixed size	Variable size
Memory Utilization	Partially used block	Scattered free memory
Main Cause	Large fixed partitions	Non-contiguous free space
Occurs In	Paging, fixed partitioning	Segmentation, dynamic allocation
Example	10 KB allocated, 7 KB used	45 KB free but unusable
Severity	Less severe	More severe
Solution	Reduce block size	Compaction, paging
Can be Eliminated?	Cannot be fully removed	Can be reduced

🔷 Advantages & Disadvantages

Internal Fragmentation

✔ Advantages

- Simple memory allocation
- Fast execution

❌ Disadvantages

- Memory wastage inside block
- Inefficient for small processes

External Fragmentation

✔ Advantages

- Flexible allocation
- Efficient for varying process sizes

❌ Disadvantages

- Memory unusable despite being free
- Requires compaction (time-consuming)

……..00