Memory Management

The term memory can be defined as a collection of data in a specific format. It is used to store instructions and process data. The memory comprises a large array or group of words or bytes, each with its own location. The primary purpose of a computer system is to execute programs. These programs, along with the information they access, should be in the **main memory** during execution. The **CPU** fetches instructions from memory according to the value of the program counter.

In a multiprogramming computer, the **Operating System** resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution.

Why Memory Management is Required?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize **fragmentation** issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

Logical vs. Physical Address Space

Memory management involves mapping logical addresses (virtual addresses) generated by the CPU to corresponding physical addresses used by the memory unit.

Logical Address

- **Definition**:
  - Address generated by the CPU during program execution.
  - Also referred to as a **virtual address**.
- **Use**:
  - Independent of the actual physical memory location.
- **Logical Address Space**:
  - The set of all logical addresses generated by a program.

Physical Address

- **Definition**:
  - Address seen by the memory hardware (actual location in RAM).
- **Physical Address Space**:

- The set of all physical addresses corresponding to the logical addresses.

Address Binding

Address binding refers to mapping addresses at different stages of a program's lifecycle.

1. **Source Code Stage**:
   - Addresses are symbolic (e.g., variable names).
2. **Compile-Time Binding**:
   - Symbolic addresses are converted to relocatable addresses (e.g., "14 bytes from the start").
3. **Linker/Loader Binding**:
   - Relocatable addresses are bound to absolute physical addresses (e.g., 74014).
4. **Execution-Time Binding**:
   - Logical addresses are dynamically mapped to physical addresses using hardware (e.g., Memory Management Unit).

Why Address Binding?

- **Flexibility**:
  - Programs can execute from any location in memory, not just a fixed address like 0000.
- **Efficiency**:
  - Enables multitasking and memory sharing by mapping logical to physical addresses dynamically.

Relocation

Relocation involves ensuring that processes can run in any memory location, even if they are moved during execution.

Why Relocation is Needed:

- **Multitasking:** When multiple processes run, the OS may need to move them to different memory locations for efficiency.
- **Dynamic Loading:** A process may load additional data or modules during execution, requiring relocation.

How Relocation Works:

The OS uses **Base Registers** to store the starting address of a process. Every memory address used by the process is **relocated** relative to this base address.

Example:

1. Suppose a process is loaded at memory location **1000**.

2. If the process uses an instruction to access memory address **200**, the OS calculates the actual address as:

   Actual Address = 1000 + 200 = 1200

Protection

Protection ensures that processes do not interfere with each other's memory and that a process can only access its allocated space.

Why Protection is Needed:

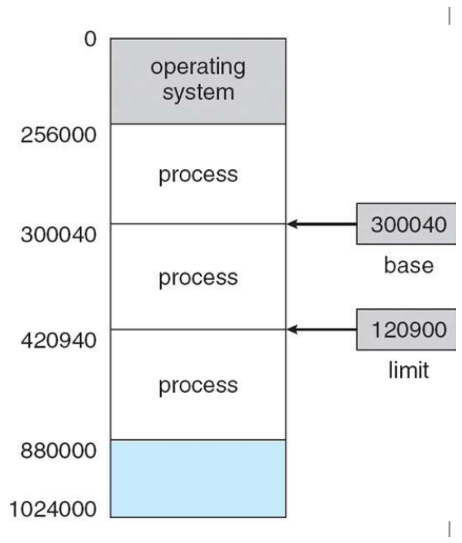
- To prevent **unauthorized access** by processes to memory areas of other processes or the OS.
- To avoid accidental overwrites of memory, ensuring stability and security.

How Protection Works:

- **Limit Registers:** These store the **upper limit** of a process's accessible memory.
- If a process tries to access memory outside its range, the OS generates a **segmentation fault** or an error.

Example:

1. A process is allocated memory from **1000 to 2000**.
2. If it tries to access memory address **2500**, the OS detects this as a violation and blocks it.



Uni-programming vs. Multi-programming Systems

Uni-programming allows only **one program** to run at a time.

- The **memory** is divided into two parts:
  - **User Program**: For the program being executed.

    - **Operating System (OS)**: For managing system resources.
- The OS performs the following steps:
    1. Copies the program from the disk to memory.
    2. Executes the program.
    3. After execution, returns control to the user (prompt).
- The **new program** overwrites the memory occupied by the previous program.

Multi-programming allows **multiple processes** to reside in memory and execute simultaneously, utilizing the CPU more efficiently.

- Multiple processes are **loaded into memory partitions** to share memory resources.
- The CPU switches between processes to ensure progress for all.

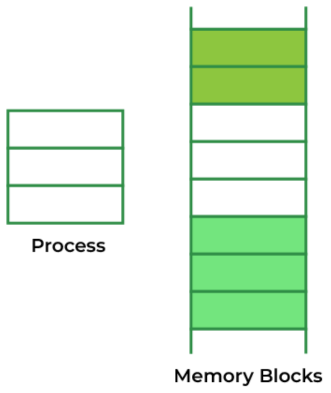
How It Works:

1. Memory is divided into **n partitions** to accommodate multiple processes.
2. The OS schedules processes to run based on priority or availability of resources.

Memory Allocation Mechanisms

1. Contiguous Memory Allocation

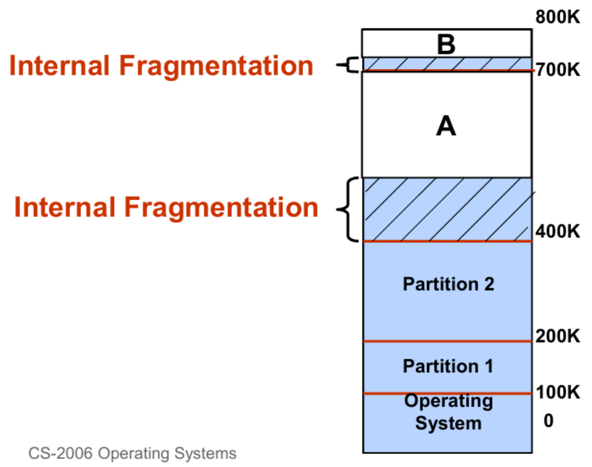
This mechanism assigns a **single block of contiguous memory** to each process. Memory is divided into **fixed-size** or **variable-size** partitions.



a. Fixed (Static) Partitioning

- Memory is divided into **fixed partitions** at system initialization.
- **Advantages:**
    - Simple to implement.
- **Disadvantages:**

- **Internal Fragmentation:** Unused memory within a partition cannot be used by other processes.
- Example: If a 400 KB process is assigned to a 700 KB partition, 300 KB remains unused.



b. Variable (Dynamic) Partitioning

- Partitions are created dynamically based on process size.
- **Advantages:**
  - Reduces internal fragmentation.
- **Disadvantages:**
  - **External Fragmentation:** In **External Fragmentation**, we have a free memory block, but we can not assign it to a process because blocks are not contiguous.
  - Requires **compaction** to combine free blocks.
- Example: **Example:** Suppose three processes p1, p2, and p3 come with sizes 2MB, 4MB, and 7MB respectively. Now they get memory blocks of size 3MB, 6MB, and 7MB allocated respectively. After allocating the process p1 process and the p2 process left 1MB and 2MB. Suppose a new process p4 comes and demands a 3MB block of memory, which is available, but we can not assign it because free memory space is not contiguous. This is called external fragmentation.

Memory Management in Multi-Programming with Fixed Partitions

1. **Process Allocation:**
   - A process is assigned to the smallest partition that can accommodate it.
   - **Problem:** Memory is wasted if the process size is smaller than the partition.
2. **Single vs. Multiple Queues:**
   - **Multiple Queues:** Processes are assigned based on partition size.
     - **Issue:** Small jobs may wait unnecessarily if larger partitions are idle.
   - **Single Queue:** Jobs are allocated to the next available partition regardless of size.

3. **Internal Fragmentation:**
    - Memory inside a partition that is not utilized.
    - Example: A 300 KB process in a 400 KB partition leaves 100 KB unused.

Swapping

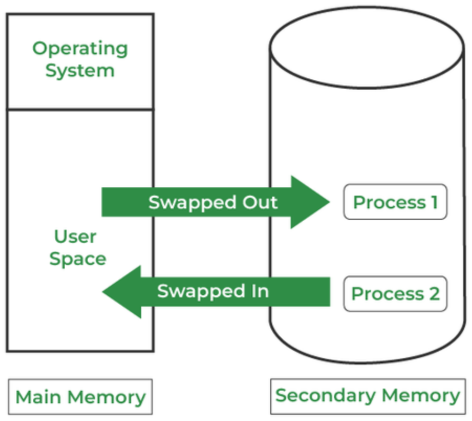
Swapping involves moving processes between memory and disk when memory is insufficient. A swapping allows more processes to be run and can be fit into memory at one time.

Types:

- **Full Swapping:** Entire processes are swapped in and out.
- **Partial Swapping (Virtual Memory):** Parts of processes are swapped using paging.

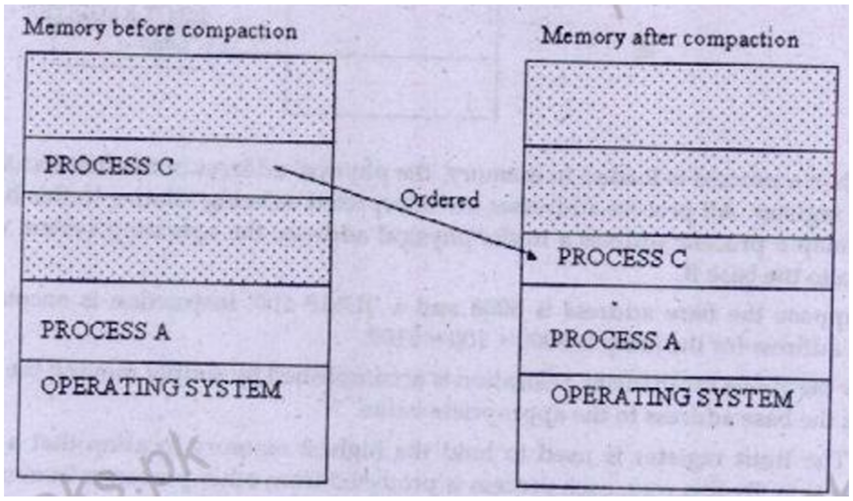
Examples:

- **Round-Robin Scheduling:** Swap out a process when its time quantum expires.
- **Priority Scheduling:** Swap out a lower-priority process for a higher-priority one.



Compaction:

- Combines free blocks into a single large block to reduce external fragmentation.
- **Issue:** Requires CPU overhead for address adjustment.

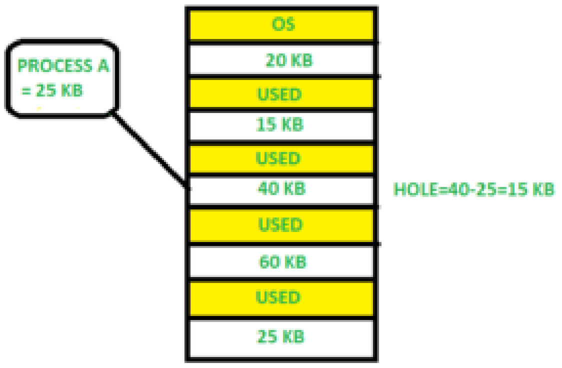


Memory Allocation Algorithms

Memory allocation with involves managing free and allocated memory segments using algorithms that determine how to allocate available holes to processes. Below are the common algorithms:

1. First Fit

- **Process**:
  1. Scan the segment list to find the first hole large enough for the process.
  2. If the hole size exceeds the process size, split it into two parts:
     - One part for the process.
     - The remaining part as a smaller free hole.
- **Advantages**:
  - Fast, as it minimizes the search effort.
- **Disadvantage**:
  - May lead to memory fragmentation over time.



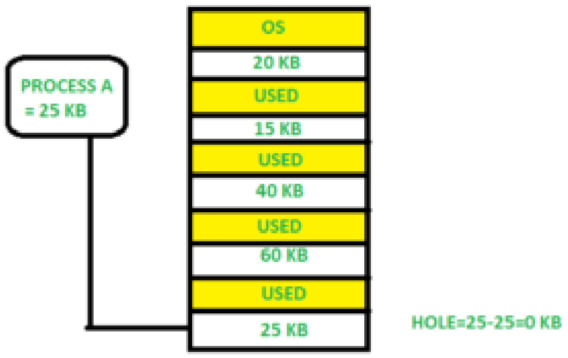
Here, in this diagram, a 40 KB memory block is the first available free hole that can store process A (size of 25 KB), because the first two blocks did not have sufficient memory space.

2. Next Fit

- **Process**:
    1. Similar to First Fit, but resumes the search from where the last allocation ended, instead of starting from the beginning of the list.
- **Advantages**:
    - May slightly reduce search time for some cases.
- **Disadvantage**:
    - Simulations show no significant improvement over First Fit.

3. Best Fit

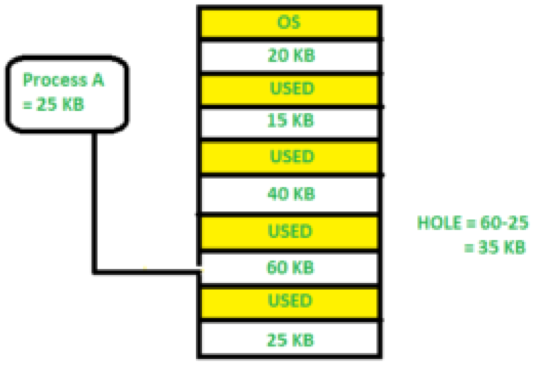
- **Process**:
    1. Searches the entire list to find the smallest hole adequate for the process.
    2. Allocates the hole, leaving minimal unused memory.
- **Advantages**:
    - Reduces leftover free space for each allocation.
- **Disadvantages**:
    - Slower due to the need for a full search.
    - Can create many tiny, unusable holes, leading to increased memory wastage.



Here in this example, first, we traverse the complete list and find the last hole 25KB is the best suitable hole for Process A(size 25KB). In this method, memory utilization is maximum as compared to other memory allocation techniques.

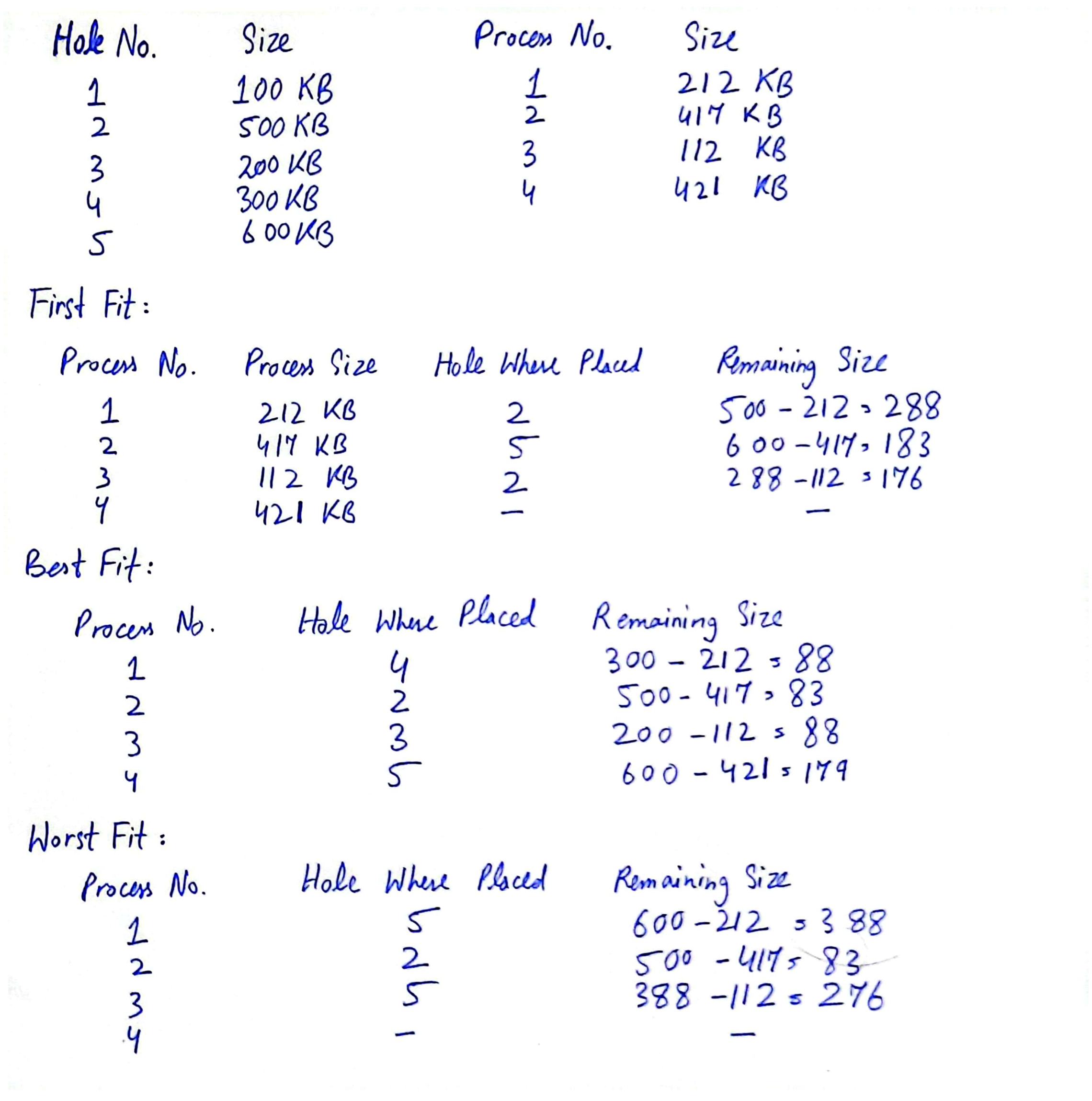
4. Worst Fit

- **Process**:
  1. Allocates memory from the largest available hole.
  2. The leftover memory is expected to remain useful for future processes.
- **Advantages**:
  - Leaves larger free holes after allocation.
- **Disadvantages**:
  - Can result in larger leftover holes that may still be unusable.
  - Often wastes more memory compared to Best Fit.



Here in this example, Process A (Size 25 KB) is allocated to the largest available memory block which is 60KB. Inefficient memory utilization is a major issue in the worst fit.

Examples for Variable (Dynamic) Partitioning:



Hole No.	Size
1	100 KB
2	500 KB
3	200 KB
4	300 KB
5	600 KB

Process No.	Size
1	212 KB
2	417 KB
3	112 KB
4	421 KB

First Fit:

Process No.	Process Size	Hole Where Placed	Remaining Size
1	212 KB	2	500 - 212 = 288
2	417 KB	5	600 - 417 = 183
3	112 KB	2	288 - 112 = 176
4	421 KB	–	–

Best Fit:

Process No.	Hole Where Placed	Remaining Size
1	4	300 - 212 = 88
2	2	500 - 417 = 83
3	3	200 - 112 = 88
4	5	600 - 421 = 179

Worst Fit:

Process No.	Hole Where Placed	Remaining Size
1	5	600 - 212 = 388
2	2	500 - 417 = 83
3	5	388 - 112 = 276
4	–	–

2. Non-Contiguous Memory Allocation

Memory is allocated in **non-adjacent blocks**. It is managed using **paging** or **segmentation**.

a. Paging (Fixed Partitioning)

- Divides memory into **fixed-size blocks (pages)**.
- Each process is divided into pages that can be loaded into any available frame.
- **Advantages:**
    - No external fragmentation.

Important Terms:

- **Logical Address or Virtual Address (represented in bits):** An address generated by the CPU.

- **Logical Address Space or Virtual Address Space (represented in words or bytes):** The set of all logical addresses generated by a program.
- **Physical Address (represented in bits):** An address actually available on a memory unit.
- **Physical Address Space (represented in words or bytes):** The set of all physical addresses corresponding to the logical addresses.

MMU:

The mapping from virtual to physical address is done by the **memory management unit (MMU)** which is a hardware device and this mapping is known as the paging technique.

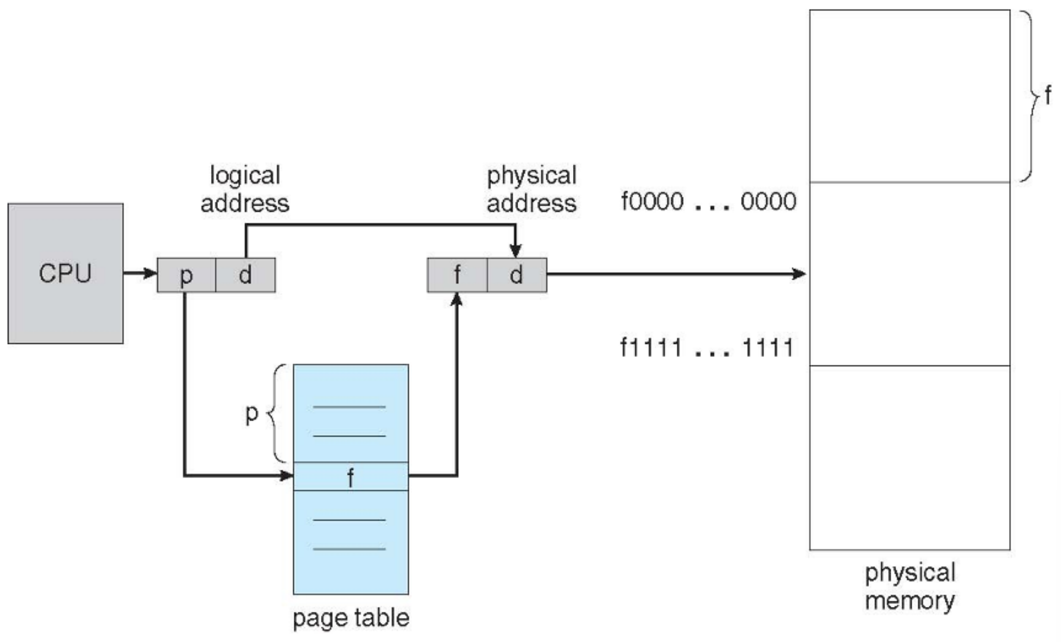
- The Physical Address Space is conceptually divided into several fixed-size blocks, called **frames**.
- The Logical Address Space is also split into fixed-size blocks, called **pages**.
- Page Size = Frame Size

The address generated by the CPU is divided into:

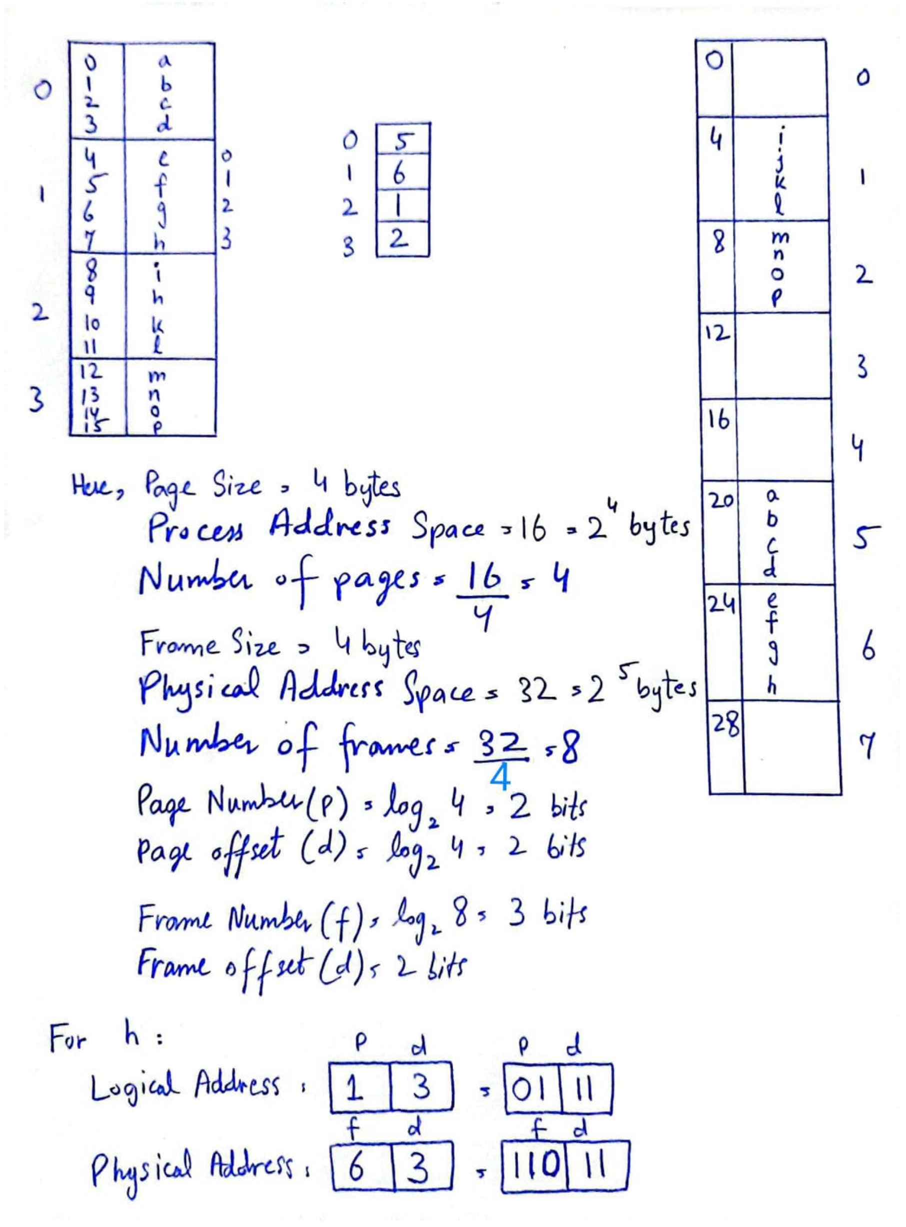
- **Page Number(p):** Number of bits required to represent the pages in Logical Address Space or Page number. Used as an index into a page table which contains base address of each page in physical memory
- **Page Offset(d):** Number of bits required to represent a particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into:

- **Frame Number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number frame
- **Frame Offset(d):** Number of bits required to represent a particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.


logical address
physical address
CPU
p
d
f
d
f0000 … 0000
f1111 … 1111
f
p
f
page table
physical memory

Example:


Here, Page Size = 4 bytes

Process Address Space = 16 = 2^4 bytes

Number of pages = $\frac{16}{4}$ = 4

Frame Size = 4 bytes

Physical Address Space = 32 = 2^5 bytes

Number of frames = $\frac{32}{4}$ = 8

Page Number (p) = $\log_2 4$ = 2 bits

Page offset (d) = $\log_2 4$ = 2 bits

Frame Number (f) = $\log_2 8$ = 3 bits

Frame offset (d) = 2 bits

For h :

Logical Address : $\overset{p}{\boxed{1}}\overset{d}{\boxed{3}} = \overset{p}{\boxed{01}}\overset{d}{\boxed{11}}$

Physical Address : $\overset{f}{\boxed{6}}\overset{d}{\boxed{3}} = \overset{f}{\boxed{110}}\overset{d}{\boxed{11}}$

Calculating Internal Fragmentation:

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
- Internal fragmentation of 2,048 - 1,086 = 962 bytes

Addressing in Paging:

- **Logical address space** of 16 pages of 1024 words each, mapped into a physical memory of 32 frames.
- **Logical address** size?
- **Physical address** size?
- Number of bits for p, f, and d?

- No. of bits for **p** = 4 bits
- No. of bits for **f** = 5 bits
- No. of bits for **d** = 10 bits bits

Logical address size $= |p| + |d|$
$= 4 + 10$
$= 14$ bits

Physical address size $= |f| + |d|$
$= 5 + 10$
$= 15$ bits

Page Table Size:

Let's calculate the **Page Table Size** step by step:

Given:

- **Number of Pages (NP)** = 16 (logical address space contains 16 pages).
- **Page Table Entry Size (PTES)** = $|f|$ = 5 bits (frame number requires 5 bits).

Page Table Size=NP×PTES

Substitute the values:

Page Table Size=16× ceil(5bits / 8)=16 bytes.

Page Table Size is **16 bytes**.

Translation Lookaside Buffer (TLB)

The **Translation Lookaside Buffer (TLB)** is a specialized hardware cache in the memory management unit (MMU) that stores recent translations of logical (virtual) addresses to physical addresses. It helps speed up the process of memory access in systems using paging.

Why TLB is Needed:

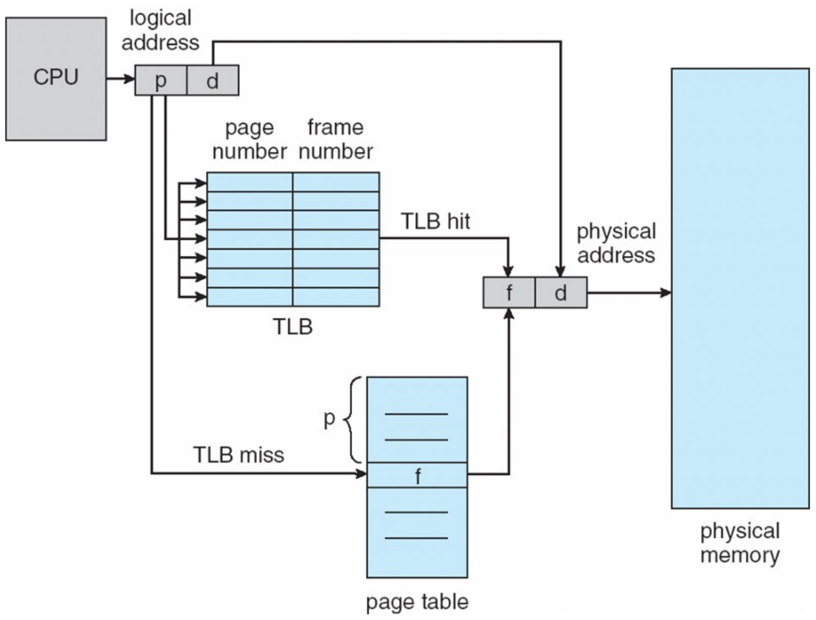
In paging, each memory access requires:

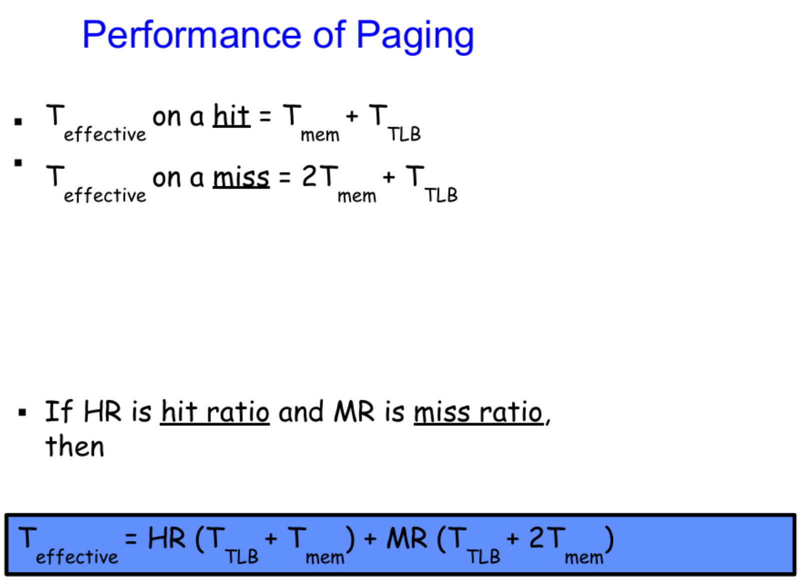
1. A lookup in the **page table** to find the physical frame number.
2. Accessing the physical memory.

This two-step process can slow down memory access. TLB reduces this overhead by caching recently used page-to-frame mappings.

How TLB Works:

1. **TLB Lookup**:
    - When the CPU generates a virtual address, it first checks the TLB for the page number.
    - If the page number is found in the TLB (**TLB hit**), the corresponding frame number is retrieved without accessing the page table.
    - If not found (**TLB miss**), the page table is consulted, and the mapping is added to the TLB.
2. **Memory Access**:
    - After determining the physical frame number, the physical memory is accessed.



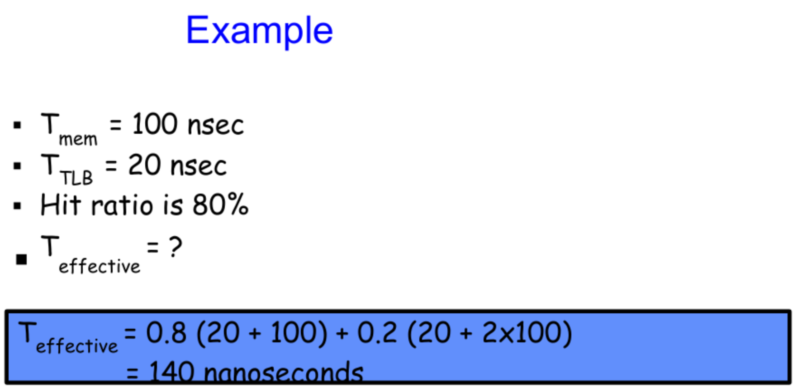


Performance of Paging

- $T_{effective}$ on a <u>hit</u> = $T_{mem} + T_{TLB}$
- $T_{effective}$ on a <u>miss</u> = $2T_{mem} + T_{TLB}$

- If HR is <u>hit ratio</u> and MR is <u>miss ratio</u>, then

$$T_{effective} = HR\,(T_{TLB} + T_{mem}) + MR\,(T_{TLB} + 2T_{mem})$$



Example

- T_{mem} = 100 nsec
- T_{TLB} = 20 nsec
- Hit ratio is 80%
- $T_{effective}$ = ?

$$T_{effective} = 0.8\,(20 + 100) + 0.2\,(20 + 2\times100) = 140 \text{ nanoseconds}$$

b. Segmentation (Dynamic Partitioning)

Segmentation is a memory management technique where a process is divided into **segments**, each representing a logical unit of the program. These units can be **code**, **data**, **stack**, or other functional areas of a program. Unlike paging, segmentation is visible to the programmer and allows the system to treat different parts of a program separately.

Key Features of Segmentation:

1. **Logical Division**: A program is divided into logical segments like functions, arrays, or data blocks.
2. **Variable Size**: Segments are of variable size, unlike fixed-sized pages in paging.
3. **Two-Part Address**: Each address is represented as a **segment number** and **offset** within the segment:
   - **Segment number**: Identifies the segment.
   - **Offset**: Specifies the position within the segment.

How Segmentation Works:

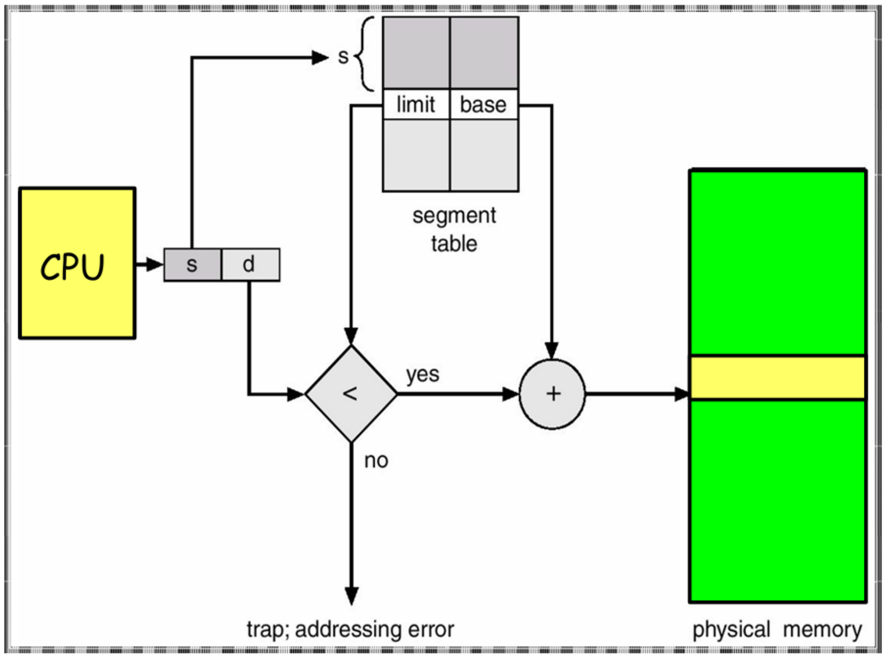
1. **Segment Table**:
    - Each process has a **segment table** that stores the **base address** (starting address) and **limit** (size) of each segment.
    - The table helps translate logical addresses into physical addresses.
2. **Address Translation**:
    - The CPU generates a logical address consisting of a segment number and an offset.
    - The segment table is used to find the segment's base address in physical memory.
    - The **physical address** is calculated as:

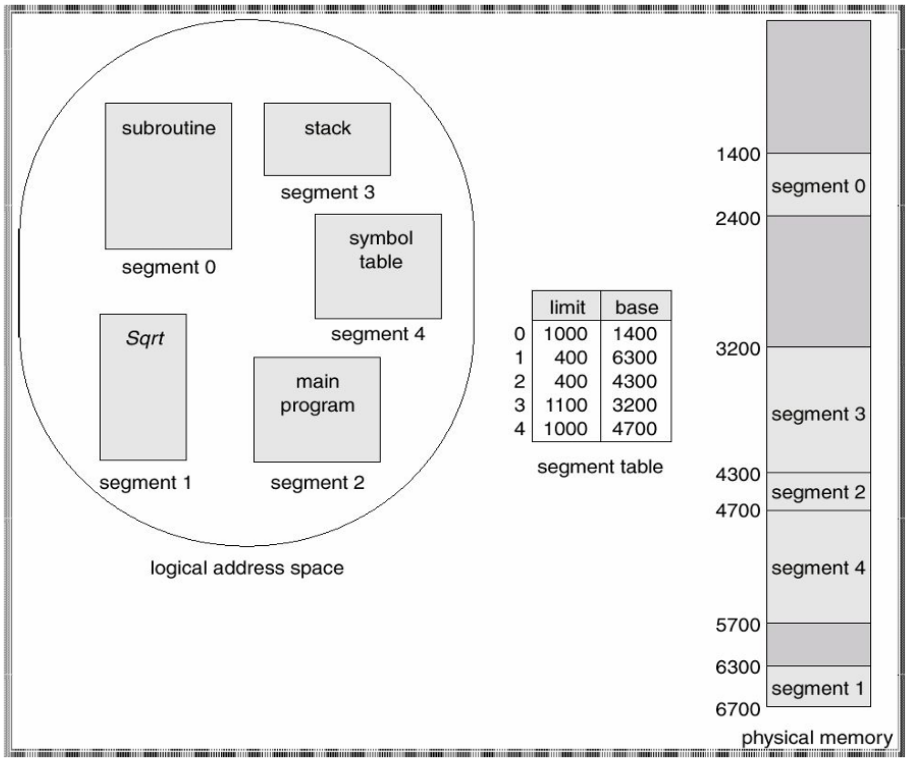
      Physical Address = Base Address + Offset
3. **Bounds Checking**:
    - If the offset is greater than the segment's limit, a **segmentation fault** occurs, indicating an invalid memory access.

Disadvantages of Segmentation:

1. **External Fragmentation**:
    - Segments are of variable size, leading to gaps between allocated memory blocks.
2. **Complexity**:
    - Managing variable-sized segments and segment tables adds overhead.
3. **Slower Access**:
    - Accessing memory requires additional steps for bounds checking and address translation.

Example:





	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

- **Logical and Physical Addresses**
  - **(2, 399) – PA: 4300+399 = 4699**

    = 4700

    - **(4, 1000) ⇒ trap**
    - **(3, 1300) ⇒ trap**