Centre Universitaire de Mila

2nd year of Computer Science degree (LMD)

Module : Operating System 1

Bessouf Hakim

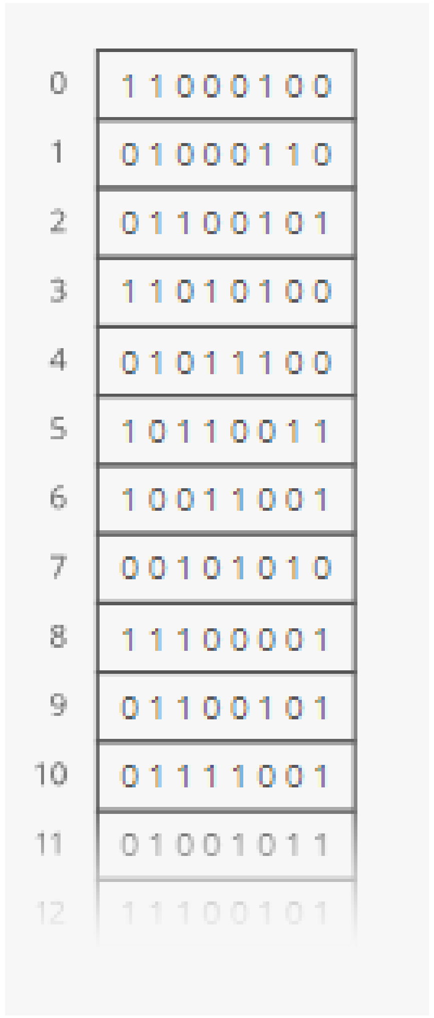
CHAPTER 5

Memory Management

1. Objectives of a Memory Manager
2. Functions
3. Memory Sharing Modes
4. Memory Protection
5. Code Sharing

Introduction

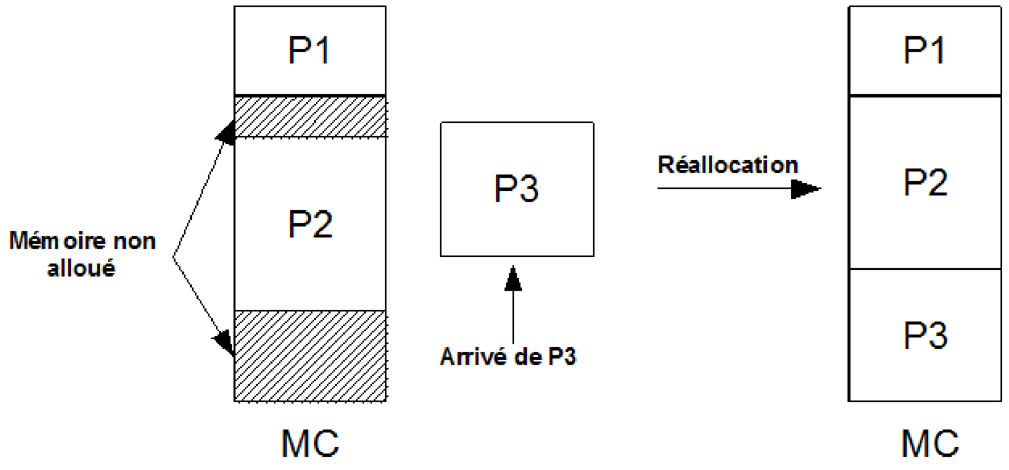
- Main memory is an essential part of the computer; it is used to load programs (data + code) executed by the processor.
- Main memory is divided into multiple cells called memory words, each with a unique address. This address is used by the processor to read and write information.



The Memory Manager

It is a module of the operating system responsible for managing main memory. Its objectives are:

- **Memory Sharing**: The manager must allocate main memory among multiple processes.
- **Memory Protection**: The manager must prevent a process from accessing another process's memory space to avoid conflicts.
- **Reallocation**: The manager must rearrange (reallocate) processes in main memory to free up space and load other processes.



The Memory Manager

A memory manager must perform the following functions:

Allocate memory space to processes: The memory manager must determine a technique for memory allocation.

Free memory space: The memory manager must establish a method to release memory space when a process completes its execution.

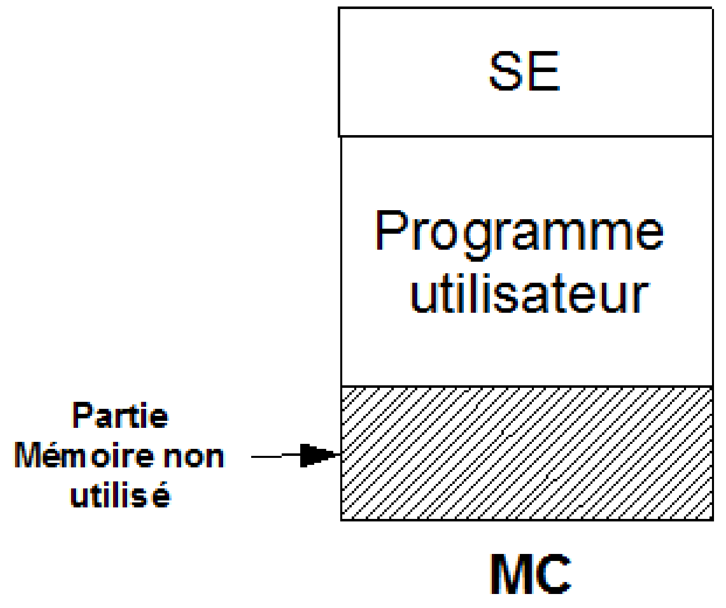
Identify free memory sections: The memory manager must keep track of each part of main memory to manage it efficiently.

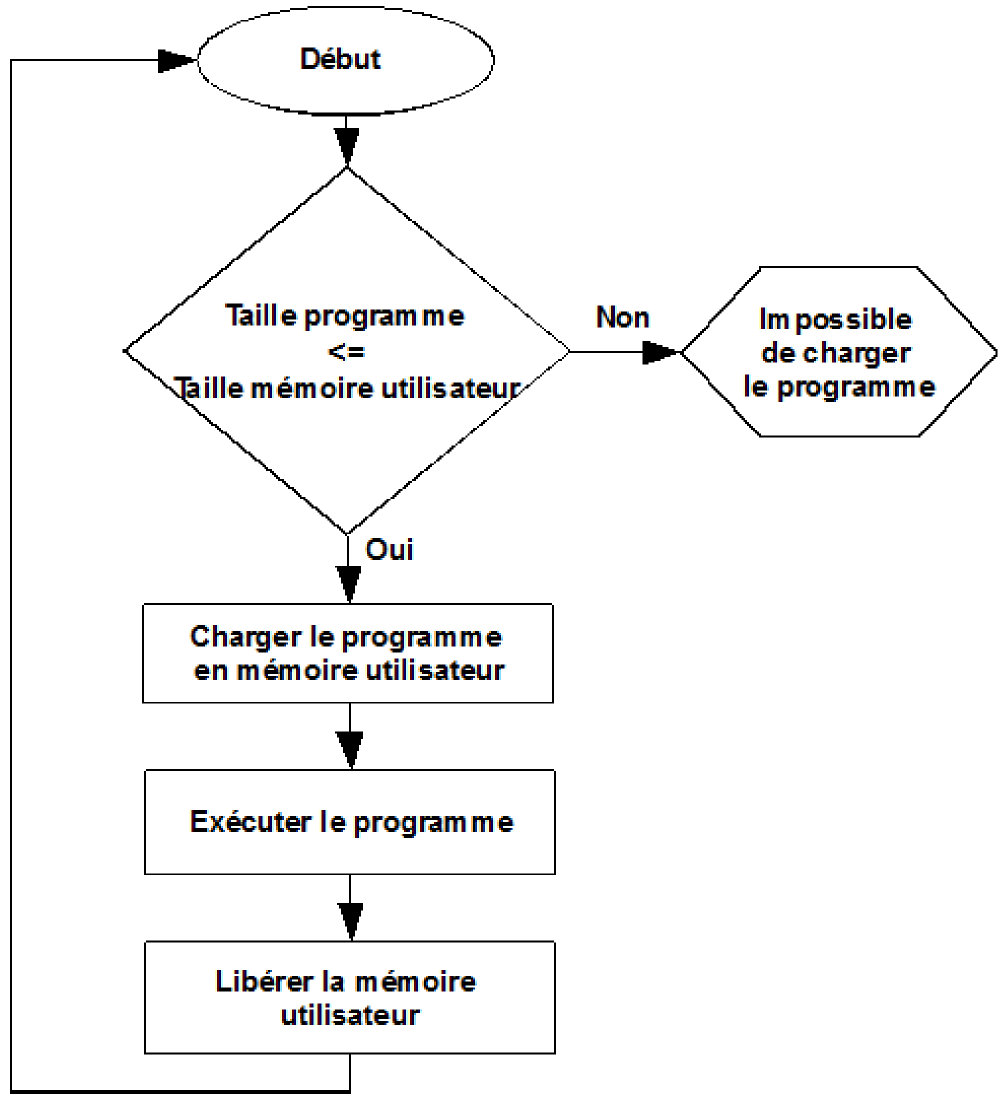
Memory Sharing Modes (Memory Allocation Strategies)

- **Single Continuous Allocation**

In this strategy, only one program is executed at a time (monoprogramming). Main memory is divided into two parts:

- The first part contains the operating system.
- The second part contains the user program




Début
Taille programme <= Taille mémoire utilisateur
Non
Impossible de charger le programme
Oui
Charger le programme en mémoire utilisateur
Exécuter le programme
Libérer la mémoire utilisateur

Flowchart of Memory Allocation with a Single Continuous Zone

Note: This allocation strategy is inefficient because part of the memory may remain unused.

Memory Sharing Modes (Memory Allocation Strategies)

- **Multiple Partitions**

In this strategy, main memory is divided into multiple partitions, with each partition containing a process. This allows multiple processes to reside in main memory simultaneously (multiprogramming).

- Static Multiple Partitions
- Dynamic Multiple Partitions

1. Static Multiple Partitions

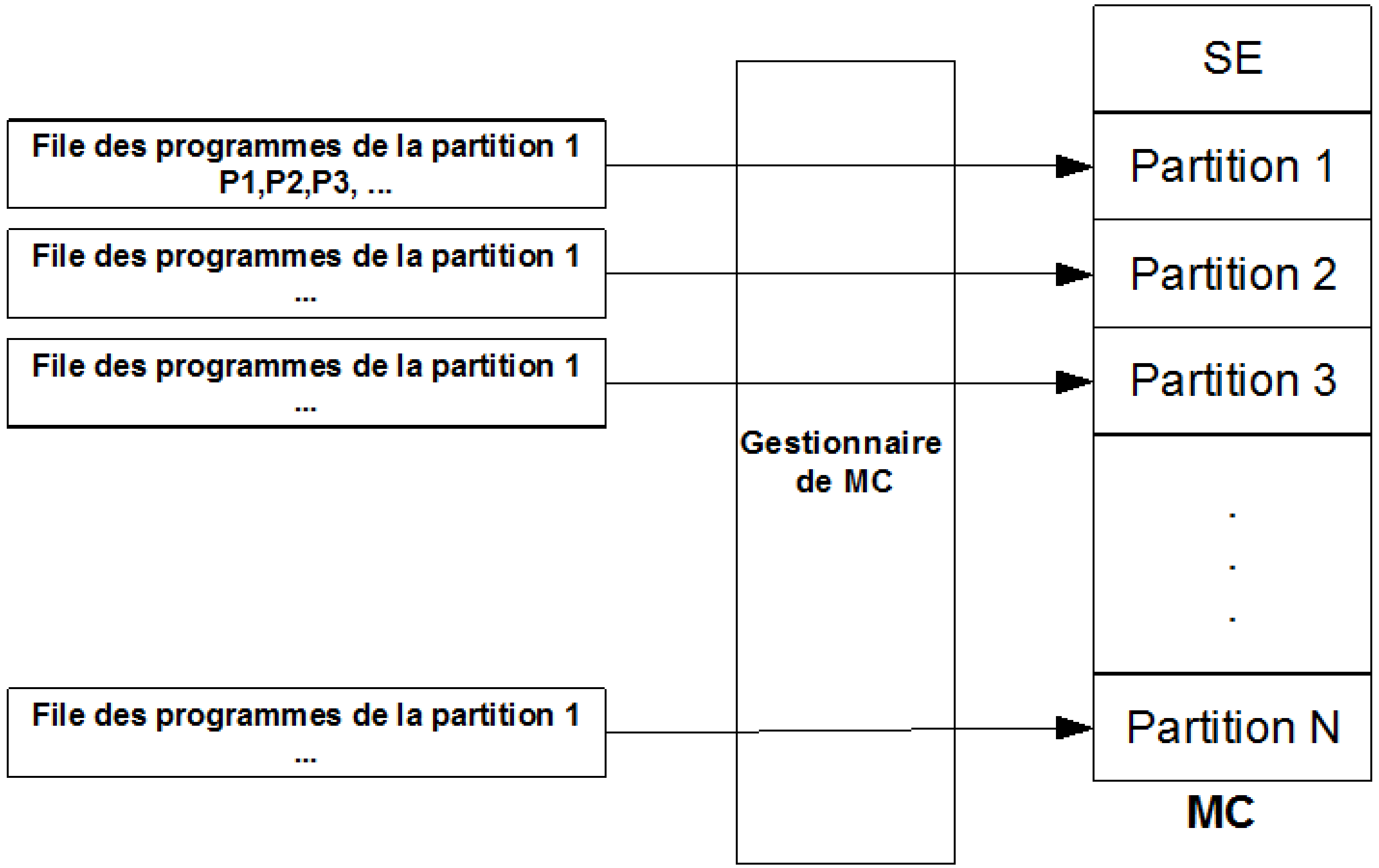
1. Static Multiple Partitions:

In this model, the size and number of partitions are fixed in advance during the operating system startup. To execute a program, there are two possible cases:

1.1 The program to be executed is in absolute code:

In this case, the program's addresses are physical addresses, meaning the program is tied to a specific partition and cannot execute in another partition.

To manage program execution, the memory manager assigns a queue of waiting processes to each partition, as illustrated in the following figure.



File des programmes de la partition 1
P1,P2,P3, ...
File des programmes de la partition 1
...
File des programmes de la partition 1
...
File des programmes de la partition 1
...
Gestionnaire de MC
SE
Partition 1
Partition 2
Partition 3
Partition N
MC

A queue of programs is assigned to each static partition.

Note: The drawback of absolute loading is that some partitions may remain free while programs are queued in another partition.

1. Static Multiple Partitions

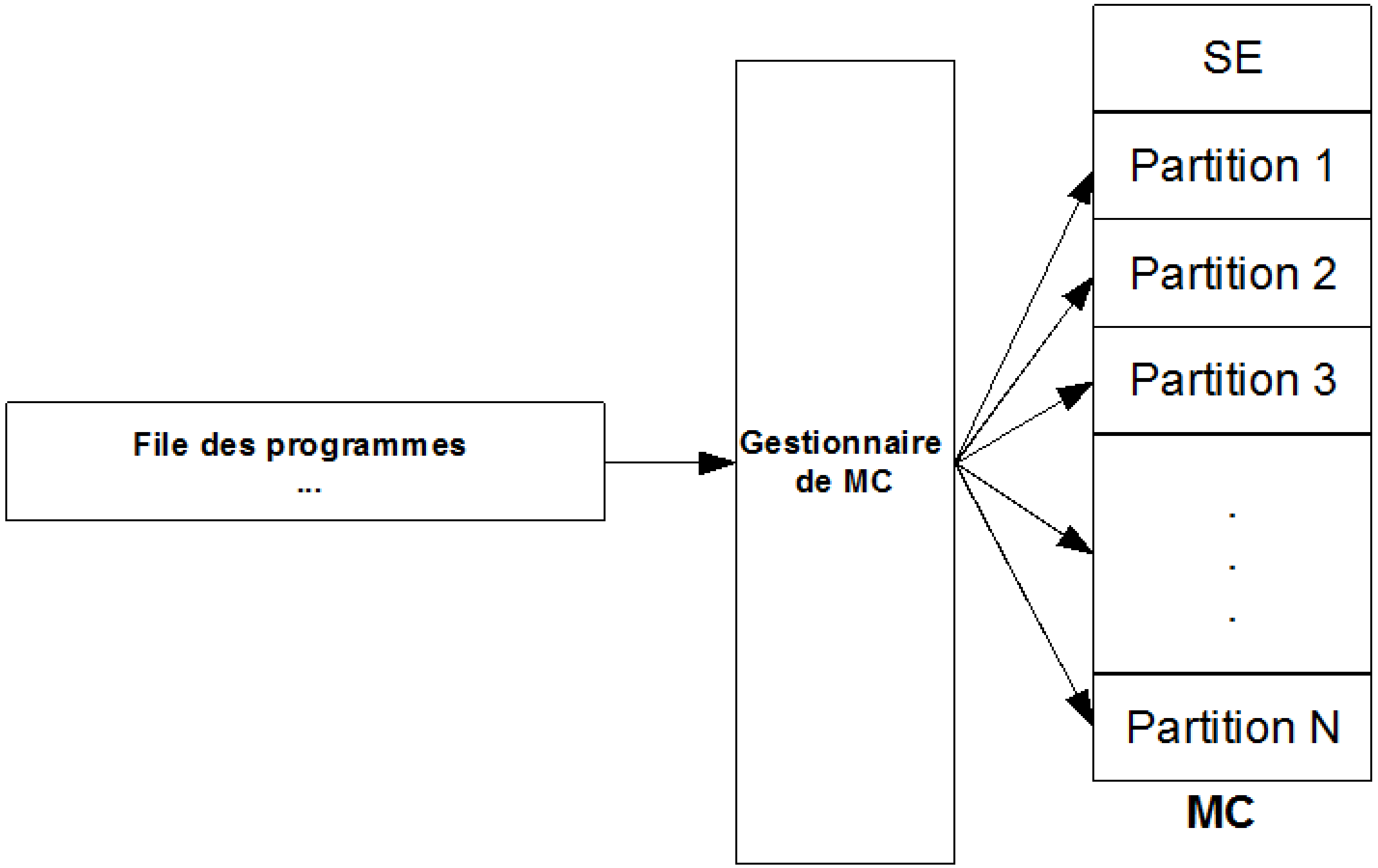
1.2 The program to be executed is in relocatable code:

In this case, the program's addresses are logical addresses (not absolute), allowing the program to be loaded and executed in any partition with sufficient size.

The memory manager assigns a single queue for all processes.

Execution Process:

- The memory manager searches for a free partition with sufficient size to load the program.
- The program is loaded into the selected partition.
- The memory manager converts logical addresses into physical addresses based on the partition used.
- Address conversion: The starting address of the partition is added to all logical addresses in the program to obtain the corresponding physical addresses.



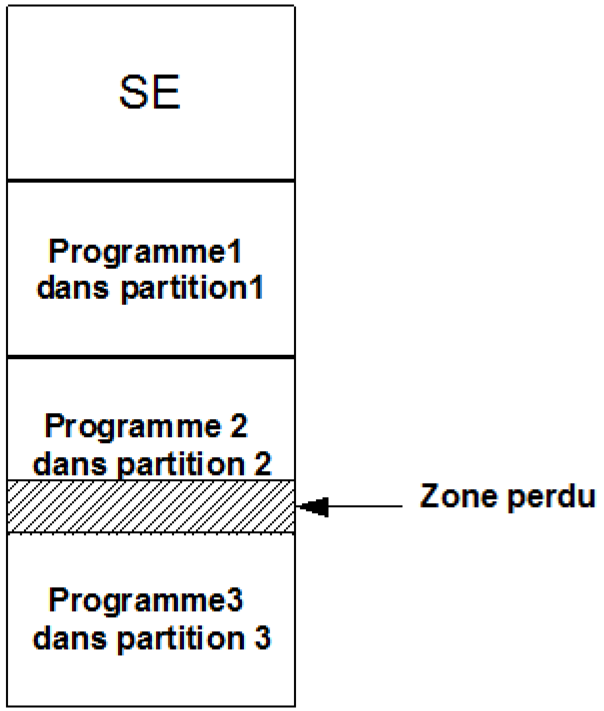
A single queue of programs is assigned to all static partitions.

1. Static Multiple Partitions

- **Fragmentation Problem**

Internal Fragmentation:

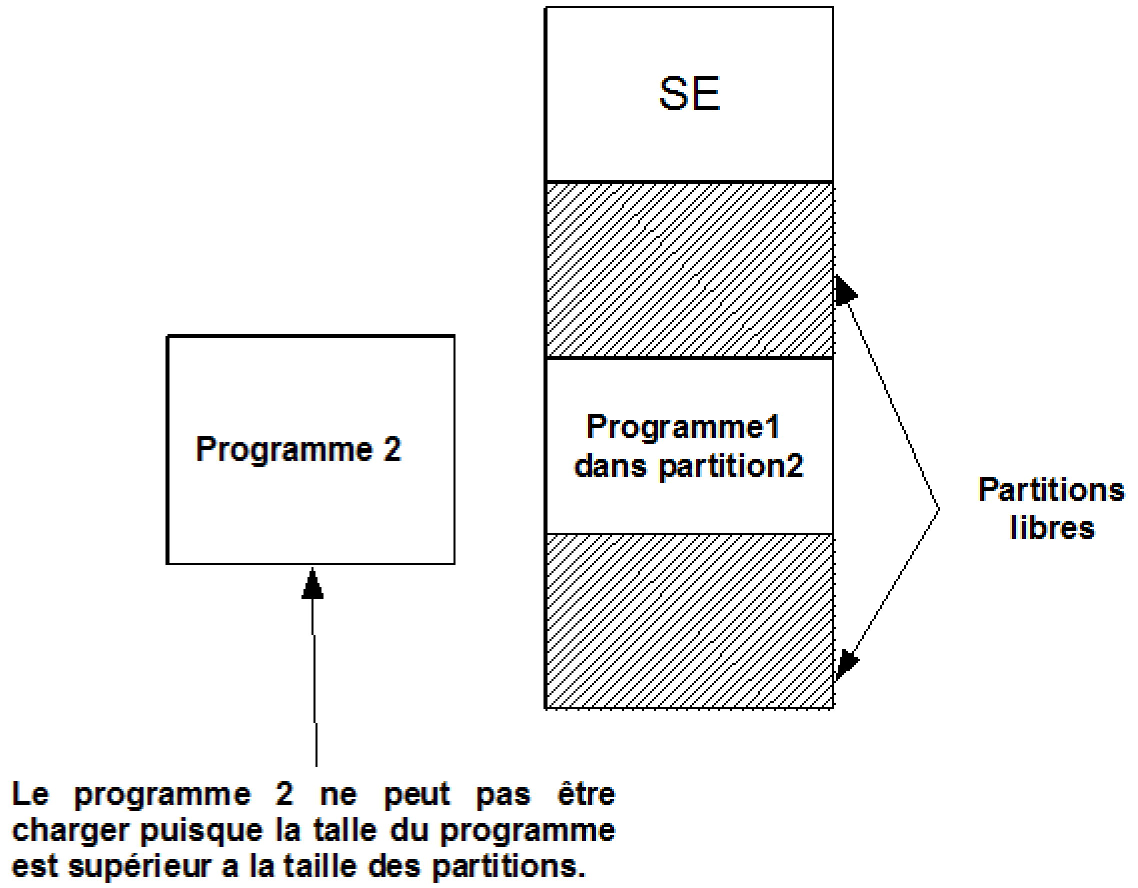
When a program is loaded into a partition and the program size is smaller than the partition size, some memory remains unused. This is known as internal fragmentation.


SE
Programme1 dans partition1
Programme 2 dans partition 2
Zone perdu
Programme3 dans partition 3

1. Static Multiple Partitions

- **Fragmentation Problem**
  External Fragmentation :

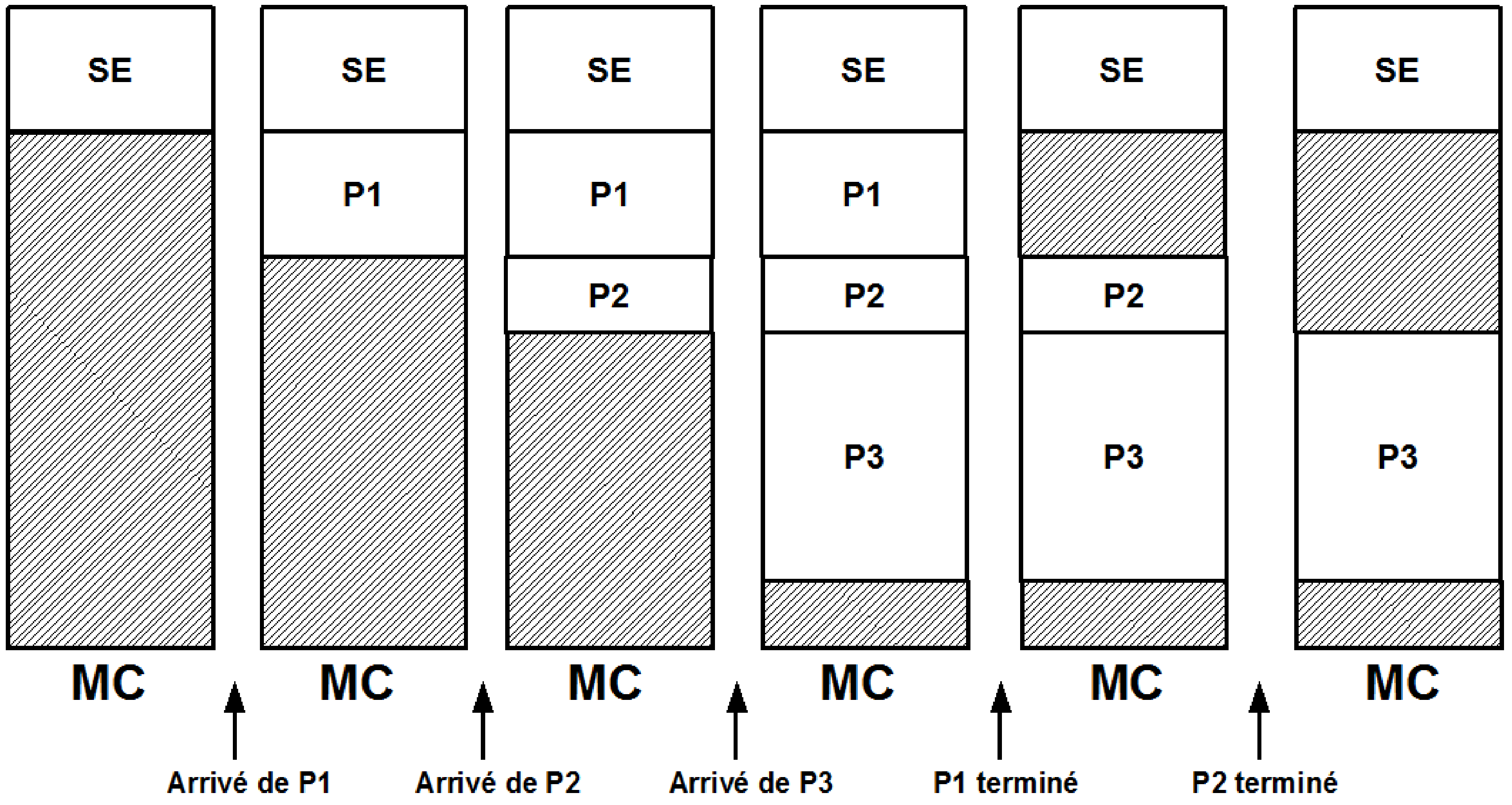
If the size of a program is larger than each of the free partitions but smaller than the sum of all free partitions, then the program cannot be loaded. In this case, we say there is external fragmentation.


SE
Programme 2
Programme1 dans partition2
Partitions libres
Le programme 2 ne peut pas être charger puisque la talle du programme est supérieur a la taille des partitions.

2. Dynamic Multiple Partitions

- The waste of main memory due to fragmentation in static multiple partitioning led to the dynamic multiple partitioning scheme.
- In this scheme, memory is partitioned dynamically on demand, and programs are allocated partitions exactly equal to their sizes.
- When a program finishes execution, its partition is reclaimed to be allocated to another program.
- If a free partition is adjacent to another free partition, the two are merged into a single larger partition.

2. Dynamic Multiple Partitions

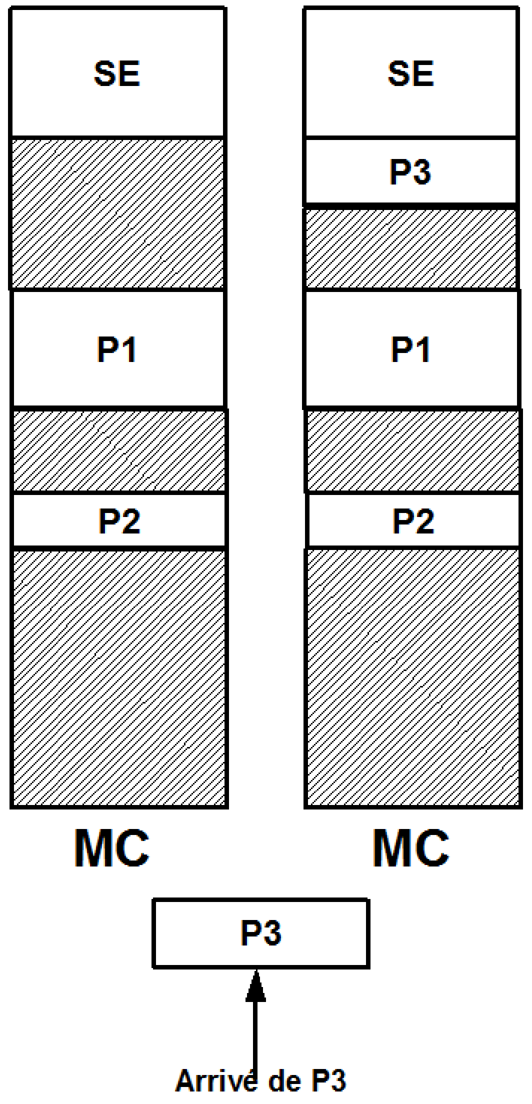


2. Dynamic Multiple Partitions

- Program placement in partitions is done according to different strategies:

First Fit Strategy:

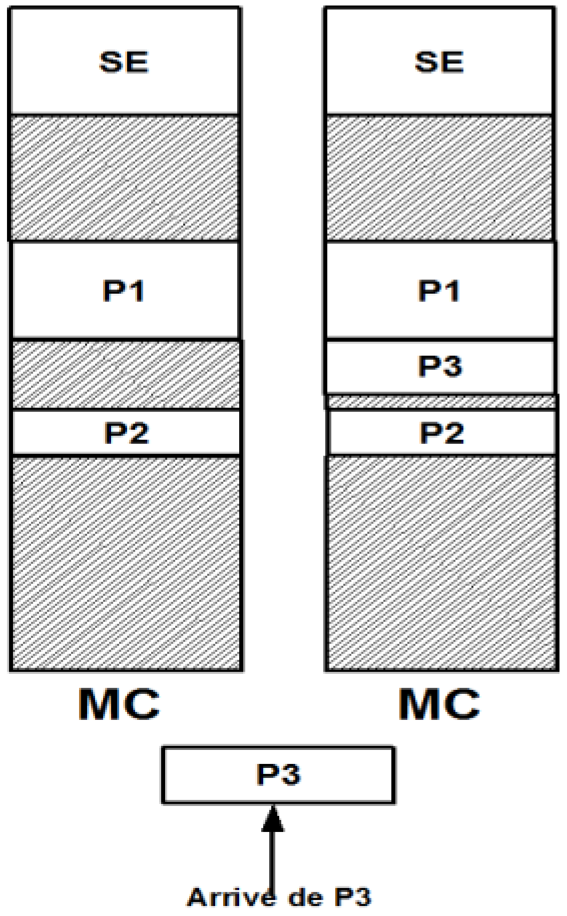
In this strategy, the memory manager places the program in the first available partition that is large enough to accommodate the program.



2. Dynamic Multiple Partitions

Best Fit Strategy:

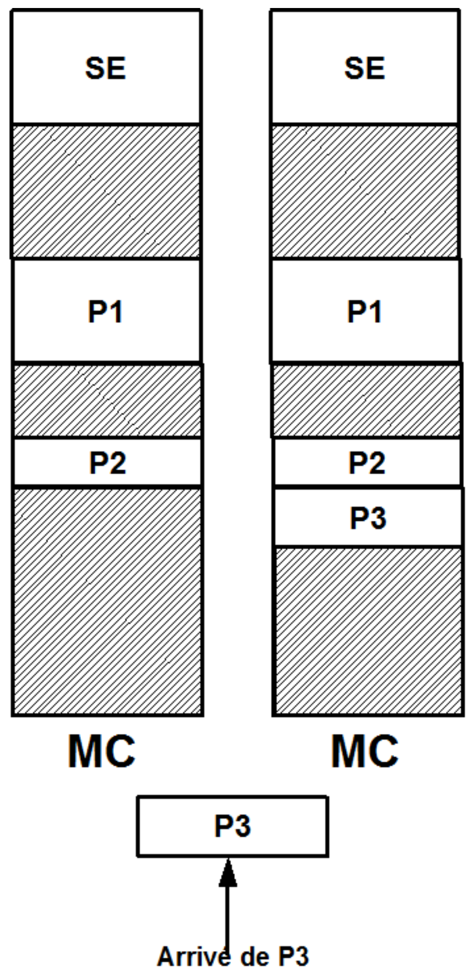
In this strategy, the memory manager places the program in the smallest available partition that is large enough to accommodate the program.



2. Dynamic Multiple Partitions

Worst Fit Strategy:

In this strategy, the memory manager places the program in the largest available partition that is large enough to accommodate the program.



2. Dynamic Multiple Partitions

The First Fit strategy is fast but leads to significant memory loss due to fragmentation.

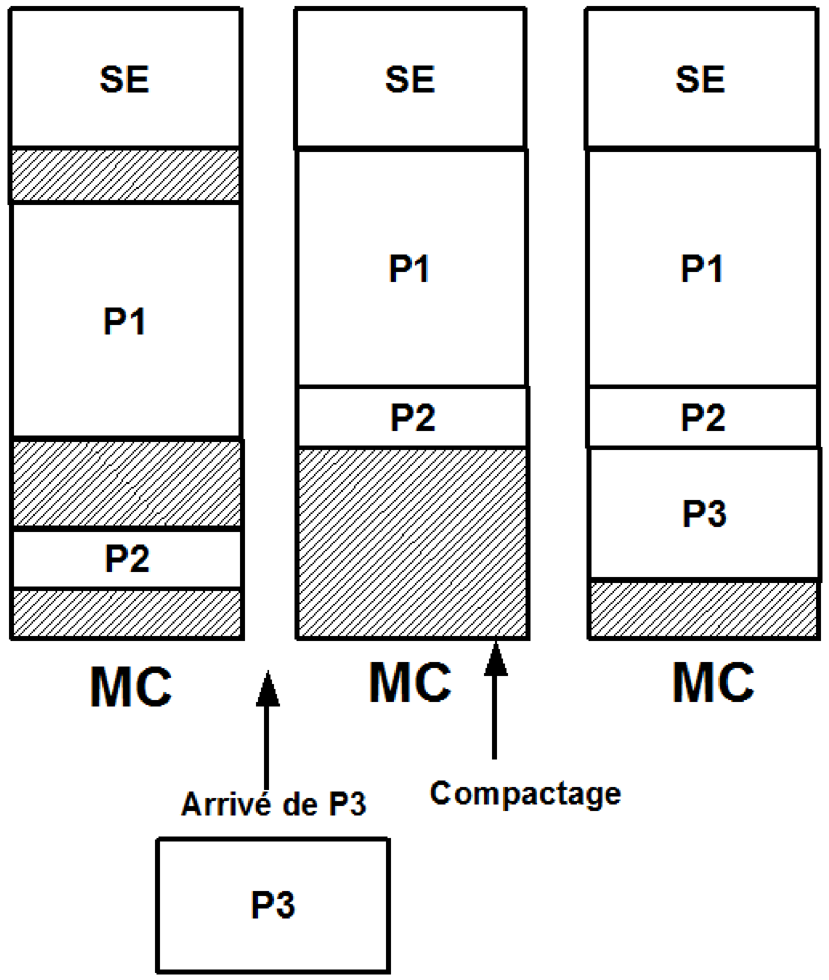
The Best Fit and Worst Fit strategies are slower because they require searching through all available free partitions.

Note: Simulations have shown that the first-fit strategy is better than the best-fit strategy, and both strategies are better than the worst-fit strategy.

2. Dynamic Multiple Partitions

- **Memory Compaction:** Regardless of the allocation strategy used, there is always external fragmentation.
- Compaction involves grouping all unused memory partitions into a single larger partition in which programs can be loaded.

- *Note: The compaction operation is very time-consuming. For example, if we have a machine with 1 GB of RAM that can copy 4 bytes in 20 ns, it will take 5 seconds to compact the entire memory.*



Memory Sharing Modes (Memory Allocation Strategies)

- **Segmentation:**

The program is divided into several segments, each corresponding to a logical entity (main program, procedures and functions, data structures, etc.).

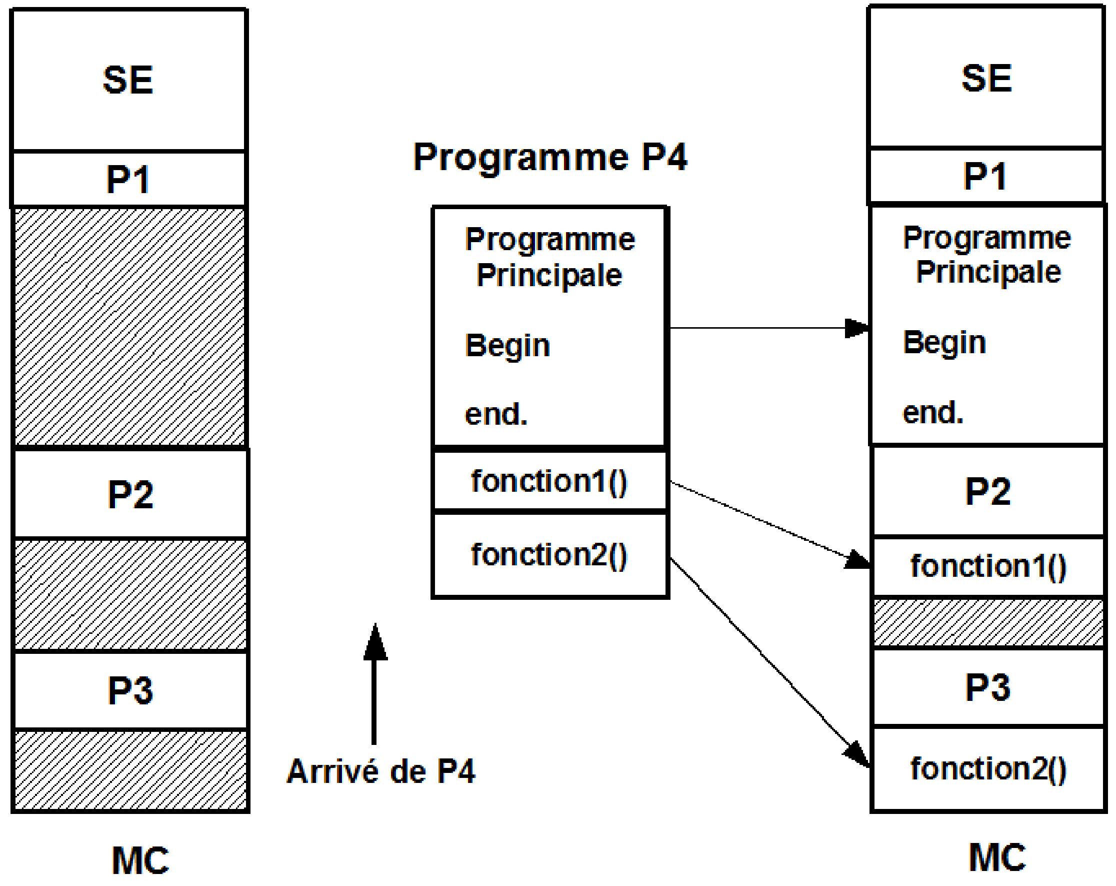
A Pascal compiler, for example, produces different segments for:

➢Global variables,

➢The procedure call stack,

➢The code for each procedure or function,

➢The local variables of each function.



Inside a segment, addresses are relative to the start of the segment (they start from 0 to the size of the segment).

The segment**s** of the same program can be loaded into separate memory spaces, making central memory management more efficient.

Each running program is associated with a segment table, which for each segment of the program provides the base (starting address) and the size of the segment in memory cells (MC).

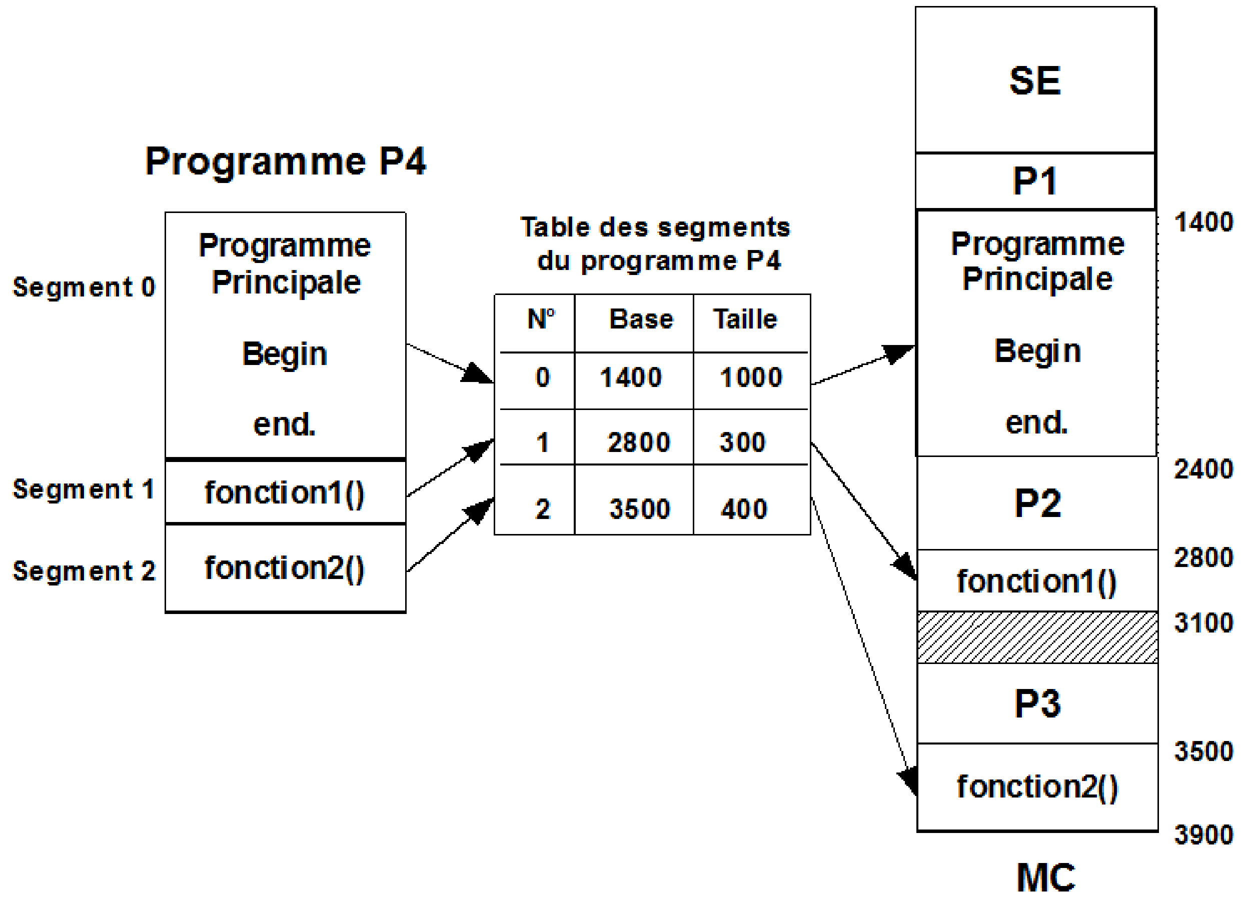
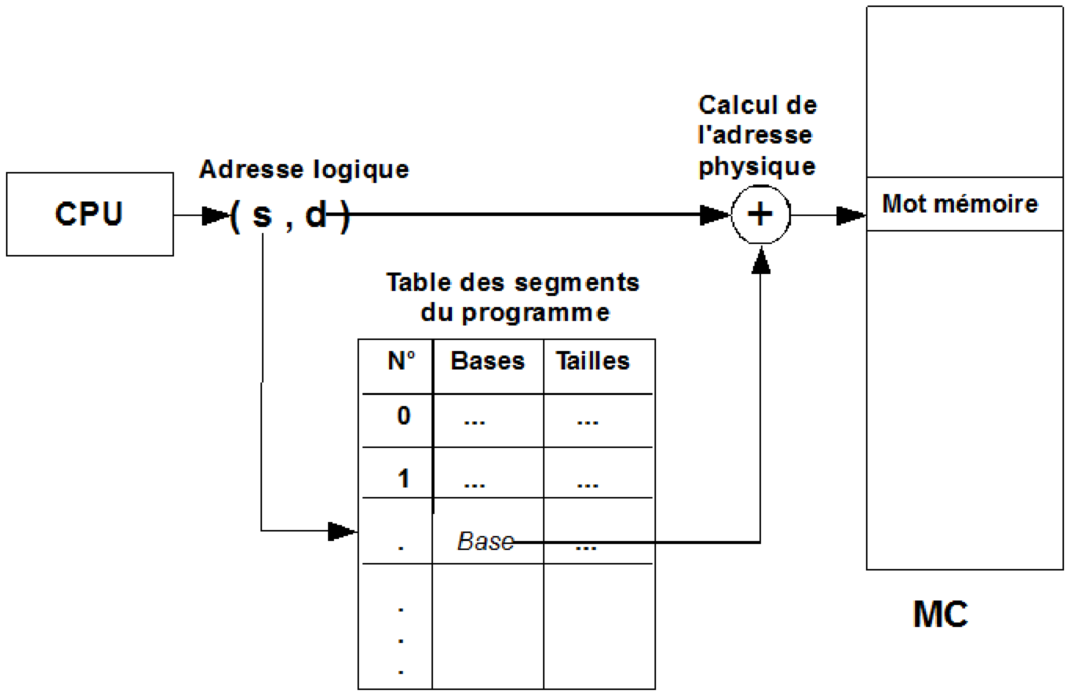


Table des segments du programme P4

N°	Base	Taille
0	1400	1000
1	2800	300
2	3500	400

- The program addresses are **logical addresses**. A logical address specifies the segment and the displacement within the segment.
- The correspondence between the program's logical address and the physical address in main memory is made using the program's segment table as follows:
- The logical address consists of the segment number **s** and the offset (displacement) **d** within the segment. Logical address = **<s, d>**
- The segment number s is used as an index in the segment table to find the base address of the segment in main memory and calculate the physical address.



*Note: The segment **s** must be less than the length of the segment table, and the displacement **d** within the segment must be between **0** and the **size** of the segment. Otherwise, an addressing error will occur.*

Modes de partage de la mémoire (stratégies d'allocation de la mémoire)

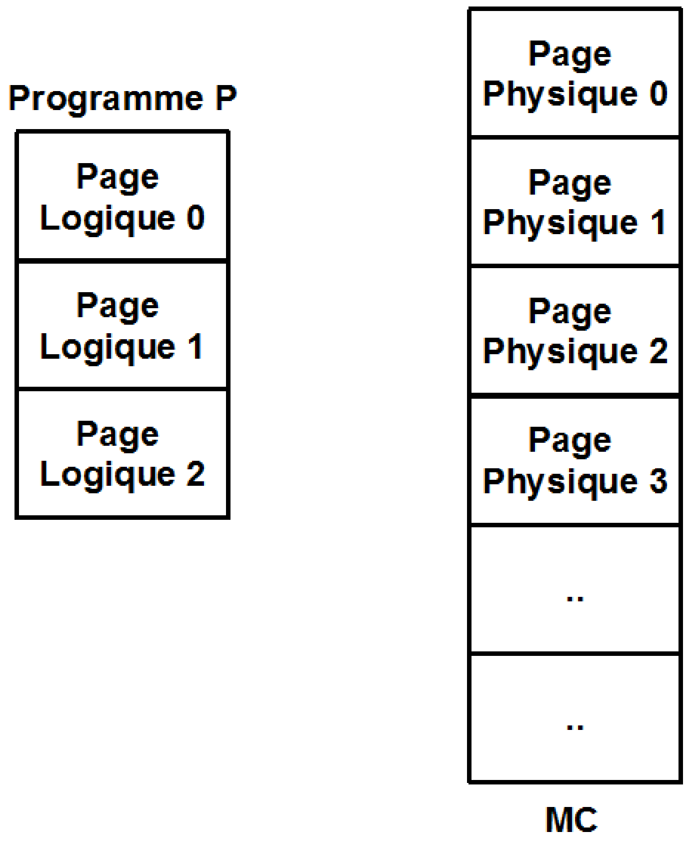
- **Paging**

In paging, main memory (MC) is divided into several partitions of equal and fixed size, called ***physical pages***.

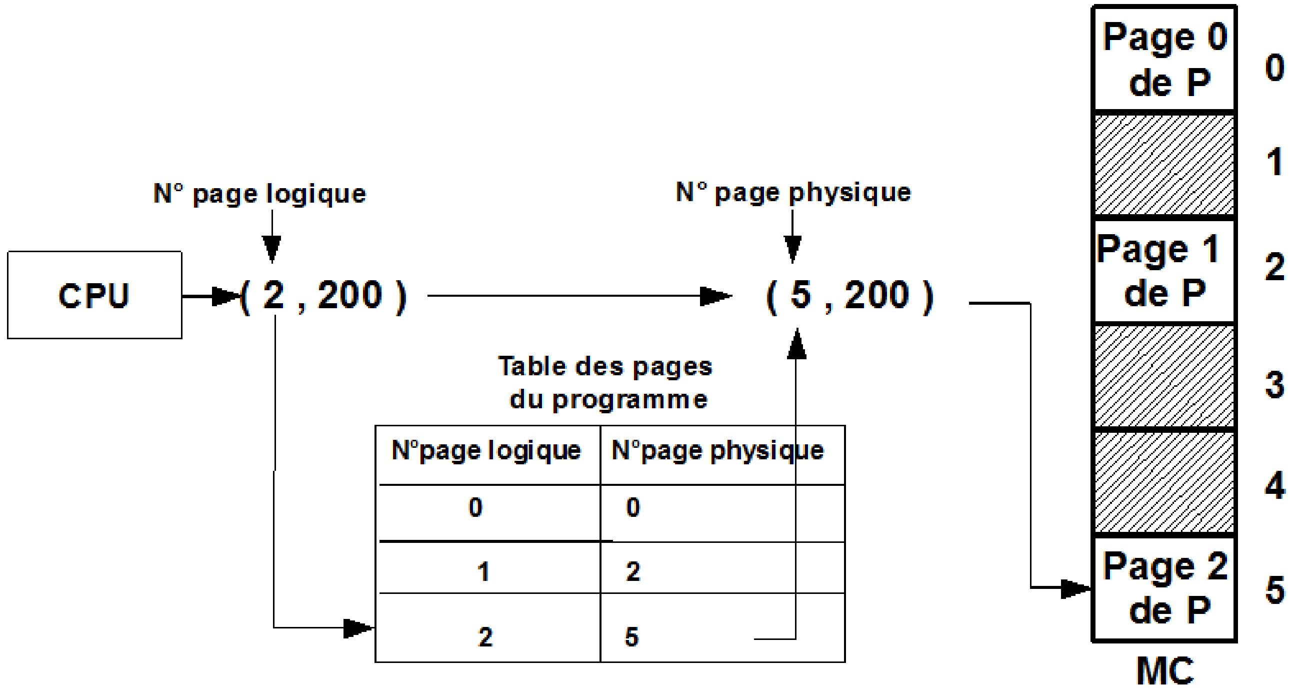
Similarly, the program's address space is divided into several partitions called ***logical pages***.

The size of the logical pages is equal to the size of the physical pages.

A program can be loaded into non-contiguous physical pages, which helps **eliminate** the problem of **external** fragmentation.



- Each running program is associated with a ***page table*** that **maps** logical pages to physical pages in main memory (MC).
- Each address generated by the processor during the execution of a program is divided into two parts: a logical page number **p** and a displacement **d**



Note: *Paging eliminates the problem of external fragmentation, but the issue of internal fragmentation remains present.*

Swapping

Sometimes the size of the main memory is insufficient to load all the programs waiting for execution.

One solution is to temporarily save programs that are blocked (waiting for I/O or an event) in secondary memory (usually the hard disk) and load other programs into the freed partitions.

The operation is carried out as follows:

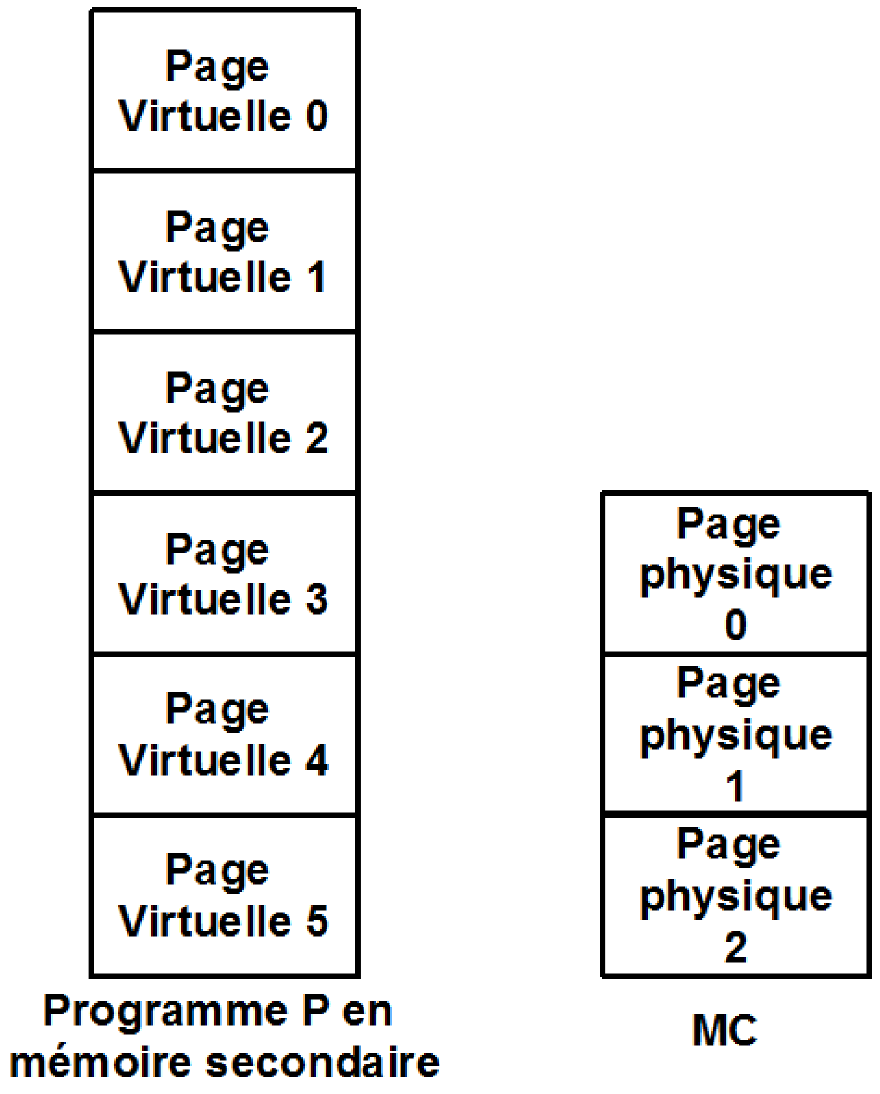
- ➢A program is fully loaded into main memory (Swap-in).
- ➢The program will remain in main memory until its termination or blocking.
- ➢In case of blocking, the program can be transferred to secondary memory (Swap-out).
- ➢One or more new programs are loaded into the freed partition.
- ➢Swapping is very time-consuming due to the transfer time of programs from secondary memory.

Virtual memory

- In the previously discussed modes, if the size of a program is larger than the size of main memory, the program cannot be executed. Virtual memory involves providing the program with a virtual address space that is independent of physical addresses.
- The size of the virtual address space can be much larger than the size of the main memory.
- To execute programs, the system uses secondary memory (usually the hard disk) to store part of the program in execution. The program is therefore loaded in parts and executed in main memory.
- To implement virtual memory, paging or segmentation is typically used.

Paged virtual memory

- In paged virtual memory, both the virtual address space and the physical address space are divided into pages of the same size. The number of virtual pages may be greater than the number of physical pages. Virtual pages are stored in secondary memory and loaded into main memory as needed.



Paged virtual memory

- To translate between virtual address and physical address in main memory (MC), the page table is used.
- The page table contains information about each virtual page of a process. Among this information, we can mention:

- A presence bit that indicates whether the page is present in main memory or not.

0 → the page is not in main memory,

1 → the page is in main memory,

- The location of the page in secondary memory,

- The memory address of the page if it is loaded in main memory,

- A modified bit that indicates whether the page has been modified or not. If the page has been modified, it must be copied back to secondary memory.

Paged virtual memory

- The translation between virtual address and physical address is performed by a specific hardware component integrated into the processor, called the Memory Management Unit (MMU).
- This unit intercepts the virtual address generated by the processor (which accesses data or an instruction) and transforms it into the corresponding physical address.

Paged virtual memory

The execution of a process for translating between virtual address and physical address is carried out as follows:

- The virtual address of an instruction is given by the program counter.
- The virtual address is divided into a page number p and a displacement d.
- The page number p is used as an index to access the page table.
- The system checks if the referenced page p is present in main memory using the presence bit in the page table.
- If the page is not in main memory, a page fault occurs. In this case, the system must load the corresponding page from secondary memory into main memory.
- If the referenced page is in main memory, the physical address is calculated from the physical page number and the displacement d.
- Finally, access to main memory is made.

Page Replacement Strategies

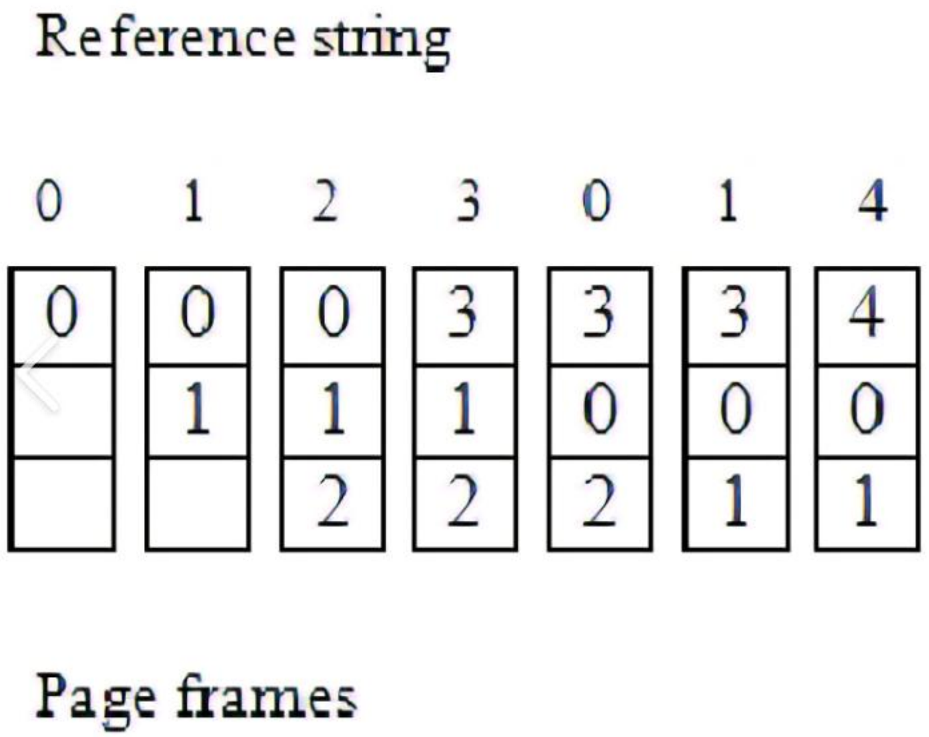
- When a memory reference causes a page fault (when the page is not in main memory), the system must free up memory to load the referenced page.
- The replacement strategy specifies which page should be evicted from main memory to free up space. The chosen page is called the victim page.
- Replacement algorithms include:

- FIFO (First In, First Out)
- LRU (Least Recently Used)
- Optimal Replacement Algorithm
- Second Chance Algorithm
- The Clock Page Replacement Algorithm

Page Replacement Strategies

FIFO Algorithm

- This algorithm chooses to replace the oldest page in main memory.
  Advantage: It is very easy to implement.
  Disadvantage: The oldest page is replaced even if it is frequently used, which can negatively impact performance.



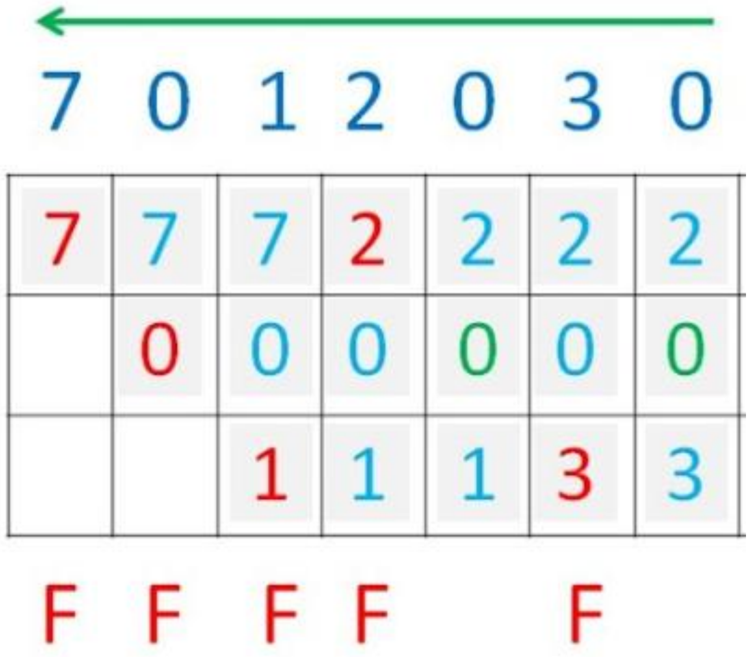
Page Replacement Strategies

LRU Algorithm (Least Recently Used)

- This algorithm selects the page that has been used least recently.

  Advantage: It reduces the number of page faults.

  Disadvantage: It is difficult to implement because it requires tracking and ordering pages based on their usage to determine the least recently used one.



7	0	1	2	0	3	0
7	7	7	2	2	2	2
	0	0	0	0	0	0
		1	1	1	3	3
F	F	F	F		F	

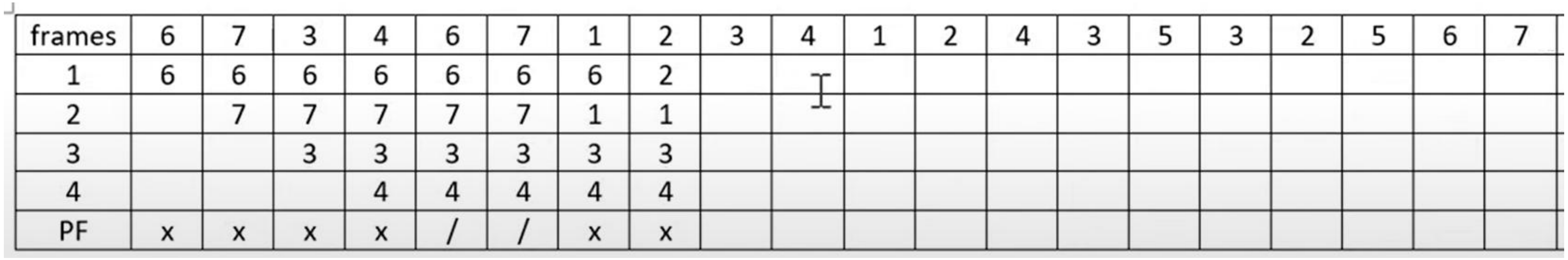
Page Replacement Strategies

- **Optimal Replacement Algorithm**

  In this algorithm, the page that will not be referenced for the longest time in the future is selected for replacement.

  Advantage: It is the optimal algorithm, minimizing page faults as much as possible.

  Disadvantage: It is difficult to implement because predicting future page references is not feasible.

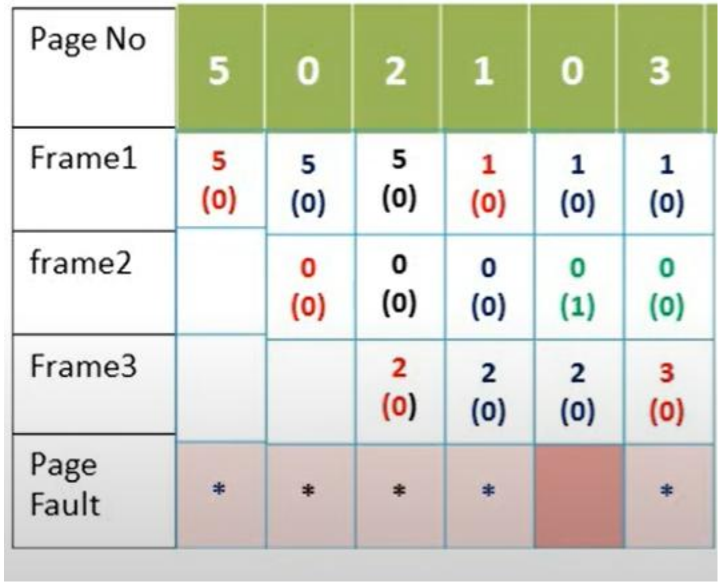


frames	6	7	3	4	6	7	1	2	3	4	1	2	4	3	5	3	2	5	6	7
1	6	6	6	6	6	6	6	2												
2		7	7	7	7	7	1	1												
3			3	3	3	3	3	3												
4				4	4	4	4	4												
PF	x	x	x	x	/	/	x	x												

Page Replacement Strategies

The Second-Chance Page Replacement Algorithm:

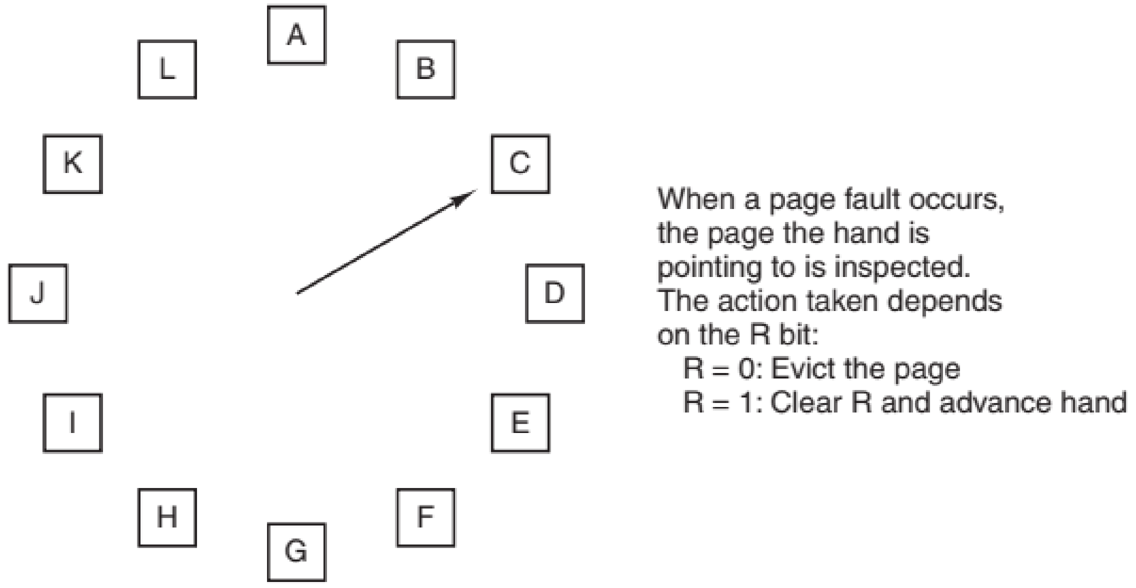
- it is a modification of FIFO that improves efficiency by avoiding the eviction of frequently used pages. It inspects the reference (R) bit of the oldest page in memory.
- If the R bit is 0, the page is both old and unused, so it is replaced immediately.
- If the R bit is 1, the bit is cleared, and the page is moved to the end of the list, as though it had just arrived in memory.
- The search then continues for a page to evict.
- The algorithm ensures that only old, unused pages are evicted. If all pages have been referenced, Second-Chance degenerates into pure FIFO, as it cycles through the pages until it finds one with a cleared R bit to evict. The algorithm always terminates when a page is evicted.



Page No	5	0	2	1	0	3
Frame1	5 (0)	5 (0)	5 (0)	1 (0)	1 (0)	1 (0)
frame2		0 (0)	0 (0)	0 (0)	0 (1)	0 (0)
Frame3			2 (0)	2 (0)	2 (0)	3 (0)
Page Fault	*	*	*	*		*

Page Replacement Strategies

- **The Clock Page Replacement Algorithm** improves upon the second-chance algorithm by avoiding the inefficiency of constantly moving pages around a list.
- Instead, it organizes all page frames in a circular list resembling a clock.
- The hand points to the oldest page.
- When a page fault occurs, the page pointed to by the hand is checked.


A
B
C
D
E
F
G
H
I
J
K
L
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

- If its R bit is 0, the page is evicted and replaced with the new page, and the hand moves one position.
- If the R bit is 1, it is cleared, and the hand advances to the next page.
- This process repeats until a page with R = 0 is found and evicted. The algorithm is named "clock" due to its circular structure.

Memory Protection

In a computer system, it's essential to protect memory blocks from unauthorized access, for example due to programming errors. There are several protection techniques:

- **Case of a Single Contiguous Memory Area**
  In this case, memory is divided into two regions: the operating system area and the user program area.
  The operating system's partition must be protected from access by user programs. This protection is ensured by specific hardware that checks each referenced address against a boundary address.
  The referenced address must be greater than or equal to the boundary address of the operating system area.

Memory Protection

Case of Fixed Multiple Partitions
In this case, two registers are used to indicate the lowest and highest addresses that bound the program. Each referenced address is checked using these two registers.

- **Case of Segmentation and Paging**
  In this case, bits are associated with each page or segment, indicating whether the process is allowed to access those pages or segments.

Code Sharing Between Programs

- Sharing code between programs helps save memory space and program loading time.
- For example, consider a system that supports 40 users, each running a text editor. If the editor's code is 30 KB and each user's data occupies 5 KB, the total memory required without sharing would be: $(30+5)\times 40=1400KB$.
- However, if the editor's code is shared among all users, only one copy of the code is needed. The memory required would then be: $30+(5\times 40)=230KB$.
- To allow a memory region to be shared by multiple processes, this region must not be modified during the execution of the processes (i.e., it must be read-only).

Code Sharing Between Programs

- In the case of paging, to share a set of pages between multiple processes, the pages are loaded into memory only once. Then, an entry for each shared page is added to the page table of every process that needs access.
- In the case of segmentation, to share one or more segments between multiple processes, the segments are also loaded into memory only once. An entry for each shared segment is added to the segment table of every concerned process.