Memory Management

Module 5

Dr. Naveenkumar J

Associate Professor,

PRP- 217 - 4

Memory Management

- The OS function/Service responsible for controlling and coordinating a computer's primary memory (RAM)
- It ensures efficient allocation and deallocation of memory spaces to various processes while they are running, so that each process has the memory it needs without conflicting with others.
- Memory management keeps track of which parts of memory are in use, by which processes, and which parts are free, facilitating optimal utilization of the available memory resources.

Memory Management – Why Necessary?

- To allocate and free memory space to processes before and after their execution.
- To keep track of allocated and free memory locations.
- To minimize fragmentation and ensure proper utilization of the main memory.
- To maintain data integrity and prevent one process from corrupting the memory of another.
- To enable multitasking, allowing multiple processes to reside in memory and run concurrently.

Memory (RAM) – How it is organized

- Primary memory or RAM (Random Access Memory) is organized into a large array of storage cells, each capable of holding a fixed number of bits (usually 8 bits or 1 byte).
- These cells are grouped into words (e.g., 16, 32, or 64 bits) for storage and retrieval.
  - RAM is essentially an addressable memory array with a series of rows and columns
  - Each memory cell has a unique address that the CPU or memory controller uses for read/write operations.
  - Memory addressing uses binary numbers where the number of address lines determines the total addressable memory size (e.g., 16 address lines allow addressing 2^16 memory locations).
  - Internally, RAM does not distinguish between code or data; it only sees bits stored at particular addresses.
  - The basic operations are reading (retrieving bits from a given address) and writing (storing bits at a given address).

Memory (RAM) – How it is organized

- The OS sees RAM as a large pool of continuous addressable memory.
- It manages memory by dividing RAM into fixed or variable-sized partitions allocated to processes.
- The OS treats RAM using abstractions like logical/virtual addresses which it maps to physical addresses in RAM via memory management units.
- It uses techniques like paging and segmentation to provide efficient, protected, and sometimes non-contiguous use of RAM.

Memory (RAM) – Hardware Protection

- Memory hardware protection is a mechanism used by operating systems and hardware to prevent a process from accessing memory segments that are outside its allocated range.
- This protects the system from bugs or malicious code that could corrupt or access other processes' memory or the OS kernel's memory.

Memory (RAM) – Hardware Protection - Working

- The system uses special hardware registers, primarily the Base Register and Limit Register, to enforce protection:
  - Base Register (Relocation Register): Holds the starting physical address of the memory block allocated to a process. *The base address tells where the process begins in memory.*
  - Limit Register: Holds the size (or range) of the allocated memory block. The value in the limit register is typically the length of the addressable memory region for that process, starting from the base address. *The limit tells how much memory (in terms of size) the process can access*, thus defining the upper boundary of accessible memory as **base + limit**

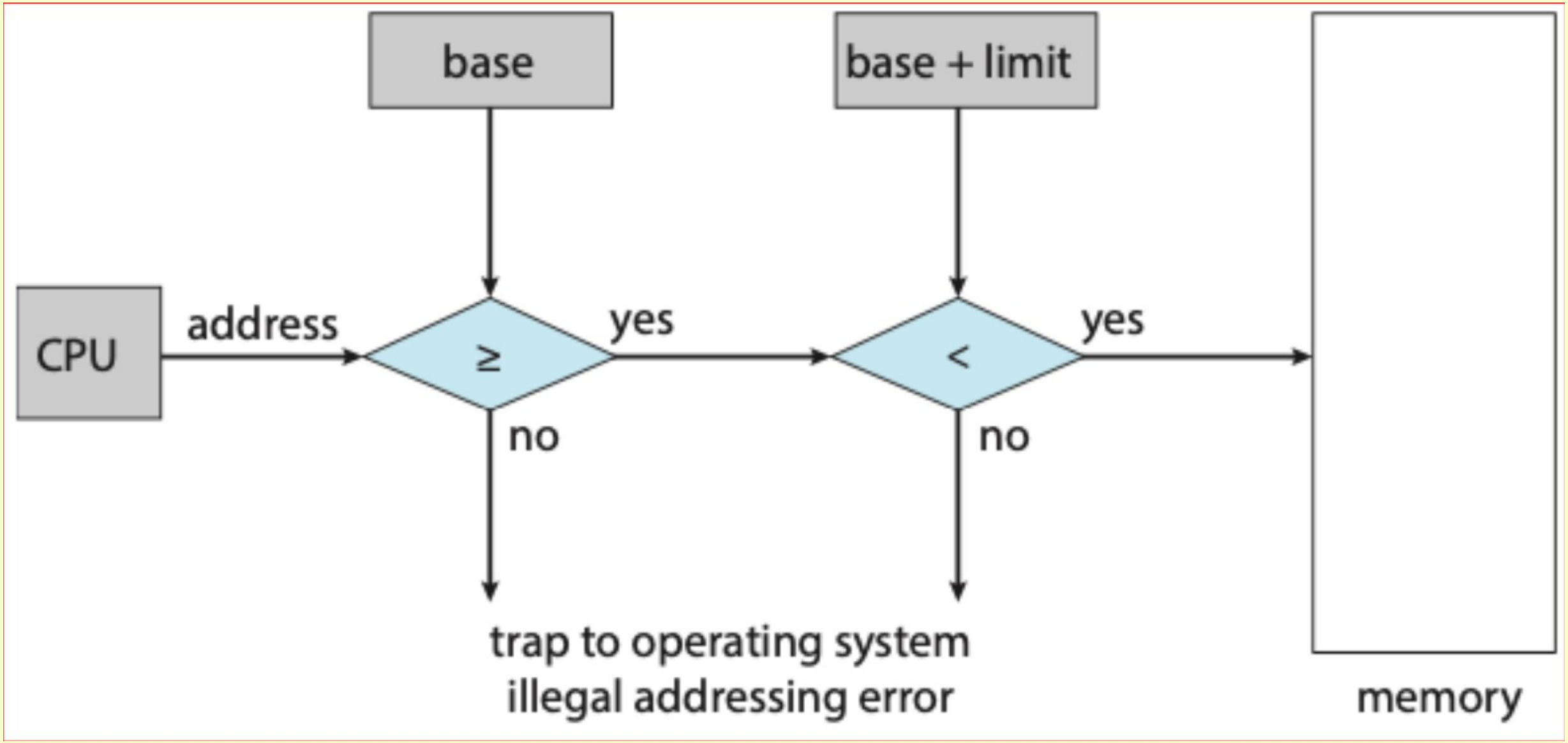
Memory (RAM) – Hardware Protection - Working

- When CPU generates a logical address during Process execution, it is first checked against the limit register.
- The logical address must be less than the limit value; if the logical address is equal to or exceeds the limit, it indicates an invalid memory access, and the hardware raises a protection fault (such as segmentation or memory violation error).
- If the address is valid (less than the limit), the contents of the base register are added to the logical address to form the physical address.

Physical Address in RAM = Value in base Register + logical address

- This physical address is then used by the memory to read or write data.

Memory (RAM) – Hardware Protection - Working



Memory (RAM) – Hardware Protection - Working

- The base register serves as a relocation register marking where in physical memory the process's memory segment begins.
- The limit register defines the maximum valid offset for logical addresses that the process can use.
- The protection check happens before the address translation (logical to physical).
- This hardware-based mechanism keeps processes from accessing memory outside their assigned range, ensuring memory protection and isolation.

Memory (RAM) – Address Binding

- Address Binding in an operating system is the process of *mapping* the *addresses used in a program* (logical or symbolic addresses) *to actual physical memory addresses* where the program's instructions and data reside.
- This mapping can happen at different stages depending on when the final physical location is decided.
  - Compile-Time Address Binding
  - Load-Time Address Binding
  - Execution-Time (Run-Time) Address Binding

Memory (RAM) – Address Binding

- **Compile-Time Address Binding**
  - The binding of addresses is done **when the program is compiled**.
  - The *compiler translates symbolic addresses into absolute physical addresses*.
  - This means the program can only run at a fixed memory location.
  - Example: If the compiler decides the program starts at memory location 1000, all addresses are fixed relative to that location. If the program needs to be moved, it must be recompiled.
  - Good for systems where memory layout is fixed and known, but inflexible.

Memory (RAM) – Address Binding

- **Load-Time Address Binding**
  - Address binding happens when the program is loaded into memory before execution starts.
  - The compiler generates relocatable addresses (relative addresses), not absolute.
  - The loader (part of OS) assigns the starting physical address and updates addresses accordingly.
  - Example: If the program is loaded starting at physical address 5000, all logical addresses are adjusted relative to 5000 at load time. The program can run anywhere in memory without recompilation.
  - Flexible and useful if the program's starting location in memory can vary but does not move after loading.

Memory (RAM) – Address Binding

- **Execution-Time (Run-Time) Address Binding**
  - Address binding happens during program execution dynamically.
  - The program's addresses are logical until resolved by hardware (Memory Management Unit) during instruction execution.
  - This allows the process to move or have memory allocated dynamically, such as in virtual memory systems.
  - Example: The program uses logical addresses; the OS/hardware translates these into physical addresses on the fly.
  - The program memory can be relocated or paged without the program being aware.
  - Most modern OSes use this for flexibility and memory protection.

Memory (RAM) – Address Binding

Binding Type	When Address Binding Happens	Address Type	Flexibility	Example Usage
Compile-Time	During compilation	Absolute	No (fixed, must recompile to move)	Embedded systems, simple OS
Load-Time	During loading before execution	Relocatable	Yes (program can load anywhere)	Batch processing systems
Execution-Time	During program execution	Logical (virtual) addresses	Yes (dynamic relocation)	Modern multitasking OSes, virtual memory

Memory (RAM) – Address Binding

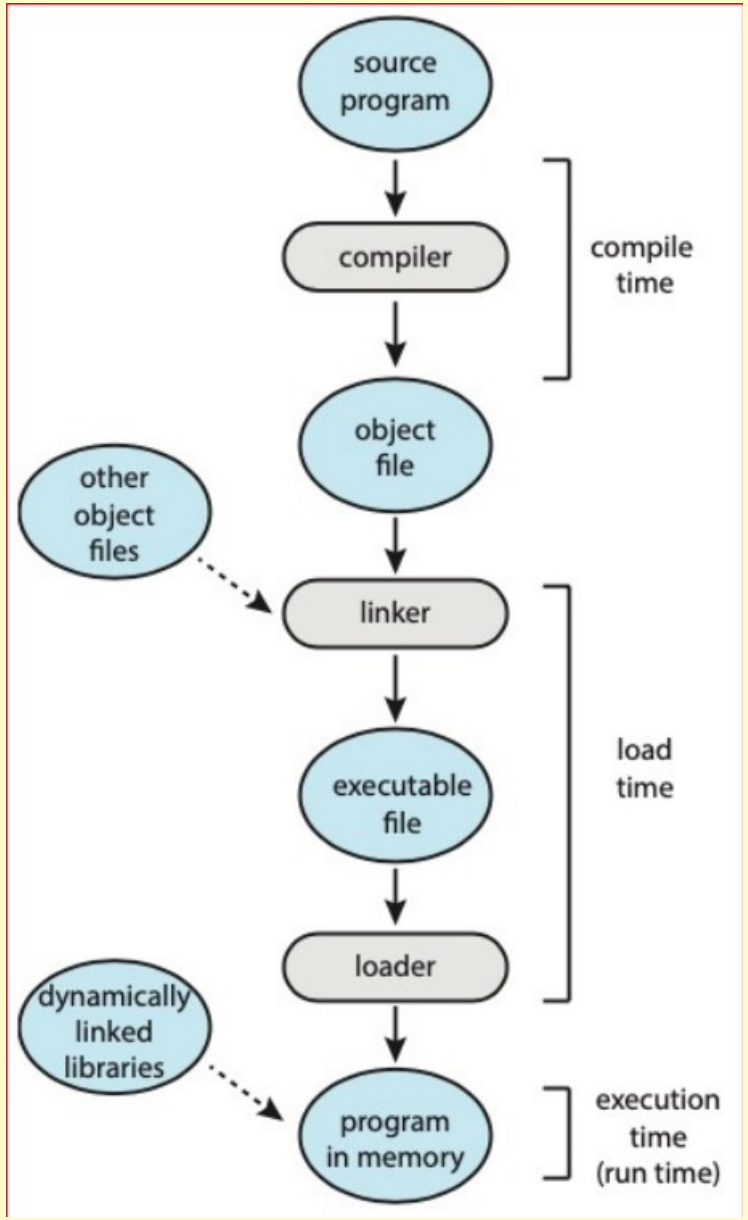
- Suppose a program uses an array: int arr[3] = {5, 10, 15};

Compile-Time Binding	Load-Time Binding	Execution-Time Binding

• The compiler decides absolute addresses for arr, arr, and arr.
 	
For example:
• arr at physical address 1000,
• arr at 1004,
• arr at 1008 (assuming 4 bytes per int).
 	
• These addresses are fixed in the generated code.
 	
• The program must load at address 1000 exactly, or it won't run properly.
	• During compilation, the compiler generates relative addresses for array elements like arr offset 0, arr offset 4, arr offset 8.
	
	• When loaded by the OS, the program's base physical address is assigned, e.g., 5000.
	
	• The loader adds this base address to each offset:
	  • arr physical address = 5000 + 0 = 5000,
	  • arr = 5000 + 4 = 5004,
	  • arr = 5000 + 8 = 5008.
	
	• The program can now run anywhere in memory as the loader adjusts addresses accordingly.
		• The program uses logical addresses (offsets) as above.
		
		• The OS and hardware memory management unit translate logical addresses (e.g., offset 0, 4, 8) into physical memory dynamically during execution.
		
		Suppose physical frame 7000 is assigned, then:
		  • arr → physical address 7000,
		  • arr → 7004,
		  • arr → 7008,
		  • all resolved at runtime.
		
		• This allows the program to move or be swapped in memory without affecting execution.

Memory (RAM) – Address Binding Done when

- The linker combines object files and libraries to create an executable file, but addresses inside the executable are typically relocatable.
- Load-time binding happens when the loader assigns the executable a base address in memory and adjusts all addresses based on this starting point.
- The program can occupy different memory locations at each run, as addresses are fixed only upon loading.



- The compiler translates the source program into an object file by assigning symbolic or, in some cases, absolute addresses to variables and instructions.
- Compile-time binding occurs here if the compiler sets all final memory addresses. The program must then always load at the same spot in memory.
- The loader places the program in memory along with any dynamically linked libraries.
- Execution-time binding refers to computing physical addresses at runtime.
- The program uses logical or virtual addresses, and these are mapped to physical memory as the instructions execute—allowing advanced features like address space relocation and virtual memory.

Memory (RAM) – Virtual/Logical Address Space

- This is the set of all addresses that a process or program can use as references to memory locations.
- These addresses are generated by the CPU during program execution but do not represent actual physical memory locations.
- They create an abstraction, allowing each process to think it has its own contiguous memory, independent of other processes or the real layout of physical memory.

Memory (RAM) – Virtual/Logical Address

- An address generated by the CPU while a program is running.
- It is an abstract address that points to a location in the virtual address space for that process.
- Logical addresses are translated to physical addresses by the Memory Management Unit (MMU) before actual access to memory occurs.

Memory (RAM) – Physical Address Space

- This is the actual set of physical memory addresses in the RAM hardware.
- It represents all real locations where data and instructions reside physically.

Memory (RAM) – Physical Address

- The real address seen by the memory hardware.
- It's where data actually lives in RAM.
- After translation from logical addresses, physical addresses are used by the memory controller to fetch or store data.

Memory (RAM) – Physical Address

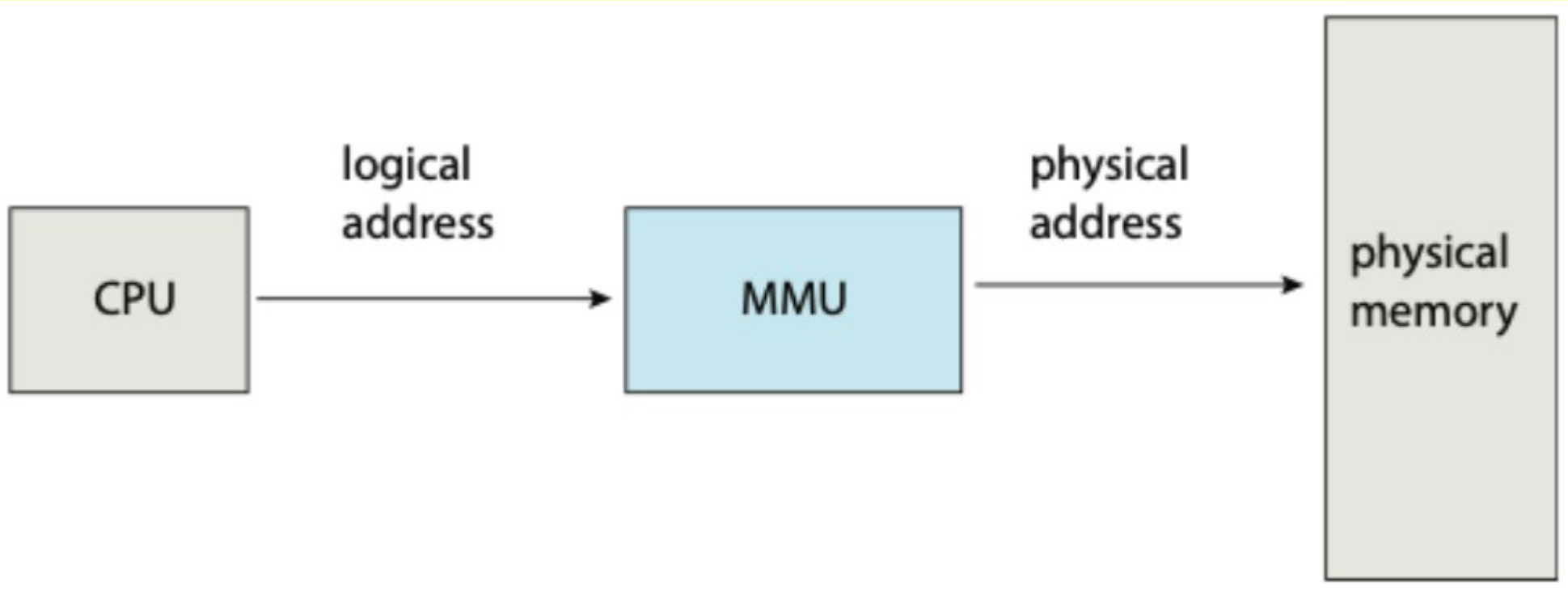
Virtual (Logical) Address	Address generated by CPU, refers to location in virtual memory	Provides abstraction and isolation for processes
Physical Address	Actual memory location in RAM hardware	Real memory access and storage of data
Virtual Address Space	Complete set of all logical addresses a process can use	Creates process-specific memory environment
Physical Address Space	Actual addressable space in physical RAM	Represents real physical memory

Memory (RAM) – Memory Management Unit

- MMU is a critical hardware component in a computer system that acts as the bridge between the CPU and the physical RAM. Its primary roles include:
  - **Translating Virtual (Logical) Addresses to Physical Addresses**: The CPU generates virtual addresses during program execution. The MMU converts these virtual addresses into actual physical addresses in RAM, enabling the CPU to correctly access data.
  - **Memory Protection**: The MMU enforces access control, ensuring a process can only access memory locations it is allowed to, preventing accidental or malicious memory access violations.
  - **Support for Virtual Memory**: By managing address translation and memory access permissions, the MMU supports virtual memory systems that allow programs to use more memory than physically available by swapping pages between RAM and disk storage.
  - **Memory Segmentation and Paging**: MMUs may implement segmentation and paging techniques that divide memory into blocks (segments or pages) for fine-grained memory management and protection.

Memory (RAM) – Memory Management Unit

- The MMU is hardware within the CPU or as a separate chip that takes the address requested by a running program and figures out where that data or instruction actually resides in physical memory. It checks permissions and ensures the system's memory is used securely and efficiently.



Memory (RAM) – Static Loading & Linking

- **Static Loading:**
  - Static loading is the process where the entire program, including all its instructions and library routines, is loaded into the main memory before execution starts.
  - The whole executable is loaded into memory at once.
  - Since everything is loaded beforehand, the program starts running immediately after loading.
  - This method simplifies execution but requires that all code must fit into memory at load time.

- **Static Linking:**
  - Static linking happens at compile-time or before execution.
  - The linker combines all object modules and required library routines into a single executable file.
  - All addresses are resolved and fixed, so the program is self-contained and has no external dependencies at runtime.
  - Every statically linked program has its own complete copy of the library code it uses.

Memory (RAM) – Dynamic Loading & Linking

- Dynamic loading is a technique where a program loads parts of itself (like functions or libraries) into memory only when they are needed during execution, not before.
- At runtime, when a program calls a function or module that is not yet loaded, the OS or program loads the required code from disk into RAM on demand.
  - Saves memory by only loading what is necessary.
  - Speeds up program startup as not everything is loaded initially.
  - Useful for large programs or modular systems (e.g., plugins).

Memory (RAM) – Dynamic Loading & Linking

- Dynamic linking refers to the process of linking external libraries or modules to programs at runtime rather than during compile time.
- Instead of copying library code into each program executable (like static linking), a dynamic linker loads shared libraries (e.g., DLLs or SO files) into memory, resolves references, and binds the libraries to the running program as needed.
  - Saves memory since multiple programs can share a single copy of a library in RAM.
  - Reduces executable size as libraries are not duplicated in each program.
  - Updates to shared libraries apply to all programs immediately without recompilation.

Contiguous Memory

- It refers to a block of memory addresses that are adjacent and sequential.
- When a process or program is allocated memory in a contiguous manner, all the memory cells allocated to it form one continuous chunk without gaps.

- Example: If a process is allocated memory addresses from 0x1000 to 0x1FFF, these addresses are consecutive and form a contiguous memory block.

- In contiguous memory allocation, the whole program or data segment resides as one continuous block in RAM.

Partitioning of Memory Or Memory Allocation

- Partitioning is the technique of dividing the main memory into blocks or partitions to allocate to different processes.
- Partitioning helps in organizing memory so that multiple processes can reside in memory simultaneously without overlapping.

Fixed-size partitioning

The memory is divided into fixed-size chunks or partitions. These partitions are static in size and predetermined.

Variable-size partitioning

Memory is divided dynamically based on the process size at runtime. Partitions vary in size.

Contiguous Memory - Partitioning of Memory

- When memory is partitioned, each process is given one contiguous block or partition of memory.
- The OS manages these partitions, allocating contiguous memory blocks to processes they require.
- Contiguous memory allocation simplifies memory access and address translation because the entire process occupies one continuous physical block.
- Drawbacks include fragmentation (both internal and external) and lack of flexibility when processes grow beyond allocated partitions.

Contiguous Memory – Memory Allocation Strategies

Fixed Partitioning (Static Partitioning)

- RAM is divided into several fixed-size partitions during system startup.
- Each process gets an entire partition. If the process is smaller than the partition, the rest of the space is wasted (internal fragmentation).
- The number and size of partitions are fixed until reboot.
- Simple but can be inefficient if process size varies widely.

Variable Partitioning (Dynamic Partitioning)

- RAM is divided into partitions dynamically based on process requirements.
- Each process gets exactly as much memory as it needs, allocated as a contiguous block.
- As processes are loaded and terminated, “holes” (gaps of free memory) form.
- This can lead to external fragmentation—the memory is free, but not in large enough contiguous blocks for bigger processes.

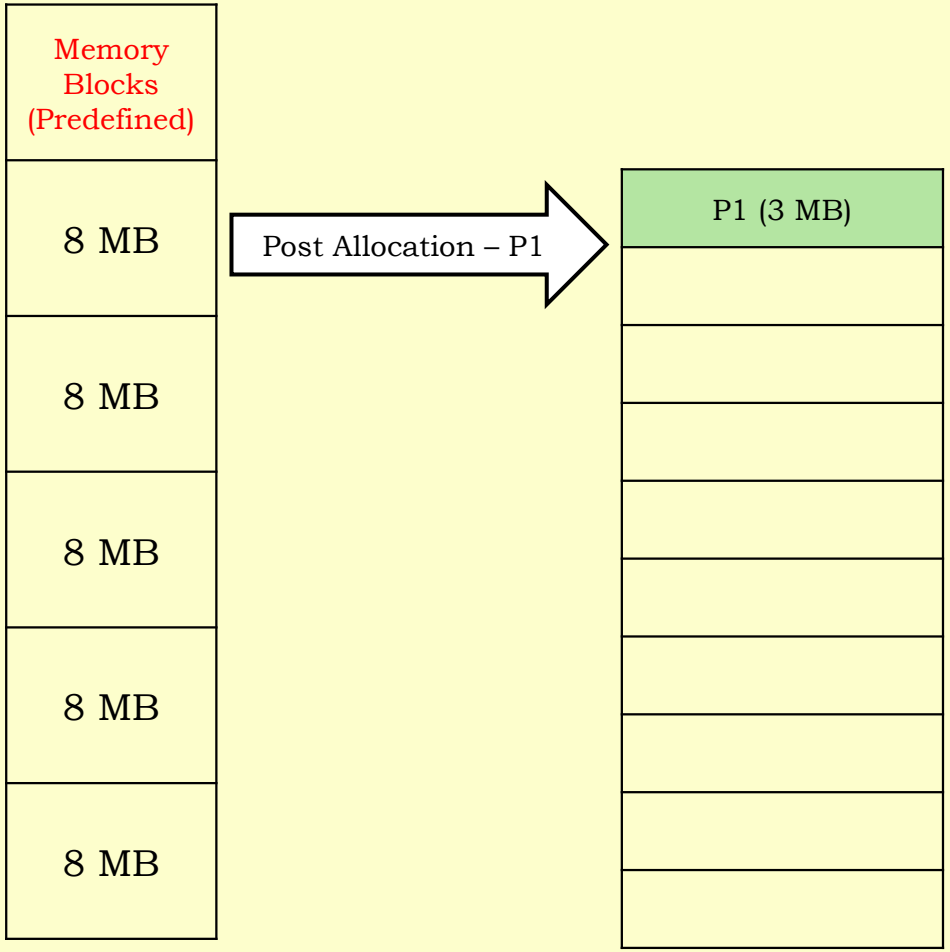
Contiguous Memory Allocation – Static Partitioning

- Fixed partitioning **divides RAM into a fixed number of partitions** at system configuration time, after allocating space for the OS.
  - **Fixed Number of Partitions**: The count is predetermined and unchanging (e.g., 4 partitions).
  - **Partition Sizes:** Can be equal (e.g., all 8MB) or variable (e.g., 4MB, 8MB, 8MB, 16MB) to accommodate different process sizes.
  - **Allocation Rules**: Processes are loaded from secondary memory into RAM partitions. Each process must fit entirely into one contiguous partition (no spanning across partitions). Allocation uses strategies like first-fit, best fit, worst fit and Next Fit.

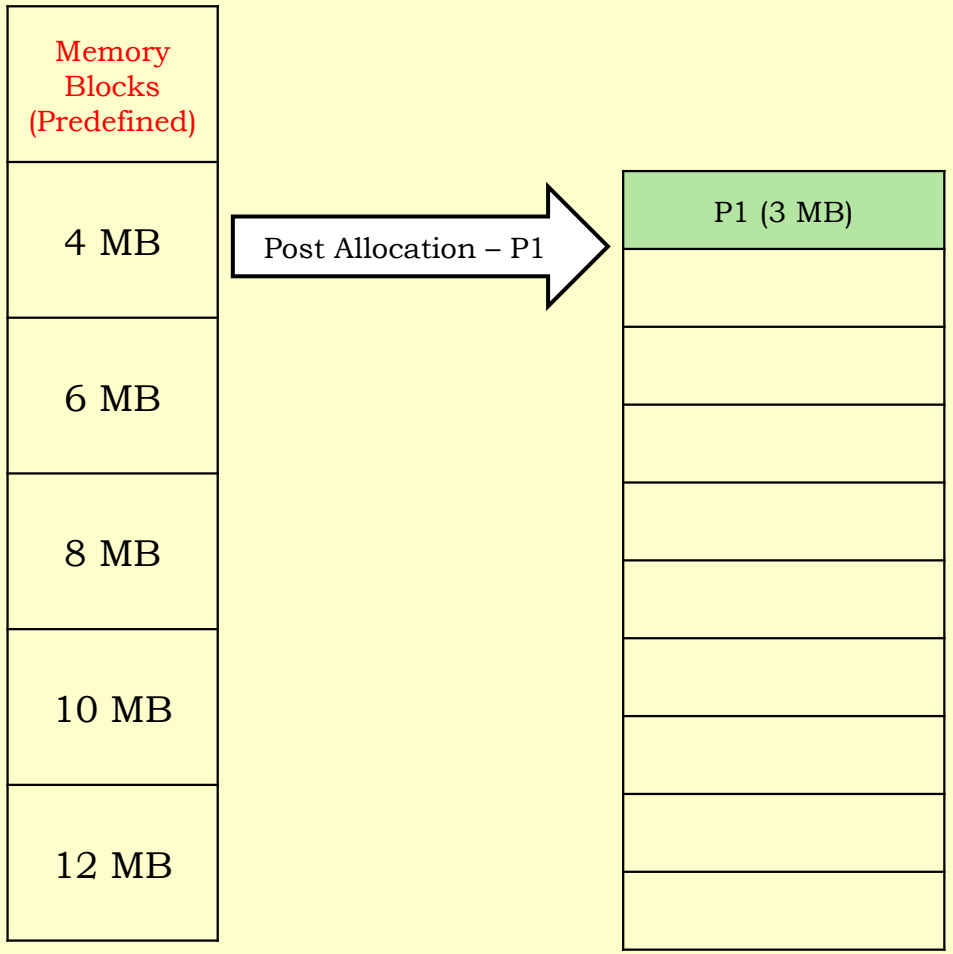
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

Fixed Partitioning with Equal Partition Sizes



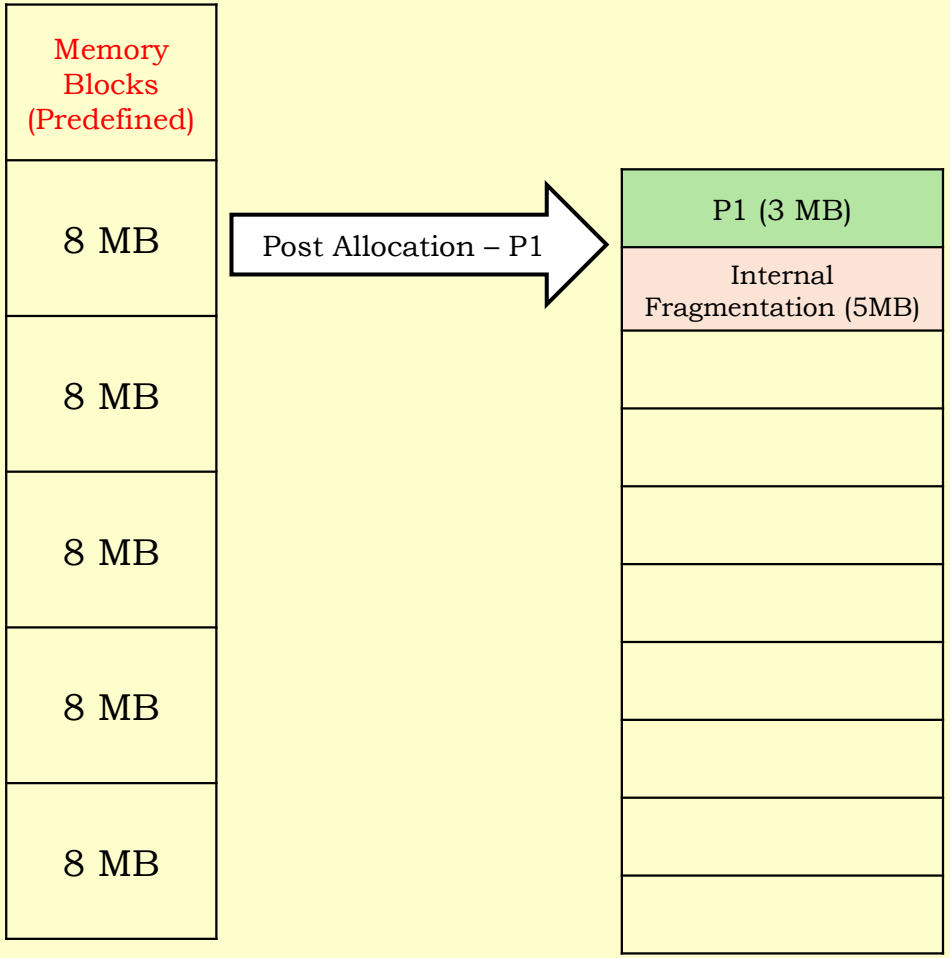
Fixed Partitioning with Variable Partition Sizes



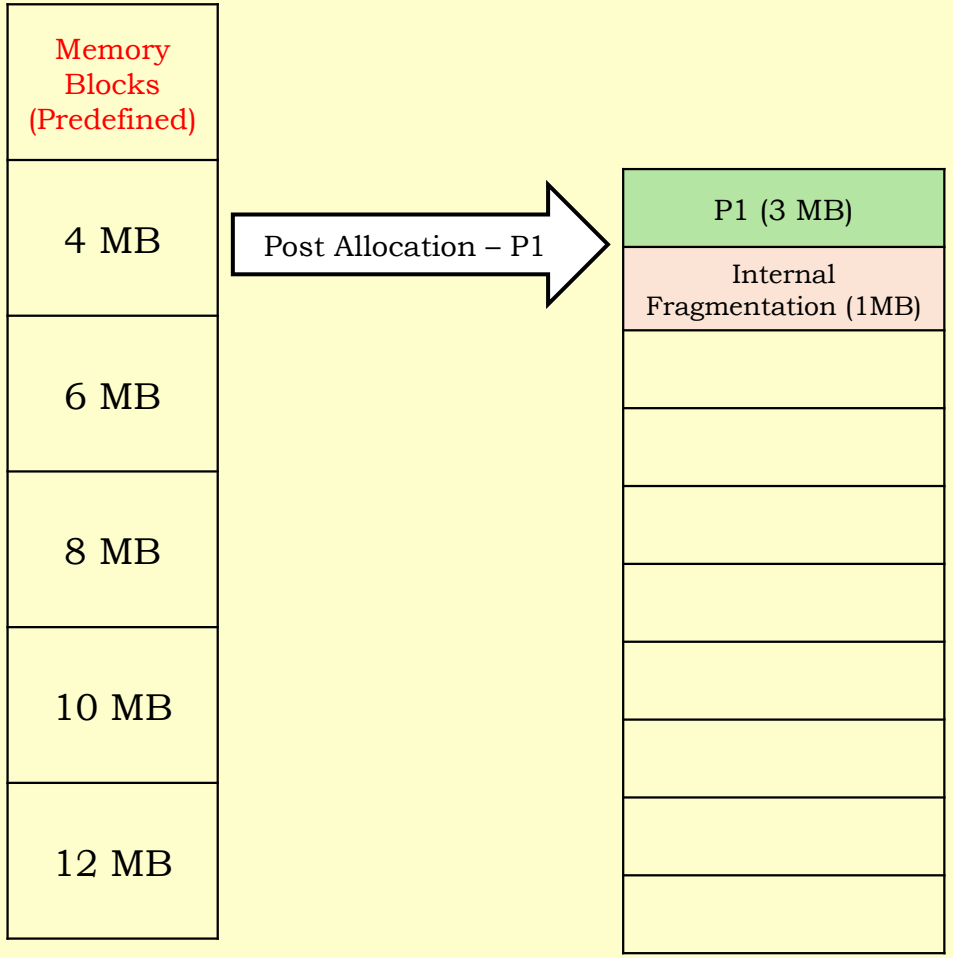
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

Fixed Partitioning with Equal Partition Sizes

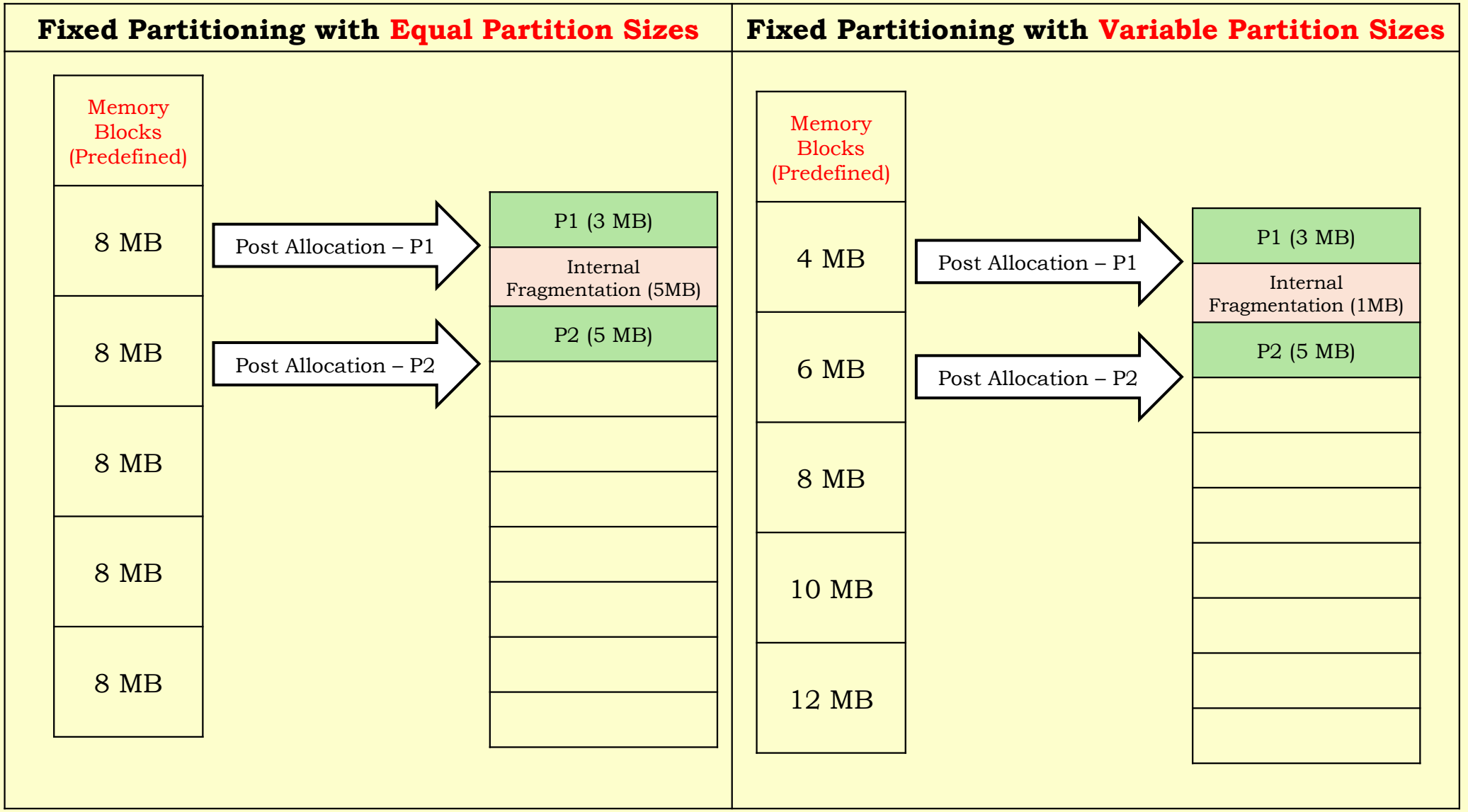


Fixed Partitioning with Variable Partition Sizes



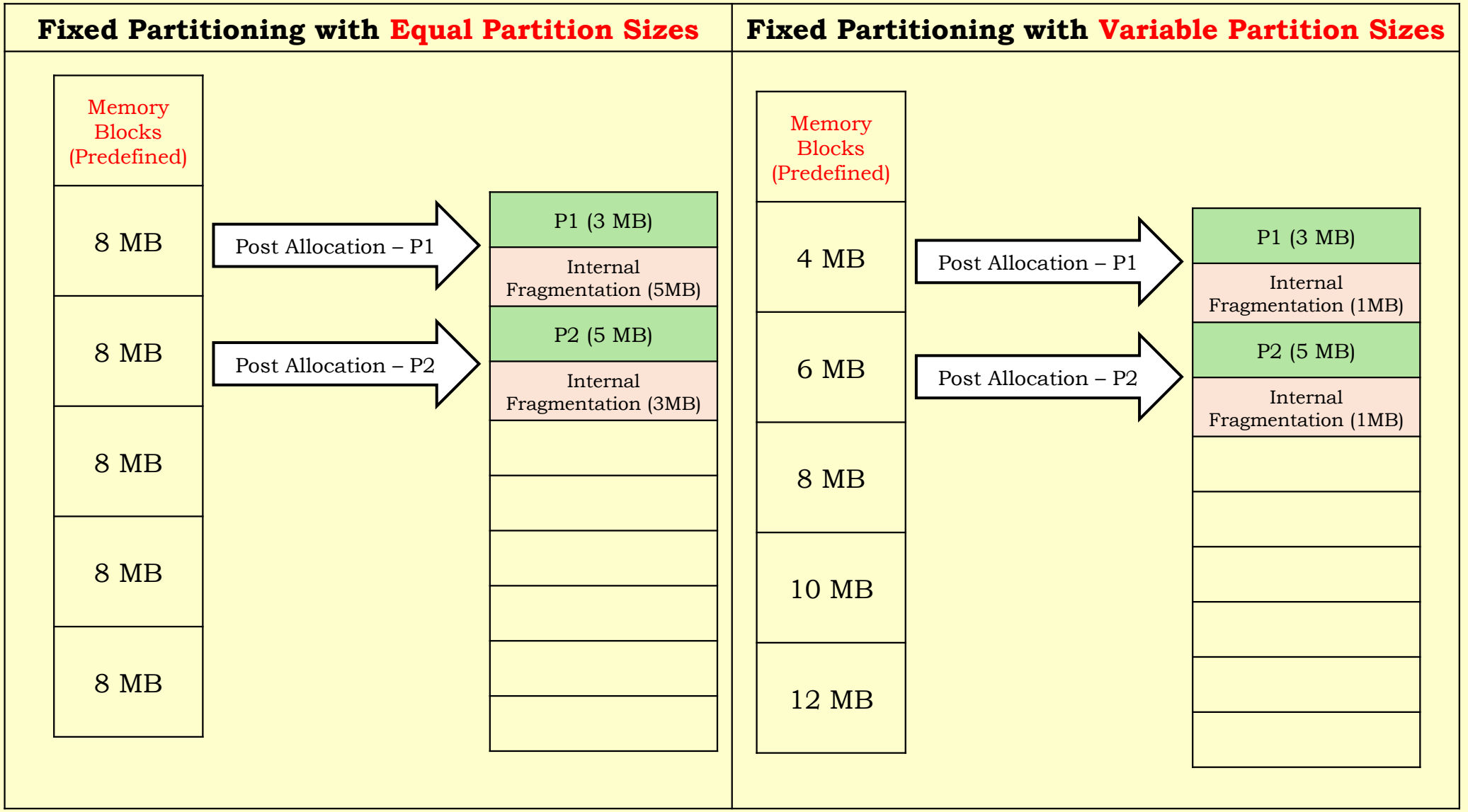
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



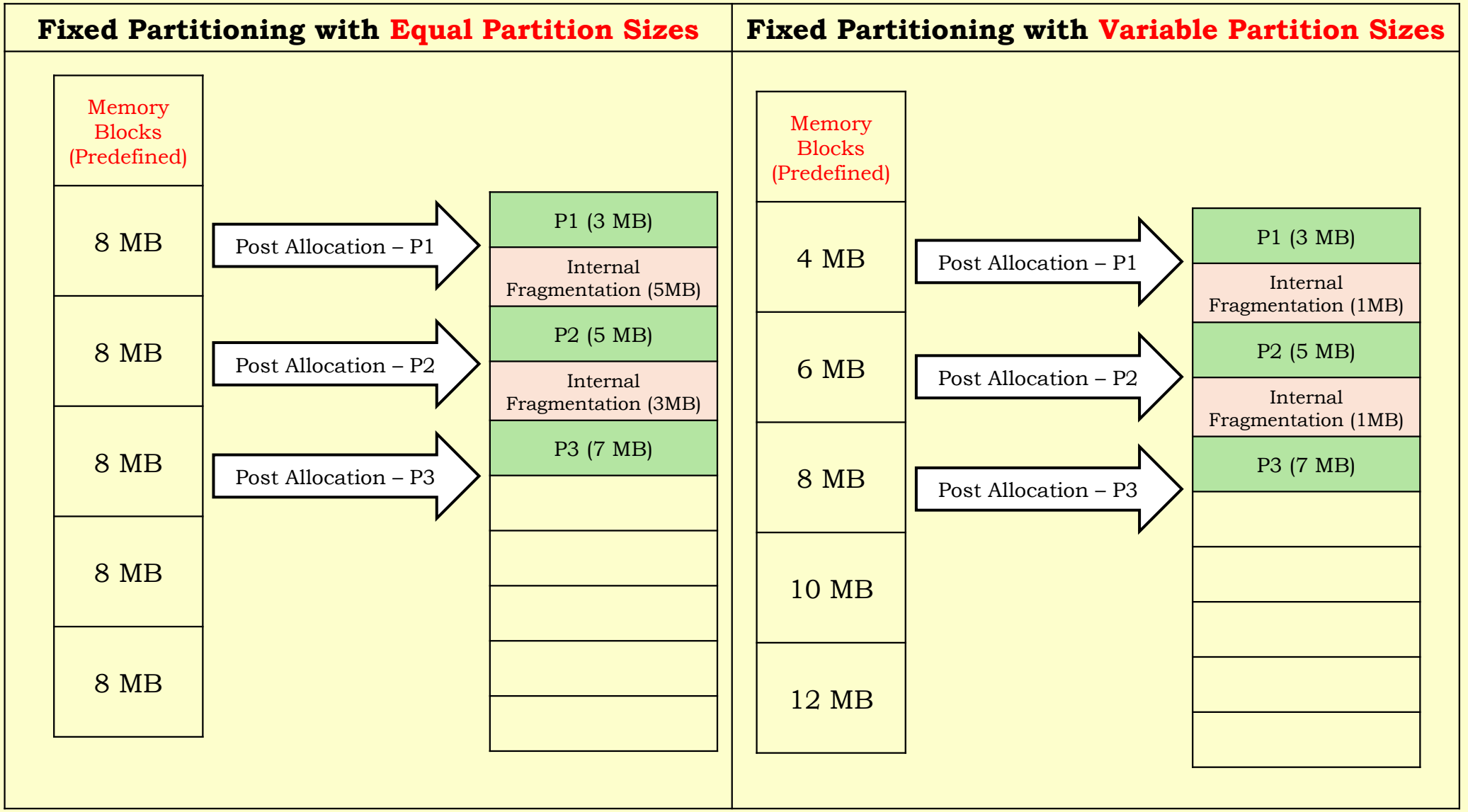
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



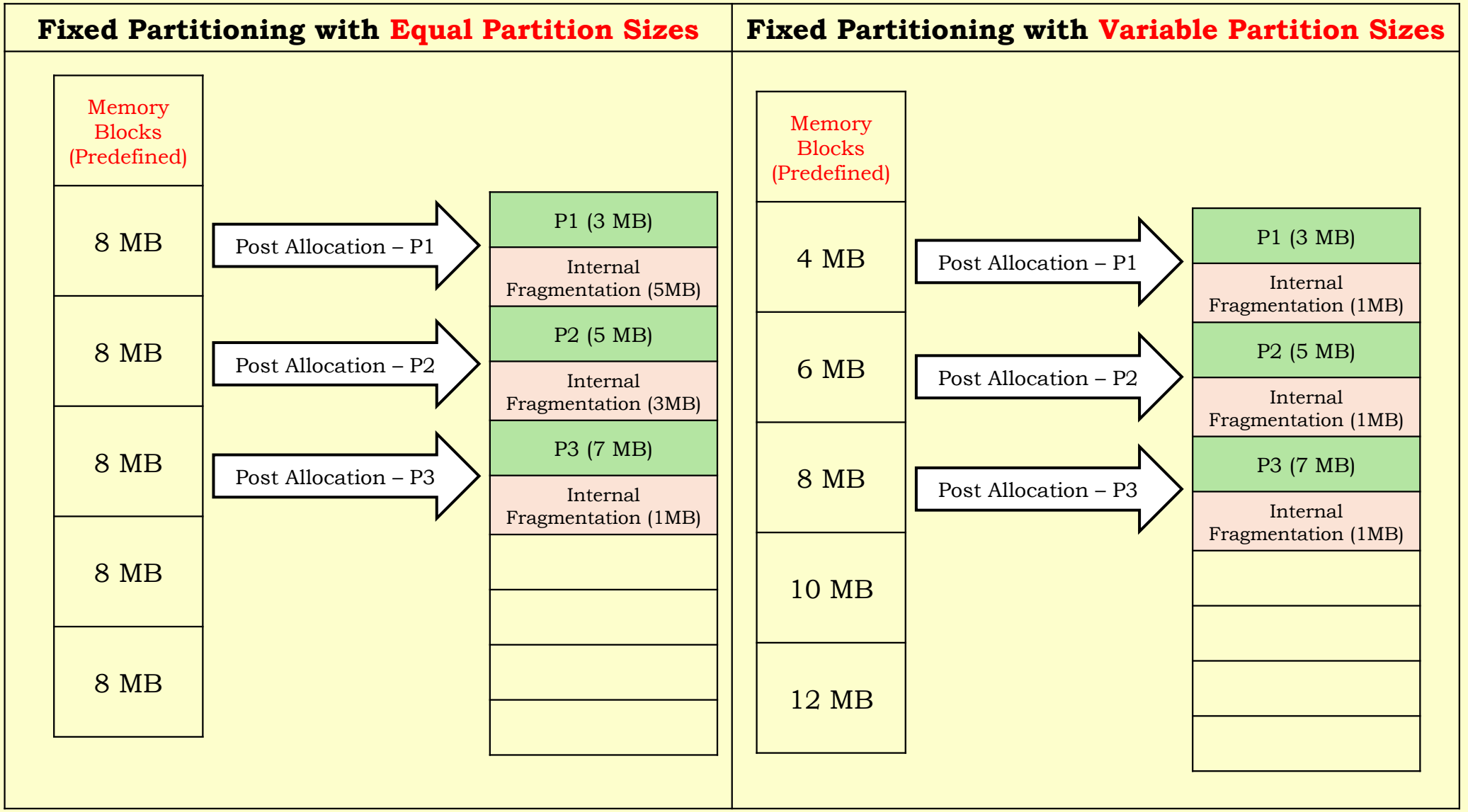
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



Contiguous Memory Allocation – Static Partitioning

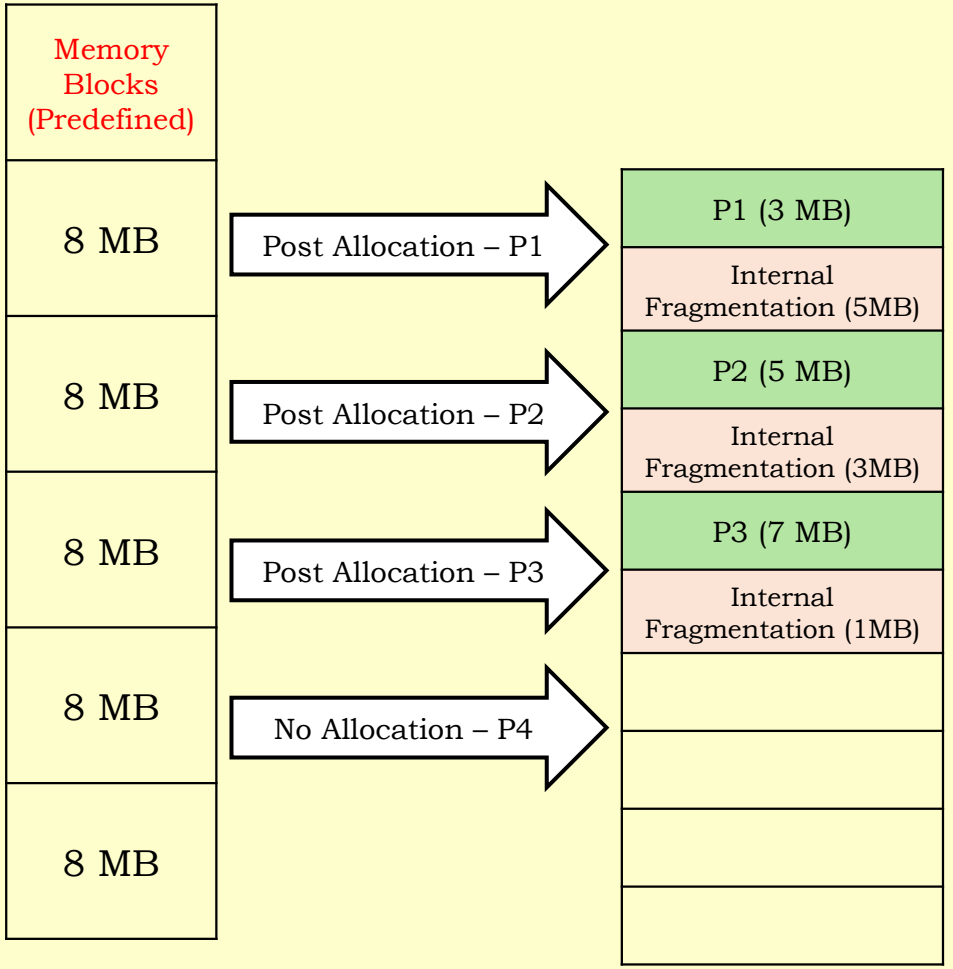
Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



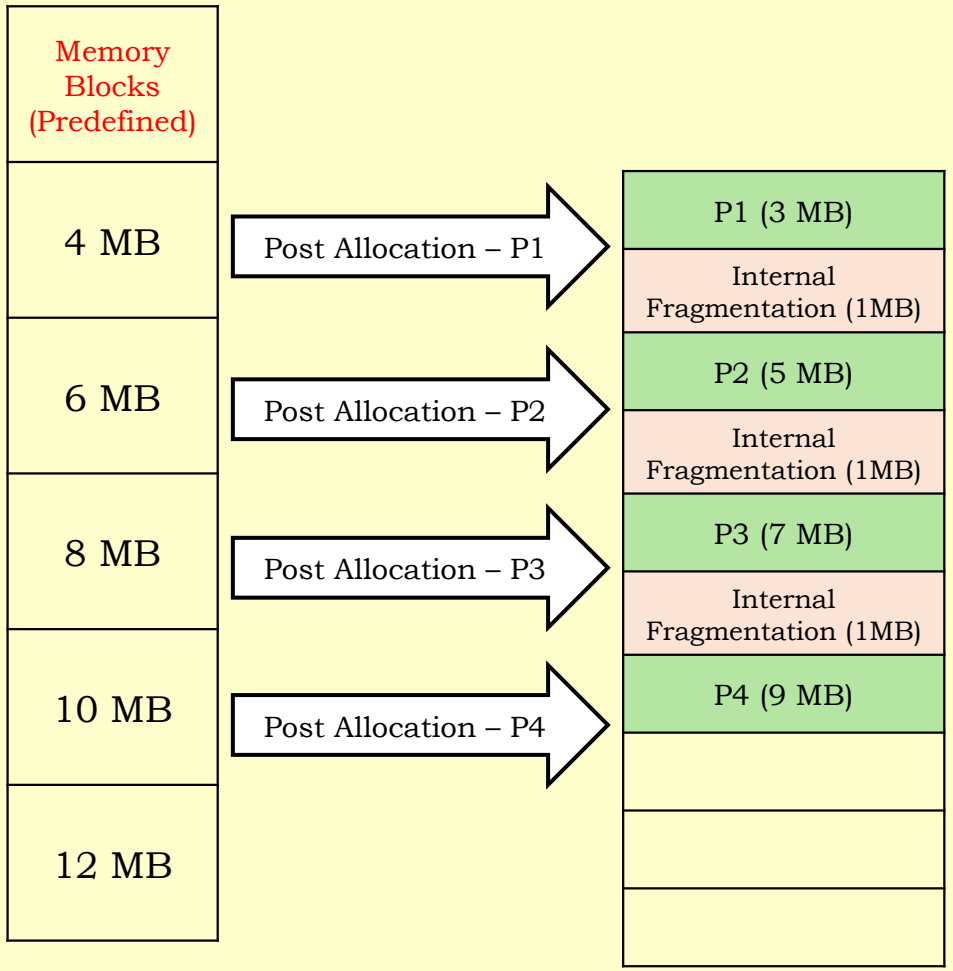
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

Fixed Partitioning with Equal Partition Sizes

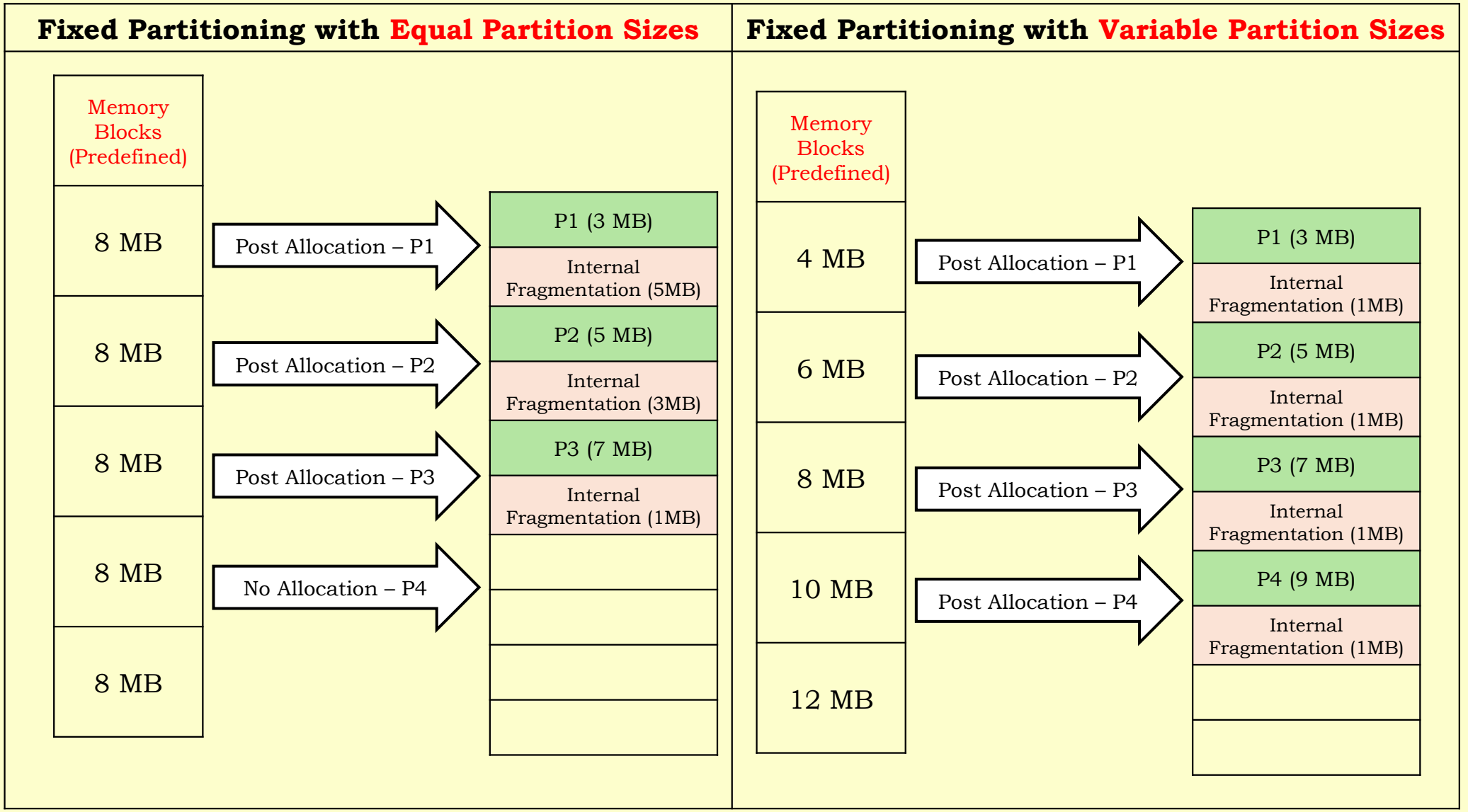


Fixed Partitioning with Variable Partition Sizes



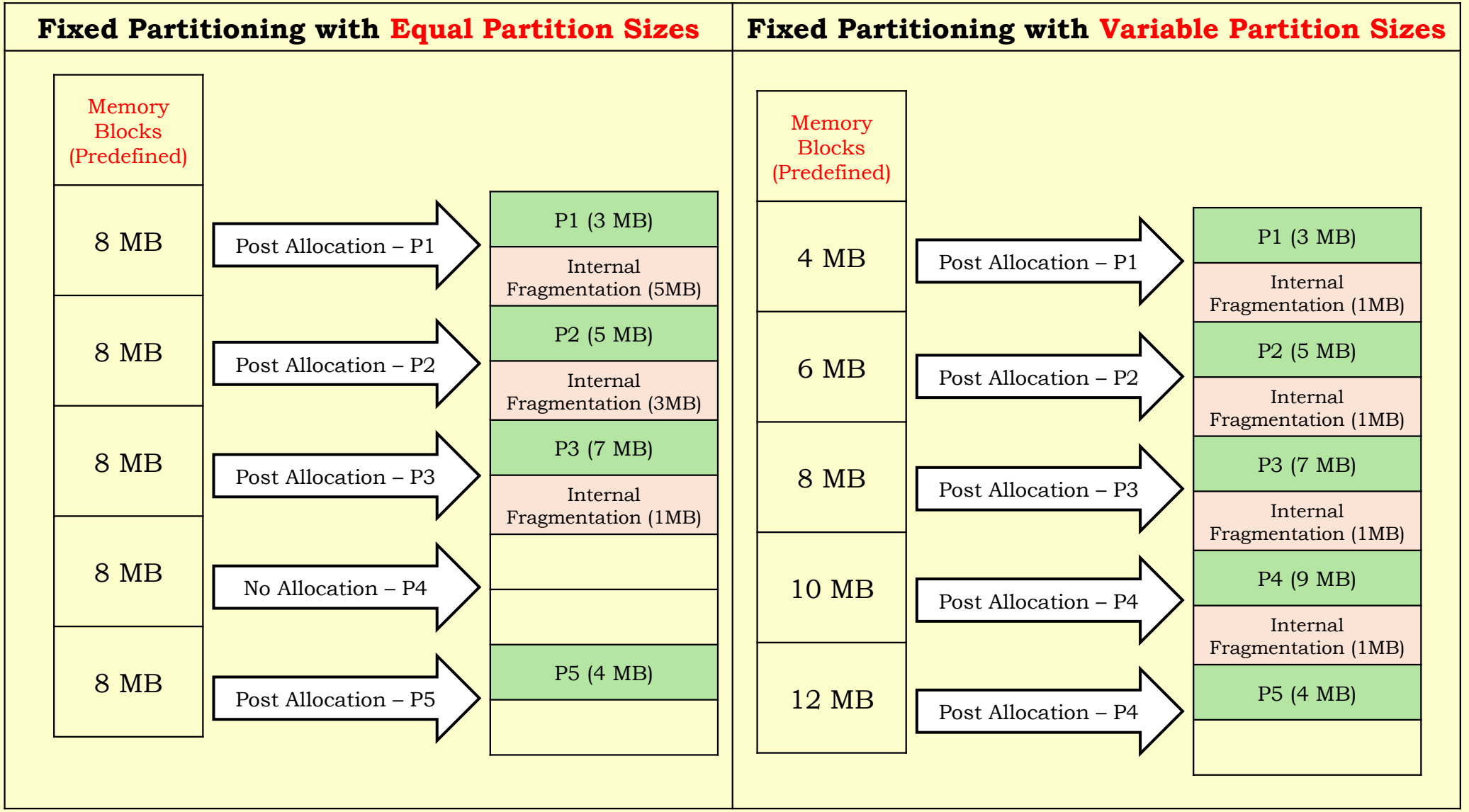
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



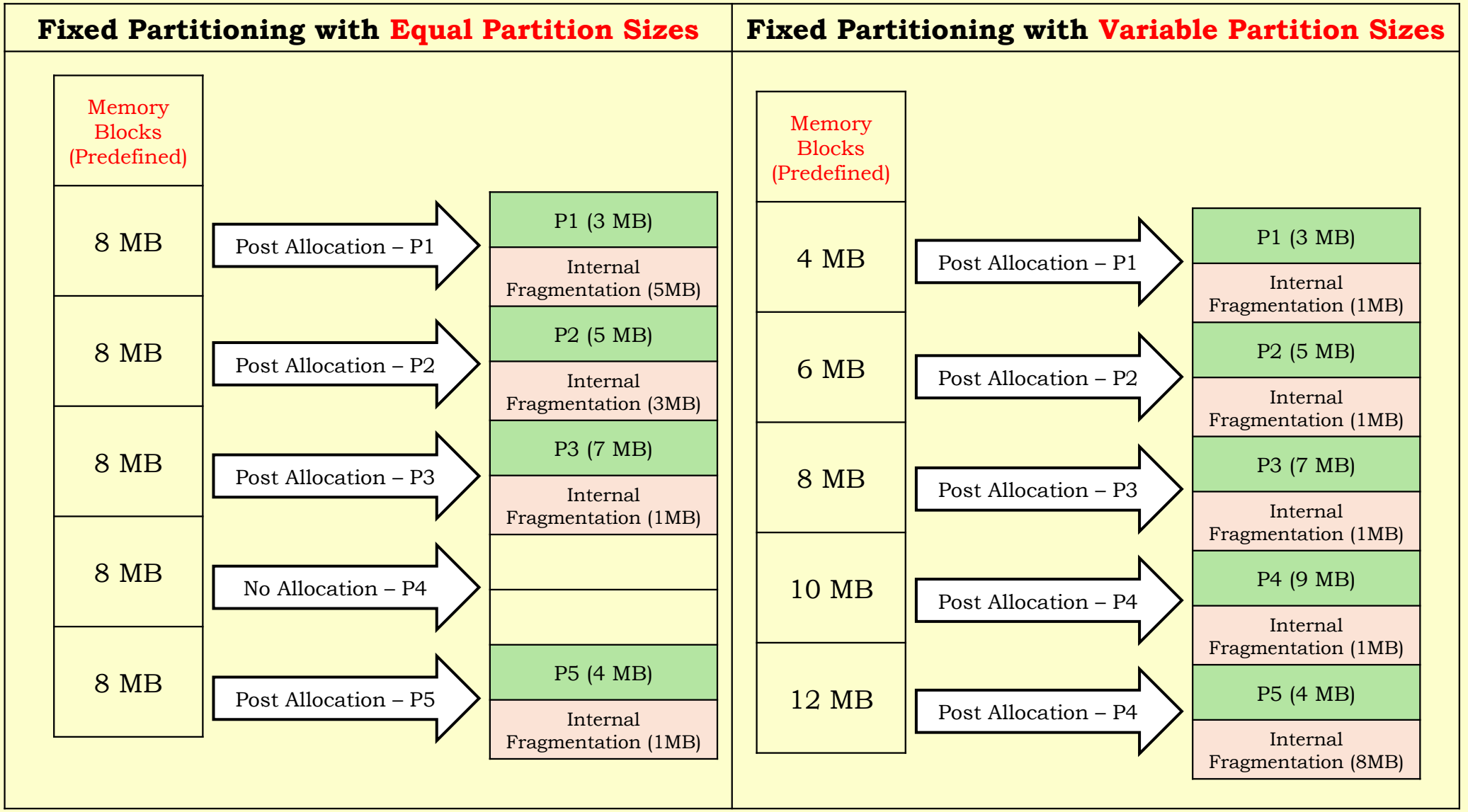
Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



Contiguous Memory Allocation – Static Partitioning

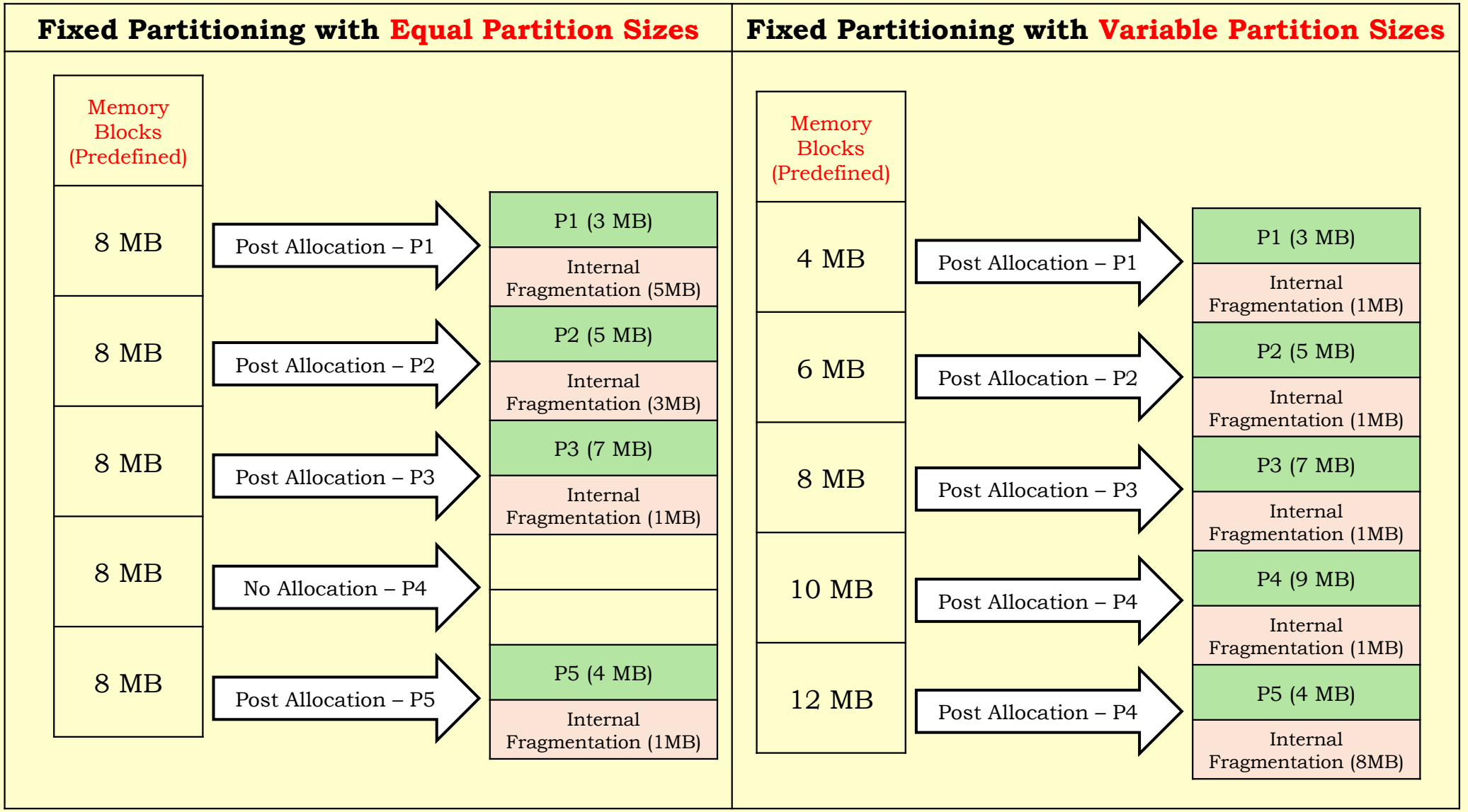
Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

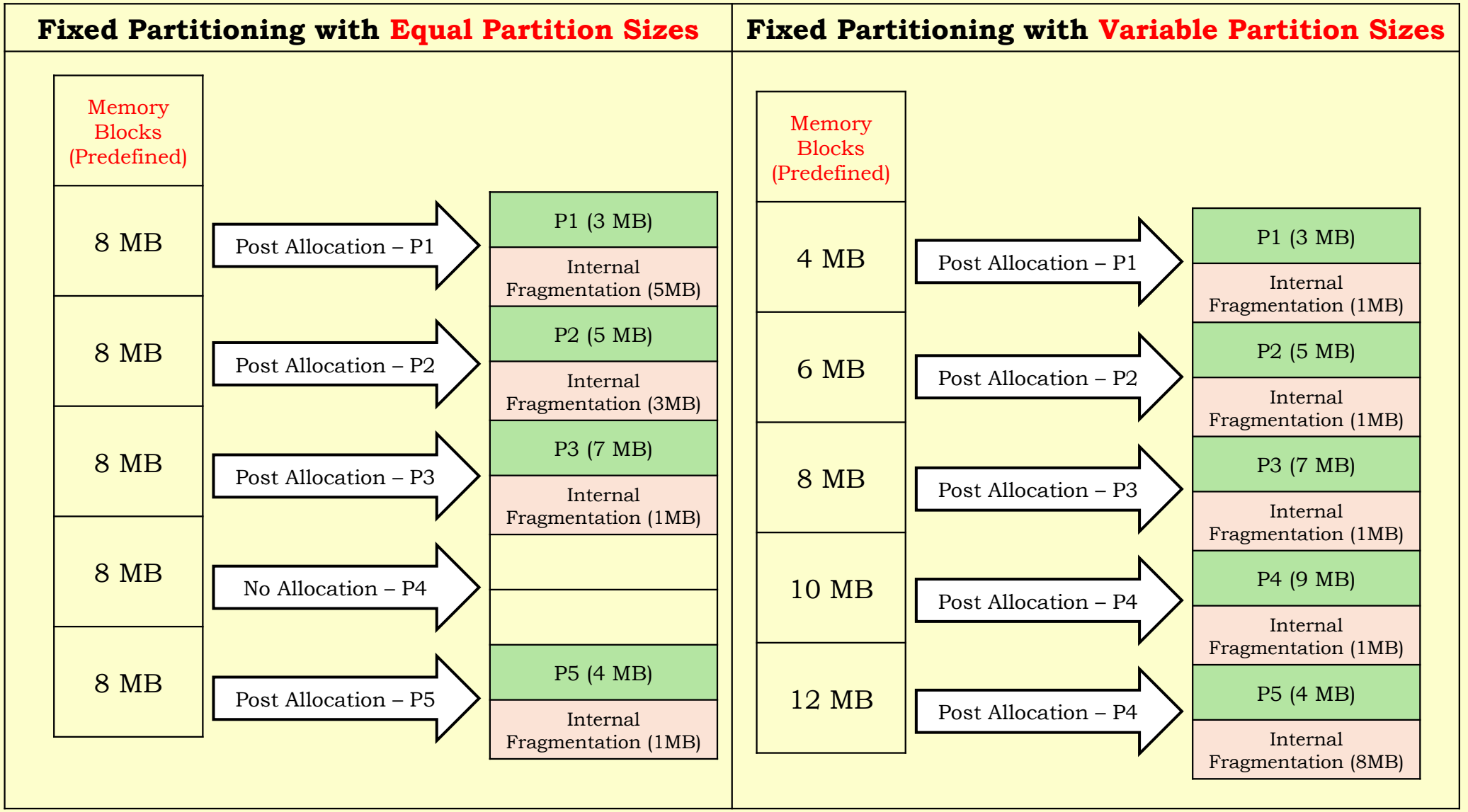
If a Process P6 and P7 arrives of size each 10 MB and 12 MB respectively still the - total free space across partitions is sufficient, a new process can't be allocated if no single contiguous partition fits it.



Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

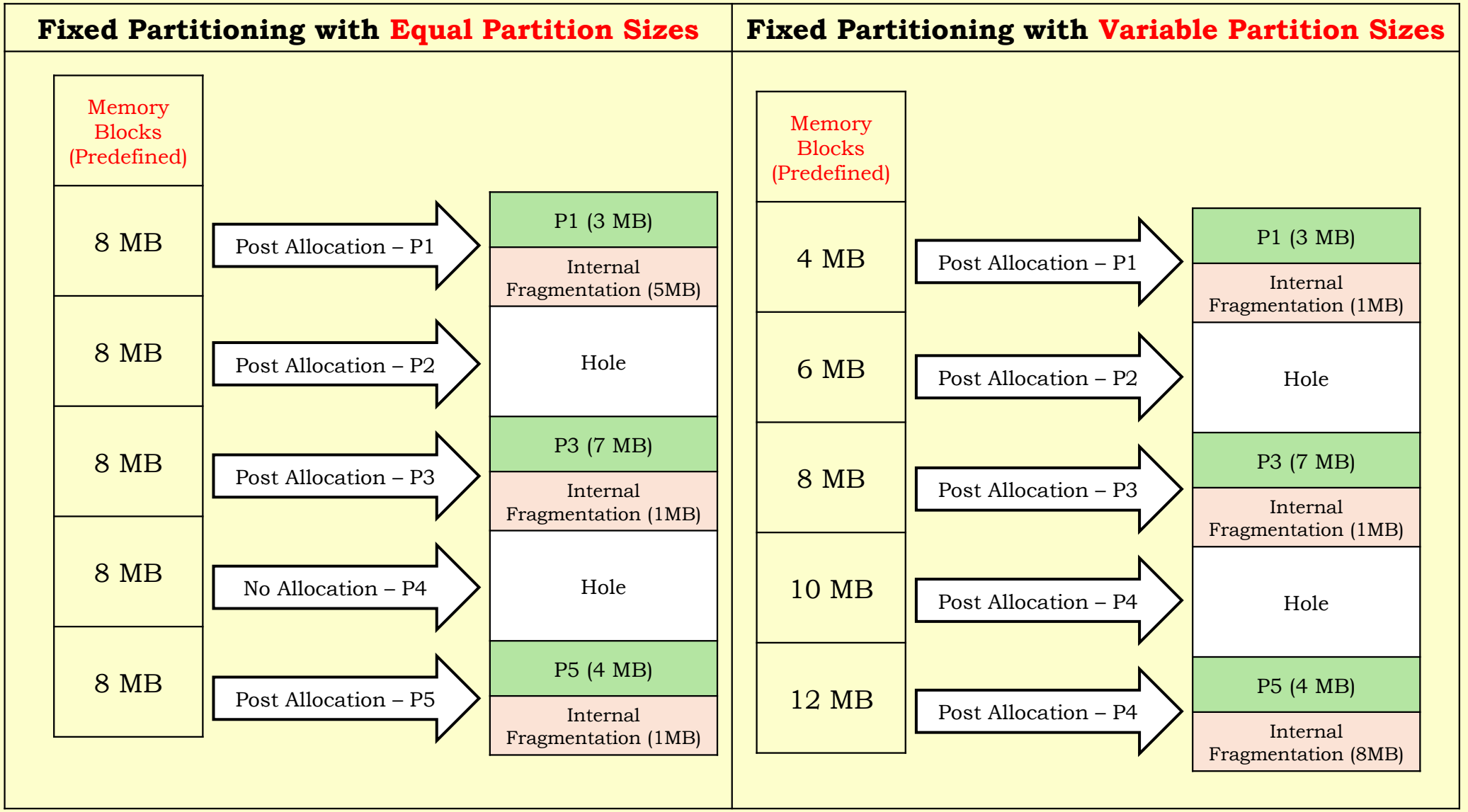
Assume P2 and P4 Terminates then Holes are created



Contiguous Memory Allocation – Static Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

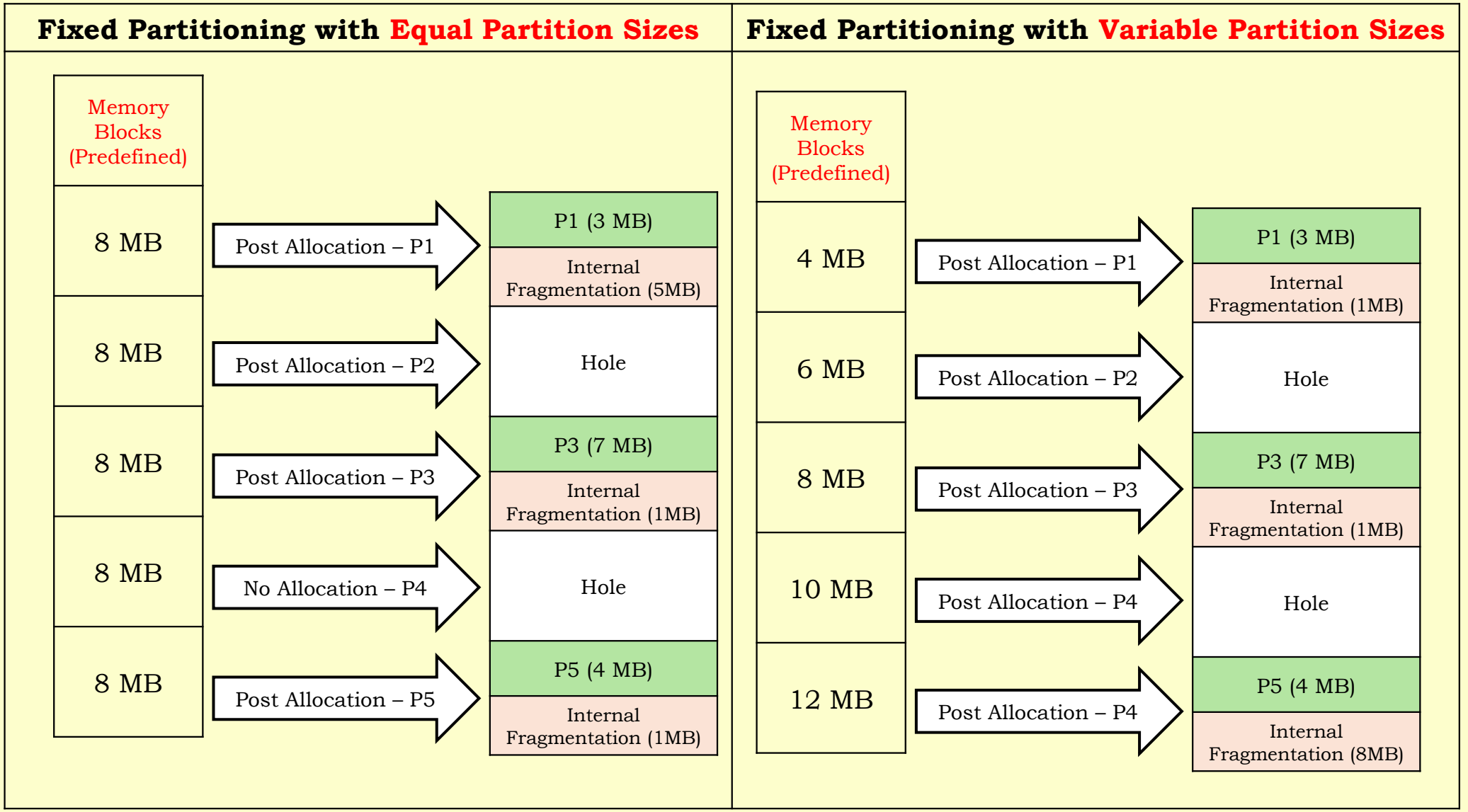
Assume P2 and P4 Terminates then Holes are created



Contiguous Memory Allocation – Static Partitioning

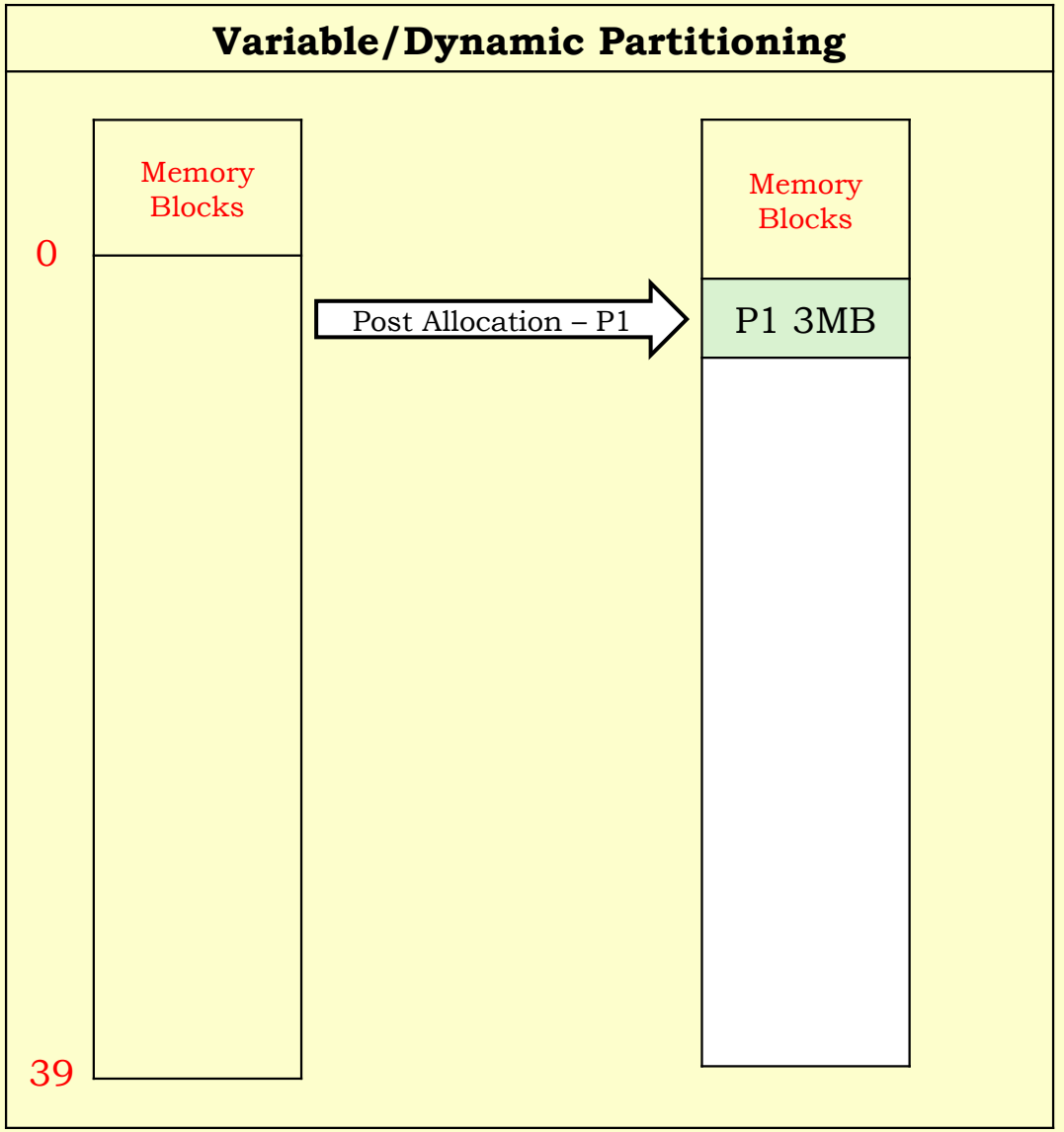
Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

Assume P9 and P10 arrives with each 16 MB size, still they cannot be allocated to memory which is External Fragmentation.



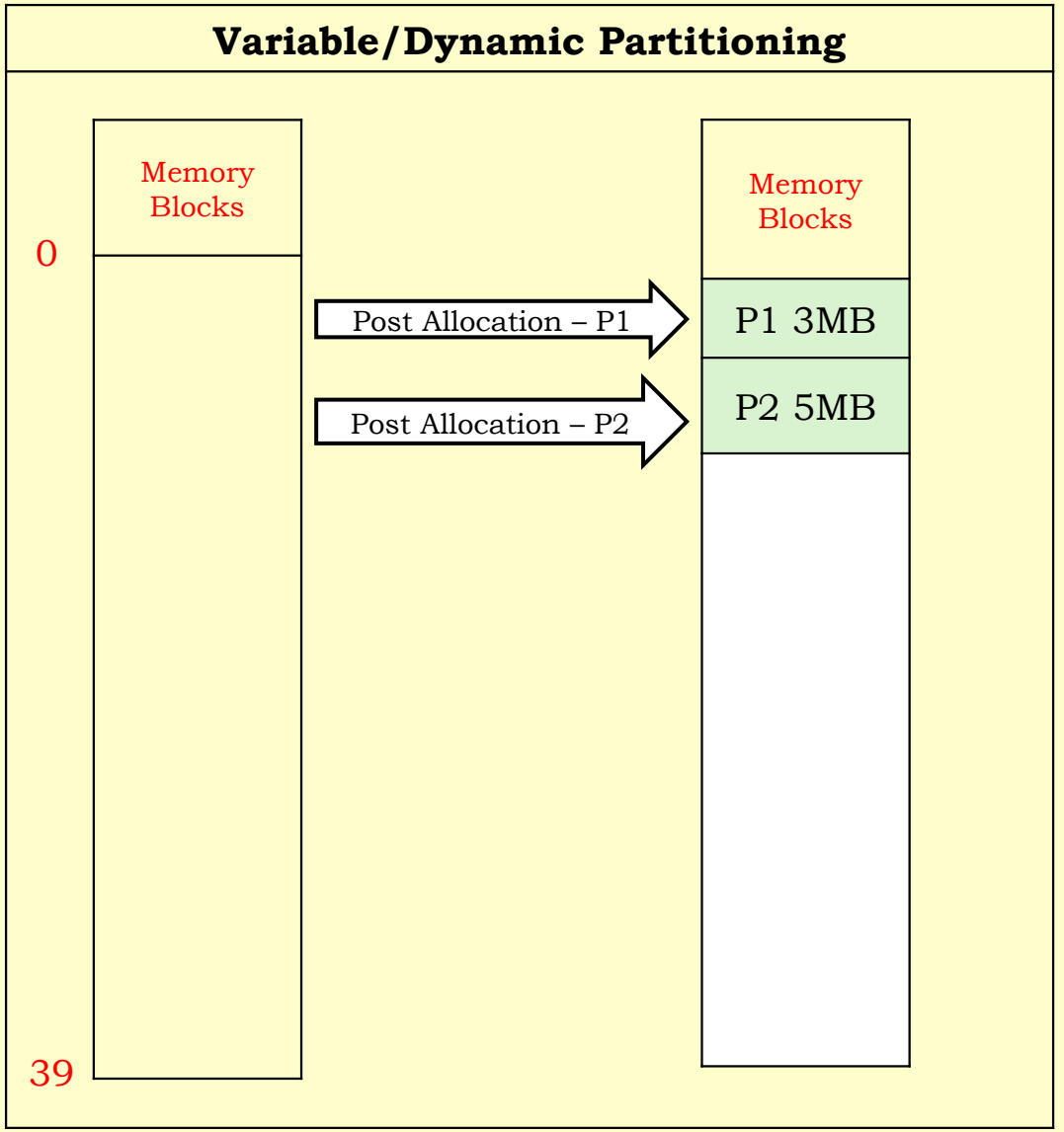
Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



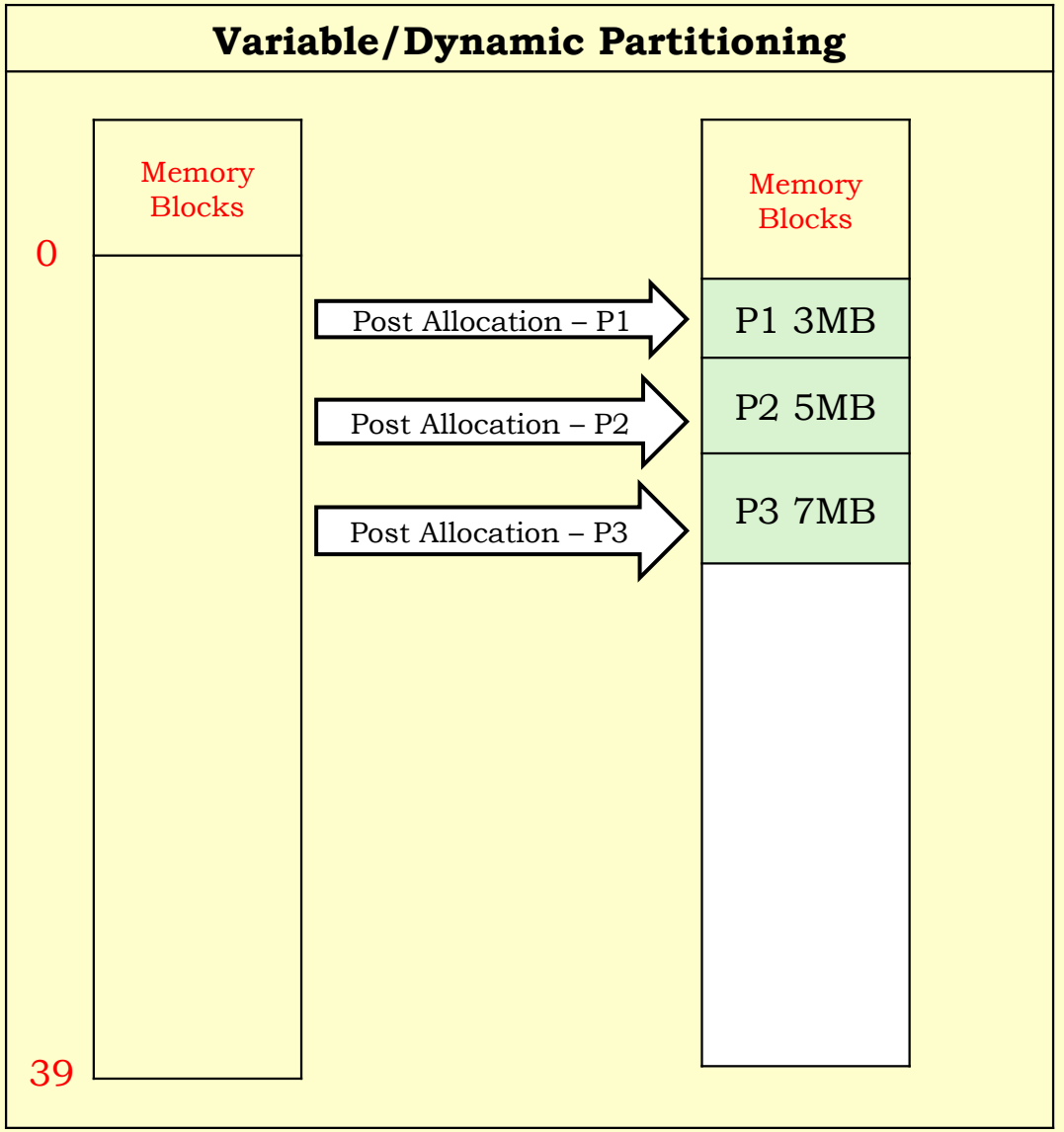
Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



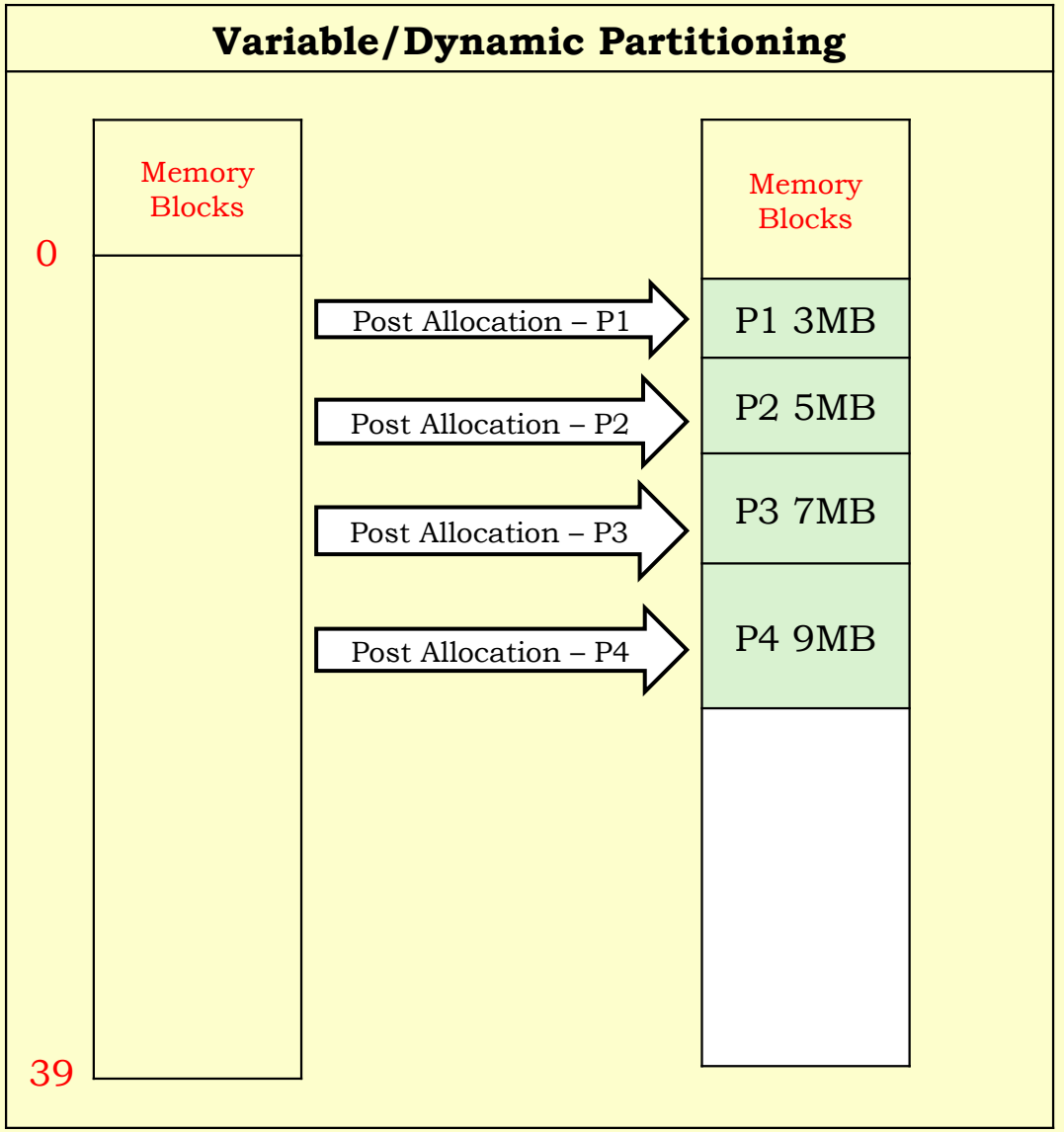
Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



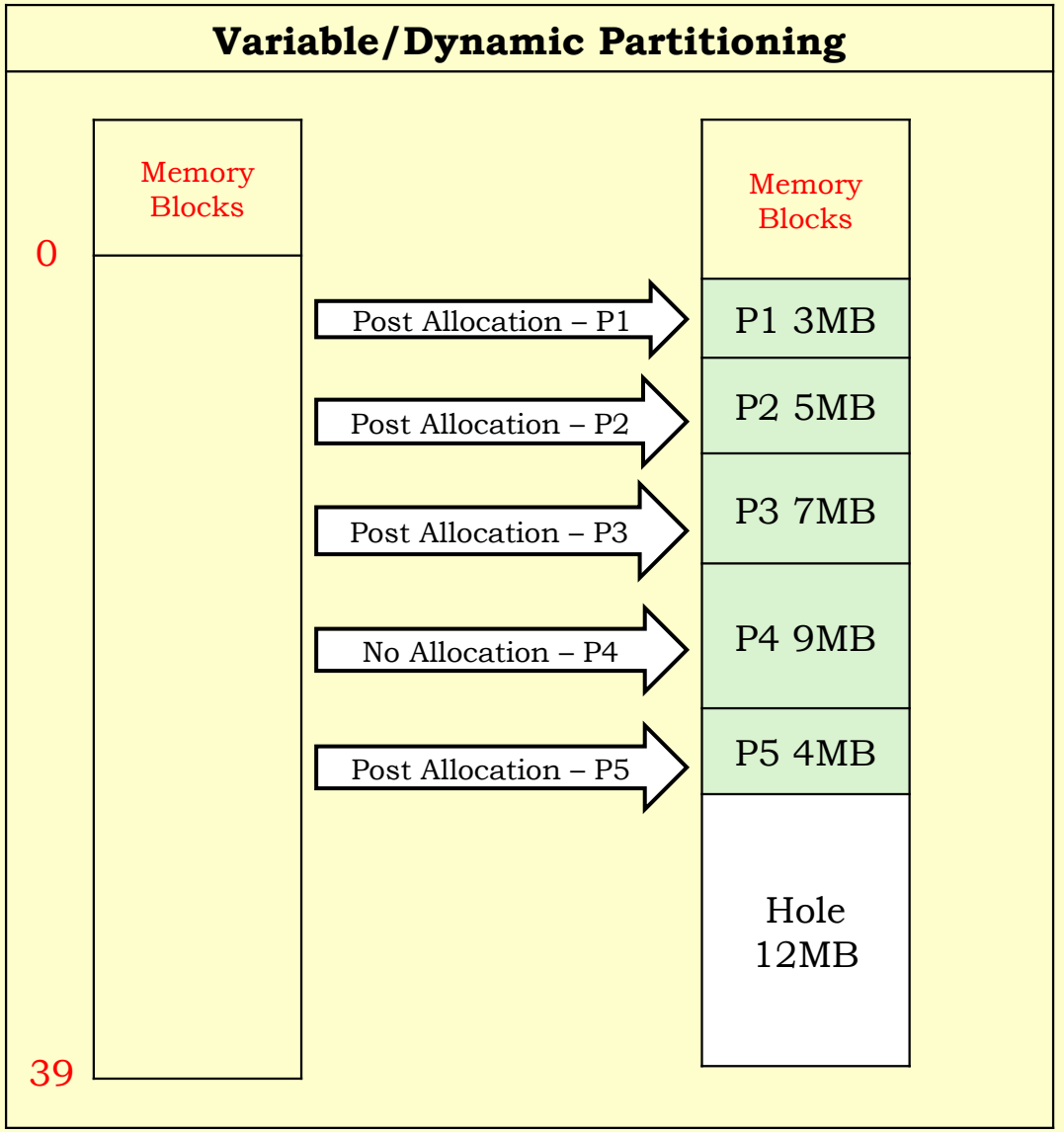
Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB



Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

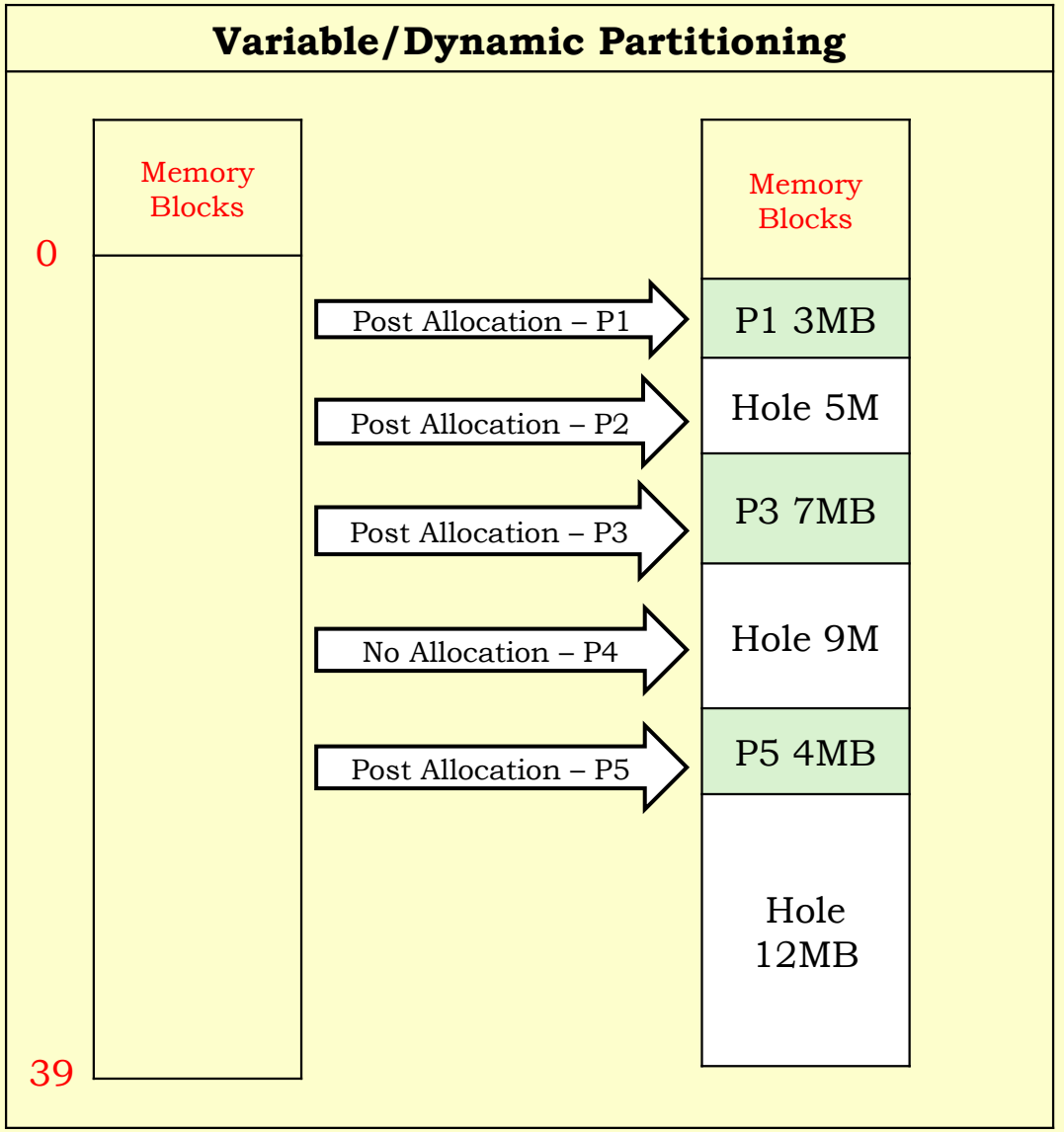


Contiguous Memory Allocation – Variable/Dynamic Partitioning

Processes
P1 - 3MB
P2 - 5MB
P3 - 7MB
P4 - 9MB
P5 - 4MB

Assume P2 and P4 Terminates then Holes are created

If P6 with size 14M arrives it cannot be allocated hence still External Fragmentation Exists. But no Internal Fragmentation.



Contiguous Memory – Memory Fragmentation

- Memory fragmentation is a problem that occurs when free memory in the system is broken into small, scattered pieces (fragments), making it difficult or impossible to allocate large contiguous blocks even though there may be enough total free memory.

- When processes are loaded and unloaded dynamically, memory space gets divided into allocated blocks (partitions) and holes (free spaces) scattered throughout RAM.

Contiguous Memory – Memory Fragmentation

Internal Fragmentation	External Fragmentation
Occurs when allocated memory blocks are larger than needed by processes.	Occurs when free memory is scattered into small non-contiguous chunks (holes).
The unused memory inside these allocated blocks results in wasted space.	Even if the total free memory is enough for a process, it may not be allocated because no single free block is large enough.
Example: If a 50 KB fixed partition is allocated but a process needs only 40 KB, 10 KB is wasted inside that partition.	Example: 100 MB of free memory is available but divided into small pieces of 10 MB, 20 MB, 30 MB – none large enough for a 40 MB allocation request.
Happens in fixed partitioning or paging systems.	Happens in dynamic partitioning and contiguous allocation systems.

Memory Allocation Algorithms For Variable Partitioning

First Fit:

- Scan memory from the beginning and allocate the first available hole that is big enough for the process.
- Fast and simple but can lead to more fragmentation near the beginning of memory.

Best Fit:

- Search for the smallest available hole that will fit the process.
- Tries to minimize leftover space but can create many small unusable holes (more external fragmentation).

Worst Fit:

- Search for the largest available hole and allocate from it.
- The idea is to leave the largest leftover holes, hoping they'll be usable for future requests; however, it can waste large spaces inefficiently.

Next Fit

- Similar to First Fit, but instead of starting from the beginning every time, it resumes the search from the location where it left off last time.
- This reduces the search overhead for large memory but still allocates contiguous blocks.

Memory Allocation Algorithms For Variable Partitioning

Algorithm	How it allocates memory	Advantage	Disadvantage
First Fit	Allocates first large enough block found	Fast, simple	Can lead to fragmentation at start
Best Fit	Allocates smallest suitable block	Minimizes leftover space	Slow, may create many small holes
Worst Fit	Allocates largest block available	Avoids tiny leftover holes	Can waste large blocks
Next Fit	Like First Fit, but search starts from last allocation point	Faster than first fit theoretically	May still fragment memory

Memory Allocation Algorithms For Variable Partitioning

- Exercise 1 - Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB. These partitions need to be allocated to four processes of sizes 357 KB, 210 KB, 468 KB and 491 KB in that order.

Perform the allocation of processes using- First, Best and Worst Fit

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
	200KB
P1	400KB
	600KB
	500KB
	300KB
	250KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
	500KB
	300KB
	250KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
	600KB
	500KB
	300KB
	250KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
	600KB
	500KB
	300KB
P2 210	250KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
	600KB
P3 468	500KB
	300KB
P2 210	250KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
P4 491	600KB
P3 468	500KB
	300KB
P2 210	250KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
P4 491	600KB
P3 468	500KB
	300KB
P2 210	250KB

WORST FIT

Partitions	SIZE
	200KB
	400KB
P1 357	600KB
	500KB
	300KB
	250KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
P4 491	600KB
P3 468	500KB
	300KB
P2 210	250KB

WORST FIT

Partitions	SIZE
	200KB
	400KB
P1 357	600KB
P2 - 210	500KB
	300KB
	250KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 357KB
P2 – 210KB
P3 – 468KB
P4 – 491KB

FIRST FIT

Partitions	SIZE
	200KB
P1	400KB
P2	600KB
P3	500KB
	300KB
	250KB

P4 – 491KB – Not Allocated

BEST FIT

Partitions	SIZE
	200KB
P1 357	400KB
P4 491	600KB
P3 468	500KB
	300KB
P2 210	250KB

WORST FIT

Partitions	SIZE
	200KB
	400KB
P1 357	600KB
P2 - 210	500KB
	300KB
	250KB

P3 – 468 KB & P4 – 491KB – Not Allocated

Memory Allocation Algorithms For Variable Partitioning

- Example 2 – There are 4 processes requesting for Memory of following size - 300, 25, 125, 50. The current memory snapshot is given to you. It follows Variable Partitioning
- Use First, Best and Worst Fit allocation

Schemes for the given Processes.

Partitions	SIZE
Occupied	50KB
	150KB
Occupied	300KB
	350KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
Occupied	50KB
	150KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
Hole	125KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT	BEST FIT	WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
	150KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
	150KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

P4 – 50KB – Not Allocated

WORST FIT

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

P4 – 50KB – Not Allocated

WORST FIT

Partitions	SIZE
Occupied	50KB
	150KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

P4 – 50KB – Not Allocated

WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
Hole	125KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

P4 – 50KB – Not Allocated

WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
Hole	50KB
Occupied	600KB

Memory Allocation Algorithms For Variable Partitioning

Processes
P1 – 300KB
P2 – 25KB
P3 – 125KB
P4 – 50KB

FIRST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

BEST FIT

Partitions	SIZE
Occupied	50KB
P2 - 125	125KB
Hole	25KB
Occupied	300KB
P1 - 300	300KB
P2 - 25	25KB
Hole	25KB
Occupied	600KB

P4 – 50KB – Not Allocated

WORST FIT

Partitions	SIZE
Occupied	50KB
P2 - 25	25KB
P3 - 125	125KB
Occupied	300KB
P1 - 300	300KB
P4 - 50	50KB
Occupied	600KB

Non-Contiguous Memory

- Non-contiguous memory is a memory management technique where a single process is allocated physical memory in separate, non-adjacent blocks.
- Instead of requiring one large, continuous chunk of memory, the operating system can scatter the parts of a process throughout available free spaces in RAM.
- This approach addresses the major limitations of contiguous memory allocation, which often leads to significant wasted space due to **external fragmentation**. External fragmentation occurs when free memory is divided into small, non-contiguous blocks, making it impossible to allocate a large process even if the total free memory is sufficient.

Virtual Memory – A memory management technique

- Virtual memory is a memory management technique that **provides an illusion of a very large main memory to programs, even if the actual physical memory (RAM) is much smaller**.
- It allows processes to use more memory than physically available by temporarily transferring data between RAM and disk storage, enabling efficient multitasking and memory utilization.

Virtual Memory

- Programs use virtual addresses without worrying about physical memory limits.
- The OS and hardware work together to map virtual addresses to physical addresses transparently.
- When a required data page is not in physical memory, it is paged in from disk (swap space), and less-used pages are paged out, making room for active data.

Virtual Memory

- When physical memory runs out, less-used parts of programs (pages) are temporarily moved to disk storage, allowing systems to run larger programs or multiple programs simultaneously without requiring all of them to be fully loaded in RAM at once.
- The Translation Lookaside Buffer (TLB) is a small, fast cache inside the Memory Management Unit (MMU) that stores recent virtual-to-physical address translations to speed up virtual memory access.

Virtual Memory – Why required?

- Allows execution of programs larger than physical memory.
- Increases the degree of multiprogramming by running multiple processes concurrently.
- Optimizes memory usage by loading only needed program parts into RAM.
- Provides memory protection and isolation between processes.
- Reduces I/O operations since programs do not need full memory allocation at once.

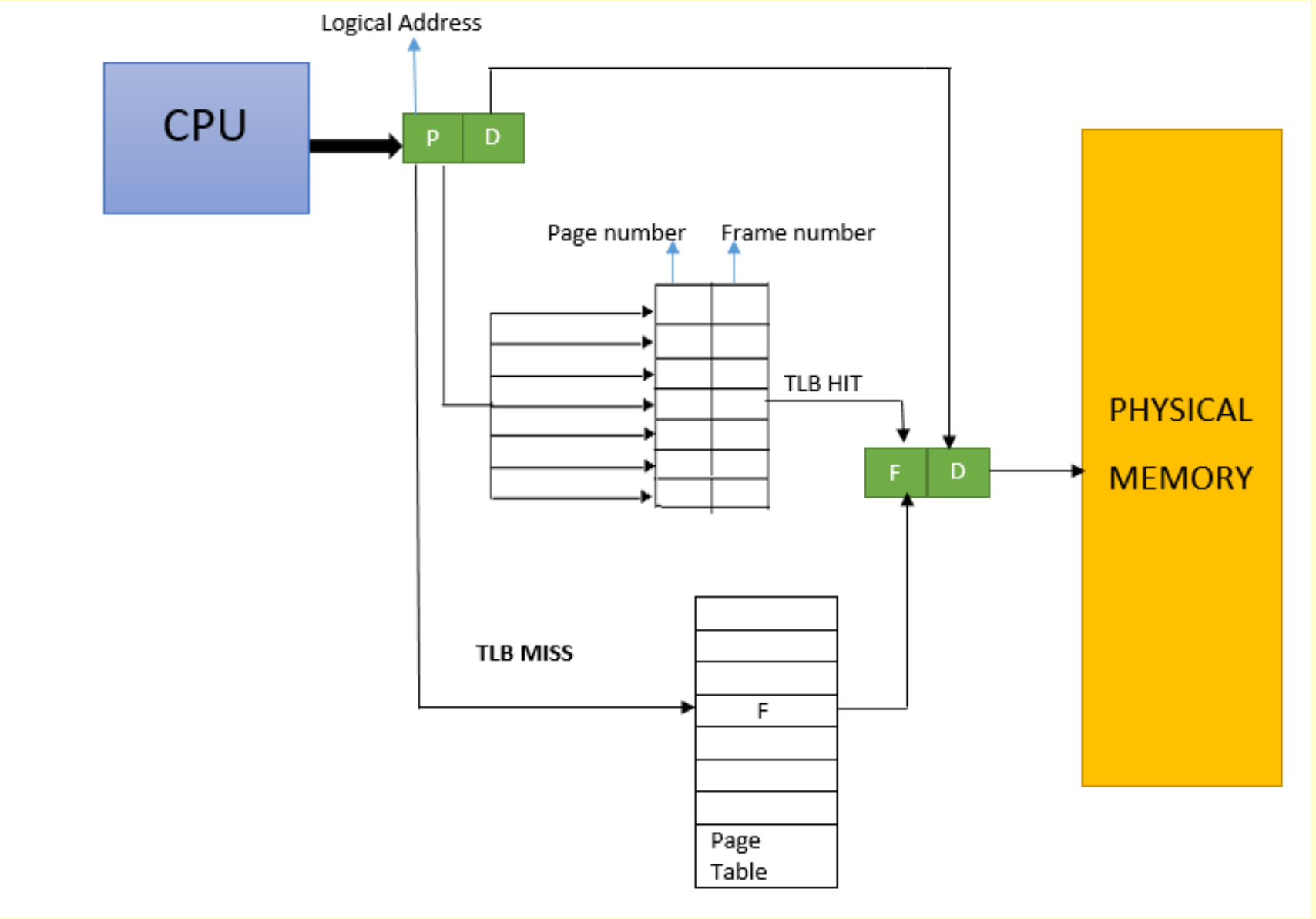
Paging

- Paging is the **key technique** used to **implement virtual memory**.
- It divides the process's address space and the physical memory into fixed-size blocks:
  - **Pages:** Fixed-size blocks of the process's virtual address space.
  - **Frames:** Corresponding fixed-size blocks in physical memory, **same size as pages**.

Paging

- The operating system maintains a page table for each process, which maps the virtual page numbers to physical frame numbers.
- This table is used by the Memory Management Unit (MMU) to translate virtual addresses into physical addresses at runtime.

Virtual Memory – Hardware Support



Virtual Memory – Hardware Support

- The CPU generates a virtual address consisting of a page number (P) and an offset (D).
  - The TLB is checked first to see if it contains the page number.
    - If TLB hit (page number found): The corresponding frame number (F) is retrieved quickly.
    - Combine frame number (F) with the offset (D) to get the physical address.
    - Access memory at that physical address.
  - If TLB miss (page number not found):
    - Page table is accessed in main memory to find the frame number.
    - The TLB is updated with this translation for future quick access.
    - The physical address is formed, and memory is accessed.

How to Implement Virtual Memory – Paging

- Paging is a memory management technique used by operating systems to efficiently manage the physical memory available to programs.
- It allows programs to use more memory than physically available and ensures efficient use of memory without requiring contiguous allocation.

How to Implement Virtual Memory – Paging - Working

- **Division into Pages and Frames:**
  - The logical address space (program memory) is divided into fixed-size blocks called pages (e.g., 4KB each).
  - The physical memory (RAM) is divided into blocks of the same size as pages called frames.
- **Page Mapping:**
  - Each page can be loaded into **any** frame in physical memory.
  - This allows **non-contiguous allocation of memory**, avoiding the problem of external fragmentation.

How to Implement Virtual Memory – Paging - Working

- **Page Table:**
  - The OS keeps a page table which maps each virtual page to a physical frame.
  - When the CPU accesses memory using a virtual (logical) address, the page number is looked up in the page table to find the frame number.
  - The physical address is computed by combining the frame number with the offset within the page.

How to Implement Virtual Memory – Paging - Working

- **Paging and Virtual Memory**
  - If a required page is not in physical memory (a page fault), it is loaded from secondary storage (disk).
  - This allows processes to run as if there is more RAM than physically present, seamlessly using disk as extended memory.

Paging - Working

- CPU is executing the Process P1 which of size → 4Bytes.
- The Page Size → 2Bytes
- The main memory or RAM total size is → 16Bytes
- Rules:
  - Page Size = Frame Size; hence Frame Size → 2Bytes
  - The logical address space(LAS) is designed to accommodate exactly the process size; hence LAS → 4Bytes

Paging – Working - – Logical Address

- When CPU executes a Process it generates Logical Address for it
- The Logical Address contains → Page No. and Page offset
- Now find the No. of Pages for this P1 and represent it in Bits.

$$No.of\ Pages = \frac{Process\ Size}{Page\ Size} = \frac{4}{2} = 2\ pages$$

- Representing the Pages in bits

$$\text{No. of bits Needed} = \log_2 2 = 1 \text{ bit}$$

- Representing the offset in bits

$$\text{No. of bits Needed} = \log_2 2 = 1 \text{ bit}$$

- Logical address generated by CPU for this process will have → 2bits, The MSB (most significant bit) represents the page number, while the LSB (least significant bit) represents the offset within the page.

Paging – Working – **Logical Address**

- So P1 will be divided into two pages shown below.

Page No. (1 Bit)	Bytes in Page
Page 0	**Offset at 0 bit** Byte 0 | **Offset at 1 bit** Byte 1
Page 1	**Offset at 0 bit** Byte 2 | **Offset at 1 bit** Byte 3

- The logical address generated by CPU for P1 will be like:

Page No. (1 Bit)	Page Offset (1 Bit)
Page 0	Since 1 bit either 0 or 1
Page 1	Since 1 bit either 0 or 1

Paging – Working – **Physical Memory & Physical Address**

- Now the RAM is of total → 16Bytes and Frame Size → 2Bytes
- Finding the No. of Frames in RAM

$$No.\,of\ Frames = \frac{Memory\ Size}{Frame\ Size} = \frac{16}{2} = 8\ \text{frames}$$

- The Physical Address contains → Frame No. and Page offset
- Representing the Frames in bits $\quad No.\,of\,bits\,Needed = \log_2 8 = 3\ bit$
- Representing the offset in bits $\quad No.\,of\,bits\,Needed = \log_2 2 = 1\ bit$
- Physical address will have → 3bits MSB (most significant bit) representing the frame number, while the 1bit LSB (least significant bit) representing the offset.

Paging - Working – Physical Memory & Physical Address

- The Frame Structure of RAM will be like:

Frame No (3 Bits)			Byte Addresses	
0	0	0	Byte 0 **Offset at 0 bit**	Byte 1 **Offset at 1 bit**
0	0	1	Byte 2 **Offset at 0 bit**	Byte 3 **Offset at 1 bit**
0	1	0	Byte 4 **Offset at 0 bit**	Byte 5 **Offset at 1 bit**
0	1	1	Byte 6 **Offset at 0 bit**	Byte 7 **Offset at 1 bit**
1	0	0	Byte 8 **Offset at 0 bit**	Byte 9 **Offset at 1 bit**
1	0	1	Byte 10 **Offset at 0 bit**	Byte 11 **Offset at 1 bit**
1	1	0	Byte 12 **Offset at 0 bit**	Byte 13 **Offset at 1 bit**
1	1	1	Byte 14 **Offset at 0 bit**	Byte 15 **Offset at 1 bit**

Paging - Working – Page Table

- The page table maps logical pages to physical frames.
- For every Process one page table is created.
- For this process P1 Page table will have the following:
  - Number of entries → 2 (As per Logical Page)
  - Frames represented as 3 bits
- So, Total **page table size**:

No. of Entries x Entry Size → 2 × 3 = 6 bits

Page Table Structure

	Frame No (3 Bits)			Present Bit (1 bit)	Protection Bits R/W/E	Dirty Bits (1 Bit)
Page 0	0	0	0			
Page 1	0	0	1			
Page 2	0	1	0			
Page 3	0	1	1			
Page 4	1	0	0			
Page 5	1	0	1			
Page 6	1	1	0			
Page 7	1	1	1			

Paging - Working – Page Table

- If the Operating System allocates: Page 0 → Frame 3, Page 1 → Frame 6. The page table would look like
- Use this equation to find the physical Address

Physical Address = (Frame Number × Page Size) + Offset

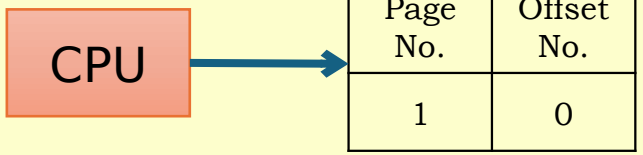
Page Table Structure

	Frame No (3 Bits)			Present Bit (1 bit)	Protection Bits R/W/E	Dirty Bits (1 Bit)
	0	0	0			
	0	0	1			
	0	1	0			
Page 0	0	1	1	1	R/W = 1	0
	1	0	0			
	1	0	1			
Page 1	1	1	0	1	R/W = 1	0
	1	1	1			

Paging - Working – Address Translation/Mapping

- CPU runs P1 and request for 1st byte data of it.
- logical Address generated by CPU → 1 0
  - Page no.→ 1 & Page Offset → 0

P1
Logical address for 1st Byte

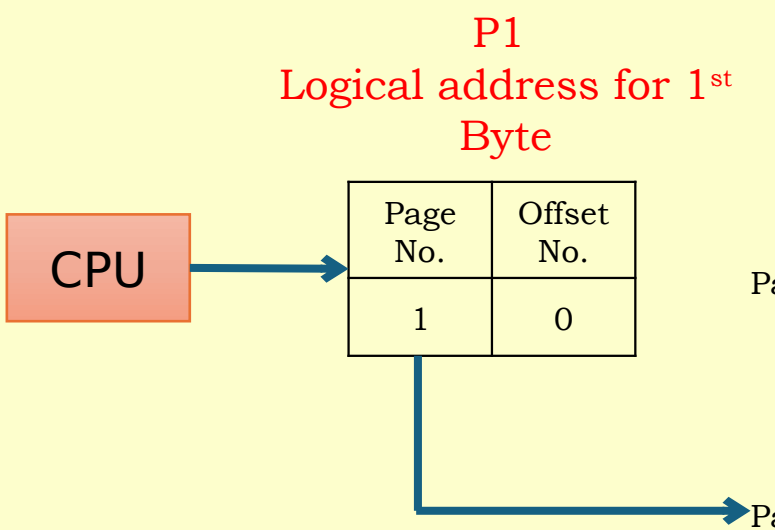


Page Table Structure

	Frame No (3 Bits)			Present Bit (1 bit)	Protection Bits R/W/E	Dirty Bits (1 Bit)
	0	0	0			
	0	0	1			
	0	1	0			
Page 0	0	1	1	1	R/W = 1	0
	1	0	0			
	1	0	1			
Page 1	1	1	0	1	R/W = 1	0
	1	1	1			

Paging - Working – Address Translation/Mapping

- Perform Lookup in Page Table structure using Page No. as index

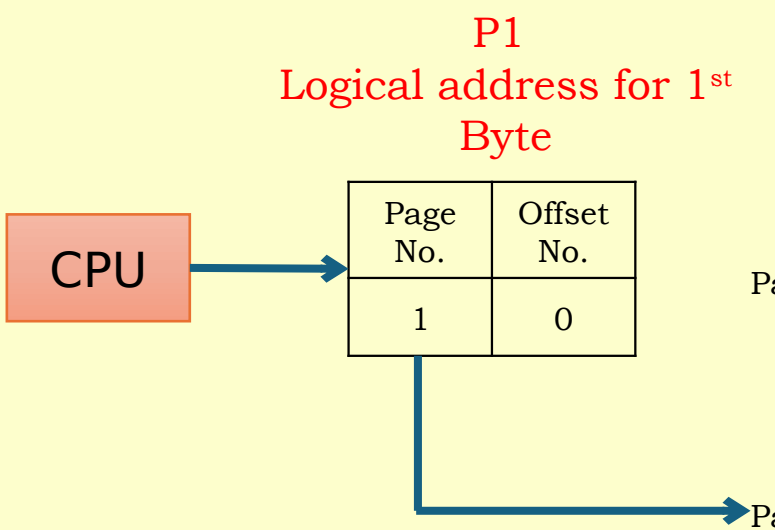


Page Table Structure

Frame No (3 Bits)			Present Bit (1 bit)	Protection Bits R/W/E	Dirty Bits (1 Bit)
0	0	0			
0	0	1			
0	1	0			
0	1	1	1	R/W = 1	0
1	0	0			
1	0	1			
1	1	0	1	R/W = 1	0
1	1	1			

Paging - Working – Address Translation/Mapping

- Extract the Frame No. from the lookup process in page table.

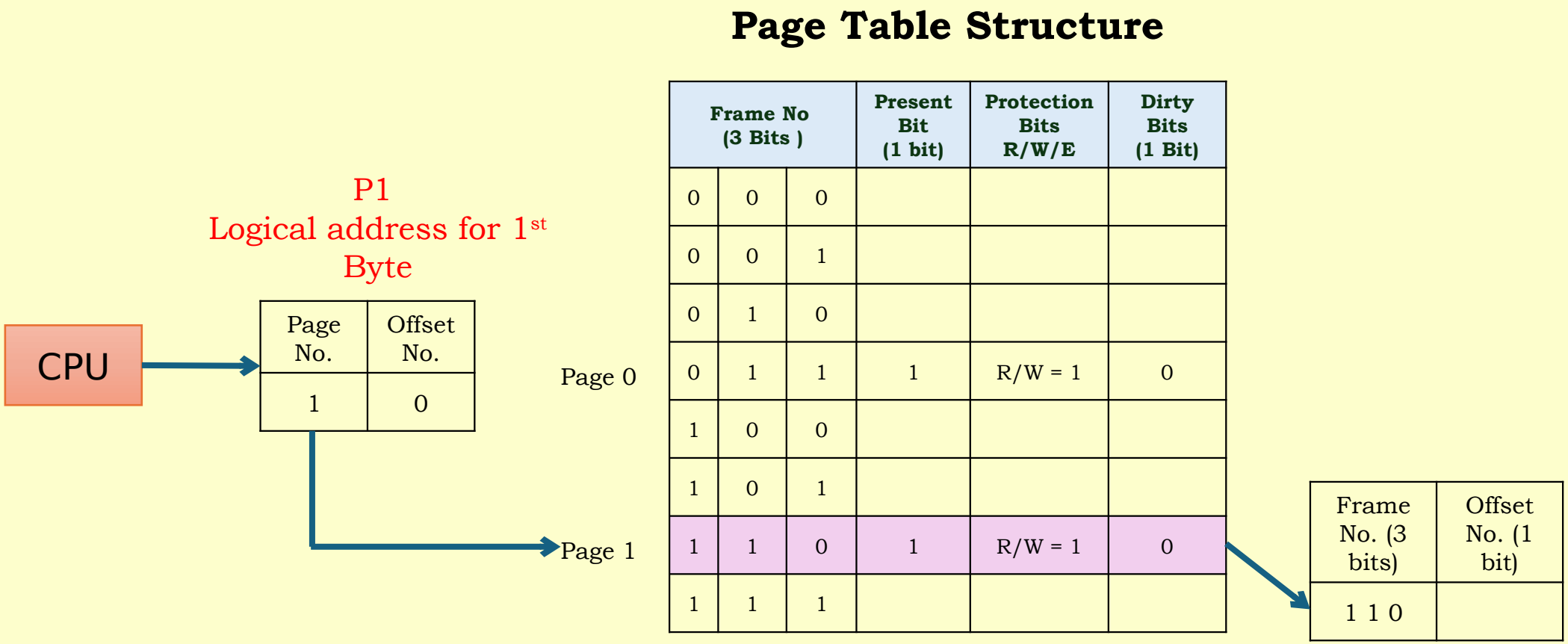


Page Table Structure

Frame No (3 Bits)			Present Bit (1 bit)	Protection Bits R/W/E	Dirty Bits (1 Bit)
0	0	0			
0	0	1			
0	1	0			
0	1	1	1	R/W = 1	0
1	0	0			
1	0	1			
1	1	0	1	R/W = 1	0
1	1	1			

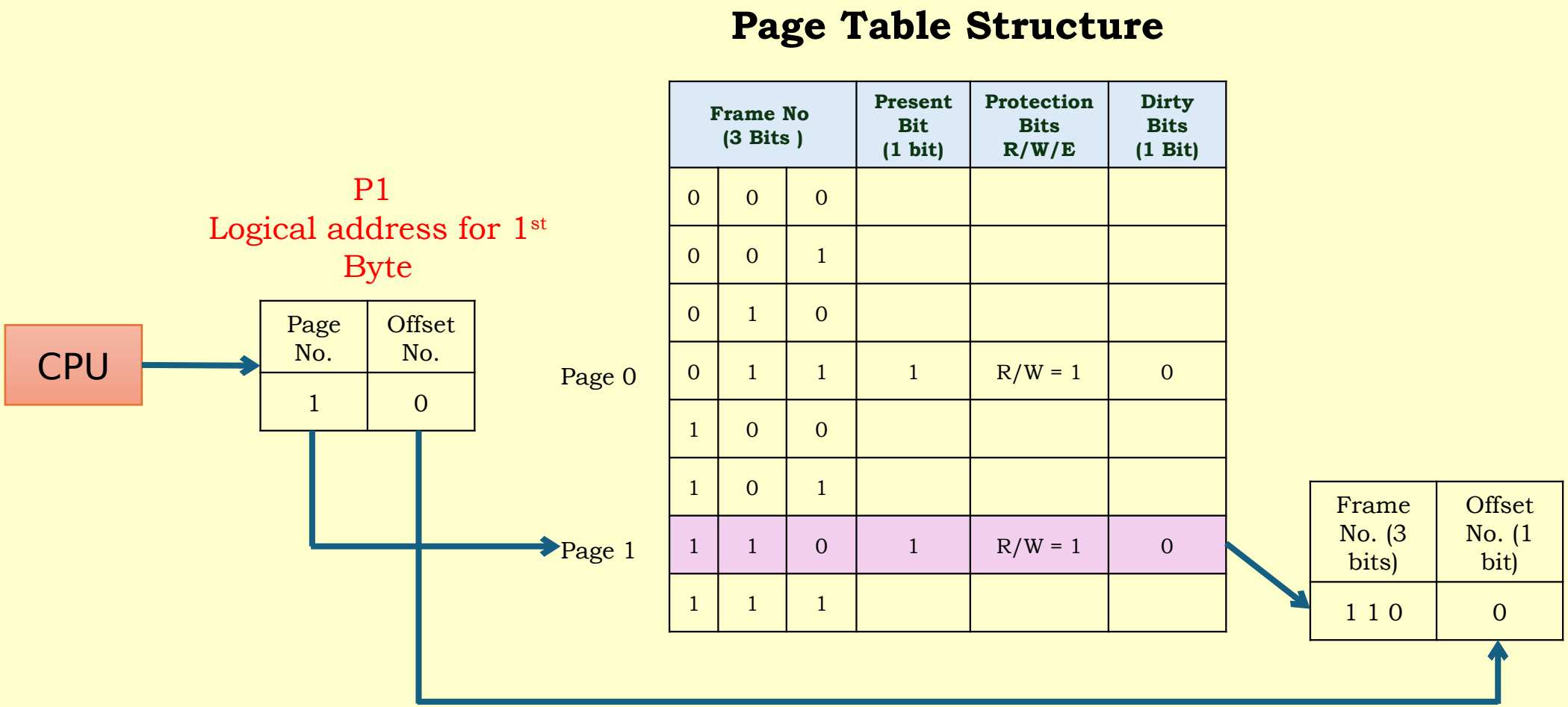
Paging - Working – Address Translation/Mapping

- Extract the Frame No. from the lookup process in page table.



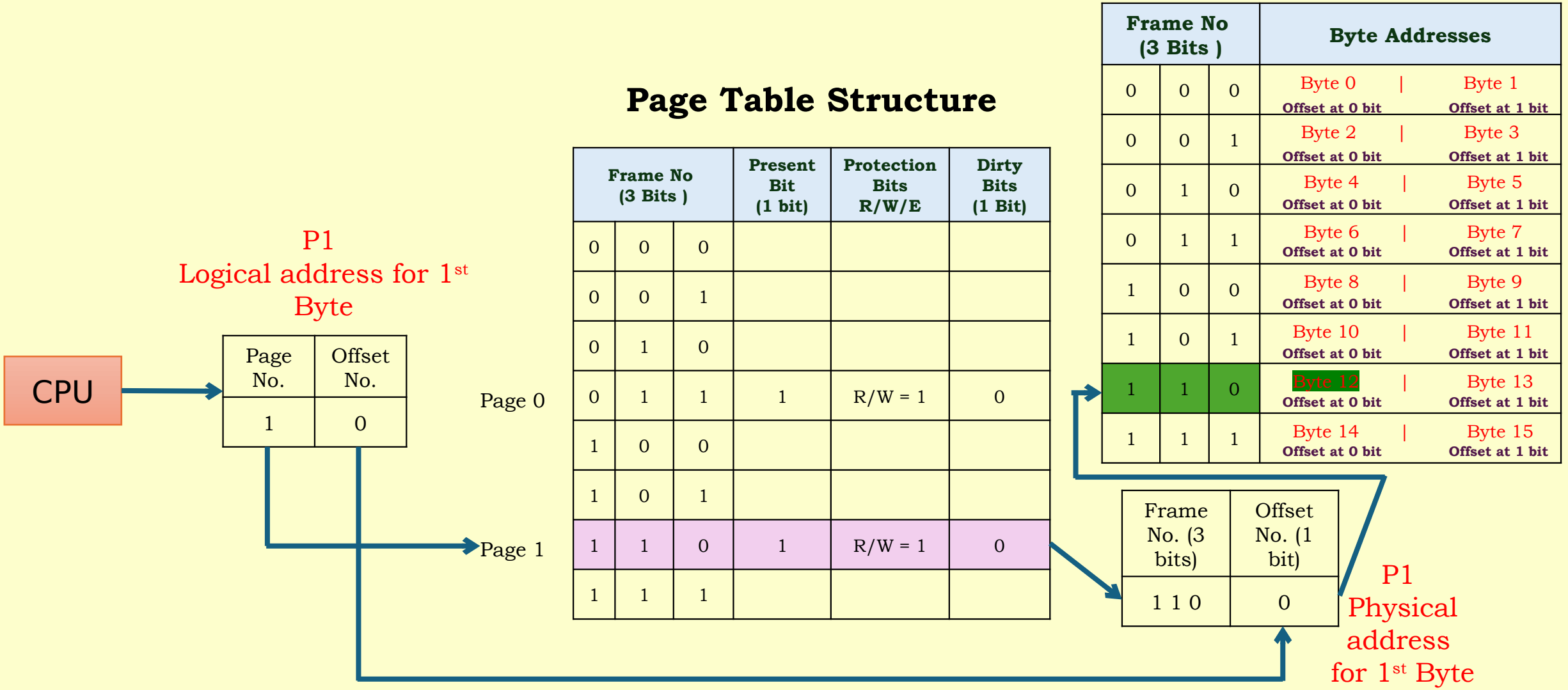
Paging - Working – Address Translation/Mapping

- Combine the offset with the extracted Frame no.



Paging - Working – Address Translation/Mapping

- Combine the offset with the extracted Frame no. and look in the RAM for the memory Location in terms of Byte which is **Byte 12**.



How to Implement Virtual Memory – Paging - Equations

- **For Main Memory**
  - Physical Address Space = Size of main memory
  - Size of main memory = Total number of frames x Page size
  - Frame size = Page size
  - If number of frames in main memory = 2^X, then number of bits in frame number = X bits
  - If Page size = 2^X Bytes, then number of bits in page offset = X bits
  - If size of main memory = 2^X Bytes, then number of bits in physical address = X bits

How to Implement Virtual Memory – Paging - Equations

- **For Process-**
  - Virtual Address Space = Size of process
  - Number of pages the process is divided = Process size / Page size
  - If process size = 2^X bytes, then number of bits in virtual address space = X bits

For Page Table-

- Size of page table = Number of entries in page table x Page table entry size
- Number of entries in pages table = Number of pages the process is divided
- Page table entry size = Number of bits in frame number + Number of bits used for optional fields if any

How to Implement Virtual Memory – Paging - Equations

In general, if the given address consists of ‘n’ bits, then using ‘n’ bits, 2^n locations are possible.

Then, size of memory = 2^n x Size of one location.

If the memory is byte-addressable, then size of one location = 1 byte.

Thus, size of memory = 2^n bytes.

If the memory is word-addressable where 1 word = m bytes, then size of one location = m bytes.

Thus, size of memory = $2^{n \times m}$ bytes.

How to Implement Virtual Memory – Paging - Equations

2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1K

2^{20}	1M
2^{30}	1G
2^{40}	1T

Paging - Problem

Given

Logical Address Space – 4GB,

Physical Address Space – 64 MB,

Page Size – 4 KB.

Find the

- No. of Pages,
- No. of Frames,
- No. of Entries in Page Table,
- Size of Page Table

Paging - Problem

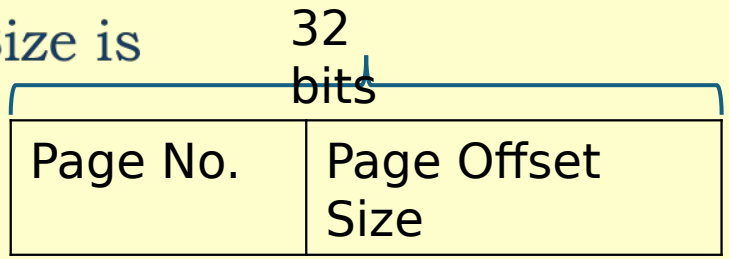
Soln.

Logical Address Space = Process Size

→4GB → $2^2 X 2^{30} = 2^{32}$Bytes

- If process size = 2^X bytes, then number of bits in virtual address space = X bits → 32bits

→So Logical Address Size is



Paging - Problem

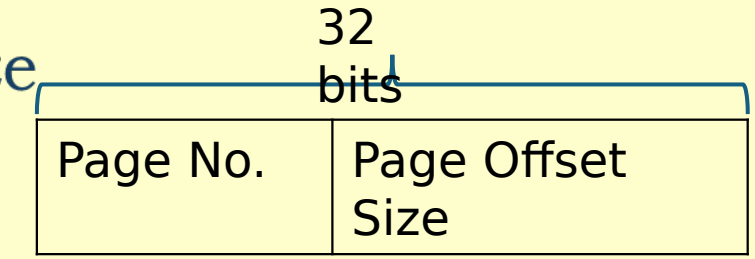
Given is **Logical Address Space – 4GB**, **Physical Address Space – 64 MB**, **Page Size – 4 KB**. Find the number Pages, No. of Frames, No. of Entries in Page Table, Size of Page Table.

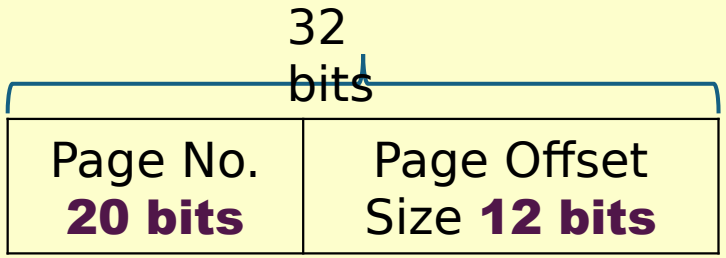
Soln.

Logical Address Space = Process Size = 4GB → represent it in bits

$$2^2 X 2^{30} = 2^{32}$$

So, Logical Address Size = 32 bits. So logical address generated by CPU will be of size





Page Size is 4 KB → represent it in bits → $2^2 X 2^{10} = 2^{12}$

Paging - Problem

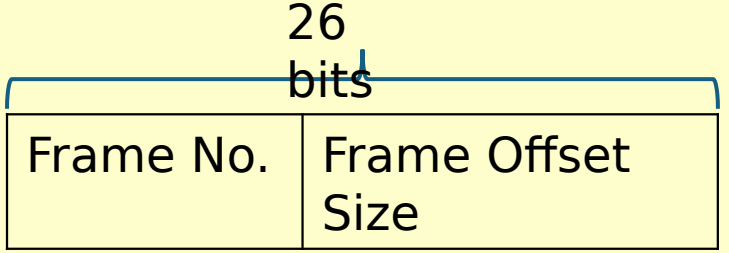
Given is **Logical Address Space – 4GB**, **Physical Address Space – 64 MB**, **Page Size – 4 KB**. Find the **number Pages**, No. of Frames, No. of Entries in Page Table, Size of Page Table.

Soln.

So the Page Offset size – 12 bits and Page no. is represented as 20 bits, The **number of pages** → 2^{20}

ii) Physical Address Space = 64MB = $2^6 X 2^{20} = 2^{26}$

So Physical Address Size = 26 Bits



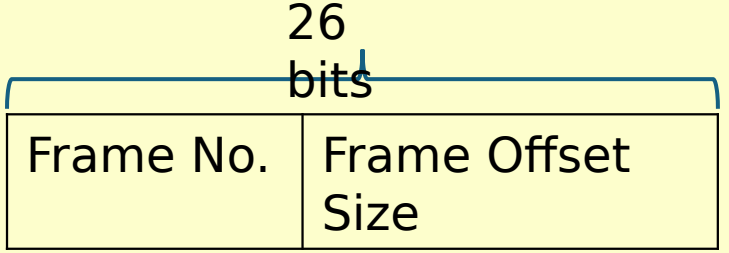
Paging - Problem

Given is **Logical Address Space – 4GB**, **Physical Address Space – 64 MB**, **Page Size – 4 KB**. Find the **number Pages**, No. of Frames, No. of Entries in Page Table, Size of Page Table.

Soln.

Frame Offset Size = Page Offset Size → 12 bits, Hence

Frame No. in bits will be 14 bits and **No. of Frames** → 2^{14}



Paging - Problem

Given is **Logical Address Space – 4GB**, **Physical Address Space – 64 MB**, **Page Size – 4 KB**. Find the **number Pages**, No. of Frames, No. of Entries in Page Table, Size of Page Table.

Soln.

iii) No. Page Table entries will be equal to No. of Pages in Process → 2^{20}

iv) Size of Page Table = No. of Pages X Frame size in bits → 2^{20} x 14

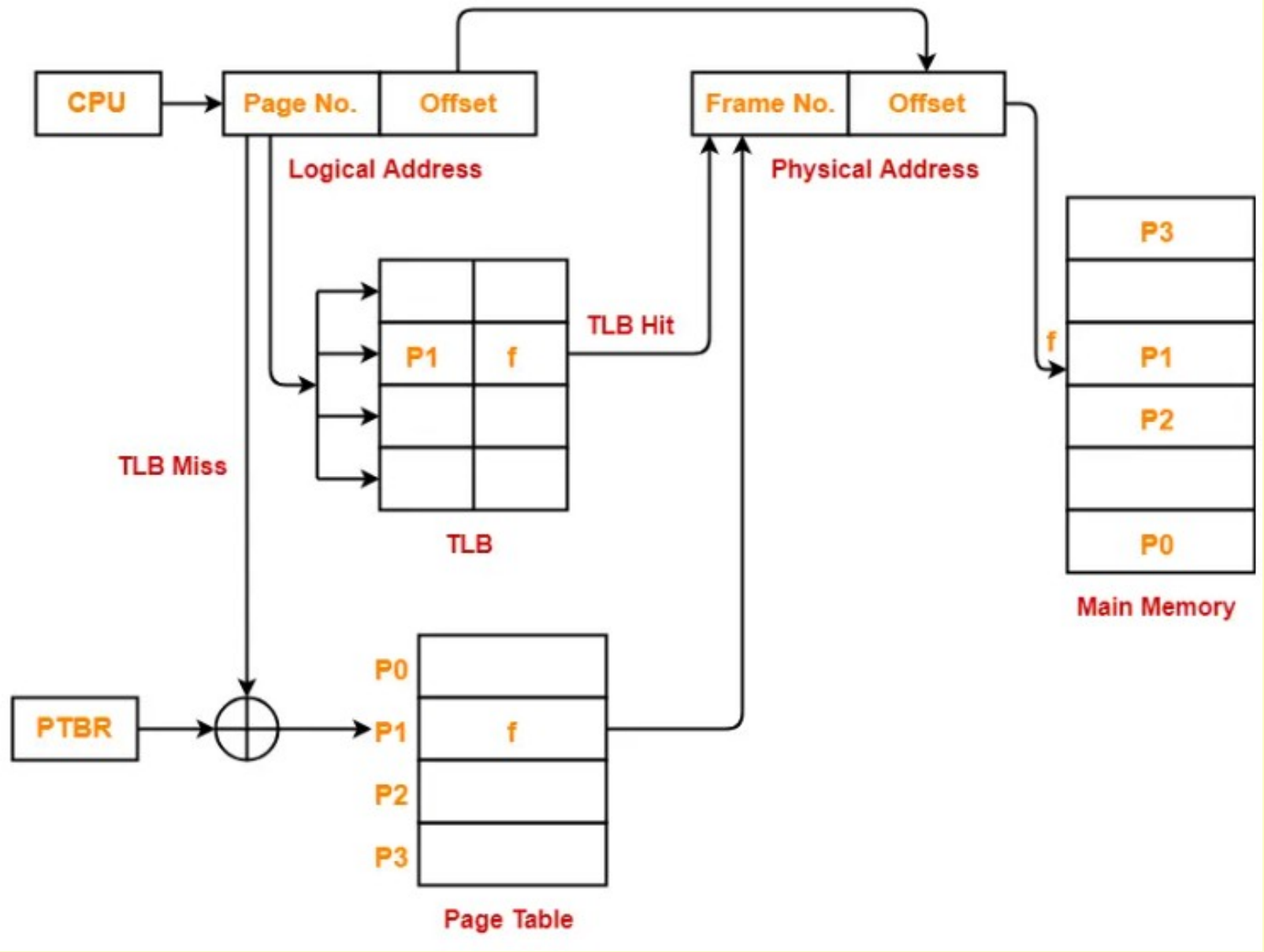
Paging – With Hardware (TLB) Support

- TLB is a high-speed cache that stores recent page table entries to speed up address translation

- Without TLB: Every memory access requires 2 memory accesses
  - Access page table entry
  - Access actual data

- With TLB: Most translations are cached
  - TLB Hit: Translation found in cache (fast)
  - TLB Miss: Must access page table (slower)

Paging – With Hardware (TLB) Support



- PTBR (Page Table Base Register) is a special hardware register that stores the base physical address of the current process's page table in main memory.
- It acts as a pointer that tells the Memory Management Unit (MMU) where to find the page table for address translation.

Paging – With Hardware (TLB) Support

- Effective Access Time for TLB =

Hit ratio of TLB x [Access time of TLB + Access time of RAM]

+

Miss ratio of TLB x [Access time of TLB + (2 x Access time of RAM)]

TLB Miss ratio = 1 – TLB Hit ratio

Paging – Multi-Level Paging

- Multi-level paging is a hierarchical memory management technique that organizes page tables into multiple levels instead of using a single flat page table.
- In this scheme, the page table is divided into a tree-like structure where:
  - Higher-level page tables contain pointers to lower-level page tables
  - Lower-level page tables eventually contain actual frame numbers
  - Page Table Base Register (PTBR) points to the outermost (first-level) page table
  - Address translation occurs in stages, with each level providing part of the translation

Paging – Multi-Level Paging – When Required?

- Consider a System with physical Address Space 32bits and Page Size of 4KB. Calculate the Page Table Size that needs to be stored in a Frame in main memory.
- Sol. Page Size → 4KB → $2^2 X 2^{10} = 2^{12}$ = 12 bits is the offset size offset

  Total Physical Address Size → 32 bits (Frame No. + Offset → Physical Addrs)

  Total Frame No. Size → 32 bits – 12 bits → 20 bits

  Page Size = Frame Size → 20 bits is the page size .

  Page Entry size → 2^{20} → 4 bits are added to 20 to add control bits → 24 bits/Entry

  Total Page Table Size → 2^{20} x 24 → 3MB

  Thus the size of Frame is 4KB and Table Size is greater than it.

The issue is 3MB page table cannot fit in a single 4KB frame

Multi-level paging becomes necessary when page table size exceeds frame size.

Paging – Multi-Level Paging – Problem

- Consider a system using paging scheme where-
  - Logical Address Space = 4 GB, Physical Address Space = 16 TB, Page size = 4 KB
- Find how many levels of page table will be required.
- Soln.

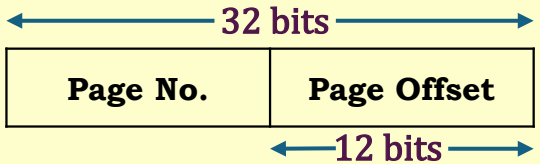
No. of bits in Logical Address = 4GB → $2^2 X 2^{30} = 2^{32}$ → 32 bits Logical Address Size

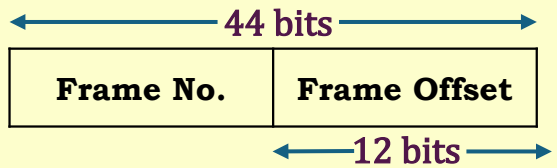
No. of Pages = 4GB / 4KB → $2^{32} / 2^{12} = 2^{20}$

No. of bits in Physical Address = 16TB → $2^4 X\ 2^{40} = 2^{44}$ → 44 bits Physical Address

No. of Frames = 16TB / 4KB → $2^{44} / 2^{12} = 2^{32}$ → 32 bits of Frame number

No. of bits in Page Offset = Page Size → 2^{12} → 12 bits





Check → Number of pages the process is divided x Number of bits in frame number → 2^{20} X 32 bits = 4MB. Thus size page table is greater than the frame size (4 KB).

Paging – Multi-Level Paging – Problem

- Number of pages the inner page table is divided = Inner page table size / Page size

  4GB / 4KB → 2^{20} / 2^{10} → 2^{10} Inner Page table

- Number of page table entries in one page of inner page table = Page size / Page table entry size = Page size / Number of bits in frame number = 4KB / 32 bits → 2^{10}
- Thus Number of Bits Required to Search an Entry in One Page of Inner Page Table → 10 bits
- Outer page table is required to keep track of the frames storing the pages of inner page table.
- Outer Page table size = Number of entries in outer page table x Page table entry size = Number of pages the inner page table is divided x Number of bits in frame number → 2^{10} X 32 bits → 2^{10} x 4 bytes → 4 KB

Paging – Multi-Level Paging – Problem

- Now, we can observe-
  - The size of outer page table is same as frame size (4 KB).
  - Thus, outer page table can be stored in a single frame.
  - So, for given system, we will have two levels of page table.
  - Page Table Base Register (PTBR) will store the base address of the outer page table.

Demand Paging

- Demand paging is a memory management technique used in modern operating systems where pages of a process are only loaded into main memory when they are actually needed (on demand), rather than loading the entire process at once.
  - When a program starts, no or few pages are loaded initially.
  - If the program tries to access a page not in memory, a page fault occurs.
  - The operating system:
    - Pauses the program,
    - Locates the required page on disk,
    - Loads it into a free memory frame,
    - Updates the page table to reflect its new location.
  - The CPU resumes running the program as if the page had always been in memory.

Demand Paging – Why?

- Allows systems to run programs larger than physical memory.
- Optimizes memory usage by keeping only frequently used pages in RAM.
- Is transparent to user programs—handled entirely by the OS.

- **Page Fault**: Happens when a requested page isn't in main memory.
- **Secondary Storage**: Where non-resident pages are stored (e.g., disk).
- **Page Replacement**: If no free memory frame is available, the OS uses a page replacement algorithm to decide which existing page to evict.

Page Fault & Page Replacement

- A page fault occurs when a program accesses a memory page that is not currently present in physical memory (RAM) and must be loaded from secondary storage (like a hard disk or SSD).
- The memory management unit (MMU) detects this condition and interrupts the program so the operating system (OS) can handle it.
- Once the required page is fetched into RAM, the process resumes as if the page had always been present.

Page Fault & Page Replacement - Working

- The MMU detects access to a missing page and raises an exception.
- The OS checks if the access is valid. If not, the process may be terminated; if valid, the OS finds a free page frame in memory.
- If no free frames are available, the OS must use a page replacement algorithm to choose a "victim" page to evict.
- If the victim page has been modified, it's written back to disk (swap/page file).
- The required page is loaded from secondary storage into the frame.
- The page tables are updated and program execution continues.

Page Replacement - Working

- When main memory is full, the OS must decide which page to replace to make space for the new page. The page replacement policy directly affects system performance by influencing the page fault rate.
  - Select a victim page using an algorithm.
  - Evict the victim from RAM.
  - Load the required new page in its place.

Page Replacement - Working

- When main memory is full, the OS must decide which page to replace to make space for the new page. The page replacement policy directly affects system performance by influencing the page fault rate.
  - Select a victim page using an algorithm.
  - Evict the victim from RAM.
  - Load the required new page in its place.

Algorithm	How It Works	Strengths	Weaknesses
FIFO	Removes the oldest page in memory	Simple	Suffers from Belady's anomaly
LRU (Least Recently Used)	Removes page unused for longest time	Near-optimal	Needs tracking/history mechanism
Optimal	Removes page not needed for the longest future time	Lowest faults	Requires future knowledge
MFU/MRU	Most/Most Frequently/Recently Used	Tunable	May not match real usage patterns

Page Replacement - Working