Operating System

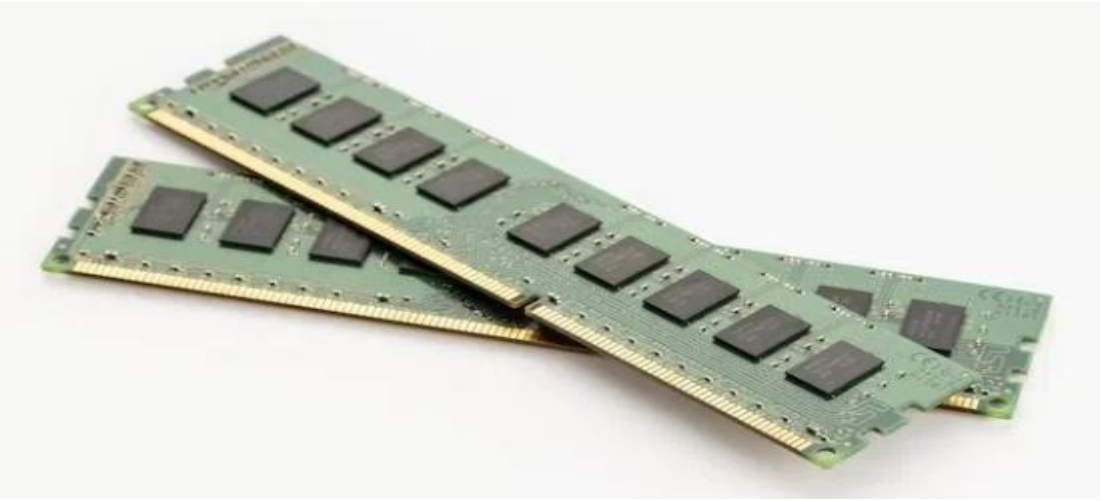
Unit – 5
(Part-A)
Memory Management

Mayank Mishra
School of Electronics Engineering
KIIT-Deemed to be University

Background

- The main purpose of computer system is to execute programs.

- During execution, these programs, together with the data they access, must be in **Main Memory** (at least partially).



RAM

- In CPU Scheduling we showed how the CPU can be shared by a set of processes

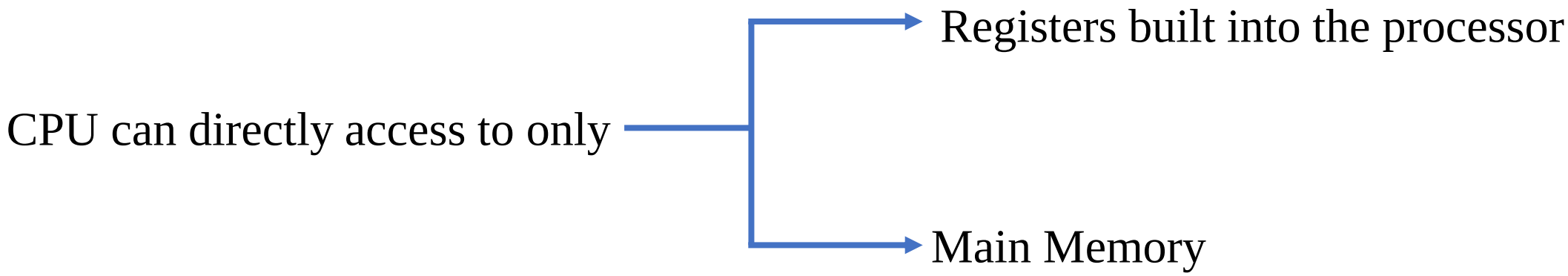
As a result of this

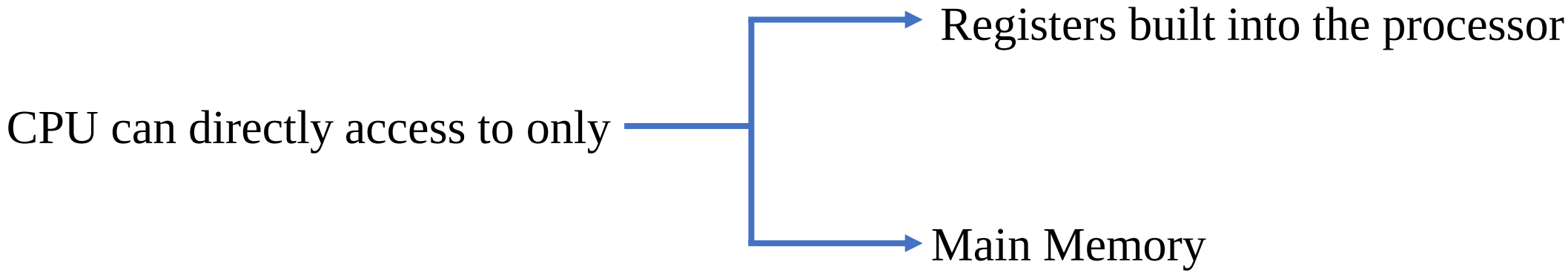
CPU Utilization, as well as **speed of computer's response to its users** could be improved

To realize this increase in performance, however, we must keep several processes in memory, that is, we must share memory.

Basic Hardware

- Memory consists of a large array of words or bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.





- There are machine instructions that take **memory addresses** as arguments, but none that take **disk addresses**.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.

- Registers that are built into each CPU core are generally accessible **within one cycle of the CPU clock**. Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take **many cycles of the CPU clock**.
- In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses.
- The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. **Cache, a memory buffer used to accommodate a speed differential.**

- The OS has to be protected from access by user processes.
- In addition, user processes must be protected from one another.
- This protection must be provided by the hardware.

How this can be done?

Ensure that each process has a separate memory space

To do this , we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses

For this we have two registers- BASE and LIMIT

- The base register holds the smallest legal physical memory address.
- The limit register specifies the size of the range.
- For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

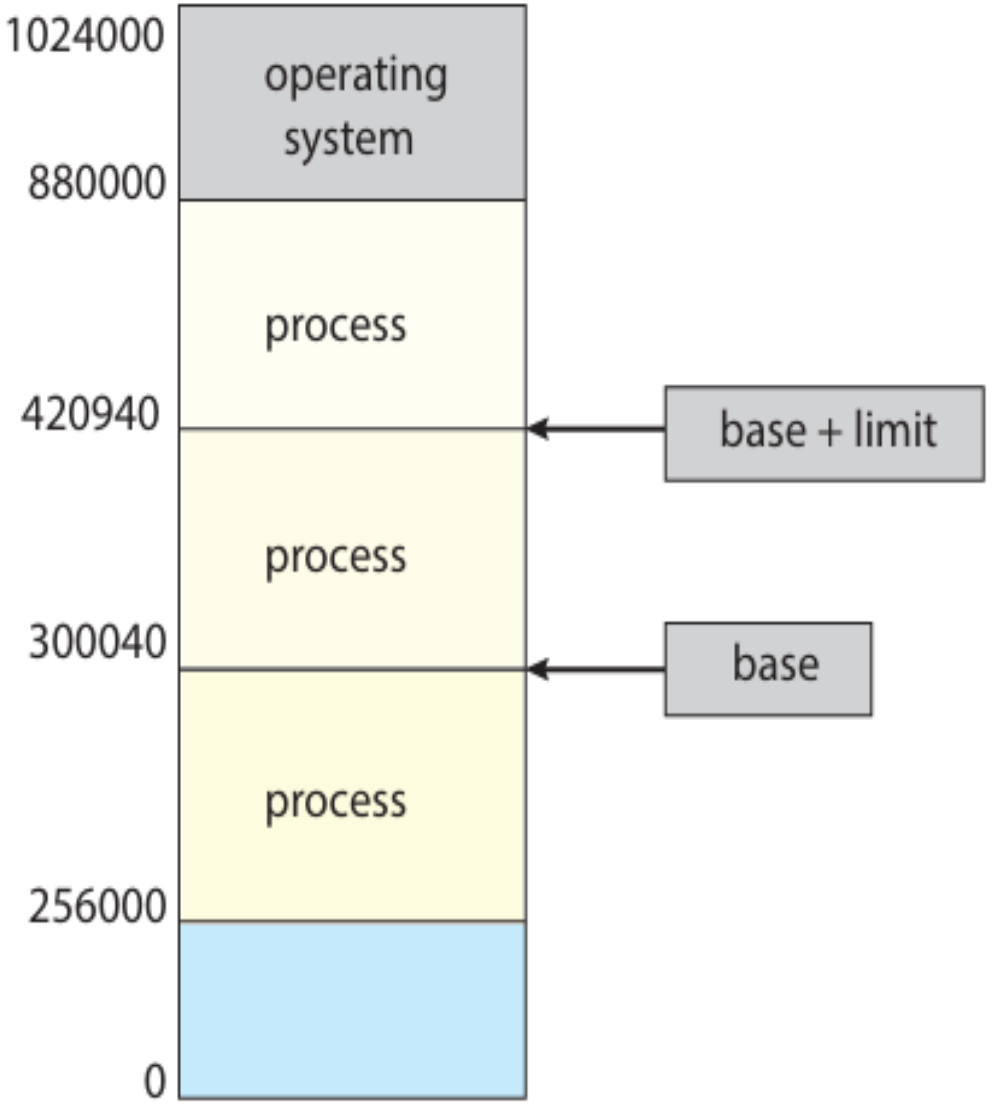


Fig. A base and a limit register define a logical address space

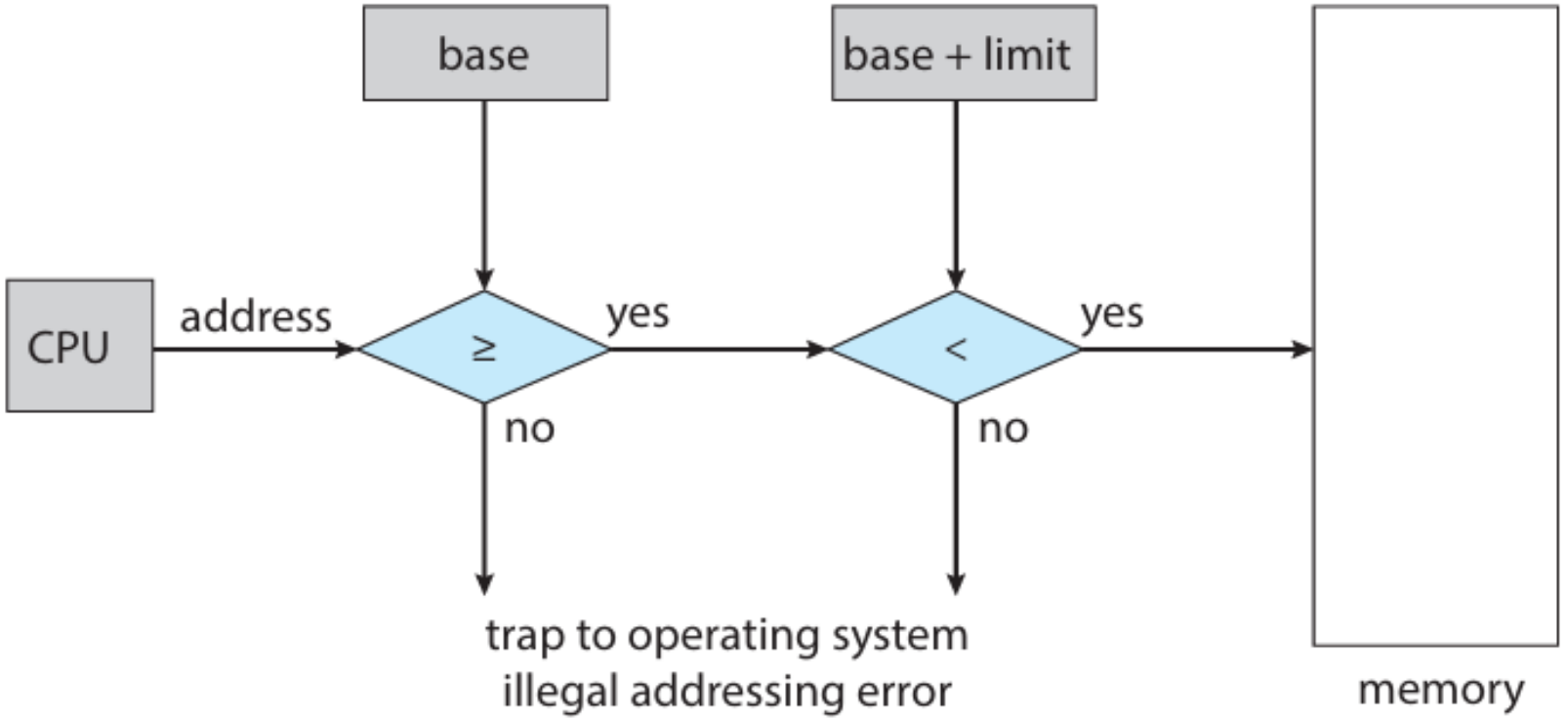
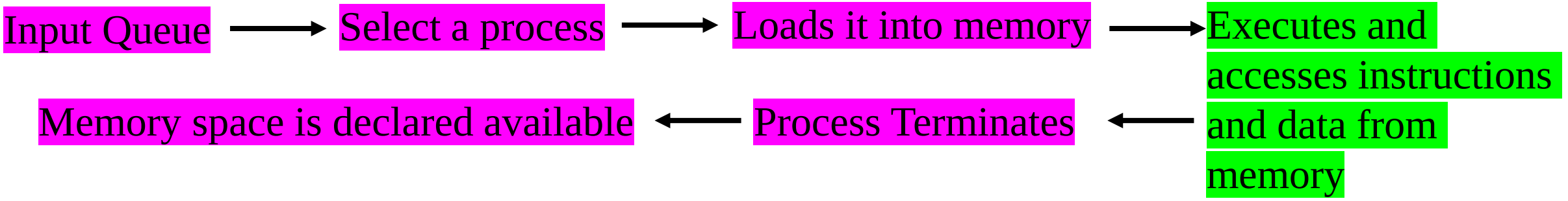


Fig. Hardware address protection with base and limit registers.

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
- Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.
- This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Usually, a program resides on a disk as a binary executable file.
- To run, the program must be brought into memory and placed within the context of a process, where it becomes eligible for execution on an available CPU.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The process on the disk that are waiting to be brought into memory for execution from the input queue.



Address Binding

- Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000.
- In most cases, a user program will go through several steps during **COMPILE TIME, LOAD TIME, EXECUTION TIME** before being executed *(Addresses may be represented in different ways during these steps).*
- Addresses in the source program are generally **symbolic** (such as the variable **count**).
- A compiler typically binds these symbolic addresses to **relocatable addresses** (such as “14 bytes from the beginning of this module”).
- The linker or loader in turn binds the **relocatable addresses to absolute addresses** (such as 74014). Each binding is a mapping from one address space to another.

There are three primary types of address binding:

Compile Time:

- If you know at compile time where the process will reside in memory, then absolute code can be generated.
- For example, if you know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there.
- If, at some later time, the starting location changes, then it will be necessary to recompile this code.
- The program will only run correctly if it is loaded into the same memory location where it was compiled.
- This is simple but lacks flexibility since it requires the program to be loaded into the same memory every time.

Load Time:

- If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
- In this case, final binding is delayed until load time.
- If the starting address changes, we need only reload the user code to incorporate this changed value.
- This type of binding occurs when the program is loaded into memory, which means the memory locations are determined during the loading phase, rather than during compilation.
- It provides some flexibility, as the program can be loaded into different memory locations, but the binding still happens before execution starts.
- The operating system loader adjusts the addresses to the specific memory location where the program is loaded.

Execution Time:

- If the process can be moved during its execution from one memory segment to another, then binding must be delayed until runtime.

- This is the most flexible form of address binding. It occurs during the execution of the program and is typically done by the memory management unit (MMU) in the CPU.

- The program can be relocated while running, allowing it to move to different memory locations dynamically.

- During program execution, address binding is handled by runtime binding, sometimes referred to as dynamic binding. More flexibility is possible with this kind of binding since memory addresses can be dynamically allocated and deallocated as needed.

Why Address Binding is Important:

- **Memory Protection**: It prevents one process from accessing the memory space of another process.
- **Efficiency**: It allows programs to be relocated in memory without modifying their code.
- **Security**: It makes it harder for malicious programs to interfere with others because each process is isolated in its address space.
- Address binding is a crucial part of how operating systems manage memory and ensure that programs run correctly and efficiently.

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

- Binding addresses at either compile or load time generates identical logical and physical addresses.

- The set of all logical addresses generated by a program is a **logical address space**.

- The set of all physical addresses corresponding to these logical addresses is a physical address space. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

Logical Versus Physical Address Space (Contd.)

- The logical address space is a part of the virtual memory. When the program runs, the operating system translates logical addresses (generated by the CPU) into physical addresses using a process known as **address translation**.

- The size of the logical address space is determined by the number of bits in the program's address register (e.g., 32-bit or 64-bit). For a 32-bit system, the logical address space is 4 GB (2^32), and for a 64-bit system, it is much larger (2^64).

- The physical address space consists of addresses that are directly mapped to locations in the physical RAM.

- The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

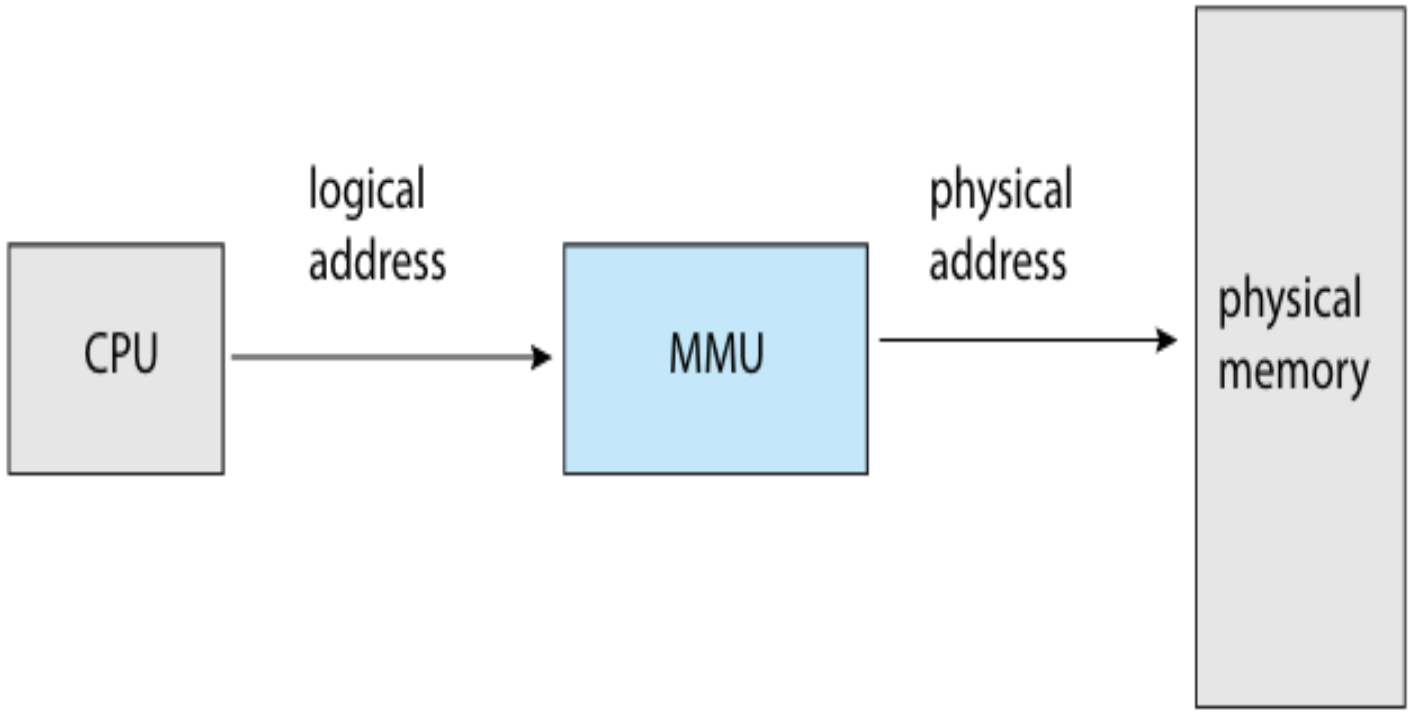


Fig. Memory management unit (MMU).

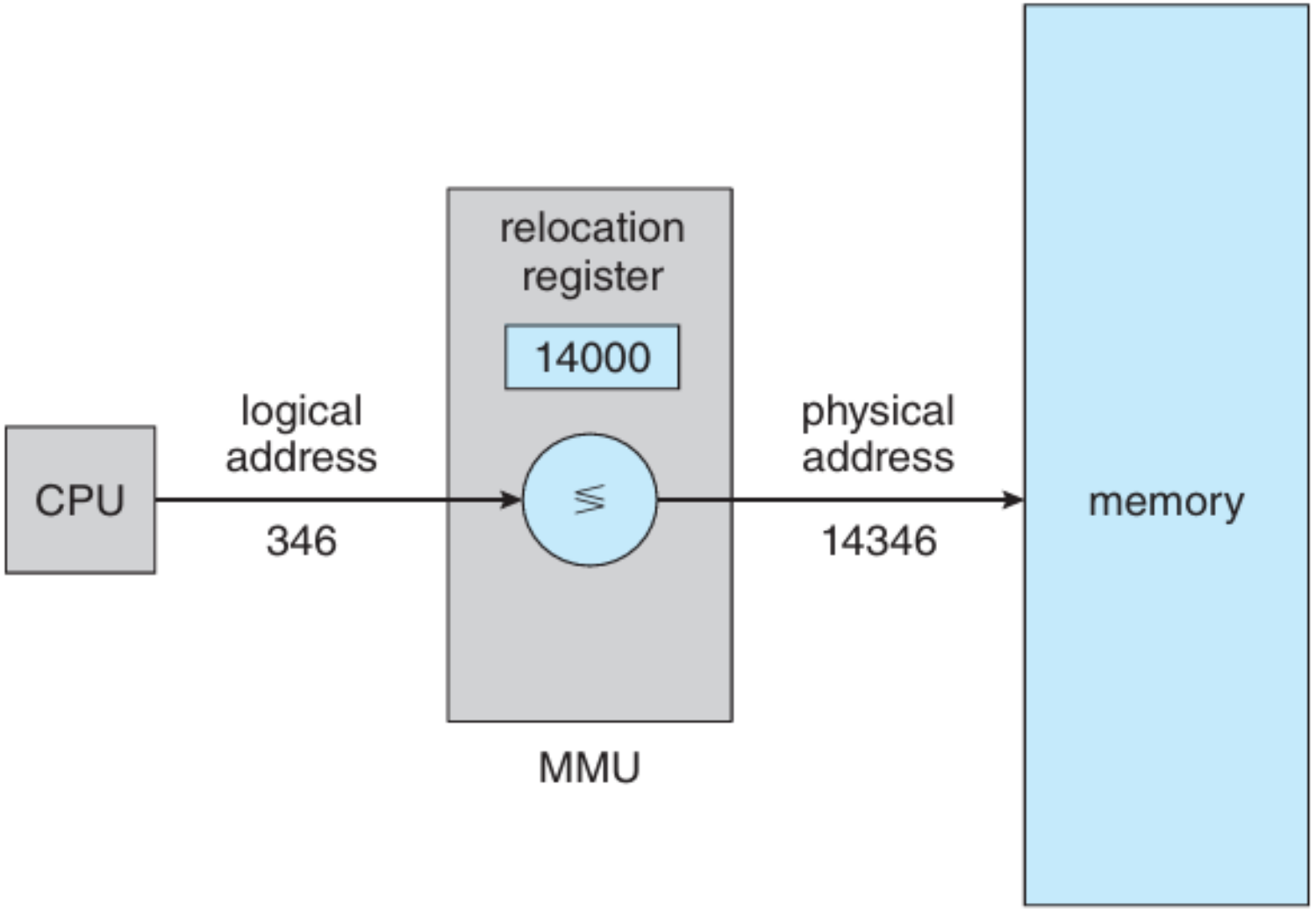


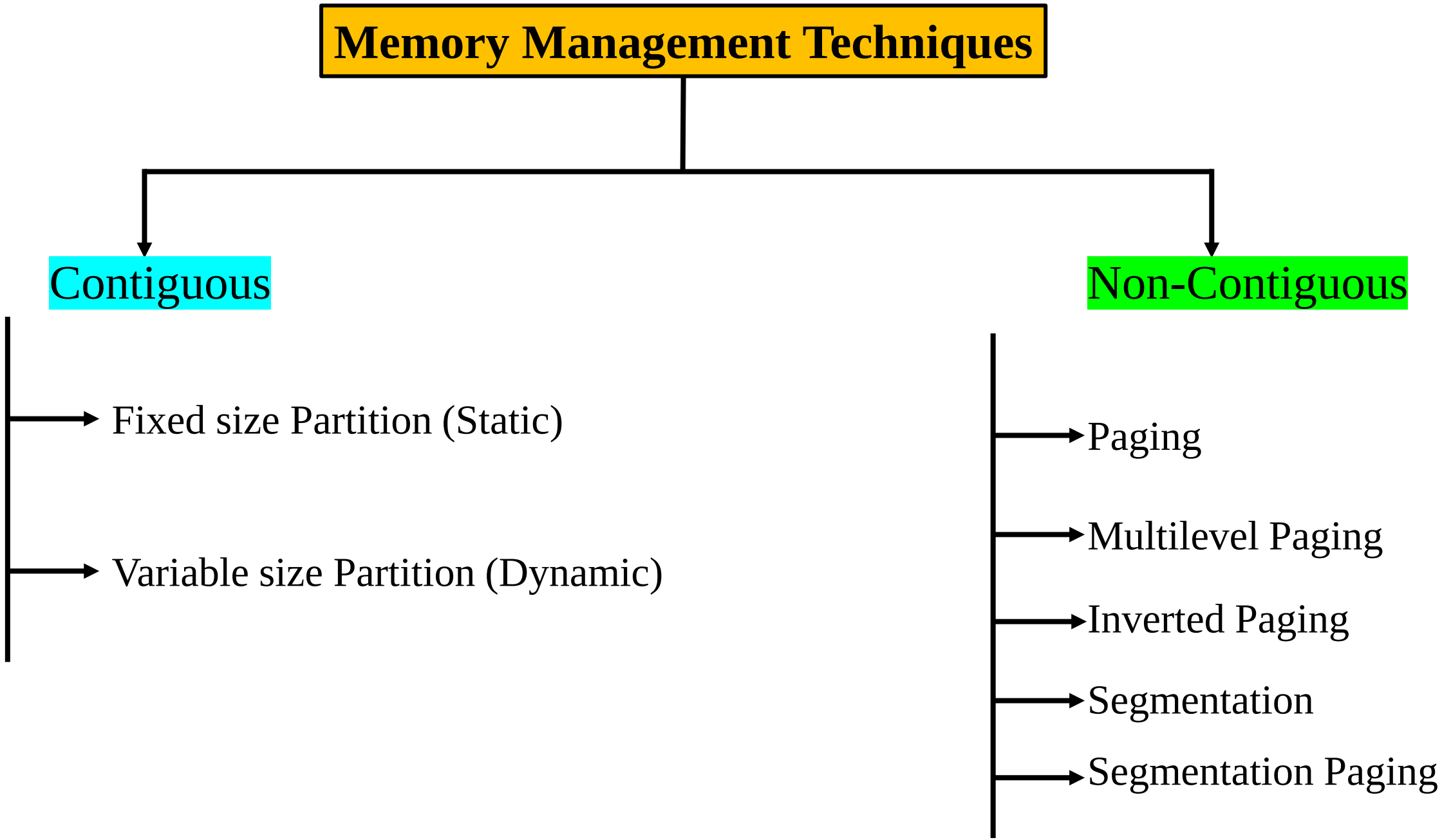
Fig. Memory management unit (MMU).

- We now have two different types of addresses: **logical addresses** (in the range *0 to max*) and **physical addresses** (in the range **R + 0** to **R + max** for a base value **R**).

- The user program generates only logical addresses and thinks that the process runs in memory locations from **0 to max**.

- However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Parameter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.
Editable	Logical address can be change.	Physical address will not change.
Also called	virtual address.	real address.


Memory Management Techniques
Contiguous
Fixed size Partition (Static)
Variable size Partition (Dynamic)
Non-Contiguous
Paging
Multilevel Paging
Inverted Paging
Segmentation
Segmentation Paging

Fixed size Partitioning

- Fixed (or static) partitioning is one of the earliest and simplest memory management techniques used in operating systems.

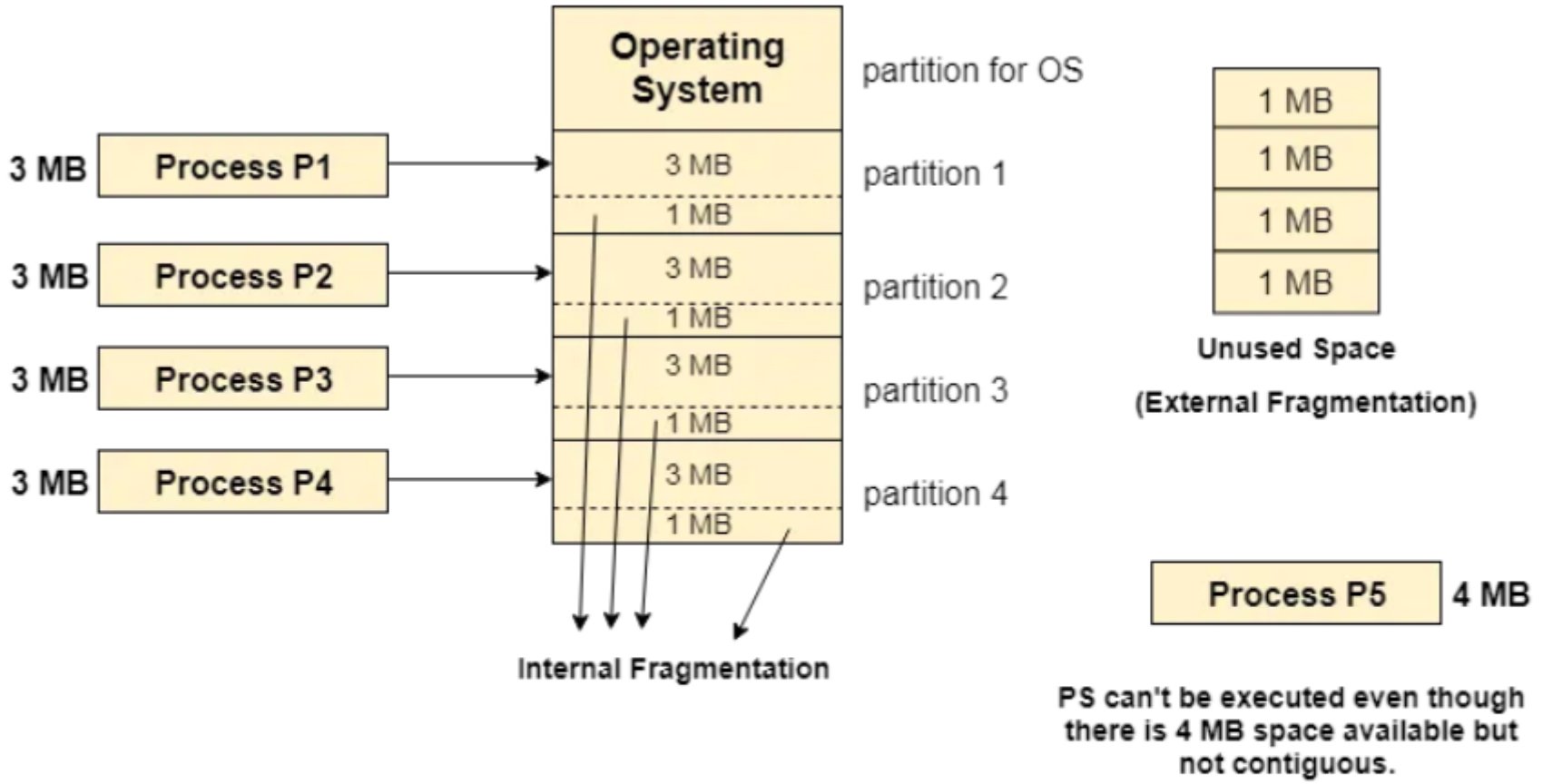
- It involves dividing the main memory into a fixed number of partitions at system startup, with each partition being assigned to a process. These partitions remain unchanged throughout the system's operation, providing each process with a designated memory space.

- This is the oldest and simplest technique used to put more than one process in the main memory. In this partitioning, the **number of partitions** (non-overlapping) in RAM is **fixed** but the **size** of each partition **may or may not be the same**.

- As it is a **contiguous** allocation, hence no spanning is allowed (Each process must fit into a single partition. A process cannot span across multiple partitions. If a process is too large to fit into a partition, it cannot be loaded into memory, causing delays or requiring the system to reject or swap out processes.)

Types of Fixed Partitioning:

1.Equal-sized Fixed Partitioning:



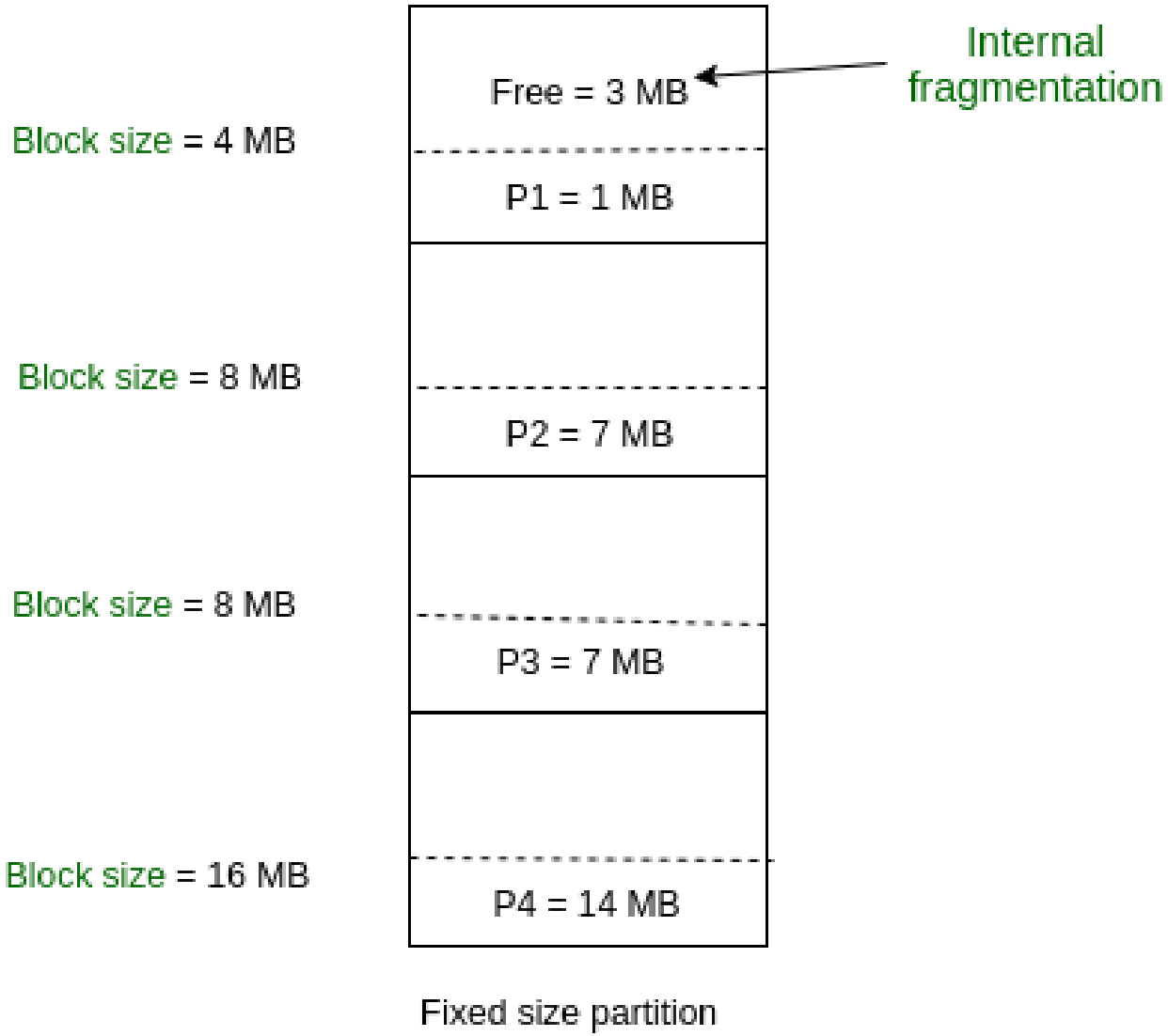
Fixed Partitioning

(Contiguous memory allocation)

1.Equal-sized Fixed Partitioning (Contd.):

- In this approach, all partitions are of equal size. This is the simplest form of fixed partitioning, where the total memory is divided into equal parts.
- While this method is simple to implement, it can lead to inefficient memory usage (internal fragmentation) if processes are smaller than the partition size.
- It can lead to External Fragmentation.

2. Unequal-sized Fixed Partitioning:



- As illustrated in the figure, first process is only consuming 1MB out of 4MB in the main memory.
- Hence, Internal Fragmentation in first block is (4-1) = 3MB.
- Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.
- Suppose process P5 of size 7MB comes. But this process cannot be accommodated in spite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.

2.Unequal-sized Fixed Partitioning (Contd.):

- In this method, different partitions are created with different sizes.
- This approach allows for better memory utilization than equal-sized partitioning, but it can still lead to fragmentation (Internal as well as External).

Advantages of Fixed Partitioning

- **Easy to implement:** The algorithms required are simple and straightforward.
- **Low overhead:** Requires minimal system resources to manage, ideal for resource-constrained systems.
- **Predictable**: Memory allocation is predictable, with each process receiving a fixed partition.
- **Suitable for systems with a fixed number of processes:** Ideal for systems where the number of processes and their memory requirements are known in advance.
- **Easy to debug:** Fixed Partitioning is easy to debug since the size and location of each process are predetermined.

Advantages of Fixed Partitioning (Contd.)

- **Prevents process interference:** Ensures that processes do not interfere with each other's memory, improving system stability.
- **Efficient memory use:** Particularly in systems with fixed, known processes and batch processing scenarios.
- **Good for batch processing:** Works well in environments where the number of processes remains constant over time.
- **Better control over memory allocation:** The operating system has clear control over how memory is allocated and managed.

Disadvantages of Fixed Partitioning

- **Internal Fragmentation:** Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

- **Limit process size:** Process of size greater than the size of the partition in Main Memory cannot be accommodated. The partition size cannot be varied according to the size of the incoming process size. Hence, the process size of 32MB in the above-stated example is invalid.

- **Limitation on Degree of Multiprogramming:** Partitions in Main Memory are made before execution or during system configure. Main Memory is divided into a fixed number of partitions. Number of processes greater than the number of partitions in RAM is invalid in Fixed Partitioning.

- **External Fragmentation**

Fixed size Partitioning

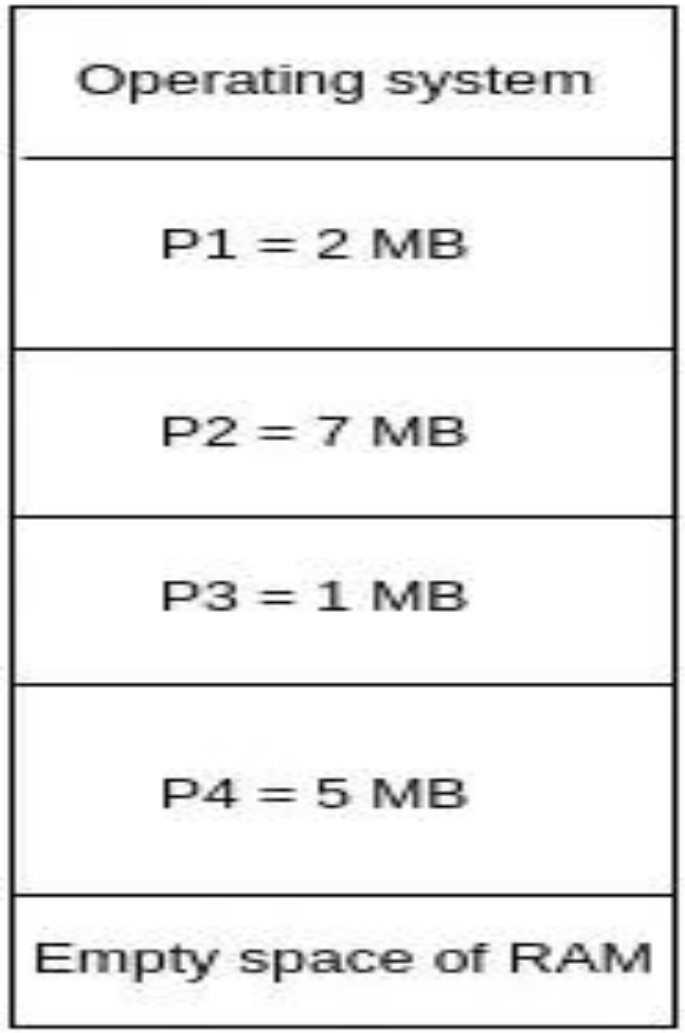
Variable size Partitioning

Variable size Partitioning

- It is a part of the Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning.

- In contrast with fixed partitioning, partitions are not made before the execution or during system configuration. Various **features** associated with variable Partitioning-
  - Initially, RAM is empty and partitions are made during the run-time according to the process's need instead of partitioning during system configuration.
  - The size of the partition will be equal to the incoming process.
  - The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM.
  - The number of partitions in RAM is not fixed and depends on the number of incoming processes and the Main Memory's size.

Dynamic partitioning



Block size = 2 MB

Block size = 7 MB

Block size = 1 MB

Block size = 5 MB

Partition size = process size
So, no internal Fragmentation

Variable size Partitioning (Contd.)

- It is a part of the Contiguous allocation technique. It is used to alleviate the problem faced by Fixed Partitioning.

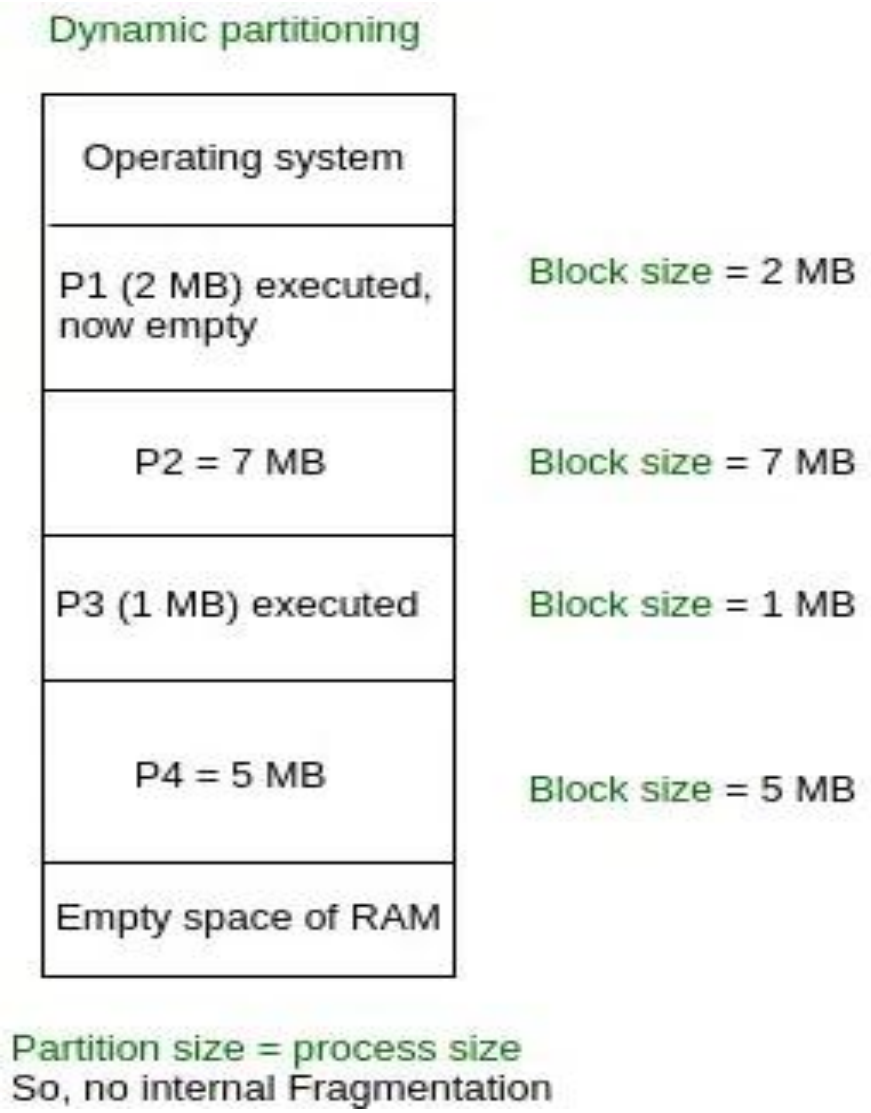
- In contrast with fixed partitioning, partitions are not made before the execution or during system configuration. Various **features** associated with variable Partitioning-
  - Initially, RAM is empty and partitions are made during the run-time according to the process's need instead of partitioning during system configuration.
  - The size of the partition will be equal to the incoming process.
  - The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM.
  - The number of partitions in RAM is not fixed and depends on the number of incoming processes and the Main Memory's size.

Advantages of Variable(Dynamic) Partitioning:

- **No Internal Fragmentation:** In variable Partitioning, space in the main memory is allocated strictly according to the need of the process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.
- **No restriction on the Degree of Multiprogramming:** More processes can be accommodated due to the absence of internal fragmentation. A process can be loaded until the memory is empty.
- **No Limitation on the Size of the Process:** In Fixed partitioning, the process with a size greater than the size of the largest partition could not be loaded and the process can not be divided as it is invalid in the contiguous allocation technique. Here, In variable partitioning, the process size can’t be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning

- **External Fragmentation:** There will be external fragmentation despite the absence of internal fragmentation.
  - For example, suppose in the above example- process P1(2MB) and process P3(1MB) completed their execution. Hence two spaces are left i.e. 2MB and 1MB. Let's suppose process P5 of size 3MB comes. The space in memory cannot be allocated as no spanning is allowed in contiguous allocation. The rule says that the process must be continuously present in the main memory to get executed. Hence it results in External Fragmentation.



#Holes may be found

Disadvantages of Variable Partitioning

- **Complexity:** Fixed partitioning and its related concepts are far simpler to handle and control as compared to variable partitions and external fragmentation.
- **Need for Compaction:** To avoid this fragmentation, compaction is needed, in which the free areas are grouped together to form bigger spaces in the memory, but this takes a lot of time and consumes many resources.

Fixed Partitioning	Variable Partitioning
Memory is divided into fixed-sized partitions.	Memory is allocated dynamically in varying sizes.
Only one process can be placed in each partition.	A process is allocated a chunk of free memory as needed.
Inefficient memory utilization due to internal fragmentation.	More efficient memory utilization with less internal fragmentation.
Internal and external fragmentation occur.	Only external fragmentation occurs.
Limited degree of multi-programming.	Higher degree of multi-programming due to flexible allocation.
Easier to implement.	More complex to implement.
Restricts process size based on partition size.	No size limitation on processes.

Dynamic Memory Allocation Strategies

- The **First Fit**, **Next Fit**, **Best Fit**, and **Worst Fit** are all **dynamic memory allocation strategies** used to allocate memory blocks to processes in a **memory management system**.

- These strategies determine how a process will be assigned to a free block of memory in a system with free memory partitions.

First Fit

- The first-fit algorithm searches for the first free partition that is large enough to accommodate the process.

- The operating system starts searching from the beginning of the memory and allocates the first free partition that is large enough to fit the process.

Next Fit

- It is same as the First Fit but it start the search always from last allocated hole.

Best Fit

- The best-fit algorithm searches for the smallest free partition that is large enough to accommodate the process.

- The operating system searches the entire memory and selects the free partition that is closest in size to the process.

Worst Fit

- The worst-fit algorithm searches for the largest free partition and allocates the process to it.

- This algorithm is designed to leave the largest possible free partition for future use.

Advantages and Disadvantages

- First Fit is fast and simple to implement, making it the most commonly used algorithm. However, it can suffer from external fragmentation, where small free partitions are left between allocated partitions.

- Best Fit reduces external fragmentation by allocating processes to the smallest free partition, but it requires more time to search for the appropriate partition.

- Worst Fit reduces external fragmentation by leaving the largest free partition, but it can lead to inefficient use of memory.

Question1:

Request from the processes is 300k, 25k, 125k, and 50k , respectively (in order).

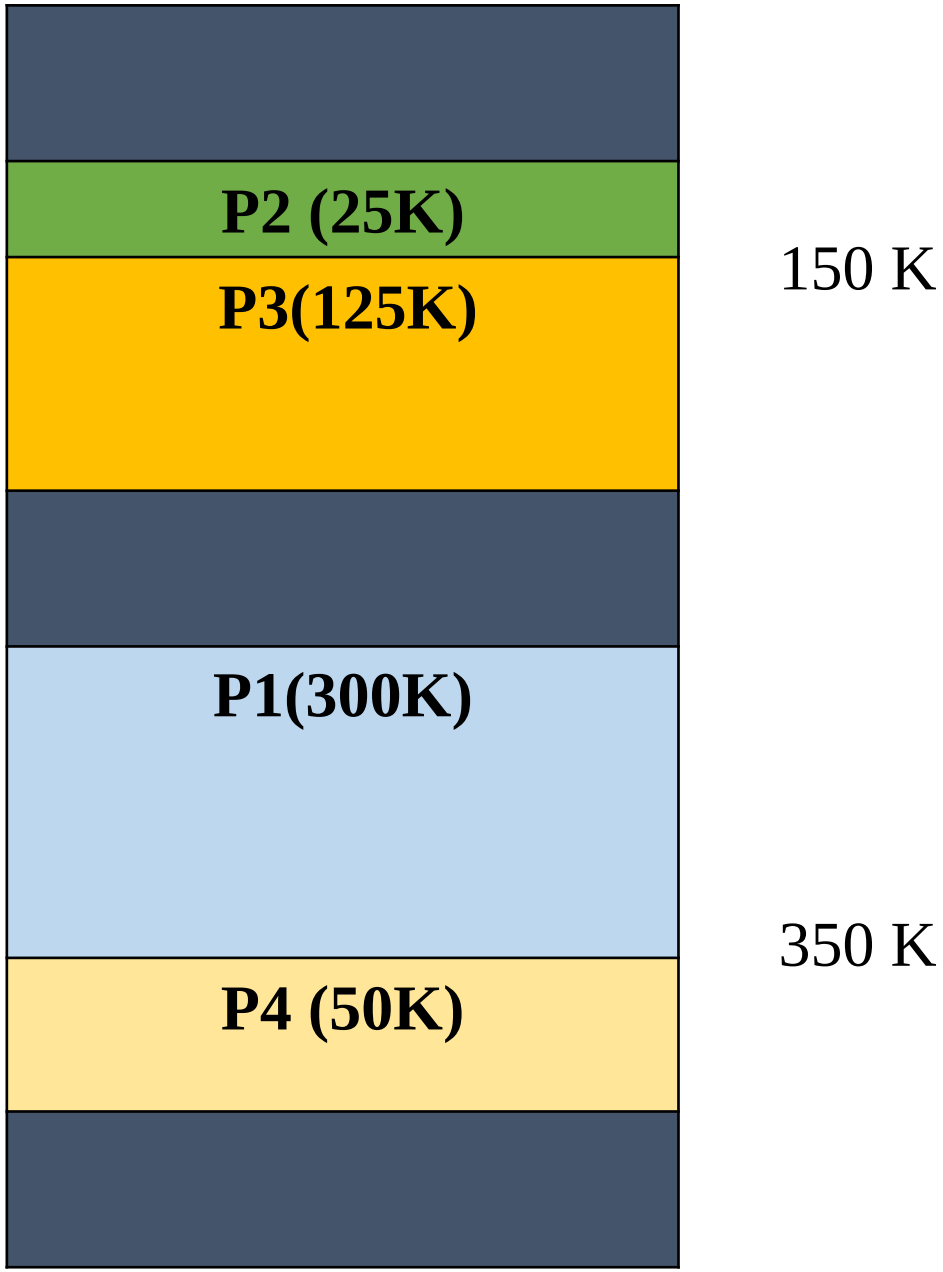
The above request could be satisfied with?
(Assume variable partition scheme)

A. Best fit but not first fit
B. First fit but not best fit
C. Both best and first fit
D. Neither best nor first fit

Solution (Question 1)

Checking for first fit:

- Let us assume P1(300k), P2(25k), P3(125k), and P4(50k)

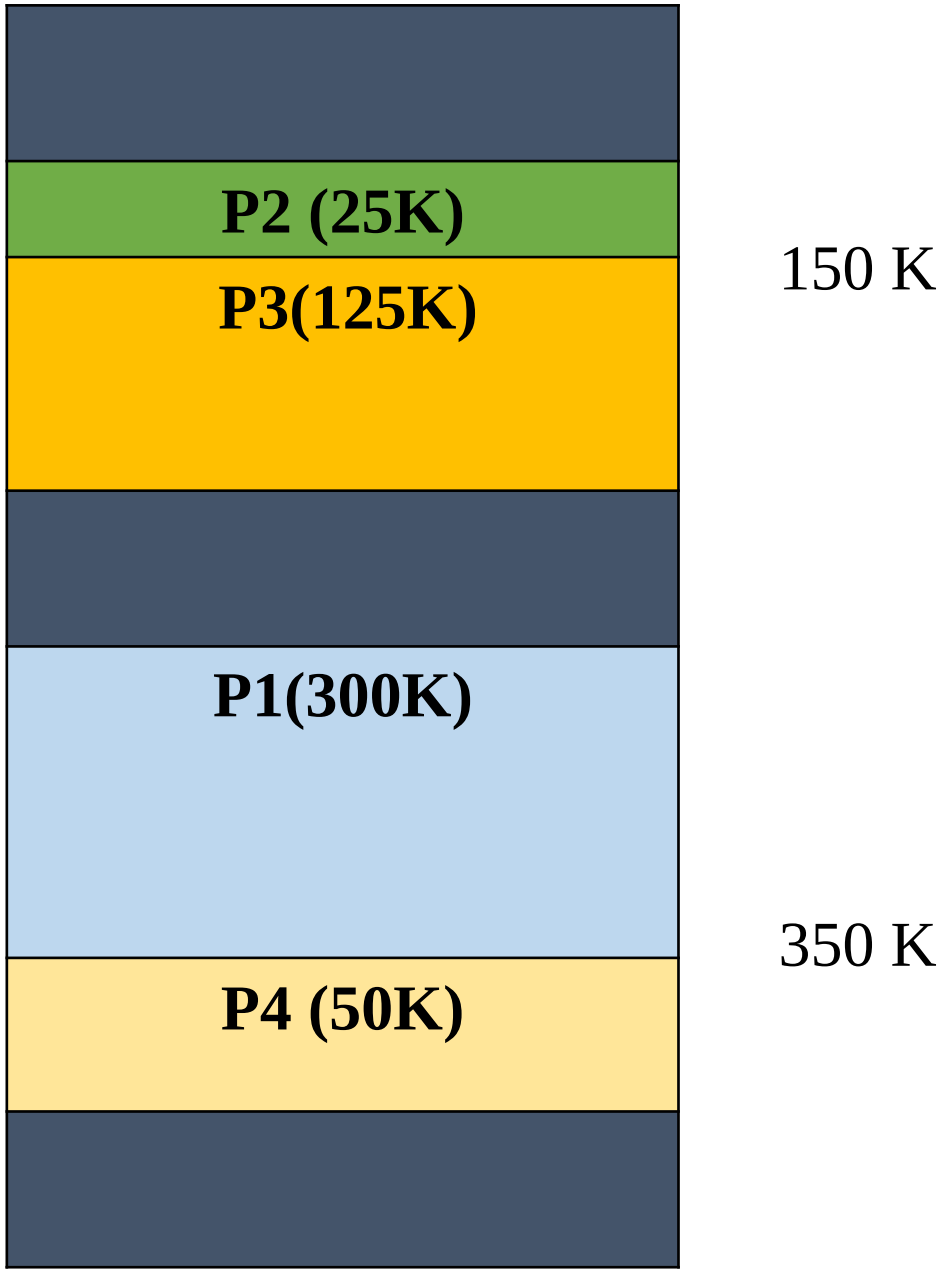


Solution (Question 1)

Checking for first fit:

- Let us assume P1(300k), P2(25k), P3(125k), and P4(50k)

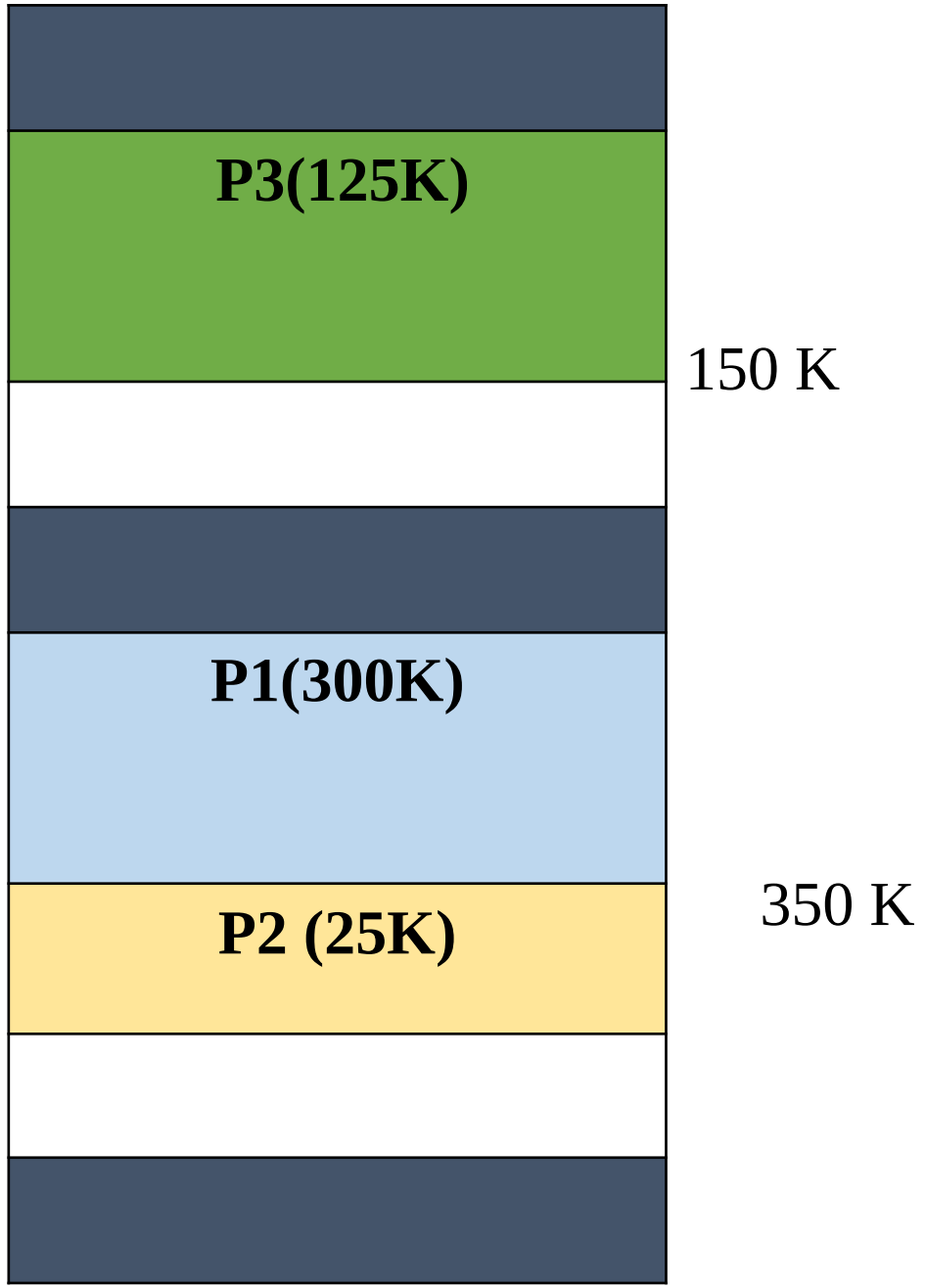
✓**First Fit is Satisfied**



Solution (Question 1)

Checking for best fit:

- Let us assume P1(300k), P2(25k), P3(125k), and P4(50k)

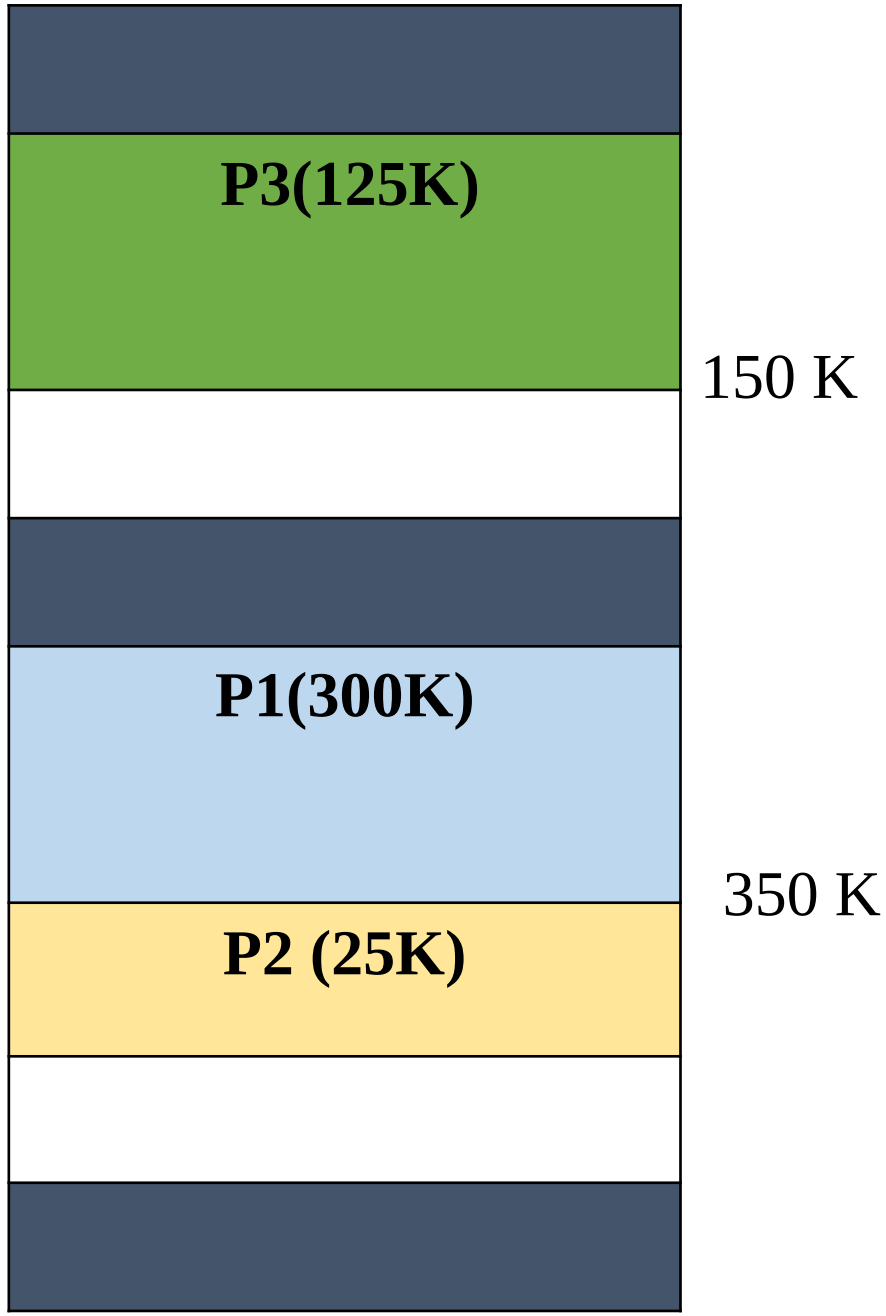


Solution (Question 1)

Checking for best fit:

- Let us assume P1(300k), P2(25k), P3(125k), and P4(50k)

Best Fit not Satisfied



Question2:

Let a memory have four free blocks of sizes $4k$, $8k$, $20k$, $2k$. These blocks are allocated following the best-fit strategy. The allocation requests are stored in a queue as shown below.

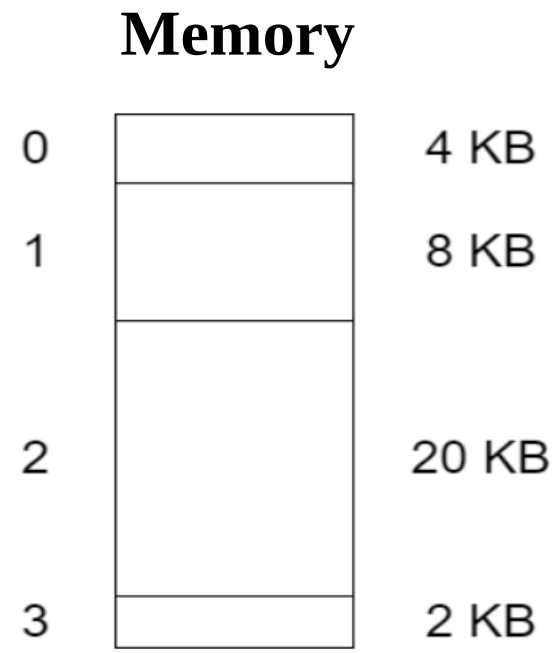
Request No	J1	J2	J3	J4	J5	J6	J7	J8
Request Sizes	2k	14k	3k	6k	6k	10k	7k	20k
Usage Time	4	10	2	8	4	1	8	6

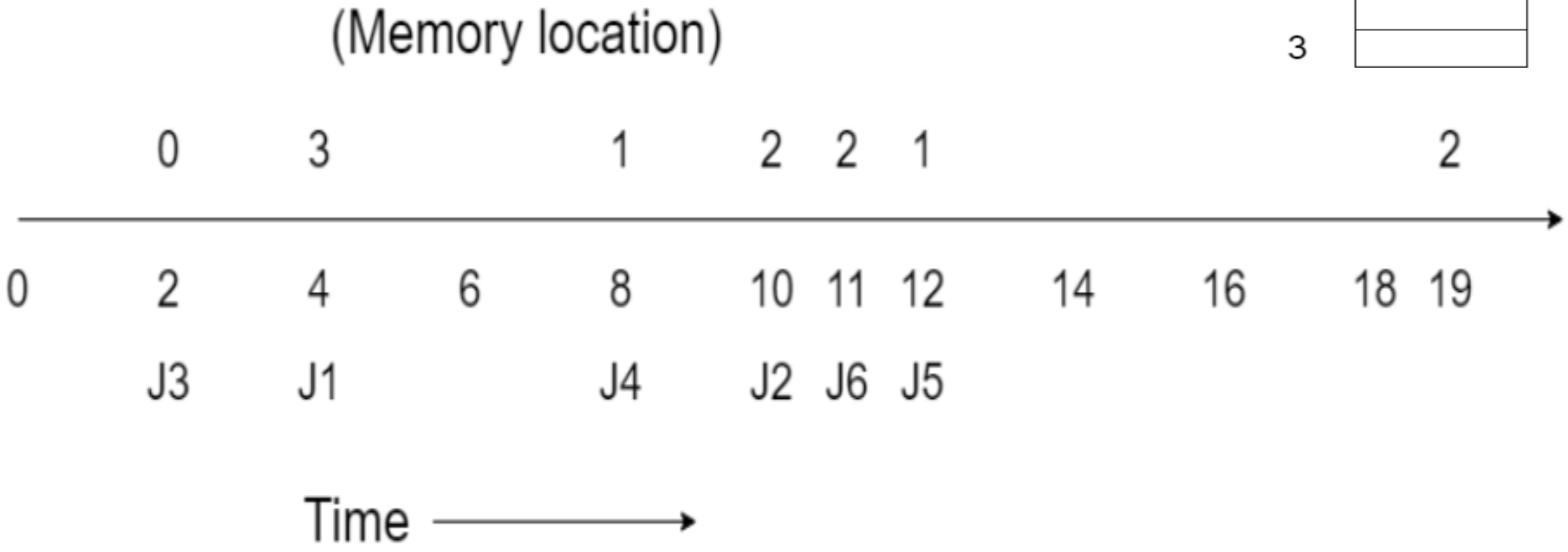
The time at which the request for $J7$ will be completed will be

A. 16
B. 19
C. 20
D. 37

Solution:

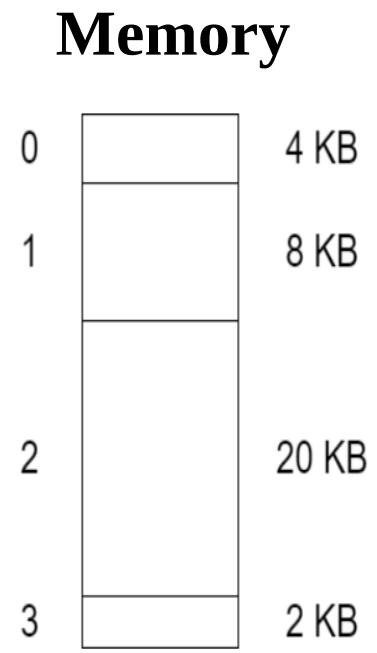
Request No	J1	J2	J3	J4	J5	J6	J7	J8
Request Sizes	2k	14k	3k	6k	6k	10k	7k	20k
Usage Time	4	10	2	8	4	1	8	6

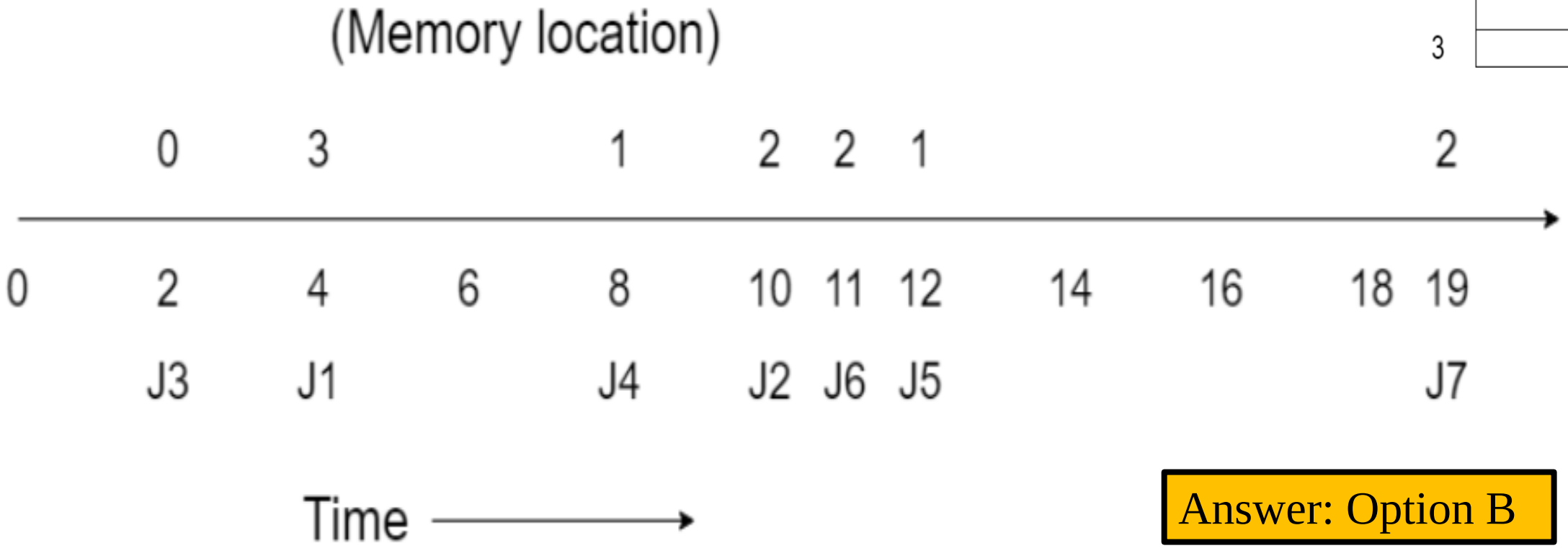




Solution:

Request No	J1	J2	J3	J4	J5	J6	J7	J8
Request Sizes	2k	14k	3k	6k	6k	10k	7k	20k
Usage Time	4	10	2	8	4	1	8	6





Answer: Option B

Explanation:

- We will put `J1` at location `3` in memory, the internal fragmentation is zero. It will execute for `4` unit of time.
- Now, we will put `J2` in memory at location `2`, where it will get execute for `10 units` of time. At present the internal fragmentation at memory location `2` is of `6 KB`.
- Now, we put `J3` in memory location `0`, where it will get executed for `2 unit` of time. At present the internal fragmentation at memory location at `0` is `1 KB`.
- Then `J4` will arrive, we put it in memory location `1`, where it will get executed for `8 units` of time. At present the internal fragmentation at memory location `1` is `2 KB`.
- We want to put `J5` in memory but there is no empty slot. So, we will wait until `J4` gets terminated and then we will place `J5` at memory location `1`, where it will get executed for `4 units` of time.
- Now we want to put `J6` in memory but there is no empty slot that can accommodate `J6`. So we will wait till `J2` gets completed and then we will place `J6` at memory location `2`. Where it will be executed for `1 unit` of time.

- We want to put `J7` in memory but again there is no empty slot which can accommodate `J7`. So we will wait till `J6` gets completed and then place `J7` at memory location `2`, where it will get executed for `8 unit` of time.
- Therefore `J7` will be completed at time unit `19` and time at which `J7` enters RAM is `11`.

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts,” Eleventh Edition (Willey).
2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition (Pearson Publications), 2014.
3. https://www.geeksforgeeks.org/
4. https://www.javatpoint.com/
5. https://www.tutorialspoint.com/
6. https://www.nesoacademy.org/
7. https://www.baeldung.com/
8. https://programmingport.hashnode.dev/
9. https://www.educative.io/
10. https://prepinsta.com