CS-3103 : Operating Systems : Sec-A (NB) :

Memory Management…..

Logical Memory Addressing

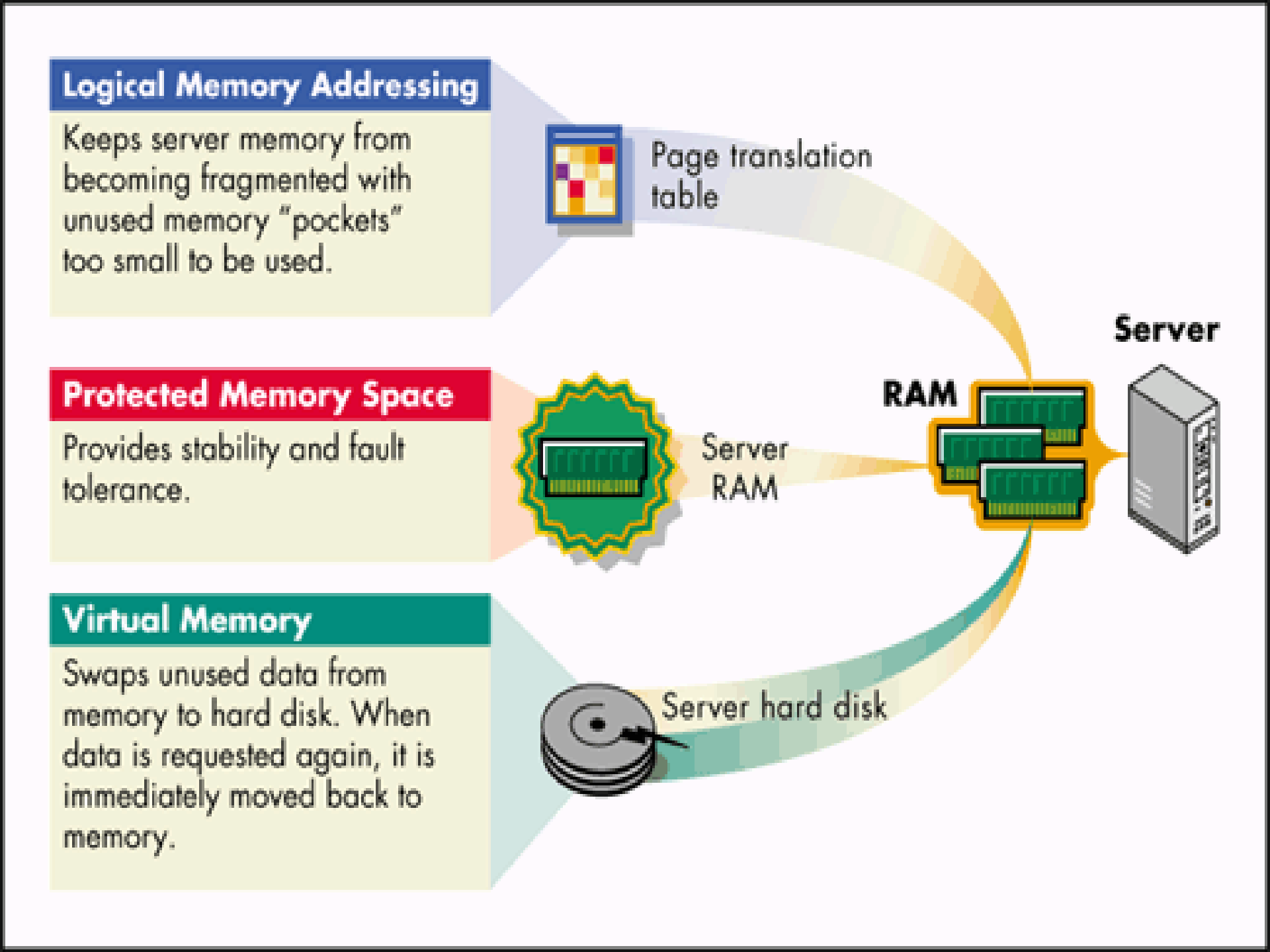
Keeps server memory from becoming fragmented with unused memory "pockets" too small to be used.

Protected Memory Space

Provides stability and fault tolerance.

Virtual Memory

Swaps unused data from memory to hard disk. When data is requested again, it is immediately moved back to memory.

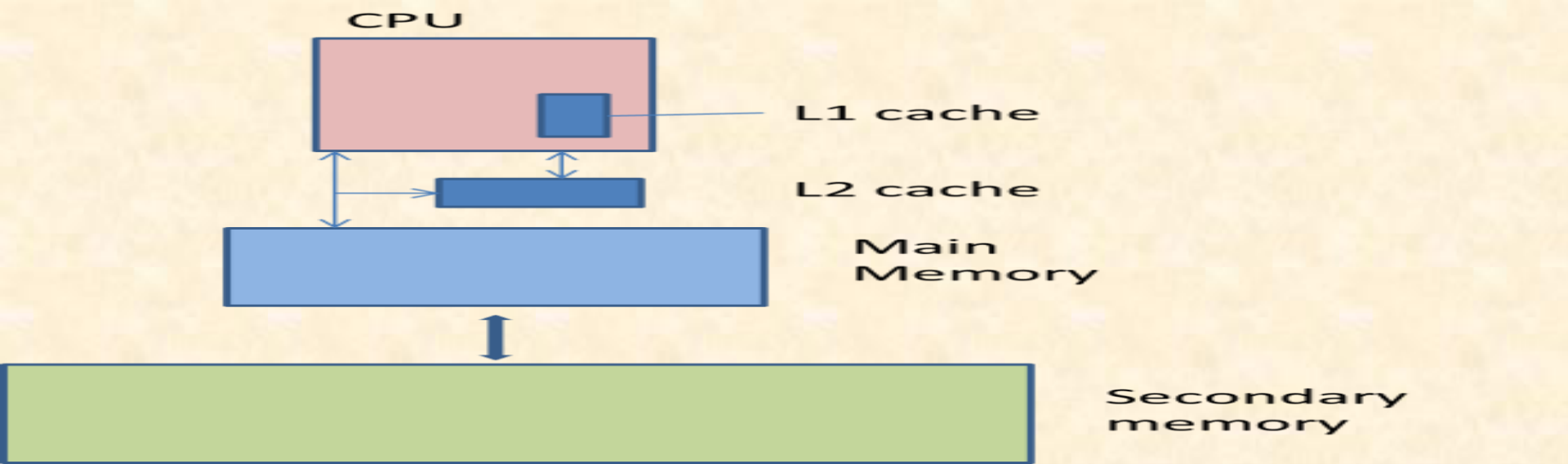


Memory Management → In the context of multiprogramming, we have two questions for OS:

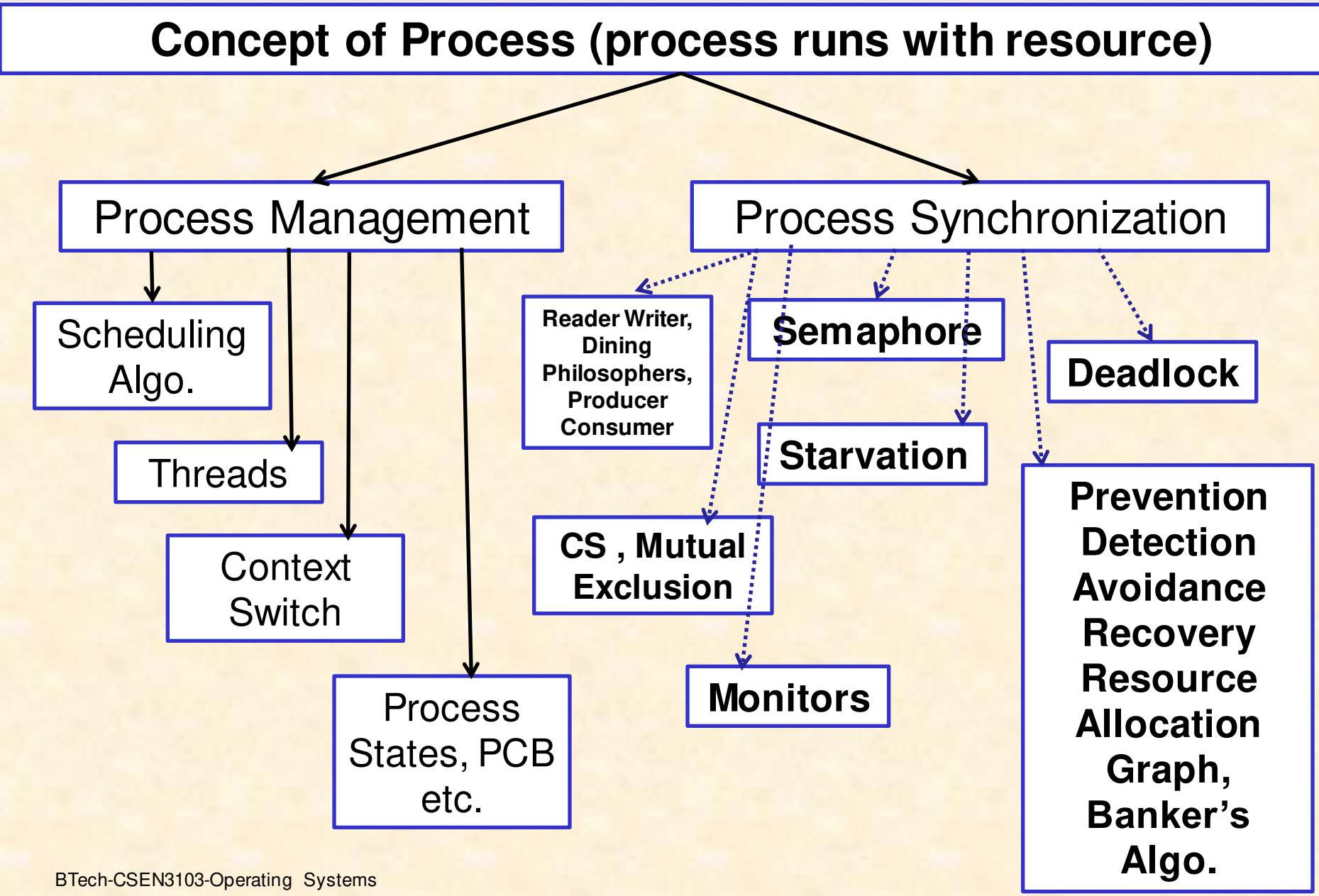
1. Which program is to be brought into the main memory?

2. Which one of the resident programs should be executed next?

- 2nd question talks about process management (next slide).
- 1st question deals with efficient memory management (5th slide).
- In a computer system, there is a hierarchy of storage elements organized such that the fastest memory is closest and directly accessible by the CPU shown in the following figure:



Just refresh with processes that are entering into Memory (Main Memory – MM; Virtual Memory - VM)



1. The main memory accommodates :

a) operating system b) CPU

c) user processes d) All of these

Ans: a & c

2. Which of the following is/are the requirements of memory management.

i) Relocation ii) Protection iii) Sharing iv) Memory organization

A) i, ii and iii only B) ii, iii and iv only

C) i, iii and iv only D) All i, ii, iii and iv

Ans: A)

3. Main memory in a computer system is …… as a linear or one dimensional, address space, consisting of a sequence of bytes or words.

Ans: d)

A) relocated B) protected C) shared D) organized

4. Once a program is compiled, it can be loaded for execution

a. Only from the compiler generated starting address

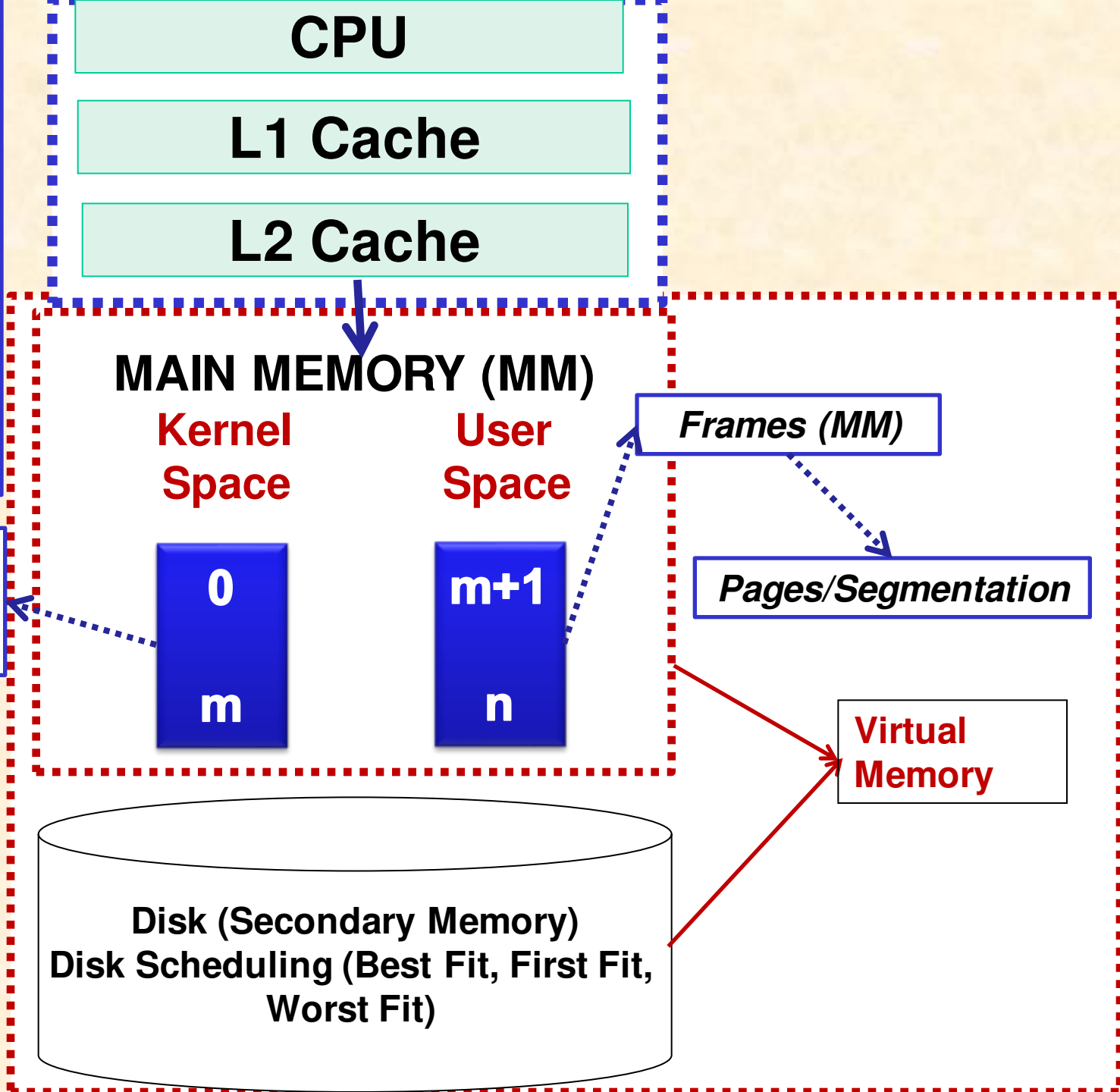
b. Any where in the main memory

c. User needs to specify where the compiled code is to be loaded

d. It is loaded starting from address 0 in the main memory.

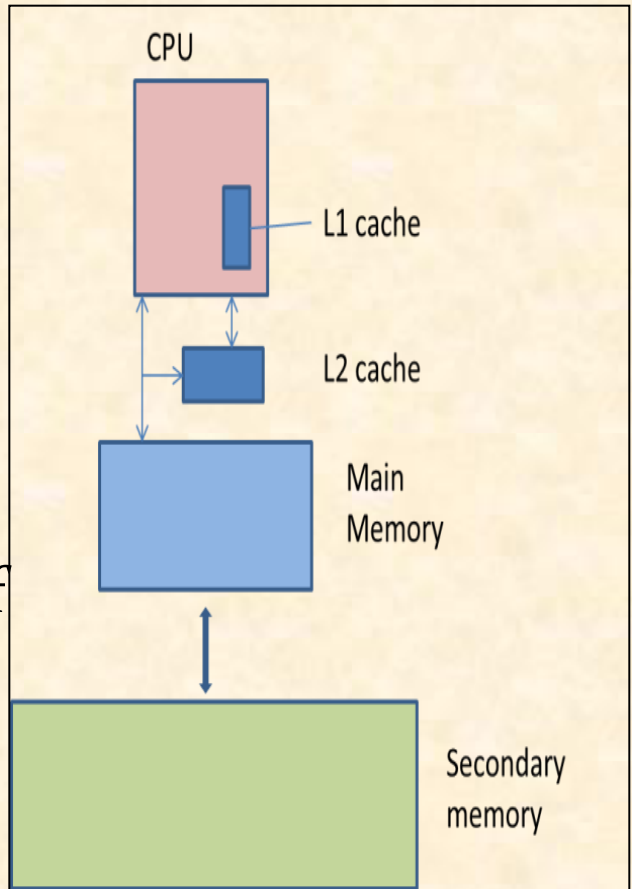
Ans: a)

MMU (responsible for translating logical addr. [CPU] to physical addr. [MM])



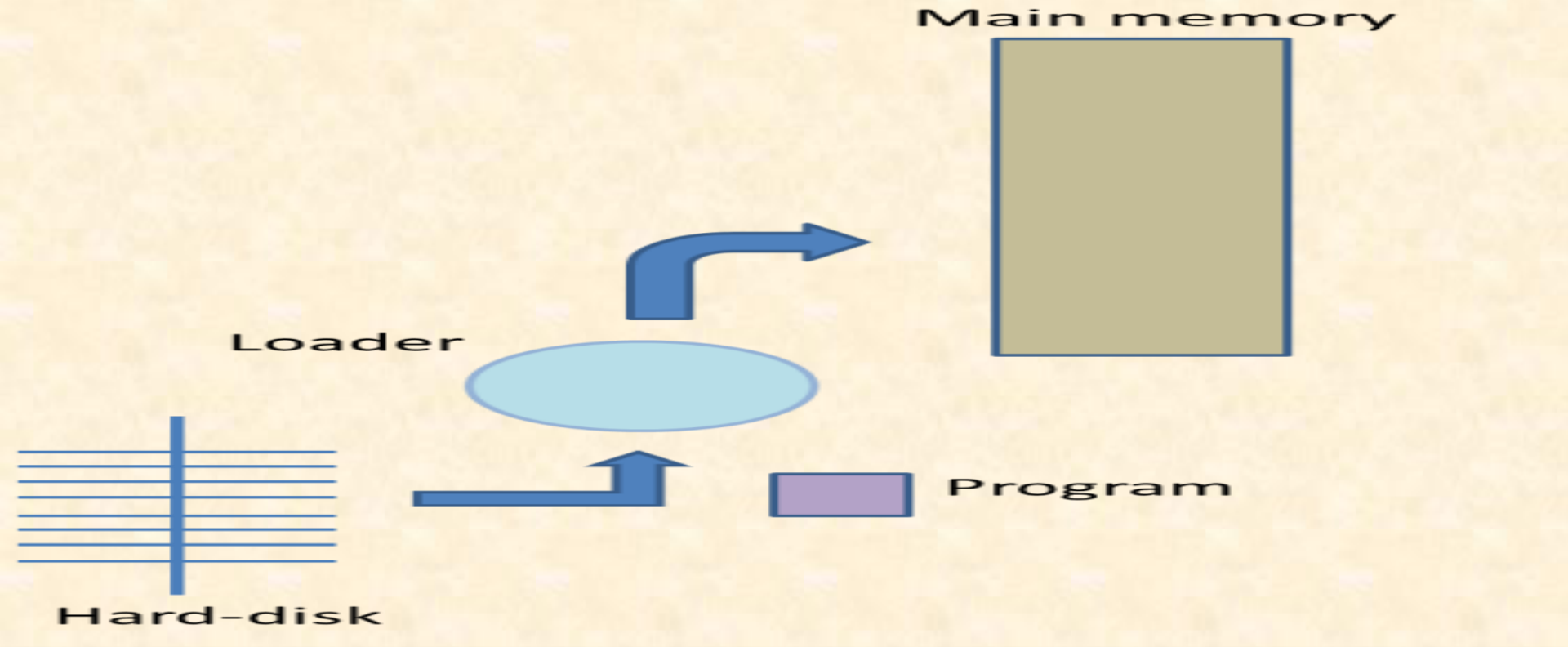
- On-chip cache (L1) is the closest and fastest memory. The secondary memory is largest, slowest and farthest from the CPU. Cache and main memory are directly accessible by the CPU, because these memories are byte or word addressable. The secondary memory, being a block device, is not directly accessible by CPU. This means the data from the secondary storage can be accessed in the form of blocks.

- **For example**: in a disk, the sector is the smallest addressable unit. Thus, the data from a disk can be brought in (or read) in terms sectors or in multiples of sectors. Generally, the size of a sector is 512 bytes. The sector, brought in from disk, is loaded into the main memory in such a way that each of 512 bytes of the sector gets an address in the memory location.

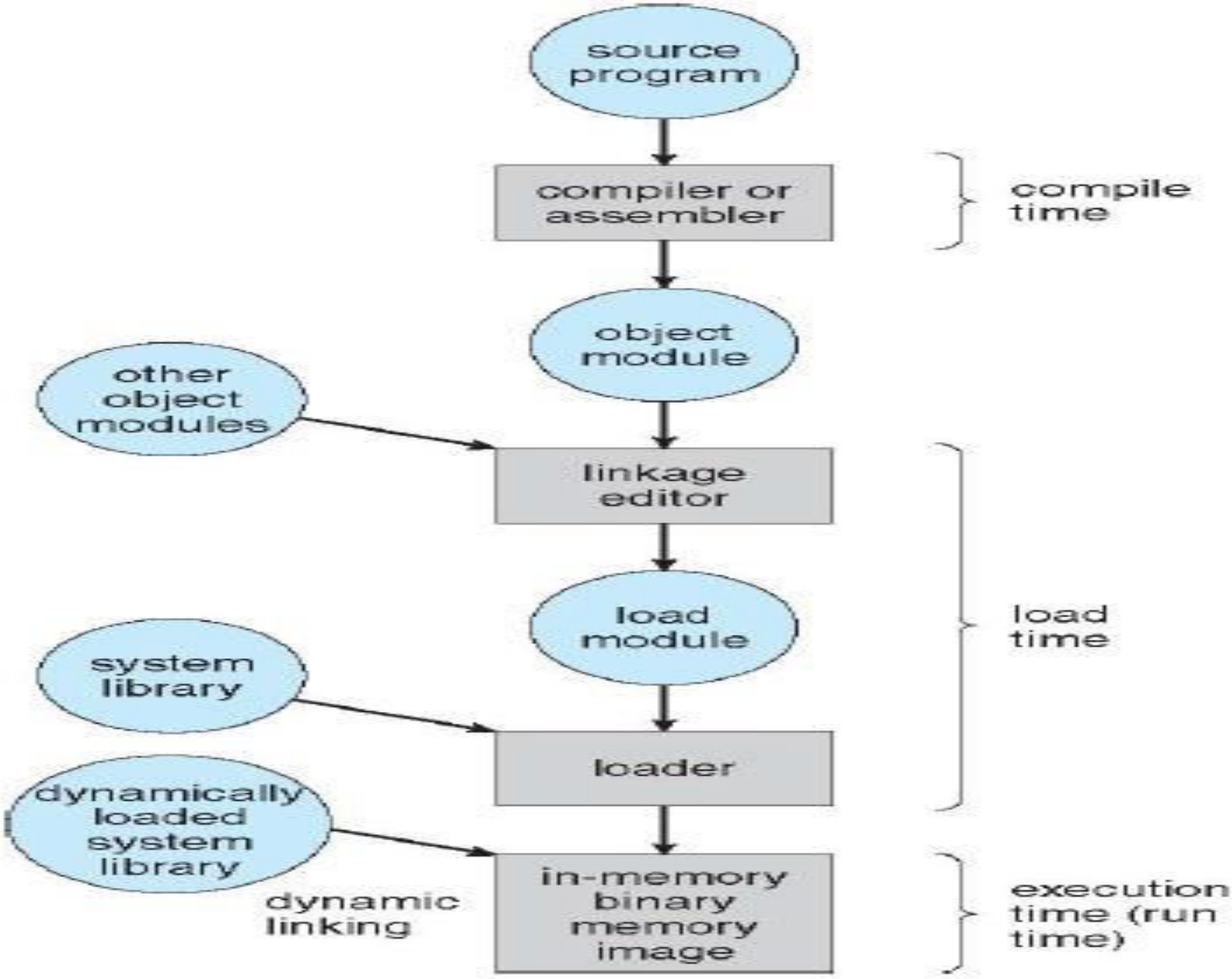


Loader

- The secondary memory is the permanent memory of the memory system. Therefore, the files are stored in secondary memory. *Ex. In a computer system, the program files and data files are on a hard disk*.
- *At the time of execution, the programs are loaded into main memory from secondary memory by a system program called* ***loader***.



Multistep Processing of a User Program



Efficient Memory Management

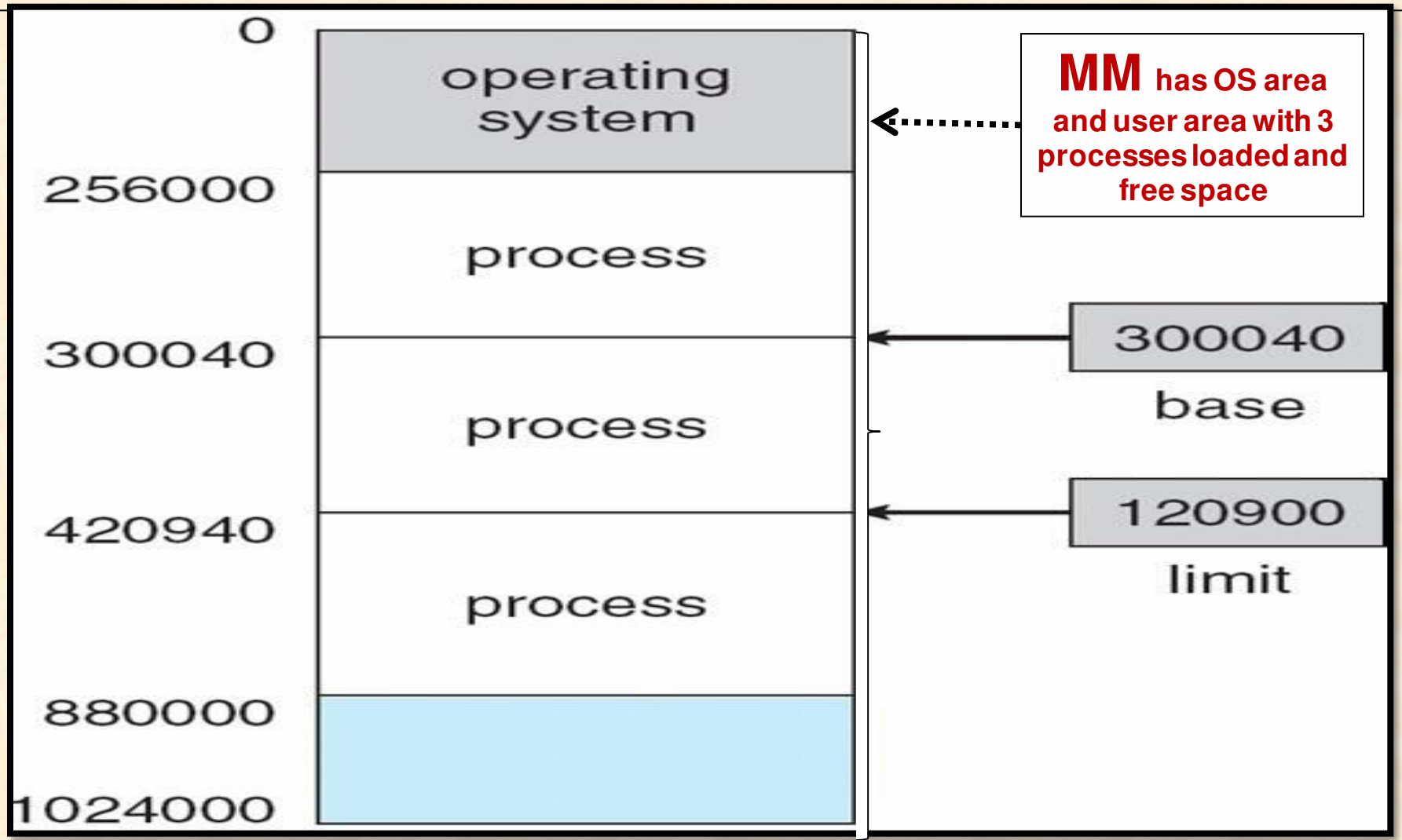
- The secondary storage contains thousands of programs, and the users may wish to run many of them. OS must know which programs needs to be moved into the MM next. For this OS has to address issues like ***Allocation***, ***Relocation***, ***Sharing*** and ***Protection*** in the context of processes running using sharable resources like CPU and MM….

- **Allocation** : The operating system must open a ***process*** for the ***program being loaded by the loader***. OS must also ***allocate main memory space*** for the incoming process and its associated data, so that it becomes a resident process (more info. on the allocation of main memory space in discussed in next slides….)

- **Relocation**, **Sharing** and **Protection**: next slides….

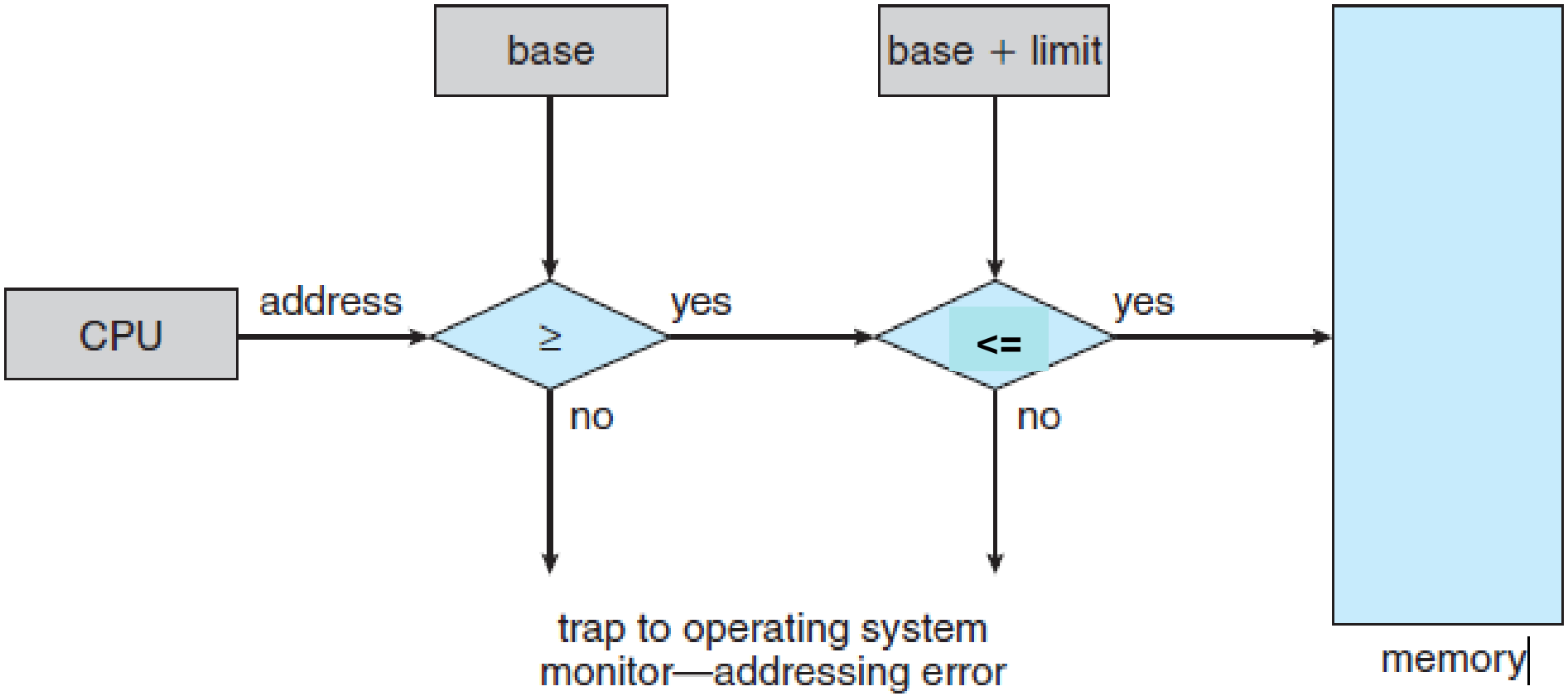
Allocation (cont.) : Base and Limit Registers needed for programs → processes loaded in MM

- A pair of **base and limit registers define the logical address space**
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user!



Allocation: Hardware Address Protection with Base and Limit Registers

- Protection of memory space is achieved by having the CPU h/w compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access OS-memory or other user's memory results in a trap to the OS, which treats the attempt as a fatal error.

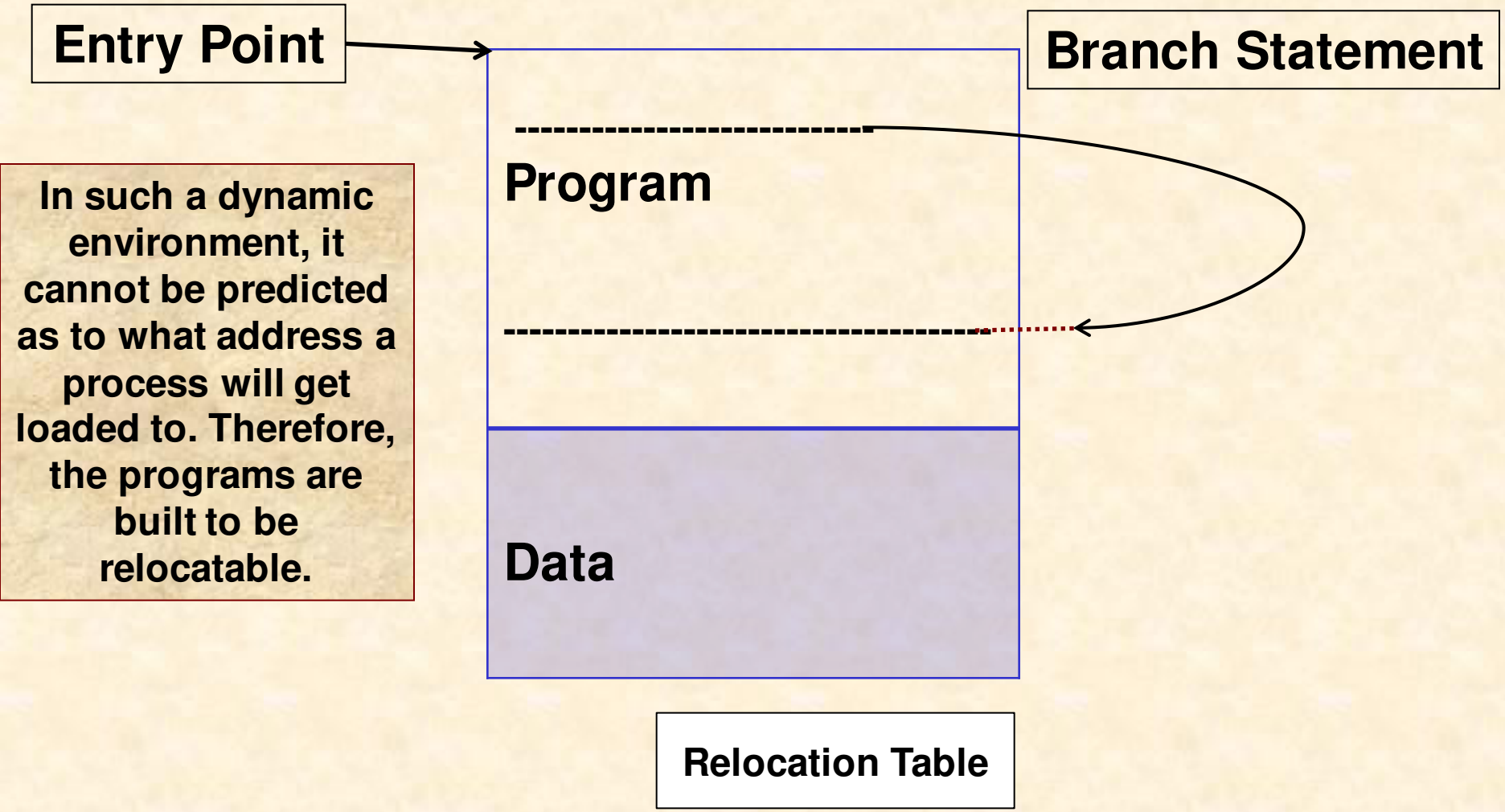


Relocation, Sharing and Protection

- **Relocation**
  - In a multiprogramming/multitasking environment, the processes are allocated memory space as and when they are loaded into memory.
  - When the ***degree of multiprogramming*** decreases, some resident waiting processes are swapped out to the secondary storage and later swapped in, as and when their resources become available (next slide….)
  - A relocatable program is compiled and linked in such a manner that its internal references are made relative to the address of the statement at its ***entry point***.
  - The program also contains a table of information about its statements which are 'address sensitive' (branch, procedure call statement etc.). This table is called the **relocation table.**

The degree of multiprogramming describes the maximum number of processes that a single-processor system can accommodate efficiently.

Relocation



In such a dynamic environment, it cannot be predicted as to what address a process will get loaded to. Therefore, the programs are built to be relocatable.

Sharing and Protection

- **Sharing**
  - The **system programs such as compilers and their libraries (stdio.h)** need to be shared by multiple processes.
  - The operating system ensures that access to shared applications, pages or databases is allowed to processes belonging to different users or groups, without compromising the integrity of the data, information or pages of the participating documents.

Protection

- The multiple resident processes frequently access their files and other resources.
- It becomes the responsibility of the operating system to manage the memory space in such a manner that no process, accidentally or otherwise, reads from or writes into the pages allotted to other processes. Example: ***C*** programs use pointers, i.e., OS monitors the addresses generated by a process at run time. This means the addresses generated by a process at run time must be restricted within the context of the process.

Dynamic Memory Allocation

Dynamic memory allocation means to allocate the memory at run time. Dynamic memory allocation is possible by 4 functions of stdlib.h header file: malloc(), calloc(), realloc(), free()

Address Binding

- Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the ***input queue***.

Binding of Instructions and Data to Memory

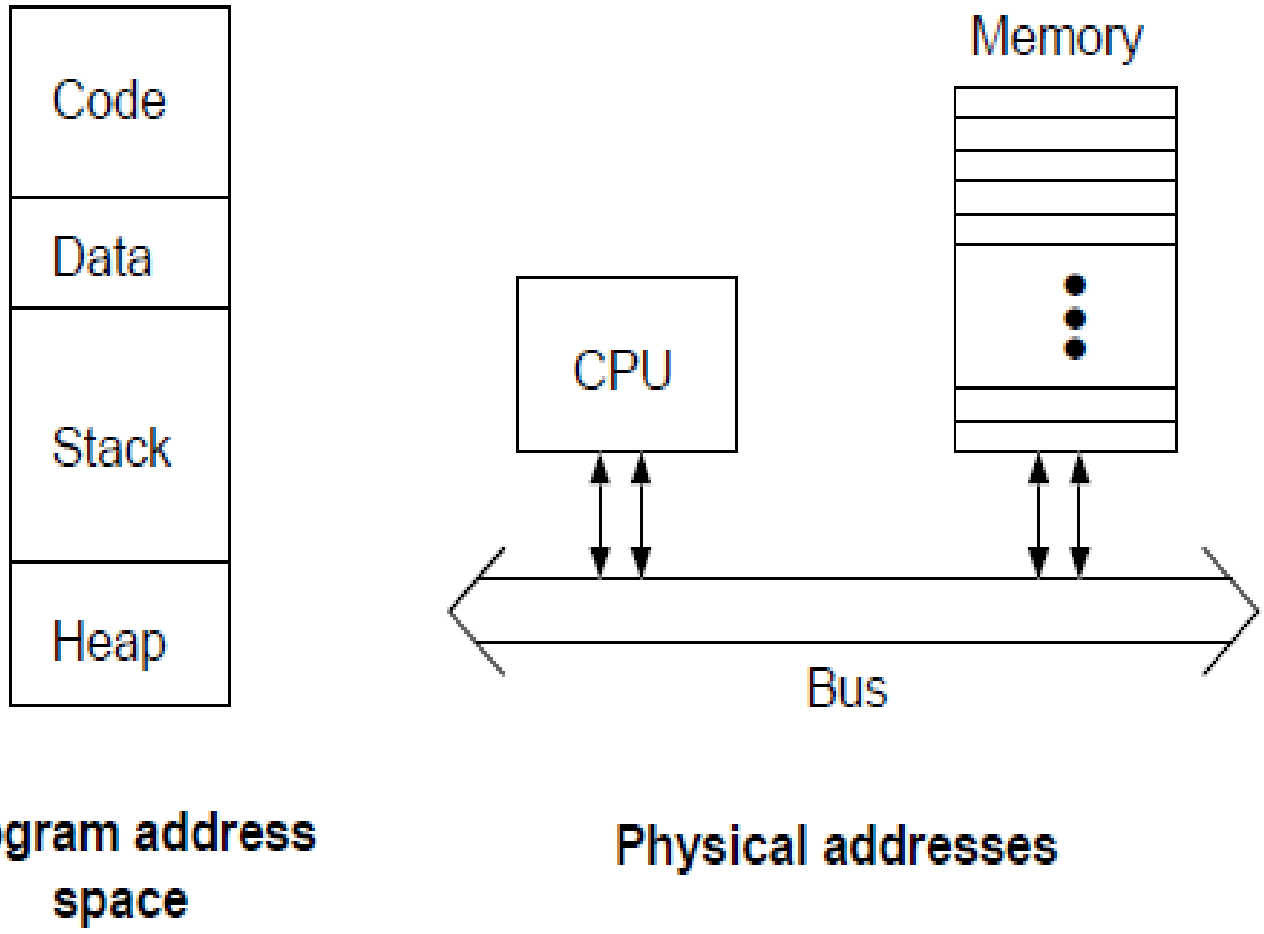
Address binding of instructions and data to memory addresses can happen at three different stages:-

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes. **(next slide…..)**
- **Load time**: Must generate ***relocatable* code** if memory location is not known at compile time. **(next slide…..)**
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*). **(next slide…..)**

Address binding – Compile Time Binding, Load time and Execution Time Binding

- **Compile Time**: If you know at the compile time where the process will reside in memory, then ***absolute code*** can be generated. 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 the starting location changes, then it will be necessary to recompile this code.
- **Load Time**: If it is not known at compile time where the process will reside in memory, then compiler must generate ***relocatable code***.
- **Execution Time**: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Address spaces



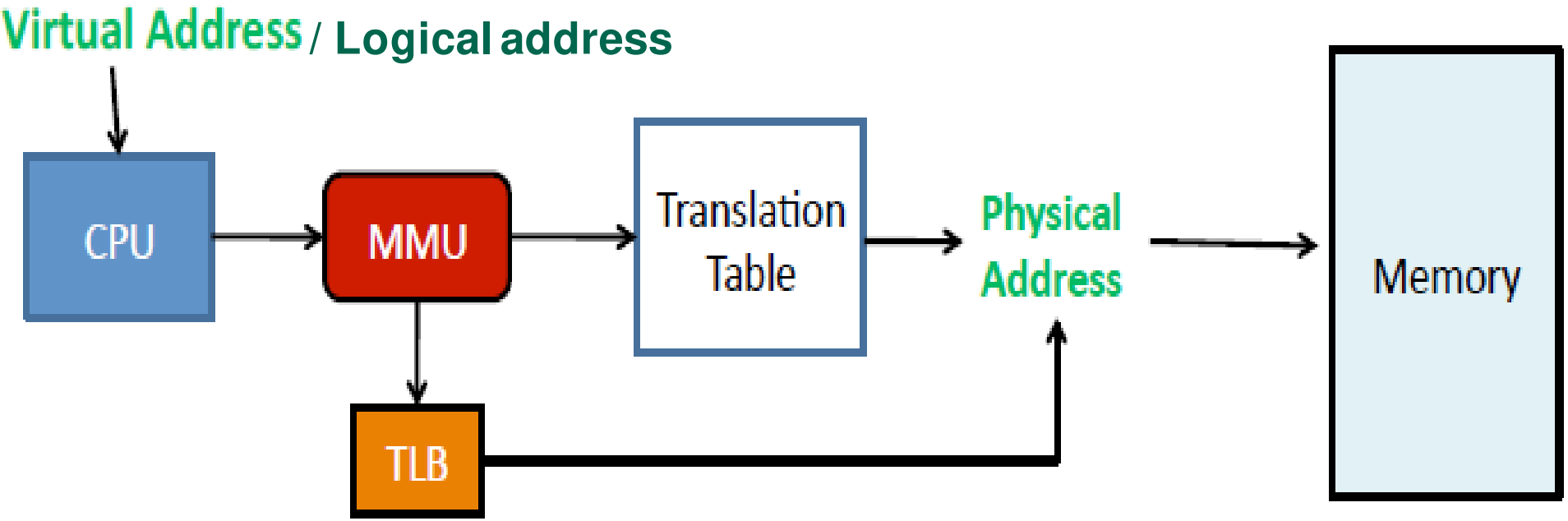
Address binding: mapping from one address space to another address space

Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - ***Logical address*** – generated by the CPU; also referred to as *virtual address*.
  - ***Physical address*** – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU)

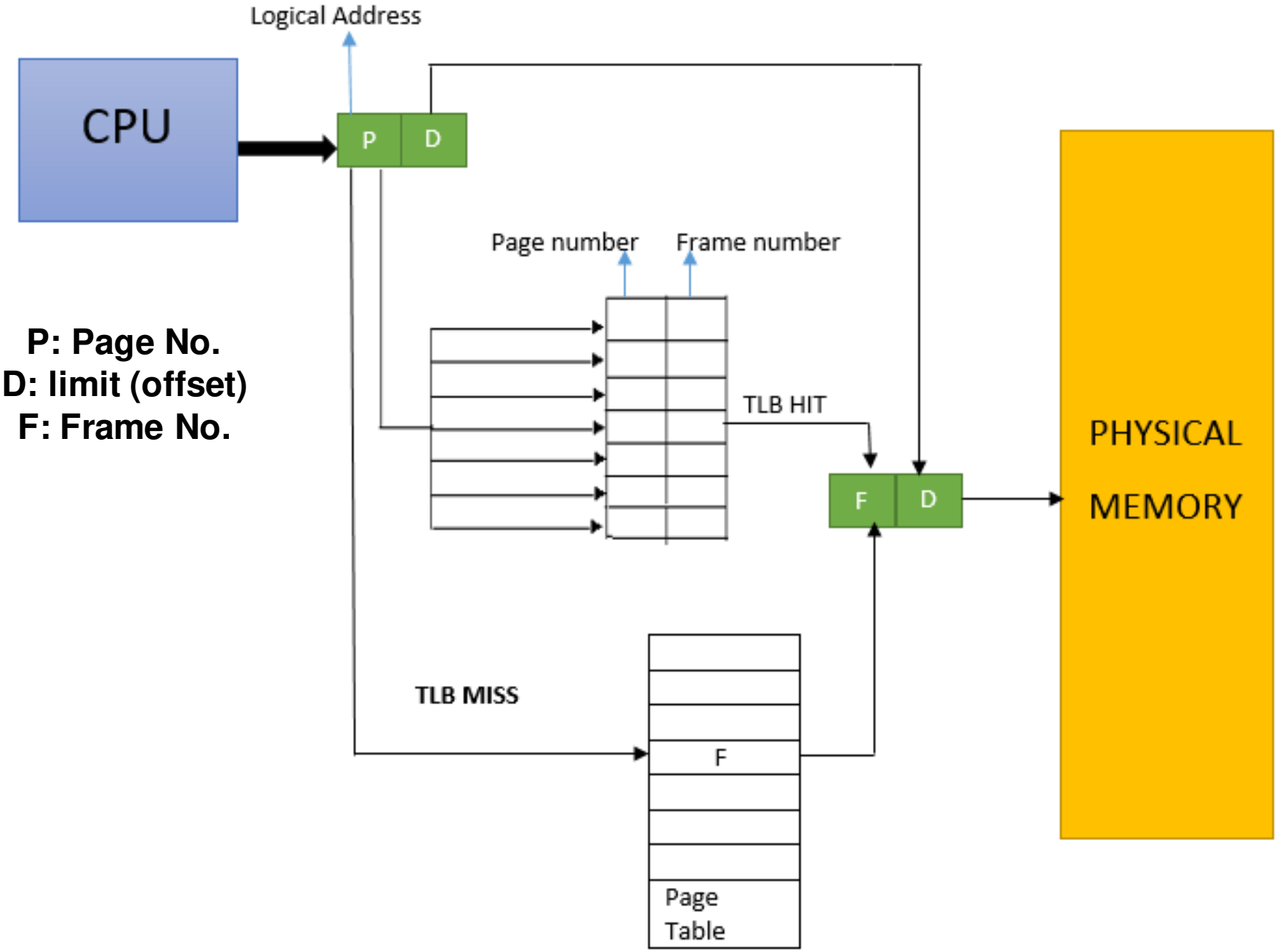
- Memory Management Unit (MMU)
  - Hardware unit that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
- Translation Lookaside Buffer (TLB)
  - Essentially a cache for the MMU's virtual-to-physical translations table
  - Not needed for correctness but source of *significant* performance gain



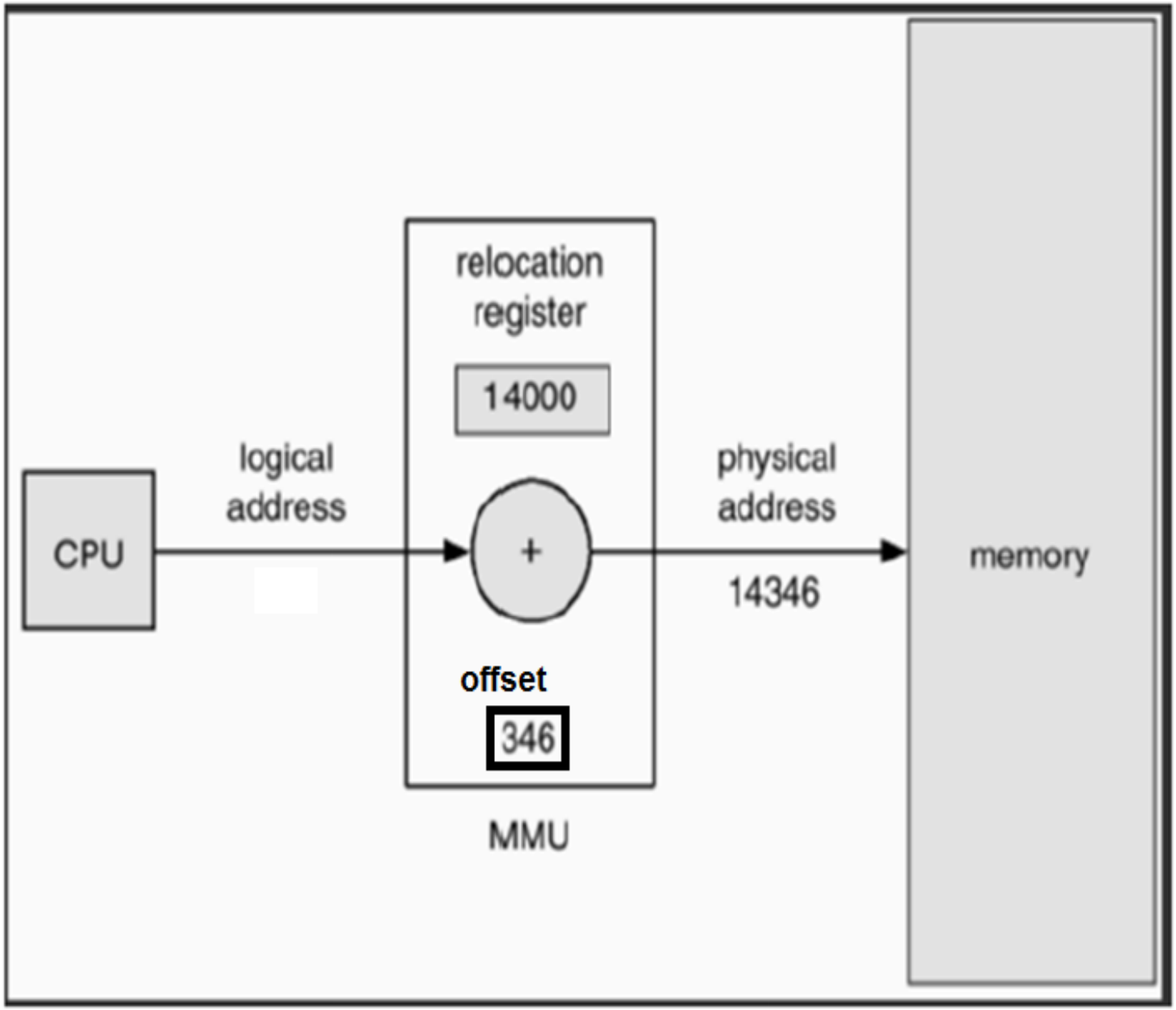
Memory-Management Unit (MMU) & TLB

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
- *A **translation lookaside buffer** (**TLB**) is a memory cache that is used to reduce the time taken to access a user memory location. It is a part of the chip's memory-management unit (MMU).* The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache. The majority of desktop, laptop, and server processors include one or more TLBs in the memory-management hardware, and it is nearly always present in any processor that utilizes paged or segmented virtual memory..

Memory Allocation



Dynamic relocation using a relocation register



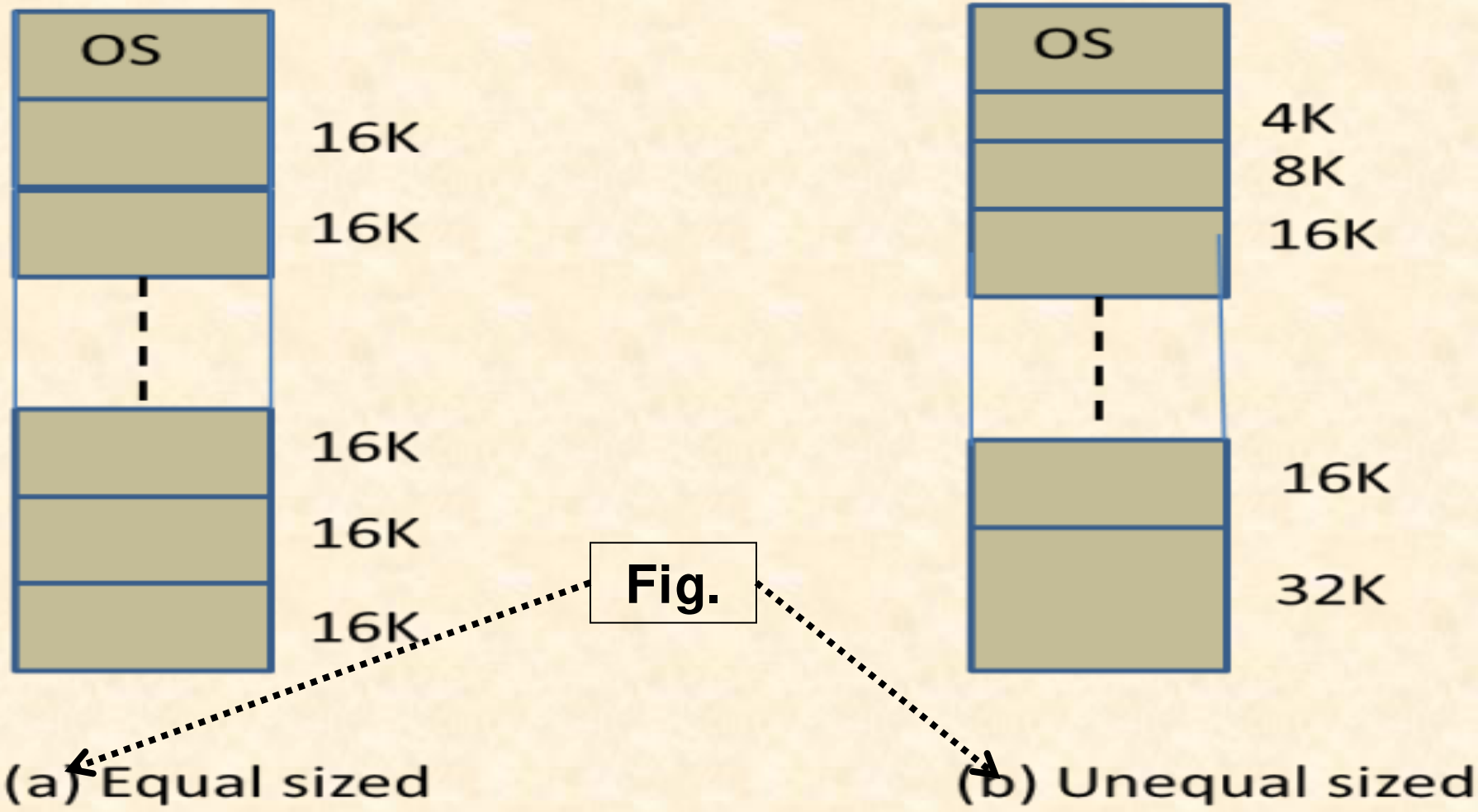
Kernel loads relocation register when scheduling a process

Memory Allocation: Loading Programs into Main Memory

- In order to accommodate multiple programs in the main memory, the OS divides the MM into two major parts, the system area and the user area. The task is to accommodate multiple user programs into the user area of the MM. The simplest way is to divide the memory into partitions:
  - **Fixed sized partitions**
  - **Variable sized partitions**

Fixed Sized Partitions can be of two types:

- Equal sized partitions
- Unequal sized partitions



Fixed Sized Partitions can be of two types (from the last slide….)

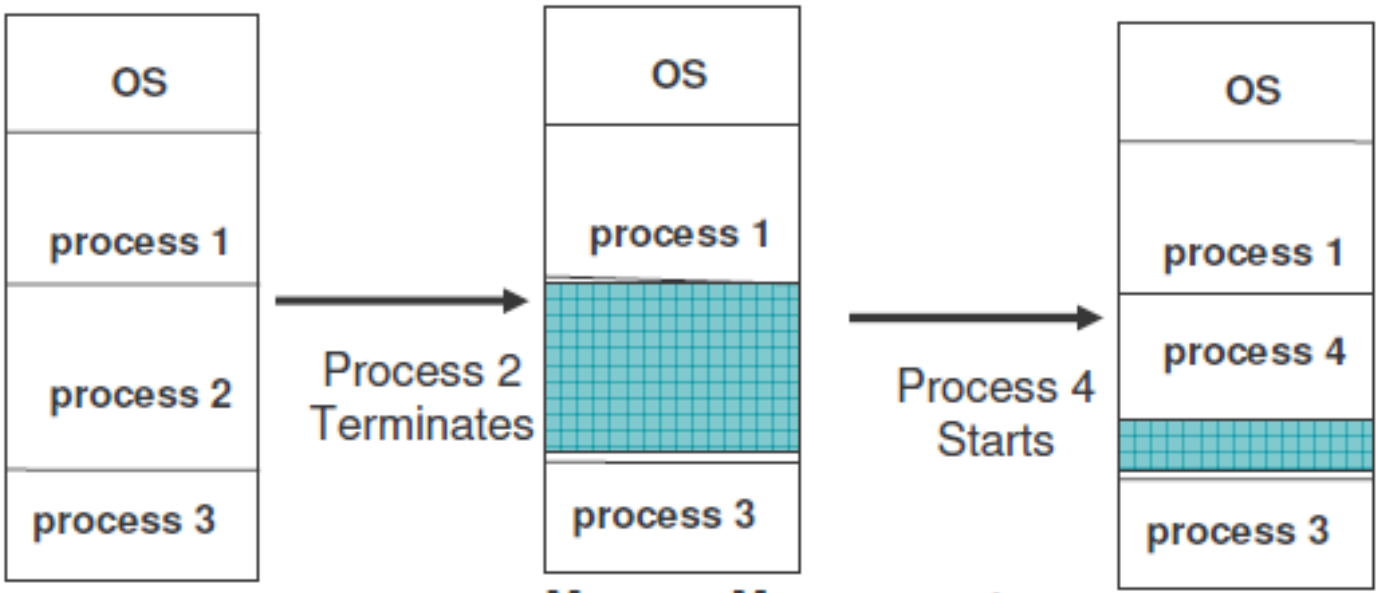
- For equal sized partition, the OS divides the main memory into equal sized partitions. Example: let us assume that the memory is divided into partitions of size 16K, each as shown in Fig. a) [previous slide]
- Now, in this case, 16K becomes the upper limit on the size of the program that can be loaded in the MM.
- Thus, a program larger than 16K cannot be executed because it cannot be loaded into the memory.
- On the other hand, a program of size less than 16K would lead to memory wastage. For instance, a program of size 10K can be loaded into any of the available partitions but a memory chunk of size 6K will be wasted.
- In fact, the smaller the size of the program to be loaded, more the wastage of memory that would occur. ***This wastage of memory within a partition is called Internal Fragmentation***.

CONTIGUOUS ALLOCATION

DYNAMIC STORAGE

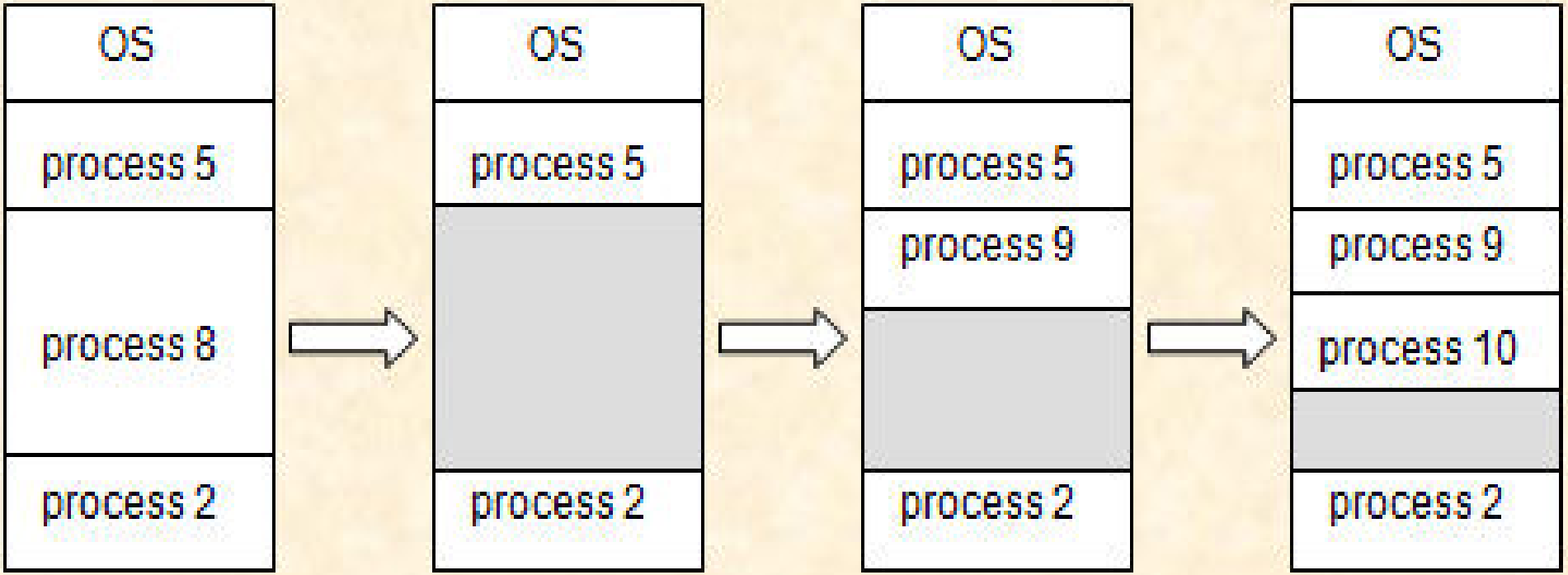
- (Variable sized holes in memory allocated on need.)
- Operating System keeps table of this memory - space allocated based on table.
- Adjacent freed space merged to get largest holes - buddy system.

ALLOCATION PRODUCES HOLES



Contiguous Allocation (Cont.)

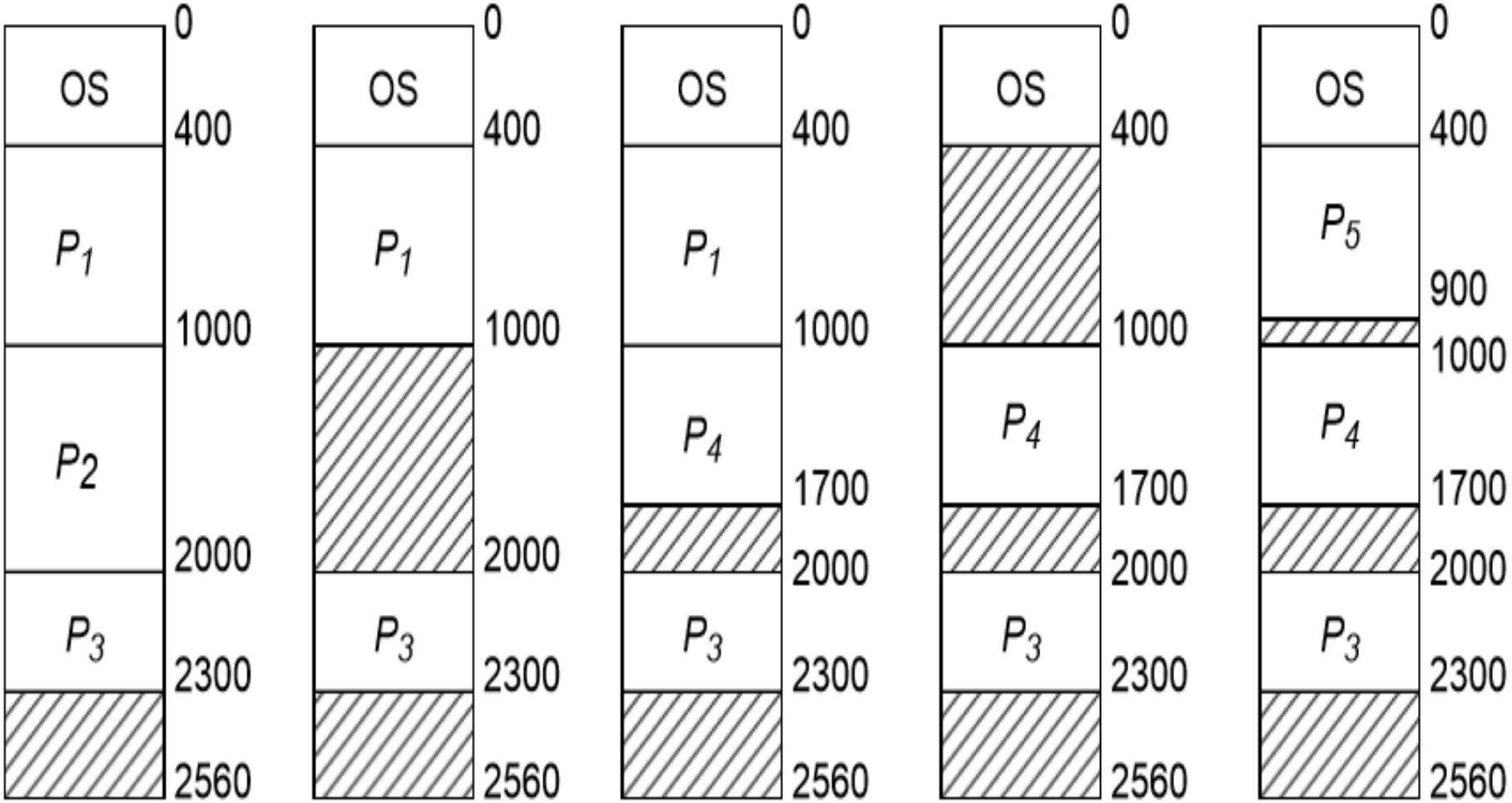
- Multiple-partition allocation
  - ***Hole*** – block of available memory; holes of various size are scattered throughout memory.
  - *When a process arrives, it is allocated memory from a hole large enough to accommodate it.*
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Example:

Process sizes:

P_1 600 P_2 1000 P_3 300 P_4 700 P_5 500



Variable Sized Partitions

- Here, the programs are loaded into the memory in the order of their arrival.
- As the programs are of different sizes, the memory is dynamically divided into variable sized partitions. Therefore, this scheme of partitioning is also called **dynamic partitioning**.

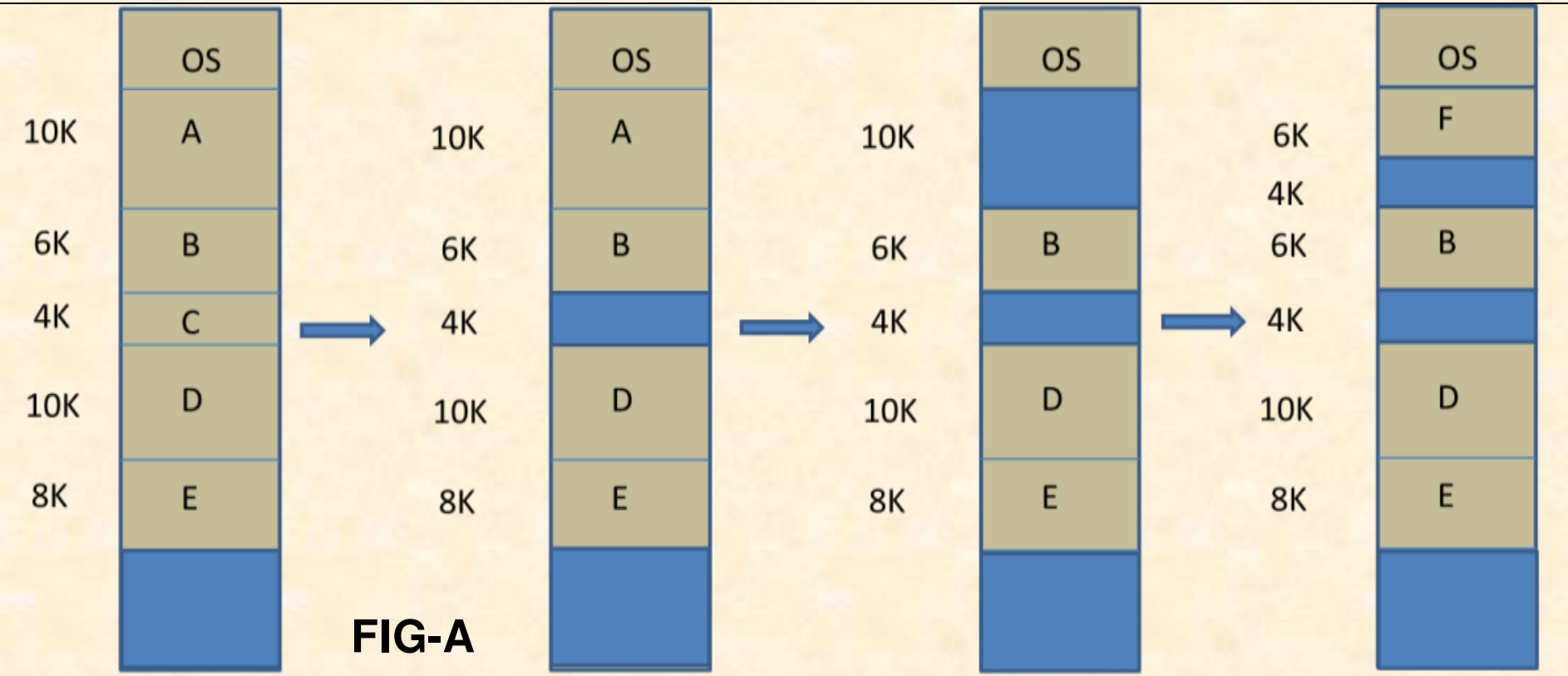


FIG-A

Variable Sized Partitions

- From the **figure A (a)** in last slide, its noted that ***initially***, there is ***no*** memory fragmentation. A chunk of memory is available towards the end for incoming user programs. Let us assume that program 'C' finishes its job and leaves the memory. It leaves behind a memory block (*hole*) of size 4K. Now we have two chunks of memory, i.e., a chunk of 4K and a large chunk at the end of the memory (**Fig A (b)**). If, at this moment, program A of size 10K also leaves, then we are left with another memory block of size 10K. In fact, by now the memory is fragmented into three memory holes (shown in **Fig A ©**).

- *This type of memory fragmentation created by outgoing programs is called as* ***external fragmentation***. As programs move in and out, the memory holes become smaller and smaller (**Fig A (d)**). Note that, in the example shown in **Fig A**, a program called F, of size 6K, has joined the system. OS has loaded the new entrant into the 10K hole leaving behind a hole of size 4K…. (moving towards compaction)…..

- **Solution**: The problem of external fragmentation is solved by moving the resident programs into a big chunk of contiguous programs and the fragmented memory into another chunk of free memory. This activity is known as ***storage compaction***.

Storage Compaction

- Incoming programs can be accommodated in the free memory space created by storage compaction.
- It is a time-consuming exercise, leading to slowing of the system.
- It avoids external fragmentation.

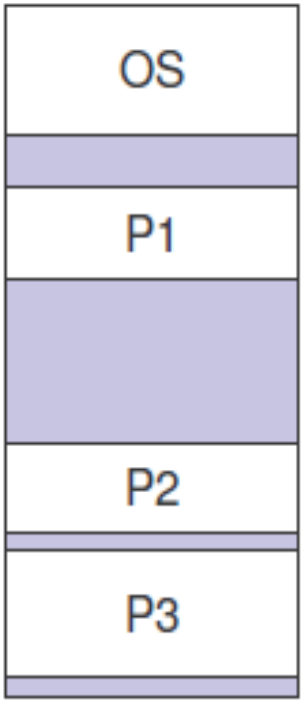
COMPACTION

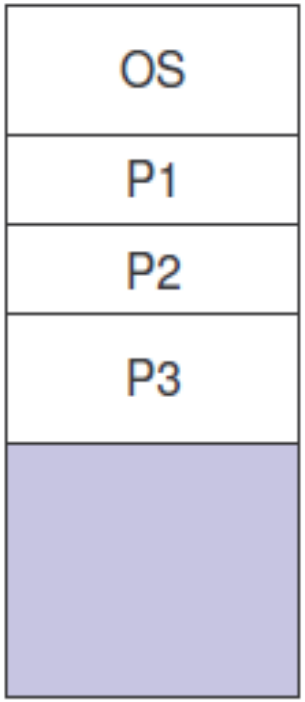
Trying to move free memory to one large block.

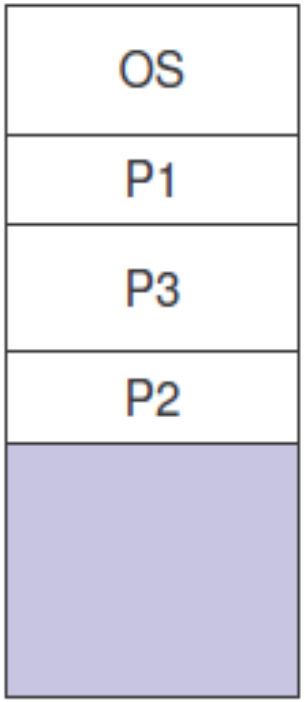
Only possible if programs linked with dynamic relocation (base and limit.)

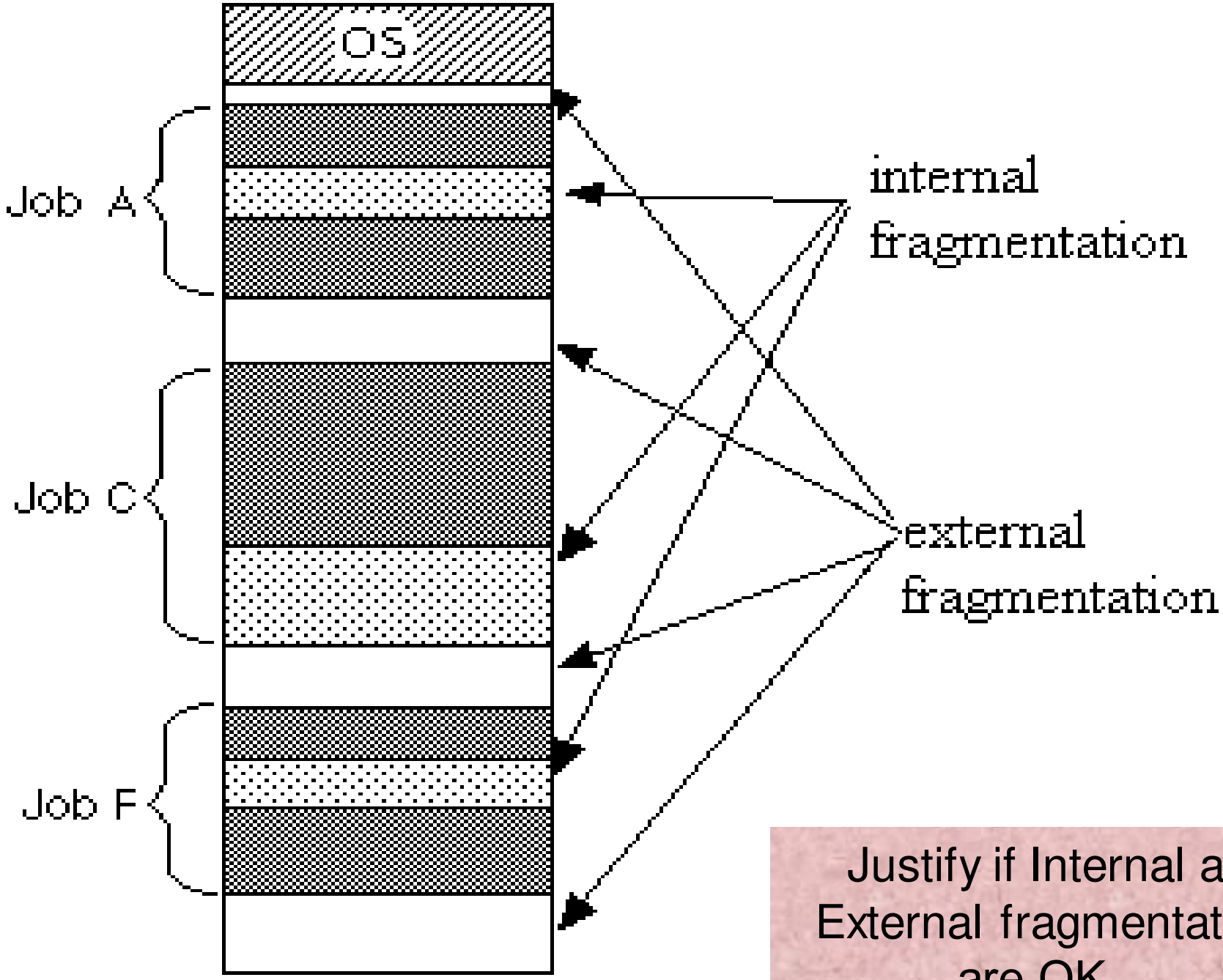
There are many ways to move programs in memory.

Swapping: if using static relocation, code/data must return to same place.
But if dynamic, can reenter at more advantageous memory.









Justify if Internal and External fragmentations are OK….

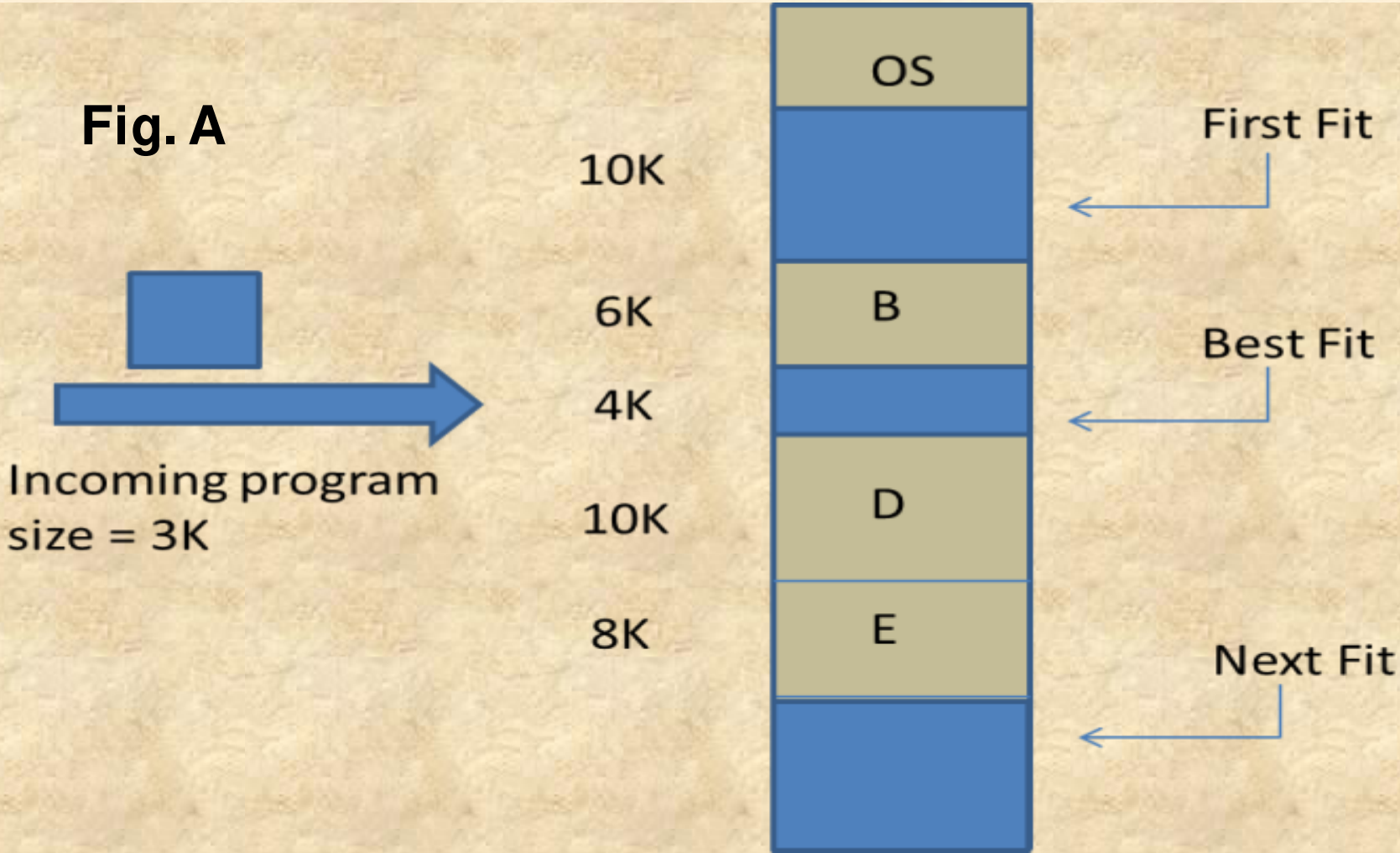
Loading Policies

- External fragmentation happens and adapts using any of the following loading policies to load the incoming programs in MM:
  - First fit policy
  - Best fit policy
  - Next fit policy

Loading Policies: First Fit Policy

- The first free memory block, from the start of the memory, whose size is more than equal to the size of incoming program is found. Thereafter, the incoming program is loaded into that free memory block as shown in **Fig. A**

Fig. A



Loading Policies: First Fit (Continued…), Best Fit Policy and Next Fit Policy

- **FF**: It may be noted that the first free memory block is of size 10K. Therefore, the incoming program of size 3K will be loaded into this block, thereby creating another memory chunk of size 7K. ***Disadvantage***: This may lead to severe problem of external fragmentation.
- **BF**: The operating system searches for a free memory block whose size is just more than or equal to the size of the incoming program. Thus, under this policy, the 4K memory block is chosen, thereby leading to the least external fragmentation.
- **BF**: As this policy spends some time searching for the best suited memory block, the system becomes slow.
- **NF**: The operating system keeps a pointer to the location where the last program was loaded.
- **NF**: From that pointer onwards, it finds the next free memory block and loads the incoming program.

Consider the data given in following table:

Partitions Size (KB)	4 KB	8 KB	20 KB	2 KB				
Job sizes (KB)	2 KB	14 KB	3 KB	6 KB	6 KB	10 KB	20 KB	2 KB
Burst time (ms)	4	10	2	1	4	1	8	6

When will the job of size 20 KB get the memory and be completed, if the best fit algorithm is used for allocating jobs to various memory partitions?

Ans: Let us construct a Gantt chart of the occupancy of the 20 KB partition, using the best fit policy. The Gantt chart is as follows:

It may be noted that the 14 KB job will be loaded into the 20 KB partition. Thereafter the 10 KB will go into this partition followed by the 20 KB job. Therefore, the time of completion of the 20 KB job is as given below:

Time of completion for the 20 KB job = 10 + 1+ 8 = 19 ms

The 20 KB job will get the memory after 11 ms (wait time = 11 ms) and will complete at 19 ms, assuming that it gets the CPU as soon as it is loaded into the memory.

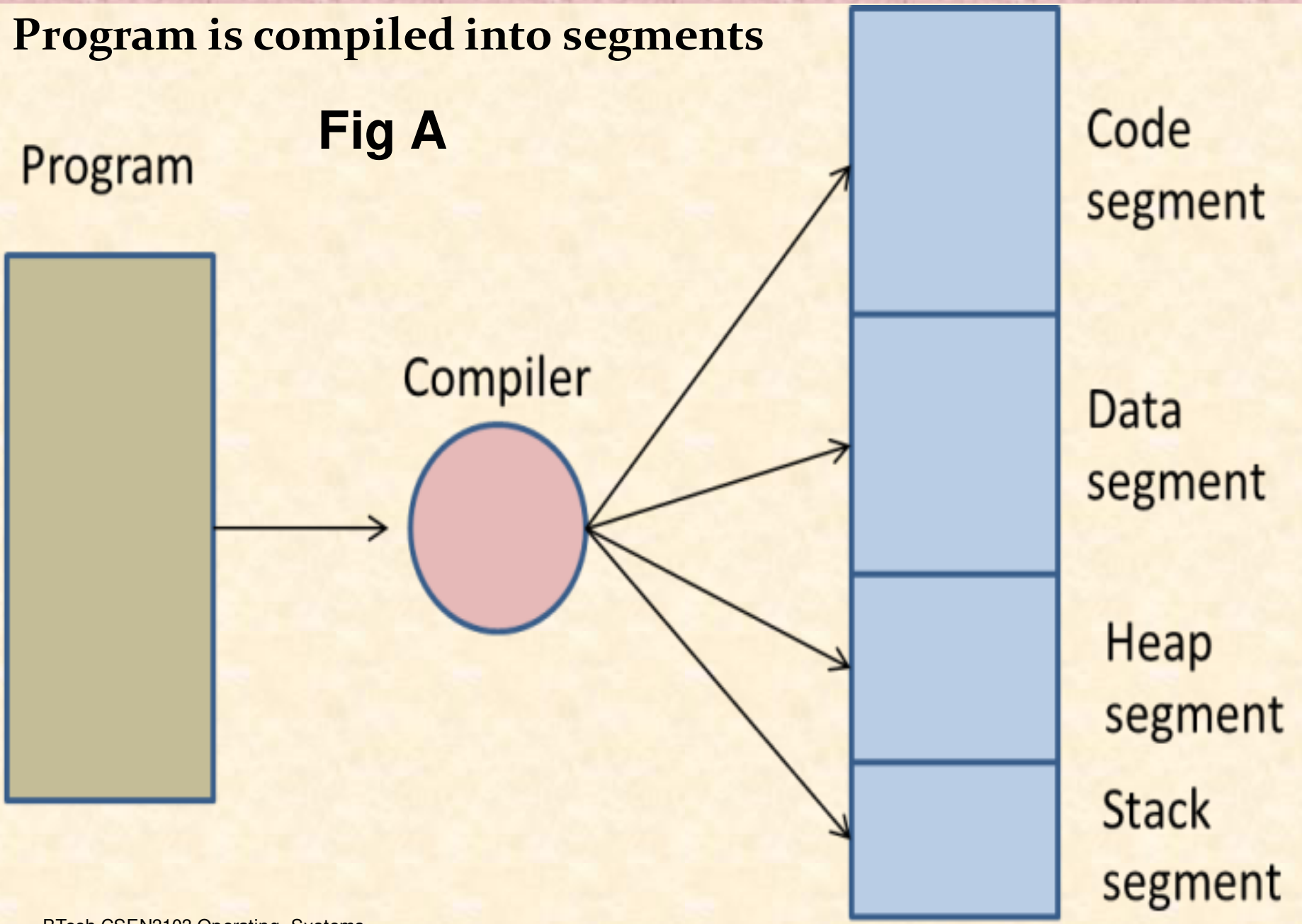
Segmentation

- A program consists of the following components:
  - **Code**: This part of the program contains the program instructions. These instructions are compiled by the compiler into the target machine code. It is marked ‘read only’, indicating that it cannot be modified.
  - **Data:** This part contains the data and the data structures defined by programmer.
  - **Heap:** This component act as a pool for dynamic memory allocation to the program.
  - **Stack:** The execution frames of functions are stored in a stack. The execution frames contain register values and contents of variables at a particular instant in the execution of the program. Thus, the compiler compiles the program into four types of relocatable segments – code, data, heap and stack as shown in fig A (next slide).

Segmentation

Program is compiled into segments

Fig A

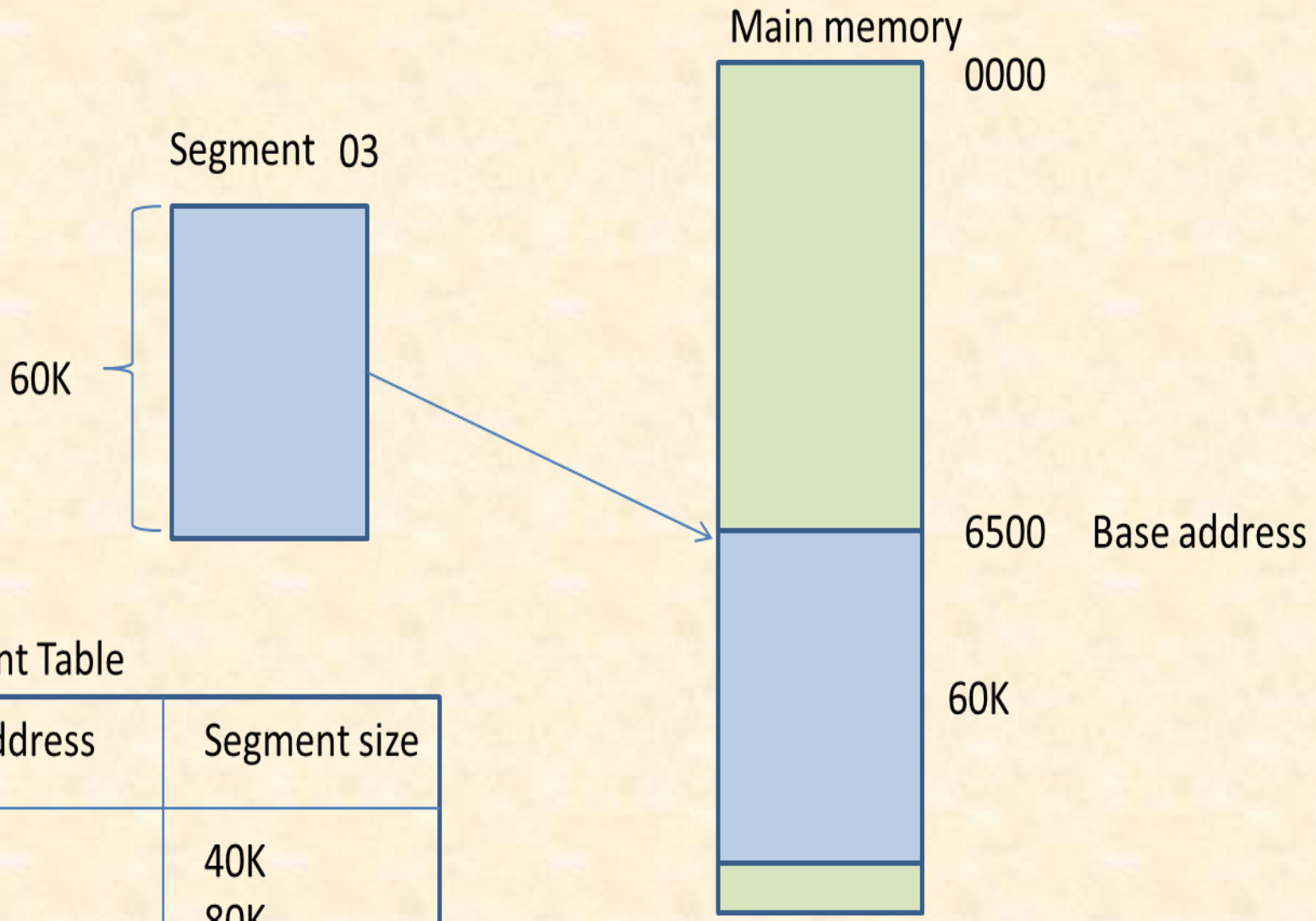


Segmentation

- When a segment is loaded into main memory, its starting main memory address is called the base address.
- The size of the segment is stored in a table called the segment table. The segment table is indexed as per the segment numbers shown in fig B.
- It may be noted that segment number 03 with size 60K has been loaded at a base address of 6500, and the corresponding entries have been made in the segment table at location 03.
- Now when the program is executed, the CPU will generate an address (id, d) of an instruction/data to be fetched (Fig C). This address is easily translated to its corresponding main memory address . The **segment id (say 03)** is searched in the segment table and its corresponding base address (**6500**) is loaded into a register. The **offset d** (say **510**) of the instruction is added to the base register, to obtain the main memory address of the instruction/data.

Segmentation

Fig B



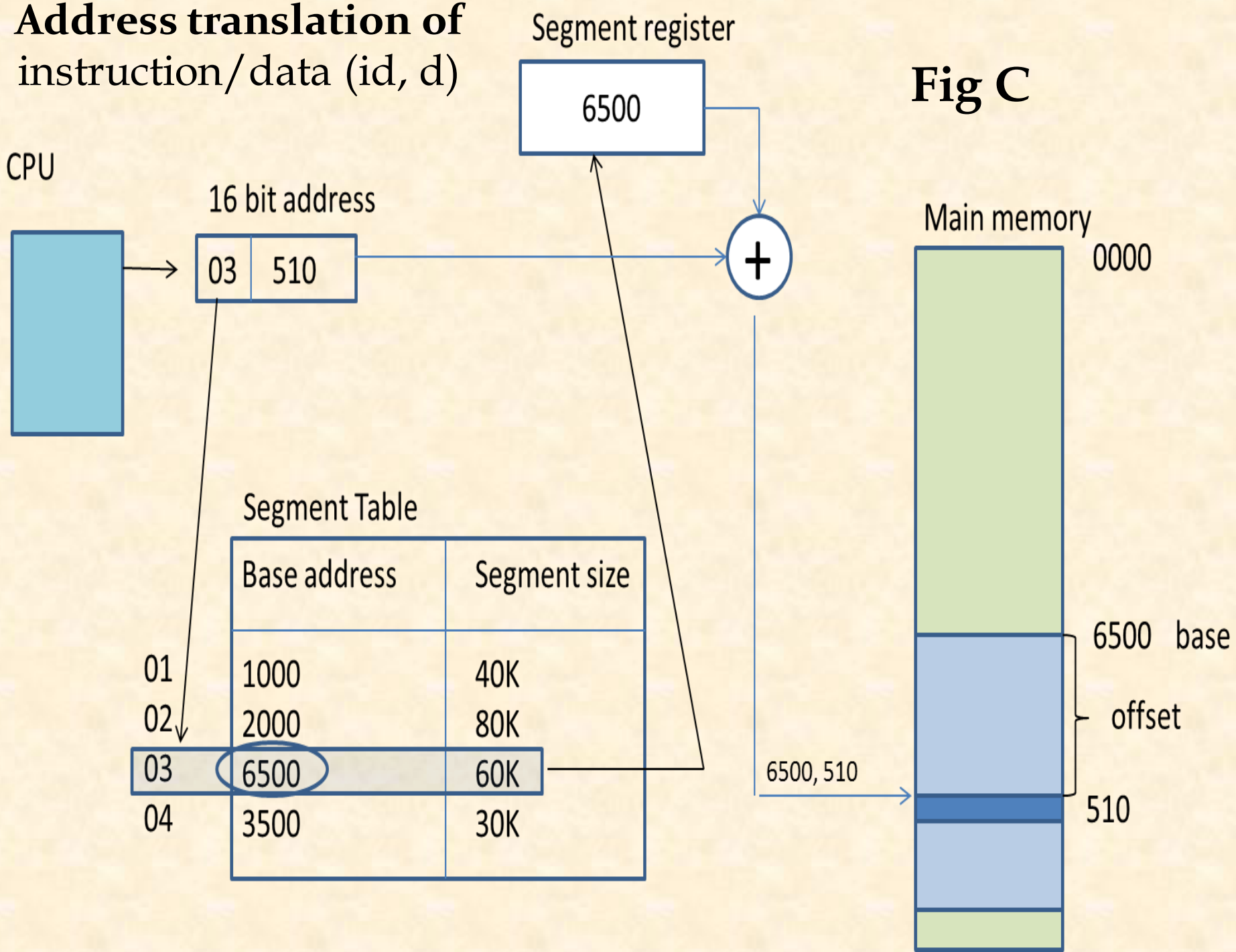
Segment Table

	Base address	Segment size
01	1000	40K
02	2000	80K
03	6500	60K
04	3500	30K

Segment 03 is loaded into main memory

Address translation of instruction/data (id, d)

Fig C



Paging

- The program and main memory are divided into equal sized partitions. The partitions of a program are called **pages**.
- The partitions of main memory are called **page frames**.
- Consider Fig P1 that shows the MM and program consisting of n and m numbers of page frames and pages respectively.
- It may be noted that both pages and page frames are equal as well as fixed sized partitions.
- Since the size of a page is the same as that of a page frame, any page of the program can be loaded into any available page frame of the main memory.
- *For each program*, OS maintains a data structure called a ***Page table.*** Assuming that page numbers 3,5 and 2 are loaded into page frames 12, 4 and 1 respectively, the corresponding entries of the page table will be shown in Fig P2. The page table is stored in the main memory of the computer system.

Paging → Pages and page frames

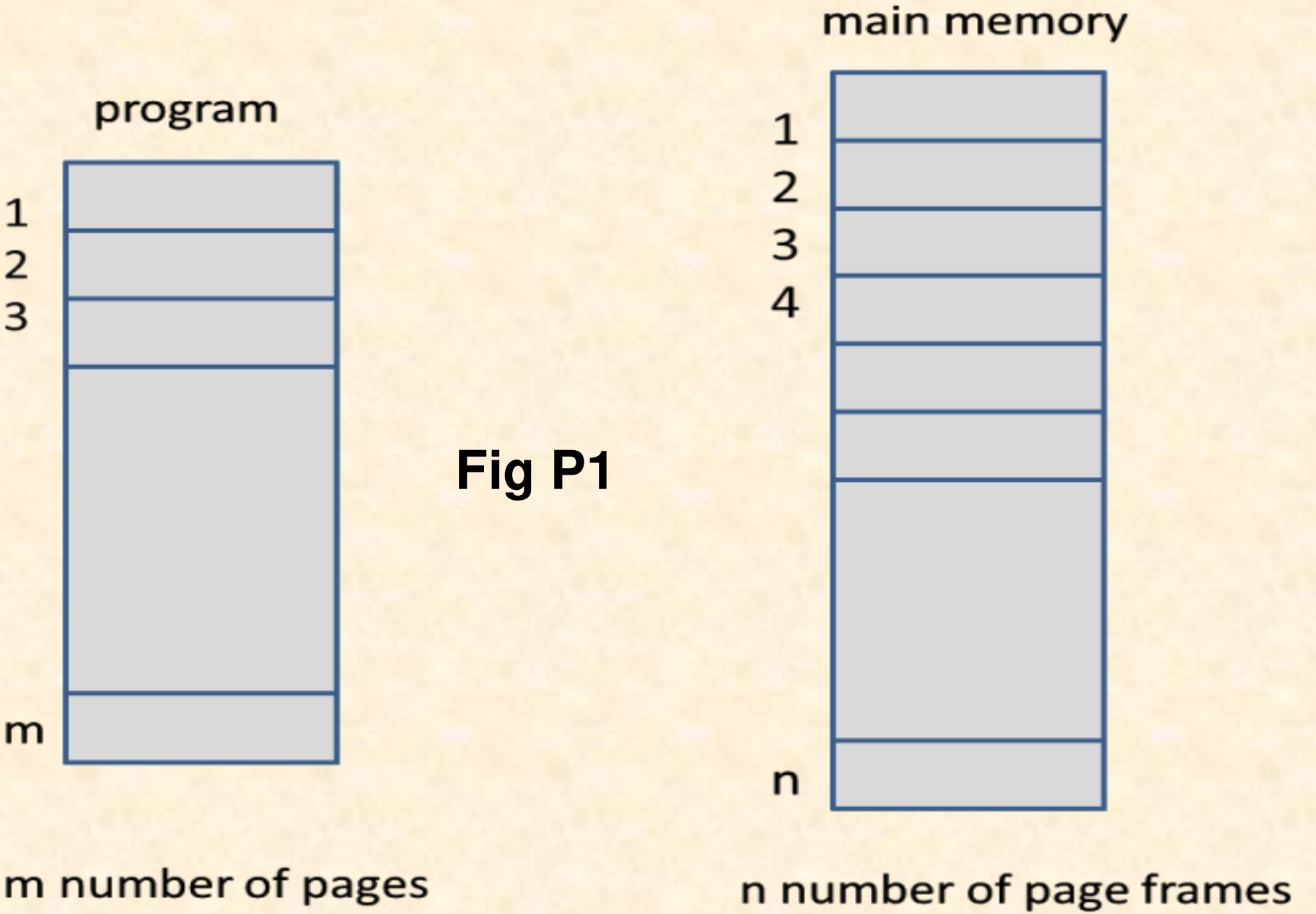
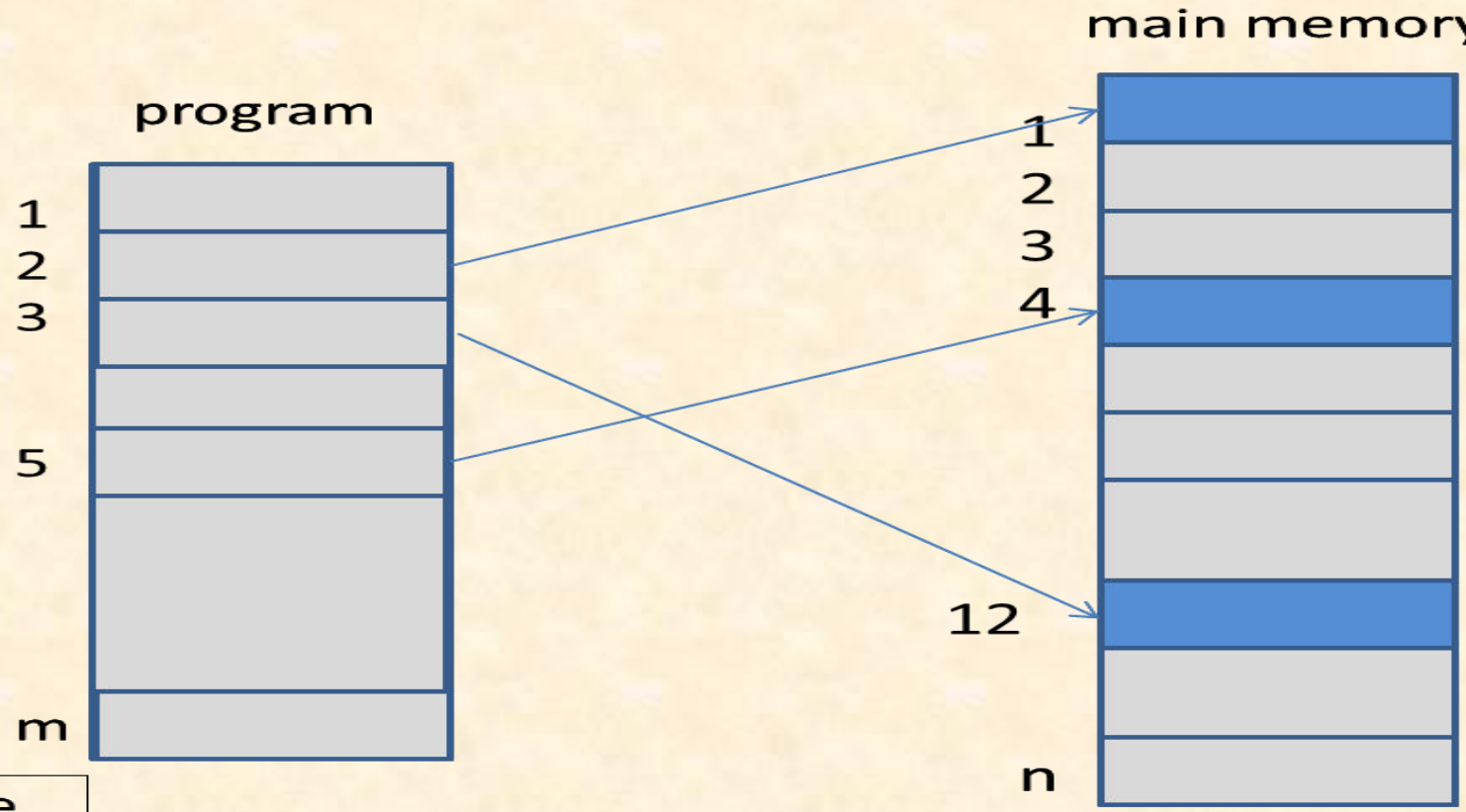


Fig P1

Paging



Page No	Page Frame No.
3	12
5	4
2	1

Fig P2

Paging

• When the program is executed, the CPU generates an address (page no., offset) of an instruction/data to be fetched (fig.P3). This address is easily translated to its corresponding memory address. The page no. (say 03) is located in page table, and its page frame number (12) is picked up from there. The offset, d (say 230), of the instruction within the page, is added to the page frame number to obtain the main memory address of the instruction/data. The address mapping scheme shown in fig.P3 is known as **direct mapping**.

• Since the page table is stored in the main memory, a complete primary storage cycle is required to access the page table. This means an access to a page involves the following two main memory cycles:

1. *One memory cycle is needed for accessing the page table to find the page frame number in which the required page is stored.*
2. *The second memory cycle is used to access the page frame to read the instruction from, or write to the page stored therein.*

Address Translation in Paging

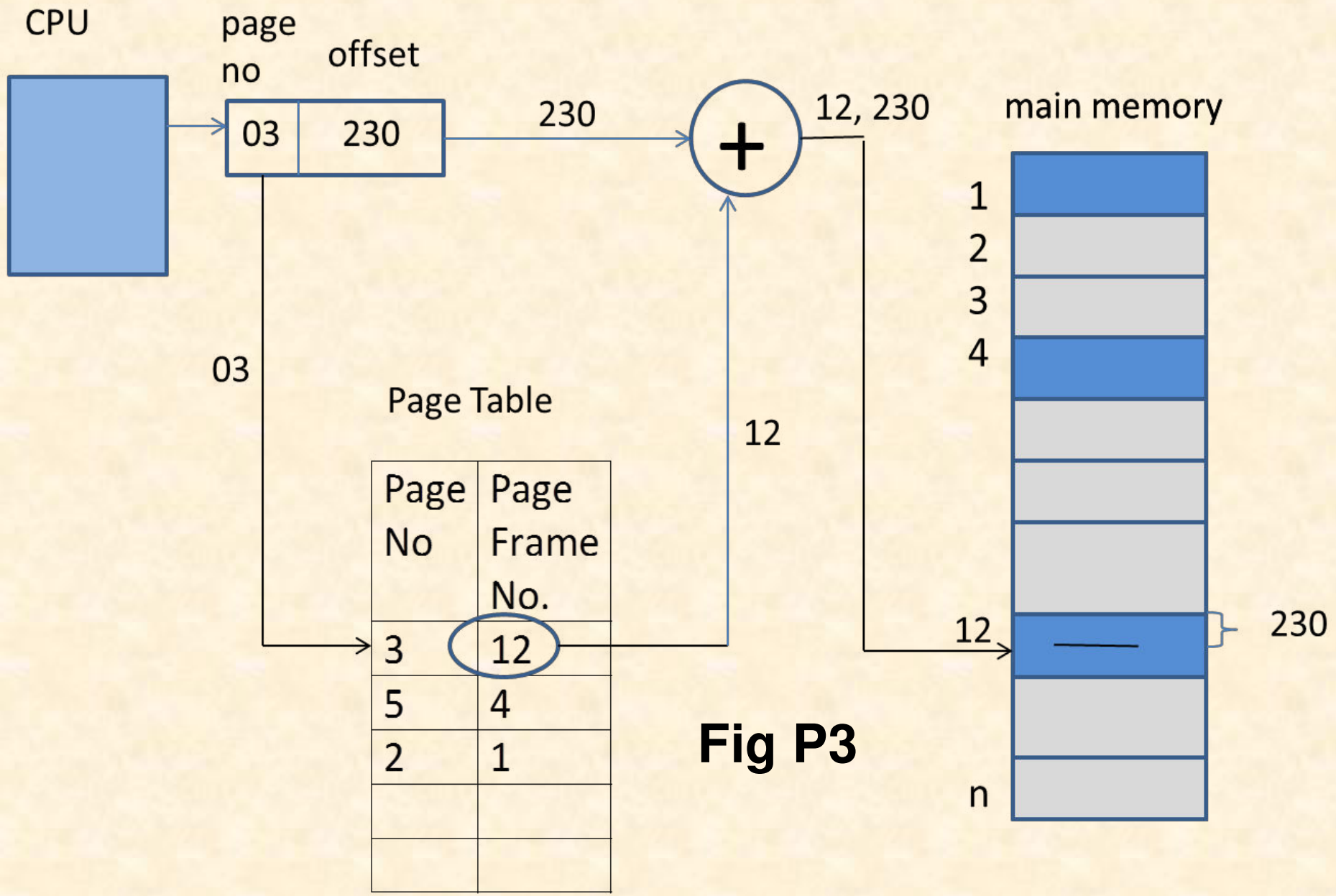


Fig P3

Hit Ratio and Miss Ratio

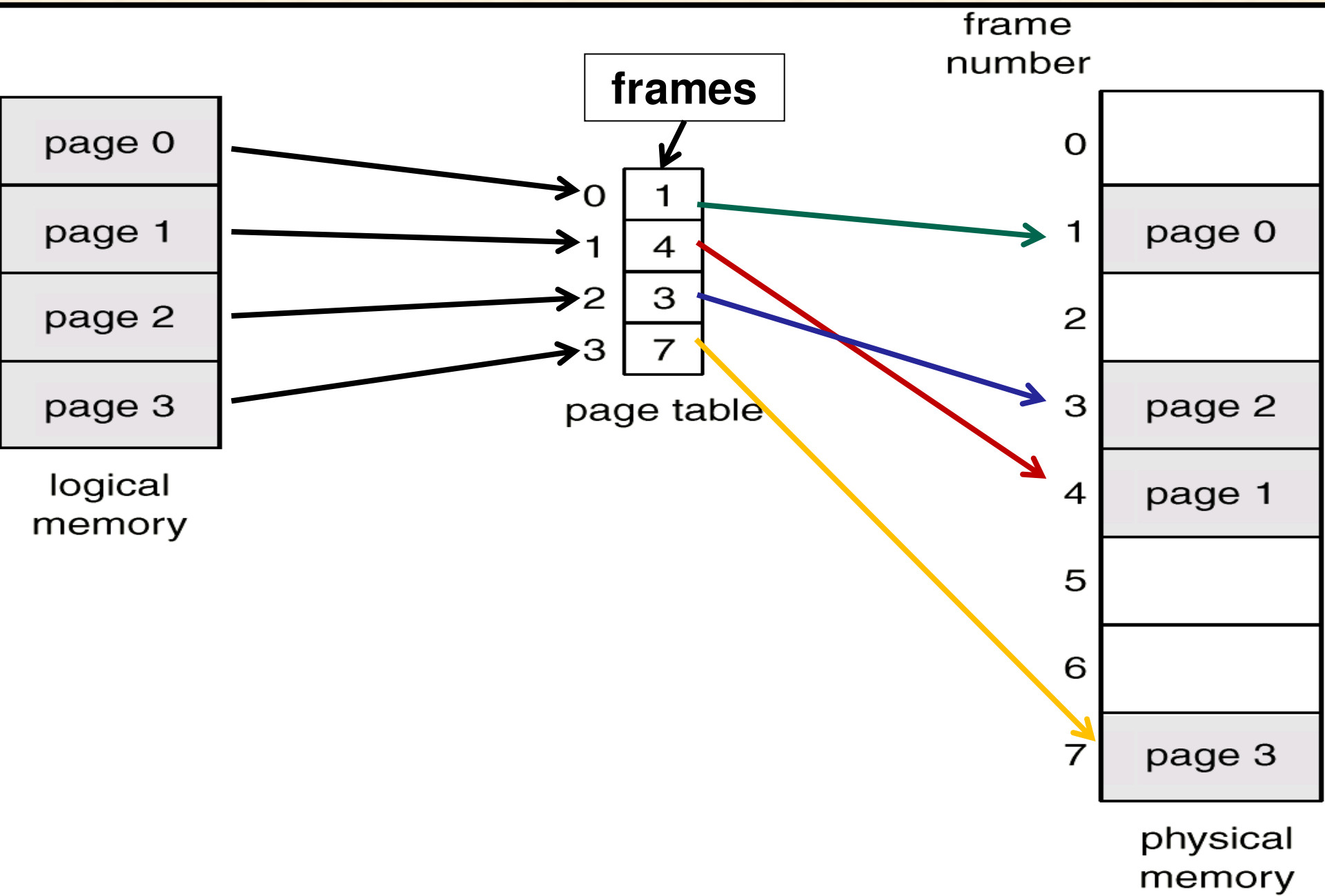
- Hit ratio: If a page table entry is found, then it is called a **hit**. In case of a **miss,** the page is bought from the hard disk.

$$\text{Hit ratio} = \frac{No\ of\ hits}{Total\ page\ references}$$

- Miss ratio: Ratio of number of misses to total number of references.

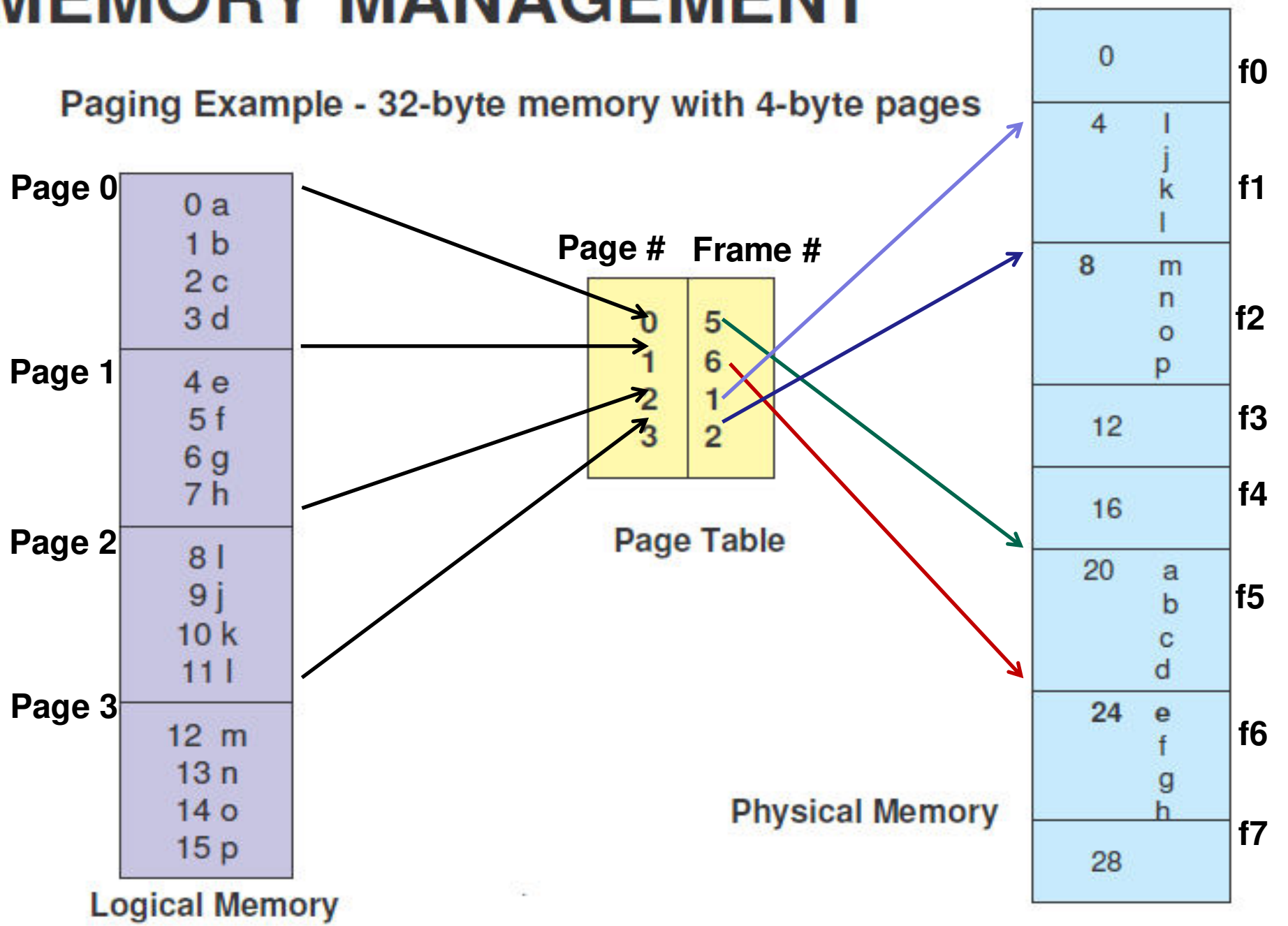
$$\text{Miss ratio} = \frac{No\ of\ misses}{Total\ page\ references}$$

Paging Example



MEMORY MANAGEMENT

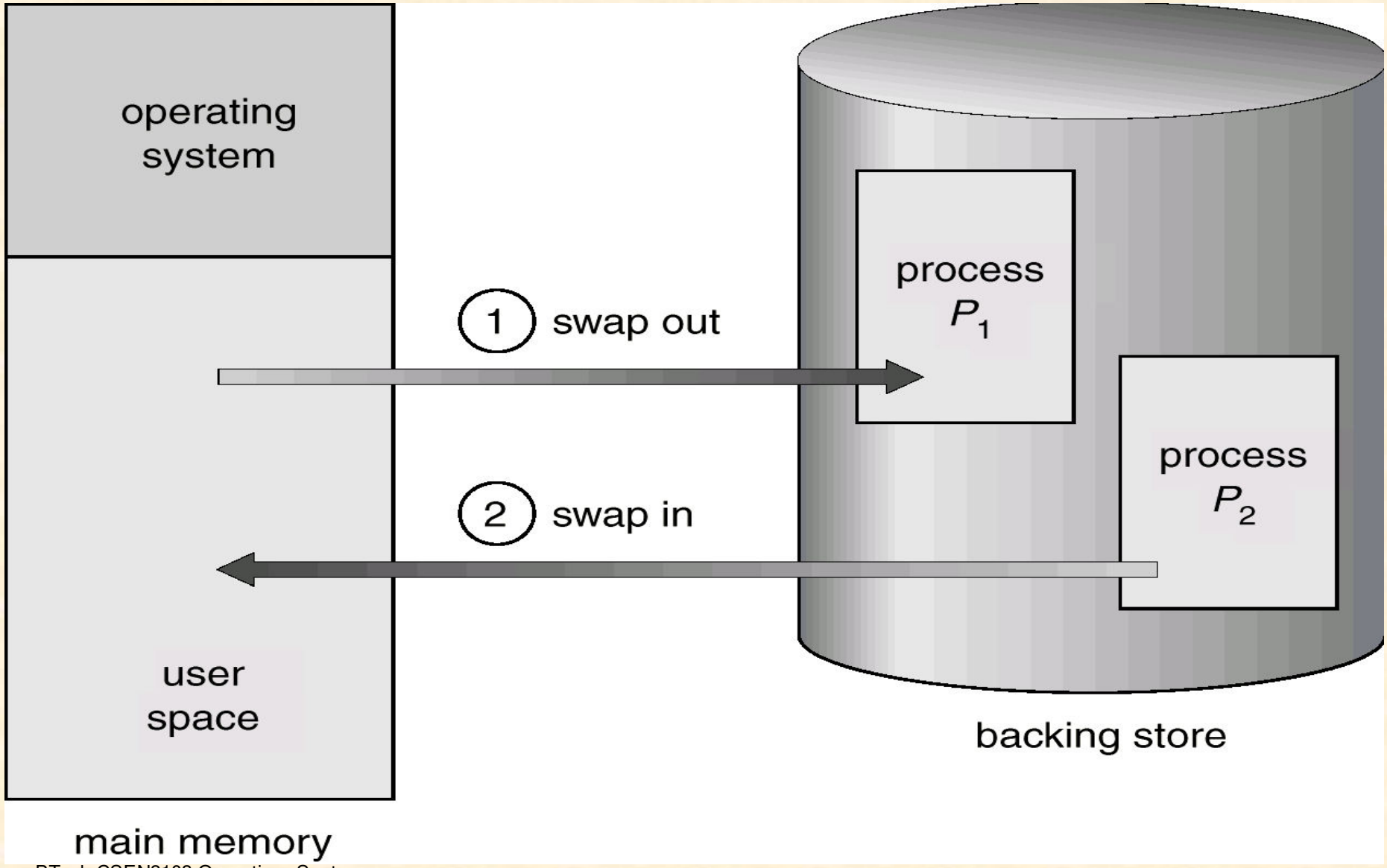
PAGING



Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- Modified versions of swapping are found on many systems, i.e., UNIX and Microsoft Windows.

Schematic View of Swapping



Swapping

Motivation:

Consider the following situation:

$P_1, \ldots, P_n$ are resident in memory and occupy all available memory

P_i forks to create a child

Principle:

- Space on fast disk (also called **Backing Store**) is used as additional / secondary memory
- Process can be *swapped out* temporarily from main memory to backing store; released memory is used for some other process; swapped process is *swapped in* later for continued execution


START

Assignments:

1. **If a process has 32 k bytes physical address space and the page size is 2048 bytes then the number of frames of that process is**

   a) 4 b) 8 c) 16 d) 32

Main memory space (physical address) = 32 k = $2^5 \times 2^{10} = 2^{15}$,
i.e., physical address space can fit into 15 bits;
Given Page size = 2048 bytes = 2^{11};
How many frames can fit into main memory?
Number of page frames = $2^{15} / 2^{11} = 2^4 = 16$ frames;

Paging

- Pages = $\frac{\text{Size of program}}{\text{size of page}} = \frac{\text{Size of logical memory}}{\text{size of page}}$

- No. of page frames = $\frac{\text{Size of main memory}}{\text{size of page}} = \frac{\text{Size of physical memory}}{\text{size of page}}$

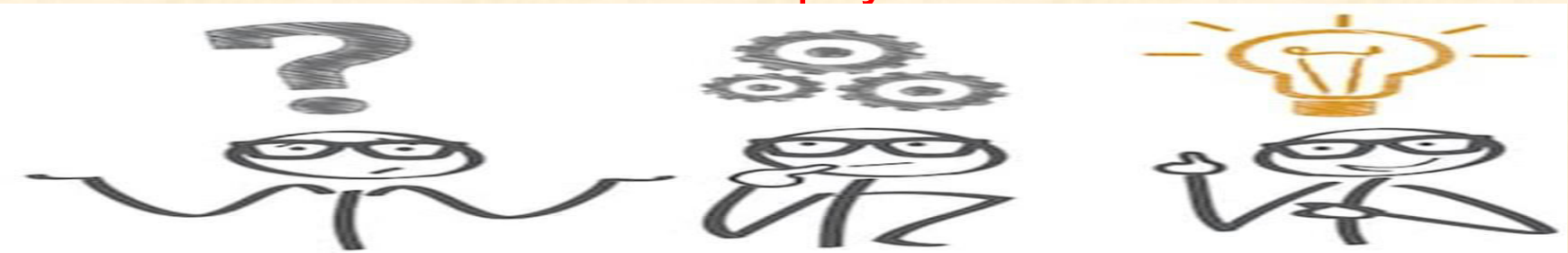
Assignments:

Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 page frames. Answer the following:

- **How many bits are there in the logical address?**
- **How many bits are there in the physical address?**

Ans: (i) Given: No of pages = 8.
Size of each page = 1024 words;
Size of logical memory = $8 * 1024 = 2^3 * 2^{10} = 2^{13}$ words
Thus the number of bits in the logical address = 13 bits

(ii) Given: No of page frames = 32.
Size of each page frame = size of page = 1024 words.
Size of physical memory = $32 * 1024 = 2^5 * 2^{10} = 2^{15}$ words.
Thus the number of bits in the physical address = 15 bits



Assignments:

Consider a paging system with the page table stored in the main memory. Answer the following:

- If the main memory access time is 200 ns, how long does a paged memory reference take?
- If we use TLB, and 75% of page table references are found in the TLB, then what is the effective reference time? (Assume that finding a page table entry in TLB takes zero time if the entry is found)

Ans: (i) Since access to a page involves two main memory cycles, the paged memory reference will take 2 * 200 = 400 ns.

(ii.) Hit ratio for the TLB can be defined as the ratio of the number of hits in the TLB to the total number of references made to the TLB, as given by the following expression:

Hit ratio (TLB) = No. of hits / Total page references

Let t1 and t2 be the access times of the TLB and the main memory. For a given hit ratio, the effective reference time can be calculated by the following expression:

T_{eff} = **hit ratio (TLB) * (t1 + t2) + miss ratio * (t1 + 2 * t2)** ------- (1)

Given hit ratio = 0.75, we have t1 = 0 ns, t2 = 200 ns;

Putting the data in expression (1), we get:

T_{eff} = **hit ratio (TLB) * (t1 + t2) + miss ratio * (t1 + 2 * t2)**

T_{eff} = **0.75 * (0 + 200) + (1 – 0.75) * (0 + 2 * 200) = 250 ns**

Assignments:

Consider a logical address space of sixteen pages of 1024 words each. If the physical memory has 32 page frames, then,

- How many bits are there in the physical address space?
- How many bits are there in the logical address space?

Ans:

1. Let the number of bits in physical memory = ***n***
   Therefore, the total address space of the physical memory $(2^n) = 32 * 1024 = 2^5 * 2^{10}$ words $= 2^{15}$ words. Hence, n = 15;
   Thus, number of bits in the physical memory = n = 15 bits.

2. Let the number of bits in the logical memory = ***n***
   Therefore, the total address space of the logical memory (2^n) $= 16 * 1024 = 2^4 * 2^{10}$ words $= 2^{14}$ words. Hence, n = 14;

Thus, number of bits in the **logical memory** = n = 14 bits

Assignments:

How many pages of size 512 words each, are contained in a program with a logical address having 16 bits?

Ans: Size of logical memory = 2^{16} words.
Size of page = 512 words = 2^9 words.

Number of pages = size of logical memory / size of page = $2^{16} / 2^9 = 2^{16-9} = 2^7$