Logical versus Physical Address Space

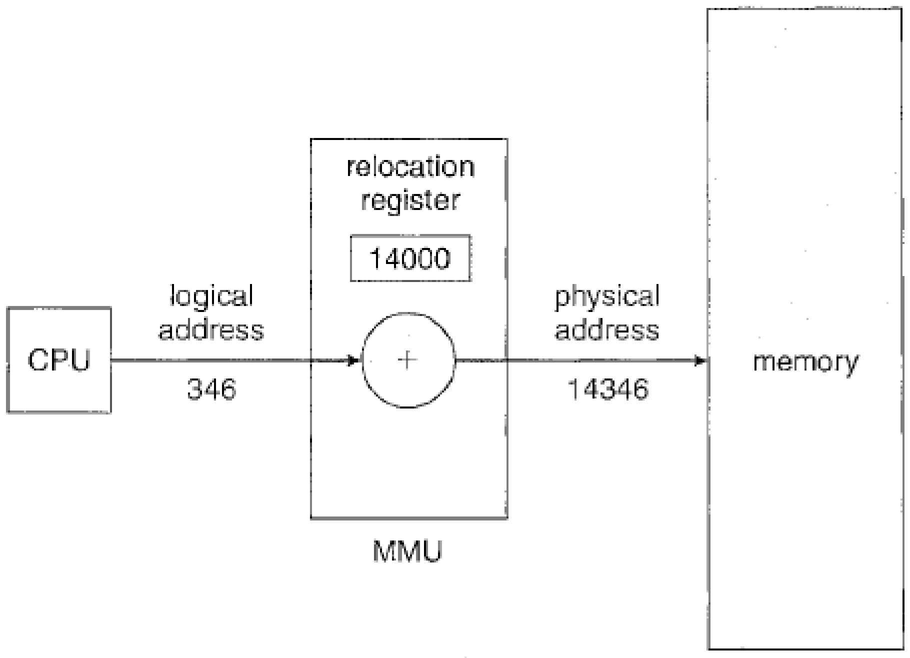
- An address generated by the CPU is commonly referred to as **Logical –Address**. Whereas an address seen by the memory unit-that is, the one loaded into the memory-address register of the memory-is commonly referred to as **Physical Address**.

- The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address.**

- 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.**

- The run-time mapping from virtual to physical addresses is done by a hardware device called the Memory Management Unit (MMU).

- We illustrate this mapping with a simple MMU scheme that is a generalization of the **base-register scheme** The base register is now called a **relocation register.** The value in the relocation register is *added* to every address generated by a user process at the time the address is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.

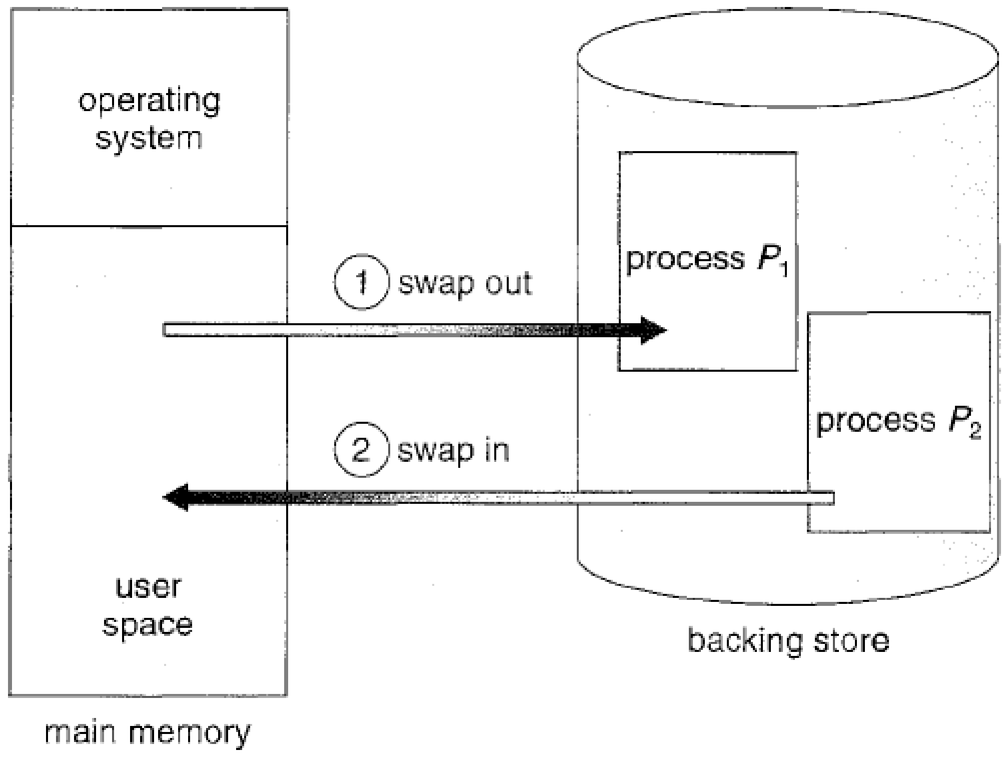


- The user program never sees the *real* physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses-all as the number 346.

- 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 generates only logical addresses and thinks that the process runs in locations 0 to *max.*

Swapping

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed.
- In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process.
- Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU.
- In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.



- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding.

- Swapping requires a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

- The system maintains a Ready Queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.

Contiguous Memory Allocation

- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

- In. contiguous memory allocation, each process is contained in a single contiguous section of memory**.**

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partition**. Each partition may contain exactly one process.

- Thus, the degree multiprogramming is bound by the number of partitions. In this multiple partition method when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

- **In variable partitioning** the scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory a **hole.**

- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory which the operating system may then fill with another process from the input queue.

- Memory is allocated to processes until finally, the memory requirements of the next process cannot be satisfied - that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

    - In general as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory.
    - When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
    - When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
    - If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

- This procedure is a particular instance of the general **dynamic storage allocation problem,** which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit, best-fit and worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

  - **First fit.** Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

  - **Best fit.** Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

  - **Worst fit**. Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Problem:

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.

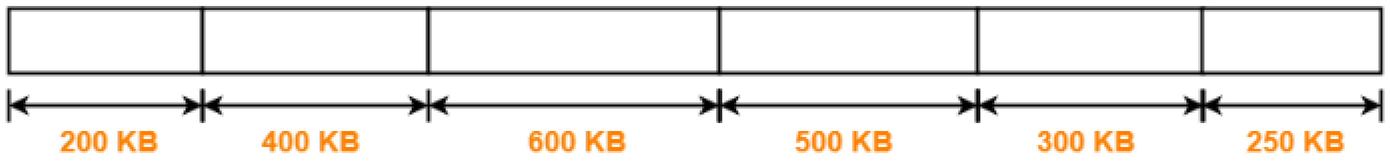
Perform the allocation of processes using-

1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

Solution-

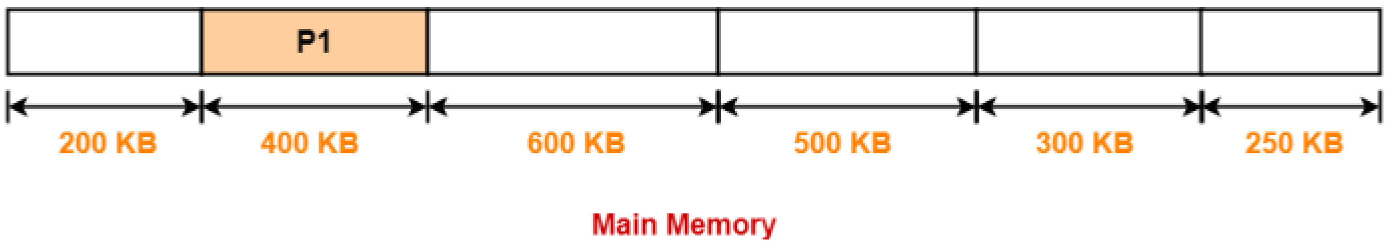
According to question,

The main memory has been divided into fixed size partitions as-

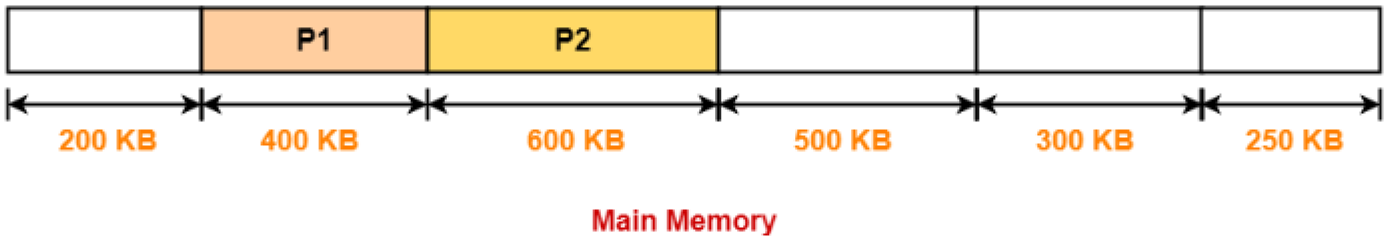


Allocation Using First Fit Algorithm-

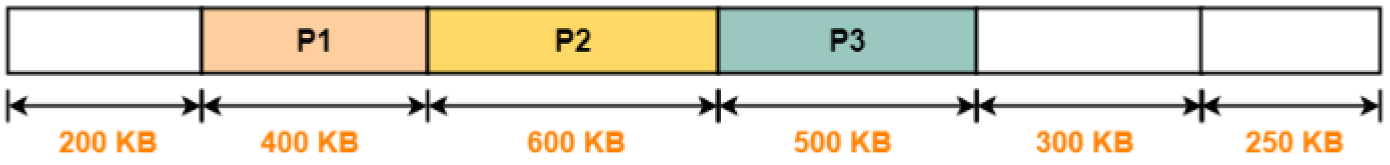
Step-01:



Step-02:



Step-03:

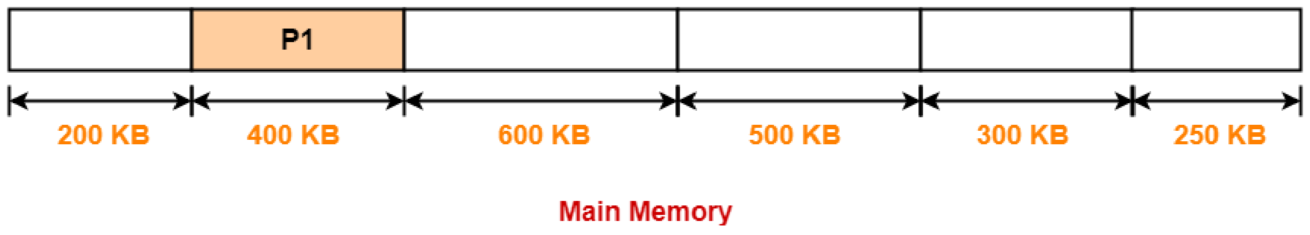


Step-04:

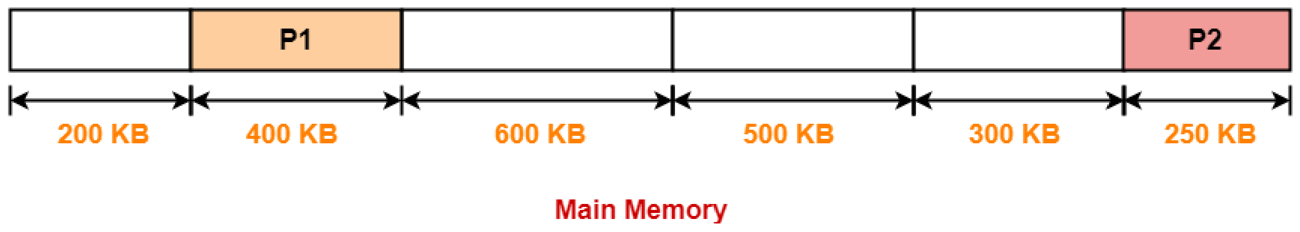
- Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P4 is available.

Allocation Using Best Fit Algorithm-

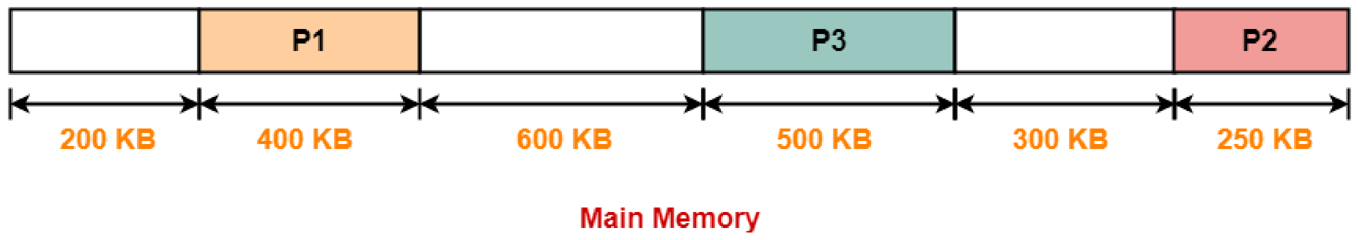
Step-01:



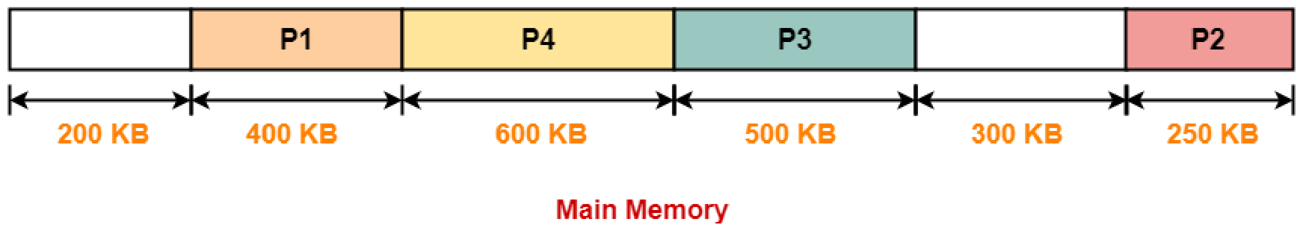
Step-02:



Step-03:

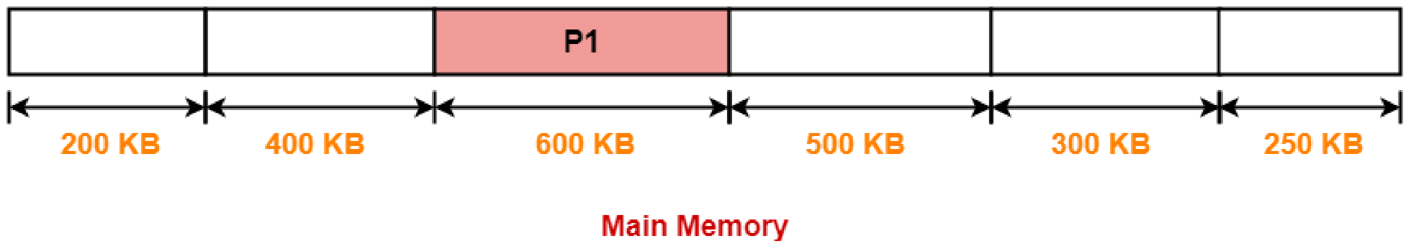


Step-04:

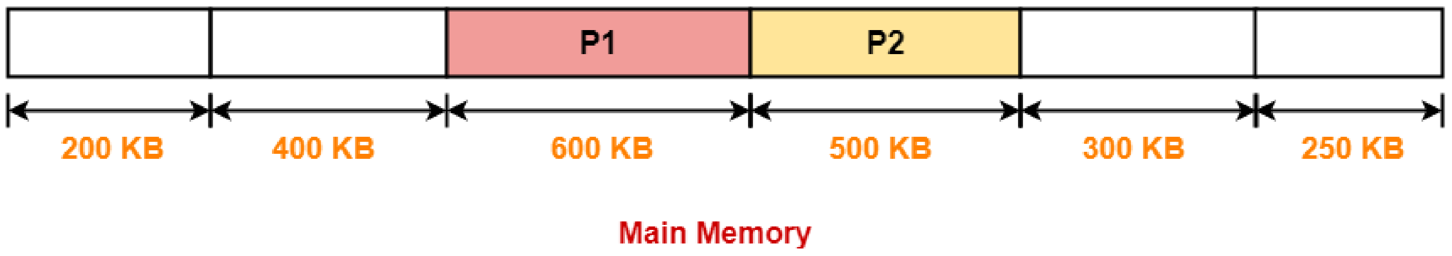


Allocation Using Worst Fit Algorithm-

Step-01:



Step-02:



Step-03:

- Process P3 and Process P4 can not be allocated the memory.
- This is because no partition of size greater than or equal to the size of process P3 and process P4 is available.

Fragmentation

- Fragmentation is an unwanted problem where the memory blocks cannot be allocated to the processes due to their small size and the blocks remain unused. It can also be understood as when the processes are loaded and removed from the memory they create free space or hole in the memory and these small blocks cannot be allocated to new upcoming processes and results in inefficient use of memory. Basically, there are two types of fragmentation:

  - Internal Fragmentation
  - External Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe.

- In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.

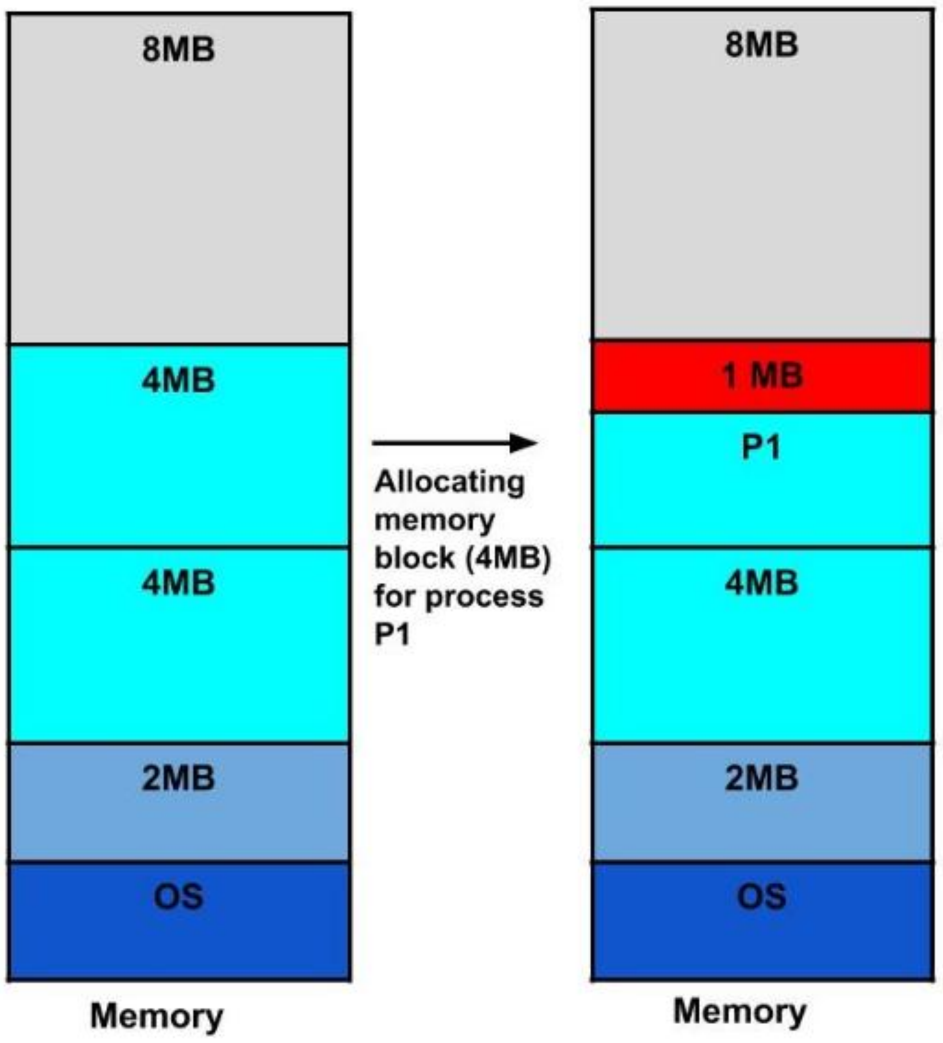
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation-** unused memory that is internal to a partition.

- One solution to the problem of external fragmentation is **Compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block

Example Internal fragmentation:

Suppose there is fixed partitioning (i.e. the memory blocks are of fixed sizes) is used for memory allocation in RAM. These sizes are 2MB, 4MB, 4MB, 8MB. Some part of this RAM is occupied by the Operating System (OS)

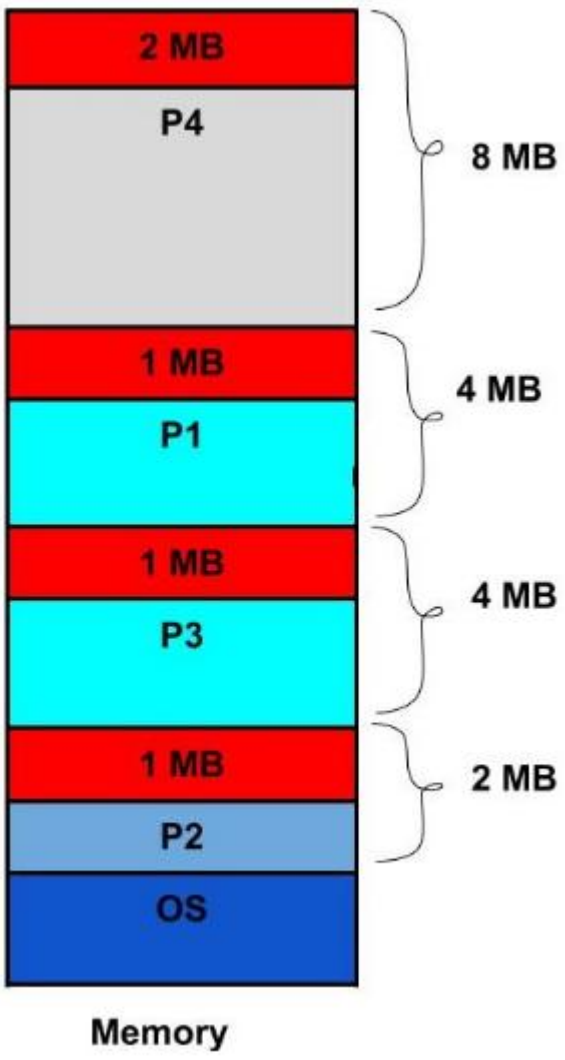
Now, suppose a process P1 of size 3MB comes and it gets memory block of size 4MB. So, the 1MB that is free in this block is wasted and this space can't be utilized for allocating memory to some other process. This is called **internal fragmentation.**



Example of External Fragmentation:

Suppose in the above example, if three new processes P2, P3, and P4 come of sizes 2MB, 3MB, and 6MB respectively. Now, these processes get memory blocks of size 2MB, 4MB and 8MB respectively allocated.

So, now if we closely analyze this situation then process P3 (unused 1MB)and P4(unused 2MB) are again causing internal fragmentation. So, a total of 4MB (1MB (due to process P1) + 1MB (due to process P3) + 2MB (due to process P4)) is unused due to internal fragmentation.



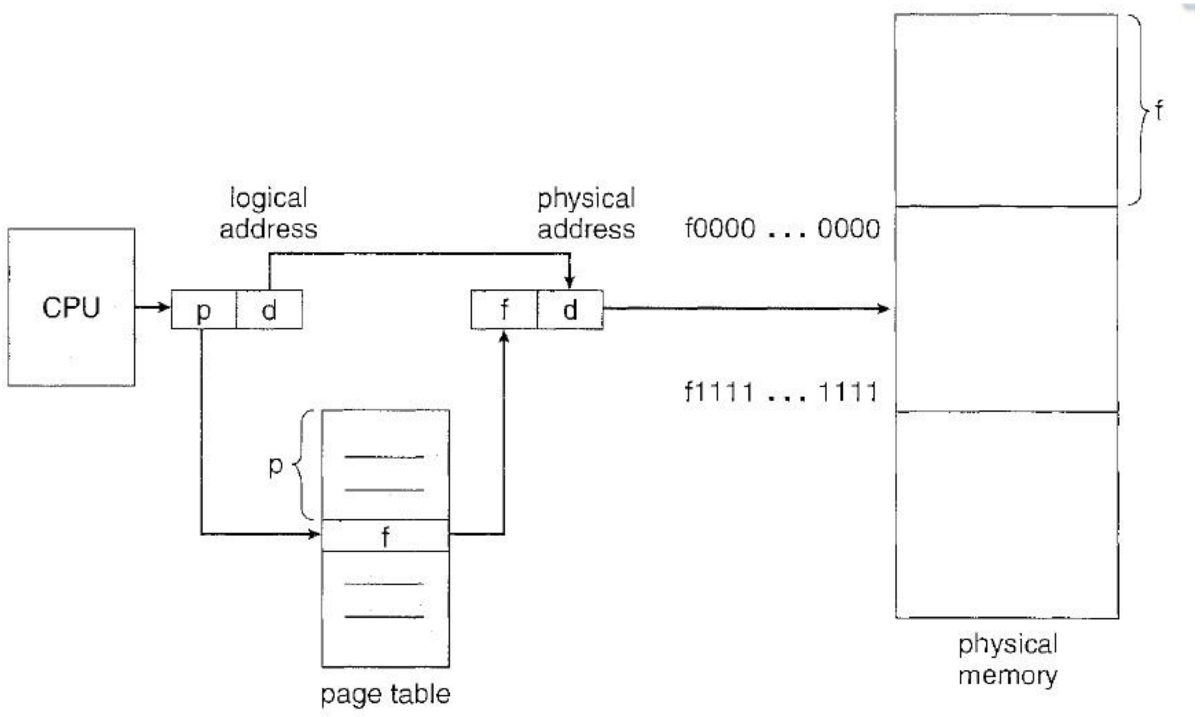
Now, suppose a new process of 4 MB comes. Though **we have a total space of 4MB** still **we can’t allocate this memory** to the process. This is called **external fragmentation**.

Paging

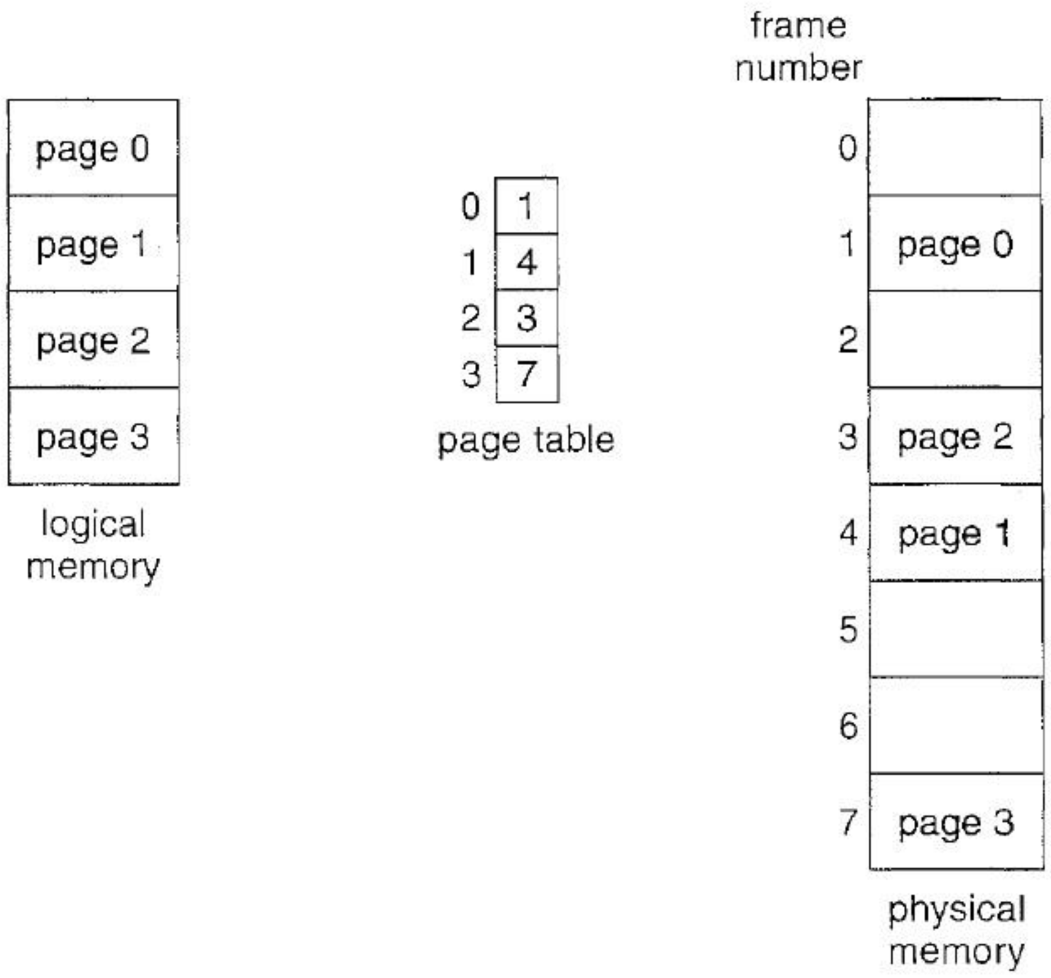
- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called page.

<u>Hardware</u>

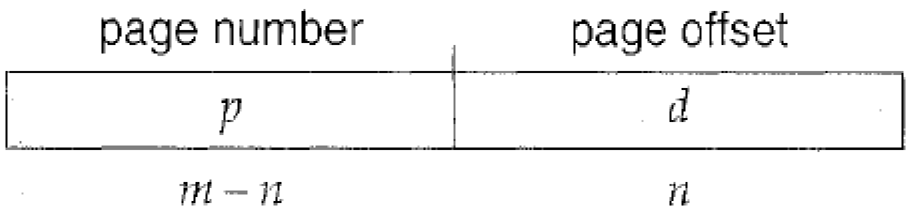
- Every address generated the CPU is divided into two parts: a page number {p) and a page offset {d}. The page number is used as an index into a Page Table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



The paging model of memory is shown in Figure



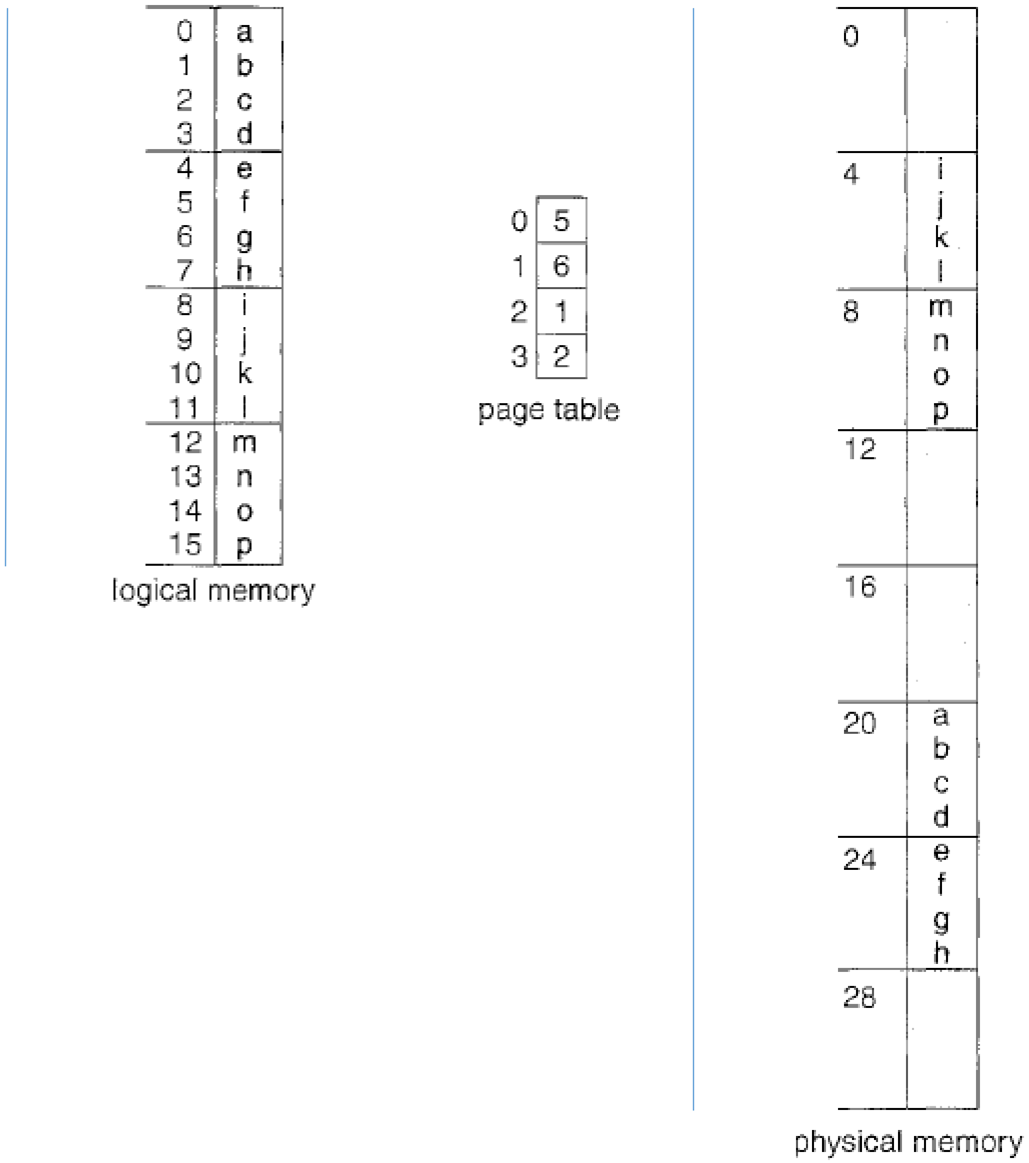
- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- If the size of the logical address space is 2^m, and a page size is 2^n addressing units (bytes or words) then the high order m- n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



Where *p* is an index into the page table and *d* is the displacement within the page

Example

- Consider the memory in given Figure. Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory.
- Logical address 0 is page 0 offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 x 4) + 0].

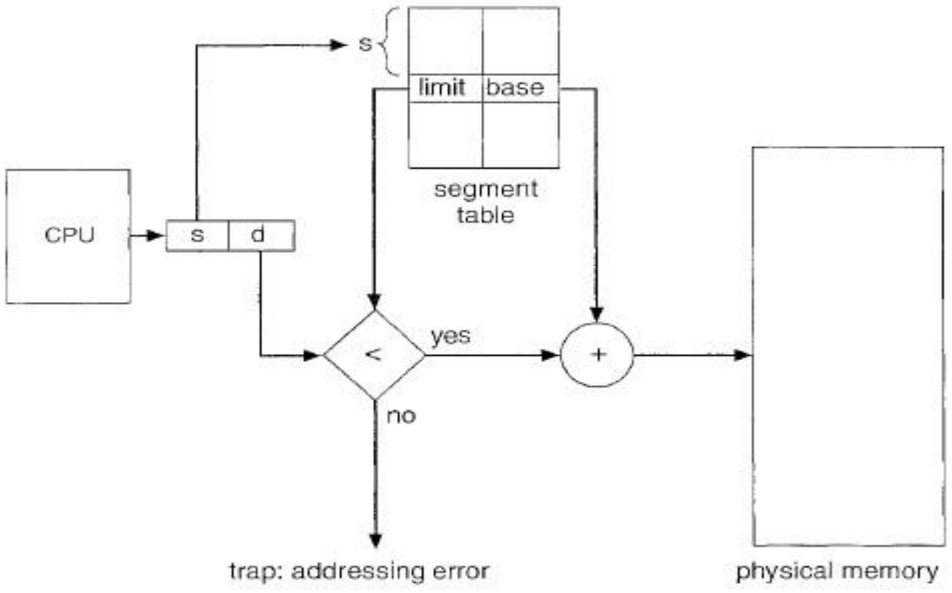


Segmentation

- Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments.
- Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.
- Thus, a logical address consists of a two tuple:
  <segment-number, offset>.

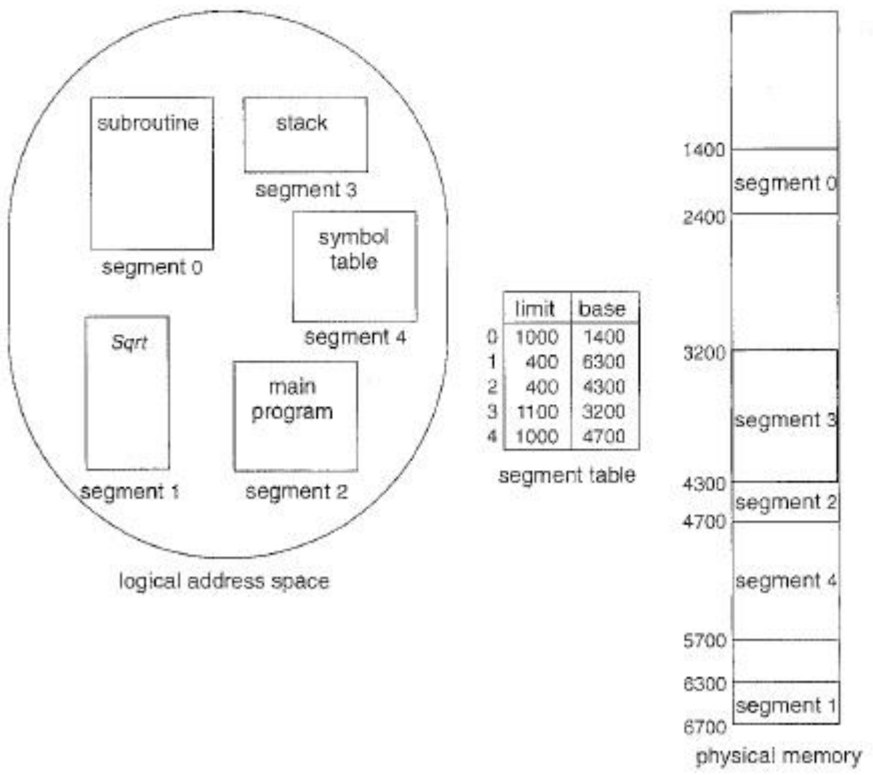
Hardware

- The user can now refer to objects in the program by a two-dimensional address but the actual physical memory is a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is effected by a Segment Table. Each entry in the segment table has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
- The use of a segment table is illustrated given Figure. A logical address consists of two parts: a segment number, s, and an offset into that segment, d.
- The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.



Example

- Consider the situation shown in next Figure. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.



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

Segmented Paging

Pure segmentation is not very popular and not being used in many of the operating systems. However, Segmentation can be combined with Paging to get the best features out of both the techniques.

In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

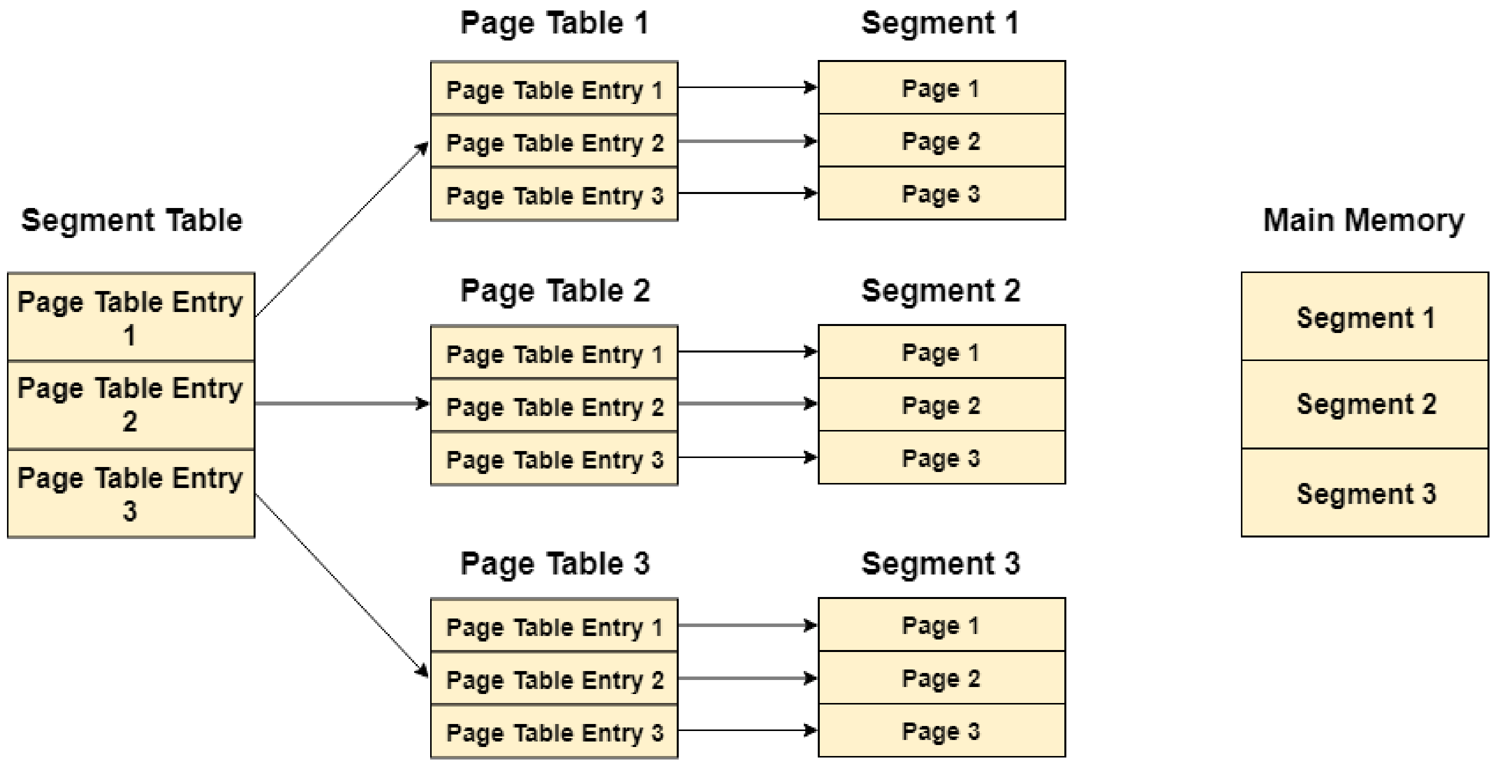
1. Pages are smaller than segments.
2. Each Segment has a page table which means every program has multiple page tables.
3. The logical address is represented as Segment Number (base address), Page number and page offset.

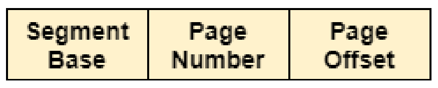
Segment Number → It points to the appropriate Segment Number.

Page Number → It Points to the exact page within the segment

Page Offset → Used as an offset within the page frame

Each Page table contains the various information about every page of the segment. The Segment Table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.



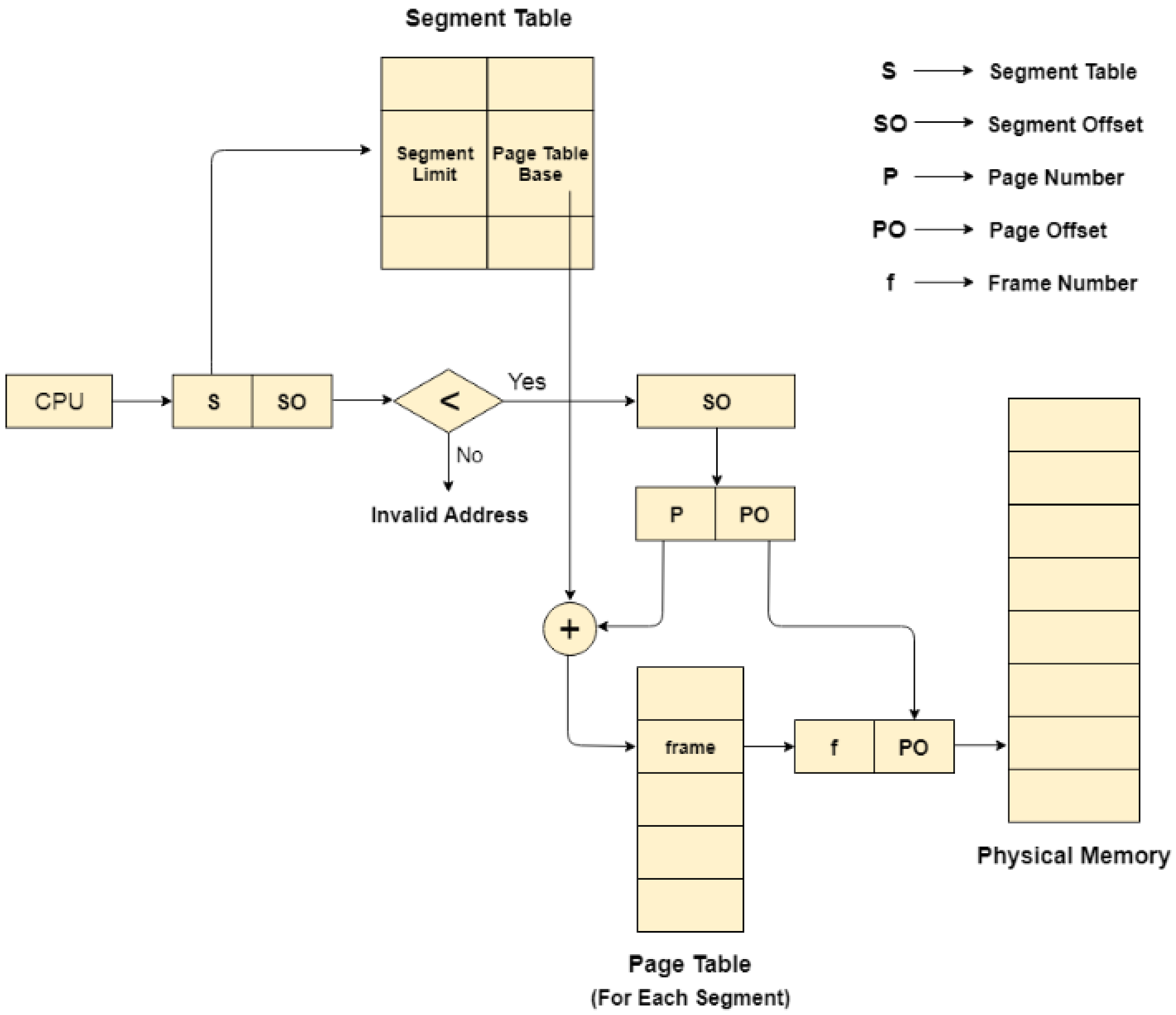


Logical Address

Translation of logical address to physical address

The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset. The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset. To map the exact page number in the page table, the page number is added into the page table base.

The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

1. It reduces memory usage.
2. Page table size is limited by the segment size.
3. Segment table has only one entry corresponding to one actual segment.
4. External Fragmentation is not there.
5. It simplifies memory allocation.

Disadvantages of Segmented Paging

1. Internal Fragmentation will be there.
2. The complexity level will be much higher as compare to paging.
3. Page Tables need to be contiguously stored in the memory.