UNIT-4
Memory Management

Syllabus:

Memory Management

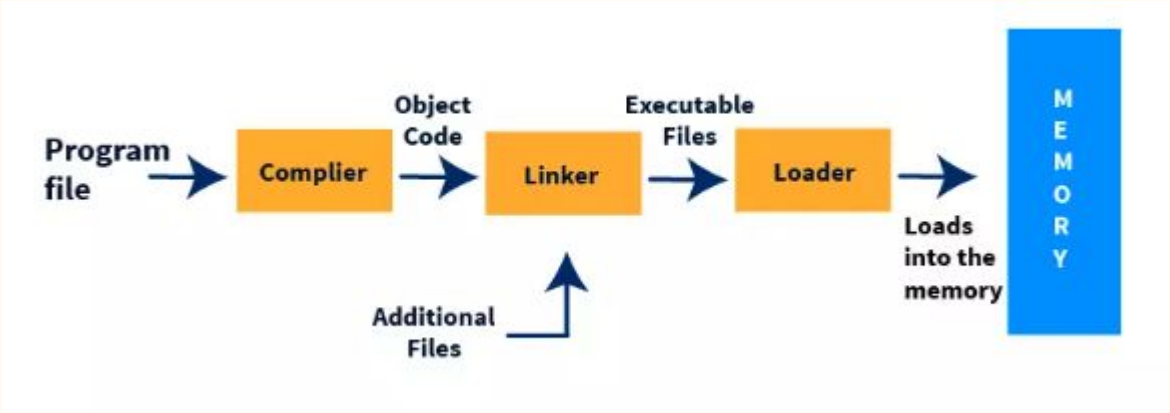
1. Swapping
2. Contiguous Memory Allocation
3. Paging
4. Structure of the Page Table
5. Segmentation

Virtual Memory Management

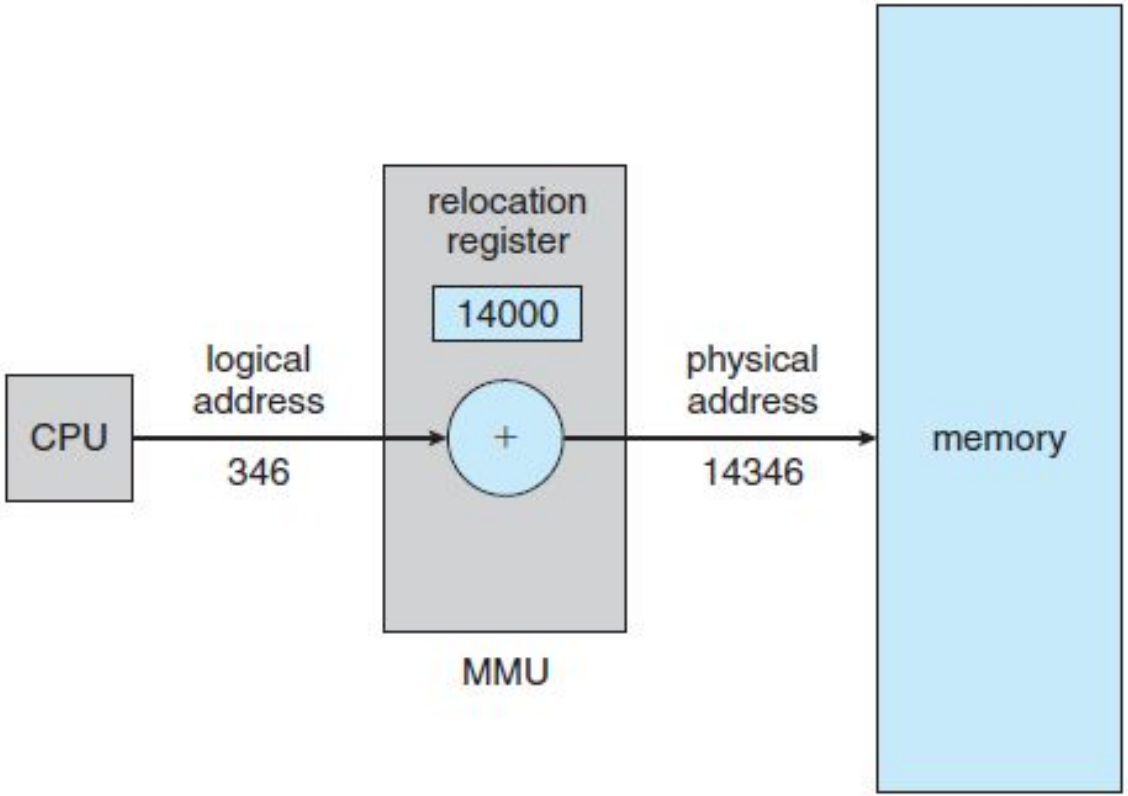
1. Virtual Memory
2. Demand Paging
3. Page-Replacement Algorithms
4. Thrashing

Address Binding

1. Compile Time Binding
2. Load Time Binding
3. Execution Time Binding



Logical Versus Physical Address Space


relocation
register
14000
logical
address
346
CPU
+
physical
address
14346
memory
MMU

In operating systems, logical and physical addresses are used to manage and access memory.

Logical address:

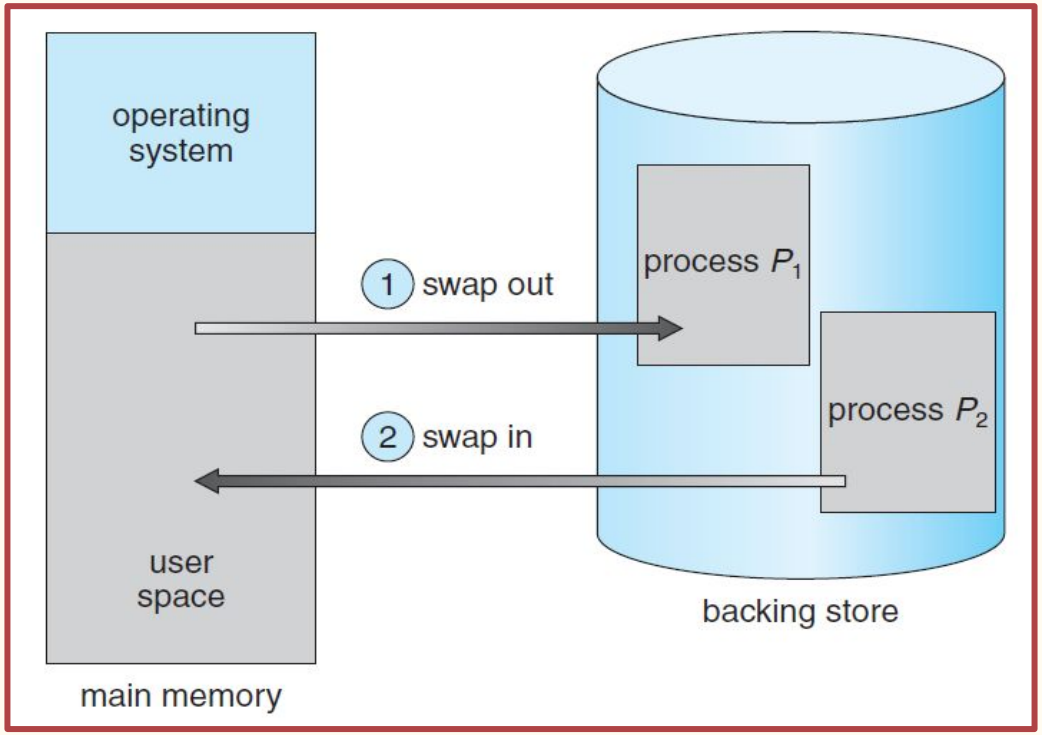
- **An address generated by the CPU during program execution** is commonly referred to as a logical address.
- A logical address, also known as a virtual address.
- The set of all logical addresses generated by a program is a logical address space.
- The process accesses memory using logical addresses, which are translated into physical addresses **by the operating system**.

Physical address:

- ➔ A physical address is the actual address in main memory where data is stored.
- ➔ It is a location in physical memory, as opposed to a virtual address.
- ➔ 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).**
- ➔ 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 location14000; an access to location 346 is mapped to location 14346.
- ➔ The MMU uses **a page table** to translate logical addresses into physical addresses.

SWAPPING

- To increase CPU utilization in multiprogramming, a memory management scheme known as swapping can be used.
- Standard swapping involves **moving processes between main memory and a backing store(Hard disk)**.
- Swapping is a memory management scheme in which **any process can be temporarily swapped from main memory to secondary memory** so that the main memory can be made available for other processes.
- The purpose of swapping in an operating system is to access data on a hard disc and move it to RAM so that application programs can use it.
- It's important to remember that **swapping is only used when data isn't available in RAM**.



Swapping has been subdivided into two concepts: swap-in and swap-out.

- **Swap-out is a technique for moving a process from RAM to the hard disc.**
- **Swap-in is a method of transferring a program from a hard disc to main memory, or RAM.**

- System maintains **a ready queue** of ready-to-run processes which have memory images on disk.
- 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.
- The CPU scheduler determines which processes are swapped in and which are swapped out.
  - Consider a multiprogramming environment that employs a priority-based scheduling algorithm. When **a high-priority** process enters the Ready queue, **a low-priority process is swapped out so the high-priority process can be loaded and executed**.
  - When a high-priority process terminates, the low priority process is swapped back into memory to continue its execution.

- ➔ The **context-switch time** in such a swapping system is fairly high.
- ➔ To get an idea of the context-switch time, let's assume that **the user process is 100 MB in size and hard disk with a transfer rate of 50 MB per second.**
- ➔ The actual transfer of the 100 MB process to or from main memory takes

  Time = Processor Size / Transfer Rate

  = 100 MB/50 MB per second

  = 2 seconds or 2000 milliseconds

- ➔ The swap time is 2000 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds.

Contiguous Memory Allocation

- **Contiguous Memory Allocation** is a type of memory allocation technique where processes are allotted a continuous block of space in memory.
- This block can be of **fixed size** for all the processes in **a fixed size partition scheme** or can be of **variable size** depending on the requirements of the process in **a variable size partition scheme.**
- The main memory must accommodate both the operating system and the various user processes.We therefore need to allocate main memory in the most efficient way possible.
- The Main memory is usually divided into two partitions:
  - one for the **operating system**
  - one for **the user processes**.

We can place the operating system in either low memory or high memory.

Memory Protection

- ★ When the CPU scheduler selects a process for execution, **the dispatcher loads the relocation and limit registers with the correct values.**
- ★ Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data.
- ★ In Memory protection, we have to protect the operating system from user processes and which can be done by using **a relocation register with a limit register.**
- ★ The relocation register has the value of the smallest physical address whereas the limit register has the range of the logical addresses.
- ★ These two registers have some conditions like each logical address must be less than the limit register.
- ★ The memory management unit(MMU) is used to translate the logical address with the value in the relocation register dynamically after which the translated (or mapped) address is then sent to memory.

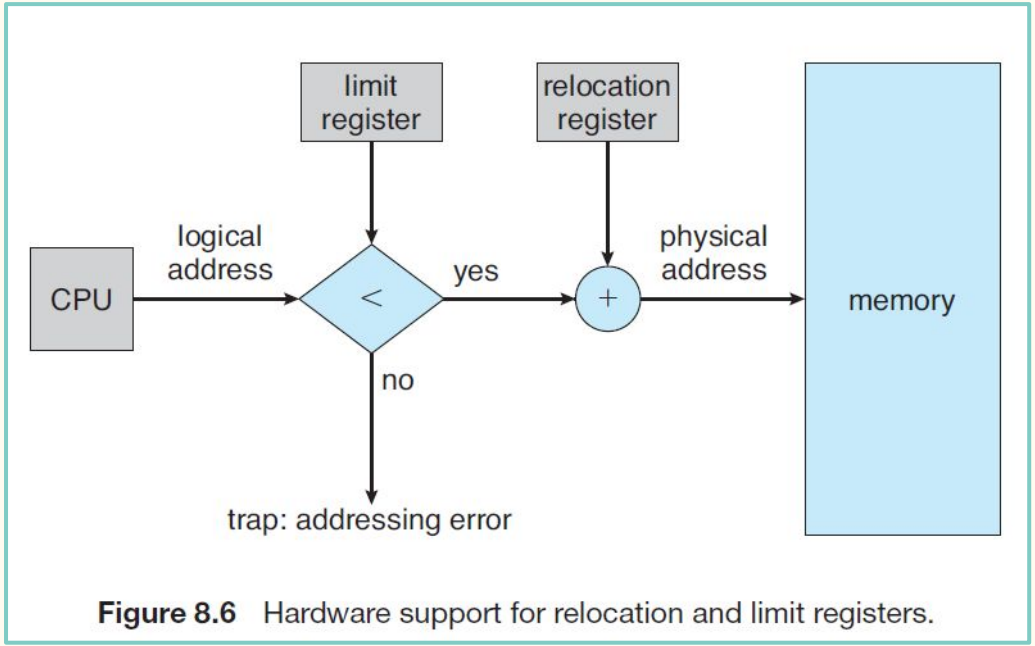
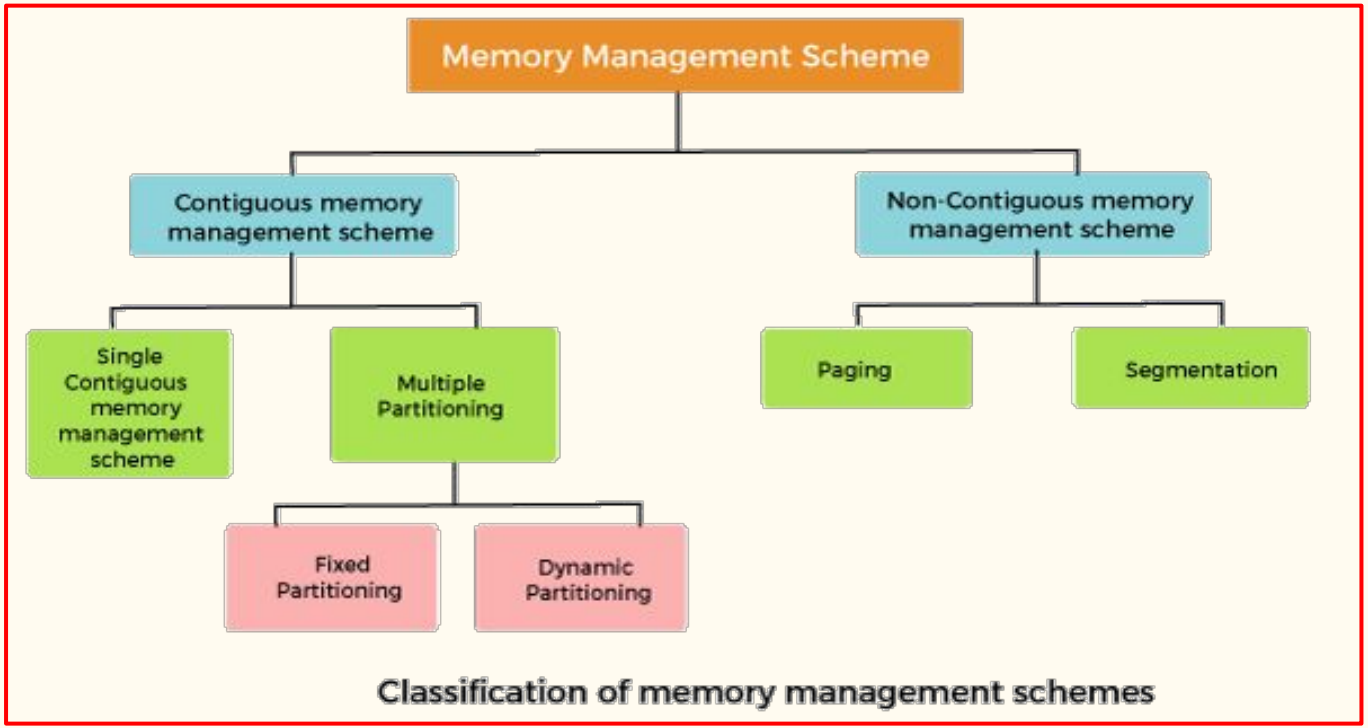


Figure 8.6 Hardware support for relocation and limit registers.

- In the above diagram, when the **scheduler** selects a process for the execution process, **the dispatcher**, on the other hand, is responsible for loading the relocation and limit registers with the correct values as part of the context switch as every address generated by the CPU is checked against these 2 registers, and we may protect the operating system, programs, and the data of the users from being altered by this running process.

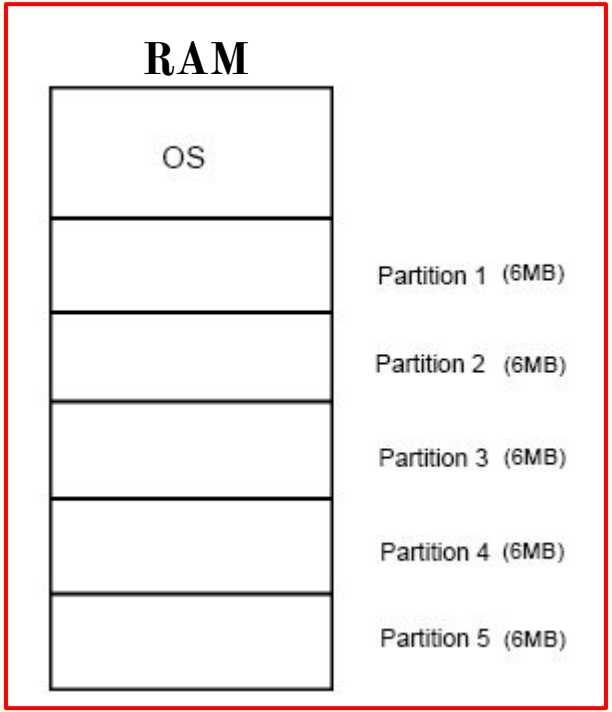
Memory Allocation



Classification of memory management schemes

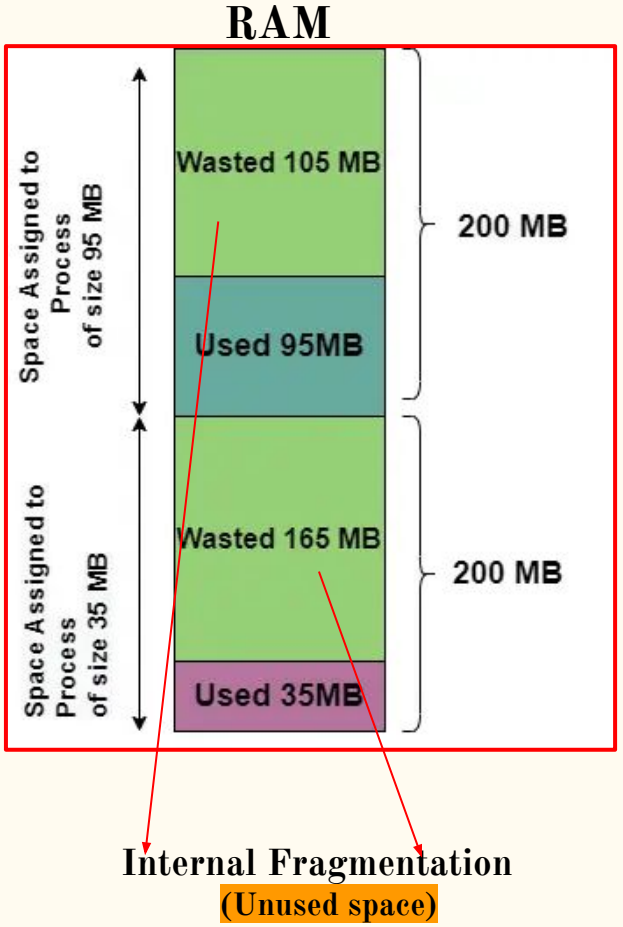
Fixed(Static)Partition:

- One of the simplest methods for allocating memory is to divide memory into **several fixed-sized partitions**. Each partition may contain exactly one process.
- Thus, the degree of 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 Fixed partition we found a problem called " **Internal Fragmentation".**



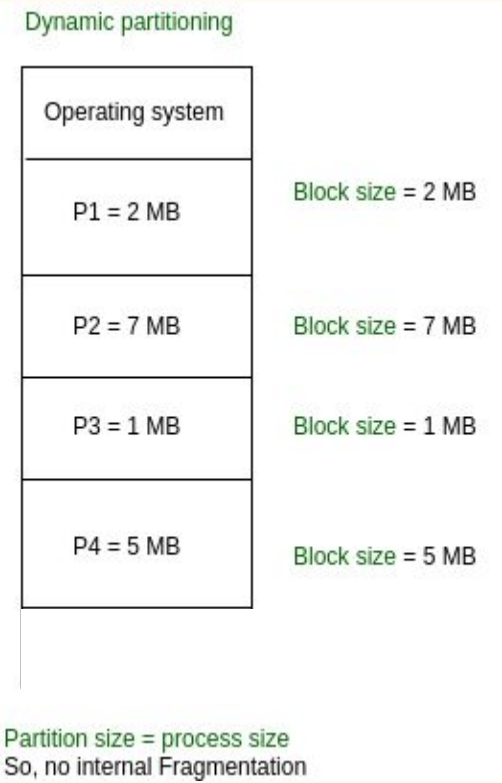
Internal Fragmentation

- Internal fragmentation happens when the memory is split into mounted-sized blocks.
- Whenever a process is requested for the memory, the mounted-sized block is allotted to the process.
- In the case where the memory allocated to the process is somewhat **larger than** the memory requested, a free space is created in the given memory block.
- Due to this, the free space of **the memory block is unused,** which causes **internal fragmentation**.
- The difference between allotted and requested memory is called internal fragmentation



Variable(Dynamic)Partition:

- It is a part of 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**.
- Various features associated with variable Partitioning-
  - Initially RAM is empty and **partitions are made during the run-time according to process's need** instead of partitioning during system configure.
  - **The size of partition will be equal to incoming process.**
  - The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
  - Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.

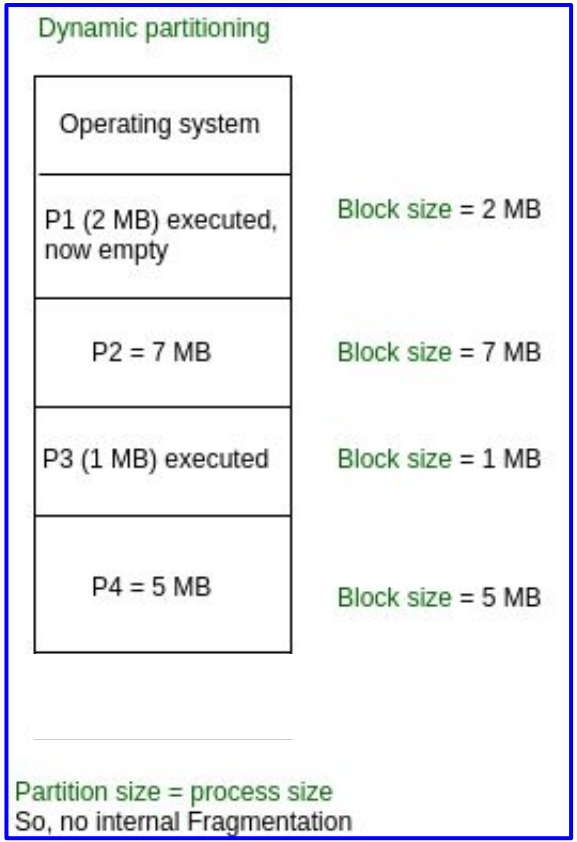


External Fragmentation

- External fragmentation happens when **there's a sufficient quantity of area within the memory** to satisfy the memory request of a method.
- However, the process's memory request **cannot be fulfilled** because the memory offered is in a non-contiguous manner.

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 empty space in memory cannot be allocated as the memory offered is in a non-contiguous manner.



P5 of size 3 MB cannot be accommodated

First Fit, Best Fit, and Worst Fit Memory Allocation strategies

First Fit:

- In the First Fit memory allocation strategy, the operating system searches for the first available memory block that is large enough to accommodate a process.
- It starts searching from the beginning of the memory and allocates the first block that meets the size requirement.
- It may result in significant fragmentation, both internal and external

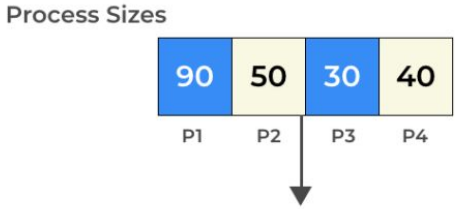
Best Fit:

- In the Best Fit memory allocation strategy, the operating system searches the entire memory space and allocates the smallest available block that is large enough to accommodate the process.
- It aims to minimize internal fragmentation.

Worst Fit:

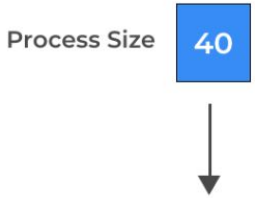
- In the Worst Fit memory allocation strategy, the operating system searches the entire memory space and allocates **the largest available block to the process**.
- It aims to maximize external fragmentation.
- It leads to more internal fragmentation, as smaller processes may be allocated in larger blocks, leaving unused memory within the allocated blocks.

First Fit Allocation in OS

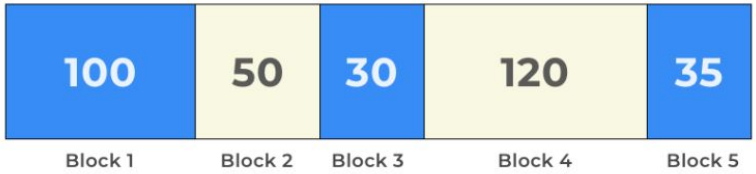


	Size	Allocated to	Memory Wastage After Process Occupies	
Process 1	90	Block2	100 - 90 = 10	
Process 2	50	Block 4	200 - 50 = 150	P4 remains Unallocated
Process 3	30	Block 3	40 - 30 - 10	
Process 4	40	Unallocated	-	

Best Fit Allocation in OS



Best FIT Allocation

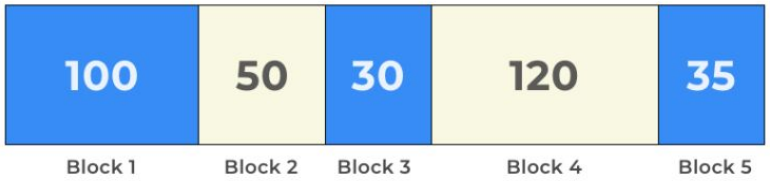


	Size	Can Occupy?	Memory Wastage After Process Occupies	
Block 1	100	Yes	100 - 40 = 60	
Block 2	50	Yes	50 - 40 = 10	Best
Block 3	30	No	-	
Block 4	120	Yes	120 - 40 = 80	
Block 5	35	No	-	

Worst Fit Allocation in OS

Process Size 40

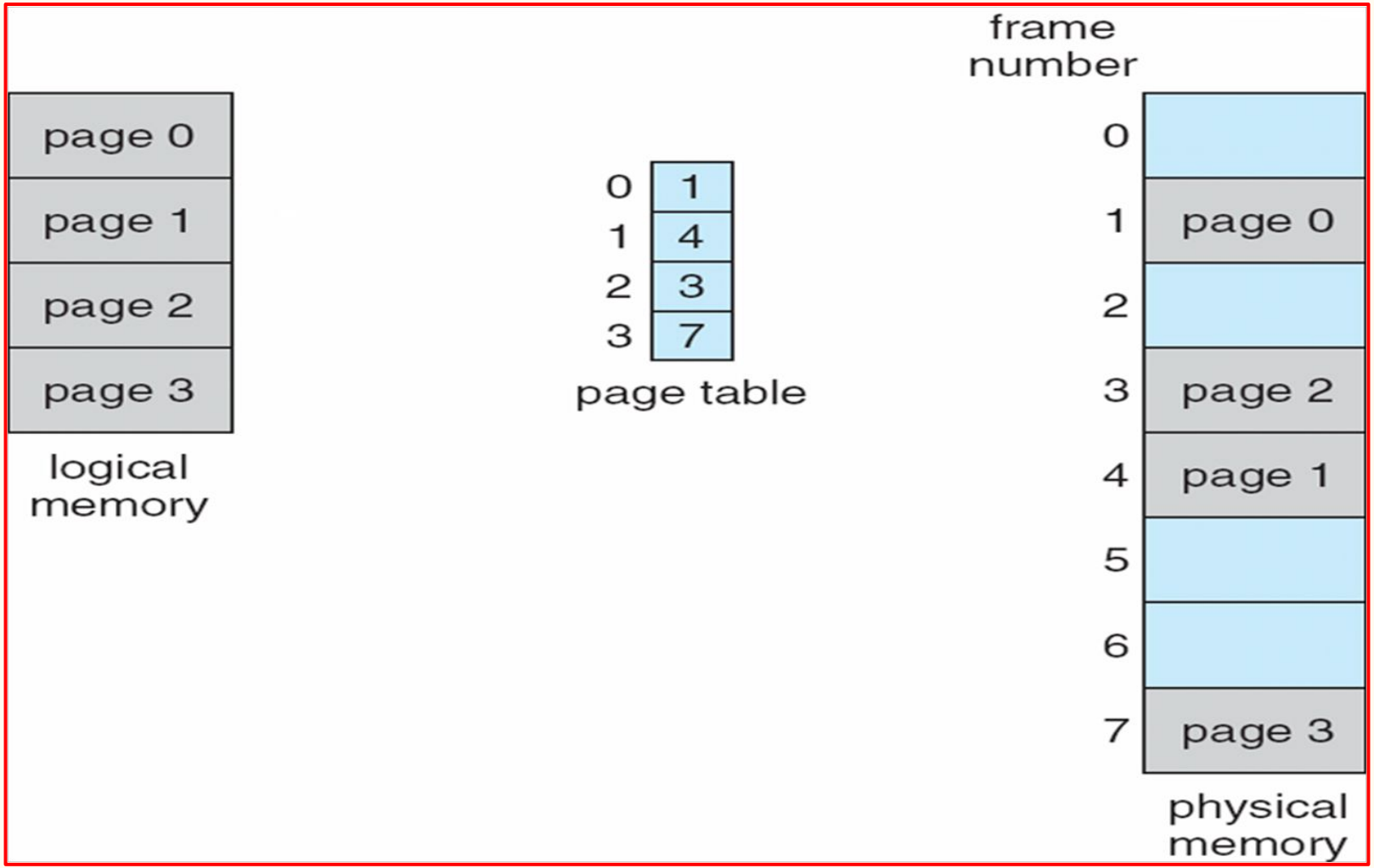
Worst FIT Allocation



	Size	Can Occupy?	Memory Wastage After Process Occupies	
Block 1	100	Yes	100 - 40 = 60	
Block 2	50	Yes	50 - 40 = 10	
Block 3	30	No	-	
Block 4	120	Yes	120 - 40 = 80	Worst
Block 5	35	No	-	

Paging

- Paging is **a memory-management scheme** that permits the physical address space of a process to be **noncontiguous**.
- Paging is a memory management scheme that **eliminates the need for contiguous allocation of physical memory.**
- The process of retrieving processes in the form of **pages** from the secondary storage into the main memory is known as paging
- The basic method for paging involves
    - Breaking **physical memory** into fixed-sized blocks called - - - > **Frames**
    - Breaking **logical memory** into blocks of the same size called - - - > **Pages**
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).


page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
frame number
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

- The hardware support for paging is illustrated in Figure(Previous Slide).
- **Every address generated by the CPU (Logical Address) is divided into two parts**
  - **(1) a page number (p)**
  - **(2) 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.

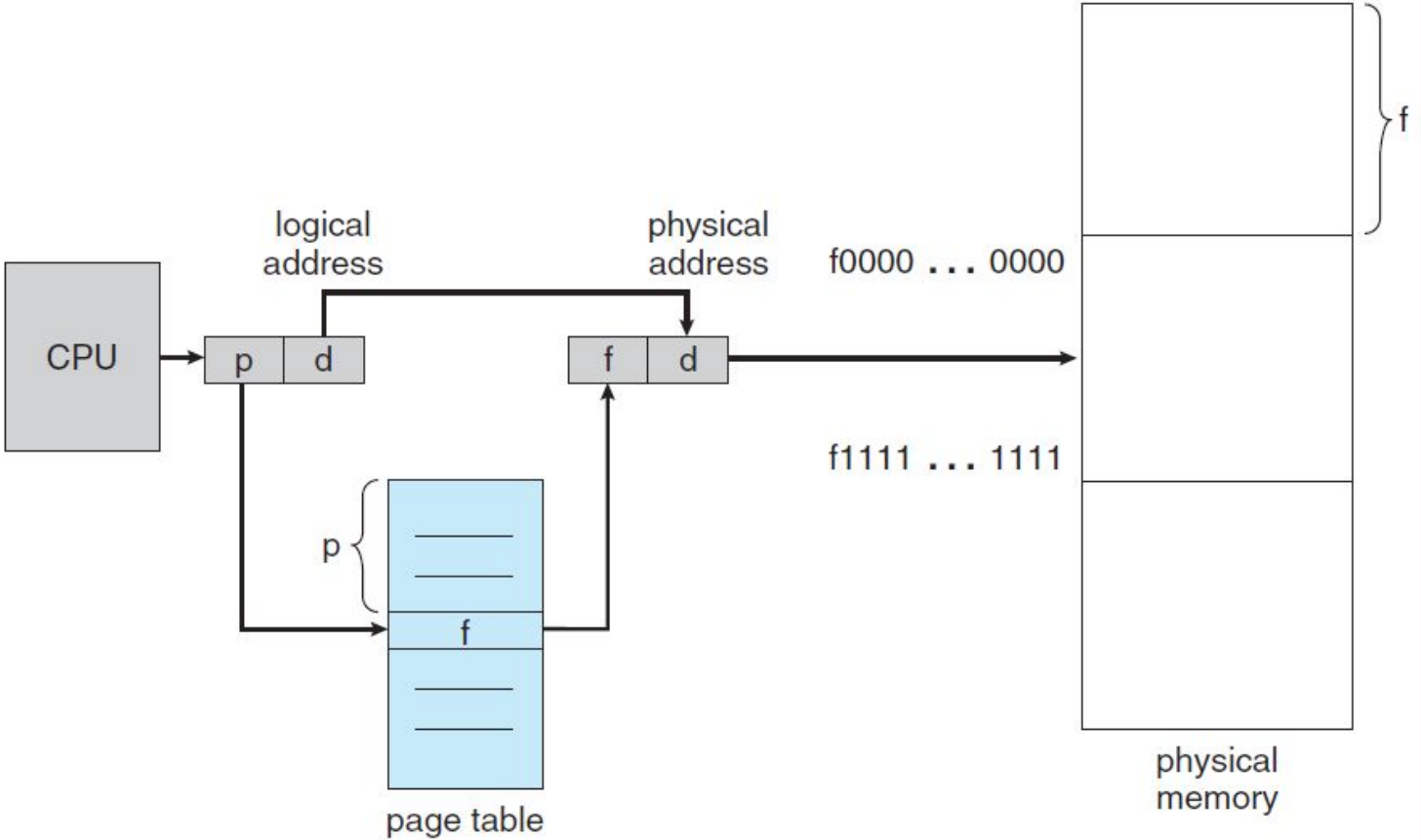
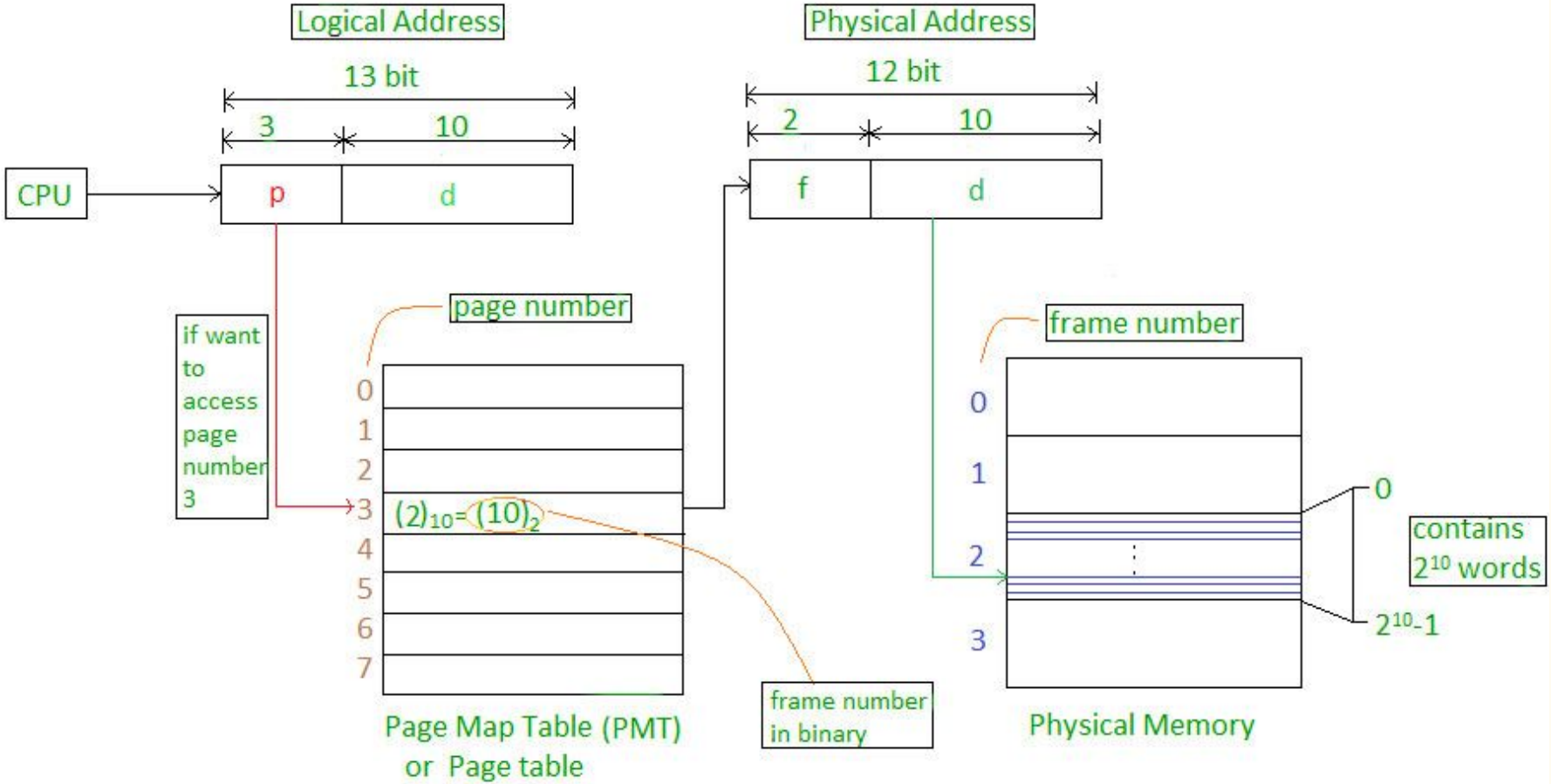
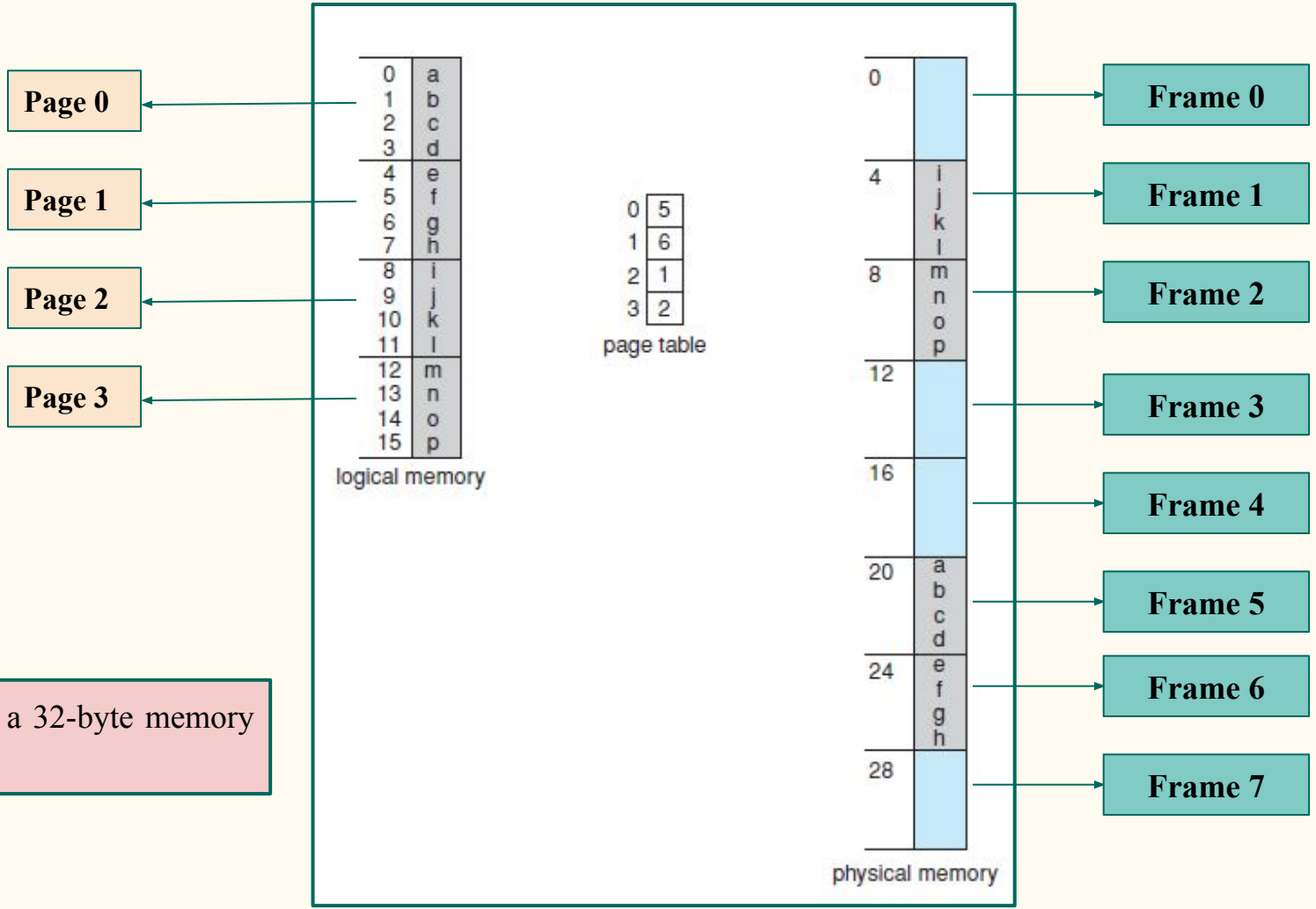


Figure Paging hardware.

Number of frames = Physical Address Space / Frame size = 4 K / 1 K = 4 = 2^2

Number of pages = Logical Address Space / Page size = 8 K / 1 K = 8 = 2^3





Paging example for a 32-byte memory with 4-byte pages

Finding Physical Address for the corresponding Logical address

Formula:

Physical Address = In which frame the page is available * Page size + Offset

- ➔ 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 × 4) + 0].
- ➔ Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 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 × 4) + 0].
- ➔ Logical address 13 maps to physical address 9.

Translation Look-a Side Buffer (TLB)
Or
Paging Hardware with TLB

- The implementation of page table or where we have to store page table either in
  - Registers or
  - Main Memory(RAM)
- The page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient.
- The use of registers for the page table is satisfactory **if the page table is reasonably small** (for example, 256 entries).
- Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).the use of fast registers to implement the page table is not feasible.

Main Memory

- The page table is kept in main memory.
- **A page-table base register (PTBR) points to the page table**.
- **Page-table length register (PRLR)** indicates **size of the page table**
- In this scheme every data/instruction access requires **two memory accesses.**
  - **One for the page table**
  - **One for the data/instruction**
- The two memory access problem can be solved by the use of **a special fast-lookup hardware cache** called **Associative Memory or Translation Look-a side Buffers (TLBs).**

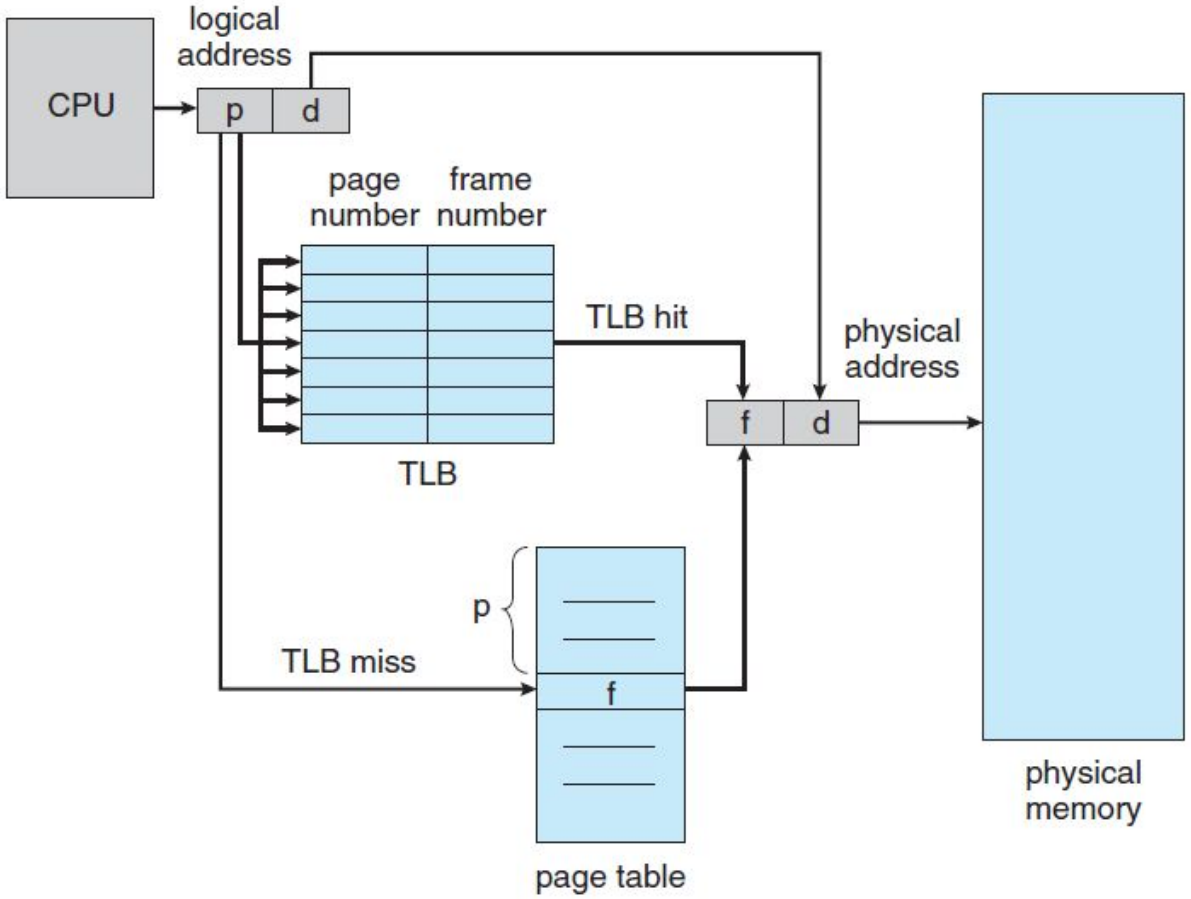


Figure 8.14 Paging hardware with TLB.

- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries(Contains Most frequently accessed pages).
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- **If the page number is found**(known as **a TLB Hit**), its frame number is immediately available and is used to access memory.
- **If the page number is not in the TLB** (known as a **TLB miss**), a memory reference to the page table must be made.
- When the frame number is obtained, we can use it to access memory (Figure8.14).
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, an existing entry must be selected for replacement.
- Replacement policies range from least recently used (LRU) through round-robin to random.

Memory Protection in a Paged Environment

- Memory protection in a paged environment is accomplished by **protection bits** associated with **each frame.**
- Normally, these bits are kept in the page table.
- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).
- One additional bit is generally attached to each entry in the page table

- One additional bit is generally attached to each entry in the page table: a
  - Valid bit
  - Invalid bit.
- When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page.
- When the bit is set to invalid, the page is not in the process's logical address space.
- Illegal addresses are trapped by use of the valid–invalid bit.
- The operating system sets this bit for each page to allow or disallow access to the page.
- Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468.
- Given a page size of 2 KB, we have the situation shown in Figure 8.15.
- Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table.
- Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

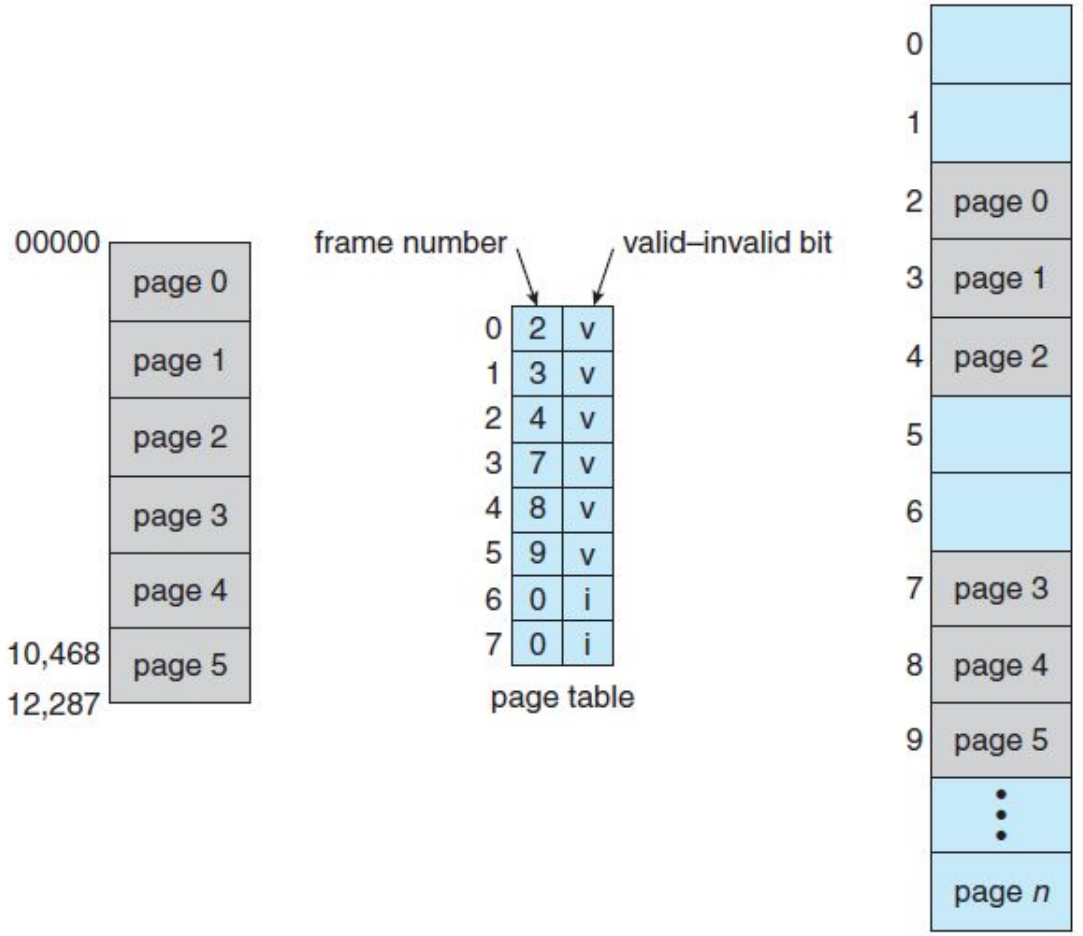


Figure 8.15 Valid (v) or invalid (i) bit in a page table.

Structure of the Page Table
Or
Types of Page Tables

Some of the common techniques that are used for structuring the Page table are as follows:

1. **Hierarchical Paging**
2. **Hashed Page Tables**
3. **Inverted Page Tables**

Hierarchical Paging

- **Hierarchical paging or Multilevel paging** is a type of paging where the logical address space is broken up into multiple page tables.
- The entries of the level 1 page table are pointers to a level 2 page table and entries of the level 2 page tables are pointers to a level 3 page table and so on.
- The entries of the **last level page table** store actual **frame information.**
- The advantage of using hierarchical paging is that it allows the operating system to efficiently manage **large logical address space**s.
- By dividing the page table into multiple levels, the operating system can minimize the amount of memory required to store the page table.

Why it is required?

- Most modern computer systems support a large logical address space (**2^32 to 2^64**). In such an environment, the page table itself becomes excessively large.
- For example, consider a system with a **32-bit logical address space**.
- If the **page size** in such a system is **4 KB (2^12)**, then a page table may consist of up to **1 million entries** (**2^32 / 2^12 = 2^20**).
- Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone.
- Clearly, we would not want to allocate the page table contiguously in main memory.
- One simple solution to this problem is **to divide the page table into smaller pieces**.We can accomplish this division in several ways.

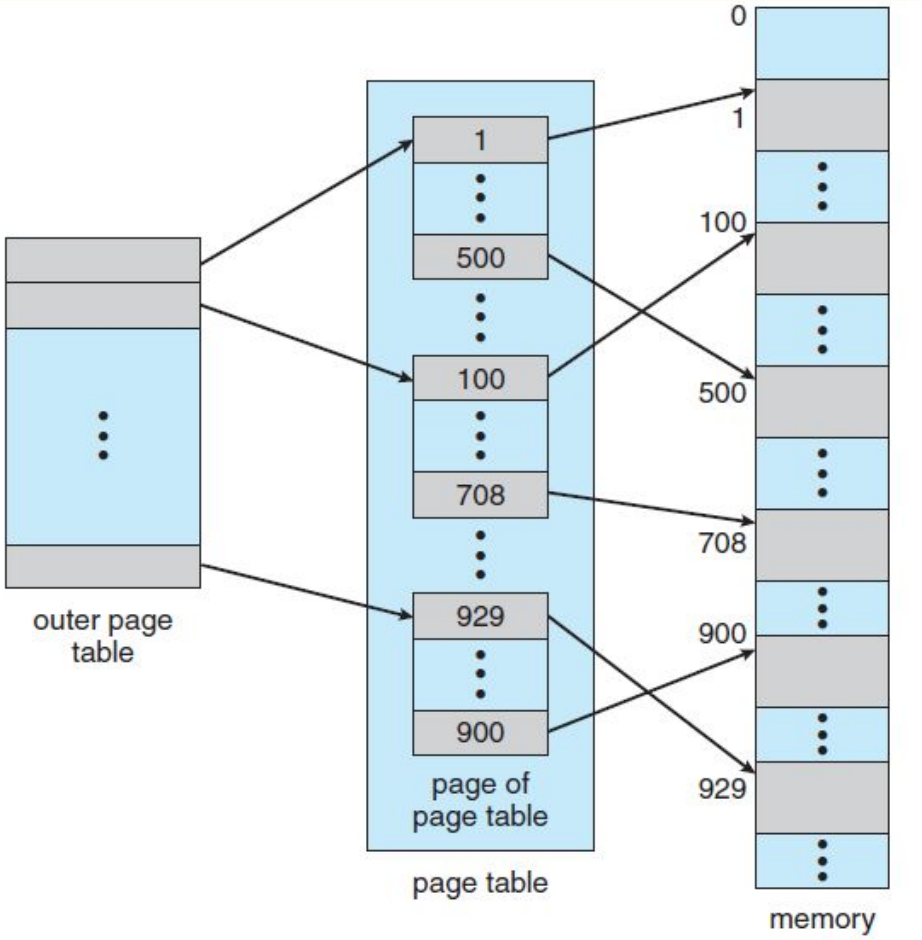
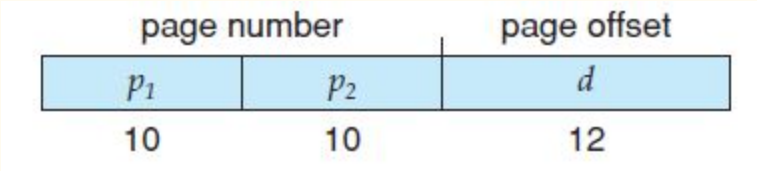


Figure 8.17 A two-level page-table scheme.

- One way is to use **a two-level paging algorithm**, in which **the page table itself is also paged** (Figure 8.17 Previous Slide).
- For example, consider again the system with a **32-bit logical address space (logical memory = 2^32) and a page size of 4 KB(2^12).**
- A logical address is divided into
  - a page number consisting of 20 bits (i.e 2^20 Pages)
  - a page offset consisting of 12 bits.
- Because **we page the page table**, the page number is further divided into **a 10-bit page number and a 10-bit page offset**. Thus, **a logical address** is as follows:



- where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table.
- The address-translation method for this architecture is shown in Figure

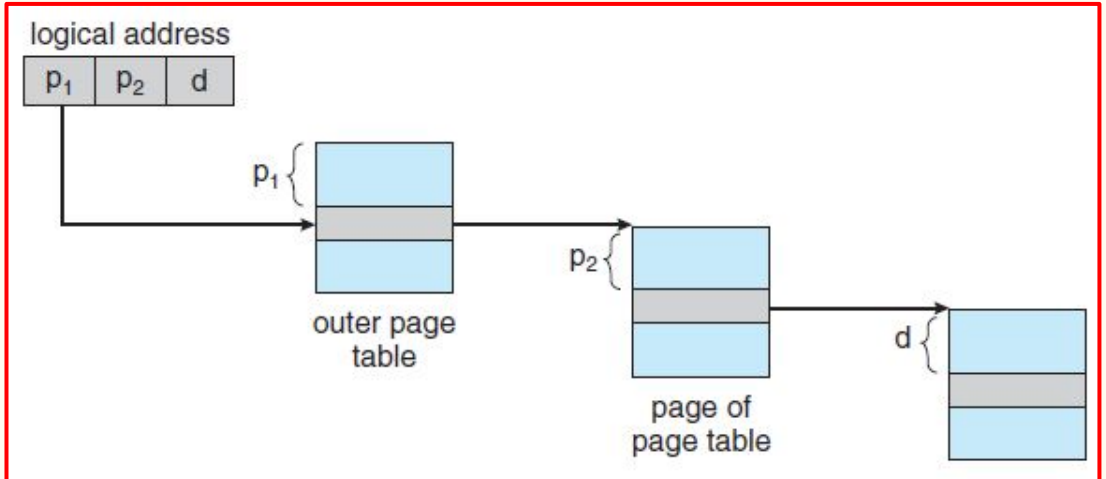


Figure 8.18 Address translation for a two-level 32-bit paging architecture.

- address translation works from **the outer page table inward**, this scheme is also known as **a forward-mapped page table.**

Hashed Page Tables

★ A common approach for handling address spaces larger than 32 bits is to use **a hashed page table**, with the **hash value** being the **virtual page number**.

★ **Each entry in the hash table contains a linked list of elements** that hash to the same location (to handle collisions).

★ Each element consists of **three fields:**

- (1) The virtual **page number**
- (2) The value of the **mapped page frame**
- (3) A pointer to the **next element in the linked list.**

★ The algorithm works as follows:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- **If there is a match**, the corresponding page frame (field 2) is used to form the desired physical address.
- **If there is no match**, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 8.19.

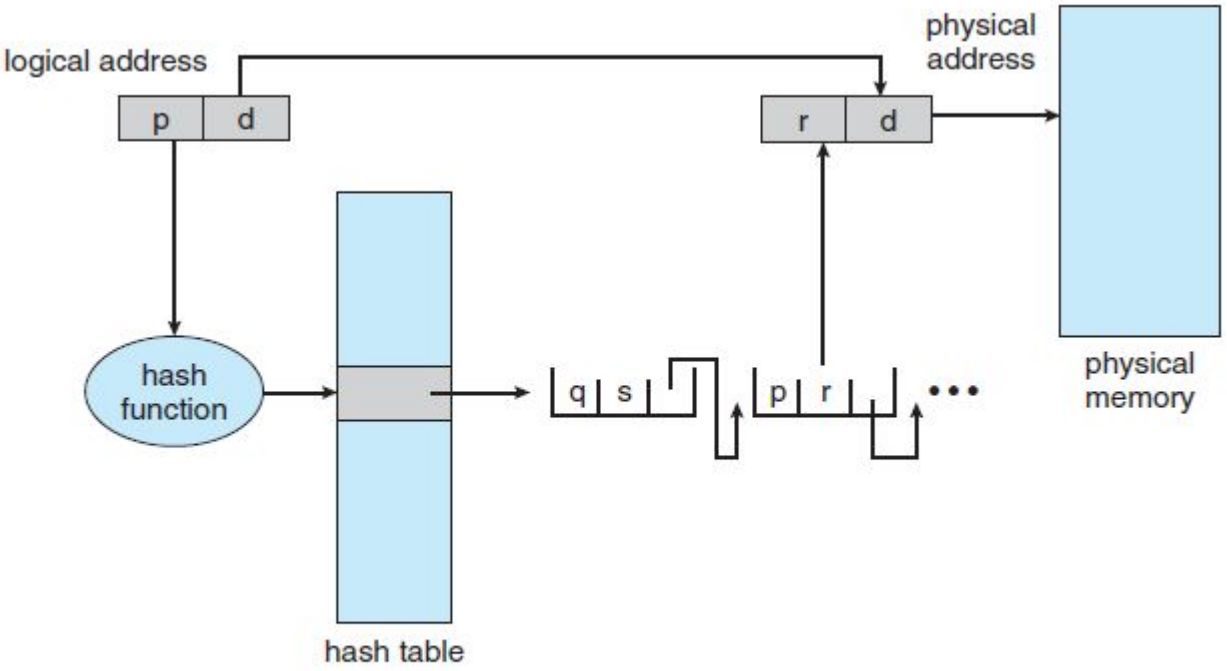


Figure 8.19 Hashed page table.

Inverted Page Tables

- ★ An inverted page table is a data structure used in operating systems for efficient memory management in virtual memory systems.
- ★ Unlike a traditional page table, which maps virtual addresses to physical addresses on a per-process basis, an inverted page table maps physical addresses to virtual addresses **across all processes.**
- ★ In a typical virtual memory system, each process has its own page table that maps virtual addresses to physical addresses.
- ★ This approach requires a separate page table for each process, which can consume a significant amount of memory for systems with a large number of processes.
- ★ In contrast, an inverted page table maintains a single table that contains entries for each physical frame in the system. Each entry in the inverted page table stores information about the process and virtual page that is currently mapped to that physical frame.

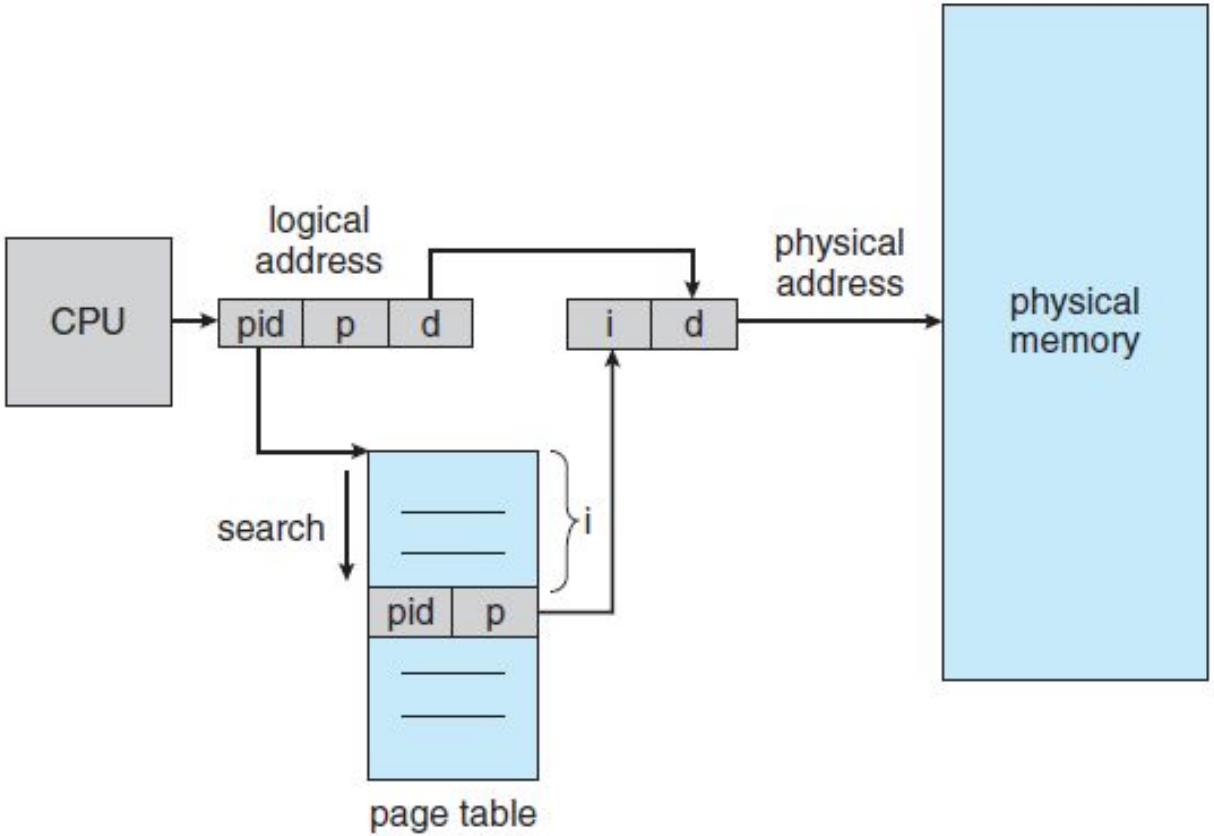
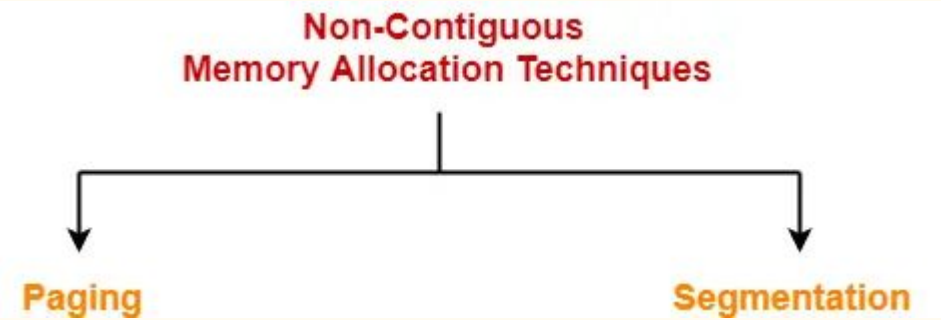


Figure 8.20 Inverted page table.

Segmentation



★ Like Paging, Segmentation is another non-contiguous memory allocation technique.

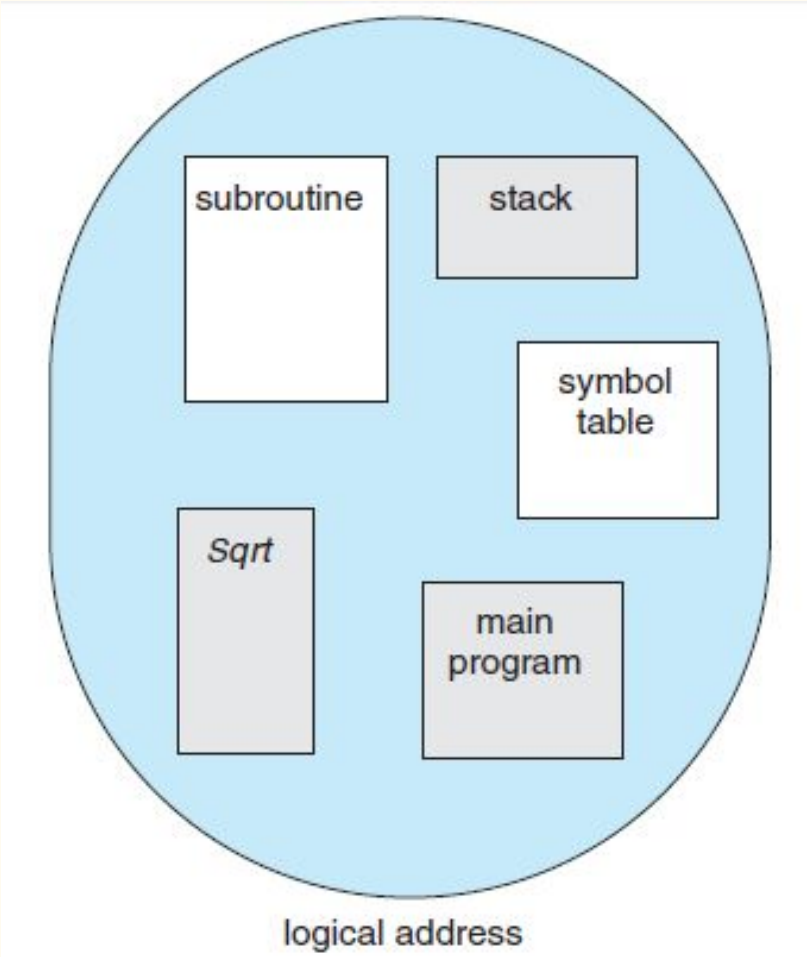
★ In segmentation, **process is not divided blindly into fixed size pages**,**Segmentation is a variable size partitioning scheme.**

★ In segmentation, secondary memory and main memory are divided into partitions of **unequal size.**

★ The size of partitions depend on the length of modules.

★ The partitions of secondary memory are called as segments.

★ The process is divided into modules for better visualization.



Programmer's view of a program

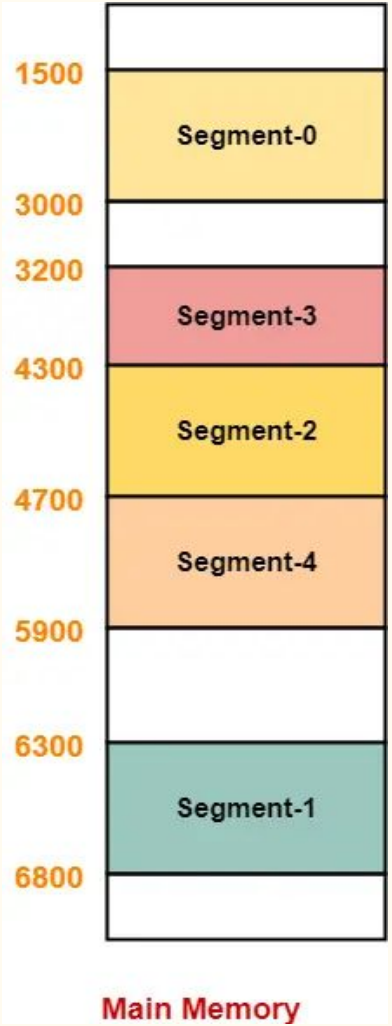
Segment table:

- ★ Segment table is a table that **stores the information about each segment of the process.**
- ★ It has two columns.
  - **First column** stores the size or length of the segment.
  - **Second column** stores the base address or starting address of the segment in the main memory.
- ★ Segment table is stored as a separate segment in the main memory.
- ★ Segment table base register (STBR) stores the base address of the segment table.

	Limit	Base
Seg-0	1500	1500
Seg-1	500	6300
Seg-2	400	4300
Seg-3	1100	3200
Seg-4	1200	4700

Segment Table

In accordance to the above segment table, the segments are stored in the main memory as



Translating Logical Address into Physical Address

Step 1:

- CPU always generates a logical address, It consisting of two parts-
  - Segment Number
  - Segment Offset
- Segment Number specifies the specific segment of the process from which CPU wants to read the data.
- Segment Offset specifies the specific word in the segment that CPU wants to read.

Step 2:

- For the generated segment number, corresponding entry is located in the segment table.
- Then, segment offset is compared with the limit (size) of the segment.
- If **segment offset** is found to be greater than or equal to the limit , a trap is generated.
- If **segment offset** is found to be smaller than the limit, then request is treated as a valid request.

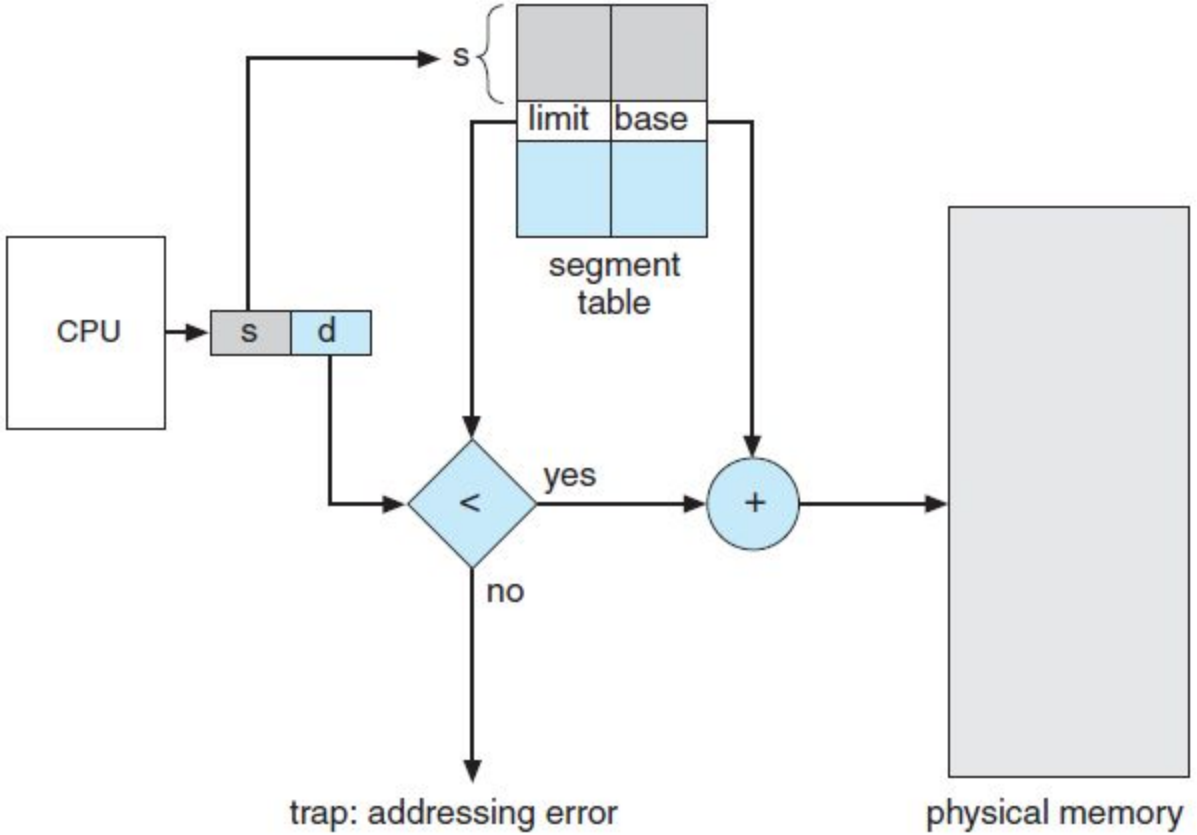


Figure 8.8 Segmentation hardware.

- 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,
  - **The segment limit** specifies the length of the segment.

Eg:

➔ We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown(Next Slide).

➔ 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 1,000 bytes long.

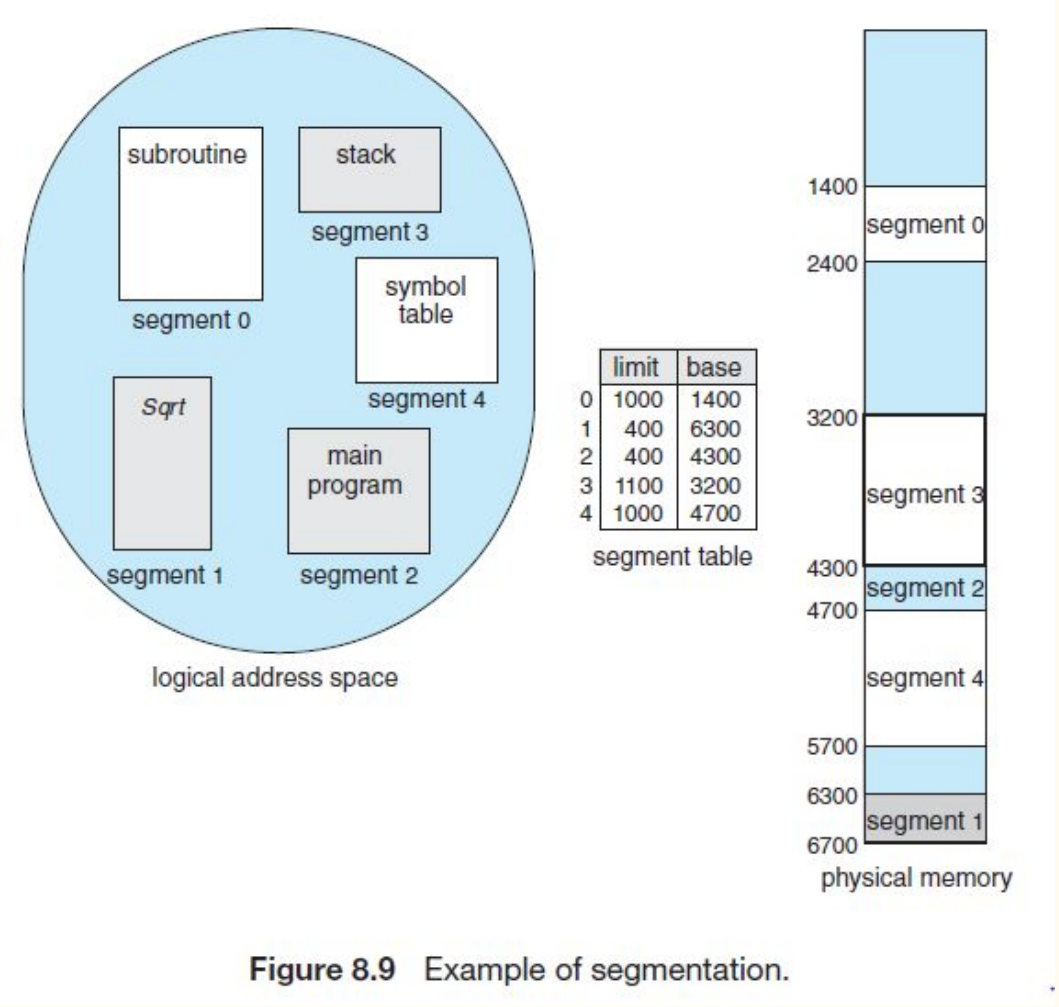


Figure 8.9 Example of segmentation.