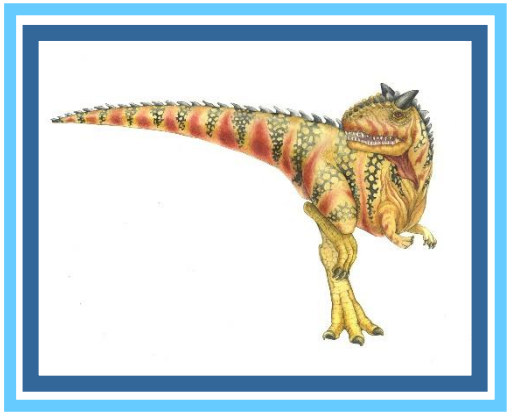
Module 4.1
Memory Management



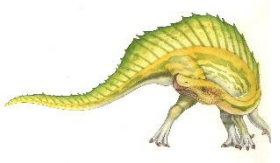
Memory Management

- Background
- Memory Hierarchy
- Memory Allocation Techniques
- Fragmentation
- Segmentation
- Paging
- TLB

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Introduction



Background

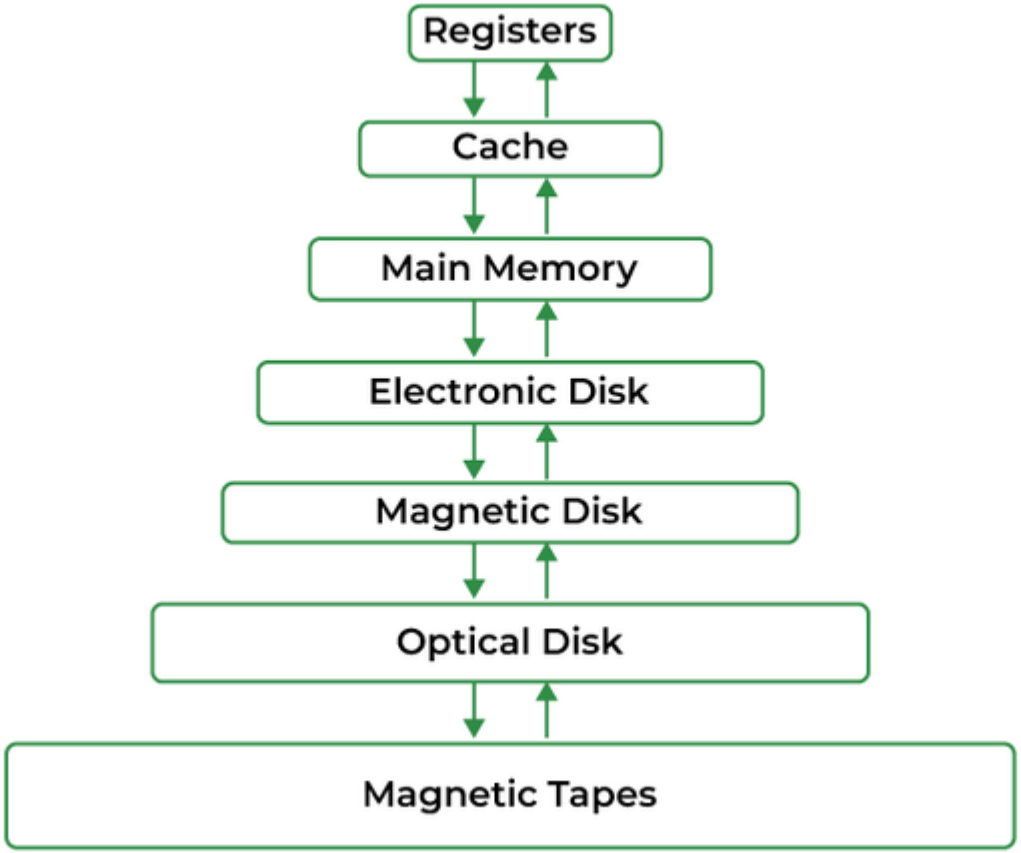
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Background

- **Why Memory Management is required:**
- Allocate and de-allocate memory before and after process execution.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

- **Goals?**

Memory hierarchy



Memory Management

Ideally programmers want memory that is

- large
- fast
- non volatile
- Memory hierarchy
- small amount of fast, expensive memory – cache
- some medium-speed, medium price main memory
- gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

3 criteria's:

- Size
- Access Time
- Per unit cost

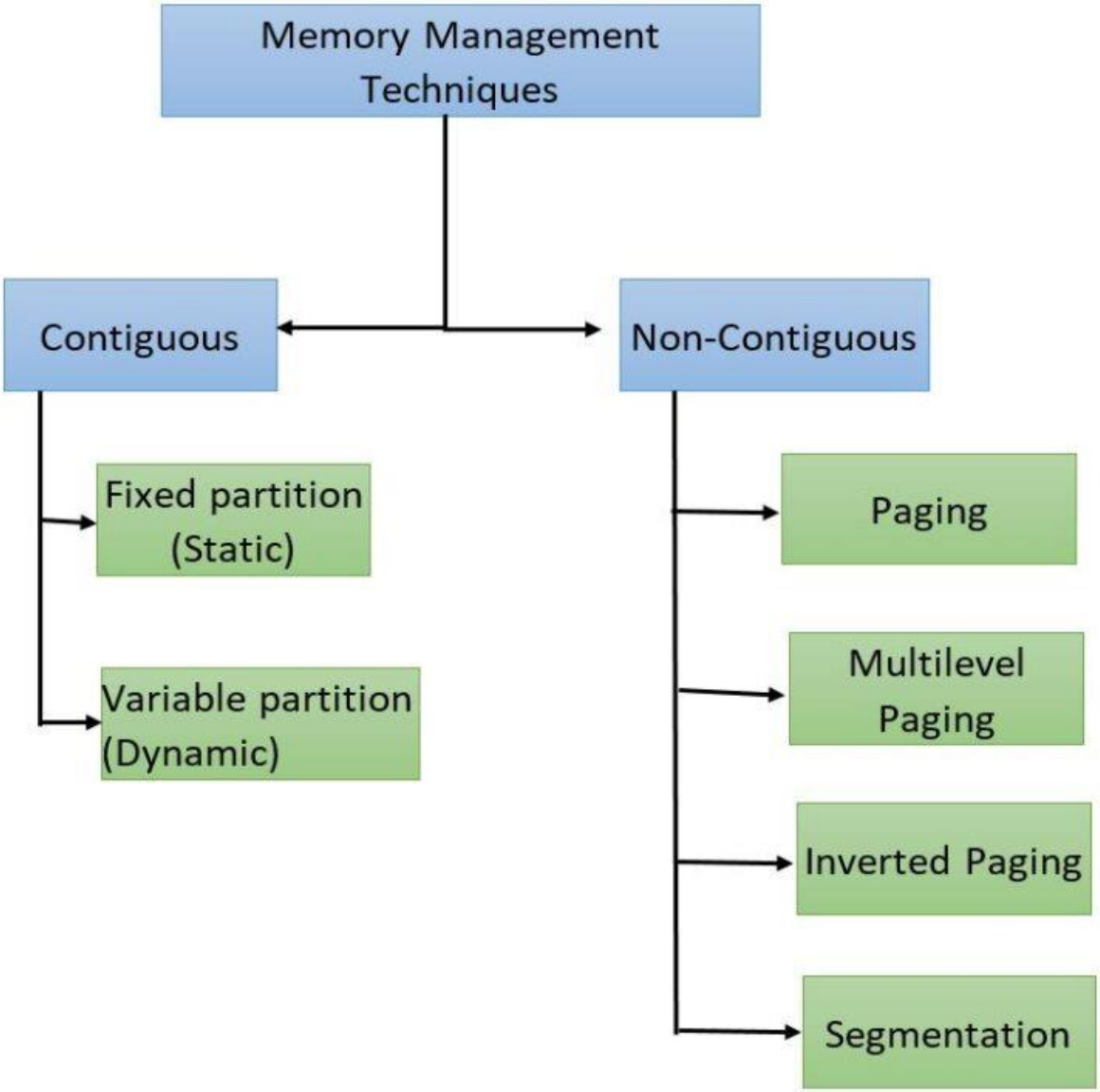
Memory Management Techniques

Contiguous Memory :

- It allows to store the process only in a contiguous fashion.
- Thus, entire process has to be stored as a single entity at one place inside the memory.

Non- Contiguous Memory :

- the program is divided into different blocks and loaded at different portions of the memory that need not necessarily be adjacent to one another. This scheme can be classified depending upon the size of blocks and whether the blocks reside in the main memory or not.

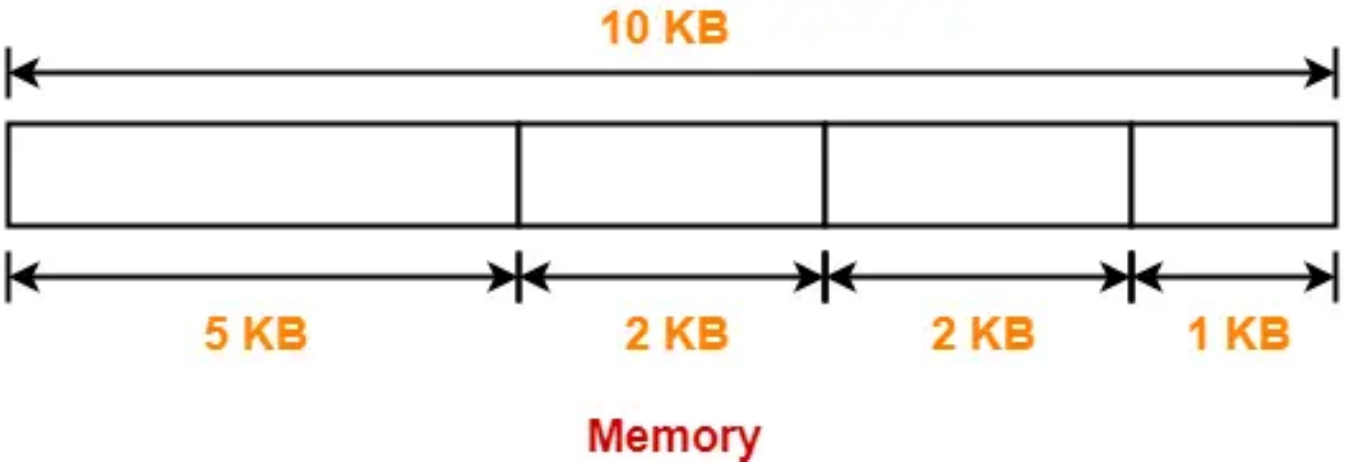

Memory Management Techniques
Contiguous
Non-Contiguous
Fixed partition (Static)
Variable partition (Dynamic)
Paging
Multilevel Paging
Inverted Paging
Segmentation

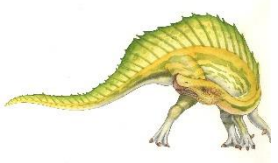
Memory Management Techniques

Contiguous Memory Management can be categorized in two:

1)Fixed Partition Scheme:

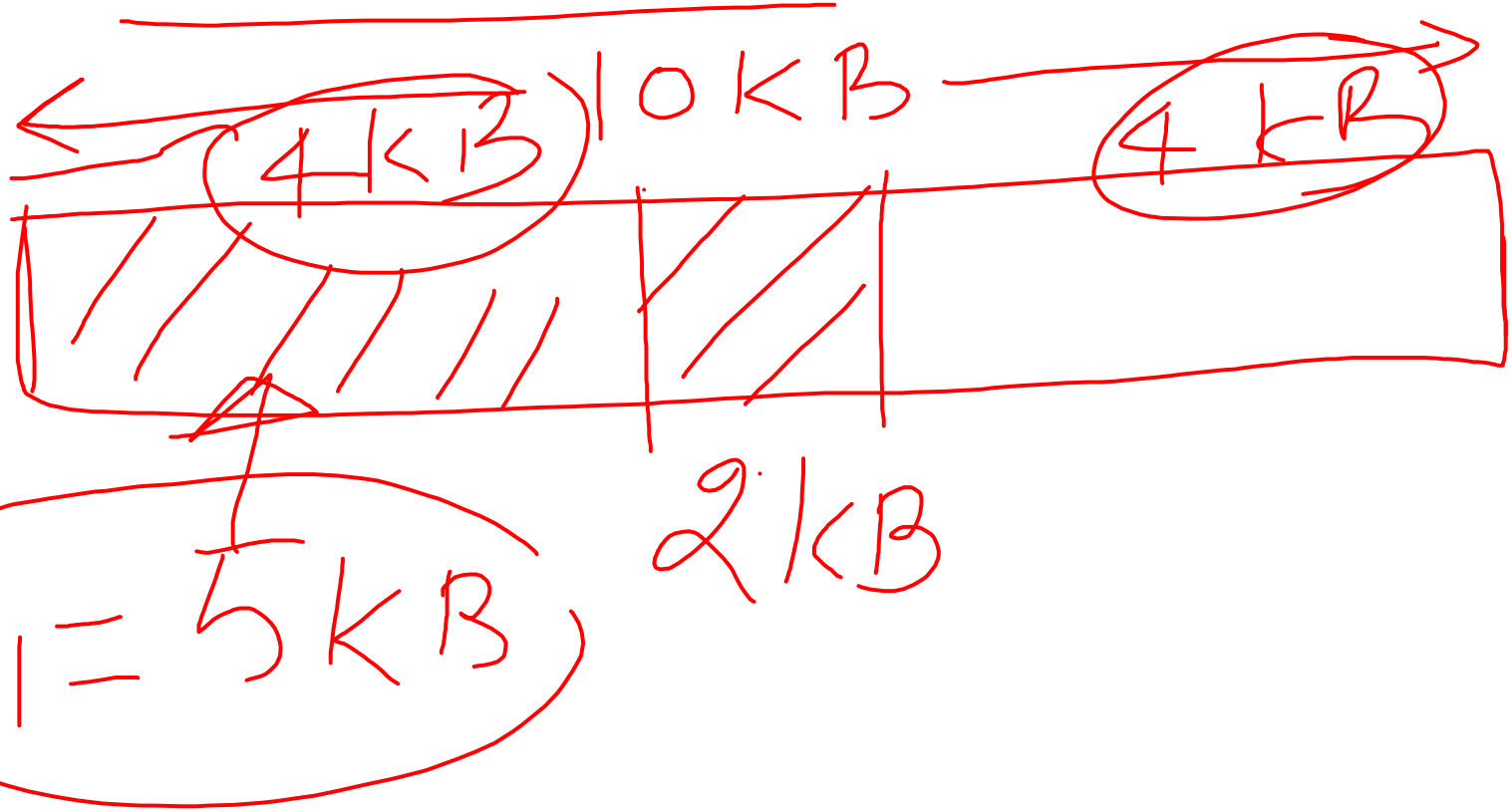
- Memory is divided into fixed number of partitions. Fixed means number of partitions are fixed in the memory.
- In the fixed partition, in every partition only one process will be accommodated. Maximum size of the process is restricted by maximum size of the partition. Every partition is associated with the limit registers.





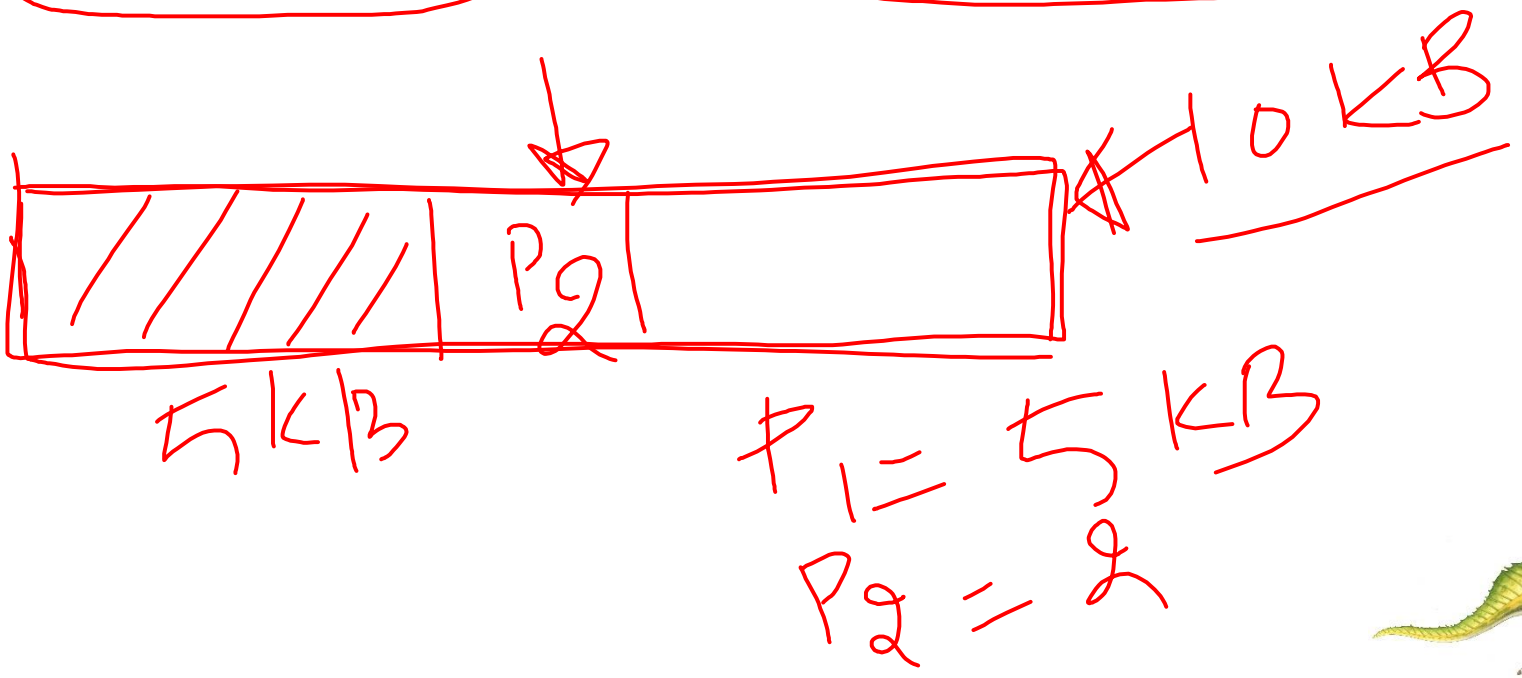
TOTAL = 8 KB

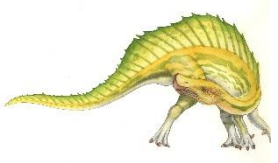
EXTERNAL



Memory Management Techniques

- **Variable Partition Scheme**:
- In the variable partition scheme, initially memory will be single continuous free block. Whenever the request by the process arrives, accordingly partition will be made in the memory. If the smaller processes keep on coming then the larger partitions will be made into smaller partitions.
- External Fragmentation is found in variable partition scheme.





Algorithms for Partition Allocation-

Problem-01:

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

- Perform the allocation of processes using-

- First Fit Algorithm
- Best Fit Algorithm
- Worst Fit Algorithm
-

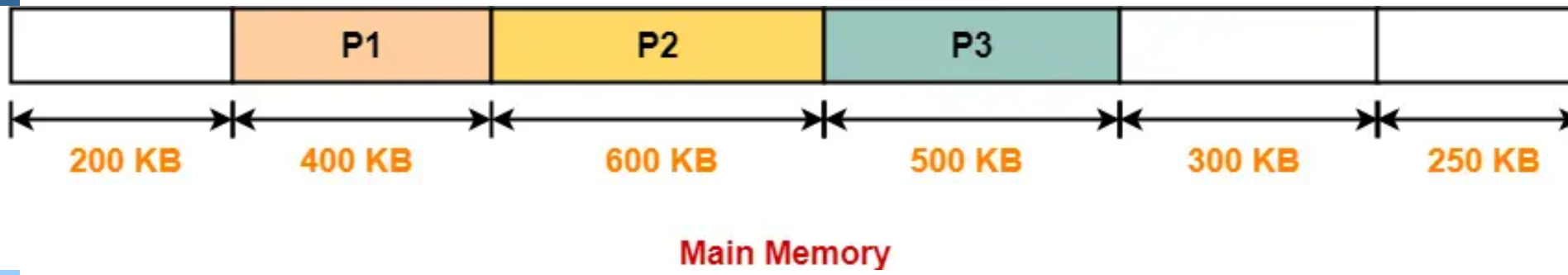
Solution

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

Let us say the given processes are-

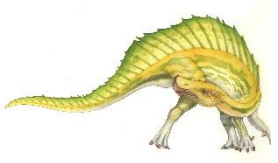
Process P1 = 357 KB
Process P2 = 210 KB
Process P3 = 468 KB
Process P4 = 491 KB

First Fit



In First Fit Algorithm,

Algorithm starts scanning the partitions serially. When a partition big enough to store the process is found, it allocates that partition to the process.



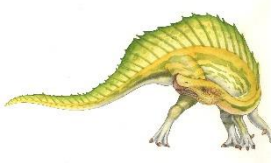
Best Fit

- In Best Fit Algorithm,

- Algorithm first scans all the partitions.
- It then allocates the partition of smallest size that can store the process.

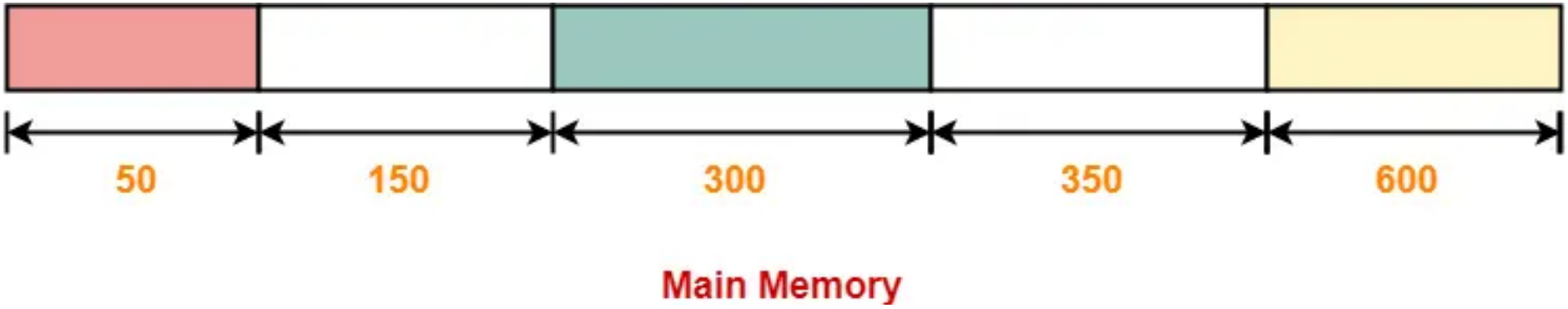
Worst fit

- In Worst Fit Algorithm,
- Algorithm first scans all the partitions.
- It then allocates the partition of largest size to the process.



Problem 2 – variable size partitioning

- Consider the following heap (figure) in which blank regions are not in use and hatched regions are in use-

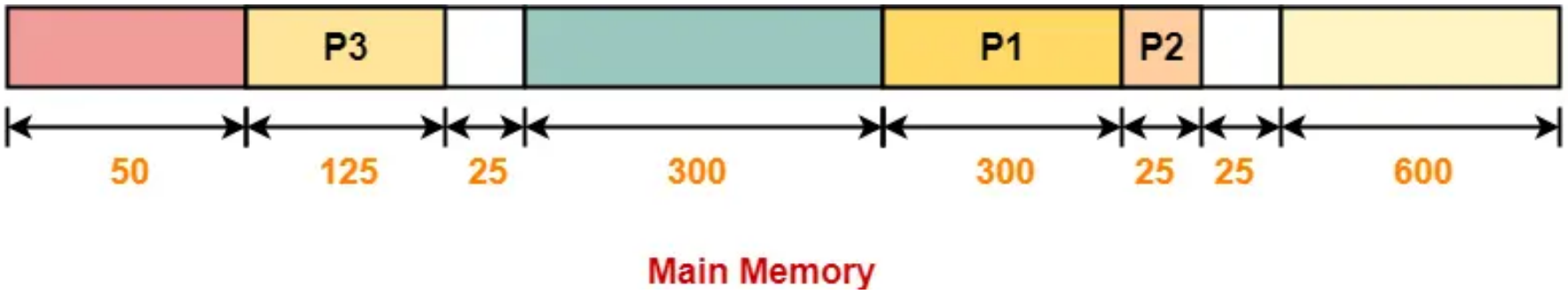


Main Memory

- The sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use-
- Either first fit or best fit policy (any one)
- First fit but not best fit policy
- Best fit but not first fit policy
- None of the above

First fit

Best fit



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.

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

Logical address space is the set of all logical addresses generated by a program

Physical address space is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

Hardware device that at run time maps virtual to physical address

Many methods possible, covered in the rest of this chapter

To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

- Base register now called **relocation register**
- MS-DOS on Intel 80x86 used 4 relocation registers

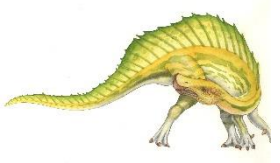
The user program deals with *logical* addresses; it never sees the *real* physical addresses

- Execution-time binding occurs when reference is made to location in memory
- Logical address bound to physical addresses

Address Translation

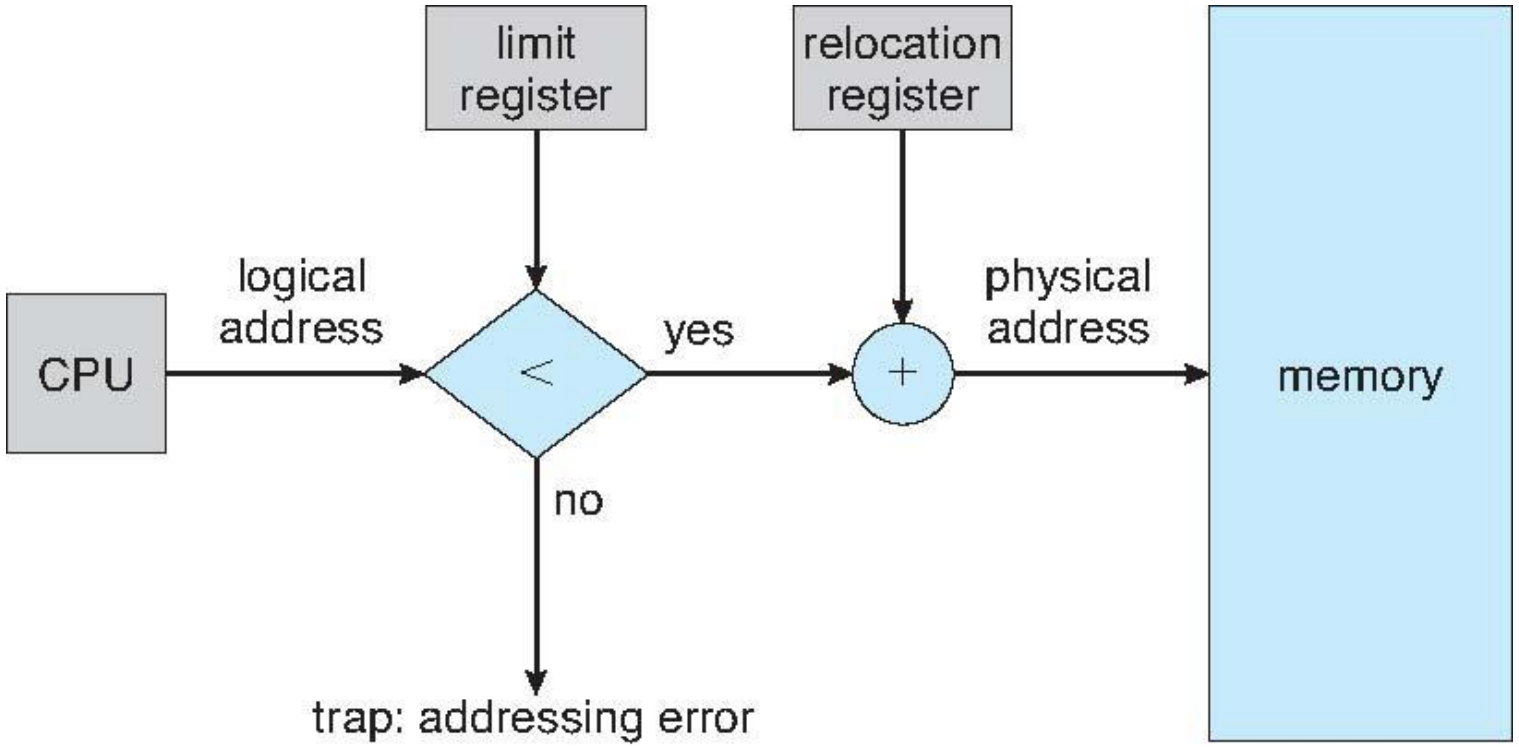
- **Translating Logical Address into Physical Address-**
- CPU always generates a logical address.
- A physical address is needed to access the main memory.

- Relocation Register stores the base address or starting address of the process in the main memory.
- Limit Register stores the size or length of the process.

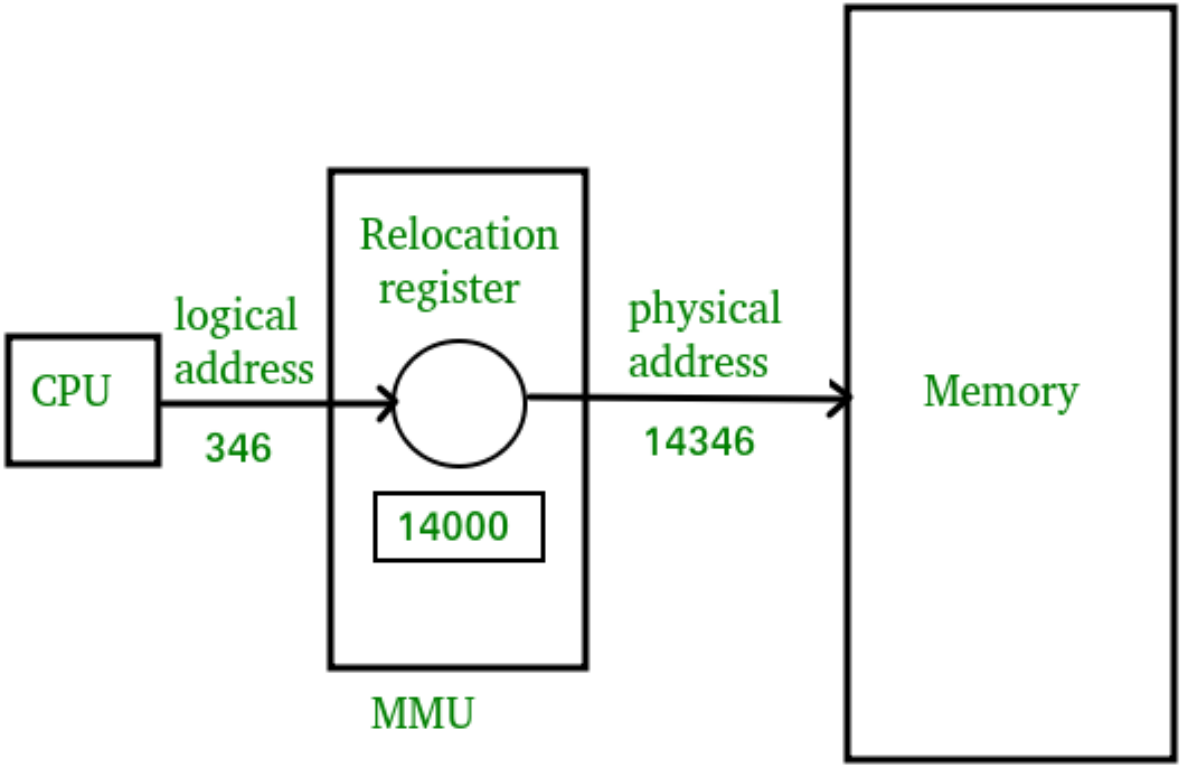


Hardware Support for Relocation and Limit Registers

Contiguous allocation



Address Translation

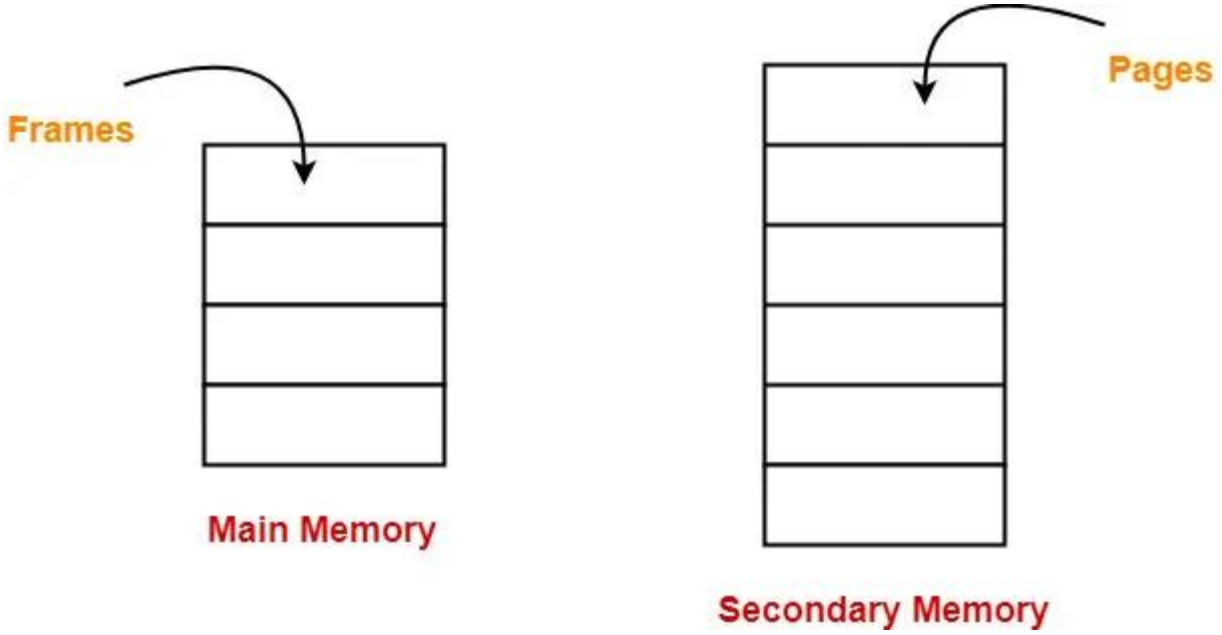


Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have Internal fragmentation

Paging

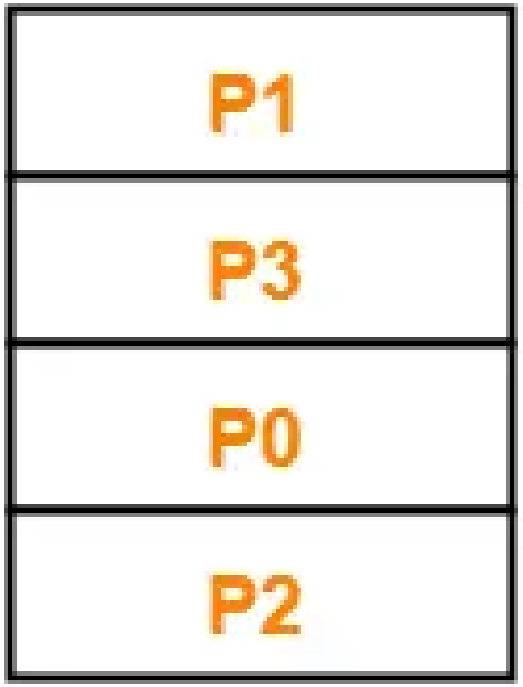
- Paging is a fixed size partitioning scheme.
- In paging, secondary memory and main memory are divided into equal fixed size partitions.
- The partitions of secondary memory are called as **pages**.
- The partitions of main memory are called as **frames**.



Paging

Consider a process is divided into 4 pages P0, P1, P2 and P3.

Depending upon the availability, these pages may be stored in the main memory frames in a non-contiguous fashion as shown-



Translating Logical Address into Physical Address-

- CPU always generates a logical address.
- A physical address is needed to access the main memory.
- Following steps are followed to translate logical address into physical address-

Step-01:

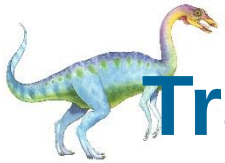
CPU generates a logical address consisting of two parts-

- Page Number
- Page Offset

- **Step-02:**

For the page number generated by the CPU,

Page Table provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.



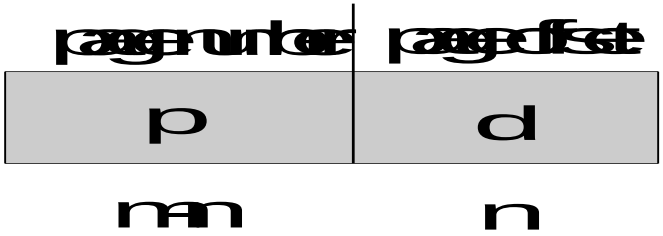
Translating Logical Address into Physical Address-

Step-03:

- The frame number combined with the page offset forms the required physical address.
- Frame number specifies the specific frame where the required page is stored.
- Page Offset specifies the specific word that has to be read from that page.
-

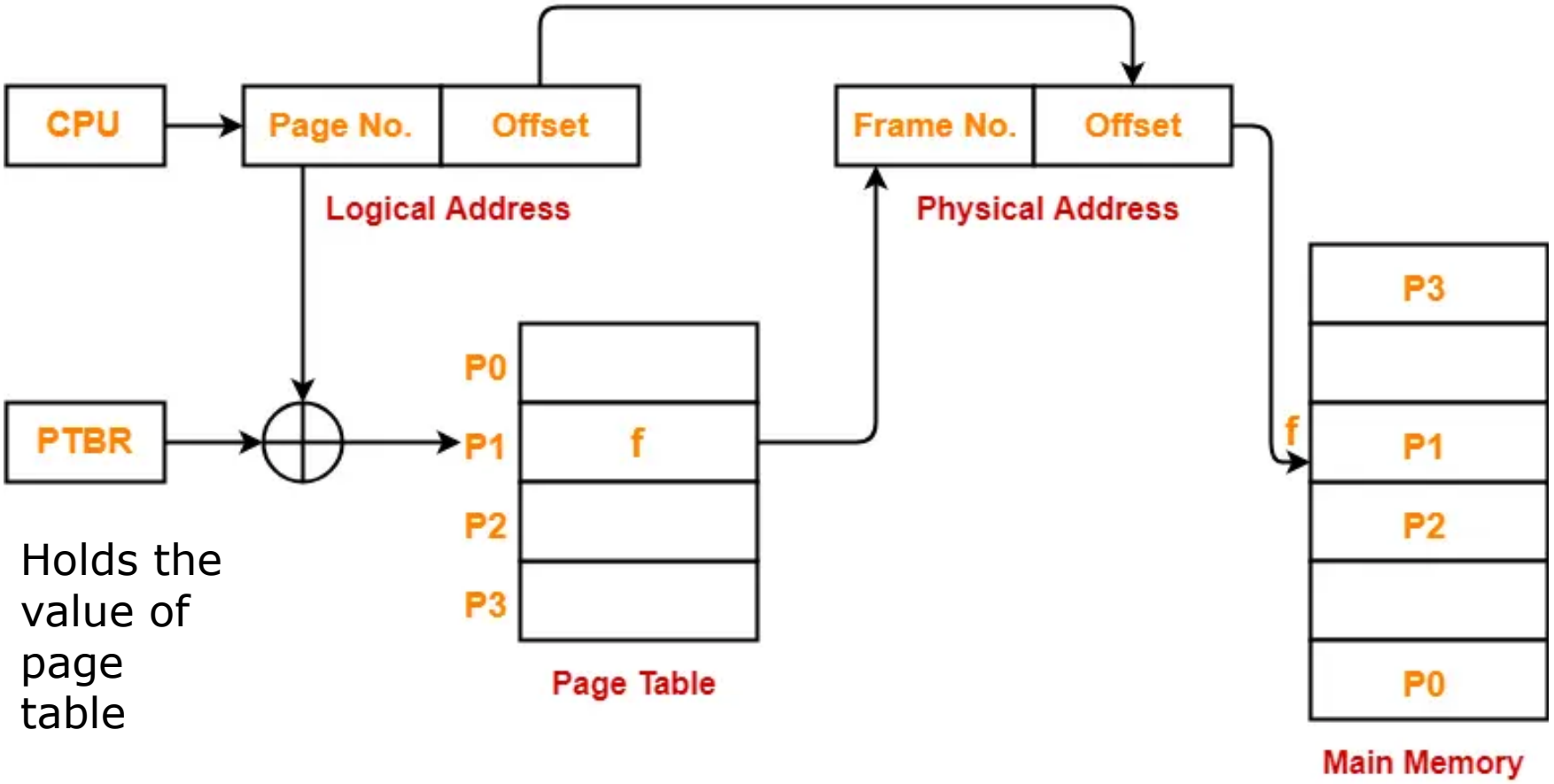
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



  - For given logical address space 2^m and page size 2^n

Paging Hardware



Holds the value of page table

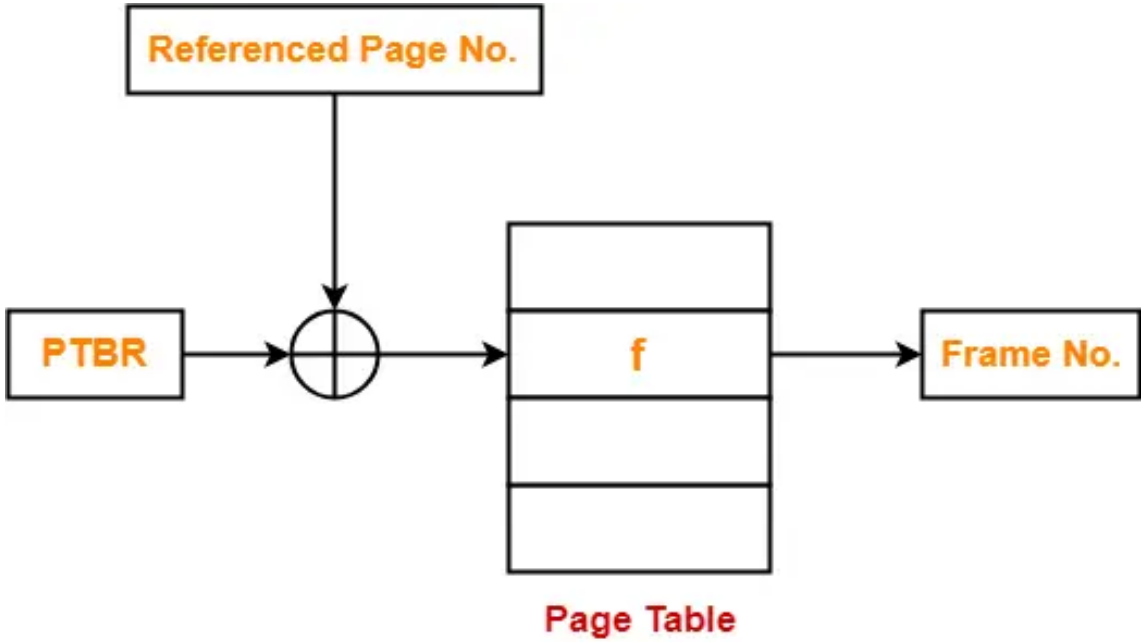
Translating Logical Address into Physical Address

Paging

- **Page Table-**
- Page table is a data structure.
- It maps the page number referenced by the CPU to the frame number where that page is stored.
-

- **Characteristics-**
- Page table is stored in the main memory.
- Number of entries in a page table = Number of pages in which the process is divided.
- Page Table Base Register (PTBR) contains the base address of page table.
- Each process has its own independent page table

Page Table Working



Obtaining Frame Number Using Page Table

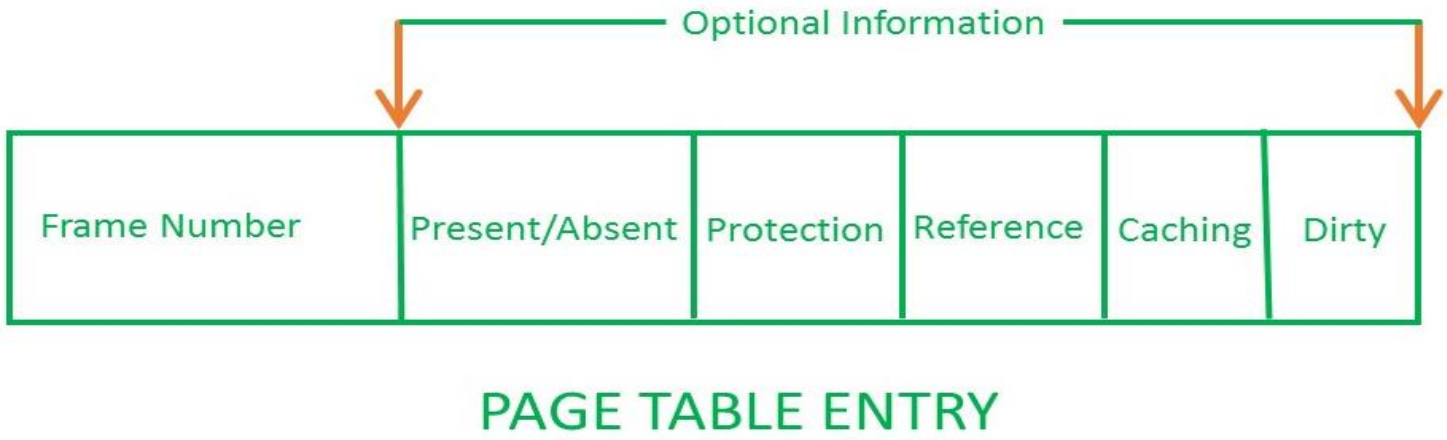
- The base address of the page table is added with the page number referenced by the CPU.
- It gives the entry of the page table containing the frame number where the referenced page is stored.

Important points

- **Features of paging:**
- Mapping logical address to physical address.
- Page size is equal to frame size.
- Number of entries in a page table is equal to number of pages in logical address space.
- The page table entry contains the frame number.
- All the page table of the processes are placed in main memory.

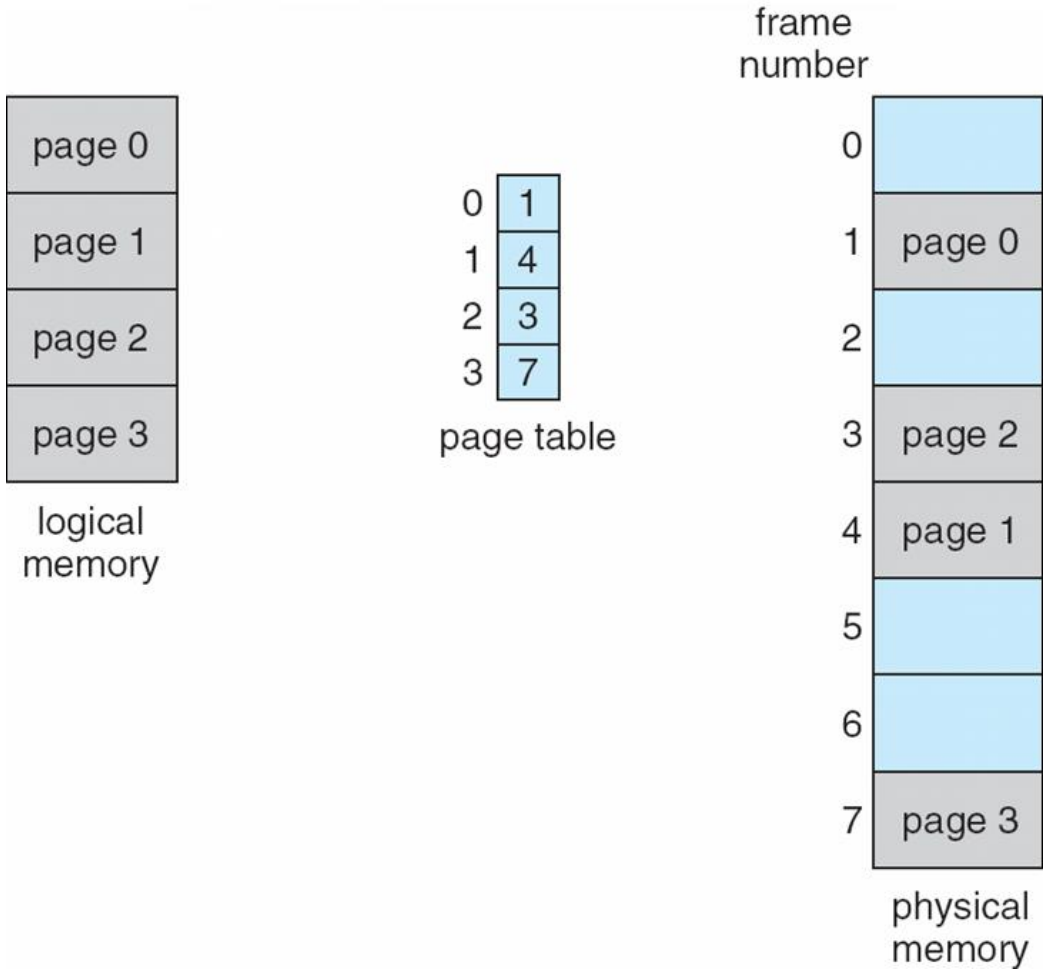
Page Table entry

- Frame number:
- It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames.Frame bit is also known as address translation bit.
- Number of bits for frame = Size of physical memory/frame size

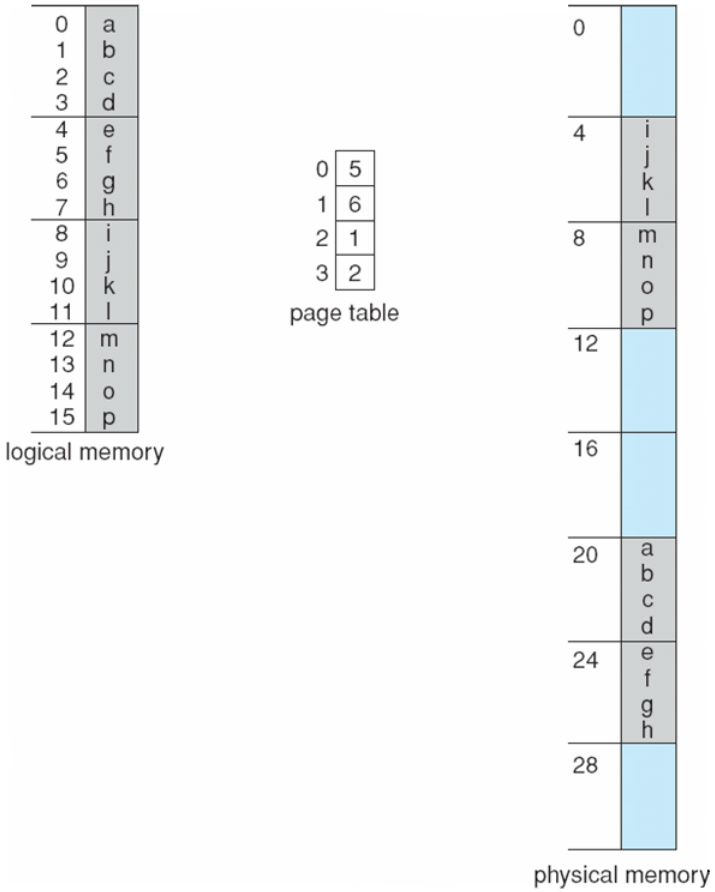


PAGE TABLE ENTRY

Paging Model of Logical and Physical Memory



Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Important Formulas

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

Important Points

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

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

Problem 1

- **Find the size of the memory if its address consists of 22 bits. Assume the memory is 2-byte addressable.**

- We have-
  - Number of locations possible with 22 bits = 2^{22} locations
  - It is given that the size of one location = 2 bytes
- Thus, Size of memory
- = 2^{22} x 2 bytes
- = 2^{23} bytes
- = 8 MB

Problem 2

- **Calculate the number of bits required in the address for memory having size of 16 GB. Assume the memory is 4-byte addressable.**

Let ‘n’ number of bits are required. Then, Size of memory = 2^n x 4 bytes.
Since, the given memory has size of 16 GB, so we have-
2^n x 4 bytes = 16 GB
2^n x 4 = 16 G
2^n x 2^2 = 2^{34}
2^n = 2^{32}
∴ n = 32 bits

Problem 3

- Consider a system with byte-addressable memory, logical address is 33bits, physical address is 24bits and size of page (total) is 2kb.
- What will be the size of memory?
- No. of bits in the offset
- No. of pages & frames
- No. of entries in page table = No. of pages in process
- Size of the page table

Problem 3

- S.M = 8GB
- M.M = 16 MB

- Size of page = 2kb
- No. of bits = 11 bits
- No. of bits in offset =11
- P= 33-11 =22

Problem No. 4

- Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each. Calculate size of page table.

- Given-
- Number of bits in logical address = 32 bits
- Page size = 4KB
- Page table entry size = 4 bytes

Solution

Number of entries in page table = 232 / 4Kbyte

= 232 / 212

= 220

Size of page table = (No. page table entries)*(Size of an entry)

= 220 * 4 bytes

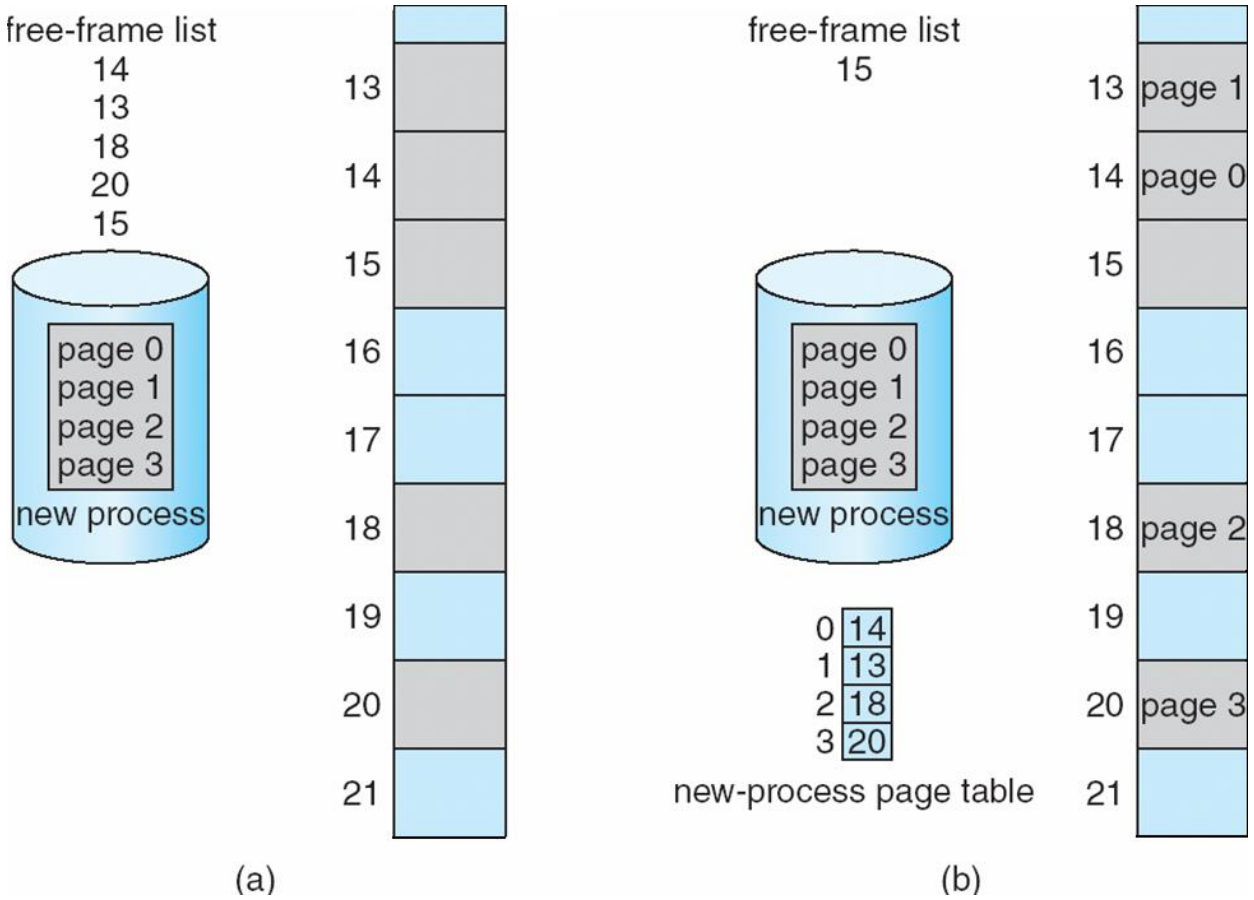
= 222

= 4 Megabytes

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and 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-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

Disadvantage of paging

One major disadvantage of paging is-

- It increases the effective access time due to increased number of memory accesses.
- One memory access is required to get the frame number from the page table.
- Another memory access is required to get the word from the page.

Optimal Page Size

In paging scheme, there are mainly two overheads-

1. **Overhead of Page Tables-**

Paging requires each process to maintain a page table.

So, there is an overhead of maintaining a page table for each process.

2. **Overhead of Wasting Pages-**

There is an overhead of wasting last page of each process if it is not completely filled.

On an average, half page is wasted for each process.

Thus,

Total overhead for one process

= Size of its page table + (Page size / 2)

Optimal Page Size

Optimal page size is the page size that minimizes the total overhead.

$$\text{Optimal page size} = \sqrt{2 \times \text{Process size} \times \text{Page table entry size}}$$

Problem 1

In a paging scheme, virtual address space is 4 KB and page table entry size is 8 bytes. What should be the optimal page size?

We know-

Optimal page size

= (2 x Process size x Page table entry size)1/2

= (2 x 4 KB x 8 bytes)1/2

= (216 bytes x bytes)1/2

= 28 bytes

= 256 bytes

Thus, Optimal page size = 256 bytes.

Problem 2

In a paging scheme, virtual address space is 256 GB and page table entry size is 32 bytes. What should be the optimal page size?

We know-

Optimal page size

$= (2 \times \text{Process size} \times \text{Page table entry size})^{1/2}$

$= (2 \times 256\ \text{GB} \times 32\ \text{bytes})^{1/2}$

$= (2^{44}\ \text{bytes} \times \text{bytes})^{1/2}$

$= 2^{22}\ \text{bytes}$

$= 4\ \text{MB}$

Thus, Optimal page size = 4 MB.

Translation Lookaside Buffer-

Translation Lookaside Buffer (TLB) is a solution that tries to reduce the effective access time.

Being a hardware, the access time of TLB is very less as compared to the main memory.

Structure-

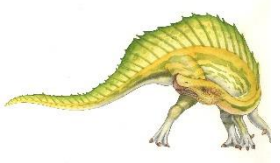
Translation Lookaside Buffer (TLB) consists of two columns-

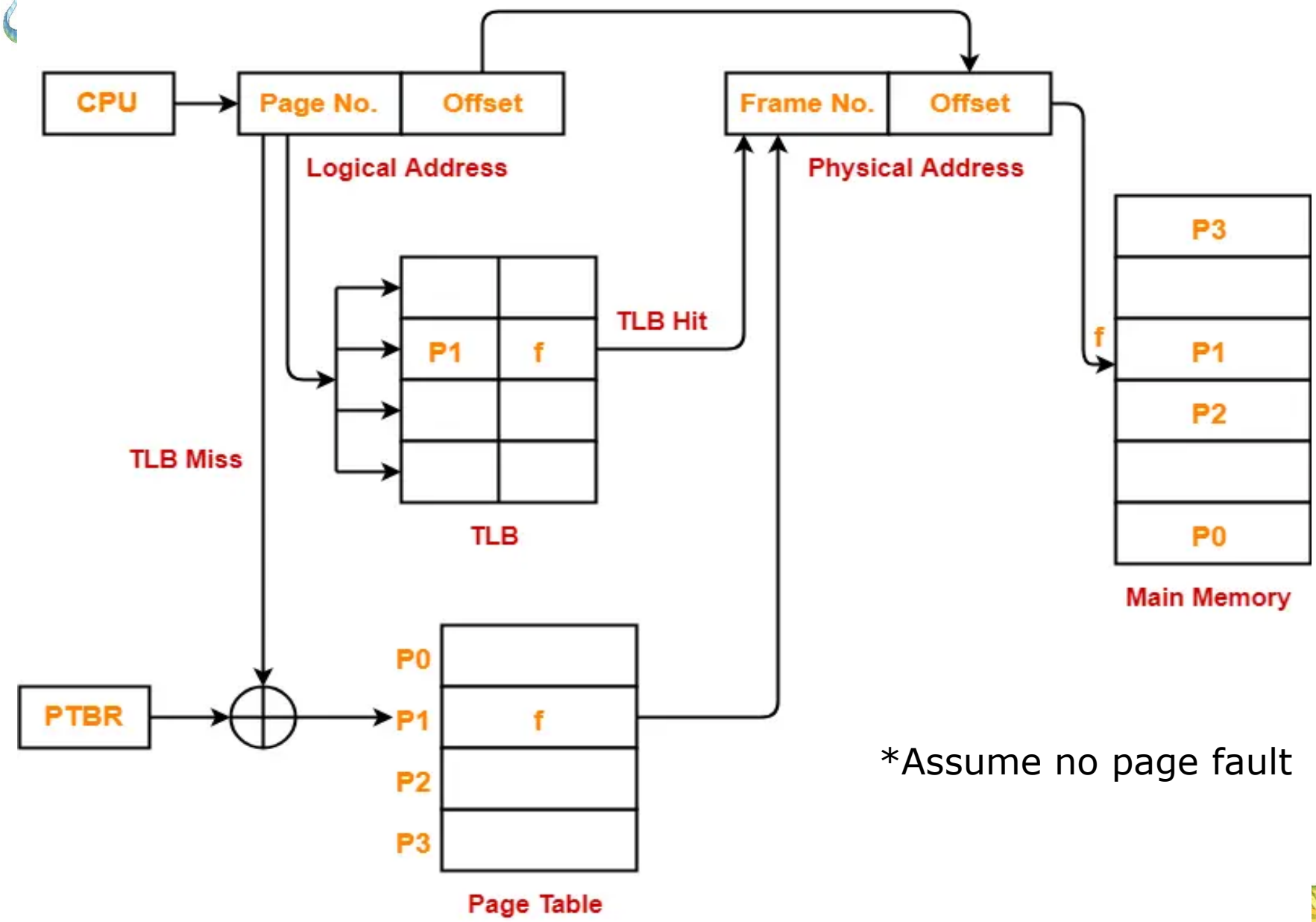
Page Number

Frame Number

Page Number	Frame Number

Translation Lookaside Buffer





*Assume no page fault

Translating Logical Address into Physical Address

TLB

- Unlike page table, there exists only one TLB in the system.
- So, whenever context switching occurs, the entire content of TLB is flushed and deleted.
- TLB is then again updated with the currently running process.

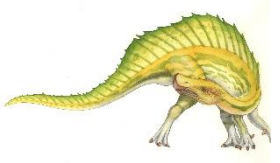
- **When a new process gets scheduled-**
- **Initially, TLB is empty. So, TLB misses are frequent**.
- With every access from the page table, TLB is updated.
- After some time, TLB hits increases and TLB misses reduces.
- The time taken to update TLB after getting the frame number from the page table is negligible.
- Also, TLB is updated in parallel while fetching the word from the main memory.
-

Page fault

When a page referenced by the CPU is not found in the main memory, it is called as a page fault.

- When a page fault occurs, the required page has to be fetched from the secondary memory into the main memory.

Page Table Entry Format



Page Fault Handling Routine-

The following sequence of events take place-

- The currently running process is stopped and context switching occurs.
- The referenced page is copied from the secondary memory to the main memory.
- If the main memory is already full, a page is replaced to create a room for the referenced page.
- After copying the referenced page successfully in the main memory, the page table is updated.
- When the execution of process is resumed, step-02 repeats.

TLB

Effective Access Time-

In a single level paging using TLB, the effective access time is given as-

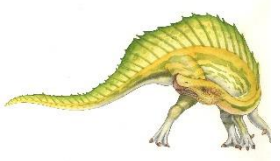
Effective Access Time =

Hit ratio of TLB x { Access time of TLB + Access time of main memory }

+

Miss ratio of TLB x { Access time of TLB + 2 x Access time of main memory }

This formula is valid only when there is single level paging and there are no page faults.



TLB: Problem 1

A paging scheme uses a Translation Lookaside buffer (TLB). A TLB access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page fault?

Solution

Calculating TLB Miss Ratio-

TLB Miss ratio

= 1 – TLB Hit ratio

= 1 – 0.9

= 0.1

Calculating Effective Access Time-

Effective Access Time

= 0.9 x { 10 ns + 50 ns } + 0.1 x { 10 ns + 2 x 50 ns }

= 0.9 x 60 ns + 0.1 x 110 ns

= 54 ns + 11 ns

= 65 ns

TLB: Problem 2

A paging scheme uses a Translation Lookaside buffer (TLB). The effective memory access takes 160 ns and a main memory access takes 100 ns. What is the TLB access time (in ns) if the TLB hit ratio is 60% and there is no page fault?

Solution

Calculating TLB Miss Ratio-

TLB Miss ratio

= 1 – TLB Hit ratio

= 1 – 0.6

= 0.4

Calculating TLB Access Time-

Let TLB access time = T ns.

Substituting values in the above formula, we get-

160 ns = 0.6 x { T + 100 ns } + 0.4 x { T + 2 x 100 ns }

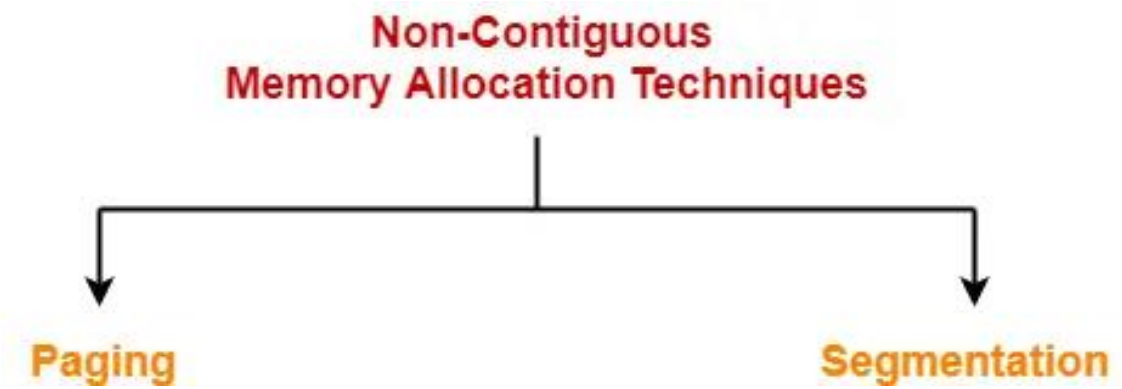
160 = 0.6 x T + 60 + 0.4 x T + 80

160 = T + 140

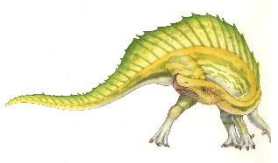
T = 160 – 140

T = 20

Segmentation-



- 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.



Segmentation

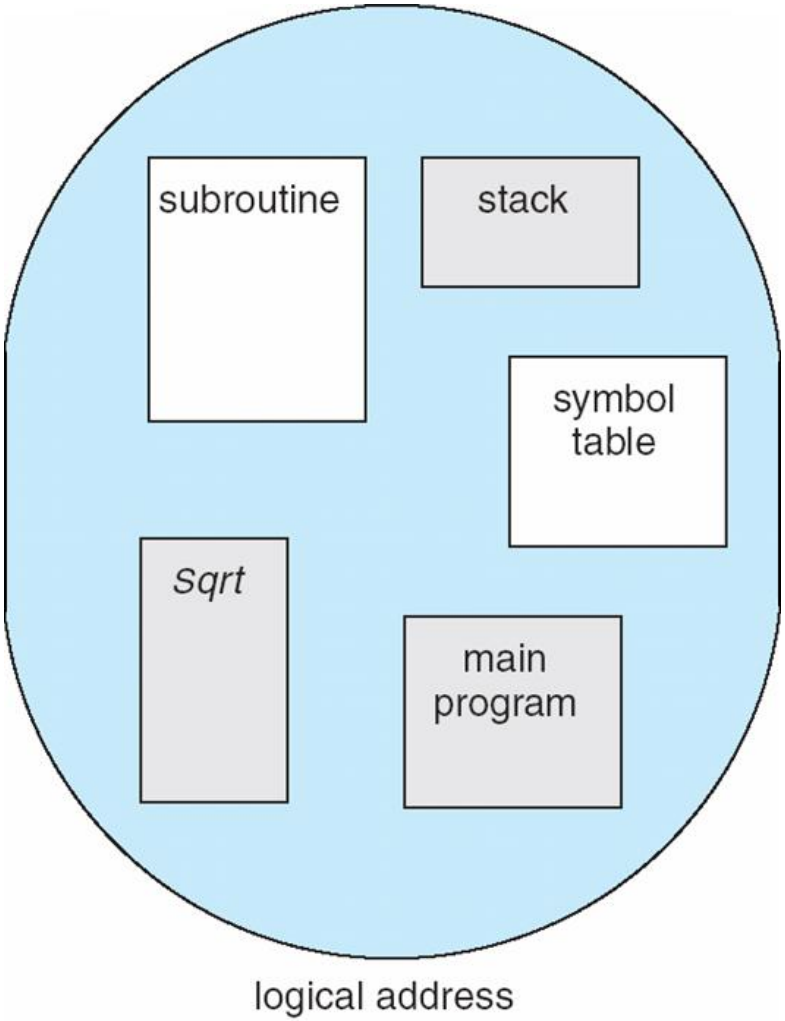
Memory-management scheme that supports user view of memory

A program is a collection of segments

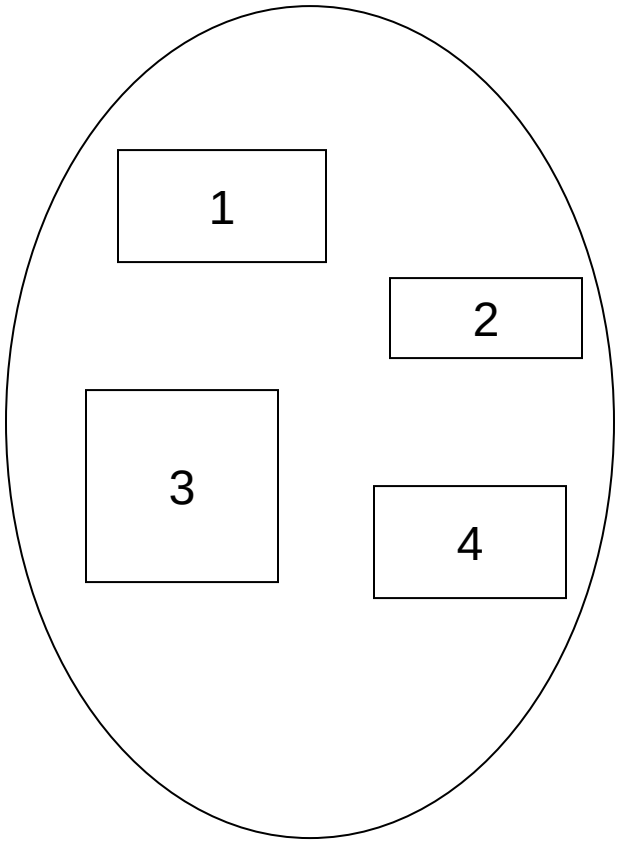
A segment is a logical unit such as:

- main program
- procedure
- function
- method
- object
- local variables, global variables
- common block
- stack
- symbol table
- arrays

User's View of a Program



Logical View of Segmentation



user space

physical memory space

Segmentation Architecture

Logical address consists of a two tuple:

<segment-number, offset>,

Segment table – maps two-dimensional physical addresses; each table entry has:

base – contains the starting physical address where the segments reside in memory

limit – specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory

Segment-table length register (STLR) indicates number of segments used by a program;

segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture (Cont.)

Protection

With each entry in segment table associate:

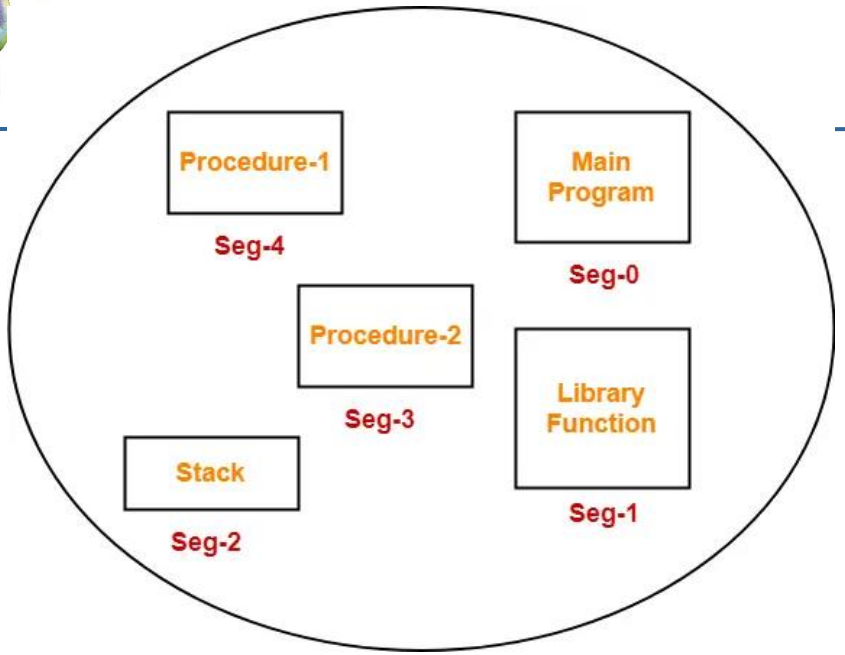
validation bit = 0 $\Rightarrow$ illegal segment

read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level

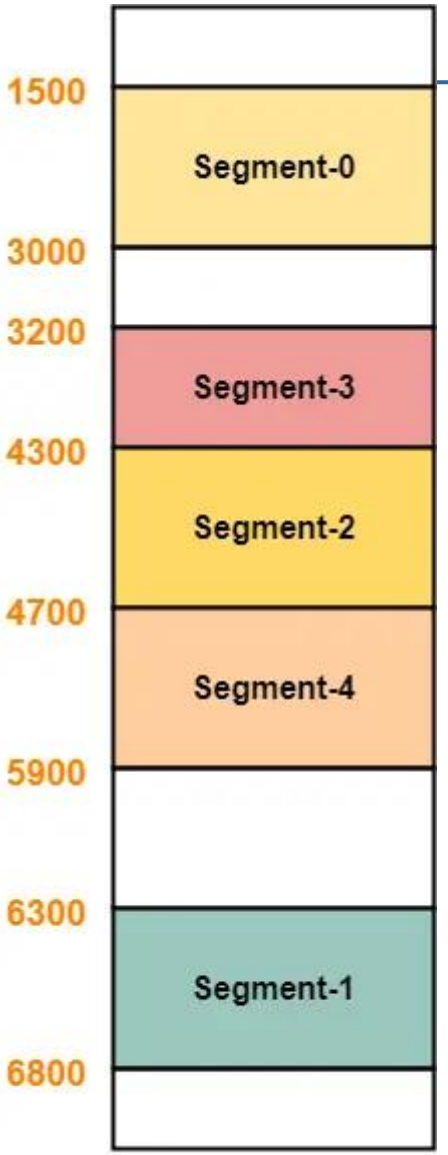
Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram



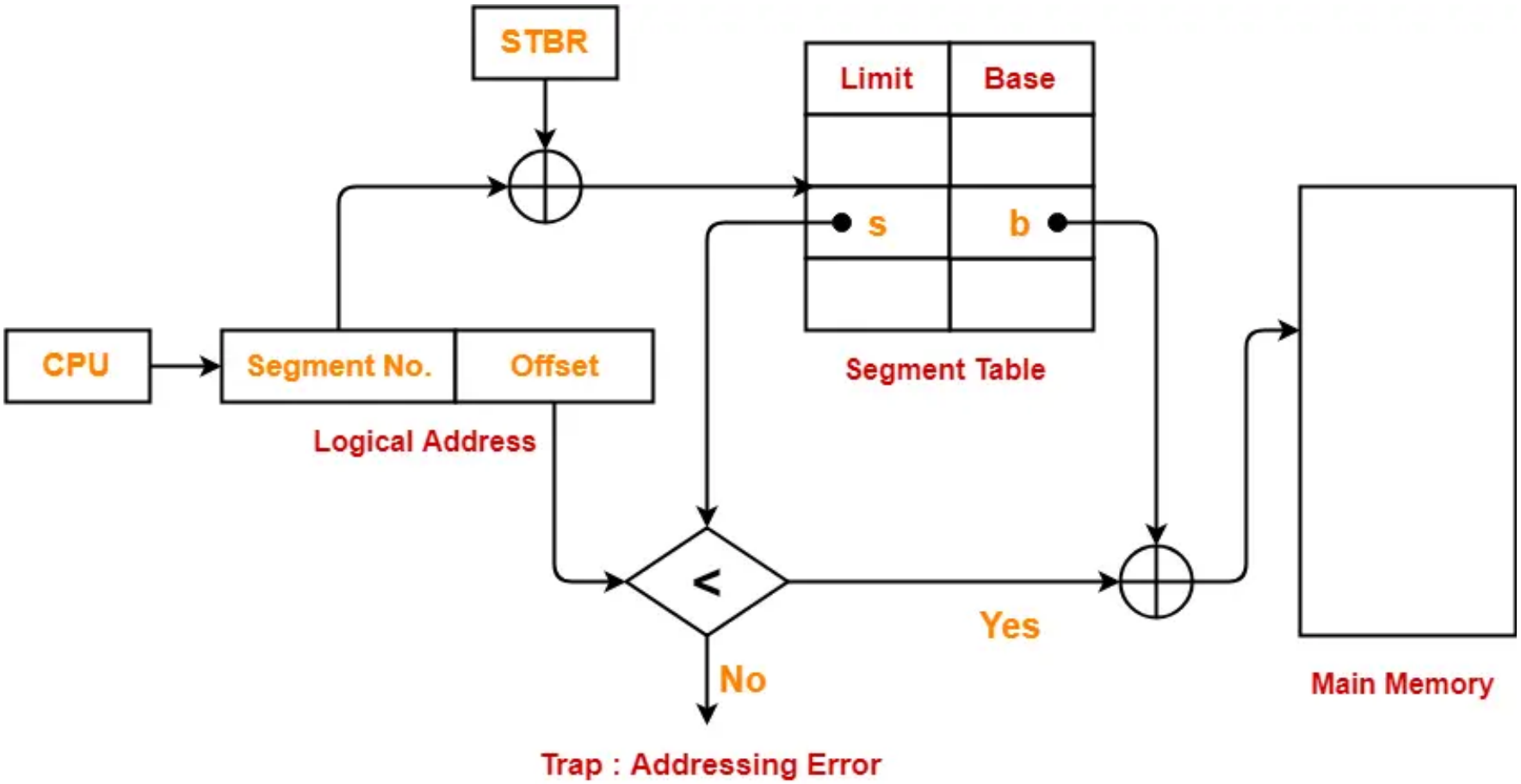
	Limit	Base
Seg-0	1500	1500
Seg-1	500	6300
Seg-2	400	4300
Seg-3	1100	3200
Seg-4	1200	4700

Segment Table



Main Memory

Segmentation Hardware



Translating Logical Address into Physical Address

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.

PROBLEM BASED ON SEGMENTATION-

Segment No.	Base	Length
0	1219	700
1	2300	14
2	90	100
3	1327	580
4	1952	96

Which of the following logical address will produce trap addressing error?

1.0, 430
2.1, 11
3.2, 100
4.3, 425
5.4, 95

Calculate the physical address if no trap is produced.

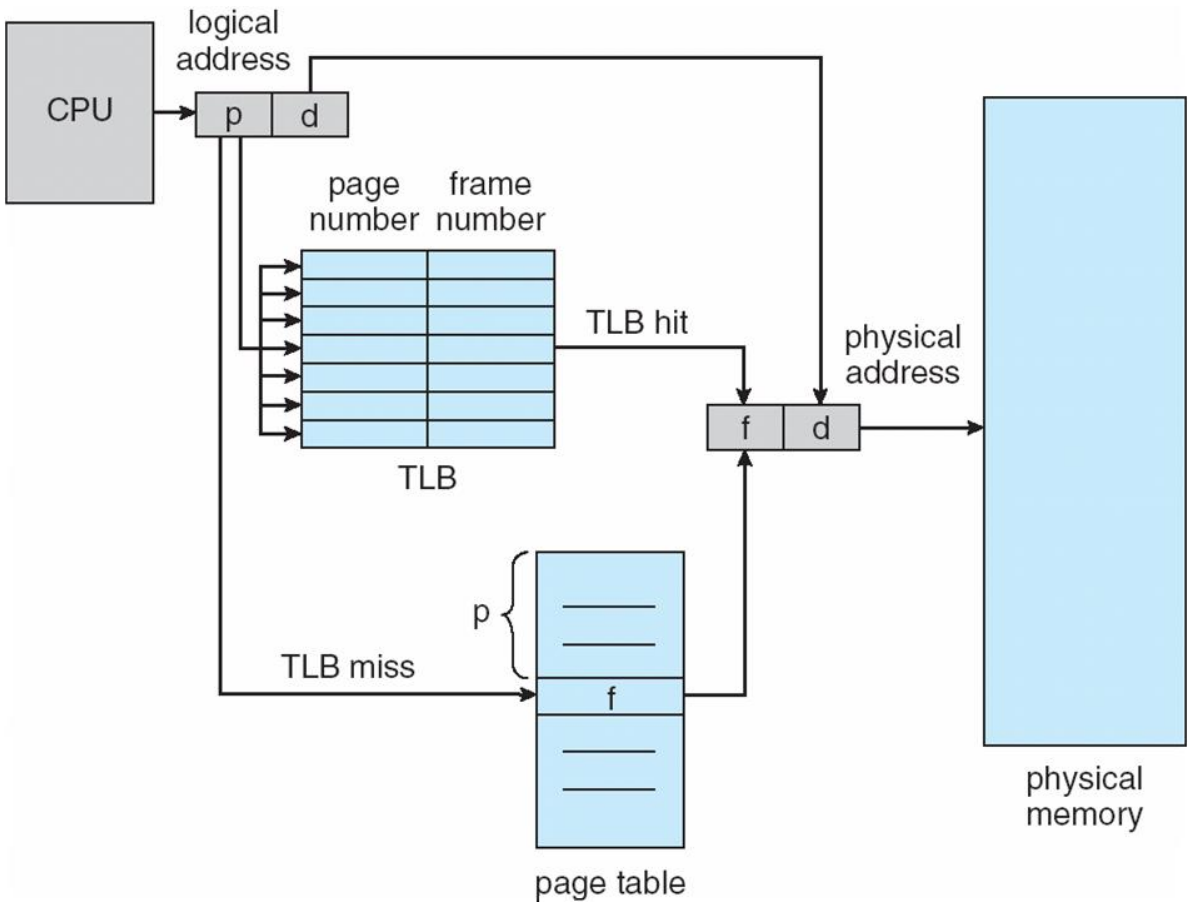
Associative Memory

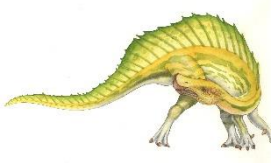
- Associative memory – parallel search

Page#	Frame#

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB





Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

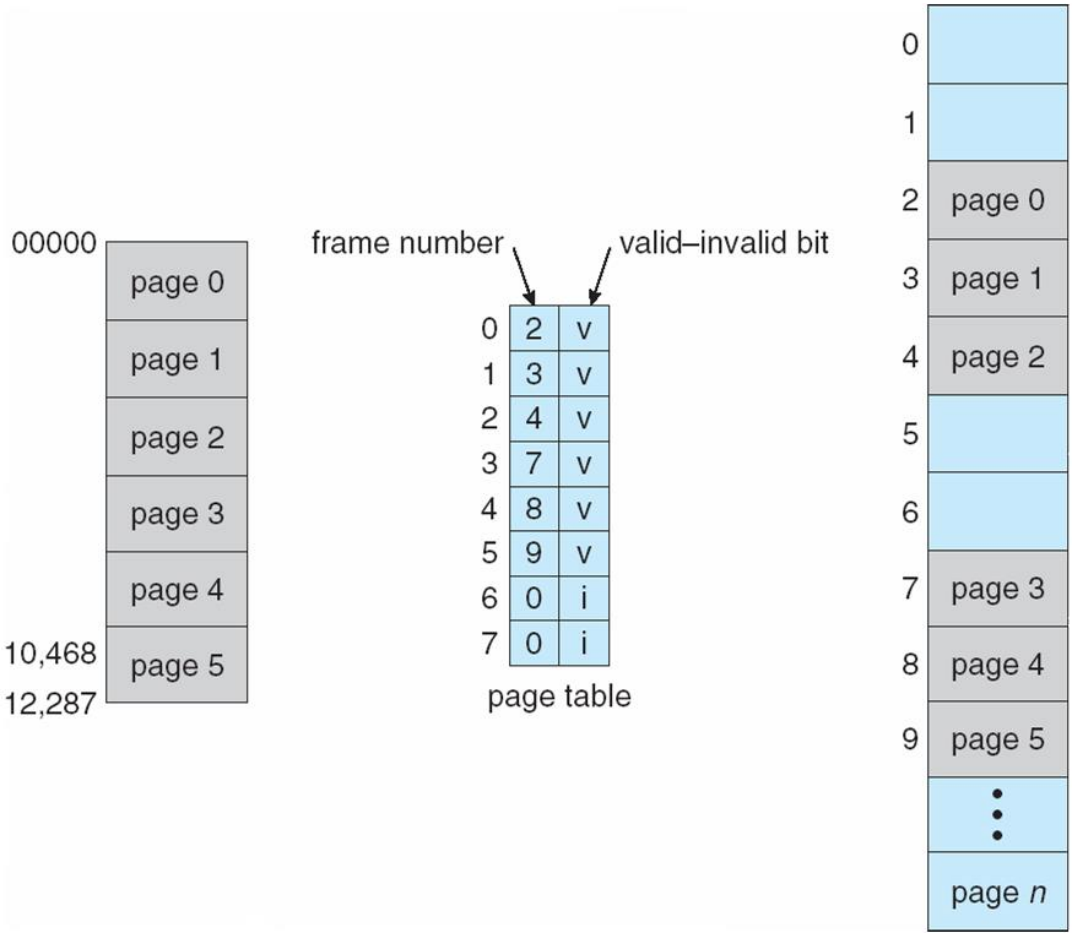
$$= 2 + \varepsilon - \alpha$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



Structure of the Page Table

Memory structures for paging can get huge using straight-forward methods

- Consider a 32-bit logical address space as on modern computers
- Page size of 4 KB (2^{12})
- Page table would have 1 million entries ($2^{32} / 2^{12}$)
- If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
  - That amount of memory used to cost a lot
  - Don't want to allocate that contiguously in main memory

Swapping

A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution

- Total physical memory space of processes can exceed physical memory

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

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

Does the swapped out process need to swap back in to same physical addresses?

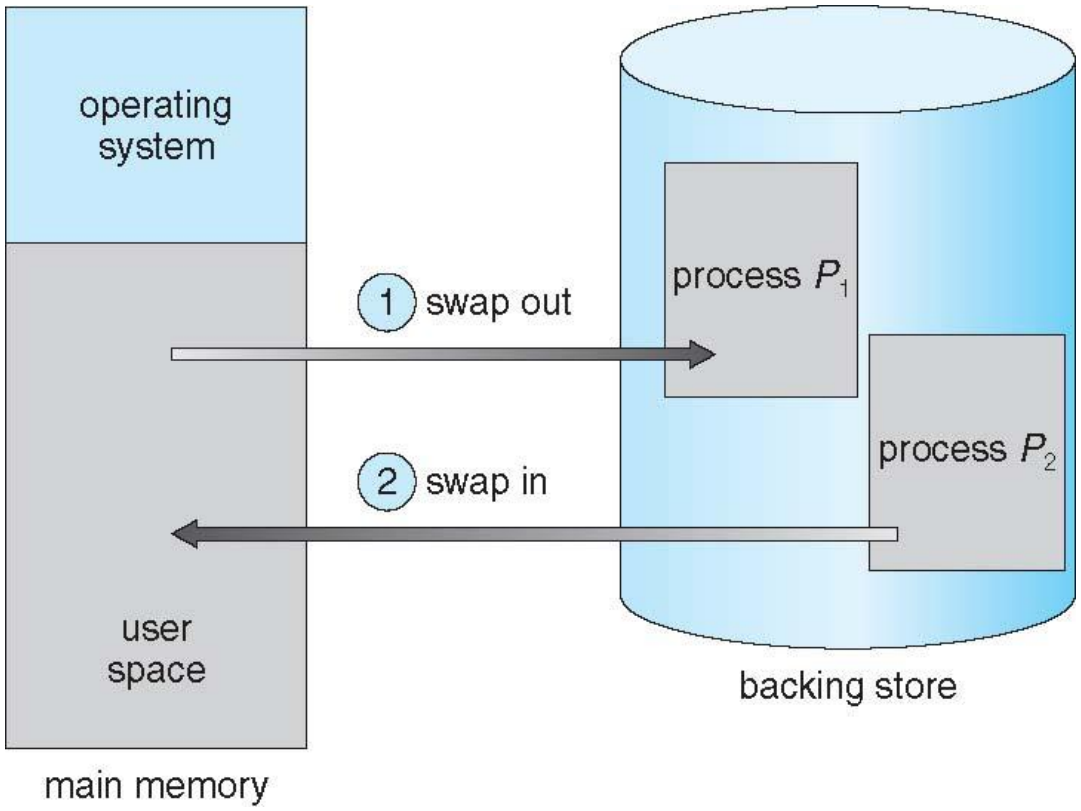
Depends on address binding method

- Plus consider pending I/O to / from process memory space

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

- Swapping normally disabled
- Started if more than threshold amount of memory allocated
- Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Context Switch Time including Swapping

If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process

Context switch time can then be very high

100MB process swapping to hard disk with transfer rate of 50MB/sec

- Swap out time of 2000 ms
- Plus swap in of same sized process
- Total context switch swapping component time of 4000ms (4 seconds)

Can reduce if reduce size of memory swapped – by knowing how much memory really being used

- System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

Other constraints as well on swapping

- Pending I/O – can't swap out as I/O would occur to wrong process
- Or always transfer I/O to kernel space, then to I/O device
  - Known as **double buffering**, adds overhead

Standard swapping not used in modern operating systems

- But modified version common
  - Swap only when free memory extremely low

End of Chapter