Unit-6 :- Memory management.

- memory management is one of the most important responsibilities of an Operating system

- **Paging & segmentation -**

- Two commonly used memory management schemes are Paging and Segmentation.
- Both help in efficient allocation of memory, but they work in completely different ways.

- **Paging -**

- Paging is a non-contiguous memory allocation technique
- It divides the logical memory of a process into fixed-size blocks called pages, and divides physical memory (RAM) into same size blocks called frames.

- **Characteristics -**

① **fixed size pages & frames**
- Page size = frame size
- Allows easy mapping

② **No external fragmentation**
- since frames are equal sized, memory never gets scattered and unusable

③ **Possible internal fragmentation**
- last page may not fully use the frame

④ **Each process has a Page Table -**
- Used to translate logical page number → physical frame no.

⑤ logical address = page number + offset

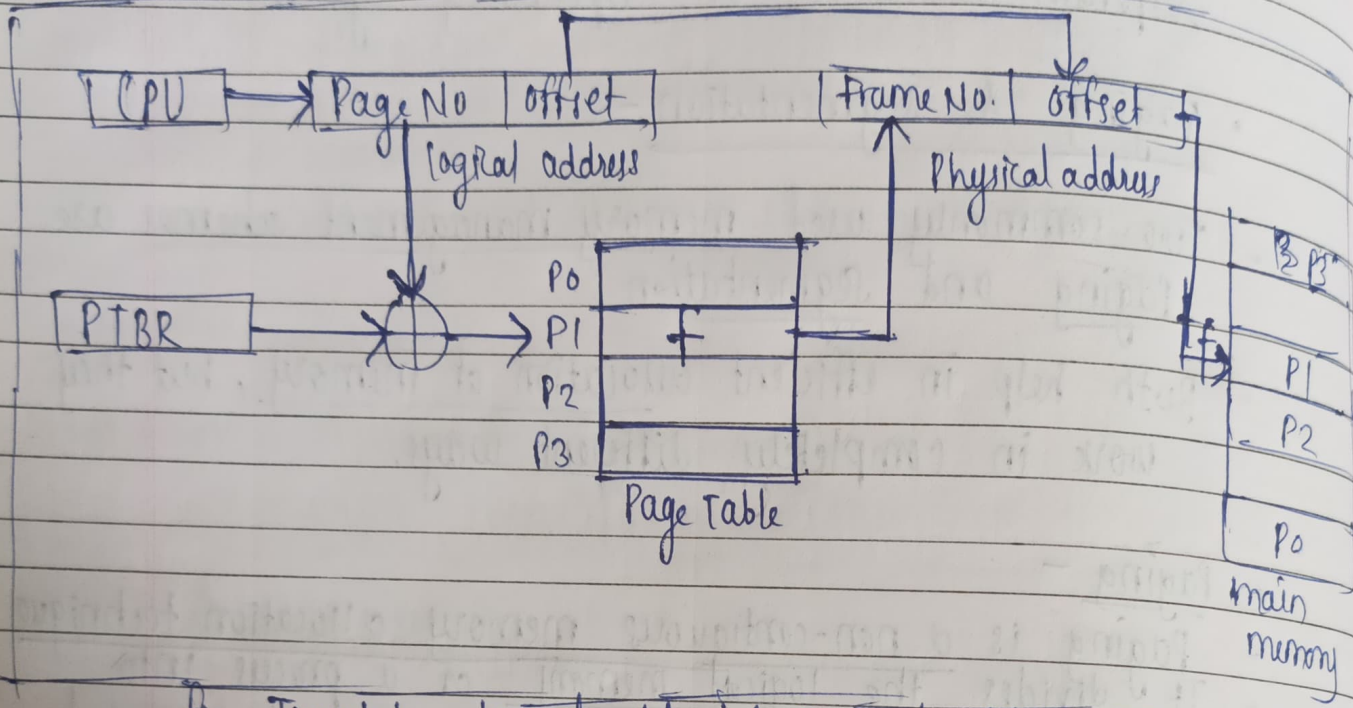


fig. Translating Logical addr into Physical addr.

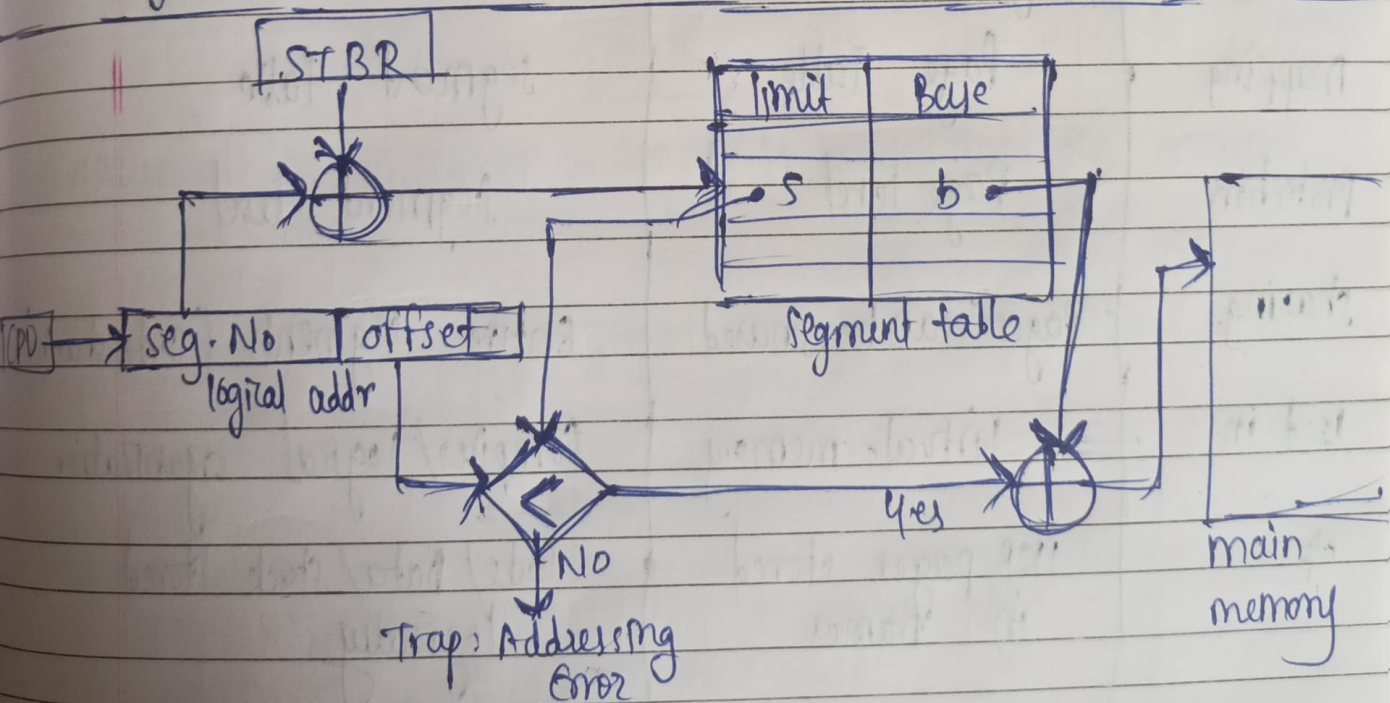
Advantages - eliminates external fragmentation
- simple memory allocation
- fast switching betn processes
- supports virtual memory

Disadv - Internal fragmentation
- Page table overhead
- logical grouping of program is lost
(Instn / data broken into fixed blocks)

- Segmentation
Segmentation divides a program into logical variable sized sections such as -
- code segment, data segment, stack segment,

- Heap segment, functions/ modules
- segments have different sizes (not fixed)

Characteristics -
1. variable - sized segments
2. Divides program based on logical structure, not size
3. No internal fragmentation
4. External fragmentation occurs
5. each process has a segment table
   - Base → starting addr
   - Limit → size of segment
6. logical address = segment number + offset



adv - - supports sharing
- supports protection
- No internal fragmentation
- Occurs independently

disadv - External fragmentation
- Complex memory allocation
- variable sized segments
- Possibility of segmentation faults

Feature	Paging	Segmentation
Basic Idea	Divide process into fixed size pages.	Divide process into logical variable size segments
Division based on	Size	Program structure
Size	fixed	variable
Fragmentation	Internal	external
Logical view to user	Not visible	visible to user (code, data, stack)
Addressing	Page no + offset	Segment no + offset
Mapping	Page Table	segment table
Protection	Page level	segment level
Sharing	Pages can be shared	Entire segments (code sharing)
Used in	Virtual memory	Compiler/logical organization
eg.	KB pages stored in frames	Code/Data/stack stored logically

- Fixed Partitioning - (Static)
  - Fixed partitioning is the simplest & oldest method of memory management.
  - The main memory is divided into a set of fixed size regions (partitions) that are determined when the system is started, & they do not change afterward

Working principle -

- fixed size - The main mem is divided into a fixed no. of partitions which can be equal or unequal size.

- One Process Per Partition - Each partition can hold exactly one process.

- Allocation - When a process arrives, it is loaded into the smallest available partition that is large enough to hold it

- Limitation - A process larger than the largest partition cannot be executed.

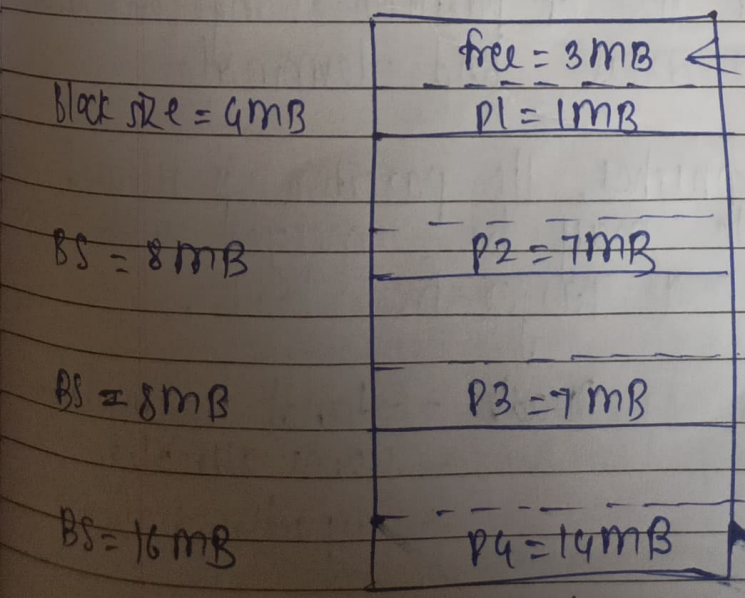


fig. Fixed size partition

Adv - Simple & easy to implement
- low overhead
- fast allocation

disadv - Internal fragmentation
- limited no. of processes
- partition size problem

- Dynamic Partitioning - (Variable)

- Dynamic Partitioning overcomes the limitations of fixed partitioning by allocating memory dynamically
- In Dynamic partitioning, memory is not divided in advance
- Partitions are created at runtime, exactly as per process size

Working principle -

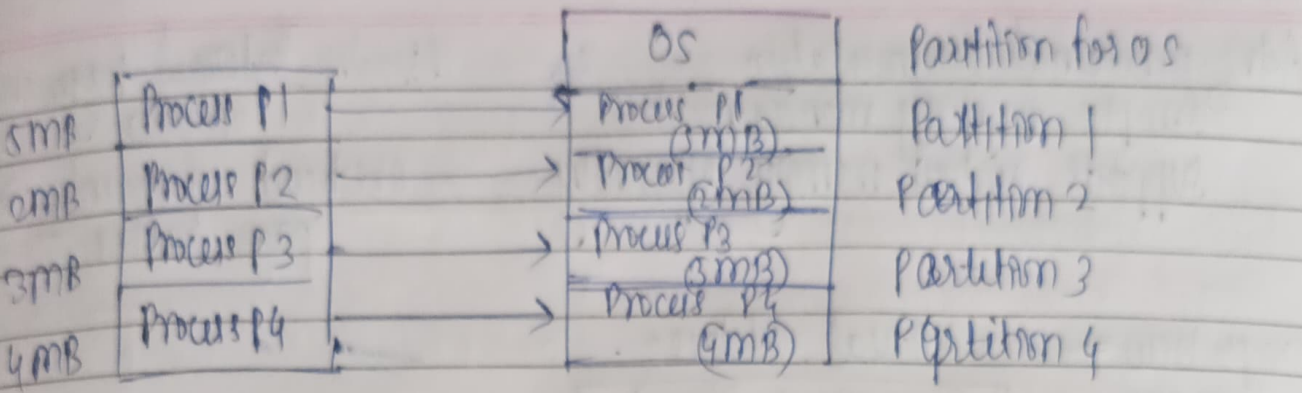
- Dynamic size - Initially, all mem is one large free block, called hole
- Exact fit - when a process arrives, OS carves out a partition exactly equal to the size of the process

&

- Partition creation - The no & size of partition change over time as processes arrive and terminate
- Process exit - When a process finishes, its partition is released creating a Hole. This hole may be merged with adjacent holes to form a larger free block.

Adv - No internal fragmentation
- Flexible memory usage
- Better utilization of RAM

Disadv - External fragmentation
- Slower allocation
- Compaction required



Dynamic partitioning
(Process size = Partition size)

- **Virtual Memory** :- It is a memory management technique that allows a computer to run programs larger than the actual physical RAM by using both RAM + Hard DISK (or SSD).

- VM with Paging :-
  - In Paging, both physical memory (RAM) and virtual memory are divided into fixed size blocks.
    - Virtual memory blocks = Pages
    - Physical memory blocks = Frames.
  - each page maps to a frame in physical memory via a page table.

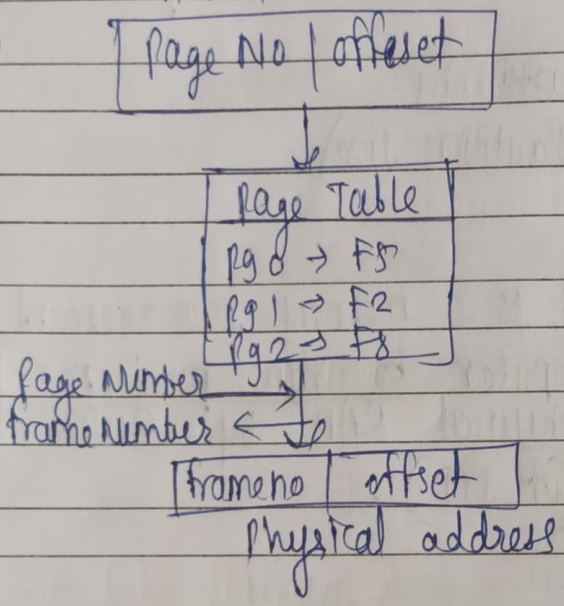
Working:
① CPU generates a virtual address (logical addr)
② virtual address is divided into -
- Page number → index in page table
- Page offset → position inside the table.

③ Page table translates page number → frame number in Physical memory
④ Physical address = frame number + page offset → memory is accessed.

Adv - no external fragmentation
Simple memory management
supports virtual memory efficiently

Disadv - Internal fragmentation can occur
- overhead of maintaining page tables

Simple new diagram



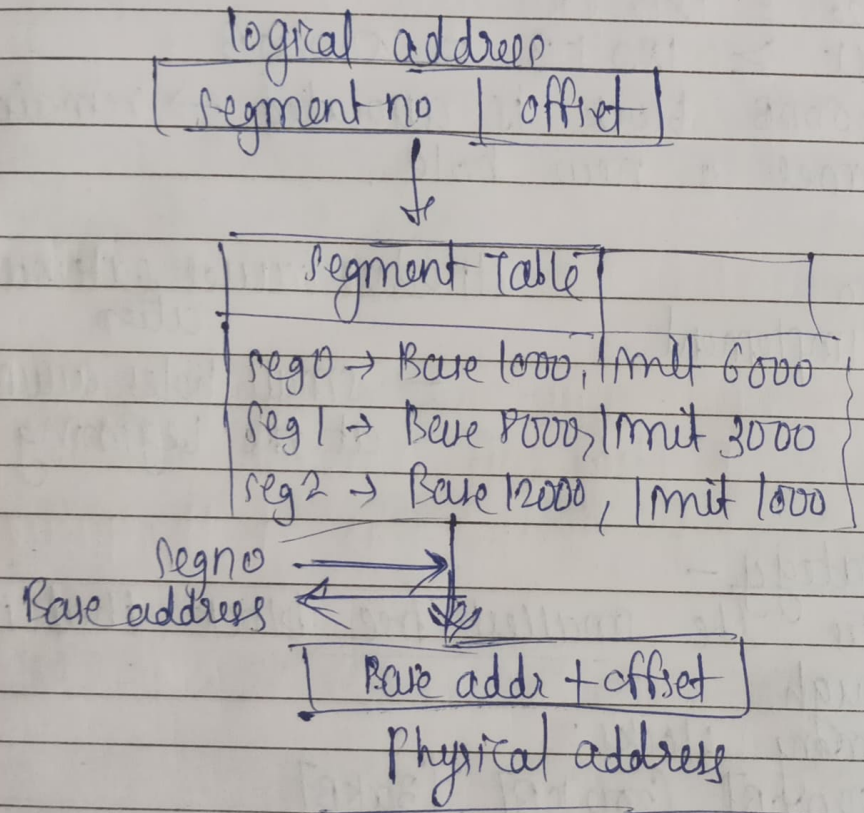
- **VM with Segmentation** -
  - In segmentation, memory is divided into logical segments of variable size based on program structure.
    eg- segments - Code, Stack, Data
  - Each seg has a base & limit in physical memory
  - Virtual address = (segment number, offset)

Working ① CPU generates logical address = (segment no, offset)
② Segment table gives
- Base addr of the segment in physical memory
- limit (size of the seg)

③ Physical address = base + offset → memory is accessed
④ if offset > limit → segmentation fault

Adv - easier to manage
- supports protection
- No internal fragmentation

disadv - external fragmentation
- more complex



• Memory Allocation Strategies -

- Memory allocation strategies are used by the OS to allocate free memory blocks to processes.
- When processes request memory, the OS searches the free list and chooses a suitable block using one of these strategies:
  - first fit
  - worst fit
  - Best fit
  - Next fit
- Each method decides how to choose a block from free memory

① First Fit strategy-

- It allocates the first free block that is large enough for the process

eg. Free memory blocks:
[100 KB], [500 KB], [200 KB], [300 KB]
Process size = 180 KB
first block ≥ 180 KB = 500 KB
so the 500KB block is allocated → remaining 320 KB becomes a new hole.

Adv - Fastest algo
- simple to implement

disadv - Causes external fragmentation
- small holes accumulate at the beginning.

② Best Fit strategy -

- It allocates the smallest free block that is large enough.

eg. Free memory blocks:
[100 KB], [500 KB], [200 KB], [300 KB]
Process = 180 KB
Blocks ≥ 180 KB = 200 KB, 300 KB, 500 KB
smallest = 200 KB → allocated
Remaining = 20 KB hole.

adv - Reduces leftover space
- Good mem utilization

disadv - Slow
- leaves very small holes which are unusable

③ Worst fit strategy :-
- It allocates the largest free block available

eg. free mem blocks:
[100 KB], [500 KB], [200 KB], [300 KB]
Process = 180 KB
largest block = 500 KB → allocate it.
Remaining = 320 KB hole

Adv - leaves large holes that may fit future requests
- Reduces chance of very small fragments

disadv - Poor memory utilization
- large holes breaks into medium holes → fragmentation occur

④ Next fit strategy :-
- similar to first fit, but it starts searching from the last allocated position (not from the beginning)

eg - free blocks:
[100 KB], [500 KB], [200 KB], [300 KB]
- If last allocation happened at 200 KB block, this time searching starts from 300 KB block, not from beginning.
- Process = 150 KB
300 KB block is allocated.

Adv - faster than first fit in long free lists.
- Distributes memory allocation more evenly.

disadv - still causes fragmentation.
may skip good blocks at the beginning.

Page Replacement strategies -

- Page replacement algorithms are used in virtual memory systems when a page fault occurs & there is no free frame available.
- The OS must select one page from memory to remove & replace it with the required page from secondary storage.
- Two commonly used page replacement strategies are —

① FIFO (First in first out) Page Replacement —

- FIFO replaces the page that has been in memory for longest time.
- It uses a queue to maintain the order of pages loaded into memory.

Working of FIFO -

① Pages are inserted into memory in the order of arrival.

② When the memory is full and a new page must be loaded, the oldest page is removed.

③ New page is added at the rear of the queue.

eg - Reference string - 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2
Number of frames = 3.

Rule - Replace the page that entered memory earliest.
(Use a queue: front = oldest, rear = newest)

Step	Reference	Frames		Page fault
1.	2	[2, –, –]		Fault
2.	3	[2, 3, –]		Fault
3.	2	[2, 3, –]		Hit
4.	1	[2, 3, 1]		Fault
5.	5	[5, 3, 1]	[replace 2]	Fault
6.	2	[5, 2, 1]	[replace 3]	Fault
7.	4	[5, 2, 4]	[replace 1]	Fault
8.	5	[5, 2, 4]		Hit

9.	3	[3,2,4] (replaces 5)	fault
10.	2	[3,2,4] replaces 2	Hit
11.	5	[3,5,4] replace 2	fault
12.	5	[3,5,2] replace 4	fault

Total Page faults = 09.

Adv - very simple to understand & implement.
- Low overhead ; uses a basic queue structure

disadv - Removes oldest page even if it is frequently used
Poor performance compared to LRU & optimal

② LRU: Least Recently Used -

- LRU replaces the page that has not been used for the longest time.
- This is about usage time, NOT insertion time.
- It tracks the last use time & replaces accordingly

Working ① The OS tracks the last used time of each page
② when a replacement is needed, the page that was least recently accessed is removed
③ LRU can be implemented using -
• Counters • Stack method

2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

Step	Reference	Frames	Page fault	LRU reasoning
1.	2	[2, -, -]	F	2 most recent
2.	3	[2, 3, -]	F	3 recent, then 2
3.	2	[2, 3, -]	H	2 most recent
4.	1	[2, 3, 1]	F	1 recent, order-1,2,3
5.	5	[2, 5, 1] (replace 3)	F	3 was least RU
6.	2	[2, 5, 1]	H	2 most recent
7.	4	[2, 5, 4] (replace 1)	F	1 was LRU
8.	5	[2, 5, 4]	H	5 recently used
9.	3	[3, 5, 4] (replace 2)	F	2 was LRU
10.	2	[3, 5, 2] (replace 4)	F	4 was LRU
11.	5	[3, 5, 2]	H	5 recent
12.	2	[3, 5, 2]	H	2 recent

Total Page fault - 07

Adv - Better performance than FIFO
Based on actual usage, so fewer page faults

Disadv - difficult & expensive to implement
- Requires time stamp or stack → overhead.

Q. 100K, 500K, 200K, 300K, 600K.
Processes (in arrival order) - 212K, 417K, 426K

① First fit - (First free block that is large enough)

1. Process 212K → first partition ≥ 212K is 500K
Allocate 212K
leftover in that partition = 500 - 212 = 288K

100K, 288K, 200K, 300K, 600K

2. Process 417K → 600K fits
Allocate 417K → 600 − 417 = 183K
100K, 288K, 200K, 300K, 183K

3. Process 426K → cannot allocate

② Best fit. -(smallest)

1. Process 212K → 100K, 500K, 200K, 300K, 600K ⇒ 300K
Allocate 212K → leftover 300 − 212 = 88K
100K, 500K, 200K, 88K, 600K

2. Process 417K → 500K
Allocate 417K → leftover 500 − 417 = 83K
100K, 83K, 200K, 88K, 600K

3. Process 426K → 600K
Allocate 426K → leftover 600 − 426 = 174K
100K, 83K, 200K, 88K, 174K

All three processes allocated
Total leftover (sum of three) = 100 + 83 + 200 + 88 + 174 = 645K

③ worst fit (largest)

① Process 212K → largest partition 600K
600 − 212 = 388K
100K, 500K, 200K, 300K, 388K

② Process 417K → 500K
500 − 417 = 83K 100K, 83K, 200K, 300K, 388K

③ Process 426K → All are small → cannot allocate

TLB - Translation Lookaside Buffer is a special, small, fast cache inside the CPU that stores recently used page table entries. It helps in quickly converting logical addr → physical addr during paging.

- It is also called associative memory
- It stores Page Number → frame Number pairs
- It reduces the time taken for addr translation

Paging system with TLB:-

- when paging is used, logical address = Page Number + offset

Every memory access must first check the Page table to find the corresponding frame number. This takes extra time because page table is in RAM.
To make this faster, we use a TLB.

Steps of Address Translation using TLB :-

① CPU generates a Logical address
Logical address = Page Number (p) + offset (d)

② TLB lookup (first check)
- CPU checks if the Page Number 'p' is already in the TLB. This is called TLB Hit.

If TLB Hit: frame number is found directly in TLB
- Physical address = Frame Number + offset
- Very fast (no need to access page table in RAM)

If TLB Miss: Page number not found in TLB
- CPU goes to Page Table in main memory
- Gets the frame number
- Updates the TLB with this new entry
- Then generates the physical address

③ CPU finally accesses the physical memory
- The required instn / data is fetched.

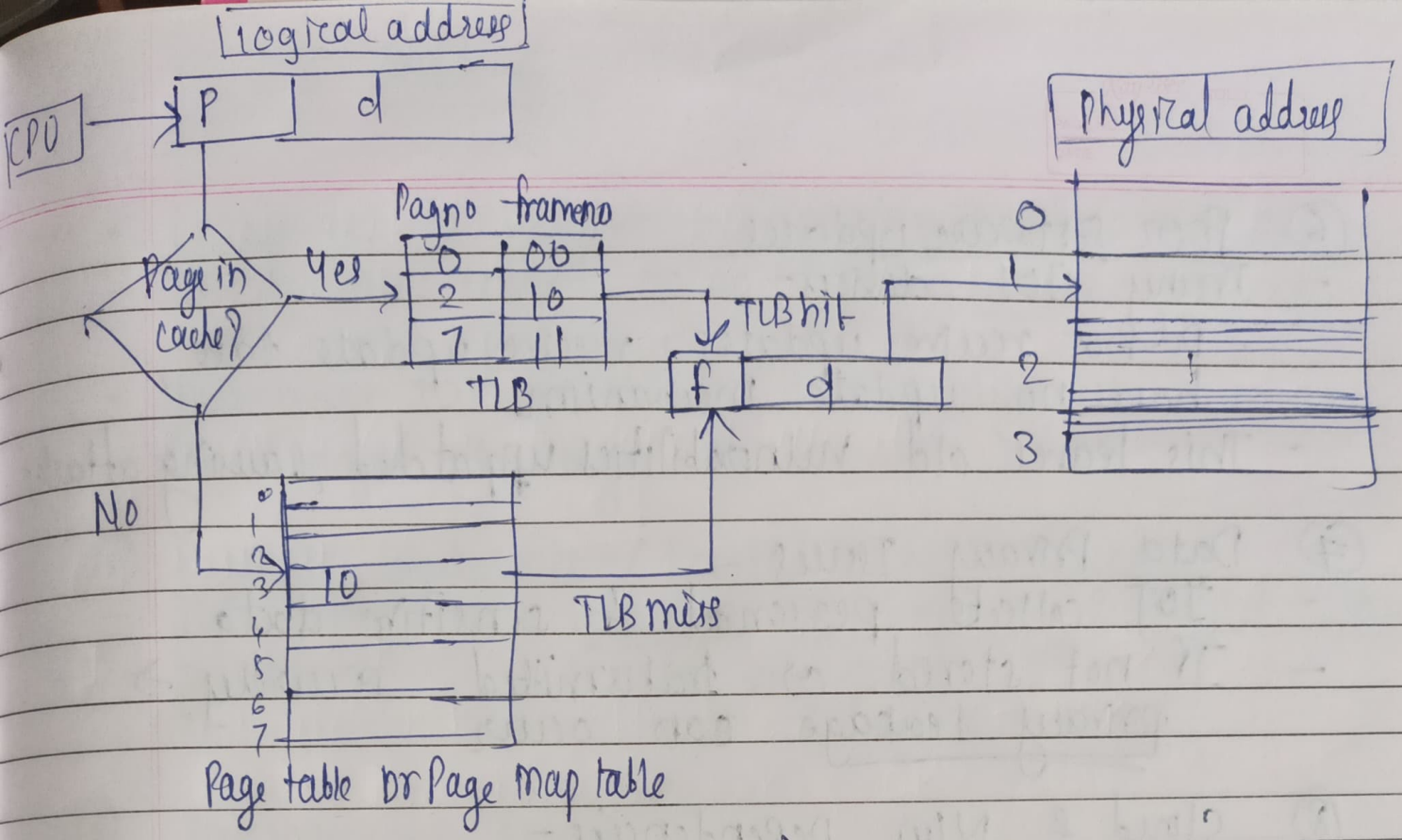


Fig TLB in paging

Adv - Faster addr translation
- Improves CPU performance
- Better utilization of paging

disadv - Complex management
- limited size
- expensive h/w

- Fragmentation - It occurs when mem is divided into small unusable pieces

Types ① Internal - unused space inside allocated block
② External - free memory is in scattered blocks

- Buddy system - is a memory allocation technique that splits the memory into blocks of size that is power of 2.

Unit-4 SPOS

Quick Work
Page No.:
Date:
M T W T F S S

Compiler	Interpreter
Translates the entire program at once	Translates one line by line at a time
Generates obj code	does not generate obj code
Faster execution after compilation	Slower execution due to line by line translation
Errors are shown after full compilation	Errors are shown immediately for each line
Requires more memory	Requires less memory
Debugging is harder because errors come at end	Debugging is easier
Execution does not need source code once compiled	Execution requires source code every time
Suitable for large programs	Suitable for small or scripting pgms
Used in langs like C, C++, Java	Used in Python, Javascript, Ruby
Compilation is time consuming	No separate compilation time

Quick Work

Page No.:
Date:
M T W T F S S

Token - A token is a category or class of basic syntactic units in a programming language
- It represents a group of characters that have a collective meaning.

eg. keywords - if, while, for
Identifiers - sum, total
operators - +, =
Punctuation - ;, {, }

Pattern - A pattern is a rule or description that defines the form of a token
- It tells what sequence of characters can be classified under a particular token

eg. Identifier pattern: [A-Za-z] [A-Za-z0-9]*
Integer constant pattern - [0-9]+

lexeme A lexeme is the actual string of characters in the source code that matches a token's pattern

eg. In the statement: int total = 10;
int → keyword (lexeme)
total → identifier (lexeme)
10 → Integer constant (lexeme)

lexical errors lexical errors are errors detected by the lexical analyzer when it encounters invalid characters or invalid sequences that don't match any token pattern.

eg. Invalid symbol - @value = 5;
Unfinished string = "Hello
Illegal no - 12AB
using space inside identifiers - my var = 5;

Quick Work
Page No.:
Date:
M T W T F S S

Scheduler - Schedulers are OS components that decide which process gets the CPU, when it gets it, & how long it stays in memory.

① Long term scheduler (Job scheduler)
- It decides which processes are admitted into the ready queue (ie into main mem)
function - Control the degree of multiprogramming
- Selects jobs from the job pool on disk & moves them to mem
Goal - maintain a proper balance betn -
· I/O bound processes · CPU-bound processes
frequency - rarely runs.
Adv - control system load ensures good mix of processes

② short term scheduler (CPU scheduler)
The short term scheduler selects one process from the ready queue & allocates the CPU to it
function - Performs context switching
Chooses process based on CPU scheduling algo like-
FCFS SJF RR Priority
frequency - Runs very frequently
Goal - Provide fast & efficient CPU allocation
Adv - Improves CPU utilization Reduces response time.

③ Medium Term scheduler (Swapper)
- medium term scheduler temporarily removes processes from memory (swapping) & later reintroduces them
function - manages swapping in/swapping out
Reduces degree of multiprogramming when sys is overhead
frequency - Runs occasionally, based on mem conditions

Quick Work

Page No.:

Date:

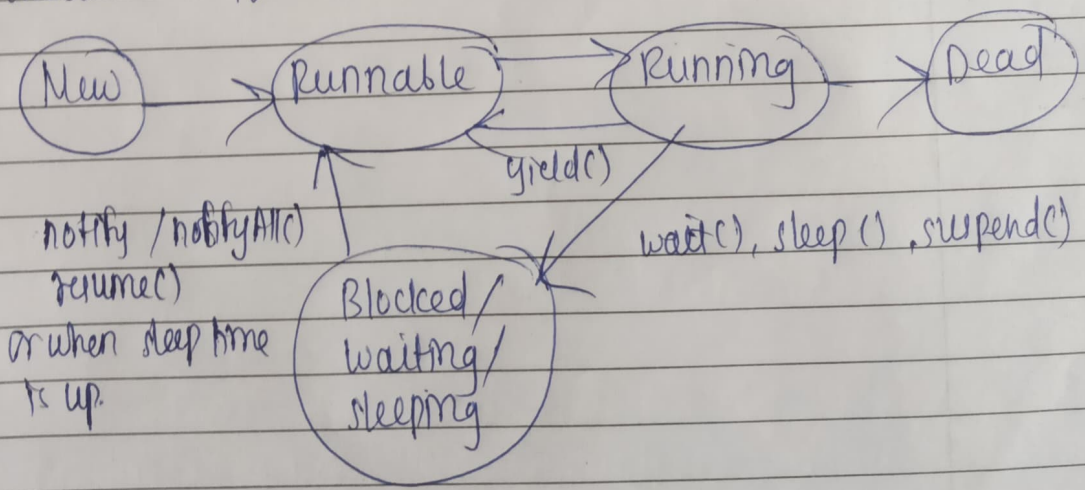
M T W T F S S

- Thread -
  - A thread is a smallest unit of execution within a process.
  - A process may contain multiple threads that share -
    - Code section
    - Data section
    - open files
    - Resources

  But each thread has its own -
    - Program counter
    - Stack
    - Register.

- In simple words, A thread is a lightweight process that allows multiple tasks to run within the same program simultaneously.

- Thread Lifecycle
  - A thread goes through several states from creation to termination



① New - Thread is created but not shared memory is allocated

② Runnable - After calling start(), it becomes ready to run. It may or may not be executing at that time waiting for turn from short term scheduler

Quick Work

Page No.:

Date:

M T W T F S S

(3) Running - Thread is assigned the CPU
executes its run() method

(4) Blocked / waiting - Thread is temporarily inactive
Happens when:
- waiting for I/O
- waiting for lock, resource
- sleep() or wait() method is called

(5) Timed waiting - Thread waits for a specific time
eg. sleep (1000ms)

(6) Terminated - Thread has completed execution
- Resources are released.