Operating Systems

↓

works as an interface between user & hardware

Applications
↕
Operating Systems
↕
Hardware

Eg → Windows
Mac OS
Linux
Android
Debian

Primary goal
↓ ↓
Convenience Throughput

Functionality of OS:

- Resource Management
- Process Management
- Storage Management
- Memory Management
- Security

Types of OS:

- Batch
- Multiprogrammed
- Multitasking
- Real time OS
- Distributed
- Clustered
- Embedded

Batch → Same jobs in the batch are executed at higher speed

↓

A process complete its execution, next job from job spool get executed without any user interactions.

Disadvantages:

- Difficult to Debug
- Costly
- If a job gets to enter in an infinite loop, other jobs wait for unknown time

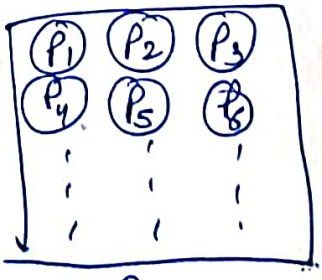
Non-preemptive

↓

Multiprogrammed OS

↓

More processes in RAM



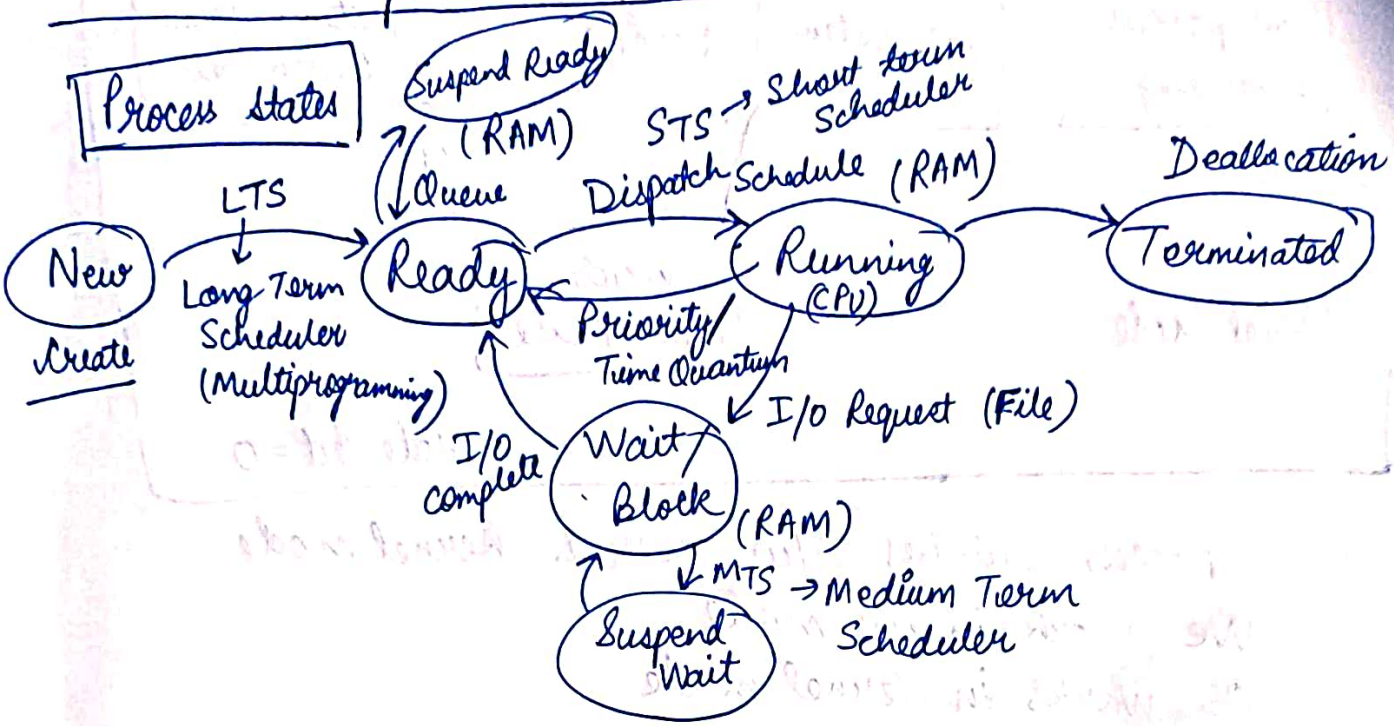
RAM

→ One process completes then only other process starts

Multitasking / Time Sharing OS

- Preemptive
- Responsiveness

Process states



Access of Hardware → Help of System call such as open(), read(), write(), close(), ioctl, etc..

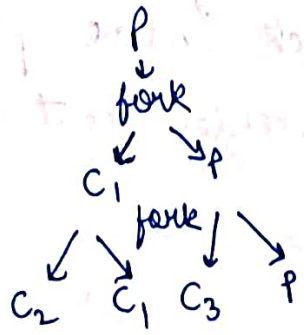
Fork()

↳ system call to create a child process

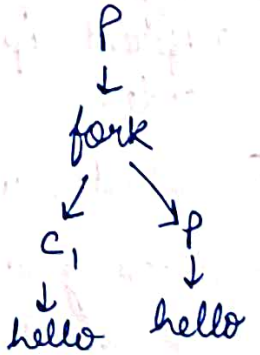
Fork() → 0 child
→ +1 Parent
→ -1 child x

```
main() {
fork();
fork();
printf("hello");
}
```

(4 times hello)



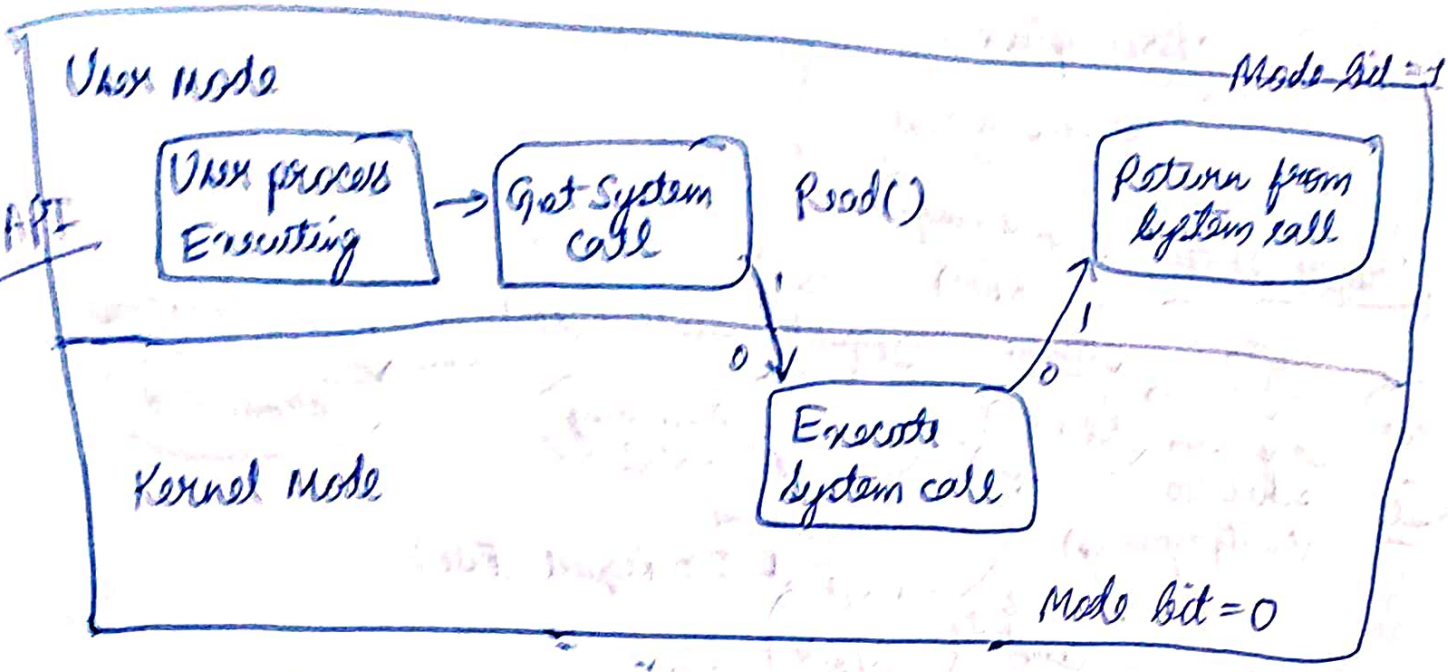

```
main() {
fork();
printf("hello");
}
```



n → times fork

Total child process ⇒ $2^n - 1$

User mode VS Kernel Mode



Process switches b/w user & kernel mode.
We work in user mode
OS works in Kernel mode

Process	Threads (User level)
① System calls involved in process	① No system call involved
② OS treats different processes differently	② All user level threads treated as single task for OS
③ Different process have different copies of Data/files/code.	③ Threads share same copy of code & data.
④ Context switching is slower	④ context switching is faster
⑤ Blocking a process will not block another.	⑤ Blocking a thread will block entire process.
⑥ Independent	⑥ Interdependent

User level Thread	Kernel level Thread
① Managed by user level library	① Managed by OS system calls.
② Typically fast	② Typically slower
③ Context switching is faster	③ Context switching is slower
④ If one user level thread blocked, then entire process ~~gets~~ get blocked	④ If one Kernel level thread blocked, No effect on others

Scheduling Algorithms → Way to select a process from ready queue and putting on running (CPU)

Preemptive

→ SRTF (Shortest Remaining time first)

→ LRTF (Longest Remaining Time first)

→ Round Robin

→ Priority based

Non Preemptive

→ FCFS (First come First serve)

→ SJF (Shortest Job First)

→ LJF (Longest Job First)

→ HRRN (Highest Response Ratio Next)

→ Multilevel Queue

CPU Scheduling

→ Arrival time : The time at which process enter the ready queue

→ Burst time : Time required by a process to get execute on CPU

→ Completion time : Time at which process complete its execution

→ Turn around time : Completion time — Arrival time

→ waiting time : Turn Around Time - Burst time

→ **Response time :**

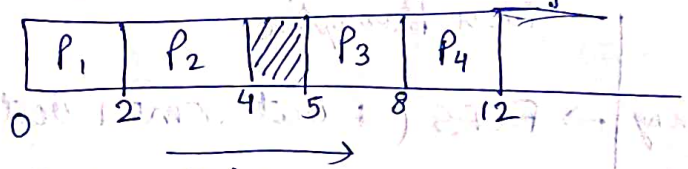
time at which process get CPU first time − Arrival time

---

① **FCFS** Mode → Non preemptive Criterial → Arrival Time

Process No.	Arrival time	Burst time	Completion Time	TAT	WT	RT
P_1	0	2	2	2	0	0
P_2	1	2	4	3	1	1
P_3	5	3	8	3	0	0
P_4	6	4	12	6	2	2

Gantt Chart

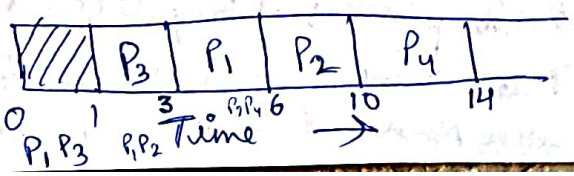


---

② **SJF** Mode → Non-preemptive Criteria → Burst time

Process No.	Arrival time	Burst time	completion time	TAT	WT	RT
~~P_1~~	1	3	6	5	2	2
~~P_2~~	2	4	10	8	4	4
~~P_3~~	1	2	3	2	0	0
~~P_4~~	4	4	14	10	6	6

Gantt Chart



③ SRTF Mode → Preemptive Criteria → Burst time

Process No.	Arrival time	Burst time	Completion time	TAT	WT	RT
P_1	0	~~5~~ 4	9	9	4	0
P_2	1	~~3~~ ~~2~~ 0	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	~~1~~ 0	5	1	0	0

Gantt Chart

P_1	P_2	P_2	P_2	P_4	P_1	P_3

0 1 2 3 4 5 9 13

P_2 P_3 P_4

④ Round Robin Mode → Preemptive Criteria → Time Quantum

Process No.	Arrival time	Burst time	Completion time	TAT	WT	RT
P_1	0	~~5~~ ~~3~~ ~~1~~ 0	12	12	7	0
P_2	1	~~4~~ ~~2~~ 0	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~ 0	9	5	4	4

Given TQ = 2

Ready Queue

~~P_1~~	~~P_2~~	~~P_3~~	~~P_1~~	~~P_4~~	~~P_2~~	P_1

0

Running Queue

P_1	P_2	P_3	P_1	P_4	P_2	P_1

0 2 4 6 8 9 11 12

P_2, P_3 P_4

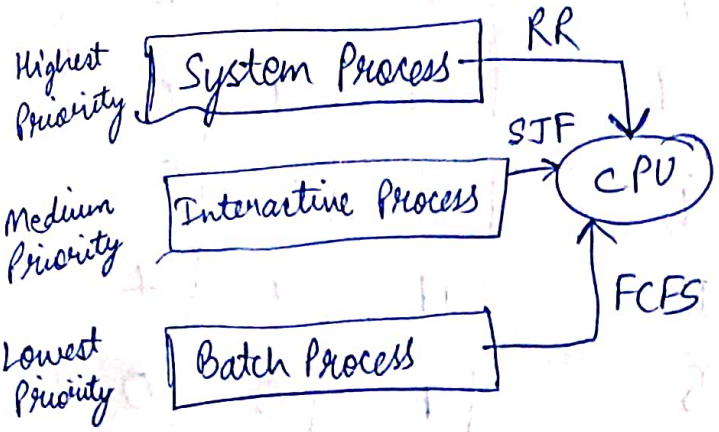
⑤ Priority Scheduling Mode → Preemptive Criteria → Priority

Priority	Process No.	Arrival time	Burst time	Completion time	~~TAT~~ TAT	WT
10	P_1	0	~~5~~ ~~4~~ 0	12	12	7
20	P_2	1	~~4~~ ~~3~~ 0	8	7	3
30	P_3	2	~~2~~ ~~1~~ 0	4	2	0
40	P_4	4	~~1~~ 0	5	1	0

High no. High priority

P_1	P_2	P_3	P_3	P_4	P_2	P_1

Multilevel Queue Scheduling



Multilevel Feedback Queue
↓
Used for removing starvation

Process Synchronization

Cooperative Process | Independent Process

Share
- Variable
- Memory
- Code
- Resources — CPU, Printer, Scanner

```
int shared=5;

P1                          P2
int x = shared;             int y = shared;
x++;                        y--;
sleep(1);                   sleep(1);
shared = x;                 shared = y;
```

Race Condition
↓
getting different values for same variable

Producer Consumer Problem

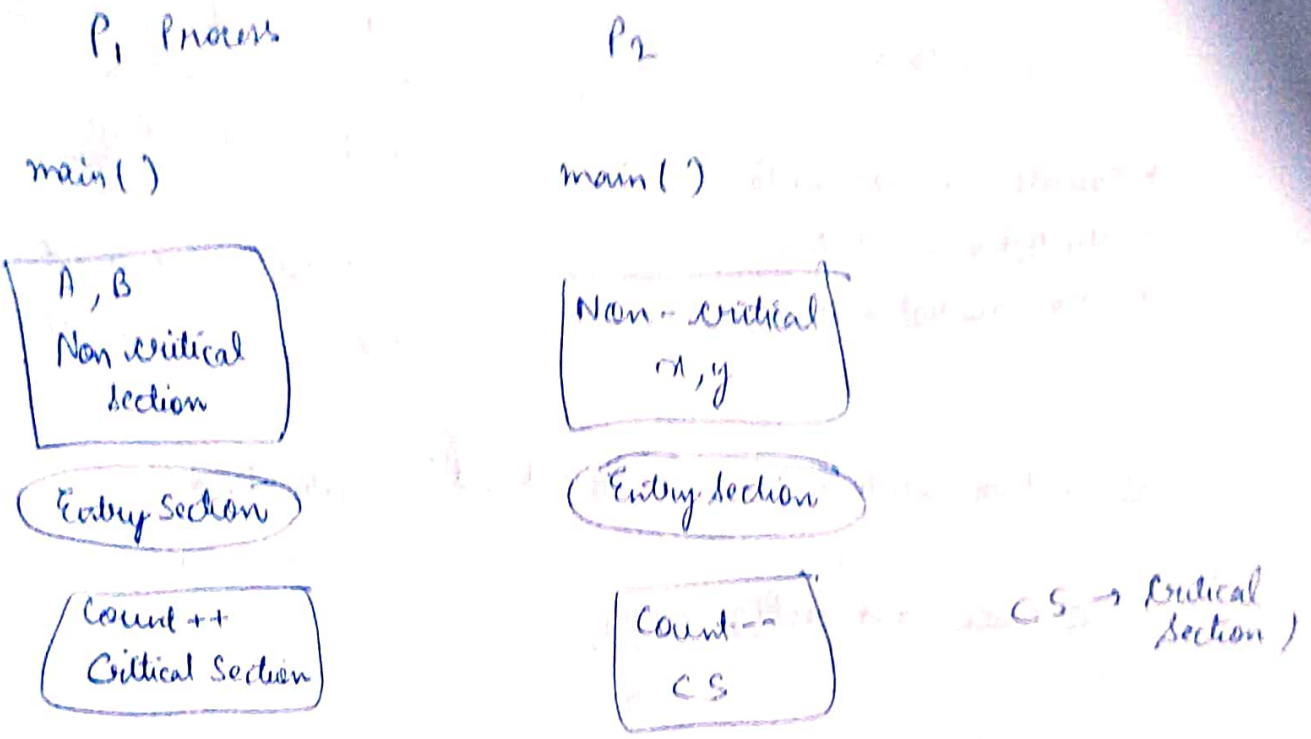
n = 8
Buffer[0...n-1]


```
void consumer(void)
{ int itemC;
  while(true)
  {
    while (count==0);
    itemC = Buffer[out];
    out = (out+1) mod n;
    count = count - 1;
    process_item(itemC);
  }
}
```

```
int count=0;
void Producer(void)
{ int itemp;
  while(true)
  { Produce_item(itemp);
    while (count==n);
    Buffer[in] = itemp;
    in = (in+1) mod n;
    count = count+1;
  }
}
```

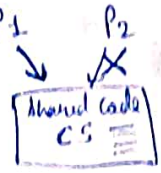
Buffer full

Critical Section → It is a part of the program where shared resources are accessed by various processes.



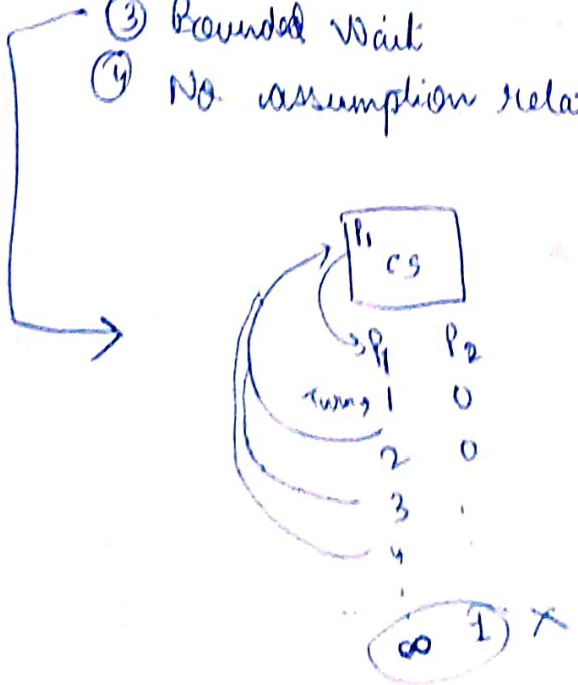
Synchronization Mechanism

4 conditions



1. Mutual Exclusion →
2. Progress → (P_2 is not allowing P_1 to enter CS (or vice versa)

3. Bounded Wait
4. No assumption related to Hardware



(Starvation)

Critical Section Solution using "Lock"

```
do {
    Acquire lock
        CS
    release lock
}
```

1. while (LOCK == 1); — Entry code
2. LOCK = 1
3. Critical section
4. LOCK = 0 — Exit code

* Execute in user mode
* Multiprocess Solution
* No mutual exclusion guarantee

lock = 0 → Vacant
= 1 → Full

Critical Section Solution using "Test_and_Set" Instruction

```
while ( test_and_set(& lock));
[CS]
lock = false;
```

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Turn Variable

* 2 process solution
* Run in user mode

Semaphore

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronization

- Counting ($-\infty$ to $+\infty$)
- Binary (0, 1)

Entry Section Code

```
Down (Semaphore S)
{
    S.value = S.value - 1;
    if (S.value < 0)
    {
        put process (PCB) in
        suspended list sleep();
    }
    else
        return;
}
```

Exit Section Code

```
UP (Semaphore S)
{
    S.value = S.value + 1;
    if (S.value ≤ 0)
    {
        Select a Process
        from suspended list
        wake up();
    }
}
```

P(), Down, wait
V(), Up, Signal, Post, Release

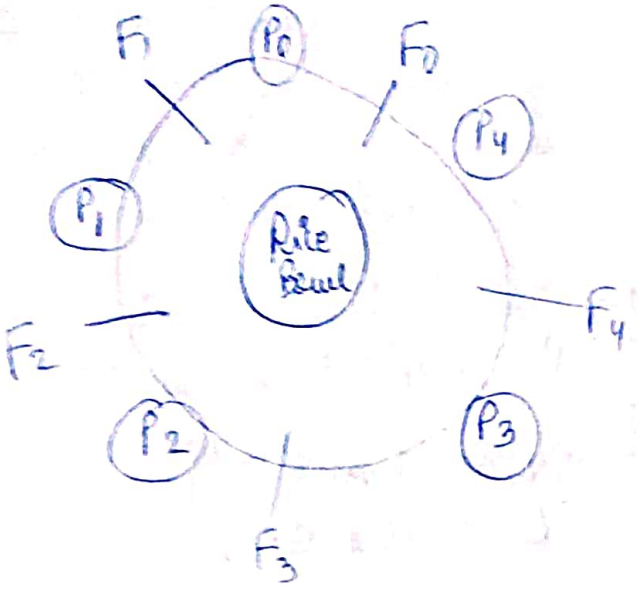
→ These operations used in entry code

Reader - writer Problem

Same data

- R - W → Problem
- W - R → Problem
- W - W → Problem
- R - R → No Problem

Dining Philosophers Problem



P → Philosophers
F → Fork

N → No. of forks

```
void philosopher(void)
{
  while(true)
  {
    Thinking();
    take_fork(i);          ← left fork
    take_fork((i+1)%N);    ← right fork
    EAT();
    Put_fork(i);
    Put_fork((i+1)%N);
  }
}
```

Entry

Race Condition

↓

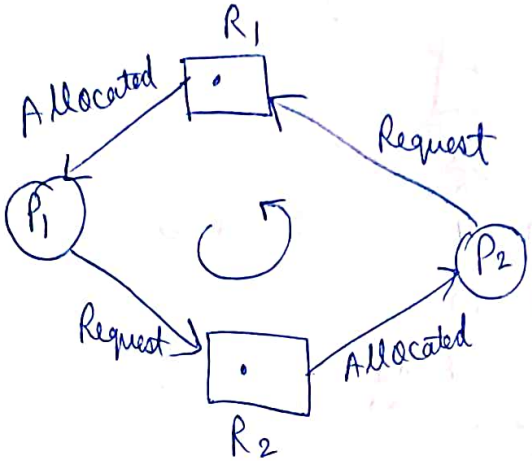
Solution → Binary Semaphore

Deadlock occurs in this.

For last philosopher, reverse the taking of fork first right and then left.

Deadlock

if two or more processes are waiting on happening of some event, which never happens then we say these processes are involved in deadlock then that state is called deadlock.

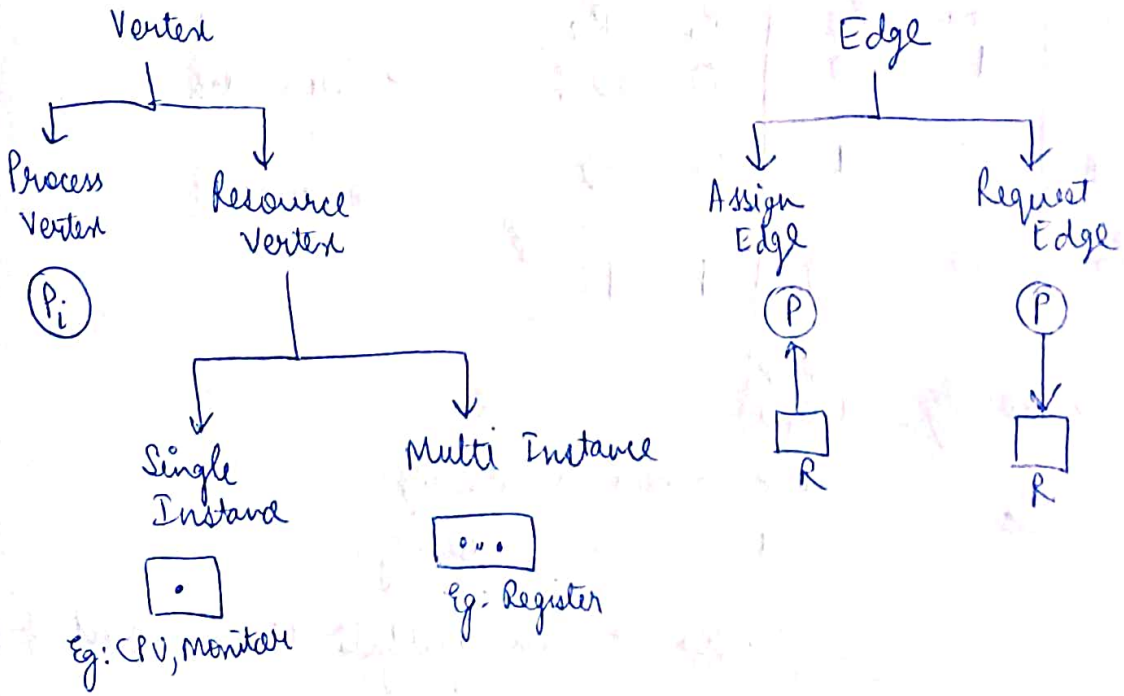


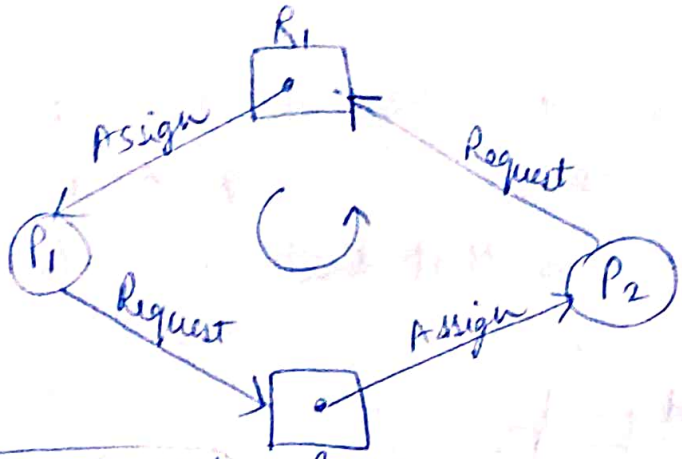
Necessary conditions

Conditions for deadlock:

1. Mutual exclusion
2. No preemption
3. Hold & wait
4. Circular wait

Resource Allocation Graph (RAG)



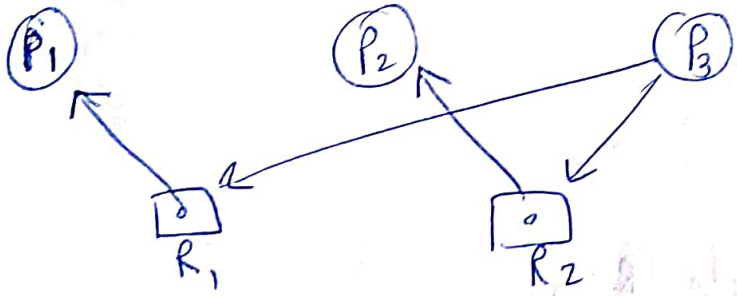


	Allocated		Request	
	R_1	R_2	R_1	R_2
P_1	1	0	0	1
P_2	0	1	1	0

Availability $(0, 0)$ (R_1, R_2)

↳ Cannot fulfill the request

↳ There is a deadlock in this



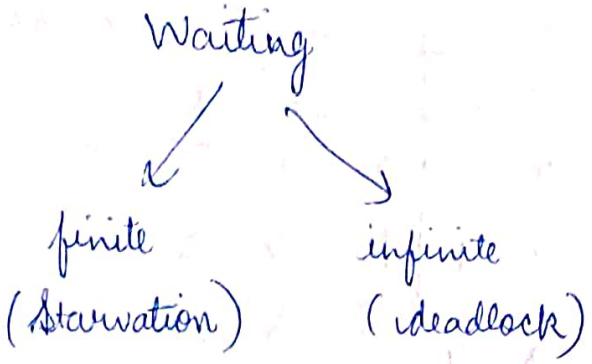
	Allocate		Request		
	R_1	R_2	R_1	R_2	
P_1	1	0	0	0	→ terminated
P_2	0	1	0	0	→ terminated
P_3	0	0	1	1	

Availability $(0,0)$ (P_1, P_2)

1,0 → new availability

1,1 → new availability.

↳ now it can fulfill the request of P_3.



if RAG has circular wait (cycle)
⇒ Always deadlock

→ Only in single instance resource

if RAG has no cycle then no deadlock

Methods to handle deadlocks

1. Deadlock ignorance (ostrich method)
2. Deadlock prevention
3. Deadlock Avoidance (Banker's Algorithm)
4. Deadlock detection and recovery

Banker's Algo

Total A=10, B=5, C=7

Process	Allocation			Max need			Current Available			Remaining Need (Max − Allocation)			
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	0	1	0	7	5	3	3 (+2)	3 (+0)	2 (+0)	7	4	3	✓
P_2	2	0	0	3	2	2	5 (+2)	3 (+1)	2 (+1)	1	2	2	✓
P_3	3	0	2	9	0	2	7 (+0)	4 (+0)	3 (+2)	6	0	0	✓
P_4	2	1	1	4	2	2	7 (+0)	4 (+1)	5 (+0)	2	1	1	✓
P_5	0	0	2	5	3	3	7 (+3)	5 (+0)	5 (+2)	5	3	1	✓
	7	2	5				10	5	7				

Sequence $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

Process	Allocation			Max Need			Available			Remaining (Max − Allocn)		
	E	F	G	E	F	G	E	F	G	E	F	G
P_0	1	0	1	4	3	1	3	3	0	3	3	0 ✓
P_1	1	1	2	2	1	4	4	3	1	1	0	2 ✓
P_2	1	0	3	1	3	3	5	3	4	0	3	0 ✓
P_3	2	0	0	5	4	1	6	4	6	3	4	1 ✓
	5	1	6				8	4	6			

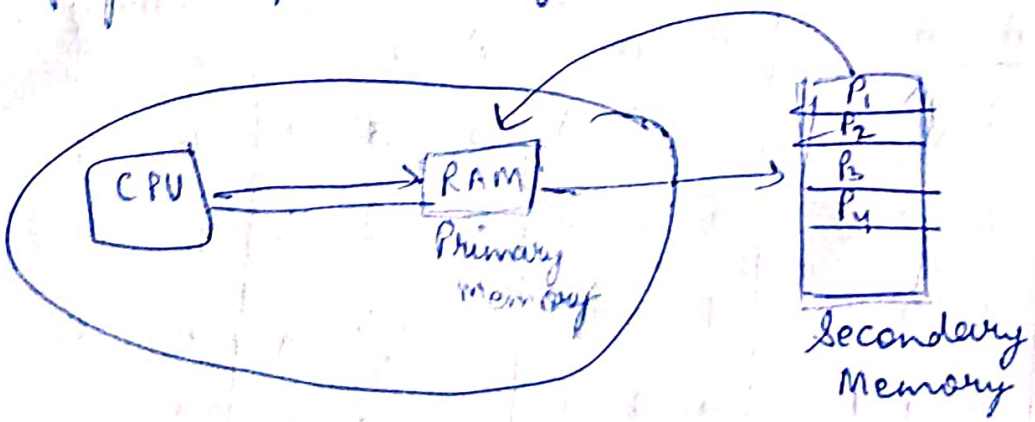
Total
E (8)
F (4)
G (6)

Sequence: $P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3$

Memory management → method of managing primary memory.

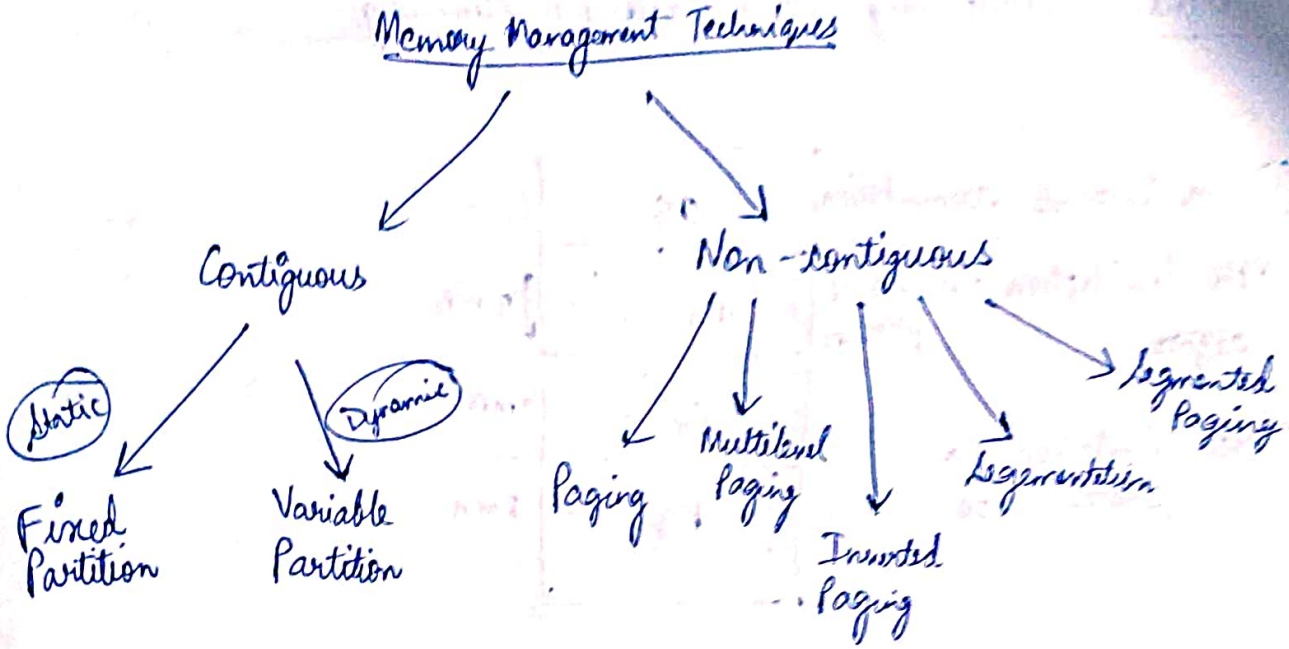
Goal: Efficient utilization of memory

Multiprogramming → Bringing more processes to RAM



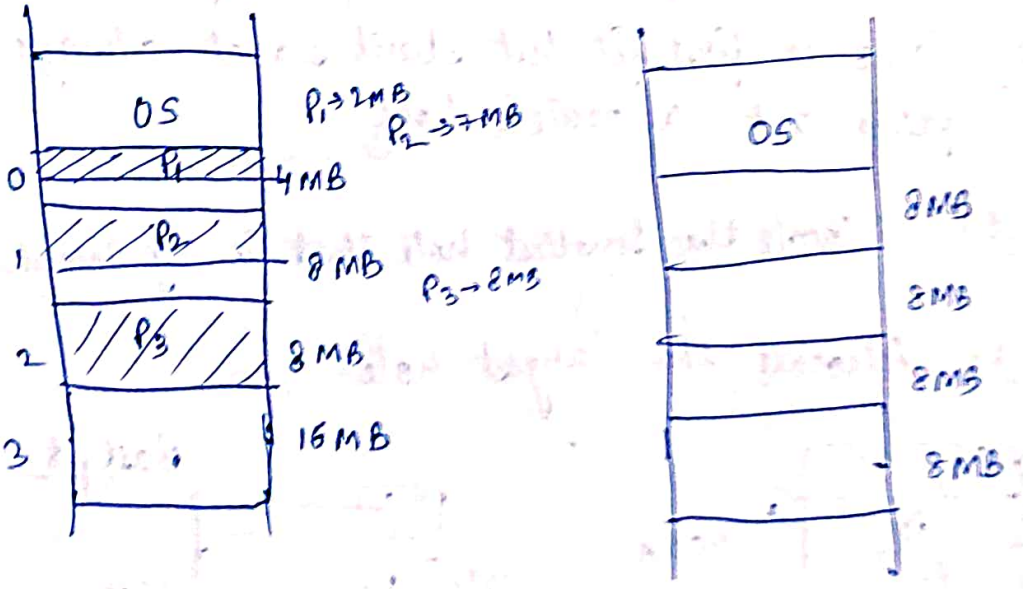
K = I/O operations

CPU Utilization = $1 - K^n$ n → no. of processes



Fixed partitioning (Static Partition)

- No. of partitions
- Size of each partition may or may not same
- Contiguous allocation so spanning is not allowed



Disadvantages:

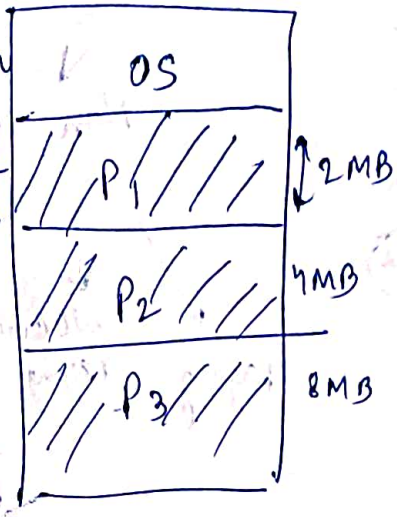
1. Internal fragmentation (space is left in b/w)
2. Limit in process size.
3. Limitation on degree of multiprogramming
4. External fragmentation.

Dynamic Partitioning or Variable partitioning

(1) No internal fragmentation

(2) No limitation on no. of processes

(3) No limitation on process size



Disadvantages:

(1) External fragmentation (space available but processes cannot be put due to contiguous memory)

(2) Allocation / Deallocation is complex

First Fit: Allocate the first hole that is big enough

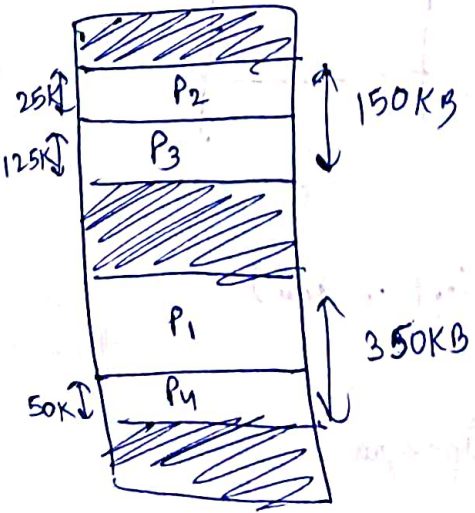
Next Fit: Same as first fit but start search always from last allocated hole

Best Fit: Allocate the smallest hole that is big enough.

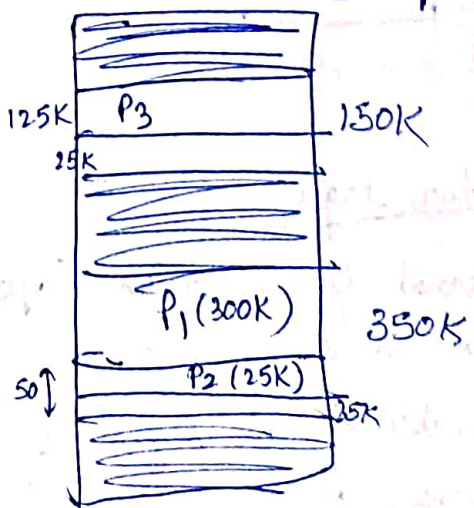
Next Fit: Allocate the largest hole.

First Fit

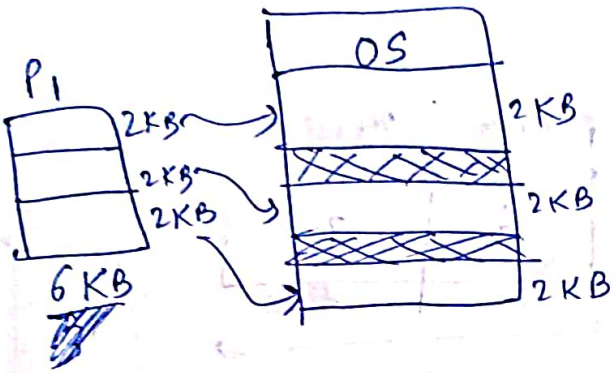
$P_1 \rightarrow 300K$
$P_2 \rightarrow 25K$
$P_3 \rightarrow 125K$
$P_4 \rightarrow 50K$

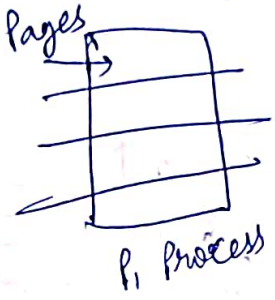


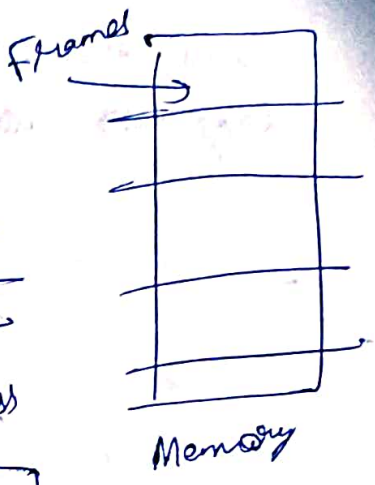
Best fit



Non-Contiguous Memory Allocation

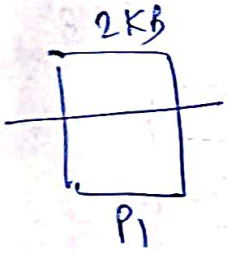


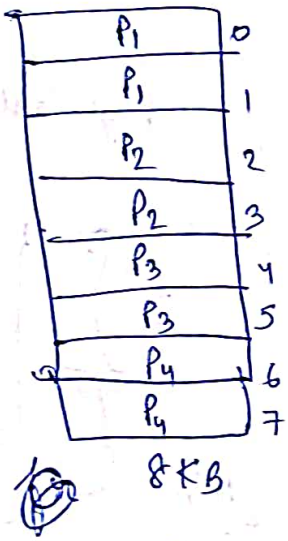




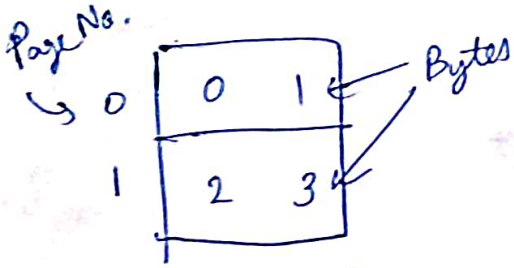
Paging

Page size = Frame size





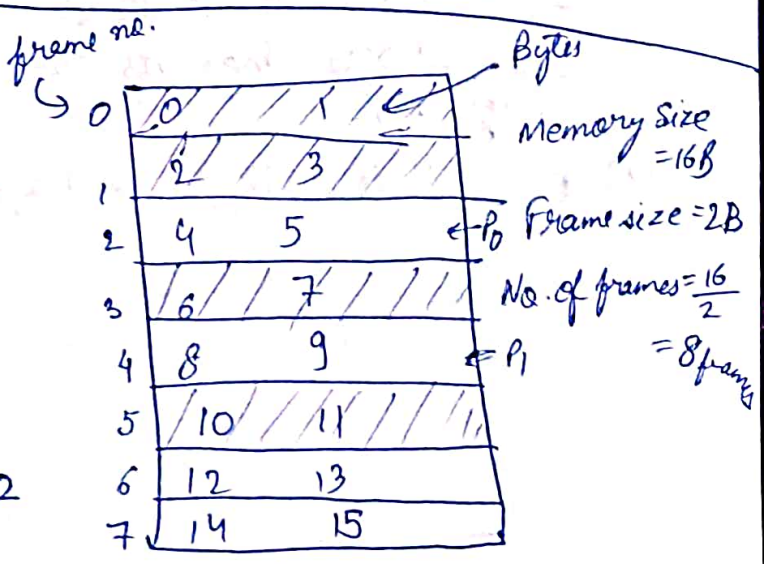
Page size = frame size = 1KB

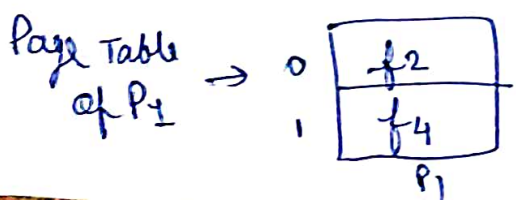


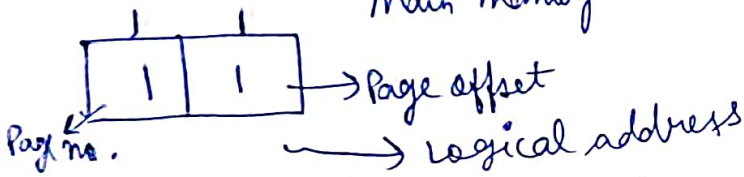
Process size = 4B

Page size = 2B

No. of pages / process = $\frac{4B}{2B} = 2$





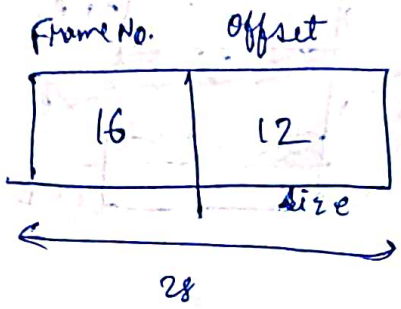


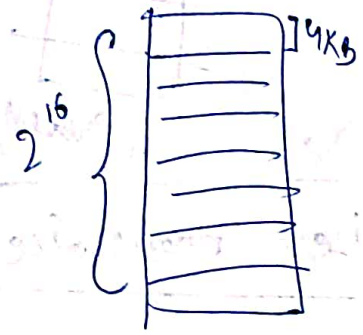
Physical address space = 256 MB = 2^{28} B

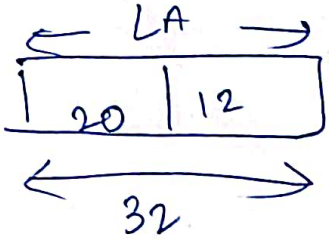
Logical address space = 4 GB = 2^{32} B

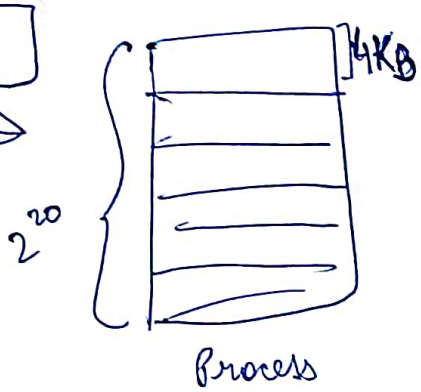
Frame size = 4 KB = 2^{12} B

Page table entry = 2 B



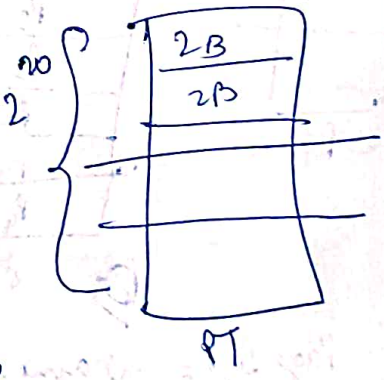






Page Table size = $2^{20} \times 2$ B

= 2 MB

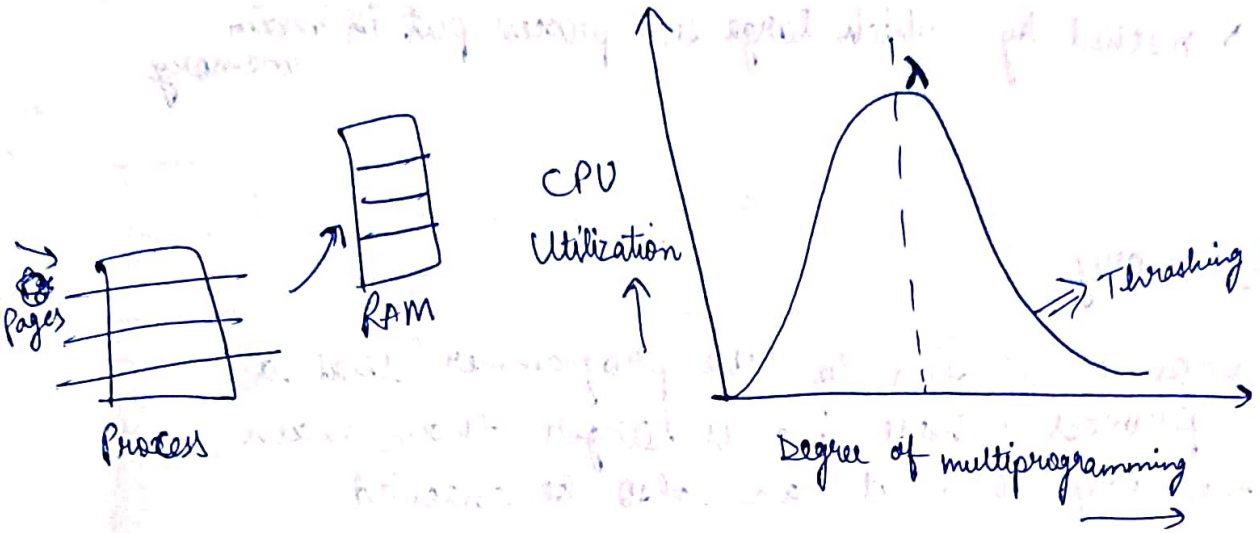


$\frac{2\text{ MB}}{4\text{KB}} = 2^9$

Disadvantages of Paging:

1. Each process has its own page table
2. Page table will be in main memory

Thrashing



Page fault → Pages not present in RAM

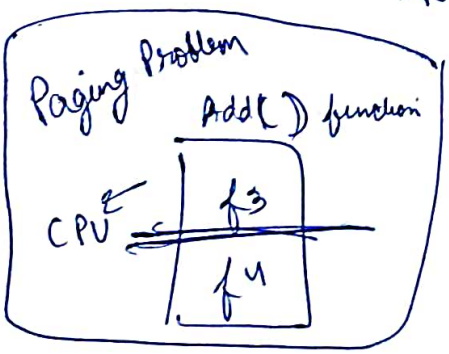
Thrashing occurs when page faults and switching occur at a higher frequency, requiring the OS to spend more time exchanging these pages. Thrashing slows down an OS as the OS is not doing any productive work during thrashing.

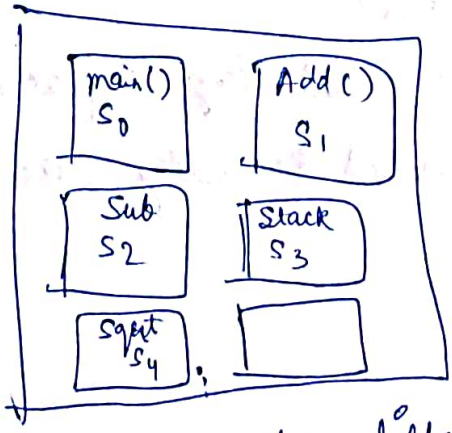
Improvements:
1) Main memory size increases
2) Long term Scheduler

Segmentation

↳ Process divided into parts before putting in memory.

→ works on user point of view.





→ Sizes of segments differ

Overlay

↳ method by which large size process put in main memory.

Virtual memory

↳ provides illusion to the programmer that a process whose size is larger than main memory then it can also be executed.

$$\text{Effective memory access time} = P\left(\text{page fault service time}\right) + (1-P)\left(\text{main memory access time}\right)$$

ma

↓ page fault

↓ milliseconds

↓ n sec

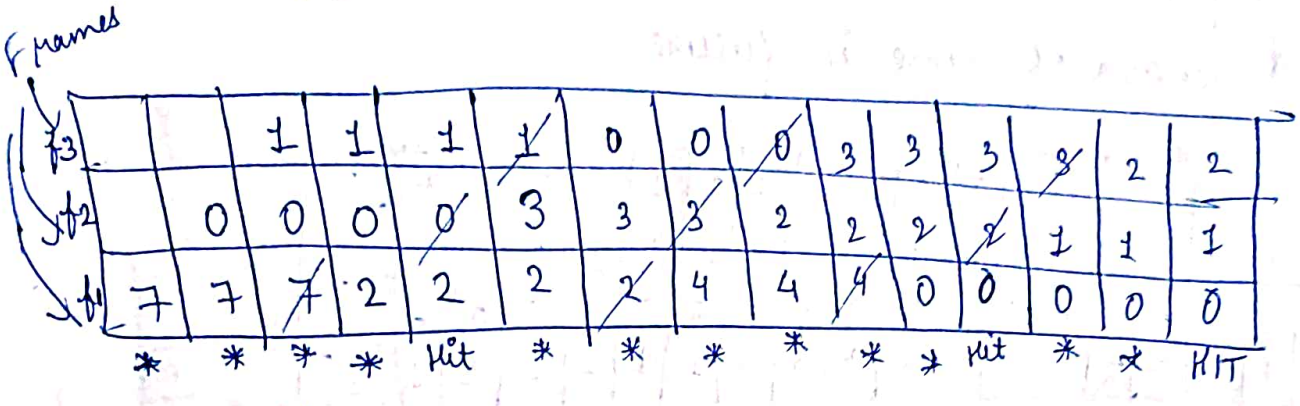
~~Trans~~ Translation Lookaside Buffer → Cache

$$EMAT = (TLB + m).Hit + (TLB + m + n).Miss$$

Page Replacement Algorithm

- FIFO
- Optimal Page Replacement
- Least Recently used (LRU)

I FIFO (First in First out)



Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0 →

(Page nos.)

$$\text{Hit Ratio} = \frac{\text{no. of hits}}{\text{no. of Reference}} = \frac{3}{15} \times 100 \simeq 20\%$$

$$\text{Page fault Ratio} = \frac{12}{15} \times 100 \simeq 80\%$$

Belady's Anomaly → On increasing frames, page faults are increases.

II Optimal Page Replacement

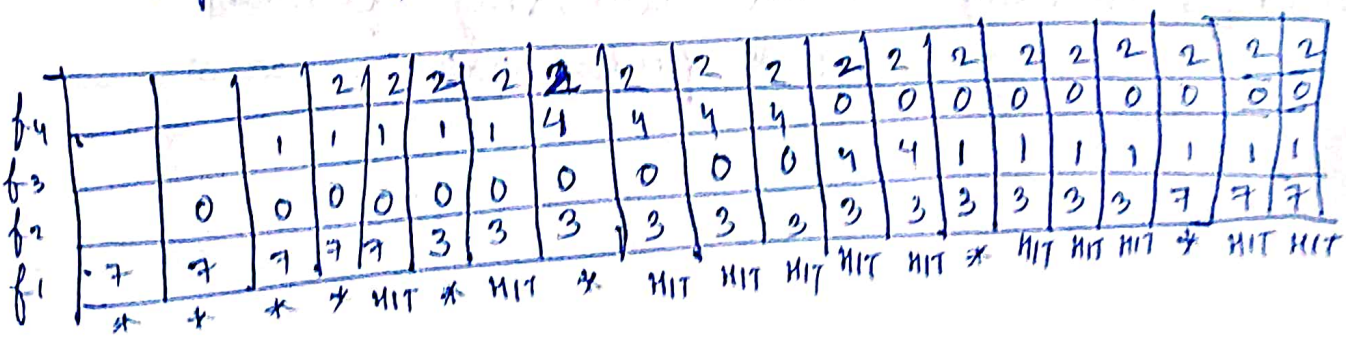
↳ Replace the page which is not used in longest dimension of time in future

Ref. string → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Page Hit = 12
Page fault = 8

III Least Recently Used

Replace the least recently used page in past



Ref. string → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Page Hit = 12
Page fault = 8

IV Most Recently Used

Replace the most recently used page in past

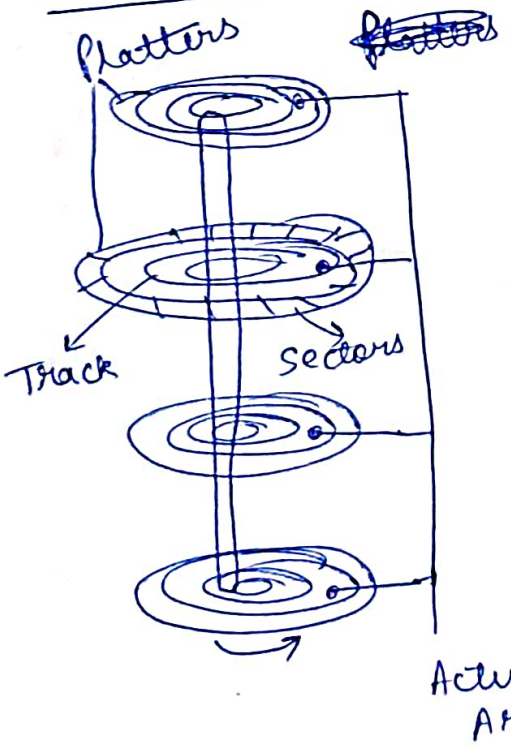
f_4				2	2	2	2	2	2	3	0	3	2	2	2	0	0	0	0	0
f_3			1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
f_2		0	0	0	0	3	0	4	4	4	4	4	4	4	4	4	4	4	4
f_1	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	*	*	*	*	HIT	*	*	*	HIT	*	*	*	*	HIT	HIT	*	HIT	HIT	HIT	HIT

Ref. string → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Page HIT = 8

Page fault = 12

Disk Architecture



Platters → Surface (upper & lower) → Track → Sectors → Data

Disk size = P x S x T x S x D

Seek time : Time Taken by read/write head to reach desired track.

Rotation time : Time taken for one full rotation (360°)

Rotational latency : Time taken to reach to desired sector. (half of rotation time)

Transfer time : $\frac{\text{Data to be transfer}}{\text{Transfer Rate}}$

Transfer Rate (Data rate) : {no. of heads × Capacity of one track × no. of rotations in one second}

Disk Scheduling Algorithms

Goal: To minimize the seek time

- FCFS (first come first serve)
- SSTF (shortest seek time first)
- SCAN
- LOOK
- CSCAN (Circular Scan)
- CLOOK (Circular Look)

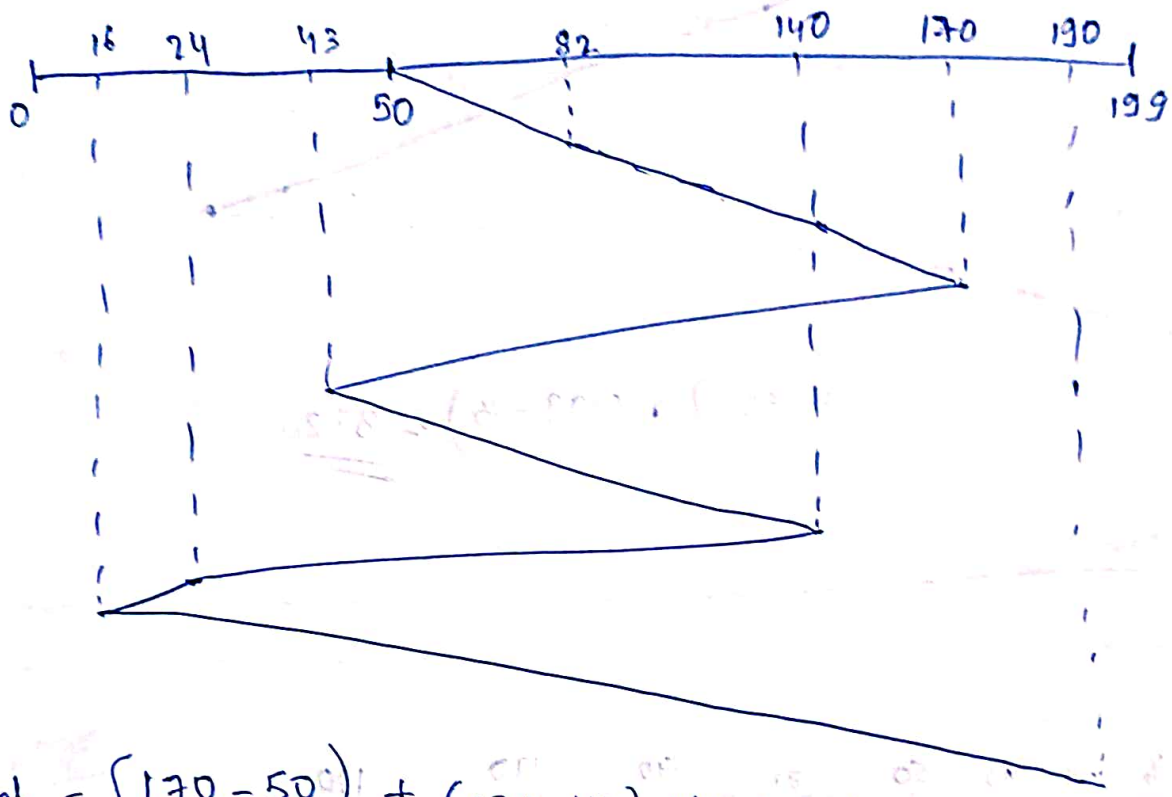
FCFS

Total tracks = 200

Request queue contain track no. 82, 170, 43, 140, 24, 16, 190

Current position = 50

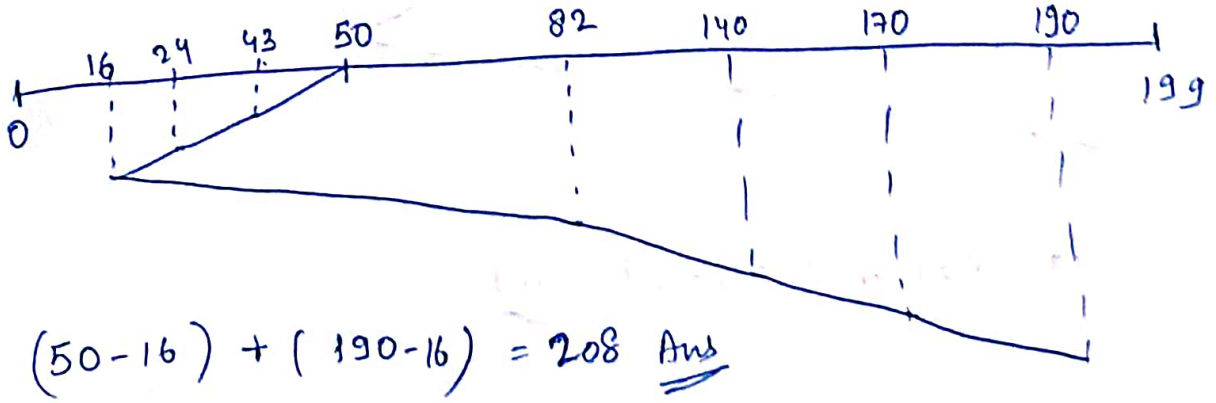
Calculate total no. of tracks movement by head?



Total movements = $(170-50) + (170-43) + (140-43) + (140-16) + (190-16)$

SSTF

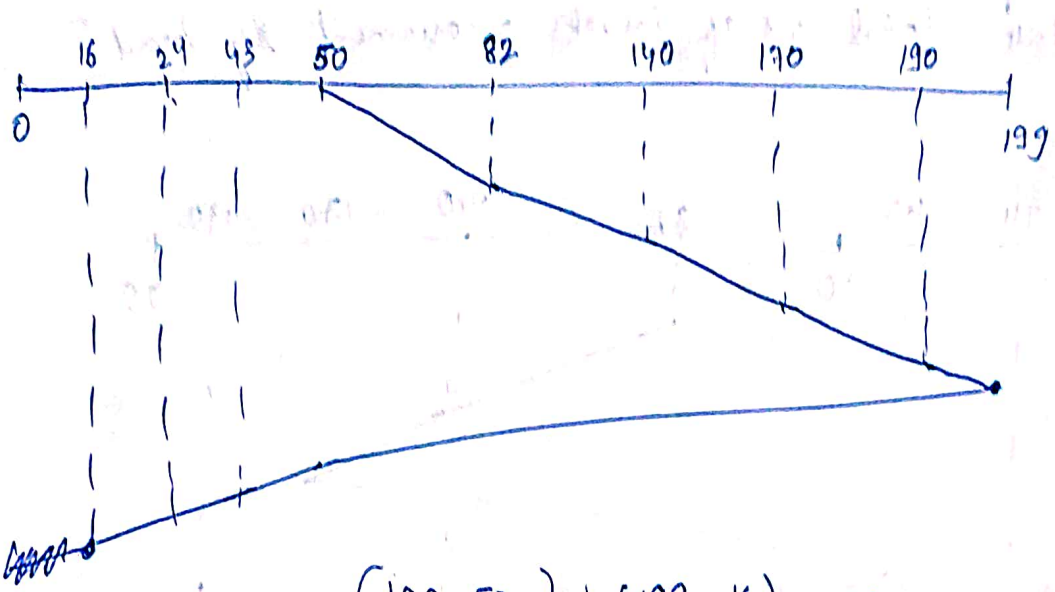
82, 170, 43, 140, 24, 16, 190 curren = 50



$(50-16) + (190-16) = 208$ Ans

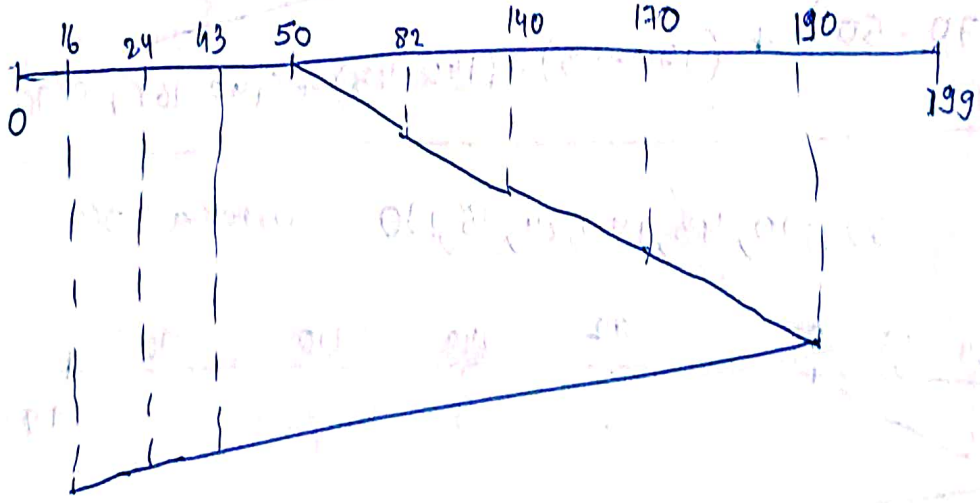
SCAN

82, 170, 43, 140, 24, 16, 190



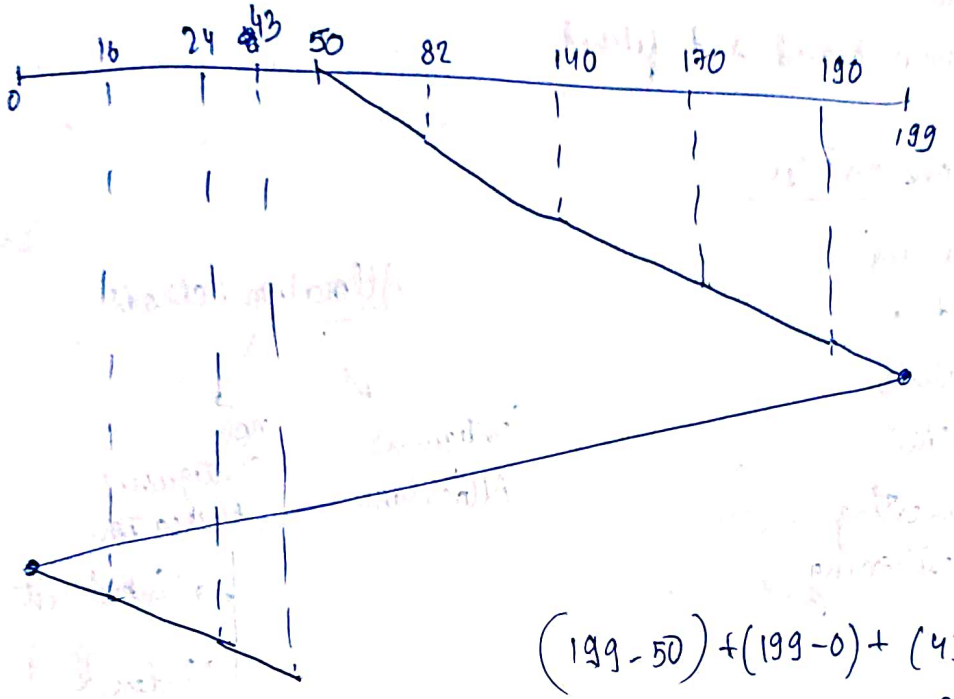
$(199-50) + (199-16) = 332$

LOOK



$(190-50) + (190-16) = 314$

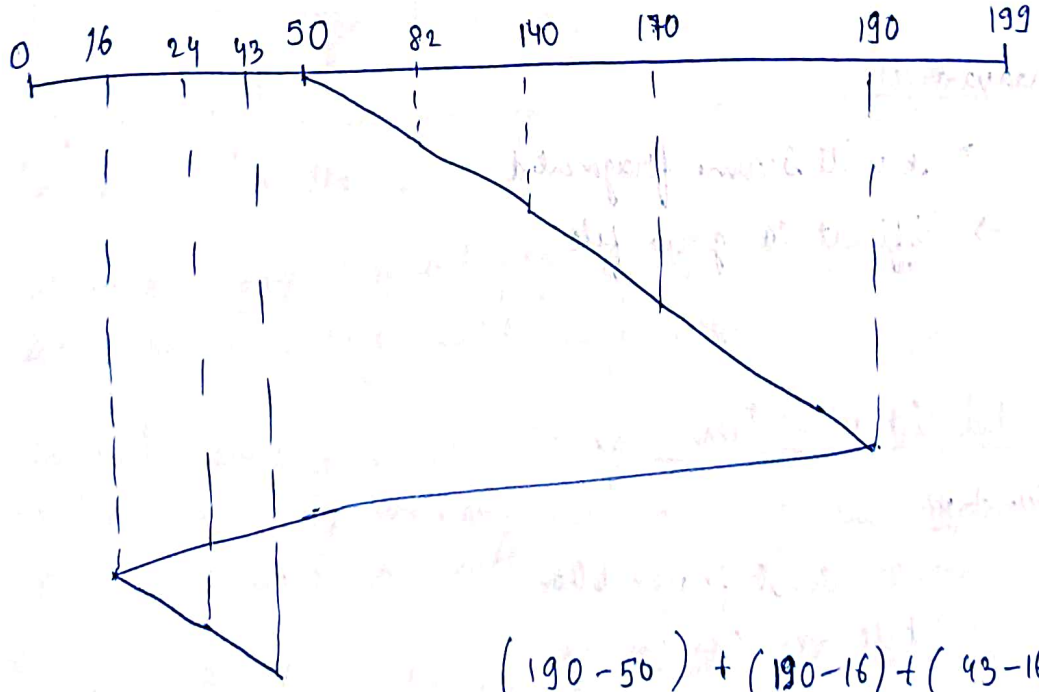
C-SCAN



$(199-50)+(199-0)+(43-0)$

$= 391$

C-LOOK



$(190-50)+(190-16)+(43-16)$

$= 341$

File System in OS

↓

Software

How stored and fetched

Operations on Files

1. Creating
2. Reading
3. Writing
4. Deleting
5. Truncating
6. Repositioning

Allocation Methods

- Contiguous Allocation
- non-Contiguous Allocation
  - Linked List
  - Indexed

Contiguous Allocation

Advantages

- Easy to implement
- Excellent Read performance

Disadvantages

- Disk will become fragmented
- Difficult to grow file

Linked list Allocation

Advantages

- No external fragmentation
- File size can increase

Disadvantages

- Large seek time
- Random access / direct access difficult
- Overhead of pointers

Indexed Allocation

Advantages

↳ Support direct access
↳ External fragmentation

Disadvantages

↳ Pointer overhead
↳ Multilevel index

→ Process control Block (PCB) is a data structure used by Computer OS to store all the information about a process

→ Process states : new, ready, running, wait/Block, suspend, terminated
Thread has 3 states : Running, Ready, Blocked

→ Mutex is a locking mechanism while a semaphore is a Signaling mechanism

→ Threads are segment of a process

→ System call is a way for a user program to interface with the OS

→ Demand Paging keep all pages of the frames in the secondary memory until they are required.

→ Virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were a part of the main memory.

→ In paging, the program is divided into fixed or mounted size pages.

→ In segmentation, the program is divided into variable size sections.

→ In multiprogramming, multiple programs execute at a same time on a single device.

→ In multitasking, a single resource is used to process multiple tasks

→ Internal fragmentation V/S External fragmentation