Operating System → work as interface

Hide the complexity

- Resource Mangement

(1) Resource Mgmt

(2) process Mgmt (CPU scheduling)

A software, it works as interface b/w user and hardware.

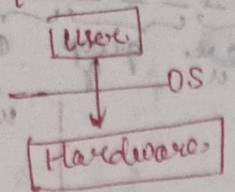
Bulky security & privacy

Isolation & protection

Storage mag → file system

made up of Collection of system software

Memory mgmt (RAM)



Type of OS

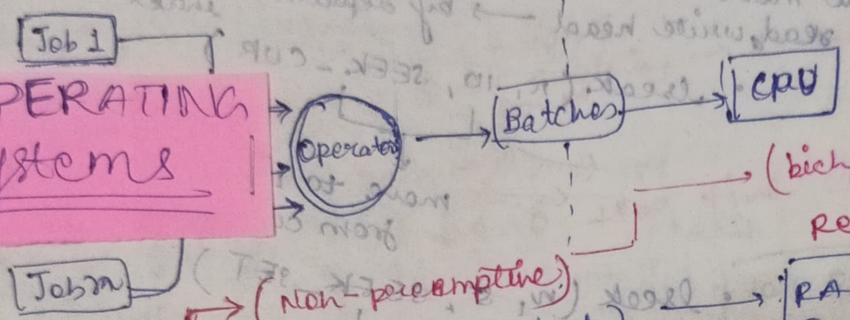
(1) Single Process OS → maximum CPU utilization X

eg: • MS DOS

Starvation X

High Priority job X

(2) Batch Processing OS → Punch cards → scan



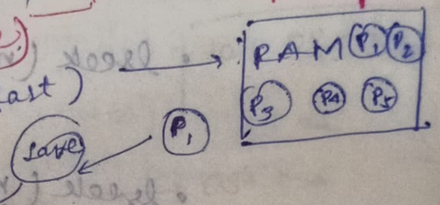
OPERATING Systems

(bich me se koe process jump kr to)

Ready Queue

(Non-preemptive)

(came to fast)



(3) Multiprograming OS

↳ Idleness

PCB → Process Control Block

context switching

Preemptive (using priority)

(4) Multitasking OS

• Single CPU (Max utilisation)

• context switching

Time sharing

Responsive • Time sharing

(5) Real Time OS

→ hard Missile system

→ soft video game

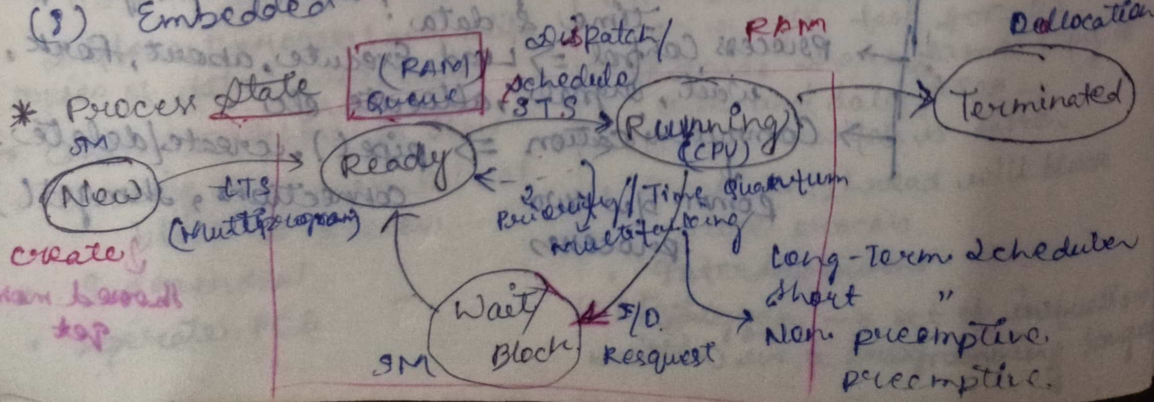
↳ No delays software → CPU should give a response in real time

(6) Distributed → enviorment → distributed → availability max.

(7) Cluster

(8) Embedded → fixed OS

* Process State



Suspend Ready

MTS

Suspend Wait SM

* chmod → change mode

r w x (user, u) r w x (group, g) r w x (other, o)

r → read, w → write, x → execute

user	u	r 4
group	g	w 2
other	o	x 1 (↳ execute)

ugo + rw → 4 + 2 = 6

chmod 666 hello

* lseek → to move read, write head → by default at 0 index → 1st position

Ex: lseek (n, 10, SEEK_CUR) → move to 10th pos. from curr pos.

• lseek (n, 5, SEEK_SET) → set at 5th posi.

• lseek (n, 5, SEEK_END) → move to 5th posi. from end pos.

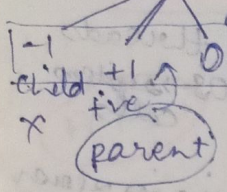
* we write anything in user mode
ex access in kernel mode → Operating System

* system call → invoke kernel mode to do some work → to access OS function

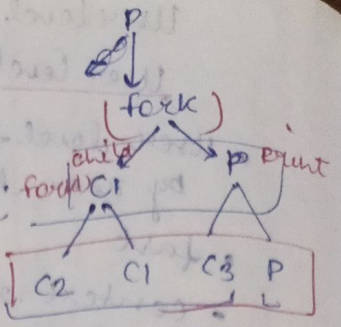
- File Related ⇒ open(), Read(), write(), close(), create file etc.
- Device Related ⇒ Read, write, Reposition, ioctl, fcntl etc. (↳ hard disk)
- Information ⇒ get Pid, attributes, get system time & data.
- Process Control ⇒ Load, Execute, abort, Fork, wait, signal, allocate etc.
- Communication ⇒ pipe(), create/delete connections, shmget() (↳ comm. b/w two processes)

System call.

Fork() → to create a child process



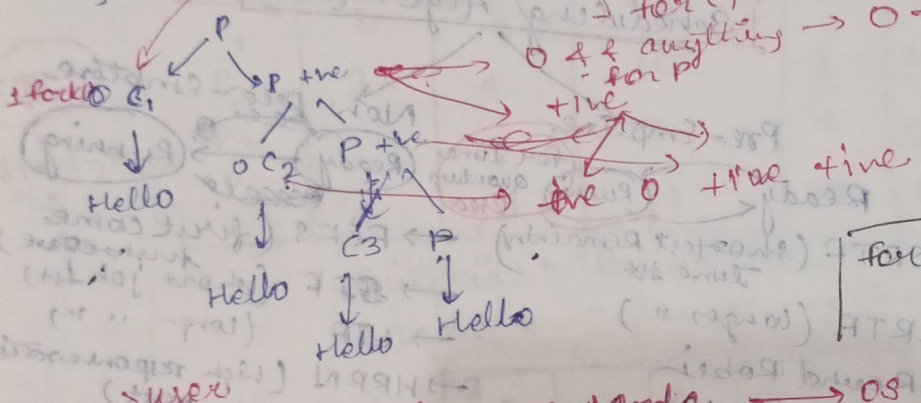

```
main() {
fork()
fork()
printf("hello");
}
```



- hello → 4 = 2^n → no. of fork
- $2^n - 1$ → no. of child process

Thread

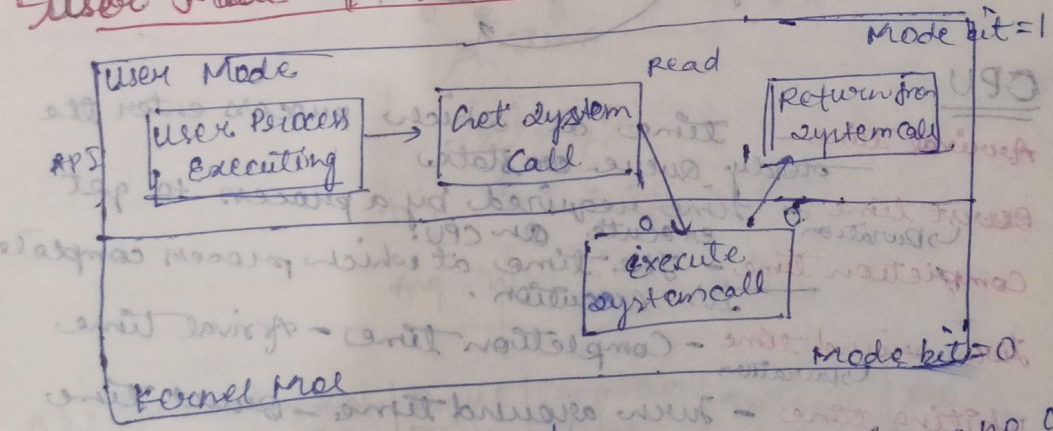
```
main() {
if (fork() && fork()) {
fork();
printf("Hello");
return 0;
}
```



for OR → (7)

→ user → OS

User Mode V/s Kernel Mode



(1) Process → using kernel OS
- System calls involved in process
- OS treats diff process differently
- Different process have diff copies of data, files, code → $2^n - 1$
- Context switching is slow
- Blocking a process will not block a process
- Independent
- Seperate PCB

(2) Threads → no OS using Application program
- User level
- No system call involved
- All user level threads treated as single task for OS
- Threads share same copy of code and data → 2^n
- Context switch is faster
- Block a thread will block entire process
- Interdependent

(3) Program → set of instructions written in a programming langu

User level vs Kernel Level Thread

User level thread

- User level threads are managed by user level library.
- fast
- context switching is fast
- If one user level threads performs blocking operation then entire process get blocked.

Kernel level thread (OS)

- Kernel level threads are managed by OS → system calls
- Slower
- context switching is slower
- If one kernel level thread blocked, no affect on others.

Processes → Ready Queue → Scheduling Algorithm → CPU (Running state)

Scheduling Algorithms

Pre-Emptive
- Ready → Running (for Time Quantum / Preemption)
  - SRTF (Shortest Remaining Time first)
  - LRTF (longest " ")
  - Round Robin
  - Priority based

Now Pre-Emptive
- Ready → Running
  - FCFS (first come first serve)
  - SJF (shortest job first)
  - LJF (longest " ")
  - HRRN (High response ratio next)
  - Multi-level Queue

CPU

- **Arrival time** – time at which process enter the ready queue or state.
- **Burst time** (Duration) – time required by a process to get execute on CPU.
- **Completion time** – The time at which process complete its execution.
- **Turn around time** (Duration) – Completion time – Arrival time
- **Waiting time** (Duration) – Turn around time – Burst time
- **Response time** (Time) – (Time at which a process get CPU first time) – (Arrival time)

Q1. (given)

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

FCFS

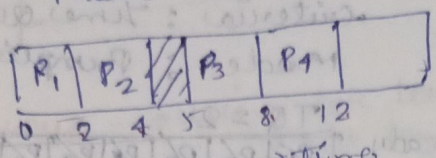
Starvation

Criteria = "Arrival time"

Mode = Non-preemptive

Gantt Chart

Gantt Chart / Running Queue



Q2. (given)

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

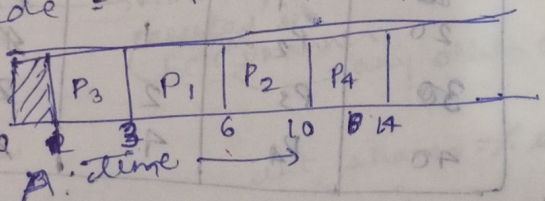
SJF

Starvation

Criteria = "Burst time"

Mode = "Non-preemptive"

Gantt chart



P1 P3 P2 P4

Avg TAT = $\frac{25}{4}$ = 6.25

Avg WT = $\frac{12}{4}$ = 3

Q3.

Process No.	Arrival Time	Burst time	Completion time	TAT	WT	RT
P_1	0	~~8 4 2 1 X~~ 6	9	9	4	0
P_2	1	~~3 2 1 X~~ 0	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	~~2~~ 1 0	5	1	0	0

Criteria = "Burst time"

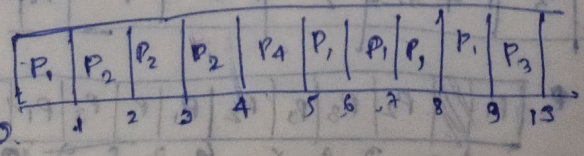
Mode = "Preemptive"

SRTF

↓

SJF with Preemption

Gantt Chart



P1 P2 P3 P4

P1 P3 P2

Q4. "Quantum time"

Round Robin

Process No.	Arrival time	Burst time	Completion time	TAT	WT	RT
P_1	0	~~8~~ ~~4~~ 1	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

Criteria : Time Quantum
Mode : Preemptive

Sequence of Process

TQ = 2

Ready Queue: ~~P_1~~ P_2 P_3 ~~P_4~~ ~~P_1~~ P_2 P_1

Running Queue: P_1 P_2 P_3 ~~P_4~~ P_1 P_4 P_2 P_1

0 2 4 6 8 9 11 12

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

P_1 P_2

Q5.

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

Priority Scheduling

Higher the no. → higher priority

Criteria = "Priority"
Mode = "Preemptive"

P_1	P_2	P_3	P_3	P_4	P_2	P_1

0 1 2 3 4 5 8 12

time →

Throughput = $\frac{4}{18}$
usage = $\frac{14}{18}$

Q6. P_1 P_2 ~~P_3~~ ~~P_4~~

Process No.	AT	Priority	CPU	I/O	CPU	CT
P_1	0	2	~~1~~ 0	~~5~~ 3	~~3~~ ~~2~~ ~~1~~ 0	10
P_2	2	3	~~3~~ ~~2~~ 1	3	1	15
~~P_3~~	3	1 → ~~H~~	~~2~~ ~~1~~ 0	~~3~~ 0	~~1~~ 0	9
P_4	3	4 → L	2	4	1	18

Mode : Preemptive
Criteria : Priority based
find CT

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

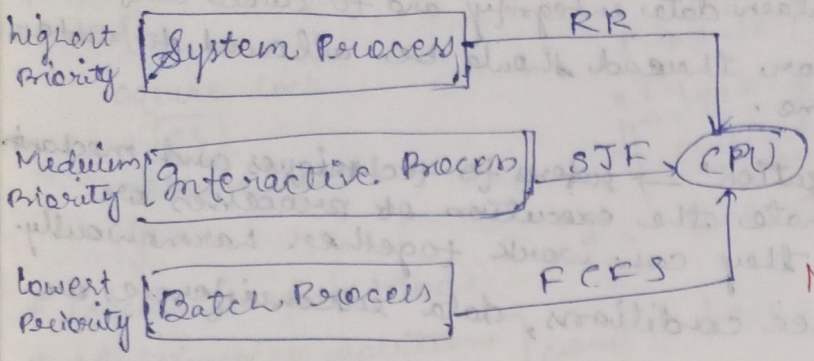
P_4 I/O

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

0 1 2 3 4 5 6 7 8 9 12 13 14 15 17 18

I/O I/O P_2 I/O

[Best] "Multilevel Queue Scheduling"



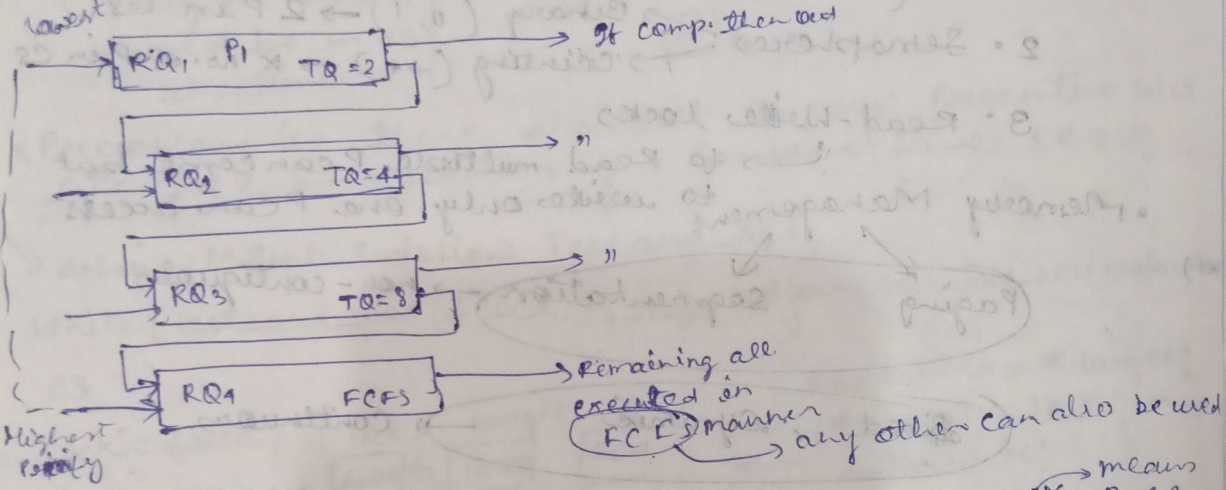
Not Practical

Causes starvation

can be solved by

Multilevel feedback Queue

Multilevel feedback Queue → Using for low level process not for high level



Process synchronization

Processes

Cooperative Processes

Independent Processes

Share: Variable, Memory, Code, Resources → CPU, Printer, Scanner

int shared = 4 → means P_1 & P_2 are not synchronized → synchronized

P_1	P_2
1. int x = shared	int y = shared
2. x++;	y--;
sleep(1)	sleep(1)
shared = x;	shared = y;

stop → preemption

Race Process

Q. Lecture - 26 - 27 - 28 - 31

'Critical Section' → represents a portion of code or block where a process or thread accesses a shared resource, To maintain data integrity and to avoid conflicts, only one process or thread should be allowed to enter the CS at a time.

'Process Synchronization' → refers to techniques and mechanisms used to coordinate the execution of processes or threads so that they can work together harmoniously.
↳ prevents race conditions, data inconsistencies or deadlocks.

Mechanisms to fulfil Synchronization Requirements

1. Lock / Mutex (Mutual Exclusion)
2. Semaphores → Binary (0, 1) → 2 P in CS
   → Counting (++) → x no. of P in CS
3. Read-Write locks
   ↳ to Read multiple P can come but to write only one P can access

- Memory Management
  - Paging, Segmentation → non-contiguous
  - Fixed, Dynamic → Continuous

Lock Variable in OS → no use of record

Critical section solution using Lock

```
do{
  aquire lock
  CS
  release lock
}
```

```
lock != 0
1. while (lock == 1);   ] Entry code
2. lock = 1
3. [ Critical section ]
                        ] exit code
4. lock = 0
```

* Execute in user Mode
* Multiprocess solution
* No mutual exclusion

] cond.

Lock = 0 → vacant
1 → full

lock = ∅ X ∅ !

guarantee
only one process should be in CS at a time

P1 | P2 ✗
P1 ✓ P2

1 2 3 4

P1 P2

• Problem is there shouldn't be any preemtive b/w line 1 & 2.
→ solution → combine line 1 & 2.

↓

Achieve Mutual Exclusion **Test and Set**

```
While (test_and_set(&lock));  ⟹
CS
lock = false;
```

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

lock: true → false (1000) ← target: 1000 ← rv: false

Turn variable (Strict alternation)

• 2 process solution
• Run in user mode

1. Mutual Exclusion ✓ → Never 2 proc will be there in CS
2. Progress X
3. Bounded wight ✓

```
int turn = ∅ 1
```

P0 P1 X
CS

P1 P0 X
CS

```
Process P0                    Process P1
In CS                         does
while (turn != 0)             while (turn != 1)
Critical section              Critical section
turn = 1                      turn = 0
```

Semaphore → tool to prevent race condition → to synchronise the processes → lock lagane ka tarika

Counting semaphore (−∞, ∞) | Binary semaphore (0, 1)

*Semaphore is an int var. which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve Synchronization"

Counting Semaphore

P(), Down, Wait
V(), UP ; Signal, Post, Release

P_1, P_2, P_3

Entry Code → CS → Exit Code

S [-4] → 4 processes in suspend/Block list

S [10] → 10 processes can successfully enter into CS → 6P 4V → 10-6 = 4+4 = 8

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

```
UP(semaphore S)
{
  S.value = S.value + 1
  if (S.value ≤ 0)
  {
    Select a process from suspended list, wake UP();
  }
}
```
Put it into Ready Queue.

Binary Semaphore

0 / 1

Down, P, wait

Up, V, Signal

S = 1 0 → successful opn

S = 0 0 → Block unsuccessful opn.

```
Down(semaphore S)
{
  if (S.value == 1)
  {
    S.value = 0;
  }
  else
  {
    Block this process and place in suspend list, sleep();
  }
}
```

```
UP(semaphore S)
{
  if (suspend list is empty)
  {
    S.value = 1
  }
  else
  {
    Select a process from suspend list and wake up();
  }
}
```

Down
CS → Common code section which all process share

Up

Q. What is maximum no. of processes that may persent in CS at any point of time? ⇒ 10

→ each processes P_i (i = 0 to 9) execute the following code

```
repeat
  P(mutex)
  CS
  V(mutex)
forever
```
(entry section, exit section)

Mutex: 1 0 0 0 0

$P_1 P_{10} P_2 P_{10} P_3 P_{10} P_4 P_{10} P_5 P_{10} P_6 P_{10} P_7 P_{10} P_8 P_{10} P_9 P_{10} P_{10}$

↓ 10

Process P_0 execute the following code

```
repeat
  V(mutex)
  CS
  P(mutex)
forever
```
(entry, exit)

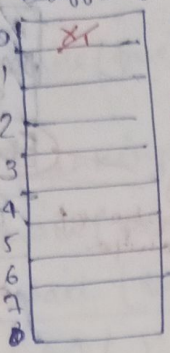
Ref (3)

Producer Consumer Problem

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

n = 8

Buffer[0 ... n-1]



out = 0

count = 0

if Buffer is empty consumer can't consume anything so, it will stuck in infinite loop

```
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;
    }
}
```

load to a register

```
load Rc, m[count];
DECR Rc
store m[count], Rc;
```

```
load Rp, m[count];
INCR Rp;
store m[count], Rp;
```

incr.

CPU execute this code

each instr. in microinstr then execute it, using register to load because of time saving

Case I: x_1 (it not synchronized)

(0+1) mod 8 = 1

(0+1) mod 1

Case II: → Race around condition

Producer, I_1, I_2 Consumer, I_1, I_2, Producer, I_3, Consumer, I_3

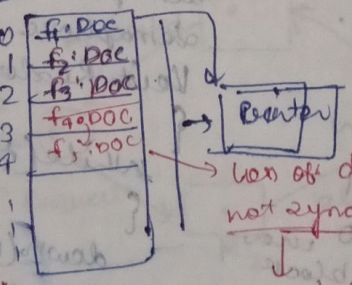
Printer-spooler Problem

```
I1. Load Ri, m[in]
I2. Store SD[Ri], "F-N"
I3. INCR Ri
I4. Store m[in], Ri
```

memory loc. of in will be loaded into Ri

mod Ri

Spooler directory



loss of data coz of not synchronous

can be done by using semaphore or monitor

Case I:

IN 0 3 4

P_1 f_1.doc

R_1 0 1

Case II: In 3

P_1 P_2

f_4.doc f_5.doc

R_1 3 4 R_2 3 4

$P_1 I_1 I_2 I_3 / P_2 I_1 I_2 I_3$

Preempt / switching

Critical section → "it is part of a program which shared resources are accessed by various Processes".

Cooperative:
→ place where shared vari., Resources are placed.

P1
#include ---
main()
A, B
Non critical section
Entry section
count++
CS
exit section

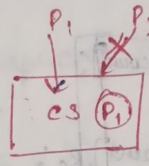
P2
#include ---
x, y
Non-CS
Entry section
count++
CS
exit section

Using semaphore / Monitor

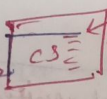
To stop the race around condn, we have synchronised the processes.

Synchronization Mechanism

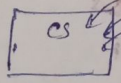
4 conditions } Rules for Sync Mech.
1. Mutual exclusion
2. Progress
3. Bounded Wait
4. No assumption related to H/W, speed.



If P_1 is there P_2 is not allowed to go CS.



If P_2 is not letting go P_1 inside CS then it is Not progress



No. of processes execute in CS should be bounded for both P_1 & P_2

P_1	P_2	
10	1	✓
∞	1	✗

Readers-Writers Problem → Using Binary semaphore

R-R → No prob.
R-W, W-W, W-R] Prob.

Case I: R-W
src = ∅ 1
mutex = ✗ ∅ 1
db = ✗ 0 → 0 then ↳ block

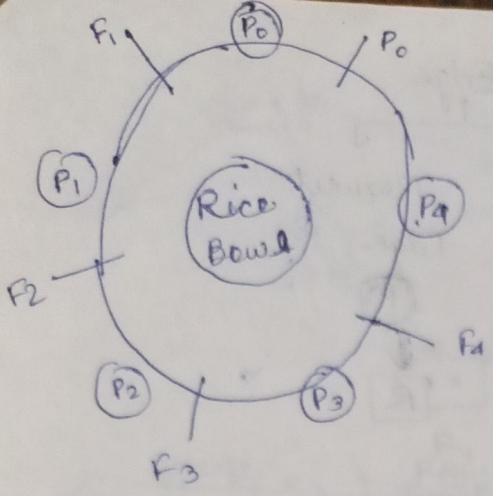
Case II: W-W
src = ∅ 1
mutex = ✗ 0
db = ✗ 0 → 0 then ↳ block

Case III: W-R
mutex = 1 → 0 → then block
db = ✗ 0

Case IV: R-R
src = ∅ ✗ 2
mutex = ✗ 0 ✗ ∅ 1
db = ✗ 0

```
int src=0
Semaphore mutex=1;
Semaphore db=1;
Void Reader(void)
{
  while(true)
  {
    down(mutex)
    src = src+1
    if(src==1) then down(db);
    VP(mutex)
    CS
    [DB data base]
    down(mutex)
    src = src-1;
    if(src==0) then VP(db);
    VP(mutex)
    process_data
  }
}
```

```
Void writer(void)
{
  while(true)
  {
    down(db);
    [DB]
    VP(db);
  }
}
```



'5' philosophers

'5' forks

Dining philosophers problem

race around condition
↳ can be solved using array of semaphore

```
Void philosopher (void)
{
while(true)
{
Thinking();
wait(take_fork(Si));
Wait(take_fork(S(i+1)%N))
EAT();
Signal(put fork(i));
Signal(put fork((i+1)%N));
}
```

P1 P2

↳ This is a special case where mutual exclusion doesn't exist.

Solution — array of semaphore

B_0 S_1 S_2 S_3 S_4 ← initialise with '1'

1 1 1 1 1

P_0 :	S_0	S_1
P_1 :	S_1	S_2
P_2 :	S_2	S_3
P_3 :	S_3	S_4
P_4 :	S_4	S_0

→ S_0 S_4

• Dead lock situation is occuring / to get of preempt

↳ one solution is reverse the semaphore values of P_4 → S_0 S_4

↳ for(N-1)P

wait(take_fork(Si))
wait(take_fork(S(i+1)%N))

for Nth P

Wait(take_fork(Si+1))
Wait(take_fork(Si))

Dead lock

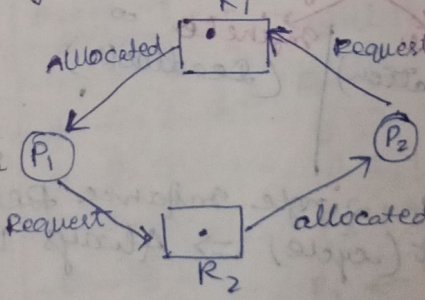
P_1 P_2
S_1 S_2 H
S_2 S_1 W

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. cond

Mutual exclusion should be non sharable

(1) Mutual exclusion
(2) No preemption
(3) Hold & Wait
(4) Circular Wait

all should be present to occur deadlock.



Resource Allocation Graph (RAG) → To check whether the system is deadlock or not

↳ efficient & convient way to represent state of the system
↳ how the resources are allocated to process & how process have been assigned to multiple resources.
↳ to represent, in our system is there any deadlock or not.

Preemption → once scheduled will be executed till the completion no other process can be scheduled meanwhile

Vertex → Process

Resource vertex → CPU / Monitor / Resister / Memory / Printer

Single Instance

Multi Instance

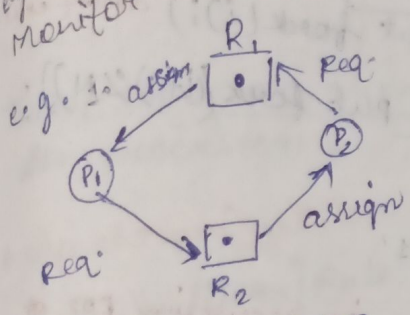
e.g. Resister

e.g. CPU, Monitor

Edge

Assign Edge

Request Edge



e.g. 1. assign

Single instance

circular wait

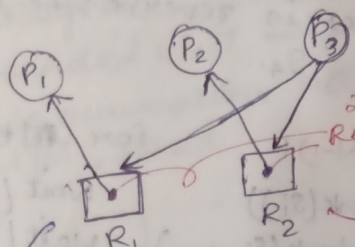
	Allocate R1	Allocate R2	Request R1	Request R2
P1	1	0	0	1
P2	0	1	1	0

Availability (R1, R2) → (0, 0) → we can't fulfill Req. of P1 & P2

→ Compare: Can we fulfil the request

⇒ Deadlock

2.



single Resource

	Allocate R1	Allocate R2	Request R1	Request R2
P1	1	0	0	0
P2	0	1	0	0
P3	0	0	1	1

→ P1 P2 P3

Availability — (R1, R2) → (0, 0) → so P1 will terminate & Resource will be get back

1 0 ← P2

1 1 ← P3

Finally P1 P2 P3 all get execute

⇒ No Deadlock.

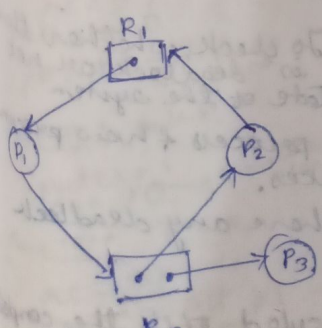
Waiting

finite (starvation)

infinite ∞ (Deadlock)

* In single instance Resource, if RAG has circular wait (cycle) ⇒ Always DeadLock and vice versa.

Multi-instance Resource Allocation



	Allocate R1	Allocate R2	Request R1	Request R2
P1	1	0	0	1
P2	0	0	1	0
P3	0	1	0	0

Availability — (R1, R2) → (0, 0) → using this avl. we can fulfil the demand of P3 → P3 will be free

0 1

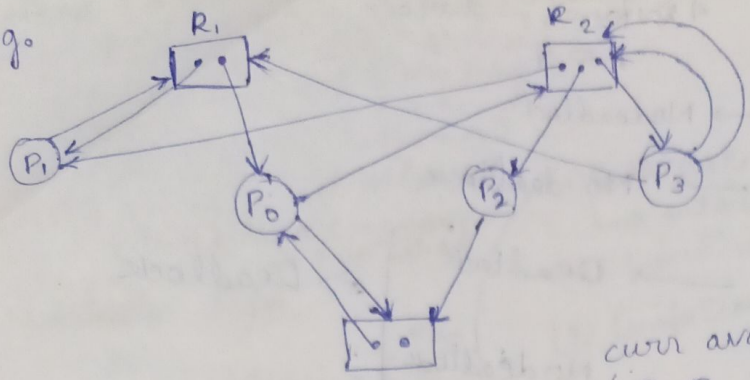
P1 can execute → 0 1

1 0

P2 can execute → 1 1

⇒ No deadlock

e.g.



	Allocate R1	R2	R3	Request R1	R2	R3
P_0	1	0	1	0	1	1
P_1	1	1	0	1	0	0
P_2	0	1	0	0	0	1
P_3	0	1	0	1	2	0

curr availibility
$(P_1\ R_2\ R_3) \rightarrow (0,0,1) \rightarrow P_2$

0 1 0

0 1 1 $\rightarrow P_2$

1 0 1

1 1 2 $\rightarrow P_1$

0 1 0

2 2 2 $\rightarrow P_3$

0 0 0

2 3 2

$\Rightarrow$ So, No Deadlock

Various Methods to Handle Deadlocks.

** (1) Deadlock ignorance (Ostrich Method) → Don't affect the perf.

(2) Deadlock prevention → we will prevent the deadlock cond.

(3) Deadlock Avoidance (Banker's Algo)

(4) Deadlock detection & Recovery

↳ we will kill all the processes which are in the deadlock situation

Resource preemption

(1) Mutual exclusion

(2) No preemption

(3) Hold & Wait

(4) Circular wait

↓

A process can req. in increasing order.

Avoidance | Detection

Banker's Algorithm ⇒ Deadlock

Processes	Allocation A	B	C	Max Need A	B	C	Current Available A	B	C	Max − Allocation Remaining Need A	B	C
P_1	0	1	0	7	5	3	3	3	2	7	4	3
P_2	2	0	0	3	2	2	5	3	2	1	2	2
P_3	3	0	2	9	0	2	7	4	3	6	0	0
P_4	2	1	1	4	2	2	7	6	5	2	1	1
P_5	0	0	2	5	3	3	7	6	7	5	3	1
	7	2	5				7	7	5			
							7	8	7			
							10	8	7			

Total:
A = 10
B = 5
C = 7

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

$P_2\ P_4\ P_5\ P_1\ P_3 \Rightarrow$ No DEADLOCK

Q. A system have 3 processes each require 2 Resource; min no. of unit of R such that no deadlock will occur

→

R_1	R_2	R_3
2	2	2
1	1	1

+1 = 4

Min req. + 1 = 4

	P_1	P_2	P_3	4 instance — share
max.	K	K	K	R = 4
let K=1	1	1	1	→ No deadlock
K=2	1	1	1	→ No deadlock
K=3	1	1	1	→ Deadlock
				→ No deadlock

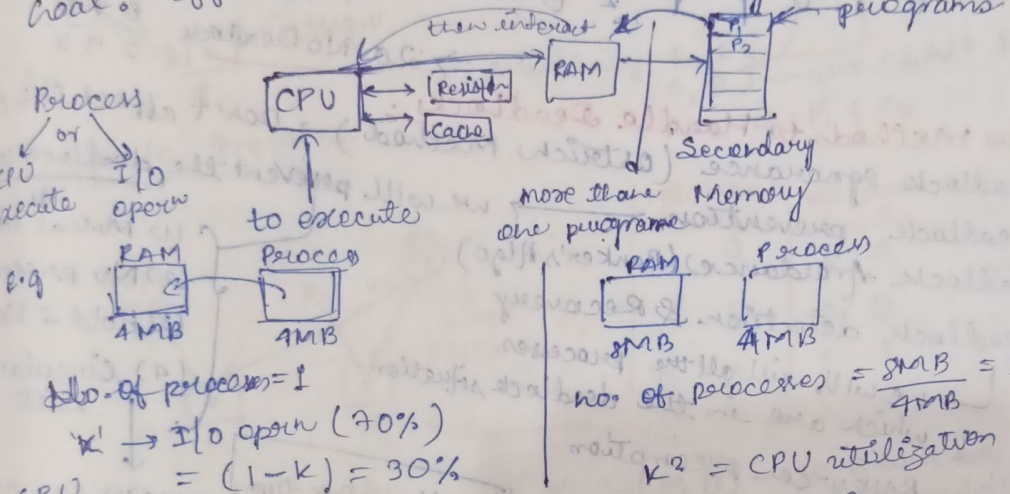
→ Deadlock

∴ K = 2 → for avoid deadlock.

Memory Management and Degree of Multiprogramming

↳ Method to managing primary memory

Goal: efficient utilization of memory.



e.g

No. of process = 1

K → I/O open (70%)

CPU utilization = (1 − K) = 30%

No. of processes = 8MB / 4MB = 2

K^2 = CPU utilization

$= 1 - K^2$

CPU $= 1 - (.7)^2$

$= 76\%$.

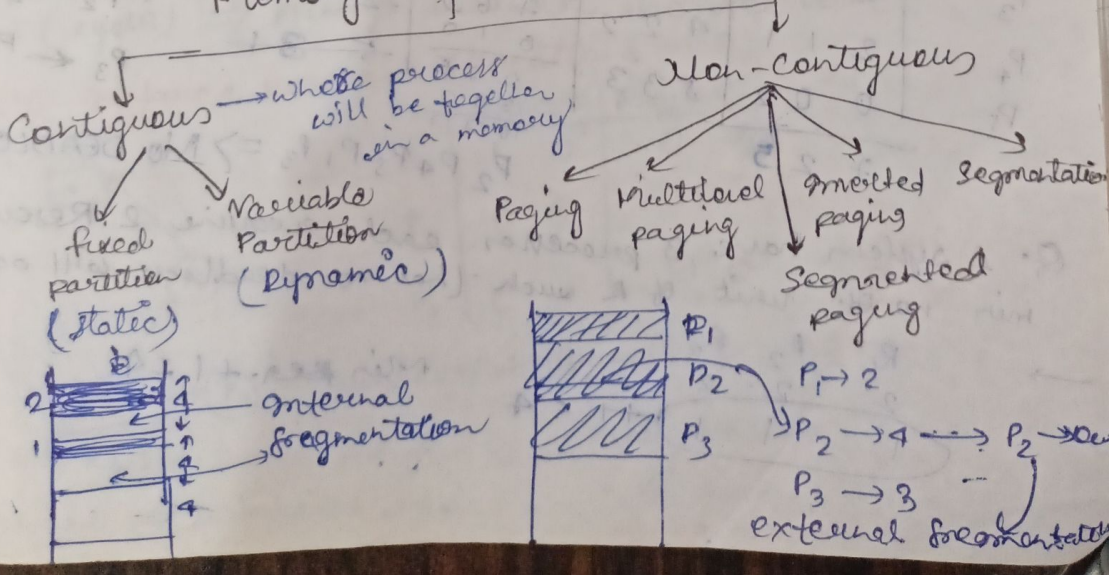
For n processes —

I/O open → K

CPU utilization $1 - K^2$

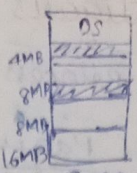
RAM :→ Primary Memory

Memory Management Techniques



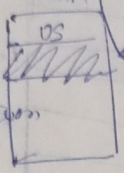
1. Fixed Partitioning (Static Partition)

1. No. of partitions are fixed
2. Size of each partition may or may not be same
3. Contiguous allocation so spanning is not allowed
   → allocate memory at Runtime



(1) Internal frag.
   ↳ internal wastage of memory
(2) limit in process size
(3) limitation on degree of multi-programming
(4) External fragmentation
   ↳ we can't accomodate process in memory coz of internal frag even if there is space

2. Variable size
or
Dynamic Partitioning



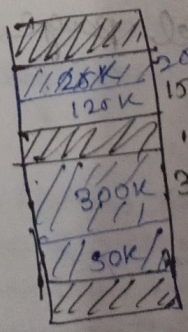
(1) No internal fragmentation
(2) No limitation on no. of process
(3) No limitation on Process size
(4) external fragmentation → ~~Compaction~~ Compaction
   ↳ compaction [undesirable]
(5) Allocation/deallocation is complex

Methods to allocate/deallocate memory to the processes in Contiguous

(1) First-fit : Allocate the first hole that is big enough
   ↳ simple | fast, internal fragmentation
(2) Next-fit : Same as first-fit but start search always from the ~~next~~ last allocated hole. → more fast
(3) Best-fit : Allocate the smallest hole that is big enough
   ↳ remaining memory will be less
   ↳ less internal fragmentation
   ↳ slow
(4) Worst-fit : Allocate the largest hole
   ↳ more ~~fast~~ internal fragmentation
   ↳ slow

(1) ≈ (2) > (3) > (4)

Q. Requests from processes are 300K, 25K, 125K, 50K respectively. The above req. could be satisfy with



(A) Best fit
(B) First fit
(C) Both
(D) None

Non-Contiguous Memory Allocation

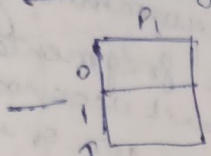
↳ Process can be allocate in non-continous manner instead of Paging or in partition.

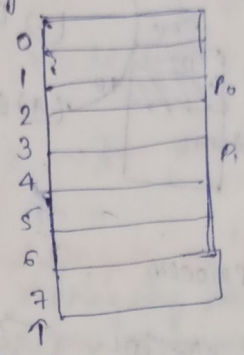
↳ Page size = frame size

↳ partion will be done in secondary memory called page

Main (RAM) memory partition called frame

eg.





Page table of P1

0	f2
1	f4

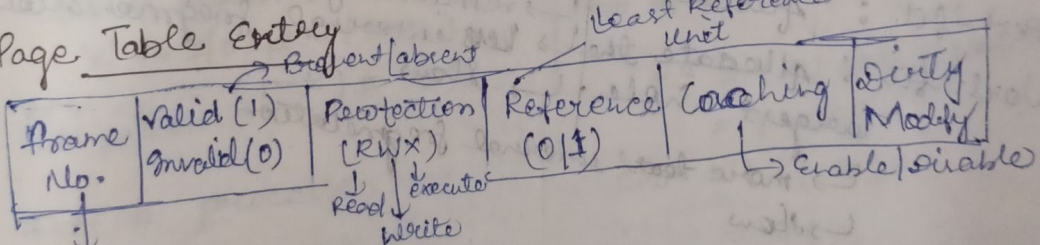
MMU (Memory management unit)

Process size = 4B
Page size = 2B
No. of pages = 4B/2B

M/M size = 16B
frame size = 2B
No. of frames = 16B/2B = 8 frames

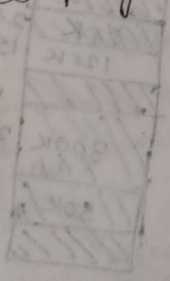
- Paging is a storage mechanism used to retrieve processes from secondary storage to main memory in form of pages. divide each process into pages. main memory in frames.

Page Table Entry



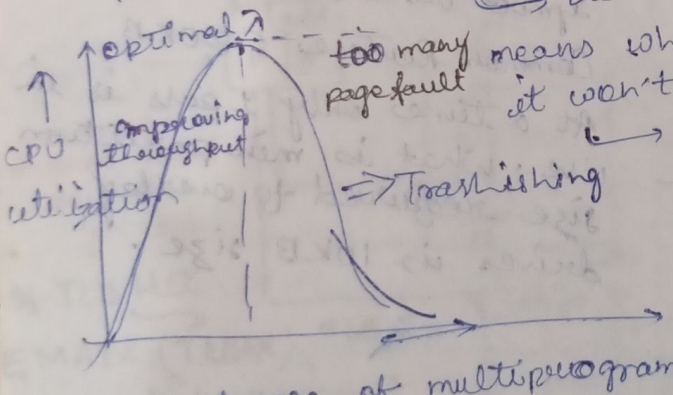
Mandatory field

Multilevel page



• Degree of multiprogramming is the no. of processes present in RAM.

• How to maximize the degree of multiprogramming → maximize the processes present in RAM (but RAM is limited) → bring only 1 page of each & every process
↳ but it will cause "page fault".
Page CPU will demand



↳ means whatever it won't be present.
↳ Page service take a lot of time
↳ To overcome Page fault
(1) Main memory size increase
(2) long term scheduler

Thrashing → refers situation in computer systems where the system spends a significant amount of time & resources cont. swapping pages b/w RAM & secondary memory. system is busy in moving pages in and out of memory rather than execute useful work.

Segmentation?

↳ method in which a process will be divided into parts & then put them into main memory.

difference with Paging

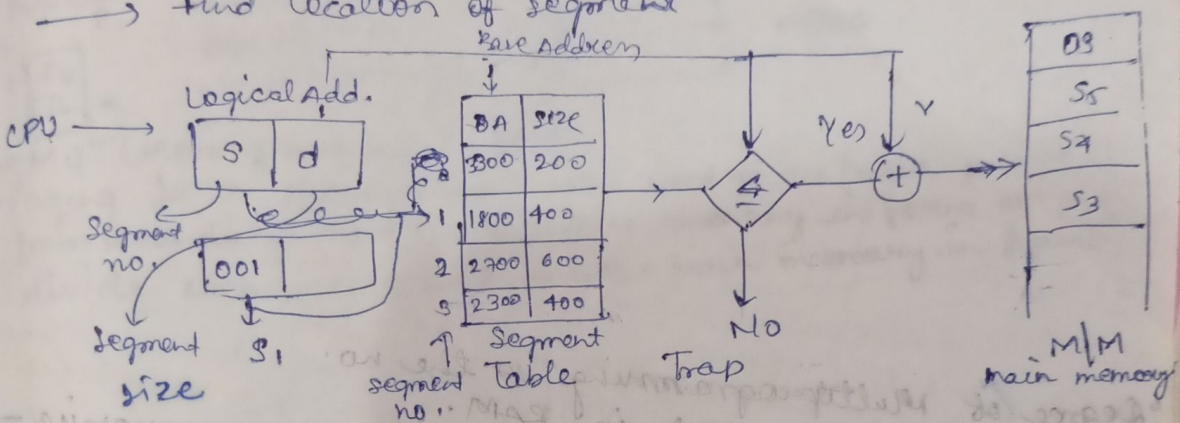
↳ it breaks the process without knowing the process like add()

segment

↳ it creates segments acc to the process of various size

- CPU doesn't know about the paging or segments it only knew the logic
- CPU generates logic address → convert → physical address

→ find location of segment



Overlays

↳ even if the my process size is more than size of memory, then i can make that process accommodate in main memory using the concept overlays.

Pass 1	Pass 2	
80KB	90 KB	80
30	30	90
20	20	30
10	10	20
140KB	150KB	10
		230KB

150KB

- Assembler use to convert source code into object code.

Consider 2 pass assembler

Pass 1 : 80KB , Pass 2 : 90KB

Symbol table : 30KB

Common Routine : 20KB

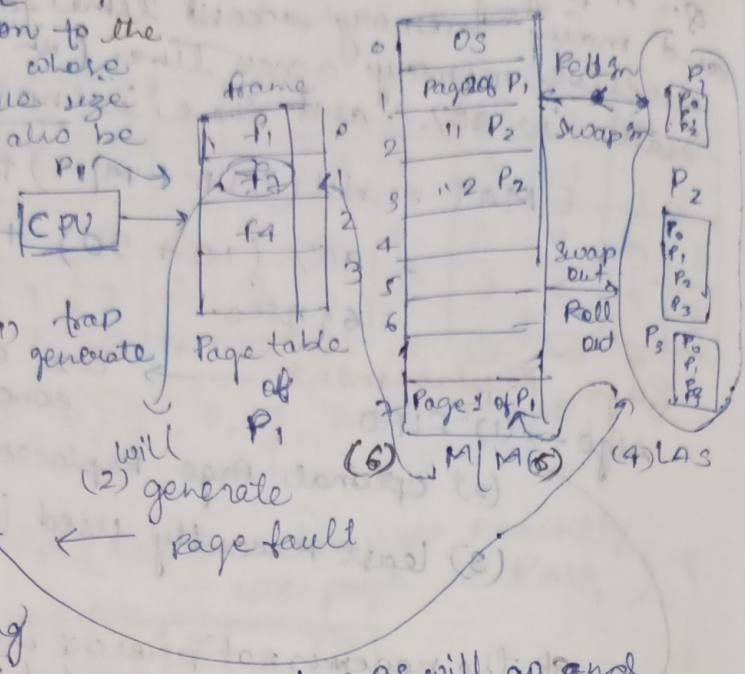
At a time only 1 pass is in use. What is min partition size required to overlay driver is 10KB size.

Virtual Memory [Page Fault]

↳ It provides illusion to the programmer the process whose size is larger than the size of main memory can also be execute.

can be done using

Paging | Segmentation



* time taken = Effective memory access time = P (page fault service time) + (1-P) (main memory access time)

(+ma → msec)

P → prob. of page fault occur (in msec)

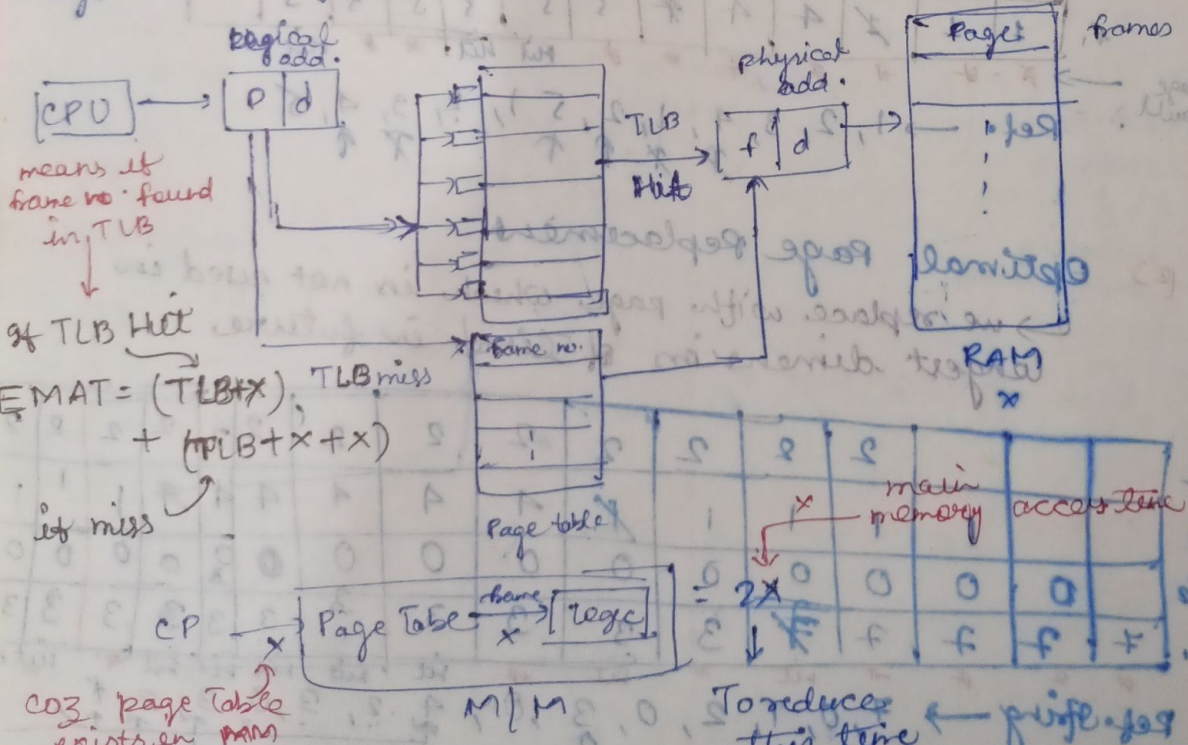
now OS will go and find P_1 and will put it to M/M

ma → msec

Translational Lookaside Buffer (Cache) [TLB]

→ limited size

If there is no Page fault
Page Table ⇒ Physical address



- TLB is faster than RAM → so we will put page table entries in TLB

Q. A paging scheme using TLB. TLB access time 10 ns & main memory access time taken 50 ns. What is effective memory access time (in ns) if TLB hit ratio is 90%. and there is no page fault.

EMAT = Hit(TLB + M|M) + miss(TLB + M|M+M|M)

= 90%.(10 + 50) + 10%.(10 + 50 + 50)

= 65 nsec.

Page Replacement Algo → (1) FIFO — coz of Virtual memory concept for swap In|swap Out

(2) Optimal Page Replacement

(3) least Recently used (LRO)

(1) If the page is not present in frame table → we need to take it from hard disk to frame table → and if frame table is full → we need Replace some page with the required page → and for that we will use FIFO means first-in first-out

Belady's Anomaly in FIFO → If we will incr. the no. of frames page hit will decrease. So FIFO is suff. from BAL

f_3			3	3	3	2	2	1	2	2	4	4	
f_2		2	2	2	1	1	1	1	1	3	3	3	
f_1	1	1	1	4	4	4	5	5	5	5	5	5	
page fault →	*	*	*	*	*	*	*	Hit	Hit	*	*	Hit	

Pagefault = 9

Hit = 3

Ref. → 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

(2) Optimal Page Replacement

↳ we replace with page which is not used in longest dimension of time in future

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

7 comes at last from [2, 1, 0, 7]

Page. Hit = 12, Page fault = 8

(3) Least Recently Used → (Replace the least recently used page in part)

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

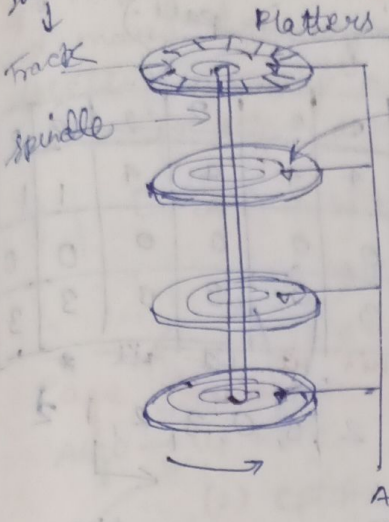
Ref. Str → 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2

(4) Mostly Recently used (Replace the most Recently use page in Part)

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

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

Disk Architecture



Platters → Surface → Track → sectors → data

Disk size = P × S × T × S × D

1K = 2^{10} B
1M = 2^{20} B
1G = 2^{30} B
1T = 2^{40} B
1 byte = 2^3 bit
1 B

* No. of bits required to represent Disk size, $2^{(n)}$ → n is bits req.

Disk Access Time

(1) Seek Time : Time taken by R/W head to reach desired track.

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

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

(4) Transfer time = Data to be transfer / Transfer Rate

(No. of heads) × (Capacity of 1 track) × (No. of Rotations in 1 sec)

↓ including above & below

3000R = 60 sec

1R = 60 / 3000 sec

Queue time

controller time

Disk access Time = ST + RL + TT + CT + QT

Disk Scheduling Algorithm

→ To minimize the seek time.

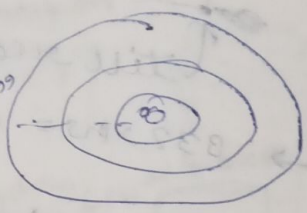
(1) FCFS (first come first serve)
(2) SSTF (shortest seek time first)
(3) SCAN OR Elevator Algo
(4) LOOK
(5) CSCAN (circular SCAN)
(6) CLOOK (circular look)

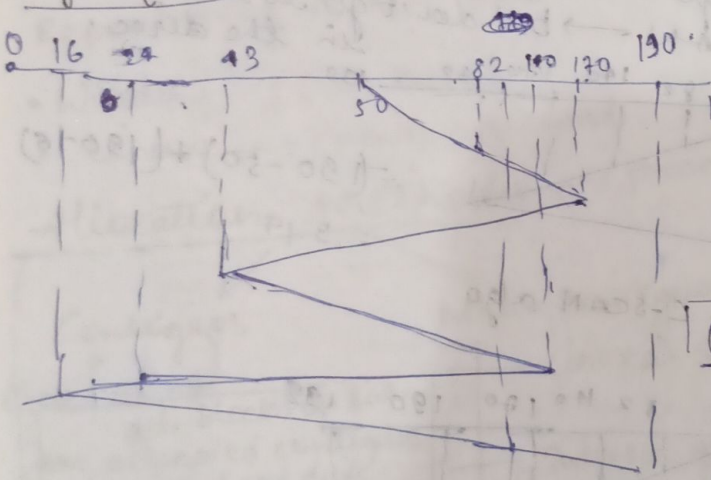
FCFS

→ first request → final point

Q. A disk contains 200 tracks (0-199)
Request queue contains track no.
82, 170, 43, 140, 24, 16, 190 resp.
curr. position of R/W head = 50
Calculate total no of tracks movement by R/W head.

seek time





No starvation
low performance

(170 - 50) + (170 - 43) + (140 - 43)
(OR) + (140 - 16) + (190 - 16)

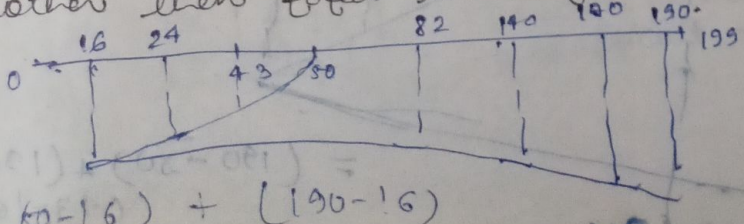
= (82 - 50) + (170 - 82) + (170 - 43) + (140 - 43) +
+ (140 - 24) + (24 - 16) + (190 - 16)

SSTF

→ which has closest seek time wrt current

Q. A disk contains 200 tracks (0-199) Request queue contains track no. 82, 170, 43, 140, 24, 16, 190 Respectively.
curr. position of R/W head = 50.
→ calculate total no. of tracks movement by R/W head using SSTF?
→ if R/W head takes 1ms to move from one track to another then total time taken?

Starvation



(50 - 16) + (190 - 16)

SCAN → in moving one direction, final all the req. in on-direction first

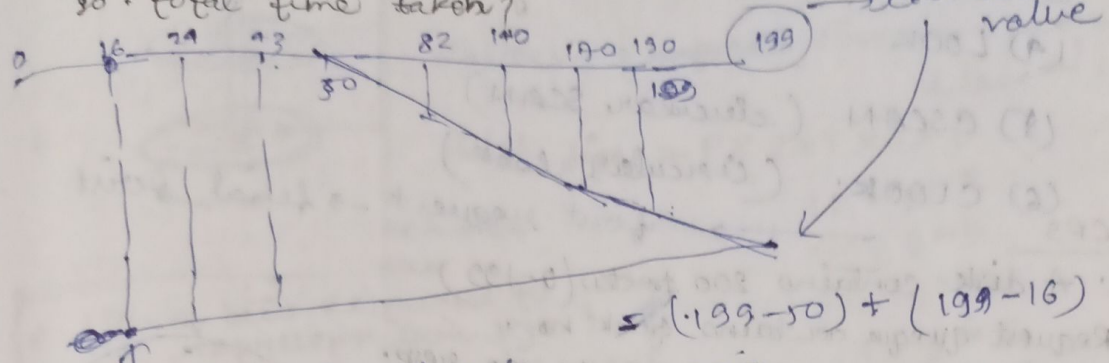
disk contains 200 tracks

ready queue 82, 170, 43, 140, 24, 16, 190] requests

curr pos = 50, direction towards the large value

→ calculate no. of track movement using SCAN?

→ If R/W head takes 1ms to move from one track to another so, total time taken?



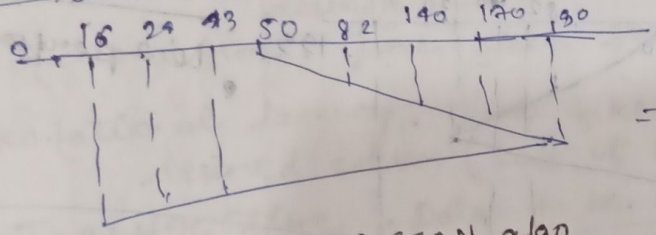
= (199 − 50) + (199 − 16)

till request only = 332

→ 332 × 1ms = 332ms

LOOK → use look algo

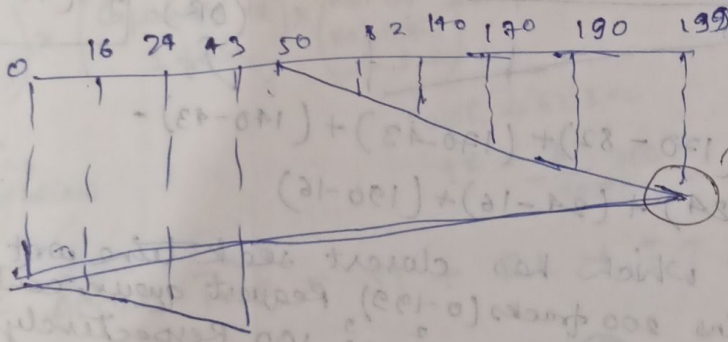
→ same as SCAN → but don't go to the last value in the direc.



= (190 − 50) + (190 − 16)

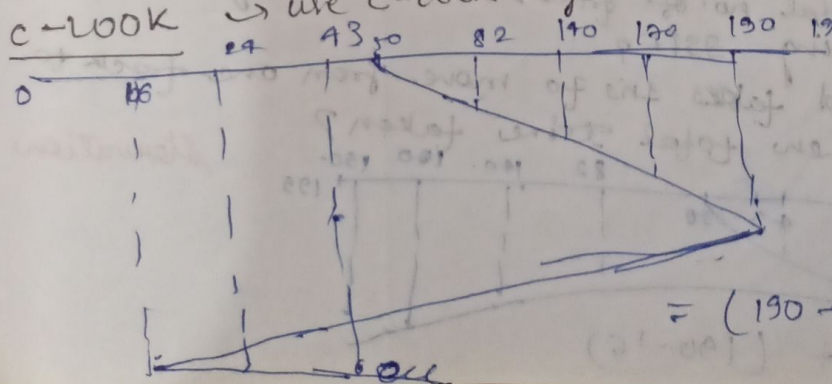
= 314

C-SCAN → use C-SCAN algo



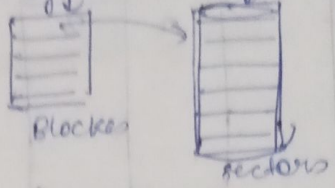
= (199 − 50) + (199 − 0) + (43 − 0)

C-LOOK → use C-LOOK algo



= (190 − 50) + (190 − 16) + (43 − 16)

File System → to manage files
↳ to store file
↳ to fetch file
user → file → folder/directory → file system → Hard Disk (permanent storage)
file → collection of data/information

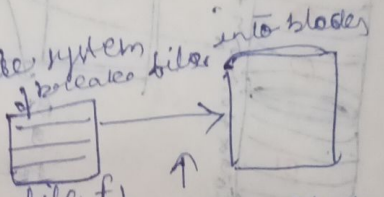


operations
(1) creating
(2) reading
(3) writing
(4) deleting
(5) Truncating
(6) Repositioning

File Attributes
(1) Name
(2) extension (type)
(3) Identifier (by OS)
(4) Location
(5) Size
(6) Modified date / created date
(7) Protection / permission
(8) Encryption, compression

Delete with attributes
Delete without attributes only data

Repositioning of R/W head
- Windows → GUI based
- Linux → command based



Allocation methods → process of allocating Blocks into sectors

Contiguous
Process of all blocks of file are allocated contiguous way in hard disk sectors

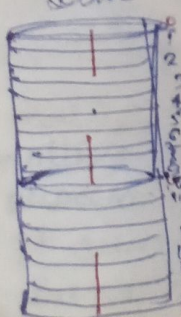
Non-Contiguous Allocation
- Linked list Allocation
- Indexed Allocation

(1) efficient disk utilization
(2) Access faster

Contiguous Allocation
↳ all blocks of file are allocated contiguous way in sectors of hard disk

Advantages —
(1) Easy to implement
(2) Excellent Read Performance
↳ less seek time (sec)
↳ less access time

Disk

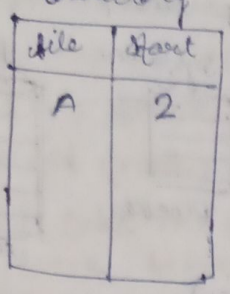


Directory

file	start	length
A	0	3
B	6	5
C	14	4

Disadvantages — (internal frag) & external frag
(1) Disk will become fragmented
(2) Difficult to grow file

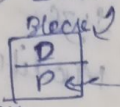
(1) Linked list Allocation → All blocks have a data & pointer [D|P]



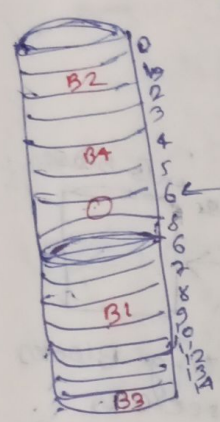
Advantages →
(1) No external fragmentation
(2) file size can increase

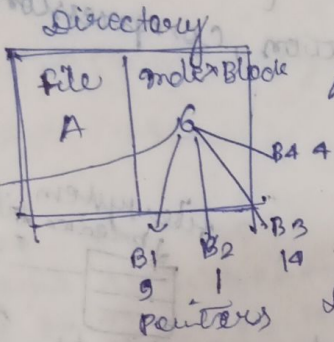
movement of R/W head will be high

Disadvantages →
(1) Large seek time
(2) Random Access/Direct access difficult
(3) Overhead of pointers



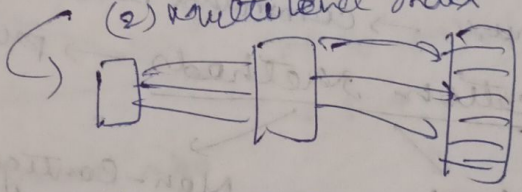
(2) Indexed Allocation → like book index



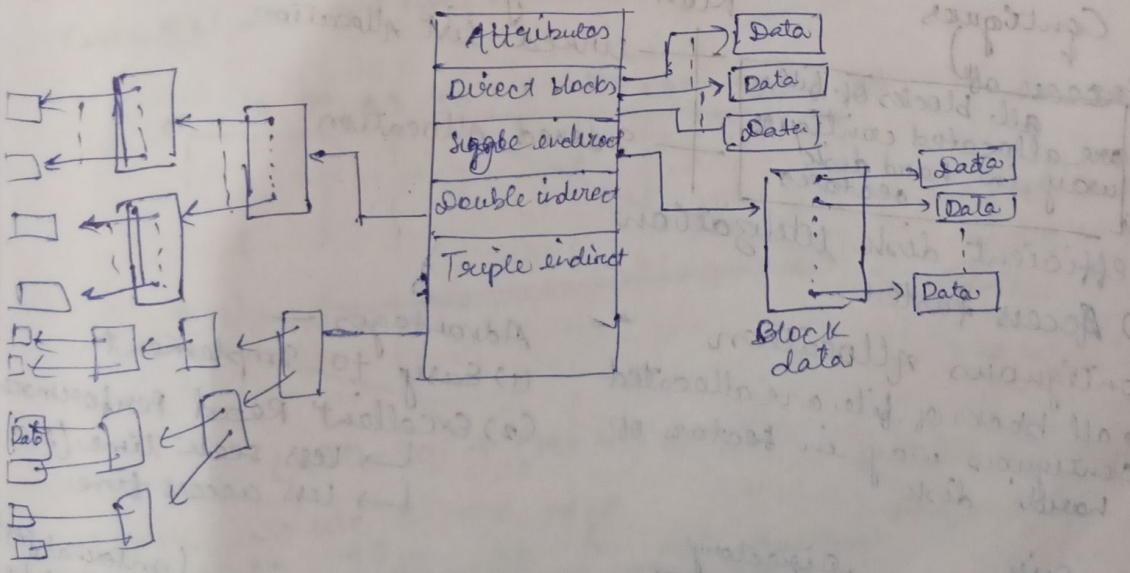


Advantages →
(1) Support direct access
(2) No external fragmentation

Disadvantages →
(1) Pointer overhead
(2) Multilevel index



Unix Inode Structure



Protection & Security in Operating system

C I A

Confidentiality — Unauthorized reading of data
↳ we can remove this using encryption

Integrity — Unauthorized modification of data

Availability — Unauthorized destruction of data

Theft of service — Unauthorized use of resources

Denial of service (DOS) → Prevention of legitimate use.

RTOS → (Real time operating systems) → designed for systems that require deterministic and real-time response. It is commonly used in embedded system, control systems and IOT devices.

A program is a set of instructions, A process is an instance of a program in execution with its own resources and memory space, and A thread is a unit of execution within a process that allows for concurrent execution of tasks. Processes provide isolation and protection b/w different instances of program, while threads within a process share resource and enable parallel execution of tasks.

1. Multiprocess : execution of multiple processes on a system with multiple CPUs. Multiple process can execute concurrently. Each process has its own memory space and resources. It aim to increase system throughput and provide faster execution by distributing the workload.

2. Multithreading → involves executing multiple threads within a single process. Threads shares the same memory space and resources, such as file handles and network connections. allows IInd execution within a process.

3. Multiprogramming → multiple program loaded into memory simultaneously and CPU switches b/w them to execute instructions. → The purpose of multiprogramming is. maximize CPU utilization and keep CPU busy by quickly switching b/w different programs.
each program has its own separate memory space.

4. Multitasking → allows multiple tasks or processes to run concurrently on single CPU. CPU time divided among tasks, giving illusion of IInd execution. OS switching b/w tasks. OS switches b/w tasks giving each task a time slice or quantum to execute.