

# Operating System

"Is a software and it works as an interface between user and hardware."

Primary Goals :-

- 1.) Provide convenience to user
- 2.) Maximize throughput (no. of tasks exec./ time)

Functionalities :-

1.) Resource management :

- When multiple users are accessing hardware
- parallel processing: Thus need to manage resource

2.) Process management :

- multi-processing (CPU scheduling)

3.) Storage Management :

- Hard disk management, file system mgmt.

4.) Memory Management :

- RAM (limited space). Thus, need to manage

5.) Security & Privacy :

- Authentication by user

- security b/w processes, so no two processes overlap.

## \* Types of OS

### 1.) Batch OS

- operations performed in batches

### 2.) Multi-programmed OS

- Aim is to get max. no. of processes in RAM.
- Non-preemptive

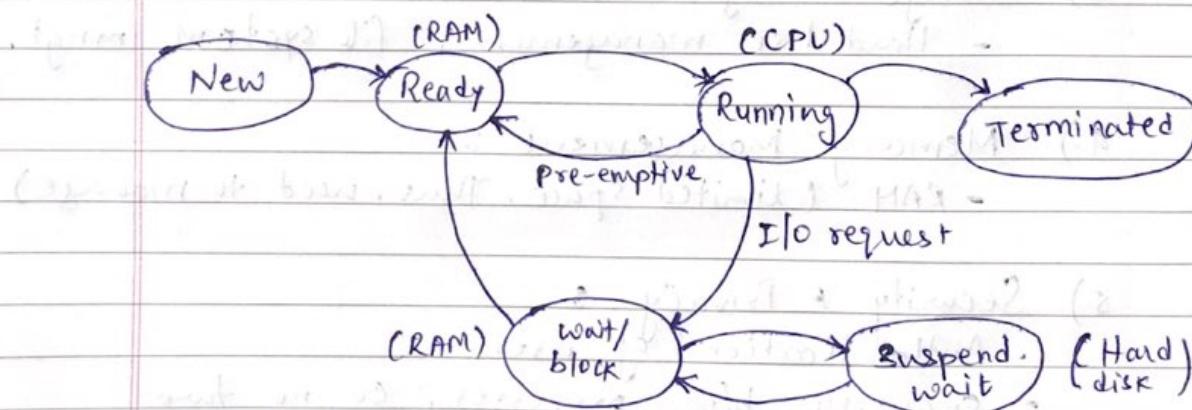
### 3.) Multi-tasking / time sharing OS

- Preemptive. No need to execute entire process at once. You can execute process in multiple parts.

## \* Process and its life cycle

Process :- is a program in execution.

- \* Non-preemptive :- Entire process is executed at once.
- \* Preemptive :- Process is executed in parts.



- First we get the max. no. of processes in Ready Queue.

- We schedule process pre-emptively to maximize throughput
- If our ready queue or wait queue is full, then we send process to hard disk (suspend state) until there is some space available in queue.
- If a process calls for Input/output, it is sent to wait state.

## \* System Calls

- Applications execute system calls to interact with hardware / OS.
- System call is a programmatic way to access kernel mode

File related :- open(); close(), read(), write()

Device related :- Accessing hardware like printer.

Information related :- getpid(), get system time()

Process control related :- load, execute, fork

Communication related :- pipe(), create / delete

Users do not have direct access to hardware, information, file, etc. so Kernel will give access to the user via system call.

- \* **Fork()** :-
- to create a child process.
- clone of parent process
- used for multiprocessing

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

Output:  
- hello  
- hello  
child Parent

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

Output:  
- hello  
- hello  
- hello  
- hello  
C1 C2 C3 C4 P

- fork 3 times (fork), 8 times : hello will be printed.

### \* User mode v/s Kernel mode

Hardware, devices are controlled by Kernel

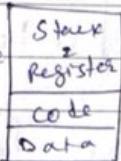
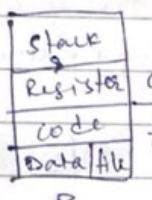
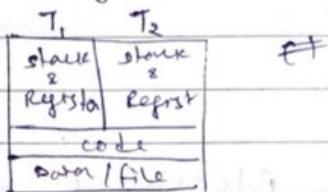
only, and not by user.

Real world Eg:- customer/you go to a bank (API)  
ask to withdraw money (System call), teller  
gives you money (Kernel mode gives result)  
from bank (Hardware)

## \* Process vs Threads (Fast multi-tasking or multi-processing)

(child)

- Process is created, using (fork) system call.

P<sub>1</sub>      P<sub>a</sub>T<sub>1</sub>      T<sub>2</sub>  
+  
Stack   Stack

Both process can perform  
a task far me parallelly

the same data & code  
to execute tasks parallelly

- |  |   |
|--|---|
| 1.) System call involved in process                                      | 2.) There is no system call involved                        |
| 2.) Os treats diff. process differently                                  | 2.) All user level threads are treated as single task by os |
| 3.) Diff process have diff copies of data, file, code                    | 3.) Same copies,  |
| 4.) & thus high overhead.  | Threads share same copies & thus, no too high overhead      |
| E- Threads can be created at API level or by os (both) at kernel level ] |   |
| 4.) Context switching is slow (heavy weight)                             | 4.) Context switching is fast (light weight)                |
- CPU will execute different processes for a certain time & then execute some other process for a certain time. The switching of processes for execution is called context switching.

The state of P<sub>1</sub> needs to be saved before executing P<sub>a</sub>. This is highly time consuming while switching context in threads are easy as it shares the same data.

5.) Blocking a process will NOT block another process and thus processes are independent.

5.) Blocking a thread will block (all other threads of) an entire process. Thus they are interdependent.

Blocking a process can be for I/O call.

### \* User level Thread v/s Kernel Level Thread

1.) User level threads are managed by user level library (API)

2.) Faster

3.) Context switching is faster

4.) If a user level thread performs blocking operation, then entire process is blocked (because kernel will block the entire process)

1.) Kernel level threads are managed by OS via system call.

2.) Slower (as system calls are involved)

3.) CTS is slower

4.) Blocking a kernel level thread does not affect others, because kernel knows which thread is blocked, as these threads are managed by kernels.

Both these threads share ~~the~~ same code & data <sup>of a process</sup> as they are threads only.

CTS of process > CTS of Kernel Thread > ~~CTS of User Thread~~

- \* Process are basically programs which are dispatched from ready state for execution (in CPU)
- \* Thread ~~are~~ is a segment of a process.  
Process can have multiple threads.

## \* Process Scheduling Algorithms

### ~~Pre~~ Pre-Emptive

- Partial process can be executed & before running another process.

### Non Pre-Emptive

- Entire process will be executed before starting to execute any other process.

Scheduling algo schedules processes from ready state to execute them in CPU.

SRTF, LRTF

Round Robin,  
Priority based

FCFS, SJF, LJF,

HRRN (Highest response ratio next), Multi-level queue.

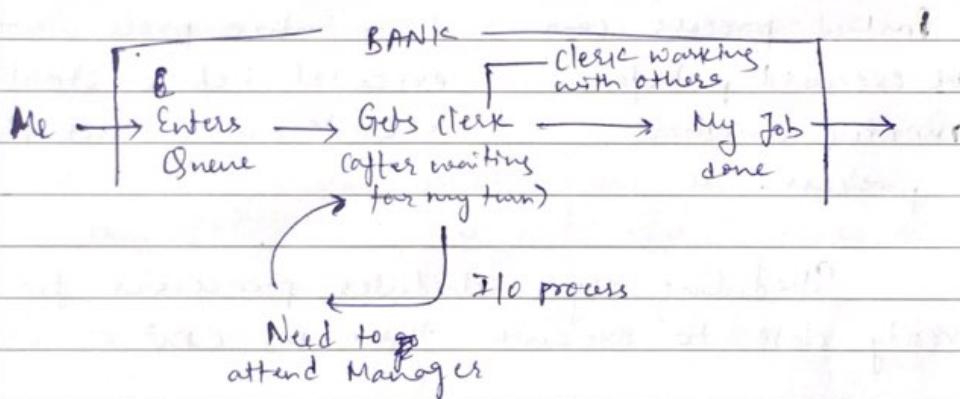
\* Arrival time :- The time at which the process enters the ready queue. (System time)

\* Burst time :- Time required to execute the entire process in CPU. (Duration)

\* Completion time :- The time at which the process is completed. (System time)

- \* Turn around time = completion time - arrival time
- \* Waiting time = Turn around time - Burst time
- \* Response time = (The time at which the process reaches the CPU for the first time) - (Arrival time)  
 [will be same as waiting time for Non-PE]

You can give an example of you going to Bank.



- \* First Come First Serve (FCFS) (Non-PE)

~~Entire final~~

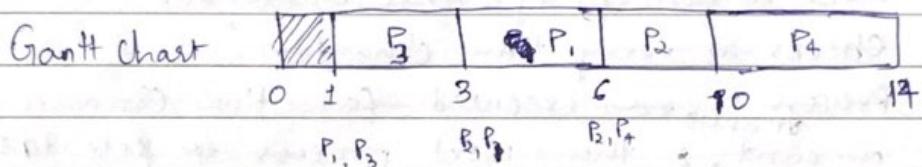
Processes are selected based on arrival time. (FCFS) and are executed fully.

### \* Shortest Job First (Non-PE)

process is selected based on least burst time. (Need to see if the process has arrived at that time or not.)

- So first check processes arrived at that syst. time, then select least burst time process.

Process	Arrival time	Burst time	Completion time
P <sub>1</sub>	1	3	6
P <sub>2</sub>	2	4	10
P <sub>3</sub>	1	2	3
P <sub>4</sub>	4	4	14



Also note, for pre

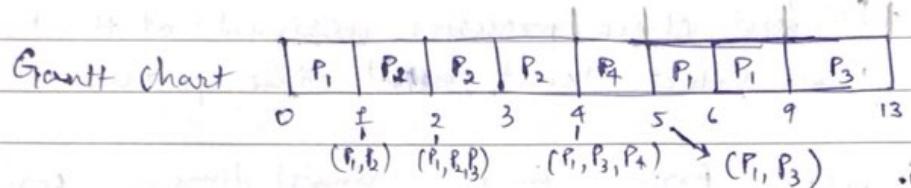
### \* Shortest Remaining Time first (SRTF) (PE)

- Can be considered as SJF with PE
- checks at every 1 unit of time.

Selects a process at every unit of time; based on remaining burst time & chooses the process with least remaining burst time.

Date \_\_\_\_\_ / \_\_\_\_\_ / \_\_\_\_\_

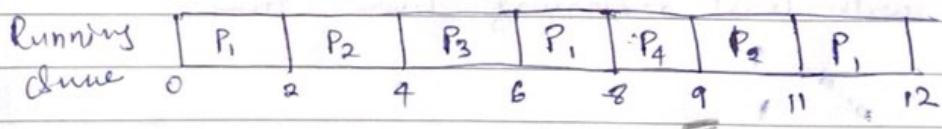
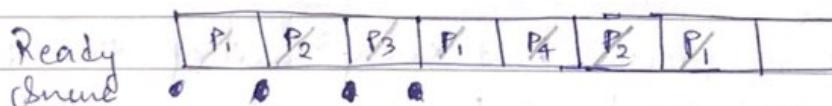
Pid	AT	BT	CT	TAT	WT	RT
P <sub>1</sub>	0	5 4 3	9	9	4	0
P <sub>2</sub>	1	3 2	4	3	0	0
P <sub>3</sub>	2	4	13	11	7	7
P <sub>4</sub>	4	1	5	1	0	0



## \* Round Robin

- Uses a concept of time Quantum.
- Checks at every time Quantum.
- Process is executed for time Quantum amount & then next process is selection is made.
- The process which came first in ready Queue will be selected.

Pid	AT	BT	CT	TAT	WT	RT
P <sub>1</sub>	0	5 8 1	12	12	7	0
P <sub>2</sub>	1	9 2	11	10	6	1
P <sub>3</sub>	2	2	6	4	2	2
P <sub>4</sub>	4	1	9	5	4	4



(Needs ready Queue for proper calculation)

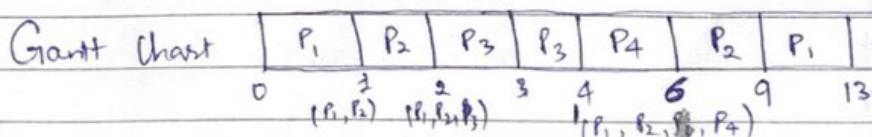
- Process is executed for time quantum amount or until it is completed, (whichever is earlier) and then selection for the next process is made
- The selection of the next process is based on the first arrived process in the ready queue.
- If the process is not completed put it back to the ready after its partial execution.

### \* Priority Scheduling (PE mode)

(can also be of Non-PE mode)

- Execute a process for 1 unit of time & make selection from the processes already arrived.
  - Selection is based on highest priority.
  - Execute the process for 1 unit time & make selection from the processes already arrived.
  - Selection is based on highest priority.
- (or choose a process based on highest priority)

Priority	Pid	AT	BT	CT	TAT	WT
10	P <sub>1</sub>	0	34	13	13	8
20	P <sub>2</sub>	1	13	9	8	4
30	P <sub>3</sub>	2	24	4	2	0
40	P <sub>4</sub>	4	2	6	2	0



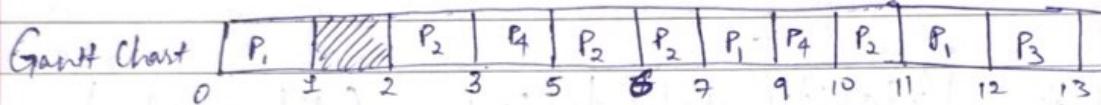
Date \_\_\_\_\_

\* Example of Mix Burst time

Mode :- PE

Criteria :- Priority

Priority	P <sub>i</sub> d	AT	Burst CPU	I/O	CPU
20	P <sub>1</sub>	0	X	8	31
30	P <sub>2</sub>	2	3	2	X
10	P <sub>3</sub>	3	X	3	1
40	P <sub>4</sub>	3	X	4	X



0 - P<sub>1</sub>

3 - P<sub>2</sub>, P<sub>3</sub>, P<sub>4</sub>

6 - P<sub>1</sub>, P<sub>2</sub>, P<sub>3</sub>

1 - 0

4 -

7 - P<sub>1</sub>, P<sub>3</sub>

2 - P<sub>2</sub>

5 -

8 -

9 - P<sub>4</sub>, P<sub>1</sub>, P<sub>3</sub>

10 - P<sub>2</sub>, P<sub>1</sub>, P<sub>3</sub>

11 - P<sub>1</sub>, P<sub>3</sub>

Turing Priority P<sub>i</sub>d AT CPU I/O CPU

↓ no. as ↑ priority. 20 P<sub>1</sub> 0 X 8 31

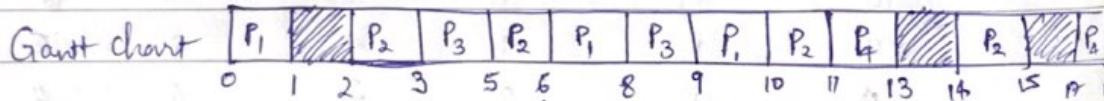
30 P<sub>2</sub> 2 3 X 3 X

10 H P<sub>3</sub>

3 X 3 X 3 X

40 P<sub>4</sub>

3 X X 4 X X



P<sub>1</sub> → 6

(P<sub>1</sub>, P<sub>2</sub>, P<sub>4</sub>)

(P<sub>2</sub>)

P<sub>3</sub> → 8

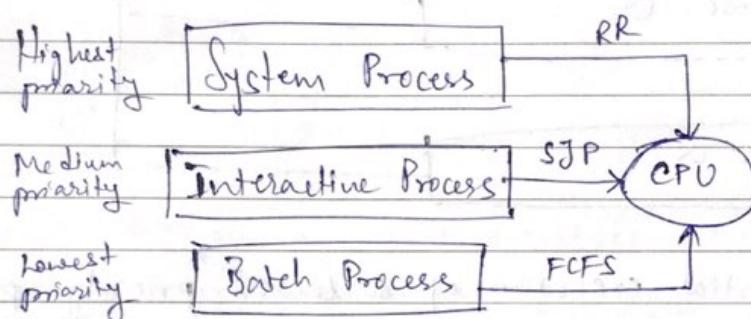
P<sub>2</sub> → 14

P<sub>4</sub> → 17

## \* Multi-level Queue Scheduling :-

- Different processes have different priority. So we keep different ready queue for them as well.

- For eg. Interrupt process has the highest priority.
- System processes have high priority.



- So, if a System process comes, then we will add it in system process ready queue, and that process will be given high priority.
- Different types of processes can use different process scheduling algorithm.

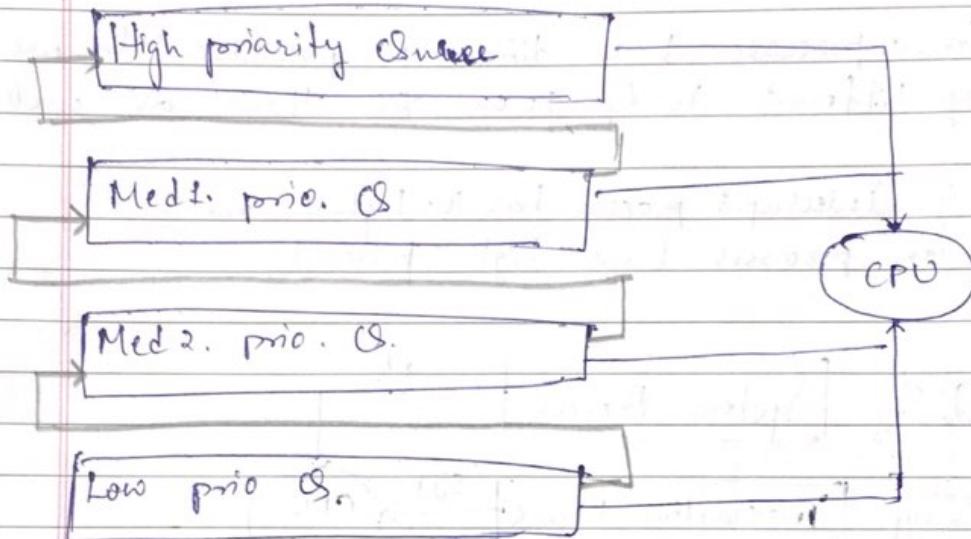
Disadvantage :- Only ~~two~~ If many high priority processes comes, then lower priority processes will get their turn after a long or need to wait for a long time.

- It is called Starvation.

8

- To overcome starvation, we use multi-level feedback queue Scheduling

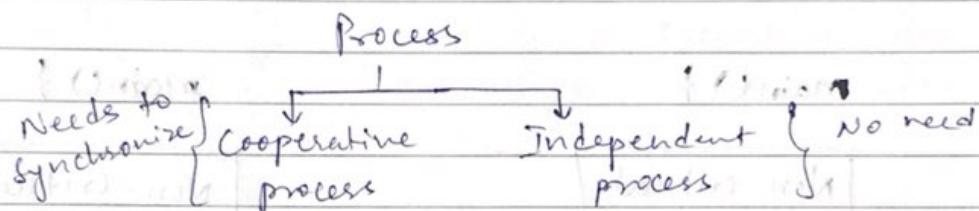
## \* Multi-level Feedback Queue



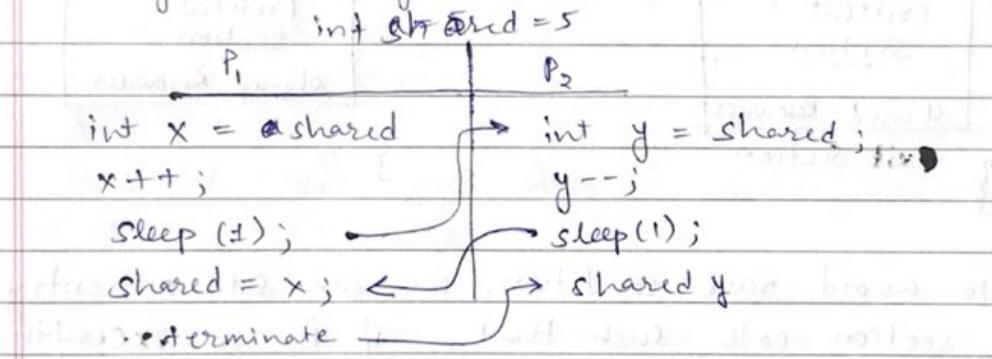
After partial execution of a lower priority process, the remaining process is inserted to a higher priority Queue, so that its turn can come early.

## \* Process Synchronization :

In Multi-processes system, it is possible that two processes are sharing resources as memory, code, variables, then executing 1 process may affect execution of other process.



No-PE are always int processes never need to worry about synchronization.



This will give two different answers. Once by starting at P<sub>1</sub>, and other by starting at P<sub>2</sub>.

This is called Race condition.

(Also, Side Note : During context switching from P<sub>1</sub> to P<sub>2</sub>, the states of P<sub>1</sub> needed to be saved, so that its remaining execution can be completed later.)

## \* Producer Consumer Problem

- \* Critical Section :- is a part of the program where shared resources are accessed by various processes.

P<sub>1</sub>P<sub>2</sub>

main() {

main() {

Non-critical  
sectionNon-critical  
section

Entry section

Entry section

critical  
sectioncritical  
section

Shared Resources

shared Resources

{ Exit section }

{ }

To avoid race condition, we can set an entry section code such that, if P<sub>2</sub> is executing critical section, then P<sub>1</sub> cannot execute critical section code (cannot enter).

Entry section code of P<sub>2</sub> will pass only after Exit section of P<sub>1</sub>.

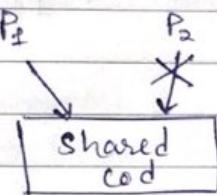
Thus, process synchronization can be achieved.

Eg, using Semaphore, ~~lock~~ ~~unlock~~

4 conditions for synchronization :-

- Primary Mandatory*
- 1.) Mutual Exclusion
  - 2.) Progress
  - 3.) Bounded wait
  - 4.) No assumption related to H/w (hardware)

### (1.) Mutual Exclusion



If  $P_1$  is executing code inside critical section, then  $P_2$  cannot use critical section & vice versa.

### (2.) Progress

If ~~both~~ shared resources are not accessed by any process, then any one of the process can access it.

But let say  $P_1$  wants to access shared resources then  $P_2$  must not block  $P_1$  from accessing shared resources.

It is possible that, there arise a situation when no process is using CS (critical section) but other processes are blocking the other process to use CS. Thus, at in that state, we ~~can~~ reach at a deadlock situation where no process can use CS.

(3.) Bounded wait :- It should not happen that only one process is using CS, and  $P_2$  never gets its chance.

1.) It should work on

2.) Synchronization must not be limited to itself due to hardware constraints.

For e.g., code will work for ~~16~~<sup>some</sup> OS or 32-bit system & will not for other OS or 64-bit system. This should not happen.

## \* Lock Variable

⇒ Initially lock = 0

1. While ( $lock == 1$ );  
2. lock = 1

Entry  
code

1. While ( $lock == 1$ );  
2. lock = 1.

3. Critical Section

3. Critical Section

A. lock = 0

Exit  
code

4. lock = 0

(P<sub>1</sub>)

(P<sub>2</sub>)

It - Initially lock = 0,  
thus P<sub>1</sub> can go inside critical section &  
 $lock = 1$ .

- As  $lock = 1$ , P<sub>2</sub> will end up in infinite loop if it tries to access critical section.
- Once P<sub>1</sub> has executed exit code,  
(i.e.,  $lock = 0$ ), only then P<sub>2</sub> can go inside critical section.

Special Case :- If  $P_1$  pre-empts after executing only stmt. 1 (i.e., not execute 2.) then  $P_2$  will ~~also~~ be able to enter inside critical section (as lock is not set to 1 yet.). And if  $P_1$  resumes, both  $P_1$  &  $P_2$  will be accessing CS. Thus, in this case, it does not guarantees mutual exclusion.

To achieve mutual exclusion with ~~diff~~

The problem here is that stmts 1 & 2. are ~~diff~~ different statements.

If we combine them to one, then we can overcome the issue.

For this we write a function "test-and-set"

: boolean test-and-set (boolean & lock) {

    boolean ~~&~~ = lock;

    lock = true;

    return ~~&~~;

} Initially lock = false

: | while ( test-and-set (lock) ): } | Entry code

Critical Section

| lock = false |

Exit code

## \* Turn Variable (Strict Alteration)

 $P_1$ while ( $\text{turn} \neq 0$ )

critical section

turn = 1

 $P_2$ while ( $\text{turn} \neq 1$ )

critical section

turn = 0;

- This ~~allows~~ satisfies mutual exclusion.
- But does not satisfies progress, ~~& Bounded wait~~.

If  $\text{turn} = 0$ , then only  $P_2$   $P_1$  can go inside critical section. But if  $P_1$  refuses to go inside critical section. & and execute the exit code (of  $P_1$ ), then  $P_2$  will not be able to move forward.

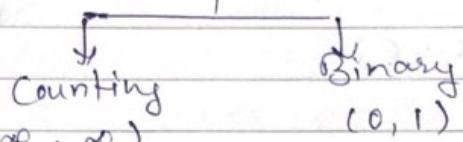
Thus,  $P_1$  is not allowing  $P_2$ 's progress.

- After ~~execute~~ complete execution  $P_1$ ,  $P_2$  has to be completely executed for  $P_3$  to be re-executed. Thus, both processes will get their turns. Thus, it satisfies bounded wait.

## \* Semaphore :

Is an integer variable, used in mutual exclusive manner to achieve synchronization.

### Semaphore types



Entry code : down, wait

Exit code : up, signal

wait ()  
wait (semaphore s)

signal ()  
signal (semaphore s)

s.value = s.value - 1;

s.value = s.value + 1;

if (s.value < 0) {

if (s.value ≤ 0) {

Put process in

~~select a process~~

suspend list

from suspend list &

sleep ();

wake up ();

else

else

return;

return;

Binary semaphore :

```

Down (semaphore s) {
    if (s.value == i) {
        s.value = 0;
    }
    else {
        Block this process
        & place in suspend
        list; sleep ();
    }
}

up (semaphore s) {
    if (s.value == 0) {
        if (suspend list is
            empty) {
            s.value = 1;
        }
        else {
            select a process from
            suspend list & wakeup
        }
    }
}

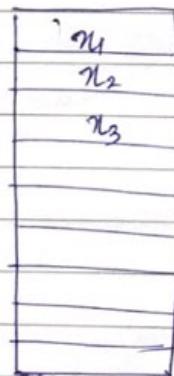
```

### \* Producer Consumer Problem

```

Producer () {
    if (buffer not full)
        produce ();
    count = count + 1;
}

```



Shared Buffer

```

Consumer () {
    if (buffer not empty)
        consume ();
    count = count - 1;
}

```

**Problem :**

- Let say buffer already has 3 items.
- producer produces an item & store it into buffer. But, before it can increment count, it gets pre-empted.
- Now buffer has 4 items, but consumer only sees 3 items. So, after complete execution of both the processes, there will be incorrect count value.

\* **Solution using Semaphore :**

~~Producer () {~~

Lets take 4 semaphores  
 ↓ down Empty, full  
 ↓ Counting -  
 ↑ S Binary

Lets take 3 semaphore values :

Empty, full → Counting Semaphores  
 S → Binary semaphore

Producer () {

1. down(Empty);
2. down(S);

Produce item  
 (critical section)

3. up(S);
4. up(Full);

Consumer () {

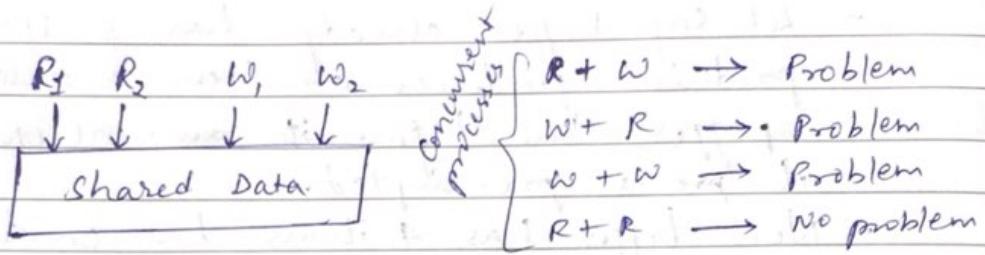
1. down(Full);
2. down(S);

Consume item  
 (critical section)

3. up(S);
4. up(Empty);

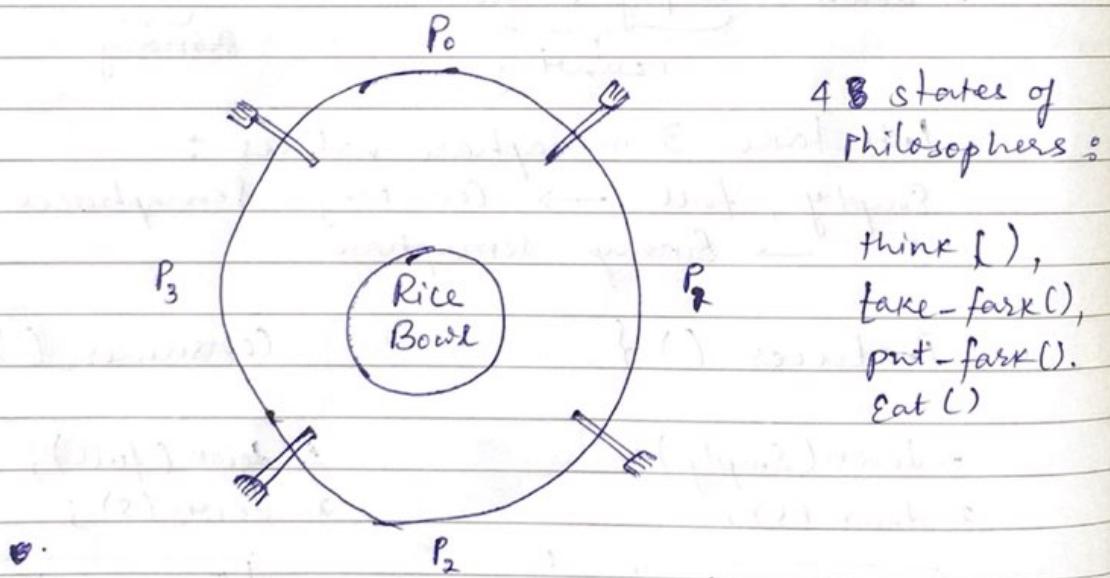
So, here, we can switch context any time, but after the complete execution of both the processes, there will always be correct count of items.

## \* Reader Writer Problem :



~~Eg.~~ Eg. Transactions in database  
Can be solved by semaphore.

## \* Dining Philosophers Problem



This can be implemented using semaphore.  
consider take-fork

We can keep an array of size = no. of forks as semaphore values for fork -

if value = 1, means fork is on the table

if value = 0, means " ", lifted

```
void philosopher () {
```

```
    while (true) {
```

1. thinking ();

2. take-fork (i); // wait

3. take-fork ((i+1)%N); // wait

4. Eat ();

5. put-fork (i); // signal

6. put-fork ((i+1)%N); // signal

```
}
```

```
}
```

- take-fork(i) will decrement the semaphore value at  $i^{th}$  position.
- This way only 2 philosophers can eat at the same time.

DeadLock :- In this code (even using semaphores) it is possible that all ~~philosophers~~ philosophers ~~will~~ execute only till stmt. 2., and preempt. ~~1.~~

At this condition, no philosopher will be able to pick another fork (i.e., execute stmt. 3) And we will be stuck forever in that situation.

It is called DEADLOCK.

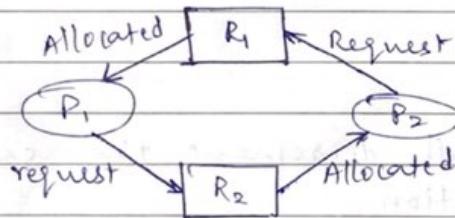
## \* Deadlock :-

If two or more processes are waiting for something to happen, but it never happens. The situation is called deadlock.

### 4 Necessary conditions :-

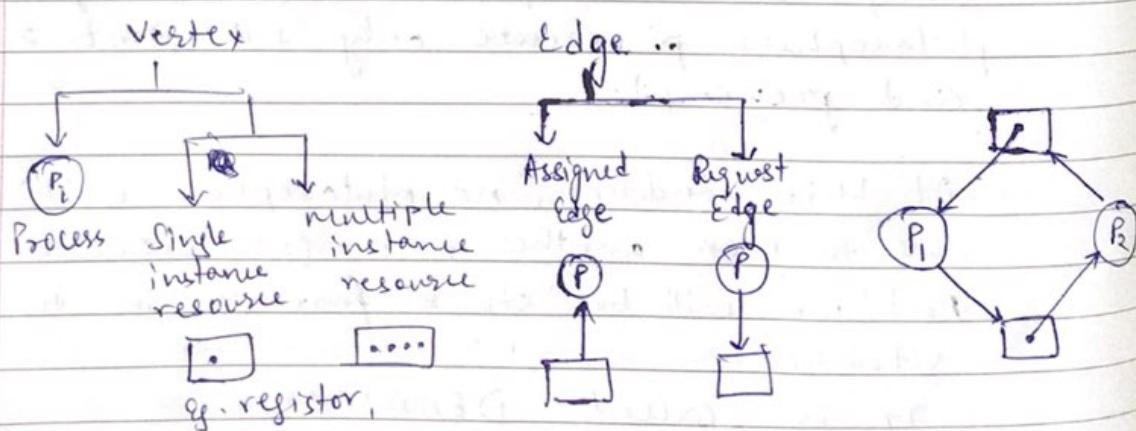
- 1.) Mutual Exclusion
- 2.) No pre-emption
- 3.) Hold & wait
- 4.) Circular wait

If a process is holding a resource, it cannot pre-empt or release the resource. Only then deadlock is possible.



## \* Resource Allocation Graph

need to represent a state of the system.



## \* Deadlock handling & Prevention

### 1.) Deadlock ignorance

- just ignore the deadlock. If the deadlock occurs, just restart CPU.
- Widely used.
- Because, deadlock occurs very very rarely.
- If we write the code to prevent deadlock, then efficiency of execution of code decreases.  
So our OS will be slower.
- windows, linux uses this.

### 2.) Deadlock prevention

- Try to remove, at least 1 of the 4 necessary condition for deadlock
  - Try to share resources
  - Try to prevent
  - Simply provide all resources a process is demanding

All these are practically ~~not~~ extremely difficult to implement. So these are not practical approaches.

### 3.) Deadlock avoidance

- Before allocating resources, check whether is it safe to allocate or not

### 4.) Deadlock detection & recovery

- Kill the processes (or few processes)

Question: A system is having 3 processes such that all processes require 2 instances of resource 'R'. Calculate / Find the minimum no. of R resources required for all the processes to be executed without any deadlock.

Ans. If every process get 2 separate instances of R, then ans will be  $= 3 \times 2 = 6$ .  
But this is not minimum number.

⇒ Let's see from by taking 2 Rs.

P<sub>1</sub> P<sub>2</sub> P<sub>3</sub> Deadlock occurs.

| | |

⇒ If we have 3 Rs.  
Then except for the following instance, all the other instances will execute all 3 Processes.

P<sub>1</sub> P<sub>2</sub> P<sub>3</sub>

| | |

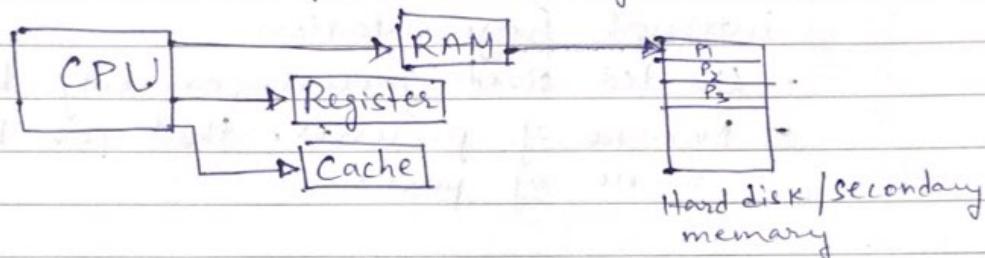
⇒ If we have 4 R's

P<sub>1</sub> P<sub>2</sub> P<sub>3</sub> ← 1 Resource can be given to any  
1 1 1 of the process, and once given,  
the process will be executed fully  
and there won't be. will be  
2 new free resources R, which  
can be used by other 2 processes.

Thus, ~~ans~~ ans is 4.

# \* Memory Management

Goal - ~~to~~ Efficient utilization of memory  
(Can be primary or secondary)

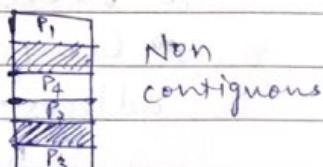
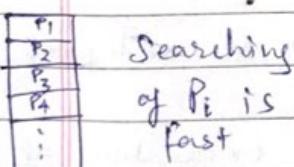
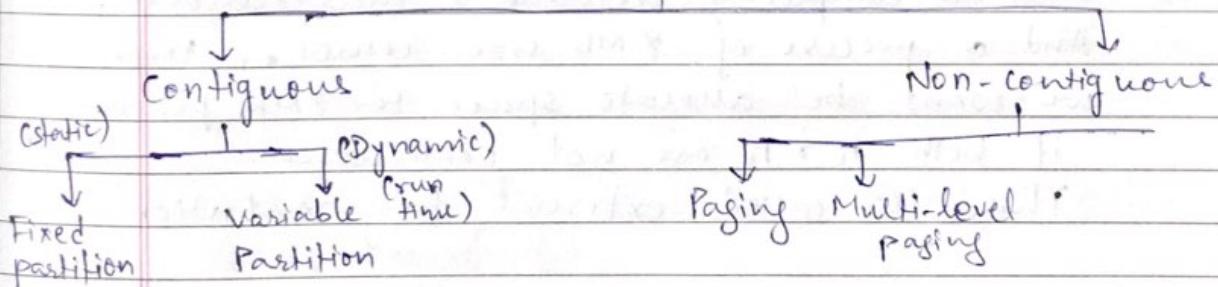


CPU does not have direct access to secondary memory. And processes are stored in Secondary memory.

So, we try to bring as many processes as we can to RAM. (degree of multi-programming)

## Memory Management

### Techniques:

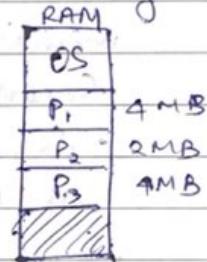


## (1.) Fixed size partition

- Put <sup>only</sup> 1 process in 1 partition, and no more
- Thus, in every partition, there will be some space wasted. This is called internal fragmentation.
- limited sized process can only be allocated
- Max. no. of processes that can be inserted = no. of partitions.

## (2.) Variable or dynamic partitioning

- Contiguous
- No chance for internal fragmentation
- No limitation on no. of processes or process size.



Problem :- Let say 2 processes P<sub>1</sub> & P<sub>3</sub> of size 4 MB are completely executed & thus removed, And a process of 8 MB size comes, then we won't be able to allocate space to 8 MB process if both P<sub>1</sub> & P<sub>3</sub> are not contiguous.

This is called external fragmentation.

To overcome :-

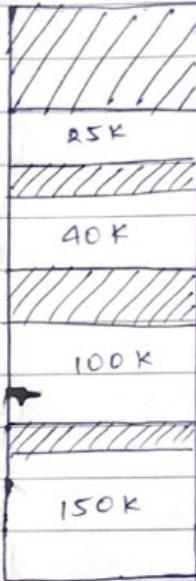
⇒ Compaction :- Copy & paste it add prev. address of P<sub>3</sub>.

- But we will need to stop execution of P<sub>2</sub>.
- Time consuming.

## Algorithms for contiguous memory mgmt.

(1.) First fit :- Allocate the first hole that is big enough.

- fast
- But can lead to high left over space (High internal fragmentation)



(2.) Next-fit :- Same as first fit, but starts from the last allocated process

- Even faster
- But same problem.



(3.) Best fit :- Allocates to the ~~best~~ smallest hole that is big enough to hold the process

- least internal fragmentation
  - But needs searching, thus slow
  - It is also possible that it can create very tiny-tiny holes that cannot allocate ~~any~~ process.
- Thus, some space will be wasted.

(4.) Worst fit :- (Opposite of Best-fit) Allocates to the largest hole

- Now the hole will be big enough to hold any other process (as ~~to~~ it will provide ~~be~~ highest contiguous left over space).
- But still slower

## \* Non-contiguous Memory Allocation

The process is divided into many segments which are called pages.

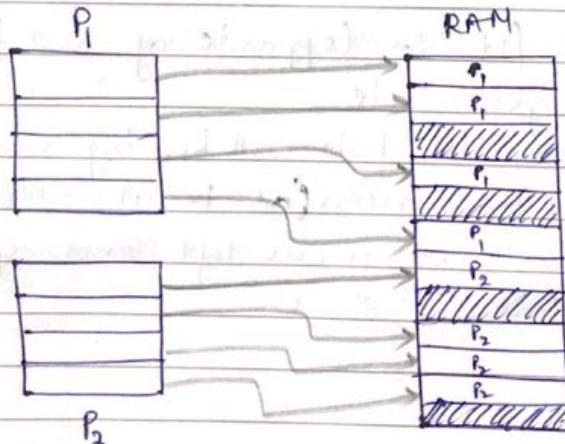
Now these pages can occupy spaces efficiently in RAM.

The holes are created dynamically (run-time), so to divide a process into pages dynamically is very costly. So we divide process into fixed size pages before bringing it to RAM.

We also divide RAM into fragments / frames

size of page = size of fragment / frame

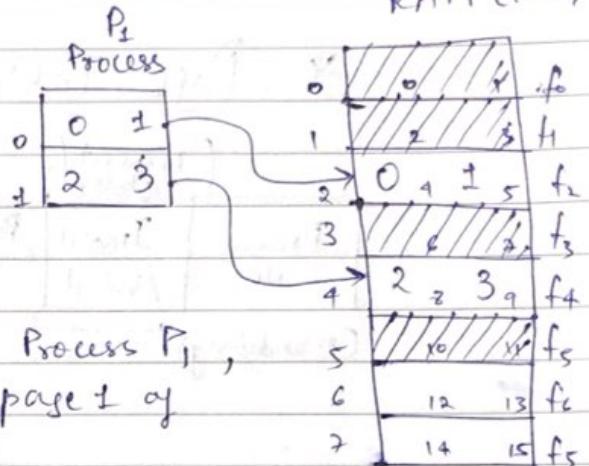
This way, memory will never be wasted.  
(internal fragmentation will never occur.)



## \* Paging

Let say we have a process of size 4 bytes.

While executing Page 0 of Process P<sub>1</sub>, CPU might need data from page 1 of the same process.



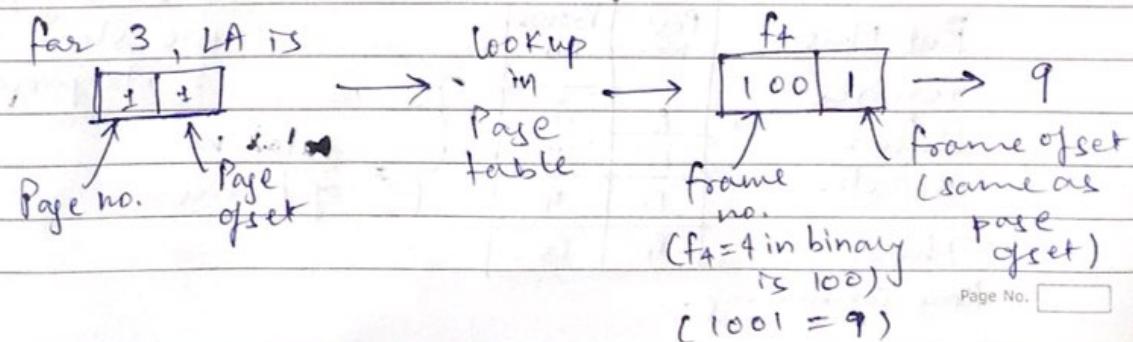
But page 1 can be located anywhere in the RAM.  
So, we will need to create some kind of mapping that tells us which data (byte) is located at which physical address of RAM.  
That mapping is done using "page table".

Page table of P<sub>1</sub>,

0	f <sub>2</sub>
1	f <sub>4</sub>

Now if CPU asks for data (byte) 3, then we can look up that data in RAM using page table.

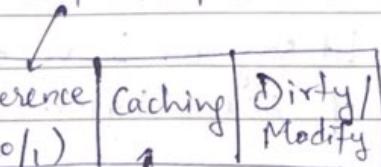
Also CPU will ask in logical address, so that is needed to be converted into physical address of RAM.



## \* Page Table Entry

(Present/ absent)		Frame No.	Invalid /Valid	Protection R/W/X	Reference (0/1)	Caching	Dirty/ Modify
(mandatory)				optional			

useful for page replacement



modified by user in Primary memory as not?

If the information is frequently needed, then we can keep it in cache.

There is 1 page table for 1 process.  
Page tables are present in hard disk.

But if any of the page is present in main memory, then its entire page table should also be in main memory.

So, if 10 pages of 10 different processes are present in main memory, then all the 10 page table will also be present in main memory, (which is space consuming)

## \* Inverted paging

It only keeps a global page table for all processes.

But this requires linear search.

Thus,

time consuming;

Page No.	Process Id
P <sub>0</sub>	P <sub>1</sub>
P <sub>1</sub>	P <sub>2</sub>
P <sub>2</sub>	P <sub>1</sub>
P <sub>3</sub>	P <sub>3</sub>

For every frame, we start page no. & its process id.

= total no. of frames

## \* Thrashing

For a single process, not all its pages are brought to the main memory. So, for every processes, it is possible that, only few pages of those processes are available.  
(Concept of Virtual Memory)

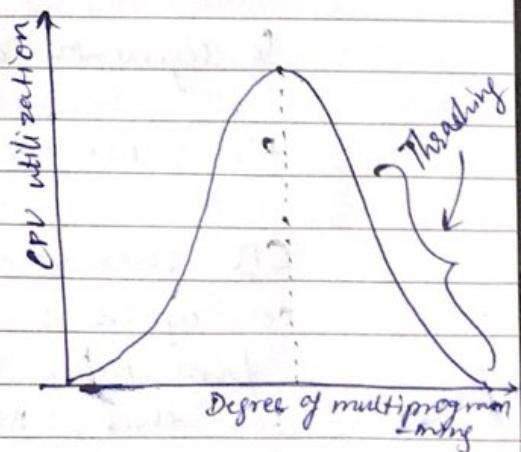
So when a CPU tries to find let say page 2. of Process 1 and it doesn't find it, then it is called page fault.

When we try to maximize the no. of pages in our main memory, it is possible that only few pages of all processes are available in main memory. So chances of frequent page fault is very high.

When a page fault occurs, OS has to look for the required page demanded by CPU on hard disk. And it is very time consuming.

Thus, after certain point of degree of multiprogramming (bringing max. process <sup>in RAM</sup>), the CPU utilization is observed to be decreasing.

It is called \* Thrashing.



## \* Segmentation

Segmentation is also a process of dividing a process into different segments.

In paging, we divide a process without knowing what's in it.

- So let say, when  $f_1$  is a process has a function  $f$  in it divided into two pages  $f_1$  &  $f_2$ .

- When CPU starts executing page  $f_1$ , it immediately requires  $f_2$ . So now, we will need to find  $f_2$ . (which is time consuming specially if  $f_2$  is not present in main memory.)

In segmentation, we do not uniformly divide a process, but we divide it into separate logical blocks based on user point of view.

For eg. 1 segment with of func  $f_1$  & other segment with of func  $g$ .

So segments are not of same size.

Now, we can fit all segments into RAM.

CPU can also demand for any segment, so again we will need to fetch a segment from main memory.

For this, Memory management unit (MMU) has to convert logical address given by CPU to physical address.

Thus, we need a segment table

Base Address	Size
3300	200
1800	400
1500	30

1500	OS
1700	S <sub>3</sub>
2200	S <sub>2</sub>
2400	S <sub>6</sub>
2600	S <sub>7</sub>
2800	⋮
3000	⋮

### \* Overlay

Also a process of dividing a process into different parts. But here, the user itself will need to divide the program.

No very practical when we have to implement many functionalities.

Was used in embedded systems earlier.

The parts should be independent of each other.

### \* Virtual Memory

We do not need to bring an entire process (with its pages) into main memory. We can only bring few required pages of this process.

This way, we can accommodate / execute a process without the ~~available~~ & available free space to accommodate a process.

This illusion is provided by virtual memory, that our process is present in main memory, even though its only some few parts are present.

Still used in operating system these days.

But the problem is, if CPU does

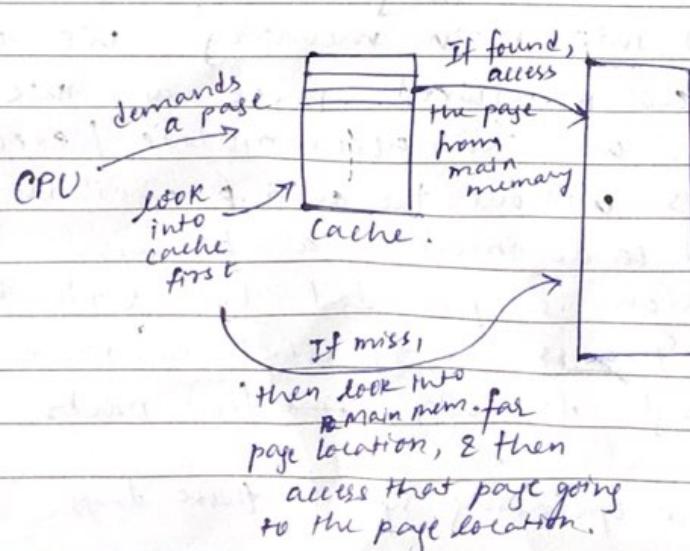
But the problem of page fault can also occur frequently.

Page fault - CPU trying to access a page which is not present in main memory.

Thus, Searching for a page in main memory only takes about order of ns time, while searching for a page in secondary memory is of the order of ms.

### \* Translation lookup table (Cache)

- Cache is even faster than main memory.
- So we can keep page table information in cache.
- But cache is limited, so we can only keep data of only few pages.



Here, if you look into main memory for the page position (using page table), then also you will need to look for that page in main memory. Page No. 22

Question :- TLB access time is 10 ns. Main memory access time is 50 ns. If TLB hit ratio is 90%, no page fault. What is effective memory access time?

$$\text{Ans.} = (0.9)(10 + 50) + (0.1)(10 + 50 + 50)$$

## \* Page Replacement Algorithm

As we want to execute maximum processes, we use virtual memory (otherwise my main mem. will be filled by few process only).

So we will need to Swap-In & Swap-Out pages from main mem. to sec. mem.

3 methods :

- 1.) FIFO
- 2.) Optimal page Replacement
- 3.) Least Recently used (LRU)

FIFO Motive is also to minimize page faults.

\* FIFO Reference string (CPU demand) :

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

		1	1	1	1	0	0
frame	f <sub>3</sub>	0	0	0	3	3	3
frame	f <sub>2</sub>	7	7	7	2	2	2
frame	f <sub>1</sub>	7	7	2	2	2	4

We replace page based on FIFO.

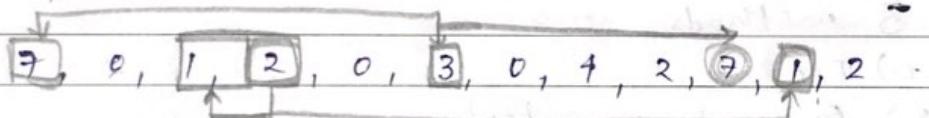
~~It is observed that~~

logically, if we increase no. of frames, then we should see decrease in no. of page faults.

But in FIFO (only in FIFO) it is observed that there are cases, when increasing the frame size also increases no. of page faults for a given reference string. It is called Belady's anomaly.

### \* Optimal Page Replacement

We will replace the page ~~which is~~ which is not in demand for the longest time in future.



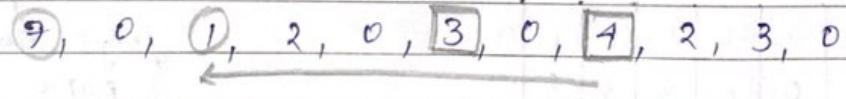
If frame size is 3.

- 1 will be replaced by 2
- 7 will be replaced by 3

### \* LRU

we replace the page which was least recently used in past.

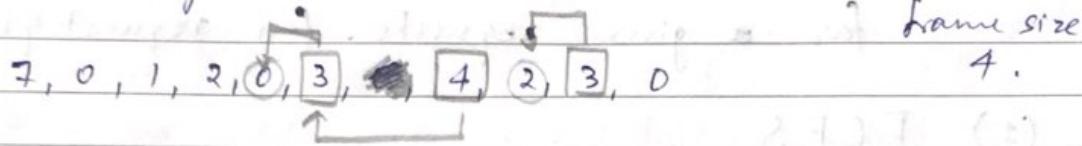
Let say  
frame size  
is 4.



- 7 will be replaced by 3
- 1 " " " " 4.

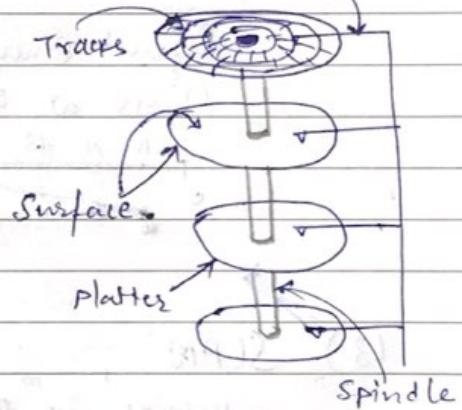
### \* Most Recently Used

- We replace the page which was most recently used.



### \* Disk Architecture

Platter → Surface → Track  
(upper & lower)  
→ Sector → Data



\* seek time :- Time taken by read/write head to reach desired track.

\* Rotation time :- Time taken for 1 full rotation.

\* Rotation latency :- Time taken to reach desired sector.

## \* Disk Scheduling Algorithm :

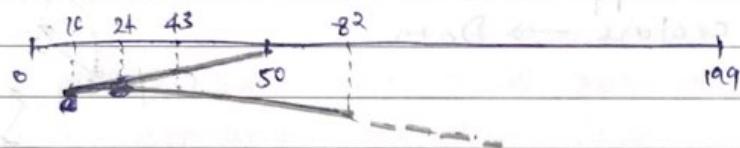
Is used to minimize the total seek time for ~~all~~ given requests. (in request queue)

### (1.) FCFS

- No problem of starvation
- High seek time

### (2.) Shortest Seek time first (SSTF)

Request Queue : 82, 170, 43, 140, 24, 16, 190  
Starts at 50.



### (3.) SCAN

- moves in one direction till its end.
- starting moving in ~~direction~~ larger direction.

### (4.) LOOK

- Similar to SCAN
- Move till the largest / smallest value in that direction

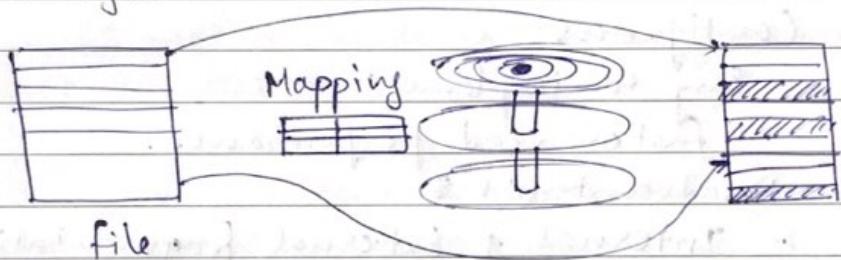
### (5.) C-SCAN

- Move in one direction till the end (larger direction)
- Come back to zero (0), without handling any request.
- Start handling request from 0, in 1 direction.

(E.) C-LOOK :- C-SCAN with LOOK.

## \* File System in OS

- Users deal with files & folders.
- The data of the files for me is needed to be stored in hard disk.
- File system stores data into harddisk & manages it.



~~Data~~ Data divided into parts & stored in disk.

Need to maintain mapping of storage

## \* File Attributes & Operations

Attributes :- Name, extension, location, size, ~~prot~~ permission, encryption, etc., last modified, etc.

Operations :- Create, read, write, delete, truncate.

(truncate - will delete content of file but will preserve attributes).

Attributes info has to be stored in hard disk only.

## \* File Allocation

Contiguous      Non-Contiguous

(Easy, faster read performance) → Linked List Allocation  
 (Internal fragmentation) → Indexed Allocation

Purpose :-

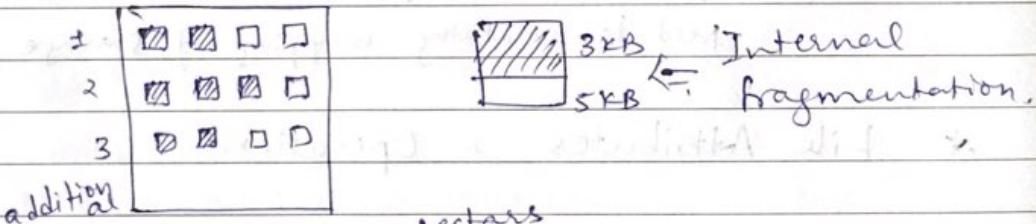
- Efficient disk space utilization.
- Faster access time.

### \* Contiguous

- Easy to implement.
- faster read performance.

Disadvantages :-

- Internal + External fragmentation



If a file requires 4 blocks, then we cannot allocate it to track 1, 2, 3. Even though there is space empty but are not contiguous.

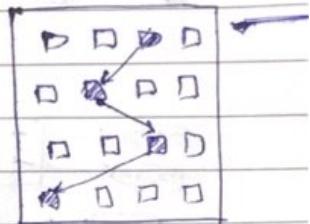
- Here, we already define the file size and put everything contiguously.

If we increase the file size, then there is only a limit till we can add data, coz only limited space is available next to the file.

So, we cannot grow file size or add more data.

## \* Linked list Allocation

Data stored in linked list

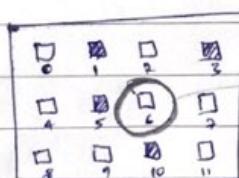


Advantages :- No <sup>external</sup> fragmentation (Internal will be there)  
File size can be increased.

Disadvantages :- Larger seek time, as r/w head will need to make a lot of movement to move to the next head.

- Direct or random access is not possible.  
Can only start looking from the beginning of file.
- Needs additional ~~dat~~ space to store pointer data.

## \* Indexed file Allocation



Adv.- Direct Access

- No ~~frag~~ external frag.

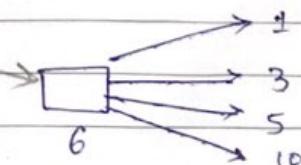
Disadv :- Pointer/index overhead.

- Multilevel indexing

( It is possible that the file size is too large to store in 1 sector. So, its data will be needed to store in multiple sectors. )

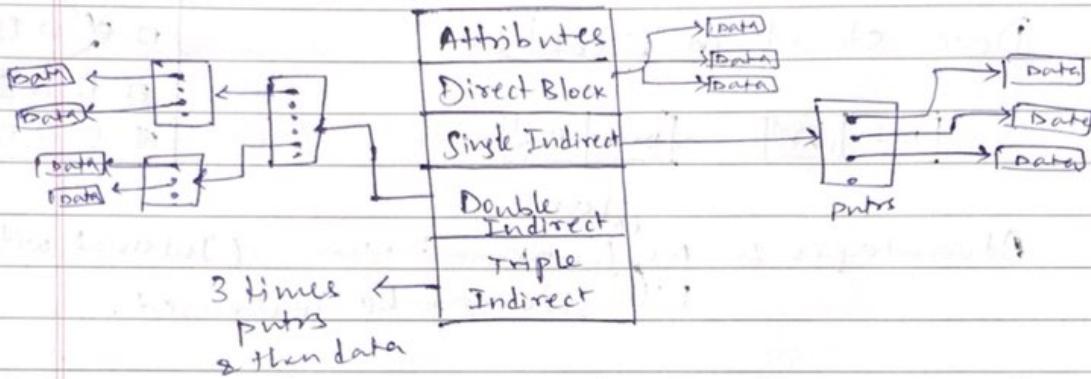
Dictionary

File	Index block & other attr.
A	6 -----



Address of diff. blocks of a file will be in 6<sup>th</sup> sector.

## \* Unix I-node Structure ( i - far index ) \*



Every block will start the following structure

inode of size 128 bytes