Date __/__/____

Saathi

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 mngt.

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.

Saathi

Date __/__/____

* Types of OS

1.) Batch OS
- operations performed in batches

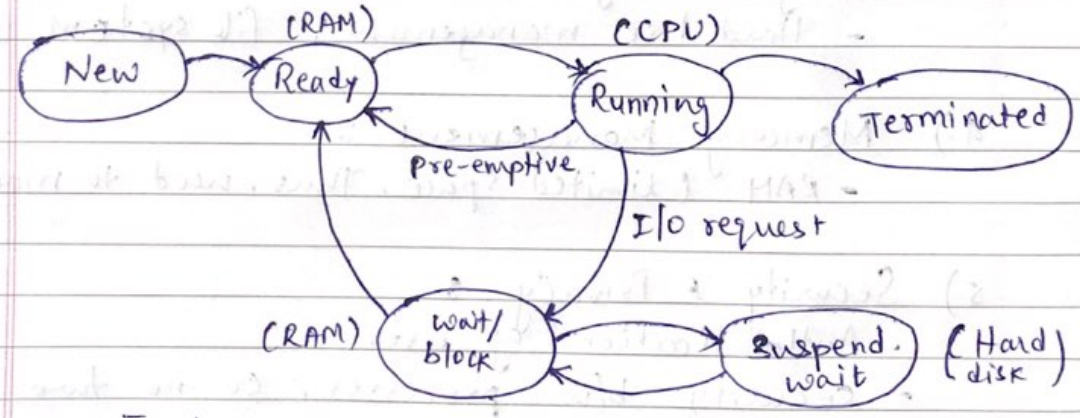
2.) Multi-programmed OS
- Aim is to get max. no. of processes in RAM.
- Non-preemptive

3.) Multi-tasking / time sharing OS
- Pre emptive. 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.
* Pre Emptive :- Process is executed in parts.



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

Page No.

Date __/__/____

- 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) untill 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 kernal 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

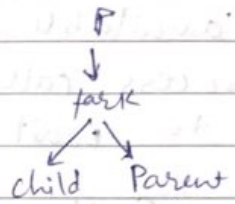
Communication related :- pipe(), create/delete

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

Date __/__/____

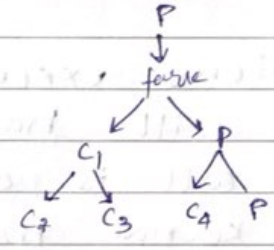
* Fork():
  - to create a child process.
  - clone of parent process
  - used for multiprocessing

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



Output:
- hello
- hello

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



output:
- hello
- hello
- hello
- hello

- for 3 times fork, 8 times hello will be printed.

* User mode v/s Kernal mode

Hardware, devices are controlled by Kernal 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 (Kernal mode gives result) from bank (Hardware)

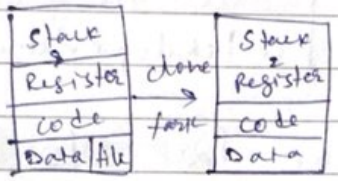
-

Date __/__/____

Saathi

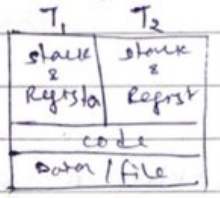
* Process v/s Threads
(For multi-tasking & multi-processing)

(child)
• Process is created using (fork) system call.



P_1 P_2

Both process can perform a task for me parallely



Threads access the same data & code to execute tasks parallely.

1.) System call involved in process	1.) 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 & thus high overhead.	3.) Same copies, Threads share same copies & thus, no to high overhead
[- Threads can be created at API level or by OS (both) at Kernal level]	
4.) Context switching is slow (heavy weight)	4.) Context switching is fast (light weight)

CPU will execute a process 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_1 needs to be saved before executing P_2. Thus, is highly time consuming. While switching context in threads are easy as it shares the same data.

Page No.

Saathi

Date ___ / ___ / ___

5.) Blocking a process will NOT block another process and thus processes are independent	5.) Blocking a thread will block (all other ~~thread~~ threads of) an entire process. Thus they are interdependent.

Blocking a process can be for I/O. call.

* User level Thread v/s Kernal Level Thread

1.) User level threads are managed by user level library (API)	1.) Kernal level threads are managed by: OS via system call.
2.) Faster	2.) Slower (as system calls are involved)
3.) Context switching is faster	3.) CTS is slower
4.) If 1 user level thread performs blocking operation, then entire process is blocked (because kernal will block the entire process)	4.) Blocking 1 kernal level thread does not affect others, because kernal knows which thread is blocked & as these threads are managed by kernals.

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

CTS of process > CTS of Kernal threat > ~~th~~ CTS of User Thread.

Date __/__/____

Saathi

* Process are basically programs which are dispatched from ready state for execution (in CPU)

* Threads is a segment of a process.
Process can have multiple threads.

* Process Scheduling Algorithms

Pre-Emptive	Non Pre-Emptive
- Partial process can be executed before running another process.	- 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)

Page No.

Saathi

Date ___/___/_____

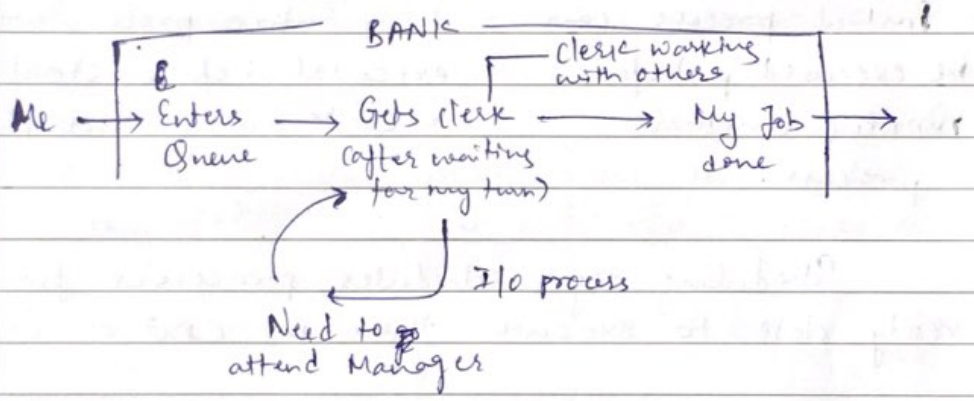
* Turn around time = Completion time - Arrival time

* Waiting time = Turn around time - Burst time

* Response time = (The time at which the ~~CPU~~ process reaches the CPU for the first time) - (Arrival time)
(sys. 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~~ ~~first~~

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

Page No.

Date ___/___/_____

Saathi

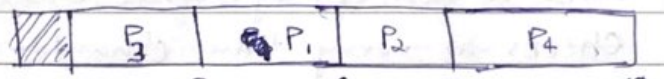
* Shortest Job First (Non-PE)

A 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_1	1	3	6
P_2	2	4	10
P_3	1	2	3
P_4	4	4	14

Gantt chart



P_1, P_3 P_3, P_1 P_2, P_4

~~Also note, for pre~~

* Shortest Remaining Time first (SRTF) (PE)

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

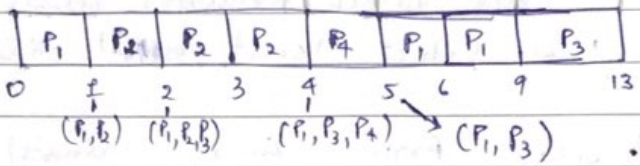
Selects a process at every unit of time, based on remaining burst time & chooses the process with least remaining burst time.

Saathi

Date ___/___/_____

Pid	AT	BT	CT	TAT	WT	RT
P_1	0	5 4 3	9	9	4	0
P_2	1	3 2	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	1	5	1	0	0

Gantt chart

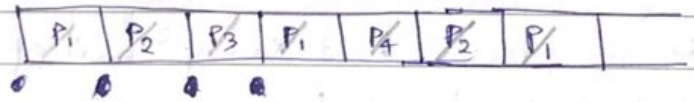


* Round Robin

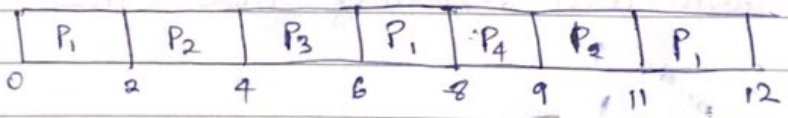
- Uses a concept of time Quantum.
- ~~Checks at every time Quantum.~~
- ~~Process is executed for time Quantum or until amount is then next process is selection is made.~~
- ~~The process which came first in ready Queue will be selected.~~

Pid	AT	BT	CT	TAT	WT	RT
P_1	0	8 7 1	12	12	7	0
P_2	1	4 3	11	10	6	1
P_3	2	2 1	6	4	2	2
P_4	4	1	9	5	4	4

Ready Queue



Running Queue



(Needs ready Queue for proper calculation)

Page No.

Date ___/___/_____

- Process is executed for time quantum amount or untill 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
- 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)

(No., ↑ priority)

Priority	Pid	AT	BT	CT	TAT	WT
10	P_1	0	5	13	13	8
20	P_2	1	4	9	8	4
30	P_3	2	2	4	2	0
40	P_4	4	2	6	2	0

Gantt chart

P_1	P_2	P_3	P_3	P_4	P_2	P_1

0, 1 (P_1, P_2), 2 (P_1, P_2, P_3), 3, 4 (P_1, P_2, P_3, P_4), 6, 9, 13

Page No. ____

Saathi

Date ___/___/_____

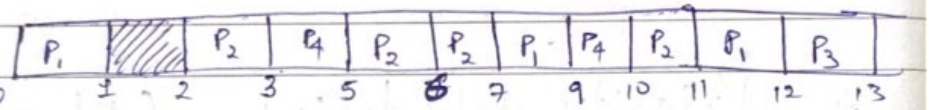
* Example of Mix Burst time :

Mode :- PE

Criteria :- Priority

Priority	P id	AT	CPU	I/O	CPU
20	P_1	0	1	5	3 1
30	P_2	2	3 2	3	1
10	P_3	3	2	3	1
40	P_4	3	2	4	1

Gantt Chart



0 - P_1 | 3 - P_2, P_3, P_4 | 6 - P_1, P_2, P_3

1 - 0 | 4 - | 7 - P_1, P_3

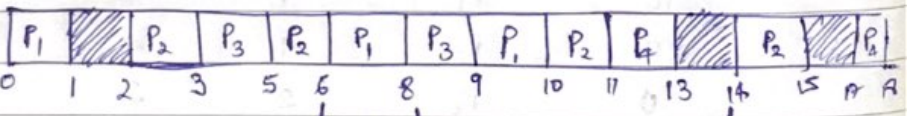
2 - P_2 | 5 - | 8 -

9 - P_4, P_1, P_3 | 10 - P_2, P_1, P_3 | 11 - P_1 P_3

Taking ↓ no. as ↑ priority.

Priority	P id	AT	CPU	I/O	CPU
20	P_1	0	1	5	3 1
30	P_2	2	3 2 1	3	1
10 H	P_3	3	2	3	1
40	P_4	3	2	4	1

Gantt chart



$P_1 \rightarrow 6$ (P_1, P_2, P_4) ($A \parallel P_3$) (P_2)

$P_3 \rightarrow 8$

$P_2 \rightarrow 14$

$P_4 \rightarrow 17$

Page No.

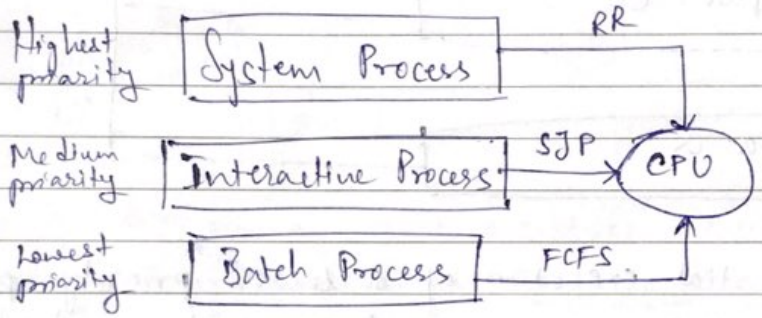
Date ___/___/_____

Saathi

* 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 that turn after a long or~~ need to wait for a long time.
- It is called Starvation.

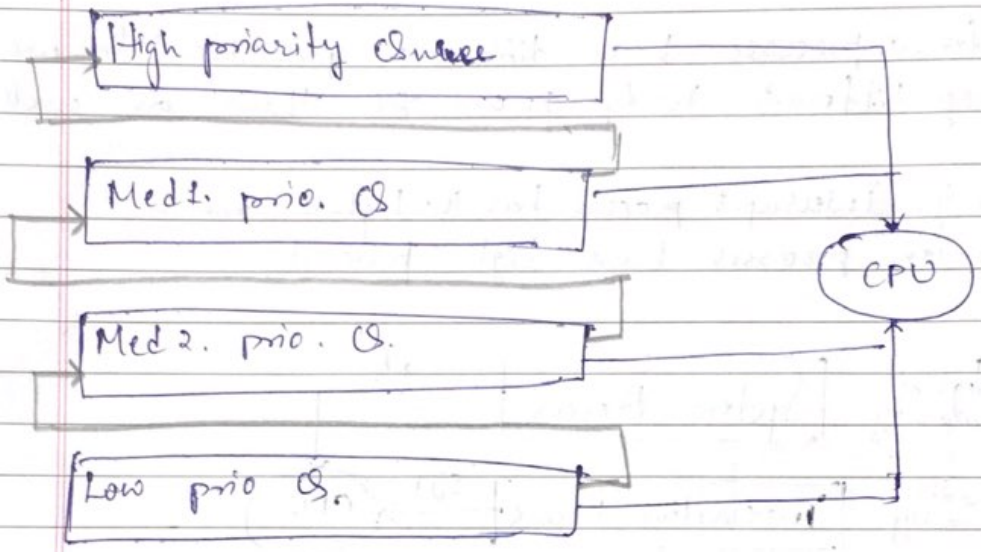
- To ~~re~~ overcome starvation, we use multi-level feedback Queue Scheduling

Page No.

Date ___ / ___ / _____

Saathi

* 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.

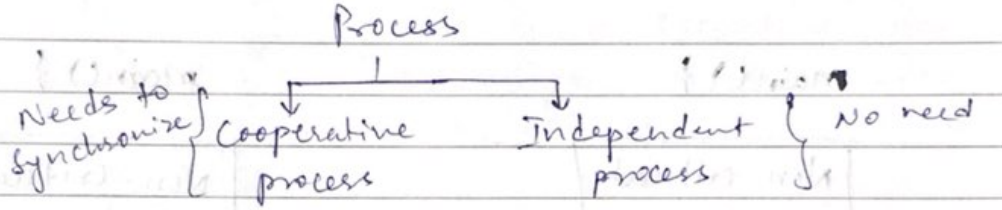
Page No.

Date __/__/____

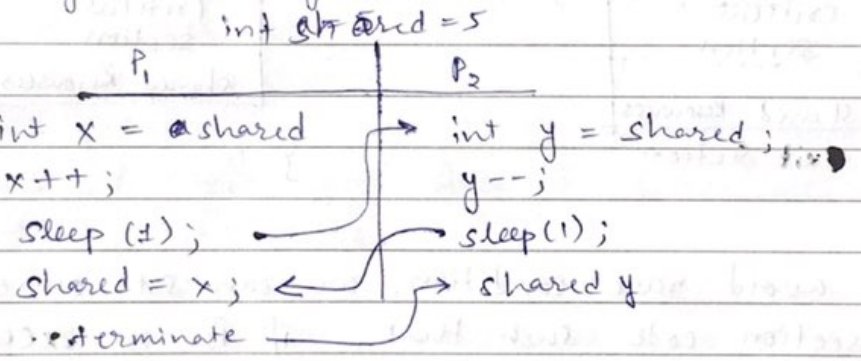
Saathi

* 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_1, and other by starting at P_2.

This is called Race condition.

(Also, Side Note :- During context switching from P_1 to P_2, the states of P_1 needed to be saved, so that its remaining execution can be ~~completed later~~ resumed later.)

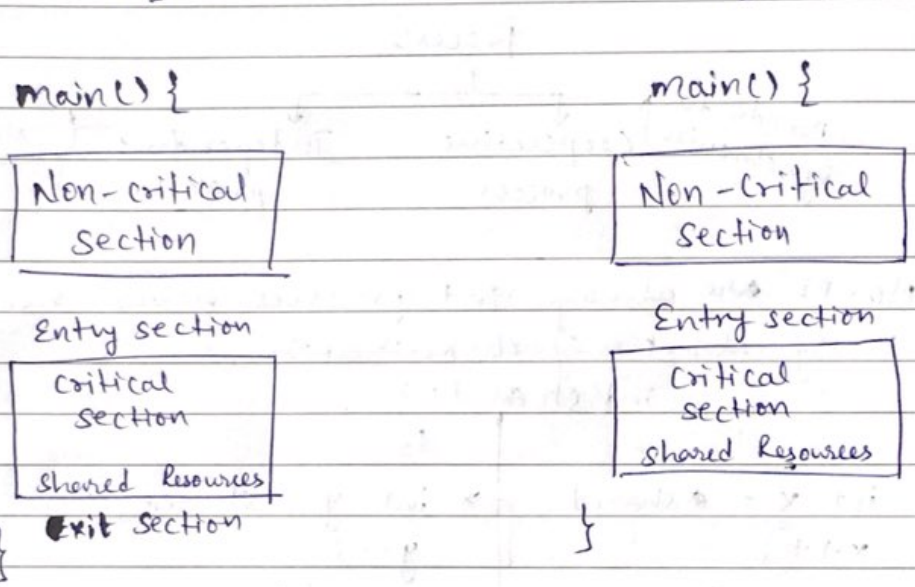
Page No.

Saathi

Date ___/___/_____

~~Producer Consumer Problem~~

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



To avoid race condition, we can set an entry section code such that, if P_1 is executing critical section, then P_2 cannot ~~en~~ execute critical section code (cannot Enter).

Entry section code of P_2 will pass only after Exit section of P_1.

Thus, process Synchronization can be achieved.

Eg, using semaphore, ~~binary~~

Page No.

Date __/__/____

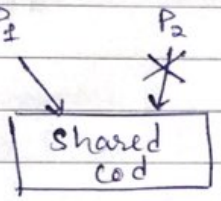
Saathi

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~~ ~~all~~ 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 ~~each~~ reach at a deadlock situation where no process can use CS.

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

Page No.

Date ___/___/_____

4.) ~~It should work on~~

4.) Synchronization must not be limited ~~to~~ ~~hard~~ itself due to hardware constraints.

For eg, code will work for some 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);	Entry code	1. while (lock == 1);
2. lock = 1		2. lock = 1.
3. Critical Section		3. Critical Section
A. Lock = 0	Exit code	4. Lock = 0
(P_1)		(P_2)

~~If~~ - Initially lock = 0,

- Thus P_1 can go inside critical section & lock = 1.
- As lock = 1, P_2 will end up in infinite loop if it tries to access critical section.
- Once P_1 ~~is~~ has executed exit code, (i.e., lock ~~=~~ = 0), only then P_2 can go inside critical section.

Page No.

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~~

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

~~to~~ 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 r = lock;
        lock = true;
        return r;
}
```

Initially lock = false

```
while ( test_and_set (lock)) ; }     Entry code
```

Critical section

```
lock = false                         Exit code
```

Saathi

Date ___ / ___ / _____

* Turn Variable (Strict Alteration)

P_1	P_2
while (turn != 0)	while (turn != 1)
critical section	critical section
turn = 1	turn = 0;

→ This ~~allows~~ satisfies mutual Exclusion.

→ But does not satisfies progress, & ~~Bounded wait~~.

If turn = 0, then only 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 ~~execu~~ complete execution P_1, P_2 has to be completely executed for P_1 to be re-executed. Thus, both processes will get their turns. Thus, it satisfies bounded wait.

Page No.

Date __/__/____

Saathi

* Semaphore :

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

Semaphore types

- Counting: $(-\infty, \infty)$
- Binary: (0, 1)

Entry code : down, wait
Exit code : up, signal

counting semaphore

~~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 suspend list sleep ( );	~~sel~~ Select a process from suspend list & wake up ( );
}	}
else return;	else return;
}	}

Date ___/___/_____

Saathi

Binary semaphore :

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

```
up (semaphore s) {
    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;
    }
}
```

Date __/__/____

Problem :

- Let say buffer already has 3 items.
- producer produces an item & stores it into buffer. But, before it can increment count, it ~~pre~~ is 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 3 semaphores~~ &
~~1. down~~ ~~Empty~~, ~~full~~ ~~S~~
~~Counting~~ ~~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);
}
```

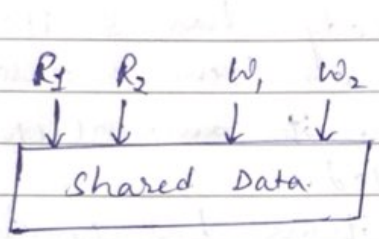
```
Consumer () {
1. down(full);
2. down(S);
   [ Consume item
     (critical section) ]
3. up(S);
4. up(Empty);
}
```

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.

Saathi

Date ___/___/_____

* Reader Writer Problem :

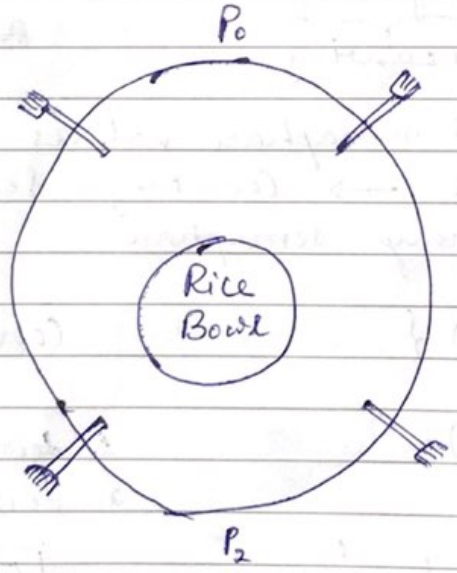


Concurrent process

R + W → Problem
W + R → Problem
W + W → Problem
R + R → No problem

Eg. transactions in database.
Can be solved by semaphare.

* Dining Philosopher Problem



4 states of philosophers :

think ( ),
take-fork ( ),
put-fork ( ).
Eat ( )

This can be implemented using semaphare.
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.

Date ___/___/_____

```
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} pasition.
- This way only 2 philosophers can eat at the same time.

DeadLock :- In this code (even using semaphares) it is passible that all philosophers 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.

Saathi

Date ___ / ___ / _____

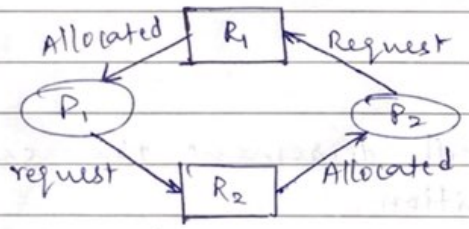
* Deadlock :-

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

4 Necessary conditions :

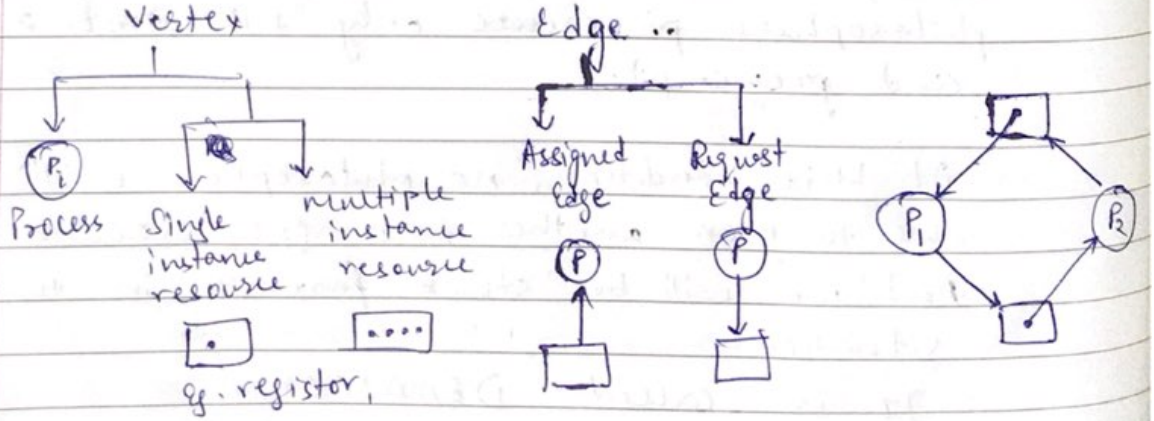
1.) Mutual Exclusion
2.) No pre-emption
3.) Hold & wait
4.) Circular wait

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



* Resource Allocation Graph

used to represent a state of the system.



Page No.

Date __/__/____

Saathi

* 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/prevent atleast 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 ~~impos~~ 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)

Page No.

Saathi

Date ___/___/_____

Question A system is having 3 processes such that all processes require 2 instance 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 seperate instance of R, then ans will be $= 3 \times 2 = 6$.
But this is not minimum number.

⇒ Lets see ~~from~~ by taking 2 Rs.

P_1 P_2 P_3 Dead lock occurs.
1 1

⇒ If we have 3 Rs.

Then except for the following instance, all the other instances will execute all 3 Process.

P_1 P_2 P_3
1 1 1

⇒ If we have 4 'R's

P_1 P_2 P_3 ⟸ 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 process.

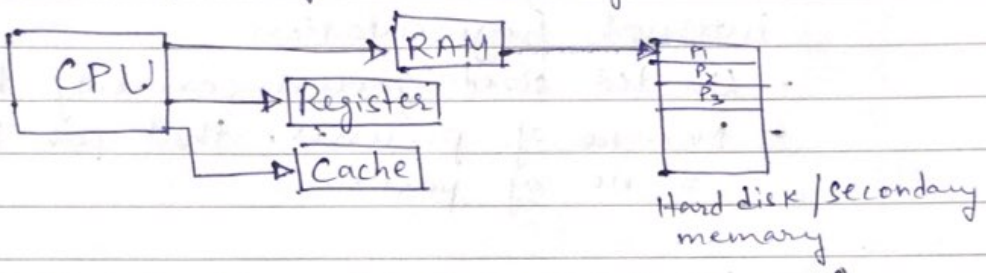
Thus, ~~this~~ ans is 4.

Page No.

Date __/__/____

* Memory Management

Goal - Efficient utilization of memory (can be primary or secondary)

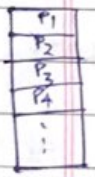


CPU does not have direct access to secondary memory. And processes are first stored in secondary memory.

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

Memory Management Techniques:

- Contiguous
  - (static) Fixed partition
  - (Dynamic) (run time) Variable Partition
- Non-contiguous
  - Paging
  - Multi-level paging

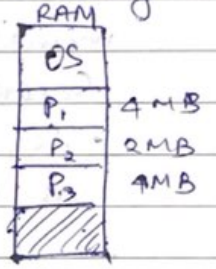


Searching of P_i is fast

Non contiguous

(1.) Fixed size partition
- Put only 1 process in 1 partition, and no more
- Thus, 1 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.



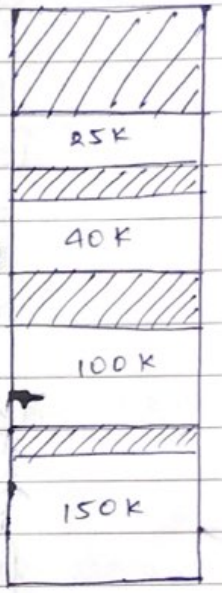
Problem :- Let say 2 processes P_1 & P_3 of size 4 Mb are completely executed & thus removed, And a process of 8 Mb size comes, then we wont be allocate space to 8Mb process if both P_1 & P_3 are not contiguous. This is called external fragmentation.

To overcome :-
⇒ Compaction :- Copy P_2 & paste it add prev: address of P_1.
- But we will need to stop execution of P_2.
- Time consuming.

Date ___/___/_____

Algorithms for contiguous memory mngt.

(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 ~~the~~ 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 :- (oppasite 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 ~~the~~ highest contiguous left over space.
- But still slower

Page No.

Date __/__/____

Saathi

* Non-contiguous Memory Allocation

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

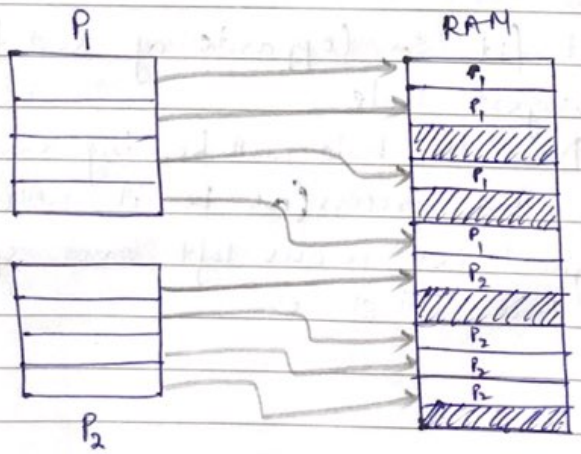
Now these pages can occupy spaces efficiently in RAM.

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.

~~Size~~ We also divide RAM into fragments / frames

size of page = size of fragment / frame

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

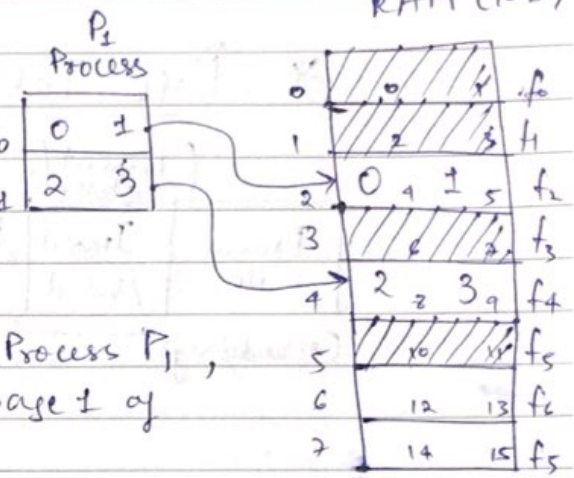


Page No.

Date ___/___/_____

Saathi

* Paging



Let say we have a process of size 4 bytes.

While executing Page 0 of Process P_1, CPU might need data from page 1 of the same process.

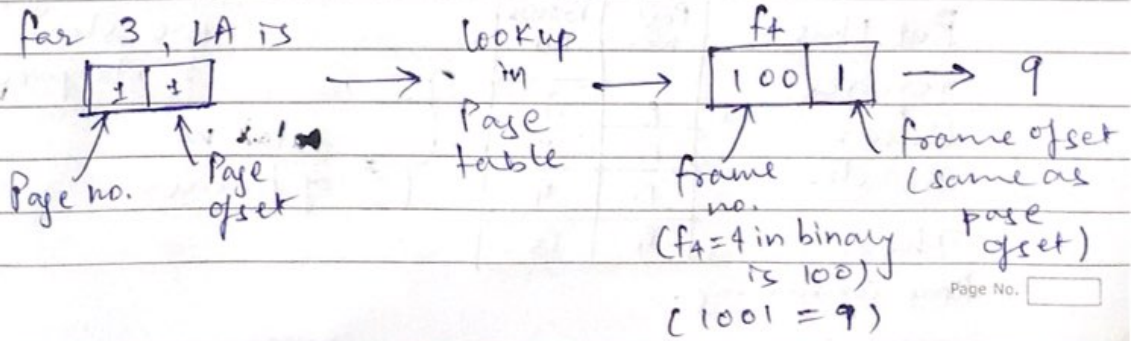
But page 1 can be located anywhere in the RAM. So, we will need to create some kind of mapping that tells us which data (byte) is located at which physical address of RAM. That mapping is done using "page table".

Page table of P_1

0	f_2
1	f_4

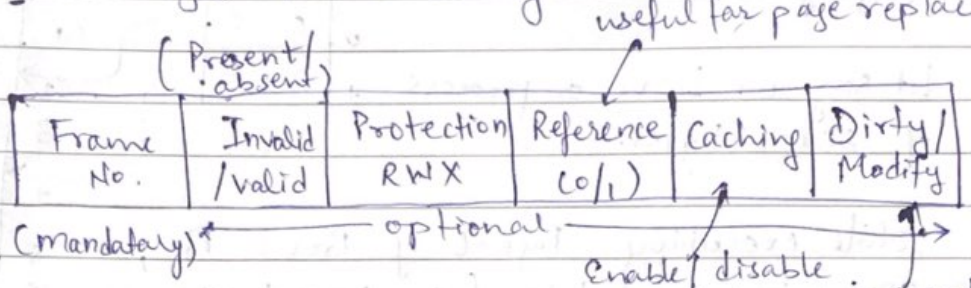
Now if CPU asks for data (byte) 3, then we can look up that data in RAM using page table.

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



Page No.

* Page Table Entry



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.

For every frame, we store Page no. & its process id.

	Page No.	Process Id
0	P_0	P_1
1	P_1	P_2
2	P_2	P_1
3	P_1	P_3

= total no. of frames

But this requires linear search.

Thus, time consuming.

Date __/__/____

Saathi

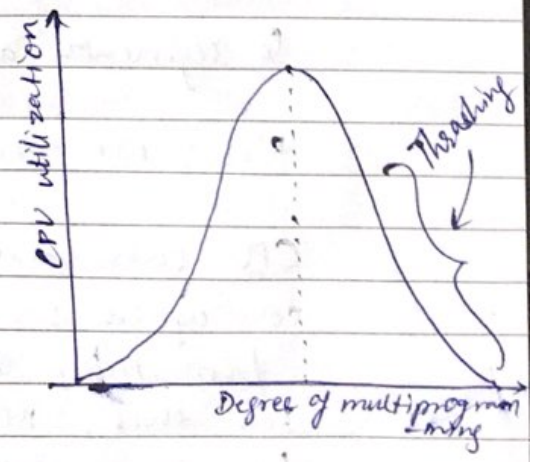
* Thrashing

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, ~~CP~~ 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 to RAM), the CPU utilization is observed to be decreasing. It is called Thrashing.



Date ___/___/_____

Saathi

* Segmentation

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

In paging, we divide a process without knowing whats in it.

- So let say, ~~when CPU~~ 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 ~~based~~ separate logical blocks based on user point of view.

For Eg. 1 segment ~~with~~ of func f & 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.

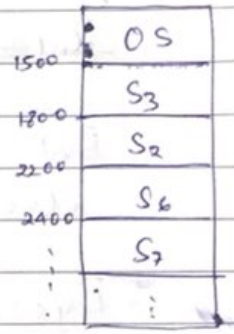
Page No.

Date ___/___/_____

Saathi

Thus, we need a segment table

Base Address	Size
3300	200
1800	400
1500	30



* 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 free space to accomodate a process.
This illusion is provided by virtual memory, that our process is present in main memory, even though its only few parts are present.
Still used in operating system these days.

Page No.

~~But the problem is, if CPU looks~~

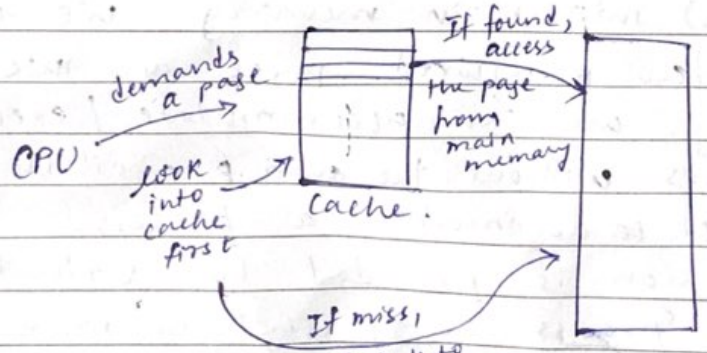
But the problem of page fault can also occurs frequently.

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

~~Thus~~, Searching for a page in ~~mem~~ main memory only takes about order of ns time, while searching for a page in secondary memory is ~~at~~ 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 ~~few~~ data of only few pages.



Here, if you ~~find~~ look into main memory for the page position (using page table), then also you will need to look for that page in main memory (~~so time 2x~~)

Date __/__/____

Saathi

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

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

frames f_3			1	1	1	1	0	0
f_2		0	0	0	0	3	3	3
f_1	7	7	7	2	2	2	2	4
	* 7	* 0	* 1	* 2	✓ 0	* 3	* 0	* 4

We replace pages based on FIFO.

Saathi

Date ___/___/_____

~~It is closer~~

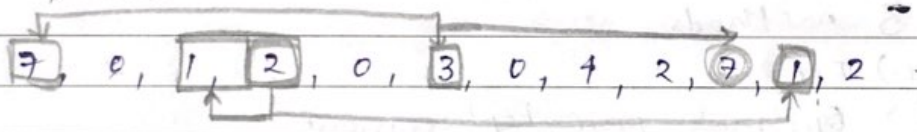
⇒ Logically, if we increase no. of frames, then we should see decrease in no. of page faults.

⇒ 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 anamaly.

6.

* Optimal Page Replacement

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



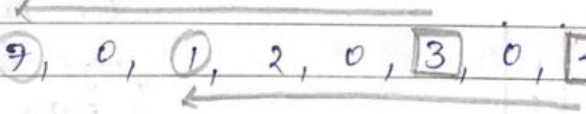
If frame size is 3.

- 1 will be replaced by 2
- 7 will be " " 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.

Page No. ____

Date __/__/____

Saathi

* Most Recently Used.
  - We replace the page ~~wit~~ which was most recently used.

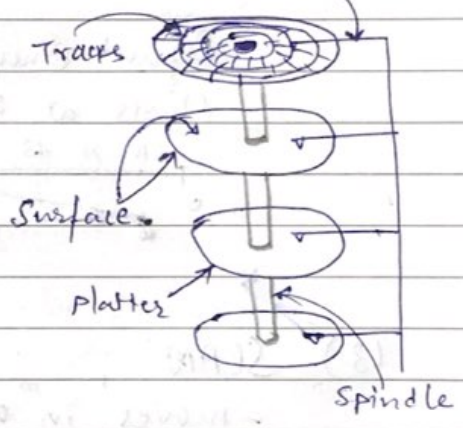
7, 0, 1, 2, 0, 3, 4, 2, 3, 0

Frame size 4.

* Disk Architecture

(moves only back & forth)
Actuator Arm

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



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

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

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

Saathi

Date __/__/____

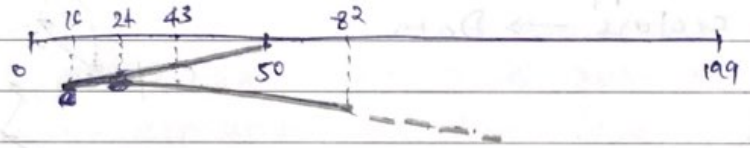
* Disk Scheduling Algorithm :

Is used to minimize the total seek time for a 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 (lasger direction)
- Come back to zero (0), without handling any request.
- Start handling request from 0, in 1 direction.

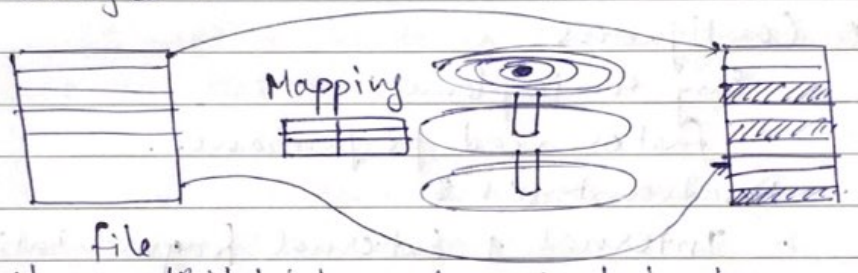
Page No.

Date __/__/____

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

* File System in OS

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



file.

~~C divid~~ 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

Operation :- (reate, read, write, delete, truncate.

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

Attributes info has to be stored in hard disk only.

Saathi

Date ___/___/_____

* File Allocation

Contiguous → (Easy, faster read performance) (Internal fragmentation)

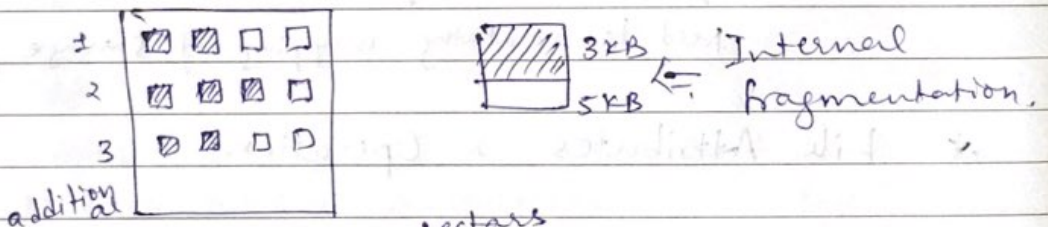
Non-Contiguous → Linked list Allocation → Indexed Allocation

Purpose :-
- Efficient disk space utilization
- Faster access time.

* Contiguous

- Easy to implement
- faster read performance.

Disadvantages :-
- Internal + External fragmentation



If any file requires 4 ~~blocks~~ sectors, 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~~ data contiguously.

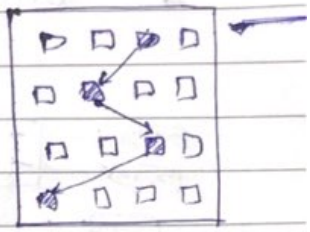
If we increase the file size, then there is only a limit till we can add data, coz only limited space is available next to the file.
So, we cannot grow file size or add more data

Date __/__/____

Saathi

* Linked list Allocation

Data stored in linked list :



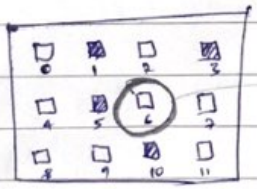
Advantages :- No external fragmentation (Internal will be there)
File size can be increased.

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.

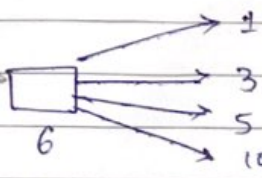
- Needs additional ~~dat~~ space to store pointer data.

* Indexed file Allocation



Directary

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



Adv :- Direct Access
- No ~~frag~~ external frag.

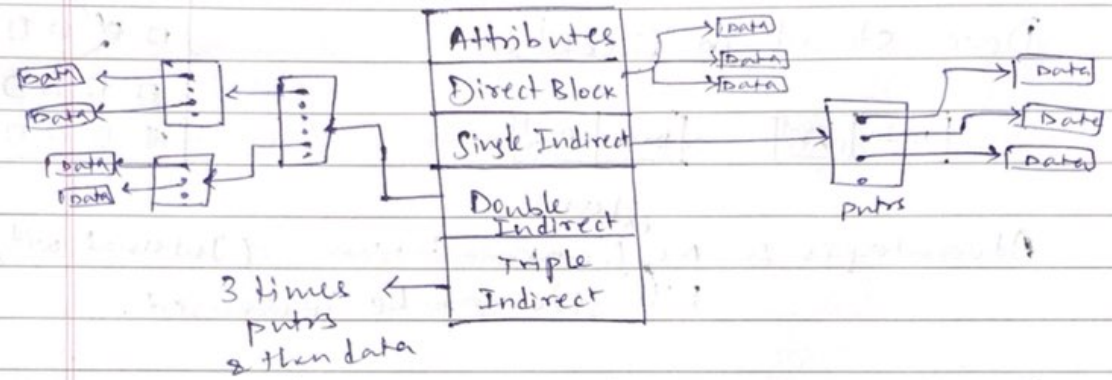
Disadv :- Pointer / index overhead.
- Multilevel indexing

(It is possible that the file size & thus no. of parts of a file ~~was~~ is too large to store in 1 sector. So, its data will be needed to store in multiple sectors.)

~~Data~~ Addresses of diff. blocks of a file will be in 6th sector.

Date ___/___/_____

* Unix I-node structure (I - for index)



Every block will store the following structur