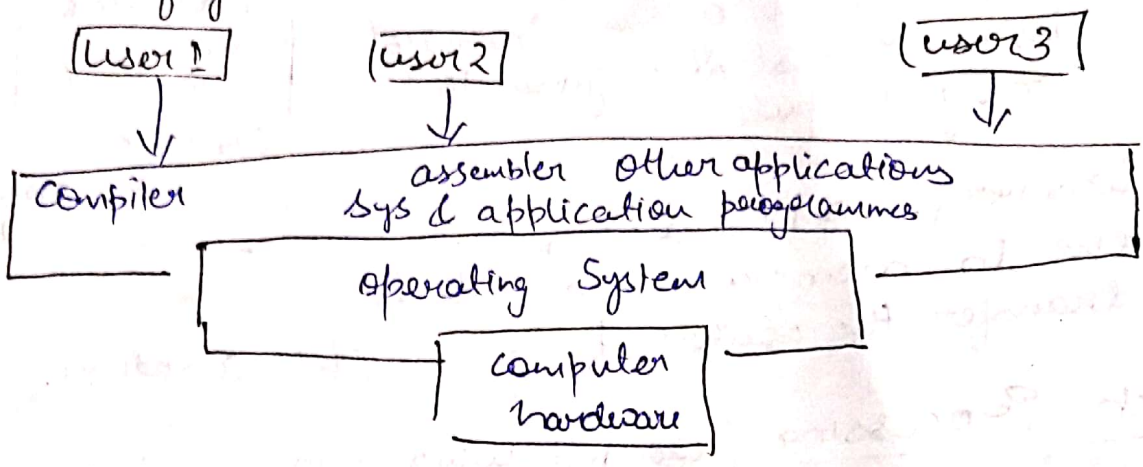
OS

→ It acts as an intermediatory b/w h/w & user.
→ It is resource manager - manages system resources in an unbiased fashion both h/w & s/w.

→ It provides a platform on which other applications programmes are installed

Abstract view of Sys



Goals of OS

Primary Goal
↳ convenience/ user friendly

Secondary Goal
↳ Efficiency

Functions of OS

→ Process management
→ Memory management
→ I/O device management
→ File management
→ Network management
→ Security & Protection

OS Kaise improve Hua

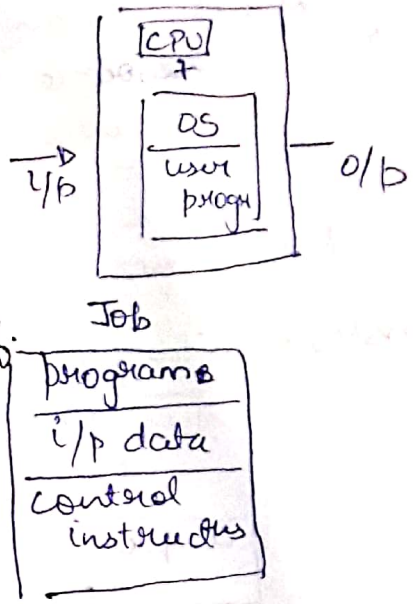
Earlier .

In starting mainframe computers

→ cnn i/p & o/p ; were card readers & tape drives

→ user prepared a job which consisted of the programme i/p data & control instructions.

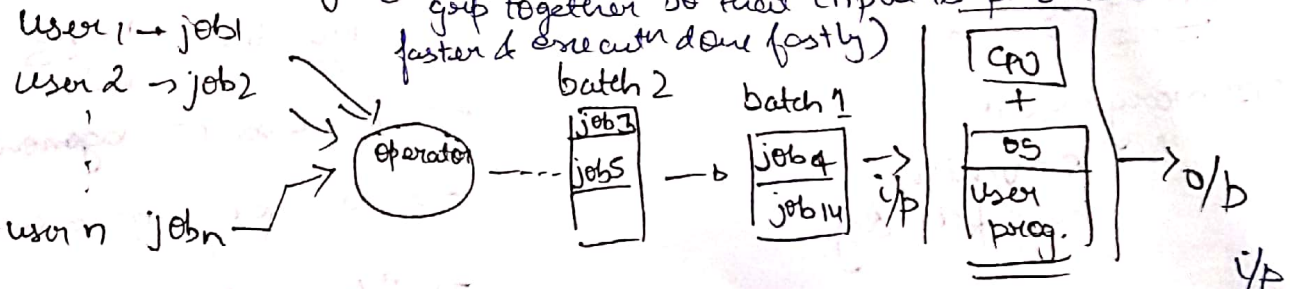
→ all 3 were given in the start itself.



Job → Same input, program & control instructns that were given to early computers.

→ OS was very simple, always present in memory major task is to transfer the control from one job to another.

Batch Processing (jobs which had similar needs were grp together so that input is provided faster & execution done fastly)



* remember CPU is very fast if we take 1 sec. to provide data it takes $\frac{1}{1000}^{th}$ second to process is thus when we were providing data manually most of the time CPU was idle & it caused delay. in o/p generatn.

① Jobs with similar needs are batched together & executed through the processor as a grp.

② Operator sorts job, as a deck of punch cards into batch with similar needs

Adv. Eg → COBOL batch, FORTRON batch etc.

→ In a batch job execute one after another saving time from activities like loading compiler.

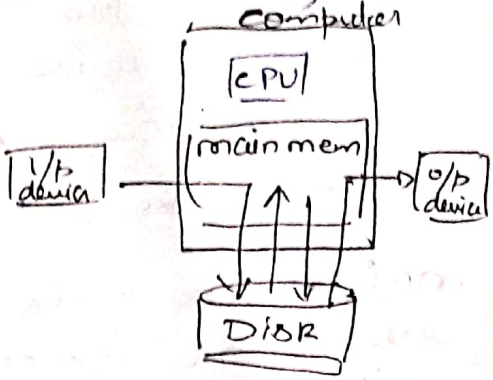
→ during execution no manual intervention needed.

Disadvantages of Batch

→ memory limitation → interaction of i/p & o/p devices directly with CPU

SPOOLING

(Simultaneous peripheral operation online.)



→ CPU ab disk se data le raha hai so the time waste which was earlier due to manual i/p given (batches) thus time improved.

→ Data stored in disk & CPU interacts with Disk (digital) via M.U.

→ Spooling is capable of overlapping i/o operations for one job with CPU operations of other jobs. as input is in disk & CPU takes from disk.

Adv.

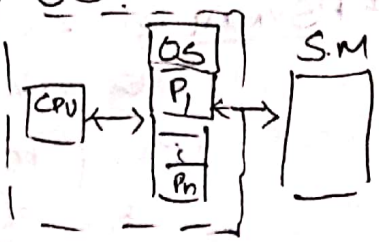
→ no interaction of i/p & o/p devices with CPU

→ CPU utilisation is more as CPU is busy most of the time.

Disadv

In starting spooling was uniprogramming

Multiprogramming OS



→ more than one process in main memory which are ready to execute.

→ maximising CPU utilisation.

→ Process generally requires CPU time & I/O time so if a running process I/O are some other event which don't require CPU then instead of waiting idly CPU switches to some other process.

→ CPU never idle unless there is no process ready to execute or at time of context switching..

Adv.

- High CPU utilisation
- less waiting time,
- May be extended to multiple user

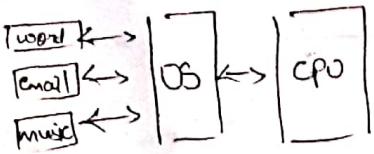
Disadv

- Difficult scheduling
- Main memory manageme is req.
- Memory fragmentation

→ Paging (Non-contiguous, memory allocation)

Multitasking OS / Time Sharing / Fair-Share

Multi-programming with Round-Robin



- Multitasking is multiprogramming with time-sharing.
- Only one CPU but switches b/w processes so quickly that it appears that all operations are being executed at same time.
- Better response time & executing multiple process together.

multitasking keeps jumping from one task to other whereas multiprogramming unless programme ~~to~~ waited or halted we didn't switch operation.

MULTIPROCESSING OS.

→ 2 or more CPU in a single computer communicating or sharing same system bus, memory etc.

→ True parallel execution

SYMMETRIC → one OS controls all CPU ; each CPU has equal rights

ASYMMETRIC → Master slave ~~with~~ architecture
Eg one CPU controls i/p & o/p rest control process

- Easy to design
- less efficient.

Adv.

→ Increased through put

→ Graceful degradation/ Increased reliability (Ek kharab to rest will work)

→ cost efficient

→ battery efficient

→ True parallel

Disadv.

→ more complex

→ overheated or coupling reduces through put

→ Large main memory.

(IMP) CPU Scheduling

→ A process execution consists of a cycle of CPU execution and I/O execution

{ We usually don't focus on I/O execution but it does contribute in process & Here we consider it }

→ Normally every process begins with CPU burst, then may be followed by I/O burst, then another CPU burst, then I/O burst and so on eventually at last ends up on CPU burst

CPU bound processes :- These are those processes which requires most of the time on CPU.

I/O bound processes :- These processes requires most of the time on I/O devices.

* In this chapter we focus on CPU scheduling

A good CPU scheduling idea should choose the mixture of both so that both I/O & CPU can be well utilised effectively.

CPU Scheduling

Non-Pre-emptive

(Koi processor snatch nahi kar sakta irrespective of operations priority)

→ apni marzi se CPU leaves process

Pre-emptive.

(Priority bases pe leaves CPU for higher priority process)

→ Jabar dasti CPU thrown out

CPS Scheduling Terminologies

1) Burst time/Execution time/running time :- time process requires for running on CPU.
2) Waiting time :- time spend by a process in ready state waiting for CPU.
3) Arrival time :- when a process enters ready state
4) Exit time :- when process completes execution & exists sys.
5) Turn Around Time :- total time spend by a process in sys.

$$TAT = ET - AT = B.T + W.T$$

Response time :- Time between a process enters a ready state and get scheduled on the CPU for the first time

(Pehli baar attend hone me kitna time)

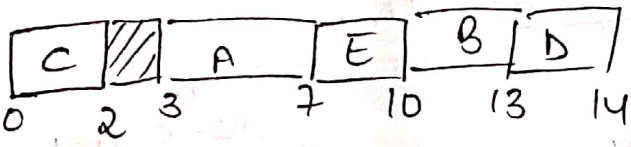
CPU Scheduling Criteria :- ① Average waiting time
② Average response time
③ CPU utilization
④ Throughput

FIRST COME FIRST SERVE (FCFS)

→ Simplest scheduling algorithm, it assigns CPU to the process which arrives first.
→ Easy to understood and can easily be implemented using Queue data structure.
→ Always non-pre-emptive in nature

Process	AT	BT
A	3	4
B	5	3
C	0	2
D	5	1
E	4	3

Gantt chart



Here we started with 0 when C arrived it took BT of 2 then 2-3 no process so idle then A has arrived then E and lastly B & D together which is practically not possible but for answers sake we consider it. In such situations take that process which is first in table.

TAT = BT + WT

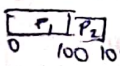
Proper Solution

Process	AT	BT	TAT = ET - AT	WT = TAT - BT
A	3	4	7 - 3 = 4	4 - 4 = 0
B	5	3	13 - 5 = 8	8 - 3 = 5
C	0	2	2 - 0 = 2	2 - 2 = 0
D	5	1	14 - 5 = 9	9 - 1 = 8
E	4	3	10 - 4 = 6	6 - 3 = 3

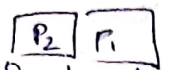
CONVOY Effect (convoy matlab Kafila)

Small processes have to wait for long time for bigger processes to release CPU.

Proc	AT	BT	WT
P_1	0	100	0
P_2	1	1	99 (huge wait)



Proc	AT	BT	WT
P_1	1	1000	
P_2	0	1	(less wait)



Advantages of FCFS

- Simple easy to use
- easy to understand
- easy to implement
- must be used for background processes where execution isn't urgent

Disadvantages of FCFS

- Suffers from convoy effect.
- normally higher avg. waiting time, ~~no consideration~~
- no consideration to priority or burst time.
- Shouldn't be used for interactive sys.

Starvation (tabhi hoga when processor is biased)

In FCFS it is convo effect, not Stowation.

(SJF)

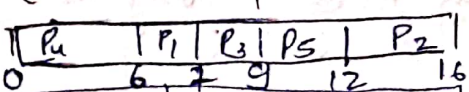
SHORTEST JOB FIRST Scheduling (non-pre-emptive)

OR

/ Shortest remaining time First (SRTF) (Pre-emptive).

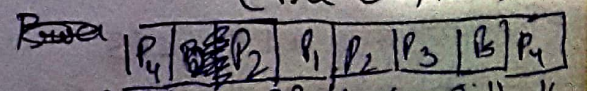
- Out of all available processes, CPU is assigned to the process having smallest burst time requirement (no priority, no seniority).
- If tie, FCFS is used to break tie.
- Can be used with both non-pre-emptive & pre-emptive approach.
- (SRTF) is pre-emptive, is also called as optimal as it guarantees minimal average waiting time.

Eg (Non Preemptive)



Proc.	AT	BT	TAT	WT
P_1	3	1	7-3=4	4-1=3
P_2	1	4	16-1=15	15-4=11
P_3	4	2	9-4=5	5-2=3
P_4	0	6	6-0=6	6-6=0
P_5	2	3	12-2=10	10-3=7

(Pre-emptive)



Proc	AT	BT	TAT	WT
P_1	3	1	4-3=1	1-1=0
P_2	1	4,3,2	6-1=5	5-4=1
P_3	4	2	8-4=4	4-2=2
P_4	0	6,5	16-0=16	16-6=10
P_5	2	3	11-2=9	9-3=6

(*) Note in both pre-emptive & non-pre-emptive Shortest Job first end time is same 16.

In pre-emptive we see after every clock if some other process available with less B.T, we don content switching to that process i.e this is absolute greedy

* SRTF Ka pre-emptive version gaurantees minimal waiting time. (isse better avg. waiting time Koi Schedule nhi deta).

Advantages of SRTF (pre-emptive)

→ Provides a standard for other algo in terms of avg. waiting time.

→ better avg. response time compare to FCFS.

Disadvantages:

→ Algo can't be implemented as there is no way to know the burst time of a process.

→ Process with longer CPU burst time requirements will go into starvation

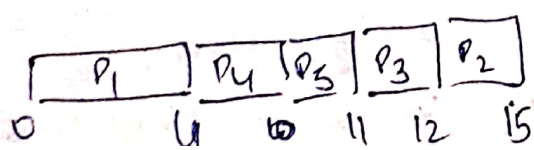
* Starvation as processor is biased.

→ No idea of priority, process with large burst time have poor response time.

PRIORITY ALGO

- Here a priority is associated with each process.
- At any instance of time out of all available processes, CPU is allocated to the process with highest priority.
- Tie broken with FCFS
- No importance of AT or BT
- Supports both non-pre-emptive & pre-emptive.

Process	AT	BT	Priority	TAT	WT
P_1	0	4	2		
P_2	1	3	3		
P_3	2	1	4		
P_4	3	5	5		
P_5	4	2	5		



(Non-pre-emptive)

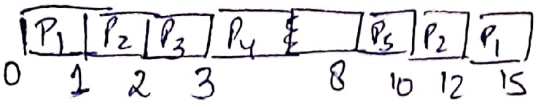
* remember how table is filled

(1) at 0 only P_1 was available though it has least priority but only this was available. thus it is runned.

(2) after P_1 we are at 4 & at 4th clock all processes have arrived. Now based on priority processes run

Now Pre-emptive of Priority algorithm

P_i	AT	BT	Priority	TAT	WT
P_1	0.	4 3	2	15-0=15	15-4=11
P_2	1.	8 2	3	12-1=11	11-3=8
P_3	2	1	4	3-2=1	1-1=0
P_4	3	5	5.	0-3=5	6-5=0
P_5	4	2	5	10-4=6	6-2=4.



Advantages of Prio. Algo

→ Provides a facility of priority scheduling for system processes

→ Allow to run important processes first even if it is a user process.

Disadvantages of Priority Algo.

- Process with shorter/smaller priority may starve for CPU
- no idea of response time or waiting time.

Aging → It is a technique of gradually increasing the priority of process that waits in the system for long.. (age badhne ke sath we increase its priority taki it gets CPU)

ROUND ROBIN

This algo is designed for time sharing systems, where it is not necessary to complete one process and then start another, but to be responsive and divide time of the CPU among the process in the ready state.

- Here ready queue will be treated as circular queue.
- We fix a time quantum up to which one process can hold the CPU in one go; with in which either a process terminates or process must release the CPU and reenter in the circular queue. and wait for the next chance.
- RR is always pre-emptive in nature.

to solve gantt chart of Round Robin

(1) make a queue yourself which helps in ~~room~~

q- $P_1 P_2 P_3 P_1 P_4 P_5 P_2 P_1 P_5$ (given $T_q = 2$)

P_1	P_2	P_3	P_1	P_4	P_5	P_2	P_1	P_5
0	2	4	6	7	9	11	12	13	14

Proc	AT	BT	TAT	WT
P_1	0	~~5~~ 3 1	13-0=13	
P_2	1	2 1	11-1=11	
P_3	2	1	5-2=3	
P_4	3	2	9-3=6	
P_5	4	3 1	14-4=10	

One P_1 is opted as it was available at 0 time, we have $T_q = 2$ thus CPU is with it for 2 clock cycles.

Now since 2 cycles done P_2 & P_3 would have arrived and would be in queue already.

& so on this cycle continues

Advantages

- Performs best in terms of average response time
- Works well in case of time sharing sys, client server architecture & interactive server
- Kind of SJF implementation

Disadvantages.

- Longer process may starve
- Performance depends heavily on time q constant
- no idea of priority.

Deadlock:- in a multi-programming sys, a number of process compet for limited number of resources and if a resource is not available at that instance then process enters into waiting state

- If a process unable to change its waiting state indefinitely because the resources requested by it are held by another waiting process, then sys enters in Deadlock.

SYSTEM MODEL

(1) every process will request for the resource

(2) if in entertained then process will use the resource

(3) Process must release the resource after use.

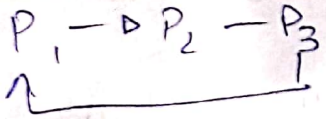
#Deadlock Ki 4 conditions

① **Mutual Exclusion**: At least one resource type in the system which can be used in non-sharable mode i.e mutual exclusion (one-at a time / one-by one)
Eg. Printer.

② **Hold & Wait**: A process is currently holding at least one resource and is requesting additional resources which are being held by other process.

③ ~~Pre~~. **No-pre-emption**: A resource can not be pre-empted from a process by any other process.
- Resource can be released by only voluntarily by the holding it.

④ **Curcular Wait** Each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.



★ For Deadlock all 4 conditions must be satisfied.

#Deadlock Handling (Deadlock हटाने के तरीके)

① **Prevention** :- means design such a sys. which voilate at least one of 4 necessary conditions of dead lock & ensures independance from dead lock.

② **Avoidance** :- System maintains a set of data using which it takes a decision whether to entertain a new request or not, to be on safe side.

③ **Detection & Recovery** :- Here we wait until deadlock occurs & once it is detected, we recover from it.

④ **Ignorance / Ostrich algo** :- We ignore the problem as if it doesn't exist.

Deadlock Prevention :-

① mutual exclusion ✗

(not possible to avoid this in prevention since some resources are such that they can't be shared eg:- a printer can print one file at a time but not combination of 2)

② Hold & wait

(a) conservative approach :- Process is to start execution if & only if it has to acquire all the resources.

(less efficient, not implementable, easy deadlock independance)

(eg: imagine P_1 needs R_1 & R_2
P_1 has R_1 when it goes of R_2
P_2 captures R_1 thus P_1 deprived of R_1, to avoid this we wait till P_1 gains both R_1 & R_2 & resumes only once it has both)

(b) Do not hold :- Process will acquire only desired resources, but before making any fresh request it must release all the resources it currently holds.

(efficient, implementable)

(if P_1 needs R_1-R_{10}, but initially it needs R_1-R_5 thus we proceed & when it needs R_6-R_{10} it must release R_1-R_5)

(c) Wait Timout :- We place a time limit upto which a process can wait, after this time it has to release all the holded resources.

(if it can run in time take again else it need to release all resources)

③ No Pre-emption

(a) Forceful pre-emption :- we allow a process to forcefully preempt the resource held by other process.

→ This method may be used by high priority process or system process.

→ The process which are in waiting state must be selected as a victim instead of process in running state

④ Circular Wait :

→ It can be eliminated by first giving a natural number of every resource.

→ Allow every process to either only in increasing or decreasing order of resource number.

→ If a process requires a lesser no. (in case of inc. order) then it must first release all the larger numbered resources than req. no.

Eg
P_1	R_2
R_1	R_2
R_2	R_1

(Here P_1 takes R_1 & P_2 R_2. Now P_1 waits for R_2 & P_2 for R_1 deadlock happens)

P_1	P_2
R_1	R_1
R_2	R_2

(Here whoever gets R_1 first completes the job then other does it's job) Deadlock resolved

PROCESS SYNCHRONISATION

Race condition (when order of execution can change result).

Eg
```
P( )
{ R(a)      | P1->11 | 10 (then context switching at →)
  a=a+1     | P2->12 |    11 (P2 completes the process since P1 isn't
  W(a)      |        |       complete yet it will read 10 & make it 11)
}                    | 11
                     |  └ Now when back to P1 it had read 10 thus it will
                     |    also result in 11
```
(A) P_1 P_2 (B) P_1 P_2

→ different answer depending on order of execution.

Critical Section :- Part in process where it access shared resource. execution

* If a resource is shareable, and is shared in one-by-one fashion it is of no problem (A) else it may cause problem (B).

P_1	P_2
① C=B-1	③ D=2+B
② B=2*C	④ B=D-1 ;

Q How many different values can shared resource B have B=2 initially

C-1	C-2	C-3	C-4	C-5	C-6
1,2,3,4	3412	1342	3124	1324	3124
C=2-1=1	D=2+4	C=1	D=4	C=1	D=4
B=2	B=4+3	D=4	C=1	D=4	C=1
D=4	C=D-2	B=3	B=2	B=2	B=2
B=3	D=B=4	B=2	B=3	B=3	B=3

(Race condition same code diff. answer on diff execution)

Critical Section problem

{ A resource can be shareable, but at a time Kewal ek Process operate Karega on it }

TO solve a critical section Problem it must follow following 3 condition

1. **Mutual exclusion** (mendatory) { critical section accessed in mutually exclusive manner }
2. **Progress**
   { Only those processes must be entered into critical section, which need to enter, we should not do round-robin on all the processes }
3. **Bounded wait** (not-necessary)
   (a max time limit after which a process can enter critical sectn)

Eg

This is Turn variable

P_0

```
while(1)
{
while(turn!=0);
Critical section
turn=1;
Remainder section
}
```

P_1

```
while(1)
{
while(turn!=0);
critical section
turn=0;
Remainder section;
}
```

- ME ✓
- Progress ✗
- BW ✓ (Bounded wait ✗) (no one wait independent of how many?)

→ this follows mutual exclusion ✓

→ Progress ✗ since P_0 ke baad P_1 apne aap jata hai, wit we don't ask wether process wants to go to critical section or not.

thus not valid.

Rectified.

P_0

```
while(1)
{ while(turn!=0);   *
critical sectn
flag[0]=F
```

flag [F][F] (0, 1)

P_1

```
flag[1]=T
while(flag[0]);
critical sectn
flag[1]=F
```

But if we do content switching at * we get deadlock.

4. Should not have restrictions like can should run only on windows or only on Linux but be universal
   - NO assumption related to H/w and S/W point KO

Soln P_0

```
while(1)
{ flag[0]=T
  turn=1
while (turn==1 && flag[1]==T);
  Critical sectn
  flag[0]=F
}
```

```
while(1)
{ flag[1]=T
  turn=0
  while (turn==0 &&
          flag[0]==T);
  Critical sectn
  flag[1]=F
}
```

Turn = 1

flag [F][F]

Semaphores: Uses ① Critical sectn problem (S=1)
② Order of execution b/w processes
③ resource management.

① ~~Critical se~~.
Semaphore is an integer ~~value~~ variable that apart from initialization, is accessed only through 2 standard atomic operations

int S=1;

(1) wait (S)
(reduces value by 1)

```
wait(S)
{ while (S<=0);
  S = S-1
}
```

(2) Signal(S)
(increases value by 1).

```
Signal(S)
{ S = S+1;
}
```

```
P, do{ entry sectn
// critical sectn
exit sectn
// remainder sectn
}while(T)
```

→

```
P, do { wait(S);
// critical sectn
  Signal(S);
// remainder sectn
}while(T)
```

{ Koi bhi Process kabhi bhi content change kar sakta hai in OS }

Process Synchronization :
(Semaphores)

For deciding Order

wait(S_1) $P_2 \rightarrow P_1 \rightarrow P_3$ execution wait(S_2)

P_1	P_2
Signal(S_2)	Signal(S_1)

$S_1 = 0$ $S_2 = 0$

Since S_1 & $S_2 = 0$ first P_2 executed even if context switched S_1 & $S_2 = 0$ thus only P_2 can be executed 1st

For managing resources.

P_1 :

```
wait(S)
CS
sig(S)
R-S
```

If we have 5 printers initially
$S = 5$,

* Here S ko usi value se initialize karo jitne resources HO.

Signal => V( )
wait => P( )

① lock based solution → but this didn't gave Mutual-exclusion

```
1 while (lock == 1);
2 lock = 1
3 critical section
4 lock = 0
```

if after 1st instruction line process P_1 gets preempted.
then P_2 will start & since P_1 left after 1st line. lock is still = 0 ∴ it will proceed and enter critical section & make lock = 1
→ Now when P_1 resumes it make lock again & enters already occupied critical section thus Race condition may occur

② Test and Set
↳ to overcome problem of lock
combine line 1 & line 2 so that no preemption b/w 2 lines possible

```
while (test_and_set(&lock))
CS
lock = false
```

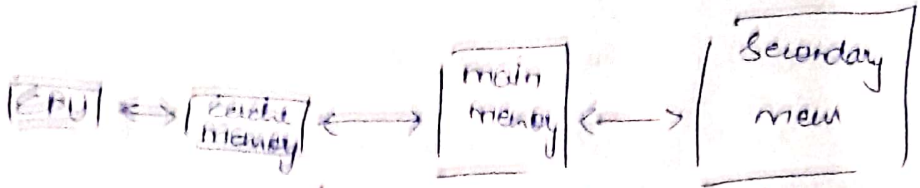
↳ No preemption possible now b/c as its single code, preempt either after or before this.

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

In test and set
→ Mutual Exclusion ✓
→ Progress ✓

Basics Of Memory Management

Criterias (1) Size(↑) (2) Access time(↓) (3) Per unit cost(↓)

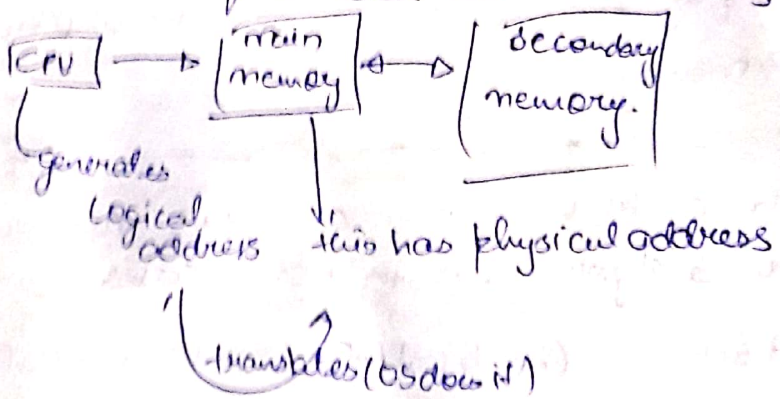


(Heirarchy of memory)

main memory me it fetches current, imp data so know access time dec. since it is small in size.

LOCALITY OF REFERENCE :- most of the time instructions run in sequential manner so we prefetch the data (imp data) from sec memory to main mem.
(so currently as per used the cache data etc as per loc data)

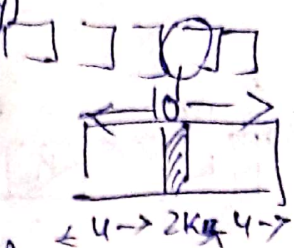
In OS cache isn't of concern so structure is



Contigious memory allocat^n (eg- array)

↳ [ ] so easy access
↳ address translat^n

↓ Always it suffers from external fragmentat^n →



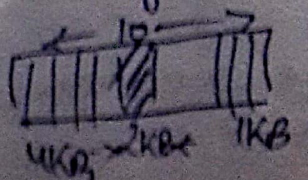
Now if a 5KB process comes, we have 8KB still we can't use it to store 5KB data this is external fragmentat^n

Non-contigious mem. allocat^n

Eg (linked list)

↳ slow since to access any data we have to start from head & reach that point

Eg



(Here 5KB stored in parts 1KB+4KB since it wasn't contigious)

CONTIGUOUS MEM ALLOCATN

fixed size partitioning

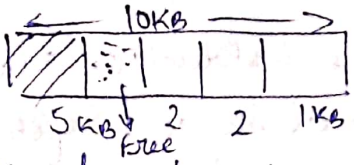
Variable size partitioning

* CPU can access only main memory.

(1) For a process to be runned, it must be first brought to main-memory.

Fixed Size partitioning

(Kitne partitions & Kit Har ek ka size may differ)



Now for a 4KB process this 1KB wasted this is internal fragmentatn

→ Easy to manage.

VARIABLE SIZE PARTITION

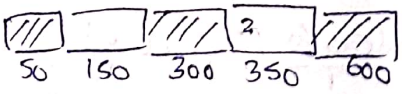
→ we dont have any fixed partitions, and the processes keep coming we allocated data's space to them & keep moving

→ NO internal fragmentatn since jitne size ka process aya hain utna we dete hain

---

In Both contigious & non-contigious memory allocation there are 3 algos used in both cases

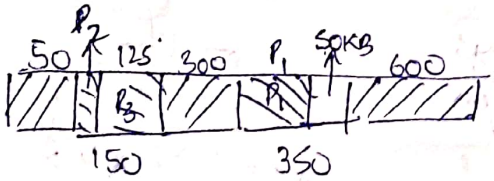
(1) First fit (2) BESTFit (3) WORST Fit.



— memory & we have to allocate free mem. to P_1-300 P_2= 25 P_3=125 P_4=50

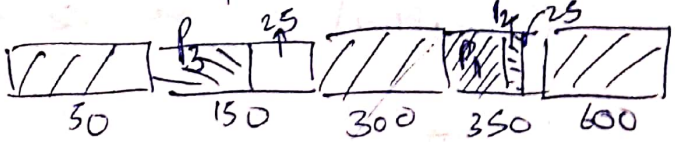
(Variable size partitioning)

First Fit



→ Start from start se searching, jahan space available waha inserted, variable size.

Best fit (search all possible & use that block which is smallest)



thus these 25+25 = 50 wasted.

Worst fit

(use largest block possible which can accomodate process.

Best Fit uses the sabse smallest block which can contain the ~~da~~ process i.e why 350 blank space was used not 150 in best fit, since 350 is the smallest block ~~to do~~ which can contain 300 bit process.

→ It uses smallest space so after use data remaining will be even smaller thus it is best suited for ~~Variable~~ fixed size partitioning me.
Since fixed size partitioning me small spaces are wasted

Worst fit → Use that block which is largest in size.
In Variable size partitioning it is best suited since largest se necessary part for a process used up now still we can use the left over size since it will be of ~~great~~ big size.

to inc. this RAM/main mem size ↑

* Degree of Multiprogramming = How many programs are there in main mem

Contiguous (External Frag)
- No. of partition fixed → Fixed partition (Table fixed) → size of partitions can be same or diff.
- Dynamic partition → allocate/Deallocate Hard complex
  - Ext frag
  - No internal fragmentation
  - No Limitation of Process
  - No Limitation on Process size.

Non contiguous
- Pa
- Multi
- → Inve
- → Segme
- → Segme

Fixed Partition
- Internal frag.
- xed size to after assign
- ocess to ~~s~~pace some space left out
- → Limit of process size
- → Limit of degree of multiprogramming
- Ex. frag.
- process can't be oken down to utilise ft out mem.

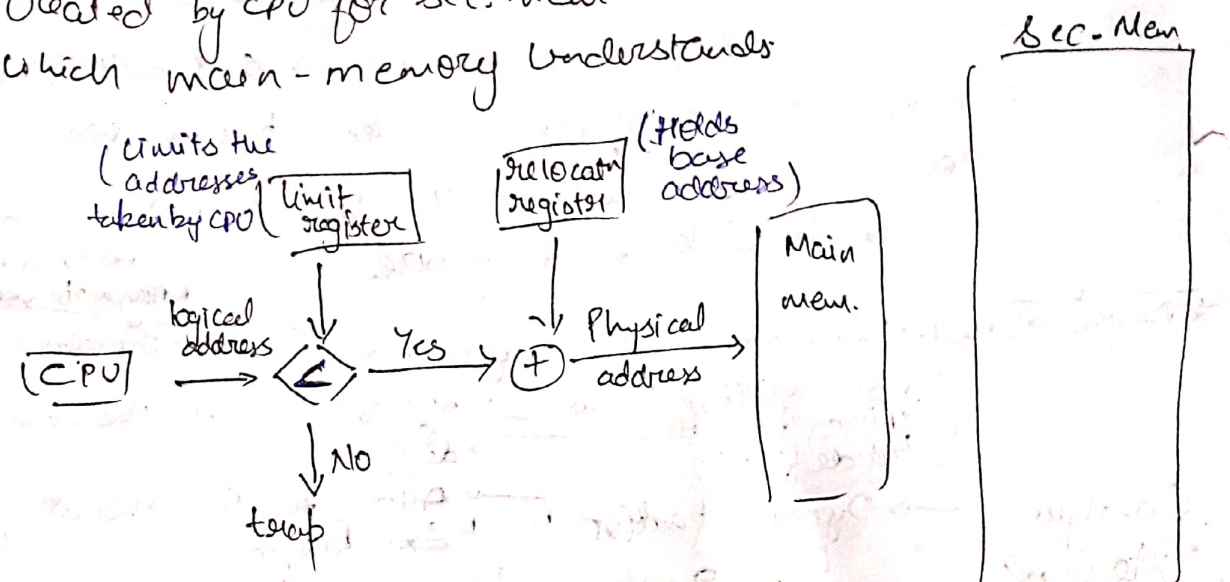
Contiguous mem
- (Ext. frag sure)
- Fixed (intern. frag)
  1. Fixed size
  2. Sizes of partitions not same
- Dynamic (no internal frag)
  1. Size of partitions same
  2. Sizes of partitions not same

Non contig. manner
to allocate mem. contig. manner
① First fit ② Best Fit (choose choose parts left)
③ Next fit ④ Worst fit (no choice parts left).

* Ext. fra ... but is slow as we copy data & paste it ~~i~~ again, we remove ~~o~~ left out empty spaces.

Address Translation

* CPU creates logical address for secondary memory thus we change logical address to physical address.
* Remember CPU interacts with main-memory only since we want fast processing, but CPU ko ni pata that instead of taking it to sec. mem we are taking it to main-memory ~~thus~~. Main mem can be read with physical address thus we convert logical address created by CPU for sec. mem to ~~logical~~ physical address which main-memory understands



Eg

	LR	RR
P_0	500	1200 [650] (450)
P_1	275	550 [X] (300)
P_2	212	880 [1080] (210)
P_3	420	1400 [X] (450)
P_4	118	200 [280] (80)

Find if process legal or illegal.

Illegal addresses are when we let suppose fetch 100 bit data from sec mem to main mem. since it is contigious Now if lets say base address of ~~mem~~ data fetched is @500.

Now if CPU generates logical address 120, it will be illegal since we have fetched the seg. of data with size 100 only, so we won't be able to access (500+120) Base addr 620

★ external fragmentation is more ~~frag~~ problematic.
& external fragmentation happens in contigious mem. allocation
thus we sort to non-contiguous.

NON-contigious ① PAGING

frames

Non Conti => Since we have divided process into partitions = PAGE, before bring into main Mem.
→ These Pages are created based on wholes in mem, and these holes are dyn thus time consumed in finding them.
||ly RAM/main Mem. also divided to hold Pages, these divisions of RAM = Frame
[Size of Page = frame size]
No. of Pages = Process size / Page size
No. of frames = Main Mem. size / frame size

① PAGING :-

Both memories are divided into ~~f~~ partitions.

Notice stored in separately.

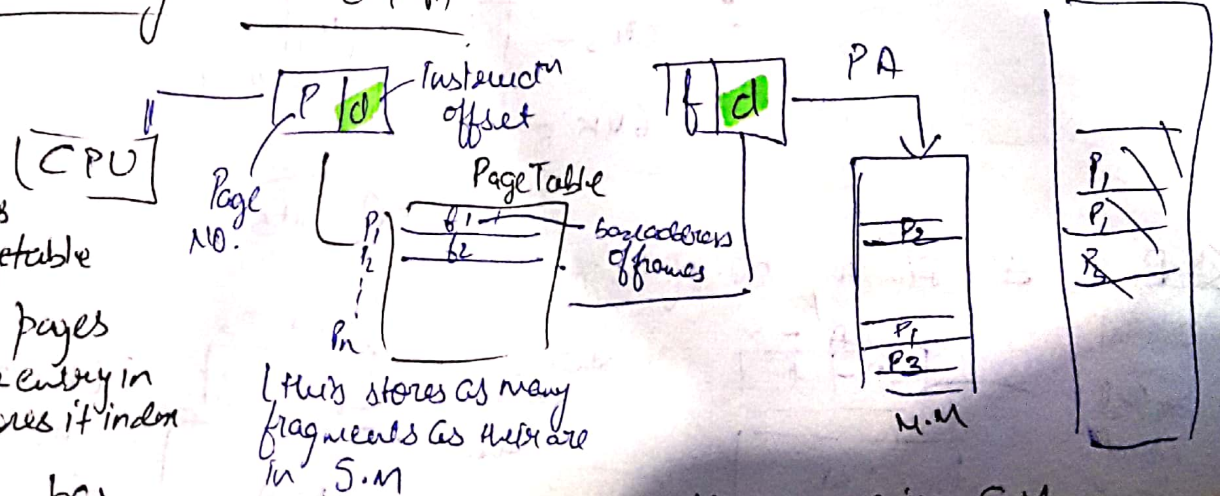
- Both MM & SM are divided into fragments of eq. sizes eq. (Page Size = Frame Size)

i.e lets suppose SM divided into fragments of 1KB each ~~them~~ then MM will also be divided into fragments of size 1KB each.

The fragments of S.M are called pages.
The fragments of M.M are called frames

★ every process has its own page table.

Translating L.A to P.A



total entries in Pagetable = No. of pages as have entry in PT stores it index

PT ~~stores~~ has as many fragments as there are in S.M
PT stores frames ki base address.

→ NO external fragmentation
→ Every process has different
→ Internal fragmentation (Ds
→ fast
→ Instruction access time has de

Page table me entries/row == no. of pages as ~~here~~ every entry/row will store address of that page.

Logical add. → | Pg. NO. | Page offset | Page offset ~~is size of~~ is No. of bits req to represent the total no. of pages
for 2 pages offset = 1
for 8 page offset = 3

Phy. Add (Related to Sec. Mem)

No. of locations in a page = 1024 (2^{10})

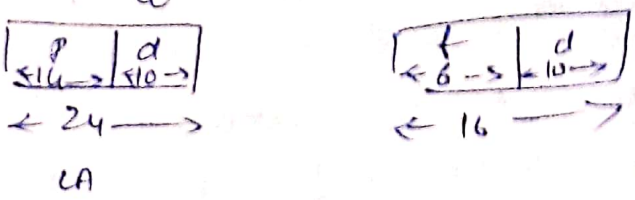
Page size = frame size [Page is term used for size of frame is term used for main mem]

∴ offset which used to tell column to get into page

No. of locations in page = 210

∴ offset bit = 10 bit

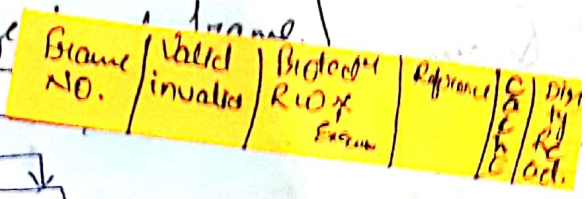
∴

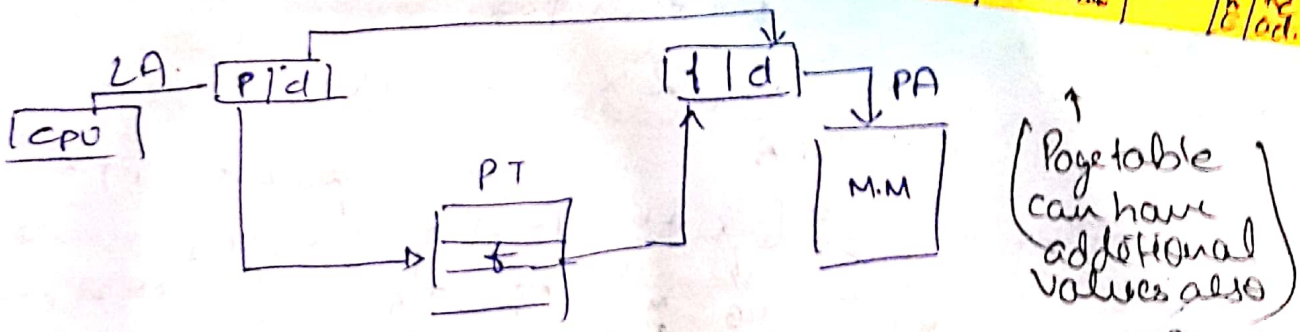


∴ total No. of pages = 2^{14} = 16K = $2^4 \times 2^{10}$

total No of frames = 2^6 = 64

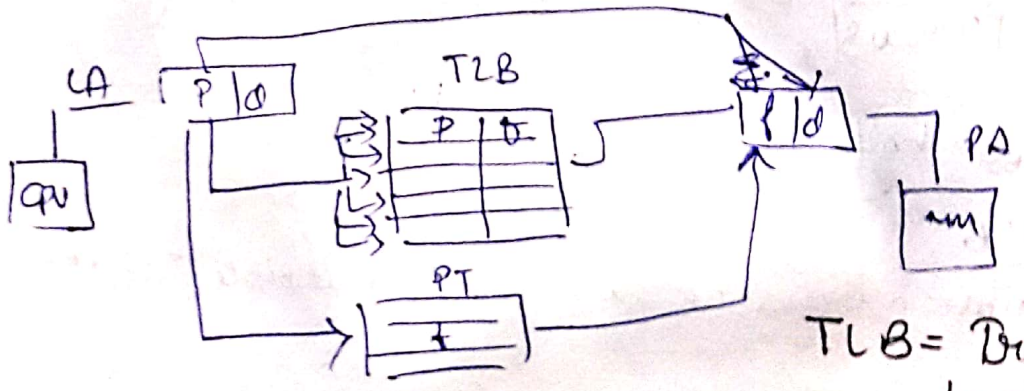
Note no. of frames & pages is different but since memories are diff. Our sec. mem & other main mem but (size of each page = size ... frame





2 access ~~to~~ to MM first for ~~PA~~ Page address since Page table is also in MM.
second for frame address

* every process has diff. PT.



TLB = Translation Look aside Buffer

With TLB, we use the locality of reference i.e most of the time instructions are executed in a sequential pattern, we use this prop.

thus ~~for~~ instead of going to PT each time for getting frame, we make a TLB which has (P.lf) in it (extra hardware hota hai small sized)
~~so after some time~~

* This TLB has the seq. instructions in it to ek baar me it gives address of frames for a lot of ~~sequent~~ instructions ahead in sequential manner.
* for every ~~v~~process or new content switch we have to flush TLB since new frames & pages are to be stored for each process.

Q MM = 400 µs
TLB = 50 µs
h = 90%,

Ans If no TLB, we have to access mem twice for each page.
∴ 2 × 400 µs = 800 µs

with TLB,

~~1st hit~~ = .9 [50 + 400] + .1 [50 + 400 + 400]
(accessing PT)

- .9 : Chances of TLB hit
- 50 : TLB access time
- 400 : MM access time
- .1 : NO hit
- 50 : TLB for next time as jab TLB me jaenge tabhi pata chalega ki TLB doesn't has page data.
- 400 : Accessing final PA Instruction accessing

= .9 [450] + .1 [950]

with TLB. = 490 µs

Almost halfed.

Disadv. of TLB

- For multiple process we might need multiple TLB (expense)

* Some part of TLB is given to OS (No process uses these) & is known as wired down entries.

Earlier when new process was called all values were flushed & TLB filled again from scratch

but now, OS has a habit of gain access to thing for small fraction of time just to check wether they are functioning properly or not.

So, Here when content switching OS makes the process continue it through its wired down entries and meanwhile all the entries are already present since content switch wala was accessed through it thus no need to flush TLB in this case.

CONCLUSION

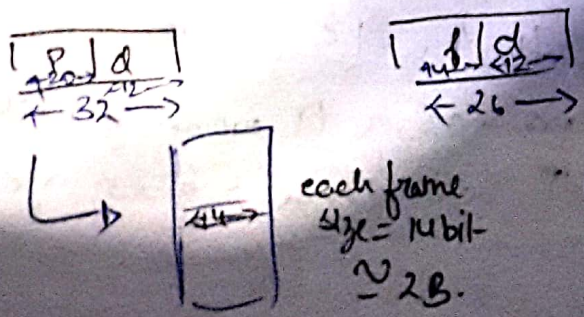
Paging had severe problem of speed.
to increase speed we used TLB.
TLB has additional cost
multiple content switch problematic
↳ tried dealt with multiple TLB but this inc. cost
↳ or by using wired down entries (OS kodake thoda part).

— ∞ — ∞ —

Multi-Level Paging

PT ka PT banado.

Q Find wastage / size of Page T.



MM = 64MB
LA = 32b
PS = 4KB

MM = 64MB
$= 2^6 \times 2^{20} = 2^{26}$
∴ PA has 26 bits.
Now LA = 32b

Page size = 4KB
$= 2^2 \times 2^{10} B$
1B (size of each entry)
$= 2^{12}$
∴ 12 bits req. for offset

Now,
Total address possible with 20 bits $= 2^{20} = 1MB$
∴ total size of PT = 1MB × 2B (size of each entry)
= 2MB

(Race condition problem)

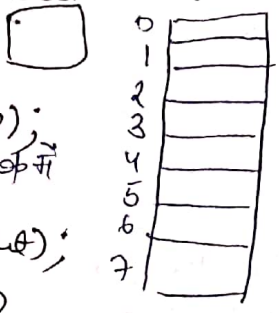
Producer Consumer Problem [Process Synchronization Problem]

Cooperative process: Producer produces a item adds it into buffer & consumer takes item out of Buffer
→ Buffer is being shared by both & count variable shared by both.

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

n = 8

Buffer [0...7]



Buffer Full case stuck in ∞ loop.

in = add of next empty slot in buffer

• in=0 initially

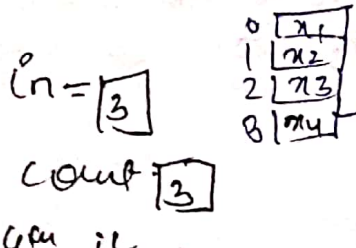
* in used to access buffer in badha do agar we produce an item

```
Void consumer(
{ int itemc;
  while (true)
  { while (count == 0);
    // empty buffer case, ∞ loop me
    // stuck
    itemc = Buffer(out);
    out = (out+1) % n
    (count = count - 1);
    Process_item (itemc);
  }
```

out

out in consumer is the variable that points to the item that will be taken out of buffer.

Eg (a) as we add item x_1 into buffer, (b) we keep increasing in
Lets suppose 3 items pushed in = 3 (i.e index of next free space in buffer)
Now for consumer we start using these items initially 0th index buffer used ∴ out = 0 after taking out 1st index out inc & points to next item 1st index & till



Now bring an item in = 4
Buffer not full → proceed → x_4 comes to 3rd index → in inc. → Now in count = count+1, CPU preempted after 2nd process that is R_p reg. ka value 4. but not loaded back in count var.

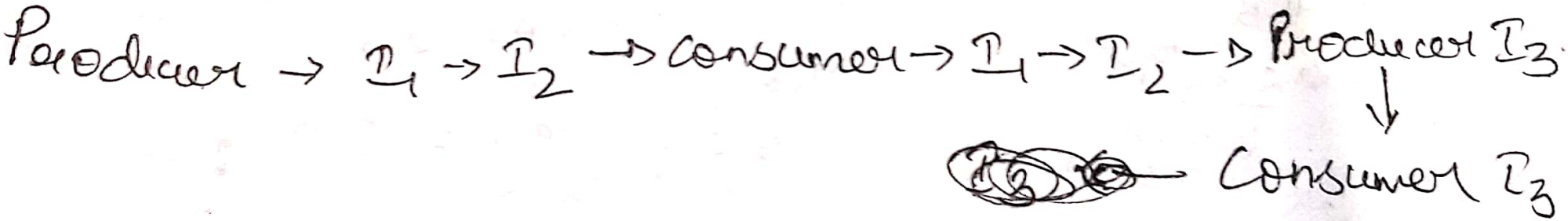
Now consumer ⇒ Not empty ⇒ Buffer item at out=0 x_1 put in R_c before itemc ⇒ ~~count = 0~~ out = out+1 = 1 ⇒ Now count = count - 1 # before 3rd instruction exe. it is preempted.
∴ on resuming p. 3rd exe ∴ R_p = 4 & count = 4 Now two process ended
back to cons. 3rd exe ∴ R_c = 3-1 = 2 = count Now this process ends & count = 2 whereas it

```
1. load Rp, m[count]
2. INCR Rp;
3. Store m[count], Rp
```

Rp reg. me count loaded.
inc. Rp value by 1
store back Rp's value into count

```
into CPU
1. load Rc, m[count]
2. DECR Rc
3. Store m[count], Rc
```

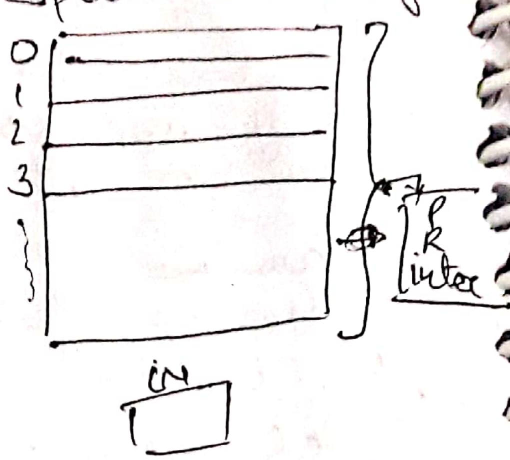
Scanned with CamScanner



Printer Spooler problem of Synchronization

★ IN is shared variable
(it tells empty pos. in spooler

Process to print, this code run



1. Load R_i, m[in]

2. Store SD[R_i], "F-N"
(file name)
(file name stored in spooler dir. of R_i^{th} pos.)

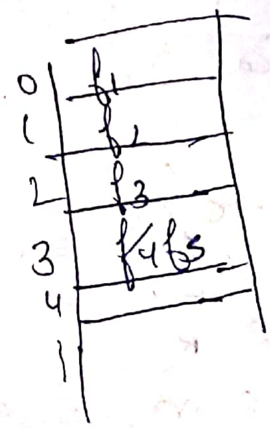
3. INCR R_i

4. Store m[in], R_i

in stores next empty pos in spooler direc

when 2 process P_1 & P_2 come, Kitne bhi process at a time single executed. both will run the same code, sharing the same variable in.

assuming 3 file already exist $f_1 = 0$, $f_2 = 1$, $f_3 = 2$ [in = 3]



P_1 [f4] P_2 [f5]

R_2 [3] 4

P_1 $I_1 I_2 I_3$ | P_2 I_1 I_2 I_3 I_4 | $P_1(I_4)$

I_2 SD[3] stored f4 & R_i inc. to 4 — pre empted

Now 3rd location be f5 stored.

$P_1(I_4)$ → this stores in [in] as R_1 "4" & In = 4

$R_i = 4$, till now P_1 ke lie R_1 4, P_2 ke lie R_2

★ Here P_1 wanted to write f4 at 3rd SD but due to no-synchronization P_2 stores f5 & finally f4 lost.

2 level Paging Using Eg

We needed multi-level Paging as when Page table size > main mem frame size.
i.e Page fit nahi Hoga frame me ∴ we do multi-level Paging.

Ques Phy-add space = 256MB this is for Phy add. = $2^8 \times 2^{20}$
Logical Address Space = 4GB Logical add. = $2^2 \times 2^{30}$
Frame size = 4KB = $2^2 \times 2^{10}$
Page entry = 2 B.
table

PA = | F.No (16) | offset = F. size (12) | = 28

LA = | Pg No. (20) | offset (12) | = 32

Q. ∴ Page table must have 2^{20} entry

Now each process of 4KB ∴ Total no of frames from PA = 2^{16}
Since 2^{16} frames present ∴ we need 16 bit to represent each frame address ∴ frame address stored in page table

Also already given Page table entry size = 2 Byte = 16 Bit

PT. Size = $2^{20} \times 2B = 2MB$
Now this PT has to be inserted in main memo frame
∴ we need to further divide it.

No. of division = $\frac{\text{total size}}{\text{size of each page allowed} = \text{size of frame}} = \frac{2^{20} \times 2B \ (2MB)}{2^2 \times 2^{10} \times 2 \ (4KB)} = 2^9$.

∴ we make a further outer PT with 2^9 entries
each entry of OTPT (Outer PT) = 2B ∴ Size of OTPT = $2^9 \times 2 = 1KB < 4KB$
∴ we can put this page table into frames now.

Now PA = | 32 | → | 20 | 12 |
| 9 | 11 |
outer P.T ← page of P.T.

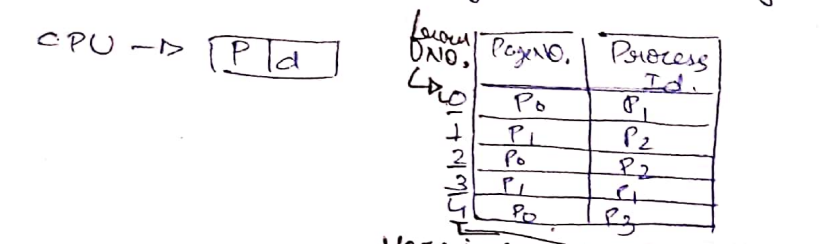
→ Every Process has its Page table
→ Page table are kept in main mem. i.e in frame

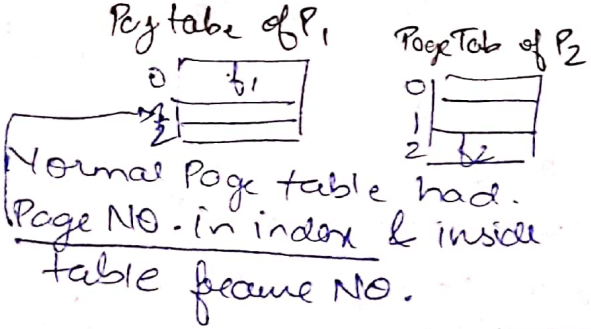
*N

INVERTED PAGING

★ In normal paging, page table was also stored in main mem. frame, which consumed crucial main mem. data to overcome this INVERTED PAGING introduced.

→ In inverted Paging we make a global page table, maintained by OS.





$$\left(\begin{array}{c}\text{NO. of entries in inverted P.T} = \text{NO. of frames}\end{array}\right)$$

★ In inverted Page table ⇒ Page NO. & Process Id are both mend as ~~only~~ every Process has its pagetable ∴ there will be multiple ~~list~~ values as each process can have 0th Page, 1st Page. NO to further classify of process is to be contacted Process Id used.

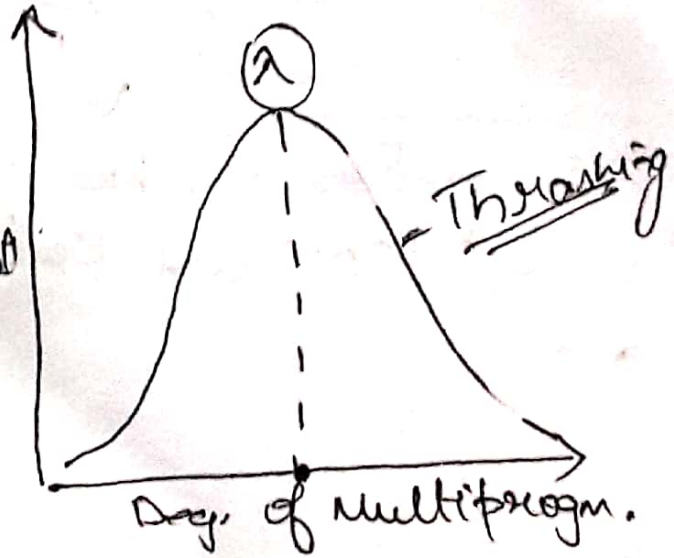
★ Due to inverted Paging now time has increased.

THRASHING → Ek Process ko ek do page hi Main Mem me Le Lale, to inc. degree of multiprogramming.

Page fault → When CPU asks for a process and that process is not in Main Mem. Page fault happens.
→ It takes large amount of time to solve this problem.

→ To inc multiprog. Ek process ke few pages brought to MM. Now when CPU asks for a Page of a process that isn't in M.M Page Fault occurs

- when we bring pages in ~~M~~ M. Mem
- CPU utilisation inc but after particular value suddenly all process will give Pagefault & CPU utilisation will dec. drastically

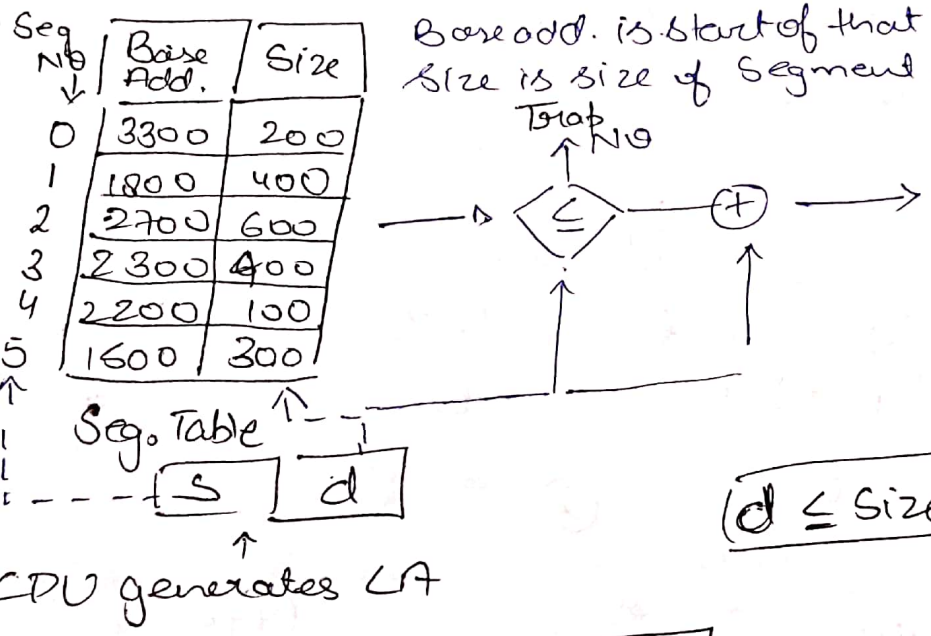


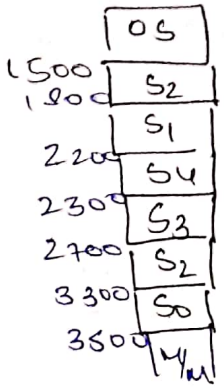
Solution (1) Inc. M.Mem. Size
(2) Long term Scheduler. को slow करदो ताकी (Process को M.Mem में लाने वाला) less Process in M.Mem.

SEGMENTATION

Here we divide process code depending on user's point of view. Like add functh Ka ek seg, sqrt ka ek seg.

- Now since segments done depending on user code.
  ∴ Segment size is not fixed.



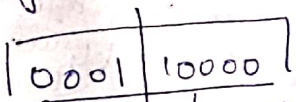


$d \leq Size$

LA — MMU → PA

mem. mang. Unit

eg we



ind seg. no. | seg size (d).

we go to 1st index & get base 1800 now. d amount 400 often 1800

∴ 1800 + 400 = 2200 tak read. ch2 ho

also if $d \geq 400$ then nahi kar pate as that seg is of only 400 thus is not possible.

* In segmentation related data kept together,

OVERLAY

put this is used to

when process size > size of partition in MM, we normally we can't put it into

kept together, ~~ø~~

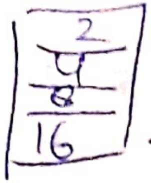
∴ 1800 + 400 = 2200
tak read करता
also if d ≥ 400 then nahi kar pate as that seg is of only 400 thus soo not possible.

OVERLAY

~~this is used to put~~

- when process size > size of partition in MM, ~~we~~ normally we can't put it into MM, but using overlay we can.

→ Used ~~in fixed~~ where process size > memory kamam. partition

Eg  to bring a process of size > 16 we will use Overlay

★ In overlay we divide the process, अगर एक part jab chaie tab lake taken out and next part taken in

Problem
↳ OS has no driver to decide which part of process should be brought in first. User divides it thus at time 2nd process brought before 1st then error occurs

→ Overlay used in embedded sys (ckt with fixed functionali Eg. washing machine)
not in PC as parts of process should be indep. also that is ek doosre pe dependent nhi"

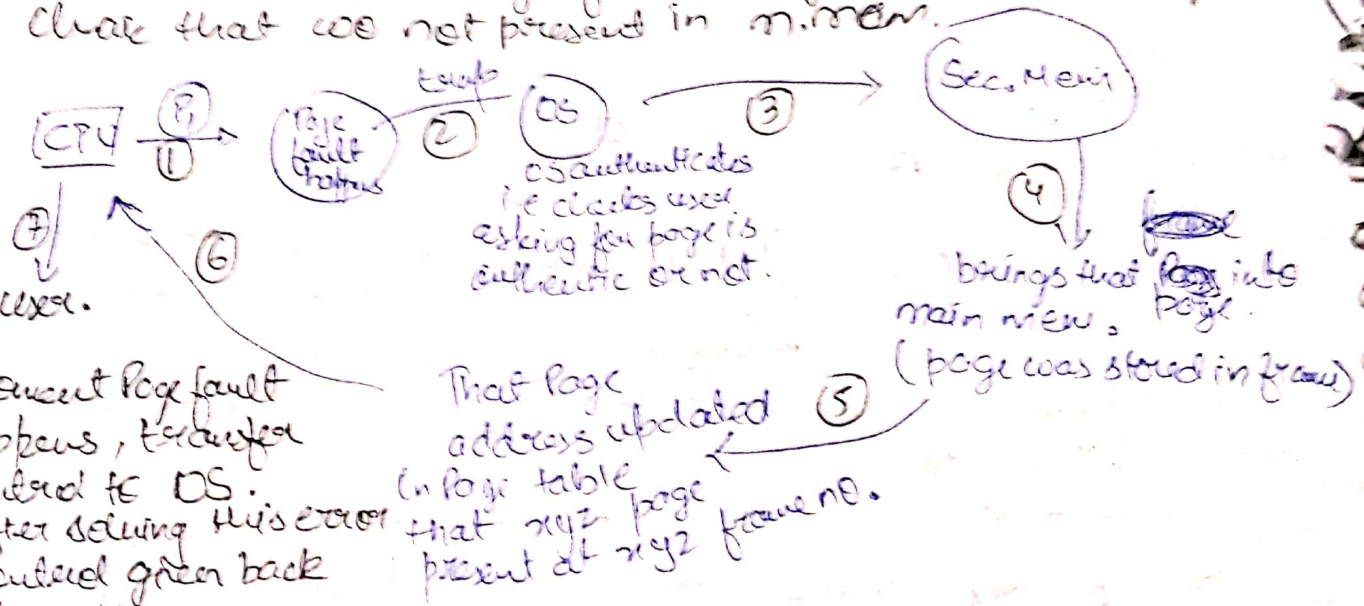
Virtual Mem. (Larger process > M.M size (जो feel करवाता कि it can be executed).

isme we bring only required pages

we bring req. pages and along with it some extra sequential pages brought into m.mem.

→ This keeps happening Jo pages chie वो रहेंगे N.Mem में else from we remove they out of m.mem using Roll In/Swap In & Roll out/Swap out.

* In virtual mem. Page fault may happen, i.e जो Page chaie that was not present in m.mem.



* Moment Page fault happens, transfer control to OS. after solving this error control given back to user

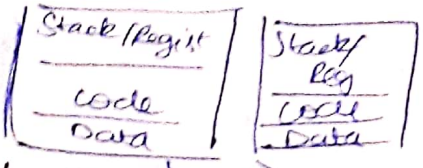
∴ Effective Mem. Access time = p (Page fault Service time (ns)) + (1−p) (Main mem access time (Ma) ns)

p = No. of times page fault happens

(Size Fixed & small) TLB (Translation look aside Buffer)

In PT, PT is stored in m.m & page also stored in m.m
∴ Rather n time to fetch PT from M.M then n time to fetch the page from a frame and in multiple Paging this n will multiply a lot of times.
→ To avoid this time, we use cache mem. (cache/TLB faster than RAM)
★ TLB me we bring few pages sequentially in it
∴ first we look in TLB if not found then we use PT from m.m.
∴ EMAT = $^{Hit}(TLB + n) + ^{Miss}(TLB + 2n)$
(n to bring PT from m.m, n to access page from frame in m.m.)

Process

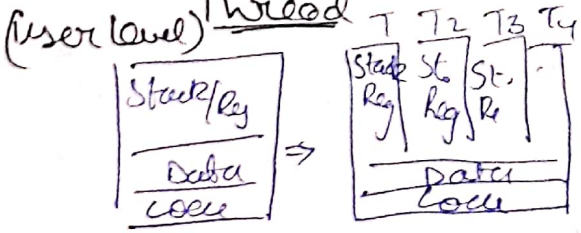


this function creates the clone → fork()

→ Sabkuch copied & recreated using fork() sys. Call

- Sys. Calls involved in Process
- OS treats diff process diff.
- Diff. process have diff copies of data, file etc.
- N times fork() 2^{N-1} clones
- Context switching is slower
- Blocking a process will not block any another process (Block = = I/O req. no more CPU req.)
- Independent.

(User level) Thread



→ Here same data & code used but each thread has its own stack & register

2 types of thre user level & Kernel le Here user level discussed.

- There is no sys call.
- All user level threads treated as single level for OS.
- Thread shares same copy of code and data
- Context switching faster
- Blocking a thread will block entire process.
- Interdependent

User Level Thread

→ Application ki resp to create user level thread

⇒ These are managed by user level libraries

⇒ Faster

⇒ Context switching faster

⇒ If one user level thread perform blocking operation then entire

Kernel Level Thread

→ Kernel level threads are managed by OS (Sys. Calls)

⇒ Slower than user level.

⇒ Context Switching is slower

⇒ If one Kernel thread blocked, No effect on

User level Thread

→ Applicatn ki resp to create user level thread.

⇒ These are managed by user level libraries

⇒ Faster

⇒ Context switching faster

⇒ If one user level thread perform blocking operatn then entire process gets blocked.

Kernel level Thread

→ Kernel level threads are managed by OS (Sys. Calls)

→ Slower than user level.

→ Context Switching is slower.

→ If one Kernel Thread blocked, No affect on others.

Process > KLT > ULT
(CT)

Sys. Calls
to access any OS functionality we must shift to Kernel mode
We use sys. calls to do this. Eg when we want to access file file in Harddisk, to access Harddisk we must go into Kernel mode

* We can directly use sys. call or we can use an API to do this sys. call Eg printf() will use sys call to access write

Brief

Process = Program in running mode

No process wants to use a file, it invokes sys. call as only kernel can access file Process $\xrightarrow{\text{Sys.Call}}$ Kernel.

→ File related sys call eg = open(), Read(), write etc.

→ Device related sys => ioctrl, read, write.
(to access Hardwares previlidge given by OS)

→ Information related sys => get Pid, get sys time & date

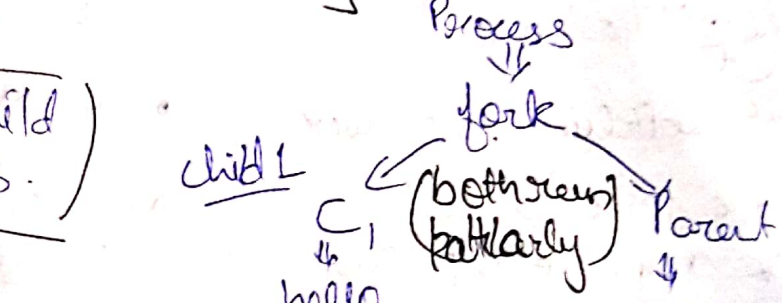
→ Process Control sys calls = fork, wait, signal.

→ Communication => pipe(), shmget (shared mem).

fork() $\begin{cases} 0 & \text{child process} \\ +1 & \text{process itself} \\ -1 & \text{child process not created.} \end{cases}$

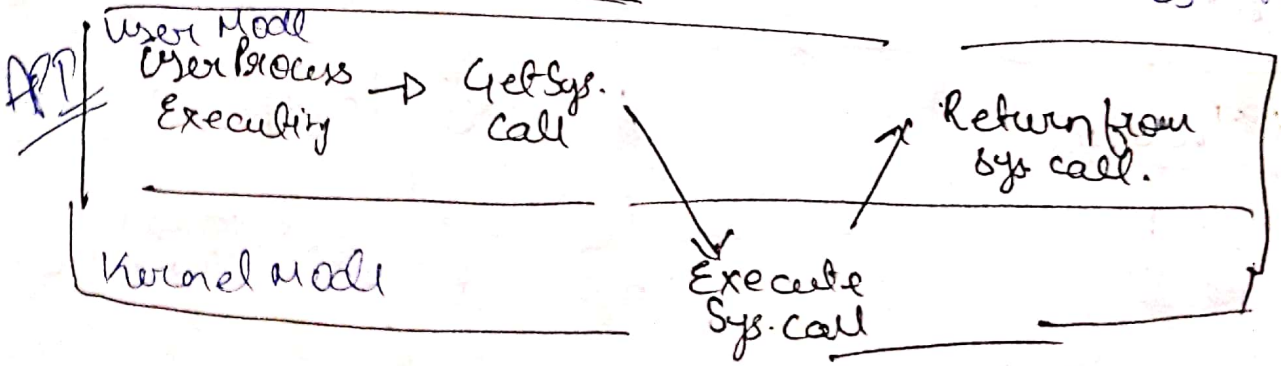
```
main(){
  fork();
  Printf("Hello");
}
```

(N fork creates = 2^N - 1 child Process.)



2 times Hello printed.

User Mode Vs Kernel Mode



PAGE Replacement Algo's (these help in reducing page faults also.)

- in virtual memory swap in & swap out done using these algo's :-

① FIFO
Frame wise FIFO nahi nikalo to replace the first filled frame with req. page

② Optimal Page Replacement
Replace page which is not used in longest Dimension of time in future

③ Least Recently Used.
Past में सबसे पहले कौनसा filled हुआ था उसे Replace Kardo.

* Belady's Anomaly
↳ No. of frames badhane से no. of faults (page faults) also inc. although expected था कि more space देने से since more pages brought in main mem. more hits should happen & page faults should dec. but this didn't happen.

Optimal Page Replacement

आगे आने वाली pages में से सबसे last में कौनसा page आएगा

Like current
f1 2
f2 1
f3 0
f4 7

pg Strng = 7 0 1 2 0 3 0 4 2 3 0 3 2 1 7 0 1
curr → Future
Here out of all pages in frames 7 page will be used in most distant future thus replace it

LRU Least Recently Used

in above eg 7 0 1 2 0 3 0 4 2 - - -
curr ↑ past
2, 4, 0, 7 in frame out of these
7 page was filled in frame sabse pehle ∴ it will be replaced

MSR (Most Recently Used) जो page just abhi use hua उसे

Ho Jaega Padhle.

All the best!

Best of luck