Date: M T W T F S S

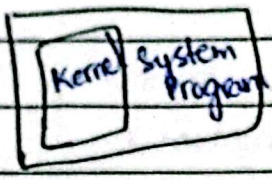
OS

What is OS → Intermediary b/w user & hardware.

Different goals from the user, end & hardware perspective?

convenient, easy to use — resource allocation, security, maximum utilization of CPU.

OS



Application programs

Device driver - manages device controller

Device controller - manages input from user / reads character one by one

Interrupt :- Every interrupt has ID.

Interrupt vector

ID	interrupt service routine
XYZ	=

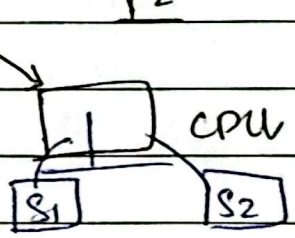
Lets say an interrupt generated, ID matched and detected that it was generated by malicious software so according to interrupt service routine that malicious software is destroyed.

Trap / Exception / Interrupt } Difference?

Interrupt Handling :-

State of system :- registers / stack / PC (Program counter) all this information is called state of system at that particular instance

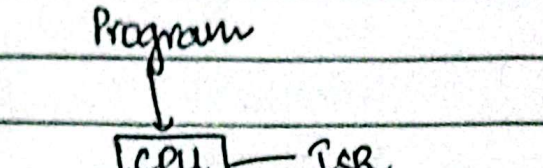
Context switching :-



Lets say Program P_1 was running but some interrupt generated so CPU saves the state S1 at that time and switches to execution of Program P_2. Next state S2 of Program P_2 is saved to return back to P1 from the point where interrupt was generated.

This is called context switching.

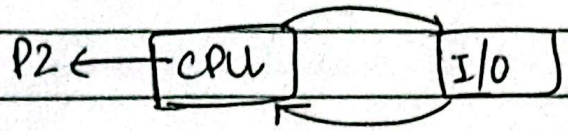
Date:
M T W T F S S



I/O Structure :-

Lets say Program P1 was running on CPU. P1 contained scanf instruction so,

Program P1



Approach 1 - Conventional system.
CPU waits until user provides input and then control is returned to Program P1.

Approach 2 - Modern systems
while user is providing input, the control is returned already even before whole input is provided, so that CPU isnot idle and CPU starts to run some other process P2.

Device status table :- entry for each I/O device

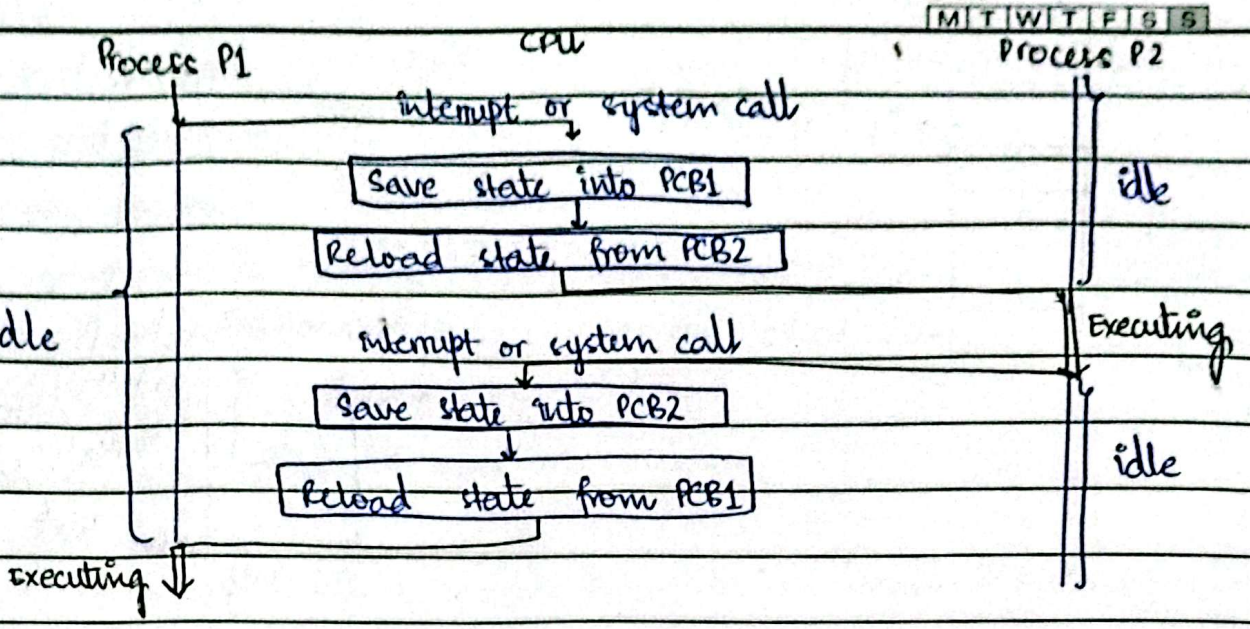
System call :- system call is a way for a computer program to request services from the operating system.

Context Switching :- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.

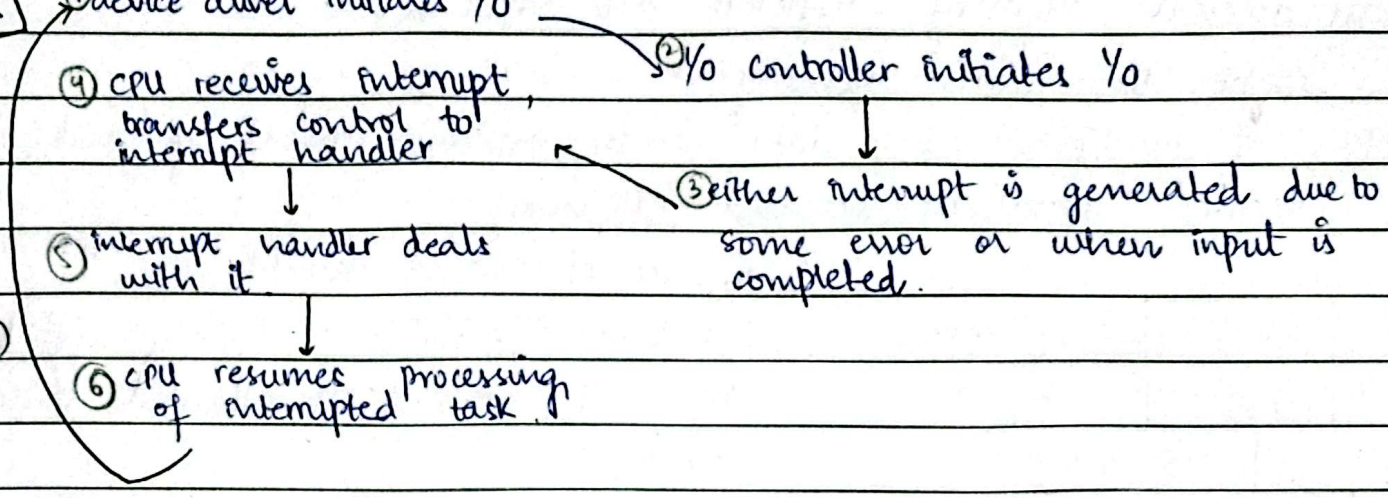
Steps
- OS takes control (through interrupt).
- Saves context of running process in the process PCB.
- Reload context of new processes from the new process PCB.
- Return control to new process.

Date: M T W T F S S

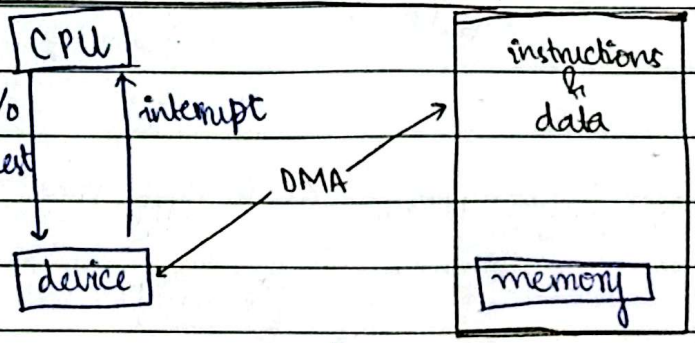
Context switching :-



Interrupt - drive I/O Cycle :- (imp)



How A Modern Computer Works.



context switching → time waste so DMA is used

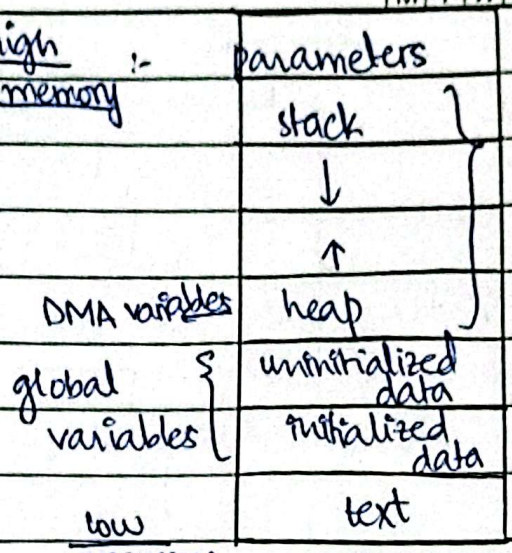
DMA [Direct Memory Access] :- allows peripherals (i.e device, hard drives etc) to communicate directly with the main memory (RAM) without involving the CPU. This process speeds up data transfer and frees up the CPU to perform other tasks.

* Every program in execution is process.

Date: M T W T F S S

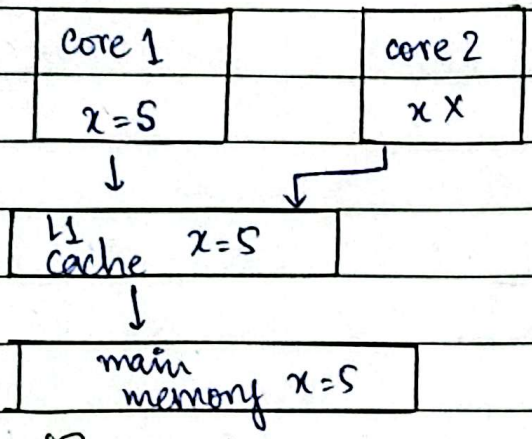
main memory :-

high memory :-



fast accessable memory = registers, but small in size.
then cache then main memory

Purpose of multiple cores :- data can be processed parallely / tasks can be executed parallely, increased efficiency.

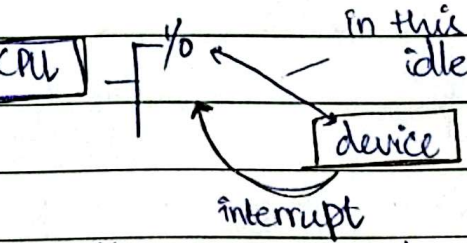


in case of shared variable
Sab takk x variable is being updated by core 1, core 2 is not allowed to access core 2.
when both cores are to work on same variable x.

MULTIPROGRAMMING

main memory	OS
P1	process 1
P2	process 2
P3	process 3
P4	process 4

i)

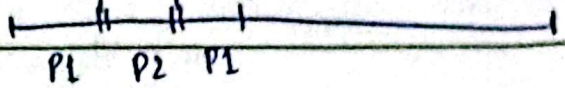


Let's say it's core 1 which will run P1 & P2.
P1 has I/O so by the time CPU is idle, P2 is loaded.
When interrupt is generated that I/O is completed, P2's state is saved and it switches back to P1.

Chapter 1.1, 1.2, 1.3, 1.4, 1.9, 1.10, 1.11
Read

time for context switching → minimal

Date: M T W T F S S



MULTITASKING :-

time sharing :- rapid context switching.
giving user an illusion of simultaneous execution.
CPU switches job so frequently that users can interact with each job while it is running, creating interactive computing.
If there would be no multitasking.
Lets say P1 & P2 was clicked by user at one time.
P1 ← takes 1hr to execute
P2 ← so for P2 to execute, user needs to wait 1hr, both will not run parallely

TIMERS :-

Uses :- time sharing in ① multitasking, generating ② periodic system events, scheduling & process ③ switching.
② → Backup
③ → on the basis of priority, time of execution e.t.c scheduling Algorithms.

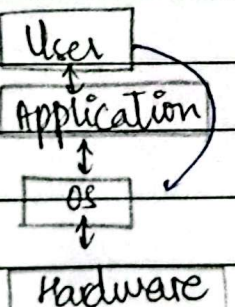
USER MODE VS KERNEL MODE → e.g chmod & sudo commands to switch to kernel mode from user mode.

User: process executing → calls system call → mode bit = 0 → Kernel: execute system call → return from system call → mode bit = 1

mode bit gives indication ki kis mode pr hai. user mode = 1, kernel mode = 0.
This is called DUAL MODE.

Designing OS

↳ should be extensible, reusable so that changes are easy to apply.



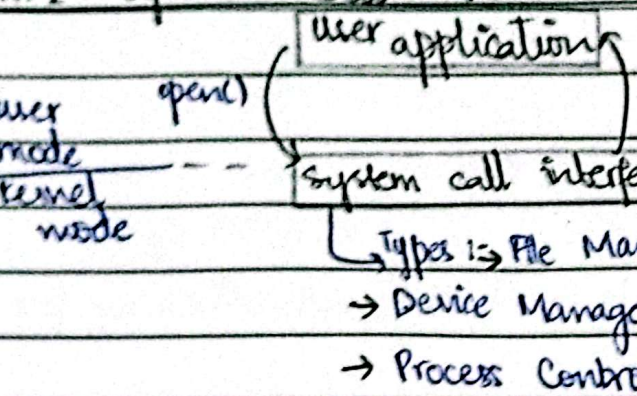
user interface :- three types
① → GUI
② → touch screen
③ → command line

Date
M T W T F S S

resource allocation :- Each process executed is allocated resources which are file size, storage, file rights, time etc.

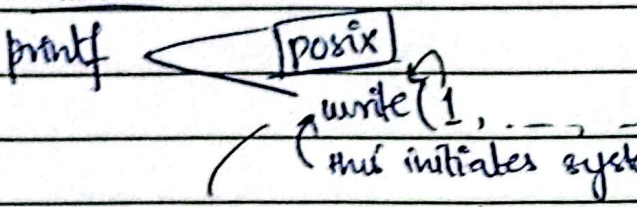
accounting :- keeping track of active processes, resources allocation, deallocation.

API - System Call - OS Relationship :-



system call will always be executed in kernel mode.

stdio.h



generates interrupt to activate syscall. (then mode changes from user to kernel to execute system call).

Printf ki definition is written in stdio library.
write ki definition is written in posix. API.
Har OS ki apni system calls hoti hain.
unistd library holds this posix.

System call Parameters passing :-

methods to pass parameters :-

① → Registers (if parameters are less)
② → block or table (if parameters are more)
③ → pushed onto stack by program & popped off by OS.

#include <sys/types.h> this library has datatypes for OS.
pid_t - process id data type.

total = 1 process

Date M T W T F S S

fork() → After fork() total = 2 processes
↳ this returns 0 for child
returns greater than 0 for parent.

Application
API → write()
OS
HW

printf → write → System write

LINKERS AND LOADER

source program → compiler → object file → linker → executable file → loader → program in memory

main.c --> gcc -c main.c → main.o --> gcc -o main main.o -lm → main --> ./main

linker :- links all other object files → header files, library etc.

Main Memory

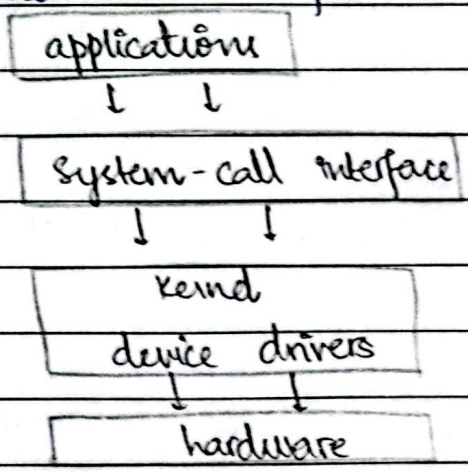
Virtual Address :- 0+4

main memory has Physical Address. (actual address)

loader :- converts virtual address to physical address.
→ brings dynamically linked libraries.

MONOLITHIC :- all OS in one file → speed ↑, efficient
↳ figure 2.13

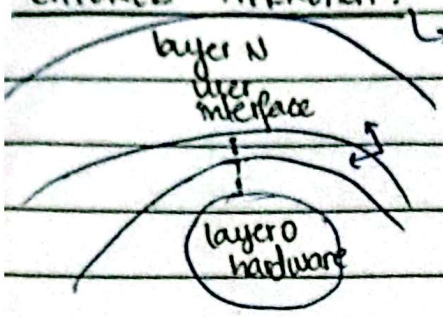
structure →



↳ tightly coupled approach

Date: M T W T F S S

LAYERED APPROACH :-



↳ reusable, loosely coupled

these two layers can communicate

every two consecutive layers can communicate.

performance slow as compared to tightly coupled approach.

MICROKERNELS :-

↳ Kernel ka size small rakha, kernel mai sirf important kaam hoga

↳ interprocess communication, memory management, CPU scheduling

↳ this will run on kernel mode.

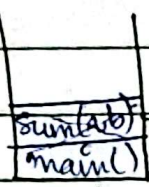
↳ File system, application program, device drivers

↳ this will now run on user mode. (although they will have communication with kernel).

overhead due to message passing.

ACTIVATION RECORD :-

```
main() {
  int a;
  int b;
  int c = sum(a,b)
```



→ activation record
(↳ values of local variables, return address, return type)

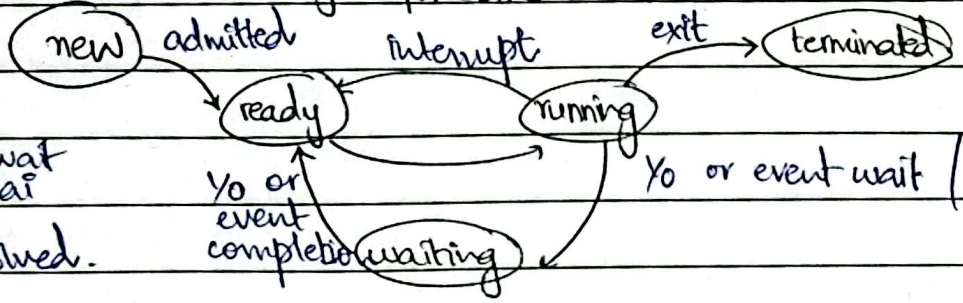
Add c=a+b

sum(a,b)

After execution of function, it is popped of from stack.

PROCESS STATES :- New, Running, Waiting, Ready.

waiting :- waiting for some I/O (such as I/O completion).



*Ready ki wait state mai queue ki involved.

I/O or event wait (for e.g :- scanf instruction).

■ interrupts

Ready ← Running due to multitasking, to run ISR (interrupt service routine)

* object containing the saved state of process this is called → PCB (Process Control Block).

↳ wo jaake queue mai save hota

PCB contains info such as :- process state, process number, program counter, registers, memory limits, list of open files.
↓
TCB also called task control block

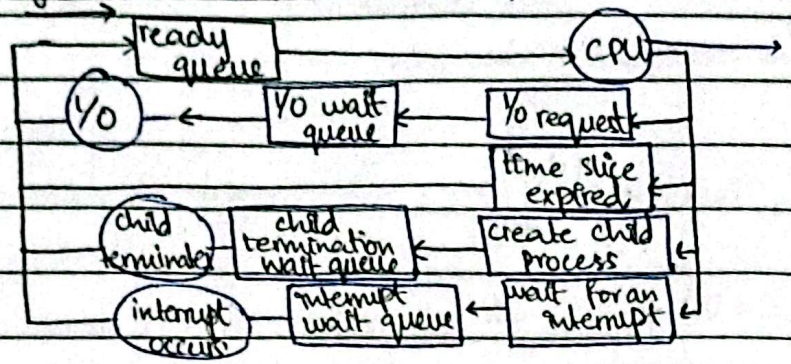
Date: M T W T F S S

currently memory mai kitne processes hain ⇒ degree of multiprogramming

I/O bound process :- more I/O based instructions

CPU-bound process :- more CPU, computations based.

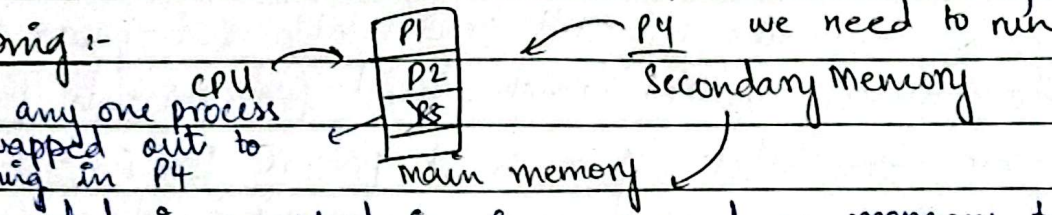
Queuing diagram representation of process scheduling



① I/O request → process will be placed in I/O wait queue
② time slice expired → during multitasking, to do context switching
③ child process → wait queue
④ wait for an interrupt

CPU Scheduling :- CPU scheduler → select processes from ready queue and allocated a CPU core to one of them on basis of Scheduling Algo's.

Swapping :-



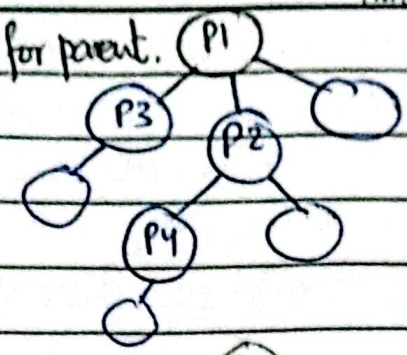
Process needed is swapped in from secondary memory to main memory.

PROCESS CREATION :-

```
int main()
→ pid = fork() ——— 1 process P1
→ ———————— 2nd process P2 (child process of P1)
    fork()
→ ——— ——— 3rd process P3 (child process of P1)
                P4 (child process of P2).
```

→ fork() creates a child process.
↳ fork() ek system call jab execute hota hai toh
it returns 0 to child process and >0 for parent.

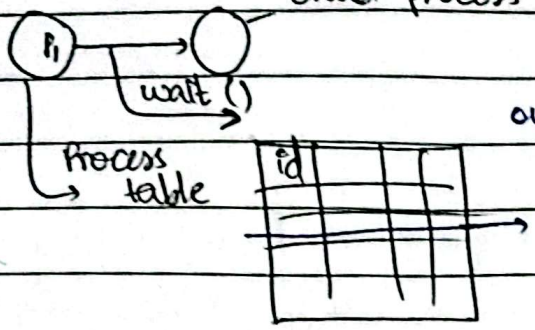
Date: M T W T F S S



Q. what if we want to create this hierarchy?

```
pid_t = pid
pid = fork()
if (pid < 0) {
    fprintf (stderr, "Fork failed");
    return 1;
} else if (pid == 0) {
    fork();
}
```

child process



* Parent process is responsible for clearing out child process.
when child process terminates, its parent's process removes id of that child process.

Zombie process :- → To alive raho but still process table mai nazar aye.
This is possible when wait() isn't used for parent process and parent process terminates before child process terminates.
A zombie process is a process that has completed its execution but still has entry in the process table.

Orphan process :- An orphan process is a process whose parent process has terminated or exited, leaving the orphaned process running. (leaving it to be adopted by init (PID 1).

Cascading termination :- refers to a situation where terminating a parent process results in termination of its child processes.
child process terminated with parent. [to prevent them from running without supervision].

y address space
main memory.

Date: M T W T F S S

execlp, ~ iis statement ki waja se child wala code directly returns from else if { }

pid = wait (&status)
↳ this tells the status k process kis status se terminate hua.
(was it successful termination ya kisi error ki waja se terminate hua.

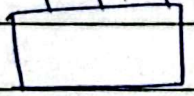
Zombie process:- resources consumption, entry present in table, parent is responsible to delete entry.

orphan process:- parent process na ho, orphan process, OS ka init ka process usko adopt krleta, Parent pehle hi execute hogaya ho.

Cascading termination:- parent process khud terminate krdeta child ko, this happens when :- child process tried to access ① resources that weren't allowed, when task done ② by child process isnot allowed.

thread → Kisi process ki running entity. EK process mai multiple threads exist krsakte

Program → thread alag se memory hold nhi krte whereas child processes k lie alag memory hold hote.
Process
Thread
{ { { → user level threads
| | | → kernel level threads.


CPU core

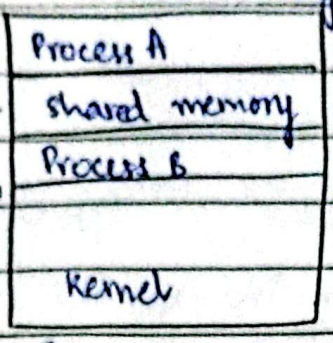
user level threads resides on kernel level threads to execute.

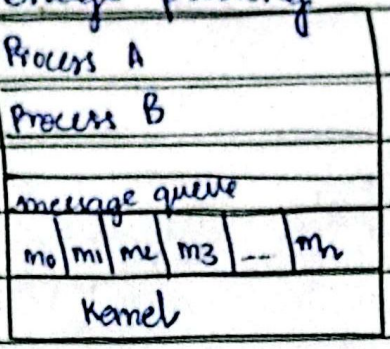
Process < Independent process (no communication)
Cooperating process (that communicates with other processes).

Q WHY IS THERE NEED FOR COMMUNICATION?
① when they want to share information.
② Computation speedup (for e g matrix multiplication < add / multiplication > result)
③ Modularity

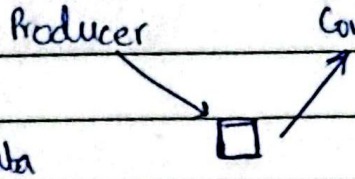
Date: M T W T F S S

Process communicate by ① Shared memory ② Message passing



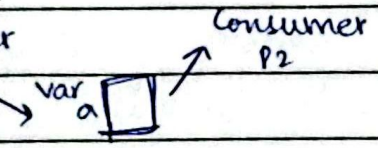


PRODUCER - CONSUMER PROBLEM :-



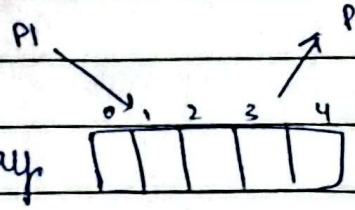
→ let's say, producer is generating / writing data & consumer is accessing data to read it or compute it.

P1 is writing on variable 'a' & P2 is reading data on variable 'a'.



Problem :-

↳ agar speeds same nhi hain different computation speed of producer & consumer results in data loss



↳ Data loss → different computation speed.

to solve it hum ek loop lagaen

```
{
  a[i] = x
}
```

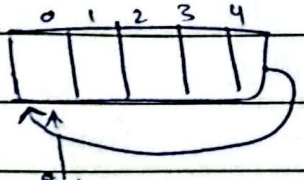
agar koi jagah empty ho toe data wahin write ho.

Unbounded buffer :-

implement using linked list.

bounded buffer :-

implement using array.



Circular array

similar to circular queue of DS

Code for Producer :-

```
(((in + 1) % BUFFER_SIZE) == out)
```

↳ condition when full array is filled.

otherwise buffer[in]

```
in = (in + 1) % BUFFER_SIZE;
```

in=0 out=1

0	1	2	3	4
x	y	z	w	u

Q) (0+1) % 5 = = 1

in = 0, out = 1 → 1 = = 1 It goes in do nothing loop.

where does program go?

Date: M T W T F S S

CHAPTER 1

Difference between Trap, Exception and Interrupt.

Type	Sync/Async	Source	Intentional	Examples
Exception	Sync	Internal	No	Overflow, divide by zero, illegal memory access
Trap	Sync	Internal	Yes and No	System call, page fault, emulated instruction
Interrupt	Async	External	Yes	I/O device completion.

PRODUCER CODE:-

fd = shm.open(name, O_CREAT | O_RDWR, 0666);

name → title of shared memory; 0666 → permission; 0777 → highest permission

fd :- file descriptor

ftruncate (fd, size)

ptr = (char*) → starting address

↳ same address return hota, address of main memory jahan shared memory segment bana hai.

mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

0 → identifier; PROT_READ | PROT_WRITE → Read and write; MAP_SHARED → tag that it is shared.

CONSUMER CODE:-

fd = shm.open(name, O_RDONLY, 0666);

No ftruncate only producer do ftruncate; O_RDONLY → Read only

ptr = (char*)

mmap (0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, MAP_SHARED, fd, 0);

COMMUNICATION:-

send (message) , receive (message)

Naming:- send (P, message) , receive (Q, message)

Send a message to process P; Receive a message from process Q.

PIPES:- Ordinary pipe:- Parent ↔ child communication, one way P → C, C ✗ P

Named pipe:- any processes not having parent child, two way

Date: M T W T F S S

fd[0] → reading , fd[1] → writing

Ordinary pipe code:-

Create pipe pipe (fd) system call

```
if (pipe (fd) == -1)
    "ERROR";
```

child ke pehle pipe create kro. so that child will have access to this pipe.

```
pid = fork();
if (pid > 0) {  // Parent process → is sender
    close ( fd[0]) ;
    write (fd[1], write_msg, strlen(write_msg)+1);
    close (fd[1]);
}
else {
    close (fd[1]);
    read (fd[0], read_msg, BUFFER_SIZE);
    close (fd[0]);
}
```

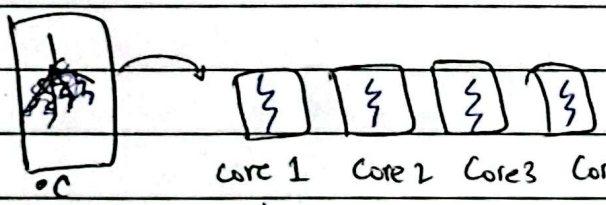
CONDITIONS WHERE PIPES ARE NEEDED:-

① Parent is taking input & child ko bhejna wo input.

② Incase of CSV file, parent needs to send msg to child.

3.8.2 → Read

CHAPTER 4 :- THREADS



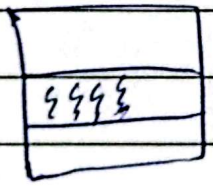
T → T/4

Multithreading

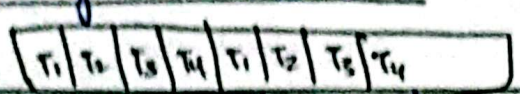
Multithreaded need :-

One thread is serving request, other thread is receiving request.

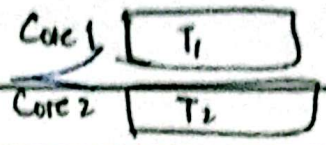
thread → light weight process → consumes less resources



Concurrency vs Parallelism:-

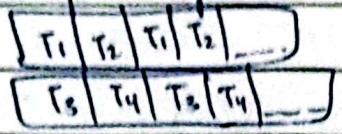


concurrency → each thread makes progress
↳ all tasks making progress.



Parallelism → more than one tasks running simultaneously

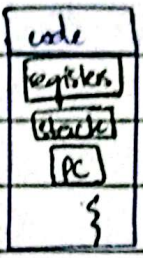
In single core computer systems → concurrency exists without parallelism.

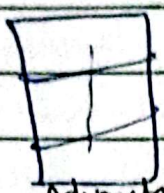


} concurrency + parallelism

Threads:-

Each thread has its own registers, stack and PC.





1000 lines of code taking 1000 sec.
divide it into 3 threads. time = 1000/3 = 333 sec.

Advantages of threads:-

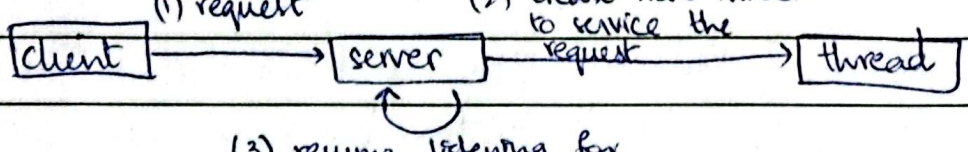
Adv:
↳ All things are shared in threads so no need to do interprocess communication
↳ Efficiency — less time taken.
↳ Multithreading enhances availability.
↳ Economy - cheaper than process creation.
↳ Thread switching lower overhead

sum=0 global variable

Program to do avg = sum/N running on diff. threads.
we need to take care of synchronization taake ek time par har thread uss variable mai na add kre.

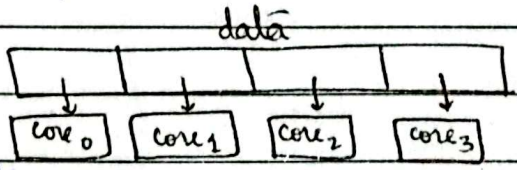
sum = sum + x ← to solve inconsistency/garbage value issues.

MULTITHREADED SERVER ARCHITECTURE:-



Data level Parallelism:-

↳ large dataset
↳ same instruction execution on multiple data elements

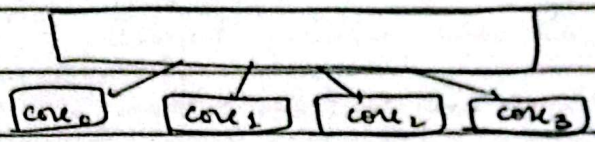


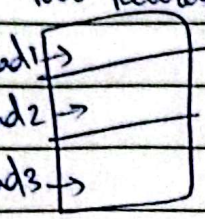
data parallelism

Date: M T W T F S S

Task level parallelism :-

different tasks / running on (same data or diff. data).
different operations

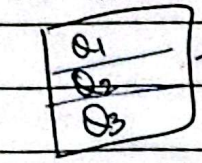




DLP ⇒ Data divided, but each performing same task

TLP ⇒ sum, Avg, Std(x), Var(x) different tasks/operation on data (or by dividing data).

In TLP data can be same or divided.



$\frac{100}{3}$ = 300's every TA check 300's paper ⇒ DLP

TLP → 1TA checks all Q_1
2nd TA checks all Q_2
3rd TA checks all Q_3

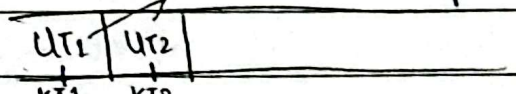
AMDAHL's LAW :-

$$\text{speed} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

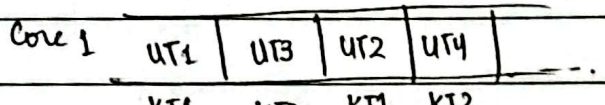
S — serial portion, (1−S) — parallel portion (Parallelizable)

User threads → managed by programmer

user threads → maps onto kernel threads



kernel thread



If anyone among UT1, UT2 is blocked, it affects all those threads attached to same kernel threads. [one blocked thread blocks all threads in the process.]

Many-to-one { { { user threads
↳ blocking behaviour { Kernel thread

One-to-one { { user thread
Kernel thread { {
↳ Restriction :-
can make limited # of user thread

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM), refers to the domain in which threads compete for resources such as processor time

Date: M T W T F S S

THREAD SCHEDULING :- → depending on the Contention Scope

(PCS) Process Contention Scope <many-to-one / many-to-many PTHREAD_SCOPE_PROCESS> (SCS) System Contention Scope

using LWP (light weight process).

one-to-one model, PTHREAD_SCOPE_SYSTEM

SIGNAL HANDLING

→ Ctrl+C to terminate program so Ctrl+C is basically interrupt/user generated → so by default we signal pr we terminate karaha.

→ User-defined signal handler → override default

THREAD CANCELLATION :-

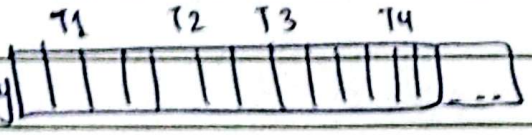
↳ target thread :- thread to be cancelled.

Asynchronous cancellation → terminates the target thread immediately

Deferred cancellation → allows target thread to periodically check if it should be cancelled.

↳ to cancel we use pthread cancel & pass the id of the thread to be cancelled as parameter.

Q.) Discussion. Array with 10 000 00 values. Task is to compute sum of all indexes. We want to do this task by creating threads.



we created 4 threads, each will calcula

Threads = 4 sum.

Each thread will execute sum_array function.

```
pthread_t threads [NUM_THREADS] → to save ids of threads
int thread_args [NUM_THREADS]
void *thread_results [NUM_THREADS]
for (int i=0; i< NUM_THREADS ; ++i) {
    thread_args [i] = i;
    pthread_create (&threads [i], NULL, sum_array, (void
```

pthread_join ka used taake parent wait kre jab tak thread computation krke wapas aaye.

Date: M T W T F S S

```
void *sum_array (void *arg){
    int thread_id = *((int *) arg);  → typecast krke thread_id mai store krwaya
    int start = thread_id * (ARRAY_SIZE / NUM_THREADS);
    int end = start + (ARRAY_SIZE / NUM_THREADS);
    int sum = 0;
    for (int i = start ; i < end ; ++i) {
        sum += globalarray[i];
    }
    return (void *)
}

for (int i = 0 ; i < NUM_THREADS ; ++i) {
    pthread_join (threads[i], &thread_results[i]);
    total_sum += (long) thread_results[i];
}
```

THREAD TRACE :- `int rc;`

`rc = pthread_create (&p1, NULL, mythread, "A"); assert (rc == 0);`
`rc = pthread_create (&p2, NULL, mythread, "B"); assert (rc == 0);`

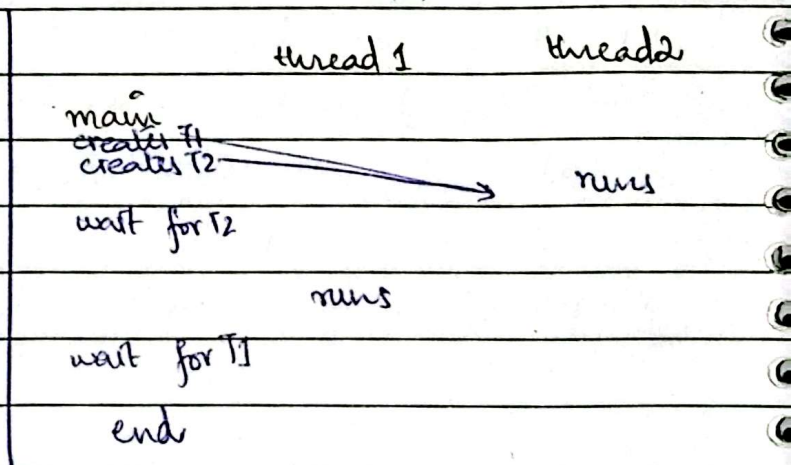
returns 0 if successful execution.

returns non-zero if not successful.

`rc = pthread_join (p1, NULL); assert (rc == 0);`
`rc = pthread_join (p2, NULL); assert (rc == 0);`

Thread trace :-

	T1 thread 1	T2 thread 2
main creates T1		
creates T2	→ runs	
wait for T1		
	runs	
wait for T2		
end		



Thread 1 Thread 2

main
runs
creates Thread1 → runs returns
creates thread2 → runs returns
waits for T1
waits for T2

Thread traces depends on scheduling / timesharing etc.
If FCFS (First come first serve basis) is used :-
main T1 T2 Main T1 T2
X X X →

RACE CONDITION :- counter → Global variable
Consider two threads :- T1 and T2

T1	T2
Counter ++	Counter ++

* Both threads are doing counter ++
↓ If in assembly breaks into 3 lines
1) load
2) inc
3) store

Main
main
creates thread1
creates thread2

T1

→ timesharing mechanism

Core 1

T1	T2	T1	T2	T1
load	load	inc	inc	store
counter=4	counter=4	counter=5	counter=5	counter=5

load T1 inc store C++
load T2 inc store C++

T2
store
counter=5

And 6 ava chaie tha but 5 araha this 5 is inconsistent value. This is called race condition.

If both threads use a shared variable, so race condition occur.
counter should have value 6 but it is having value 5 → race condition.
atomic operation :- Ex :- ATM transaction → either transaction is completed or not.
↳ runs uninterruptedly on computer.
Atomic init solution for race condition

Date:
M T W T F S S

P0 , P1 doing fork() system call.
fork()

both can have same next_available_pid due to race condition.
↳ this will happen agar proper synchronization apply nhi hua hai we

Critical Section Problem :-

critical section :-

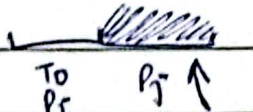
```
while (true) {
    entry section
        critical section → use part of code jismain shared variable modify horaha jisey
    exit section                counter++ / counter--;
}
```

HOW TO SOLVE CRITICAL SECTION PROBLEM.

1) At a time ek process hi shared variable ko access krraha.

Process Pj
```
while (true) {
    while (turn == i) ;  → If condition false (turn != i) tbe it will run critical section.
        Critical section
    turn = i;
    remainder section
}
```

Process Pi
```
while (true) {
    while (turn == j) ;  ⇒ Busy waiting / Busy looping
    critical section
    turn = j;   ← upon exiting critical section Pi will assign turn to j, so that Pj can now access shared variable as it would be waiting till now.
    remainder section
}
```



Busy waiting or busy looping.

Date: M T W T F S S

3 REQUIREMENTS :-

1) Mutual exclusion :- Both cannot accessing shared variable at the same time.
2) Progress :- assigning turn = j
3) Bounded waiting :- concept of bounded waiting taake aisa na ho k ek process repeatedly turns leta rahe aur dosra long time k like wait krta rahe.

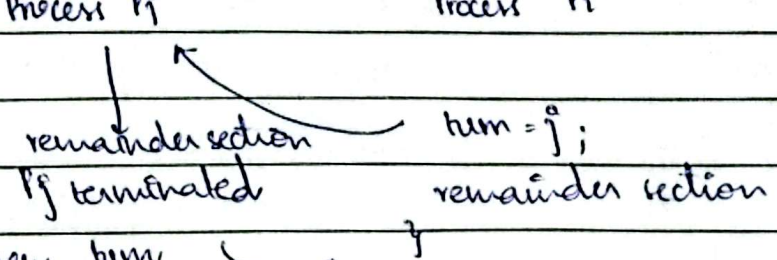
Rn in Software based approach :-

Bounded waiting is implemented as we are assigning turn at the end of critical section. turn = j; turn = i; > this ensures k ek process ki repeatedly turn nhi arahe.

Busy waiting / Busy looping refers to process synchronization technique where a process repeatedly checks for condition to be true, consuming CPU cycles while waiting, rather than relinquishing the processor & blocking.

Problem in Software based approach :-

What if my process P_j wants to terminate (donot want to access critical section again) whereas process P_i wants to access critical section again.



Now P_i is not given turn again

P_i will keep waiting for its turn infinitely.

To resolve this problem, we use PETERSON'S SOLUTION.

P_i P_0	P_j P_1

while (true) {
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
critical section
flag[i] = false;
remainder section
	while (true) {
	flag[j] = true;
	turn = i;
	while (flag[i] && turn == i);
	critical section
	flag[j] = false;
	remainder section
	}

Date: M T W T F S S

Problems in Peterson Solution

1) It is not scalable → not when we have too many processes.

2) Instruction Reordering → modern architecture reorder those instructions that are independent of each other

```
a = a+2 }
a++     } instructions dependent of each other.
a--     }

a = 7;  }
b++;    } Instructions independent of each other
c = 5;  }
```

Instruction reordering of `flag[i] = true;` / `turn = j;` these two instructions will create an issue. → Due to this ek point aega jab dono processes apne critical section ko ek sath access kreinge.

DRYRUN FOR PETERSON:- Code same as previous page. $P_i^{P_0}$; $P_j^{P_1}$

Step	Process	Action	turn	flag[i]	flag[j]	Remarks
1	P_0	flag[i] = true	P_0 (i)	true	false	P_0 sets its flag to true.
2	P_0	turn = j	j	true	false	Turn assigned to P1.
3	P_0	check flag[j] && turn==j	j	true	false	true && false → False → P_0 will enter CS.
4	P_1	flag[j] = true	j	true	true	P1 sets its flag to true
5	P_1	turn = i	i	true	true	Turn assigned to P_0.
6	P_1	check flag[i] && turn==i	i	true	true	true && true → True → P1 will do busy waiting.
7	P_0	Enters critical section	i	true	true	P_0 exists, resets flag[i] = false
8	P_0	Exits critical section	i	false	true	P_0 enters critical section
9	P_1	Enters critical section	i	false	true	P1 enters critical section
10	P_1	Exits critical section	i	false	false	P1 exists, resets flag[j] = false.

Date: M T W T F S S

Atomic :- run uninterruptedly.
either ek sath execute honge ya nhi execute honge
Hardware instructions
CAS (compare-and-swap):- Boolean variable
if (lock == F)
{ lock = True;
}

MUTEX locks :- acquire() & release() ⇒ Atomic ⇒ run uninterruptedly.
backend pr CAS (compare & swap) hi use hota hai isme.
Busy waiting :- while( ); ⇒ utilises CPU cycles, wastes CPU cycles.

```
pthread_mutex_lock (&mutex);
// CS (critical section)
pthread_mutex_unlock (&mutex);
```

```
int main() {
pthread_mutex_init (&mutex, NULL);
|
pthread_mutex_destroy (&mutex);
}
```

Adv. of busy wait: Agar busy waiting hogi for short period of time, so it is useful since it saves the overhead of context switching or sleeping.

Mutex with trylock :-

```
if (pthread_mutex_trylock (&mutex) == 0) {
    // CS
    pthread_mutex_unlock (&mutex);
} else {
    |
}
```

* Busy waiting is also called spinlock.
* When longer duration of spinlock ⟶ use sleeping.

WHY USE SEMAPHORE ?

```
acquire (lock)
reading ---
a = x + 2
release (unlock);
```

x variable is used just for reading here, so no issue of race condition.

* So in this case, mutex locks wala solution is not appropriate since it prevents two or more threads running parallely to access x. That's why we use Semaphore.

semaphore	wait(S)	post(S)
int S;	{ while(S<=0) ; // busy wait	{S++;
↑ initialise	S--;	}
	}	

```
void *threadfunc(void *arg){
  sem_wait(&sem); // wait()
    // CS
  sem_post(&sem); // signal()
  return NULL;
}
```

S = 2 ka matlab at a time 2 threads runs that code parallely.
↳ wo part of critical section jahan sirf Reading horahe.

* When critical section has reading wala code use Counting semaphore
* When critical section has writing wala code use Binary semaphore/Mutex locks.

T_2	T_1	T_3
a = c+1	c++;	b = c+1;
c is used for reading	c is used for writing	

① RR → use counting semaphore
② RW → Binary semaphore/mutex locks
③ WR → Binary semaphore/mutex locks
④ WW → Binary semaphore/mutex locks

} Mutual exclusion is required (allowed).
↳ No two threads should access critical section at the same time.

* Semaphore is also used for instruction ordering.

This code always ensures that thread1 runs before thread 2.
↳ slides wala code
↳ slides Pg 51

Date: M T W T F S S

Semaphore Implementation with no Busy waiting :-

↳ block instead of busy wait

↳ waiting queue □→□→□

```
wait(semaphore *S){
    S->value--;
    if(S->value<0){
    add this process to S->list
    block();
    }
}
```

```
signal(semaphore *s){
    S->value++;
    if(S->value<=0){
    remove a process P from S->list;
    wakeup(P);
    }
}
```

```
typedef struct{
    int value;
    struct process *list;
}
```

LIVENESS :- Scenarios that can occur when processes running concurrently :-

① Deadlock :- Threads block indefinitely, each holding a resource needed by another. For e.g.- traffic situation  3 cars in deadlock

Let's say P_1, P_2

P_1	P_2
• resource A	• resource B
• resource B	• resource A
• A & B	• A & B

Let's say P1 runs first — acquires A, acquires B, resource B not available, resource A not available — Deadlock has occured.

P_1	P_2	P_1	P_2
T_1	T_2	T_3 Busy wait	T_4 Busy wait

Solution :- Make any process exit so that resources are available.

② Starvation :- koi ek process / ya processes wait kre jaarahe for entering critical section. Let's say CPU runs SJF (shortest job first) algorithm so some processes have to wait repeatedly.

③ Livelock :- Process start hota interrupt ajata phir start krta phir interrupt ajata so process isnot able to run that task completely due to repeated interferences.

CHAPTER 7

7.1 CLASSICAL PROBLEMS OF SYNCHRONIZATION

Date: M T W T F S S

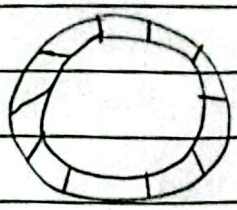
Producer - Consumer Problem :- Bounded Buffer Problem → n: buffers

```
Int n;
Semaphore mutex = 1;
Semaphore empty = n;
Semaphore full = 0
```

- Mutex binary semaphore provides mutual exclusion for accesses to the buffer pool & is initialized to the value 1.
- semaphore empty is initialized to n.
  n —(decrement)→ 0 } wait (n--)
- semaphore full is initialized to 0.
  0 —(increment n++)→ n } signal.

PRODUCER PROCESS :-

```
while (true) {
    /* produce an item in next produced */
    wait (empty);   ensure that we have ample buffers available.
LOCK → wait (mutex);
    // critical section
    /* add next_produced to the buffer */
UNLOCK → signal (mutex);
    signal (full);
```



CONSUMER PROCESS :-

```
while (true) {
    wait (full);
LOCK → wait (mutex);
    // Critical section
    /* remove an item from buffer to next_consumed */
UNLOCK → signal (mutex);
    signal (empty);
    /* consume the item in next_consumed */
    ...
}
```

we can interpret this code as producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

7.1.2 READERS - WRITERS PROBLEM

Ex:- database is shared among several concurrent processes some may want to read only, others may want to update (read + write).

RR → counting semaphore, No synchronization issue

RW, WR, WW → mutual exclusion is required. Synchronization issue

Readers - writers has several variations.

First problem → writers may starve — we allow readers to read, & make the writers wait

Second case → readers may starve — we allow writer to write, & make the readers wait.

Date: M T W T F S S

Solution to first readers-writers problem, the reader processes share the following data structure :-

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

↳ to keep track of how many processes are currently reading the object.

rw_mutex is used for mutual exclusion semaphore for writers.

Writer process :-

```
while (true) {
    wait (rw mutex); lock
    //CS
    writing performed
    signal (rw mutex); unlock
}
```

Reader process :-

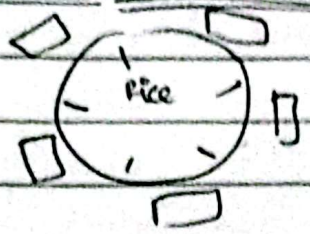
```
wait (mutex); lock
read count++;
if (read count == 1)
    wait (rw mutex);
signal (mutex); UNLOCK
    /* reading performed */
wait (mutex); lock
read count--;
if (read count == 0)
    signal (rw mutex);
signal (mutex); UNLOCK
```

DRY-RUN :-

```
wait (mutex) ; acquires lock
read_count++;   read_count = 1
if (read_count == 1) {
    wait (rw_mutex);   // it does busy waiting if writer has already acquired lock
// if acquires lock rw_mutex when writer does signal (rw_mutex)
    signal (mutex);
// Reading performed   // Next reader directly come here
    ...                 RR allowed
wait (mutex);
read_count --
(if read_count == 0)  // For last reader
{ signal (rw_mutex); }
signal (mutex);
}
```

DINING - PHILOSOPHERS PROBLEM

Date: M T W T F S S



5 philosophers, 5 chopsticks

↓ every philosopher need to acquire 2 chopsticks / semaphores.

↓ semaphore

Problem :- If all philosophers feel hungry at the same time ⇒ DEADLOCK.

```
wait (left chopstick)
wait (right chopstick)
  ...
signal (Right chopstick)
signal (left chopstick)
```

Deadlock — what if all want to eat at the same time?

① ↳ Philosopher sab ko pick kre chopstick ko jab dono available hon.

② ↳ odd numbered philosopher → picks up first her left chopstick, & then her right chopsticks.

Even numbered philosopher → picks up first her right chopstick, & then her left chopstick

③ ↳ Place an extra chopstick n+1.

Producer - Consumer problem
Reader - writers problem
Dining philosophers problem
} In chap 7 only these included.

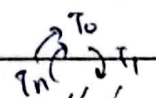
CHAPTER 8 :- DEADLOCKS discussed in chap# 6 liveness.

Process P_0	Process P_1
wait (R sem1)	wait (R sem2)
wait (R sem2)	wait (R sem1)
signal (R sem2)	signal (R sem1)
signal (R sem1)	signal (R sem2)

→ Deadlock

Necessary Conditions :- deadlock situation arise if following 4 conditions hold simultaneously.

1. Mutual exclusion
2. Hold and wait
3. No preemption :- Resources cannot be preempted
4. Circular wait — T_0 waiting for T_1, T_1 waiting for T_0.



CIRCULAR WAIT

* If we ensure 4 in mai se koi bhi cheez nhi hai toh we can prevent deadlock, for eg :- if we ensure preemption, it will ensure no deadlock.

Date: M T W T F S S

Deadlock prevention:- Deadlock can be prevented by ensuring that atleast one of the necessary conditions cannot hold.

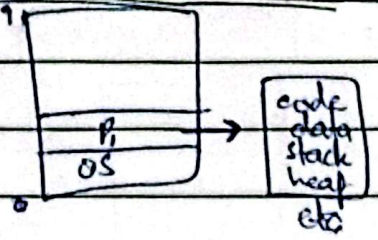
Deadlock avoidance:- threads to tell OS which resources that thread will be needing & using during its lifetime. OS will then decides k us thread ko create krna ya nhi --- OS runtime pr decision leatha k thread create krna ya nhi (depending on the resources it needs).

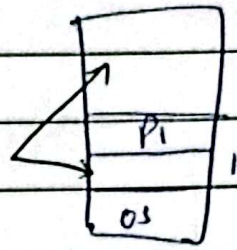
Chapter 8 mai bass yehi chezain hain.

CHAPTER # 9

during execution (Main memory) — Program (secondary memory)

CONTIGUOUS MEMORY ALLOCATION:-





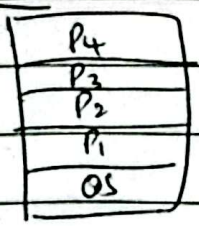
holes

$P_2 = 110kb$

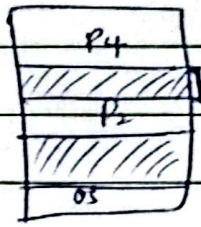
P_2 wilnot be stored since ek sath 110kb available nhi

* Ek sath poorey process ko load krwana ek jegah → This is concept of contigious memory allocation.

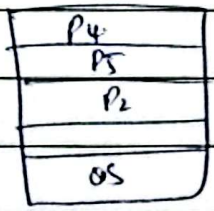
Holes → available spaces in memory.



Let's say P_1 & P_3 terminated & leaves variable partition/ variable-sized holes.

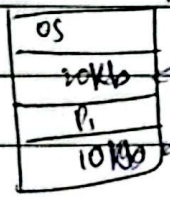


Let's say P_5 requires 28kb, toe



OS us 2kb ko alag nhi krega, balke P_5 ko hi assign krdega kyunk OS ko idea hai ke 2kb ka alag process nhi aega.

Another example:



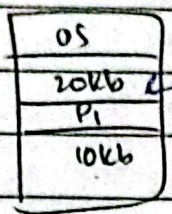
$P_2 = 8kb$

P_2 kahan store hoga?

↳ depends on technique.

Date: M T W T F S S

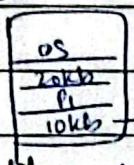
First-fit :- Allocates the first hole that is big enough. Start se search krega k kahan space available.



P_2 = 8kb

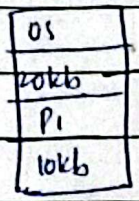
when using first-fit P_2 will be stored in 20kb hole.

Best-fit :- optimal - usme store krega jisme wastage kam ho.
Allocate the smallest hole that is big enough.



P_2 will be stored here.

Worst-fit :- Allocate the largest hole.
entire blocks ko search krk sabse ziada available space ko allocate krta.

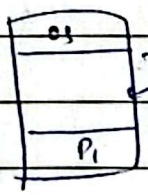


P_2 idhar save krega

FRAGMENTATION :-

External Fragmentation :- Ek sath 130 kb ki memory available nhi -- space (not contiguous.

Internal Fragmentation :-

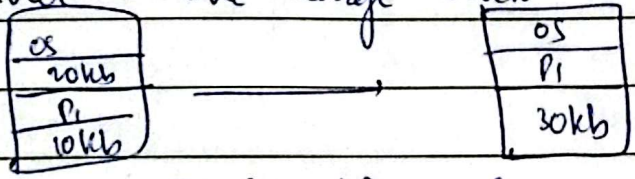


P_2 8846 kb

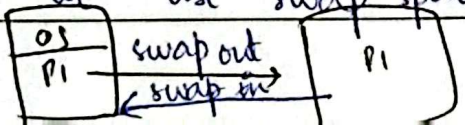
you allocate that extra 2kb space also to that P_2.

SOLUTION :-

(1.) **Compaction :-** shuffle the memory contents so as to place all free memory together in one large block



lekin reshuffling ki waja se P_1 ka base address change hojaega
oe isk lie we use swap space.



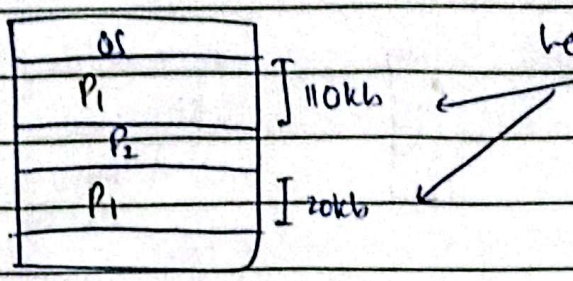
swap space (backing store)

Date:
M T W T F S S

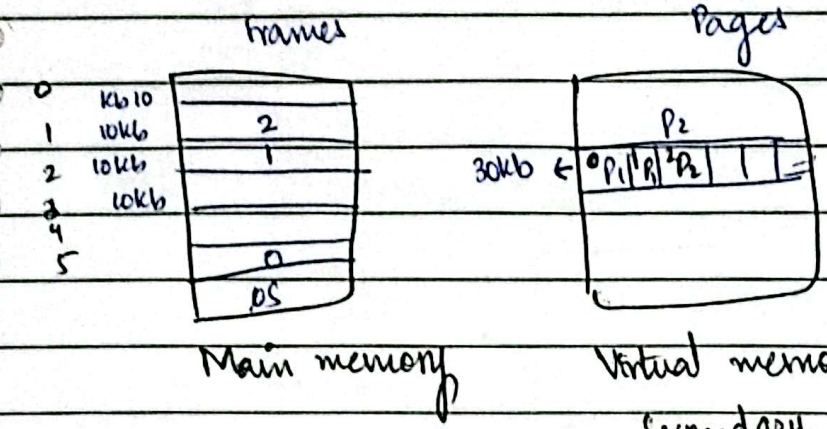
Paging :- section 9.3

contiguous memory allocation

Paging ← process break hoke store hota ← all partitions will be of same size.



Let say P1 = 130kb

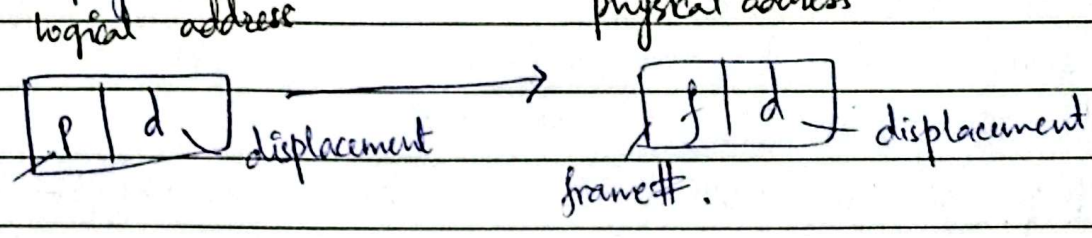


P1 :- Page 0 saved in frame 5
Page 1 saved in frame 2.
Page 2 saved in frame 1.

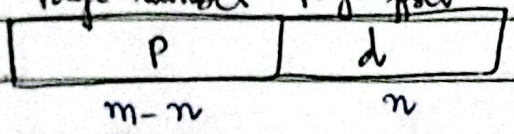
Page table :- Page (logical memory) → frames (physical memory)

Every process will have its own page table.

Agar ek process 8 pages mai divide hua wa toc it will have page table with 8 pages.



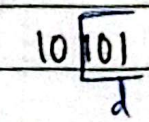
Page ya frame ka size will be in powers of 2 (2^m, 2^n).
Largest 2-digit no. = 11 → 3 in decimal
Largest 3-digit no. = 111 → 7 in decimal



That's why d is 10

largest value ki 2 bits hain toe utna hi size d ka hoga.

largest value can be represented by 3 bits.

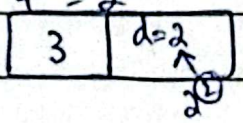


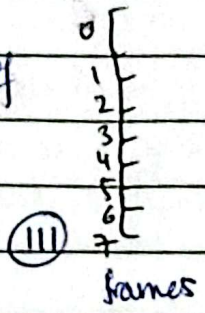
Now this will be d

Q) Agar 8 pages hain & 4 is the size of page

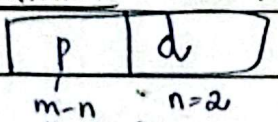
$4 = 2^2$

m = 5
n = 2



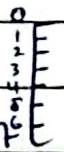


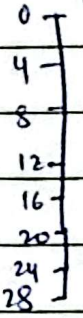
Q. EXAMPLE:- n = 2, m = 4



physical memory 32 / page size 4 = 8 frames

since page size is 4 bytes





Paging : No External fr may have some Internal Fragmentation.
Paging mai external fragmentation nhi hote.
Internal fragmentation hosakte hai.

Q:- If page size is 2,048 bytes fr, process of 72,766 bytes

$\frac{72,766}{2048} = 35.5$, # will need 35.53 pages, we will allocate 36 pages, so 0.5 page or $(1-0.53) \times 2048 =$ 0.47×2048 = 962 bytes will result in internal fragmentation

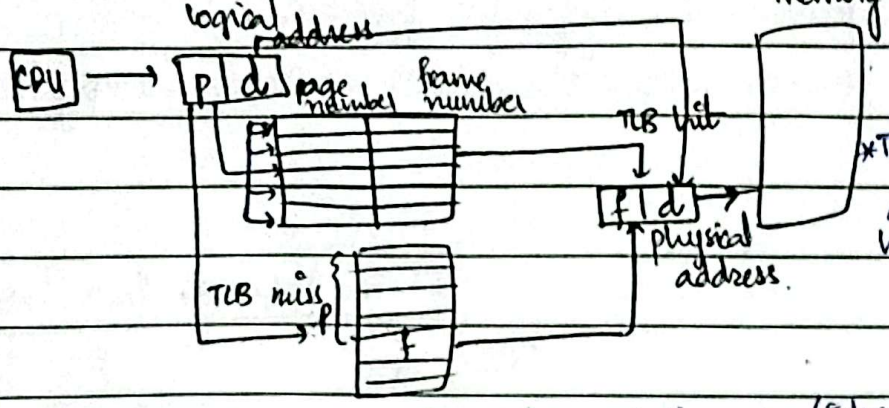
page size

Date: M T W T F S S

35.53 pages will be utilized, 36 pages we allocated.
36 - 35.53 = 0.47 page is extra that we allocated.
Page size is 2048 bytes
0.47 x 2048 bytes = 962 bytes will not be utilized that will result in internal fragmentation.
Suggestion :- if we reduce page size bytes, so less memory will be waste lekin page management mushkil hojaega since no. of pages will increas

9.3.2.1 TLB → diagram important.

2 data memory access is required. ek data page # k lie, dosse data frame # k lie



TLB → cache-like hardware component that stores recently accessed virtual-to-physical memory address translation in a processor.
*TLB mai page number aur uske against frame number save hotta hai.

hit ratio → lek say hit ratio 80% hai (iska matlab 80% hamein page # aur frame # TLB mai mil rahe

effective access time = 0.80 x 10 + 0.20 x 20 = 12 ns

TLB hit wale will require access only ek data

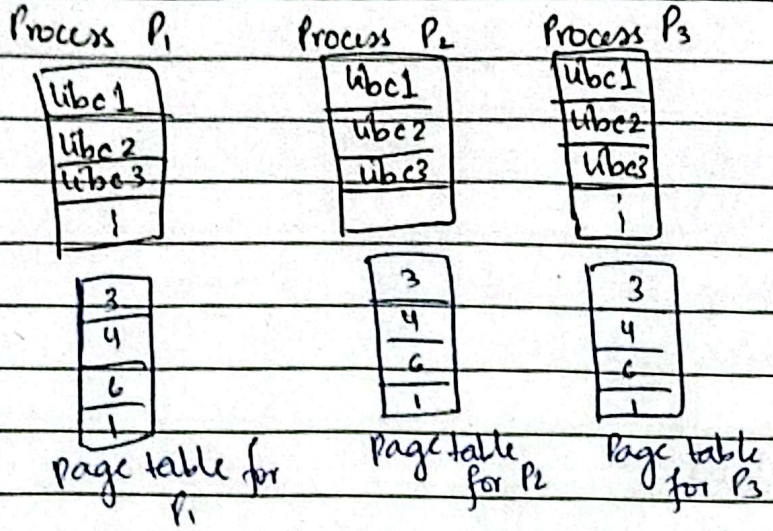
TLB miss waale will require memory access 2 data.

9.3.3 PROTECTION

valid-invalid bit → ye batate hai k ye page main memory mai load hua wa ya nhi.

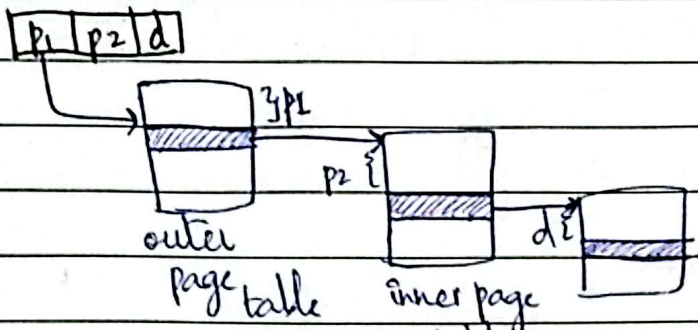
Date:
M T W T F S S

SHARED PAGES

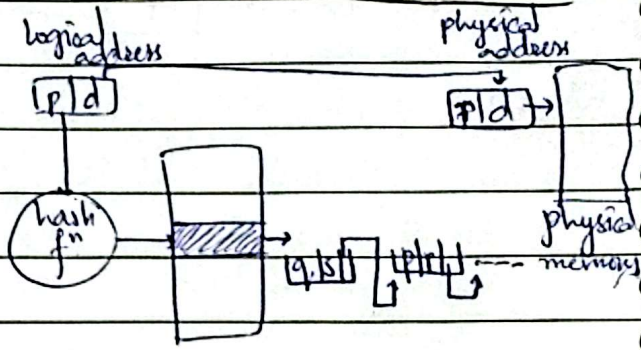


Teeno processes will share pages if all three are using pages for reading purpose only, lekin agar kisi ek process ko writing krne toe usko alag copy of pages dejaate (COW concept, copy pages on write)

HIERARCHICAL PAGING :-

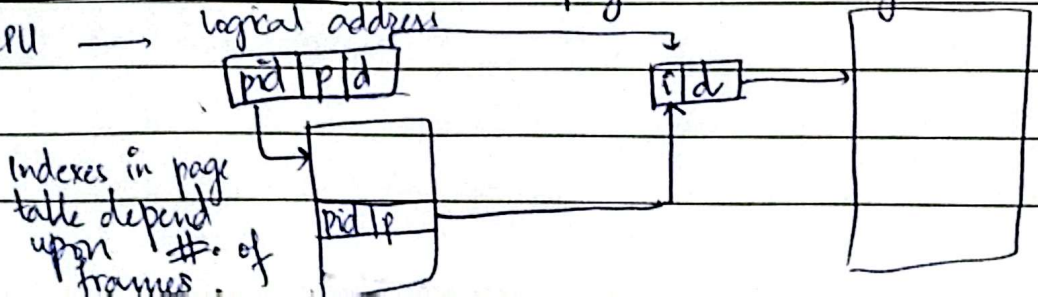


9.4.2 HASHED PAGE TABLE



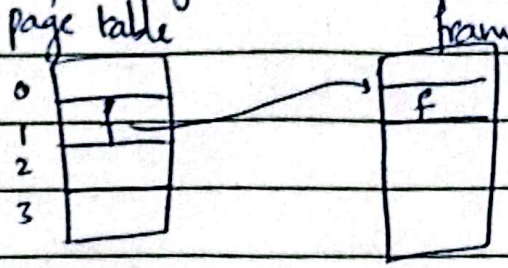
Collision k lie we use separate chaining

INVERTED PAGE TABLE → Ek hi page table hoga saare processes ka

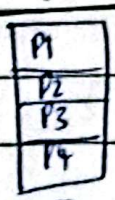


Date:
M T W T F S S

Normally, each process has its own page table.
Indexing of page table depends upon #. of pages.



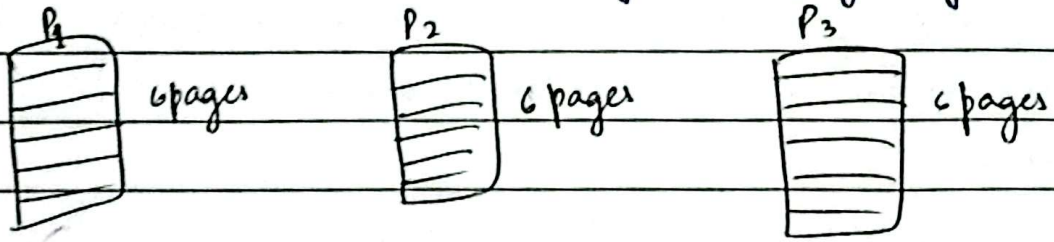
9.5 SWAPPING



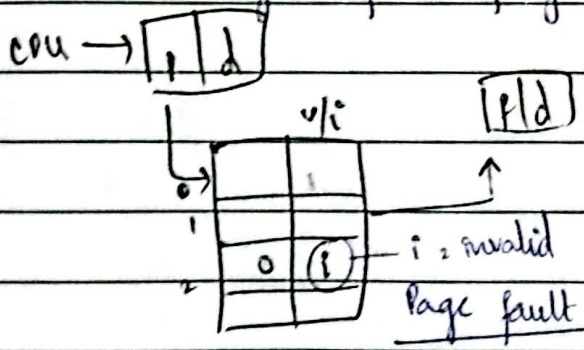
these processes already loaded into main memory.
Now, I want to run P5 but space nhi hai toe we use Swapping.
Any process is swapped out temporarily (its process state is saved).
& new process is swapped in.
Backing store ⇒ Secondary memory / Hard disk.
Sometimes, only pages within a process are also swapped in/out.

CHAPTER 10: VIRTUAL MEMORY

*allows a computer to use more memory than physically available.



Initially, it is efficient k mai teeno process k 2, 2 pages load krlon so that I can maintain degree of multiprogramming.



Page fault :- Agar kisi page to run krne ki request ae but wo already memory mai loaded nhi.
This is page fault.

Pure demand paging:-
Never bring a page to memory until it is required.

Date: M T W T F S S

DEMAND PAGING:- pages tab hi load hon jab demand ho.

Steps for handling page fault:-

1. Request as CPU se K page mangwana
2. Page table mai dekha, status invalid :- Page fault
   ↳ Trap to OS
3. page is on backing store.
4. that page is allocated free frame. → agar already free frame available nhi hai, toe OS kisi bhi frame ka page replace krwadega (swapping of page). that again depends on algorithm k wo page replace hoga jiska usage kam.
5. (Reset) Restart page table. change 'i' to 'v'.
6. Restart Instruction

Principle of locality:-
Agar N pages hain, toe ideal case mai N page faults aenge.

Total time to page fault = Swap out + Swap In

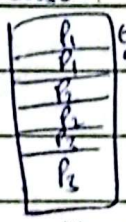
We want to reduce that time to handle page fault.

* code part mai modifications nhi hote... data part mai hote

* If page contained code, we directly do swap in step h overwrite it, since wo code already disk mai save hua wa hoga.... this saves us the time for swap out. Only swap-in required.

FRAME ALLOCATION :-
allocating frames to pages of multiple processes.

Lets say
6 pages ← Process P_1
6 pages ← Process P_2
6 pages ← Process P_3



← 2 frames allocated to the pages of each process.

Even if page fault occurs, toe bhi Process P_1 kisi apages apas mai replace honge since wo frames hamne P_1 ko pages ko allocate krdie.
This is the case where P_1, P_2 & P_3 all are of equal size.
In case of different sizes, frames allocation can be unequal to each process.

Date: M T W T F S S

EMPLOYMENT OF PAGE REPLACEMENT ALGORITHM / HANDLING PAGE FAULT.

(1) Page fault → (2) finds in a new frame → (3) if already available, directly page is swapped in

(4) if not available → (5) use page replacement algorithm to select victim frame

(6) again victim frame mai such modification was hua tbe it is saved to disk

(7) otherwise directly swapped in (rewritten). → (8) updation of page & frame table

Q. How To Modify K Wo Page Modified Hoa Thai Ya Nhi?

* there is modify bit (or dirty bit).

0/1.

0 if not modified, 1 if modified

PAGE REPLACEMENT ALGORITHMS:- to select victim frame.

FIFO.

4 pages A B C D

3 frames

Reference stream : A B C A B D A D B C A

	A	B	C	A	B	D	A	D	B	C	A
frame 1	A	A	A	A	A	D	D	D	D	C	C
frame 2		B	B	B	B	B	A	A	A	A	A
frame 3			C	C	C	C	C	C	B	B	B

#. of Page faults :- 07

replace A FIFO, pehle A load hua tha

replace with B, A k baad B load hua that at the start

MIN → future mai dekhte

	A	B	C	A	B	D	A	D	B	C	A
frame 1	A	A	A	A	A	A	A	A	A	A	A
frame 2		B	B	B	B	B	B	B	B	C	C
frame 3			C	C	C	D	D	D	D	D	D

#. of page faults :- 05

B ya D kisi ko bhi replace karskte

Date: M T W T F S S

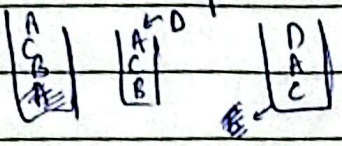
LRU:-

	A	B	C	A	B	D	A	D	B	C	A
frame 1	A	A	A	A	A	A	A	A	A	C	C
frame 2		B	B	B	B	B	B	B	B	B	B
frame 3			C	C	C	D	D	D	D	D	A

#. of Page faults :- 06

BELADY'S ANOMALY :-

Sometimes increasing # of frames increases the # of page faults.
↳ Belady's anomaly.

Q. Doubt mai how to implement LRU.
↳ Use stack  bottom of stack will be removed.
↳ Use clock, har process k sath timestamp save krega to determine least recently used.

2nd chance Algorithm

Q.) reference string :- 2 3 2 1 5 2 4 5 3 2 3 5

	2	3	2	1	5	2	4	5	3	2	3	5
frame 1	(0)*2	(0)2	(1)2	(0)2	(0)2	(1)2	(0)2	(0)2	3	3	(1)3	3
frame 2		*3	3	(0)3	5	5	5	(1)5	(0)5	(0)5	5	(1)5
frame 3				(0)*1	1	1	4	(0)4	4	2	2	2

all reference bits 0
so, decide on basis of FIFO among 3 & 1.
2 will get a second chance

Initially, all reference bits are 0. Reference bit 0 → when a page is accessed again. Then it remains 1 incase of no page fault. Then when reference bit 1 page fault occurs, page with reference bit 1 is given a second chance (it changes from 1 to 0). Others pages whose reference bits are already 0

Date:
M T W T F S S

are selected on the basis of FIFO.
Initially, when a page is accessed, reference bit is set to 0, then when that page is accessed again reference bit is set to 1. It remains 1 until & unless a page fault occurs. In case of a page fault, the page page with reference bit 1 is reset to 0. That page is given a second chance and other pages with reference bit 0 are selected on the basis of FIFO.
Basic algorithm of second-chance replacement is FIFO.
Implementation:- circular queue of frames ⟷ Page reference bits
Worst case → all reference bits are set (1). In that case all pages's reference bits will be reset to (0), and page to be replaced is selected on the basis of FIFO.

Enehanced Second chance Algorithm.
* considers reference bit & modify bit → as an ordered pair.
With these two bits, we have following four possible classes.
1. (0,0) neither recently used nor modified - Best page to replace
2. (0,1) not recently used but modified - not quite as good, because the page will need to be written out before replacement.
3. (1,0) recently used but clean - probably will be used again soon.
4. (1,1) recently used & modified - probably will be used again soon, and the page will need to be written out to the secondary storage before it can be replaced.
While replacing we see, to which class that page belongs to. We replace the first page encountered in the lowest nonempty class. We may have to scan the circular queue several times before we find a page to be replaced.

Date: M T W T F S S

OS goes around almost 3 times searching for (r=0, m=0) class.

1. Page with (0,0) ⇒ replace the page.
2. Page with (0,1) ⇒ initiate an I/O to write out the page, locks the page in memory until I/O completes, clears the modified bit, & continue the search.
3. For pages with reference bit set, the reference bit is cleared.
4. If the hand goes completely around once, there was no (0,0) page.
   - On second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0) ⇒ replace this page.
   - If the page is being written out, waits for the I/O to complete & then remove the page.
   - A (0,1) page is treated as on the first pass.
   - By the third pass, all the pages will be at (0,0).

Second chance Algorithm :-

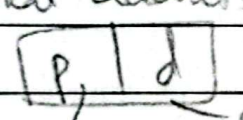
Page sequence – 0 4 1 4 2 4 3 4 2 4 0 4 1 4

Frame 1	0(0)	(0)0	(0)0	(0)0	(0)2	(0)2	(0)2	(0)2	(1)2	(1)2	(0)2	(0)2	(0)1	(0)1
Frame 2		(0)4	(0)4	(1)4	(0)4	(1)4	(0)4	(1)4	(1)4	(1)4	(0)4	(1)4	(0)4	(1)4
Frame 3			(0)1	(0)1	(0)1	(0)1	(0)3	(0)3	(0)3	(0)3	(0)0	(0)0	(0)0	(0)0
				2nd chance		2nd chance					2nd chance		2nd chance	

Min number of page faults = # of pages

Max number of page faults = length of reference string

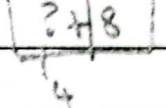
Logical address



No. of pages — offset / Page size

If virtual address space = 12 bits

Page size = 256 bytes = 2^8



$2^4 + 2^8 = 2^{12}$

No. of pages = 2^4 = 16 pages

Physical address