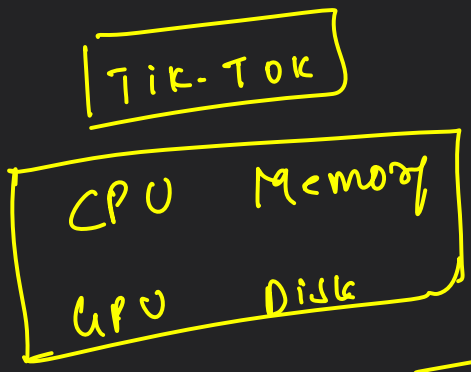
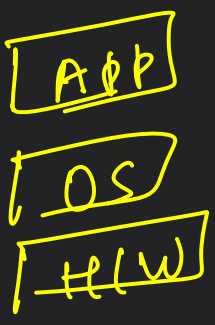
Operating Systems

P. Sailesh Kumar

Operating Systems

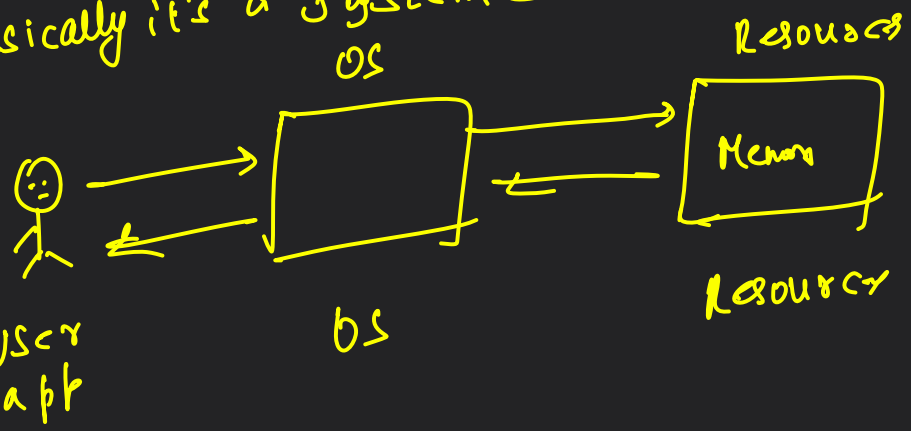


Tik-Tok App can directly access the resources (H/W) directly.



The functionality of OS is resource mgmt

* OS is an interface b/w App in & H/W (Basically it's a System Software)



what happens when OS isn't there?

App 1

4 write memory mgmt

4 write resource mgmt b/w

App 2

4 write memory mgmt

4 write resource mgmt

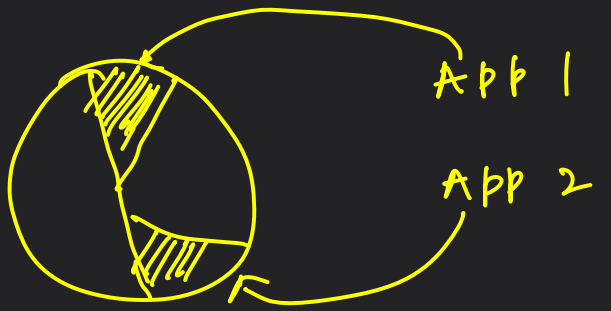
Every developer will write seperate code

So this app becomes more bulky.

DRY (Do not repeat yourself)

* To solve this all kinds of memory, resource mgmt is done by the OS.

Isolation & Protection:



With OS, we can have different memory segments so that there is isolation or else it can write into other app's by trespassing memory.

All this can lead to security loop holes.

* OS provides the abstraction, hiding the underlying H/W.
* Basically OS makes the work easy for appln developers

Types of OS

1) General purpose OS: Windows, Linux distributions, macOS, iOS, Android
2) Embedded OS: RTOS, QNX

Multi-programming OS
Allows multiple programs to run in 1 CPU
(Serial, pseudo parallelism)

Multi-processing OS
&
Multi-core
(more than 1 core)
(Real parallelism)

Multi-Threading

Process?!

↳ Basically a program under execution it called as process.

↳ Probably running an a.out is called a process

Thread!

↳ Lightweight process

↳ part of process sub-process also called as thread

Chrome app → Process

- Load webpage — Thread-1
- Rendering — Thread-2
- play audio-video — Thread-3

These threads share same resource & memory, but they run independently.

* Multi-threading is efficient in multi-core CPU's
* Multi-threading take time in single core CPU's

Now how does OS scheduler decide which thread gets the CPU?

It depends on Scheduling algorithms

Eg Process A → Threads A1, A2
Process B → Thread B1
Process C → Threads C1, C2, C3

Total 6 threads
4 - cores (CPU)

core 0 → A1
core 1 → B1
core 2 → C1
core 3 → C2

core 0 → A2
core 1 → C3
core 2 → A1 again
core 3 → B1 again

These are done by Scheduling algorithms

Multi-processing refers to running multiple process at a time

Eg Running Spotify, Same time browsing in chrome, and already in zoom call
(P1, P2, P3)

Components of OS

① User space → User appn's

② Kernel space → Has the access to underlying H/W.

User space: → NO H/W Access
↳ Environment for app's (like GUI/CLI → command line interface

mkdir test → Library call glibc-wrapper → Triggers the sys-call (sys-mkdir)
(UP) (UP) (UP)

→ SVC call (Kernel space) EL0 → EL1 → kernel receive the call decoded VFS (virtual filesystem) → checks path . & /

→ checks permission (write & executed) → call the specific FS-driver → FS-driver create & inode

→ marks inp data permissions, timestamp → write into memory → Now follows back the return path (EL1 to EL0)

0 ret val + success

if fail & already exist → −1

Functionalities of Kernel

1) Process mgmt (Process creation, termination, comm'n, Process & thread Scheduling)
   Process Sync,
2) Memory mgmt : (Allocate, Deallocate, memory track)
3) File mgmt: create, delete files
   → Directory mgmt
4) I/O mgmt: USB plug in, Game-Controller, Pen-drive
   All the I/O mgmt done through Kernel

Types of Kernel

Monolithic Kernel

entire OS runs on Kernel

(More Bulky)

Eg Server, cloud, IoT
Android

Micro-Kernel

only core fn like Memory & Process mgmt is there in kernel space

(Less bulky)

Eg Multiple context switching, overhead

IPC (Inter process comm'n)

Shared memory
Message is passed

i) Message passing through pipe

System-calls

User-space → System call interface

Kernel space

Hardware

Actual implementation of Sys-calls are implemented in 'C'-Language.

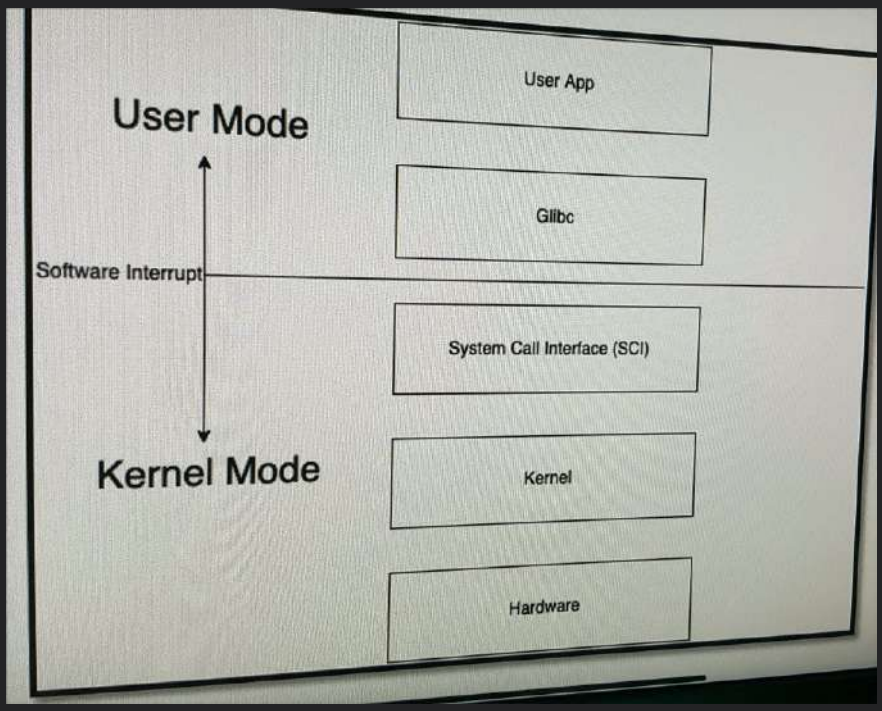
Eg ./a.out → Sys-call (SCE) → Kernel mode → Process mgmt

↳ create process → KM to UM

(Software interrupt / trap)

SVC in ARM

Sys-call in x86



	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle() SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	open () read () write () close () chmod() umask(chown()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()

How OS boots up

5 Major steps

① Power ON → Power supply (Motherboard & storage devices)

② CPU loads BIOS/UEFI (A small program/ - stored in BIOS chip (Non-volatile chip)

(BIOS Basic I/O

UEFI United extensible firmware

CPU init goes to BIOS

③ BIOS/UEFI runs the Code & init the HW

3.1 CMOS battery

3.2 POST → Power on Self Test

④ control move to bootloader

⑤ MBR → Master boot Record
(Disk 0th index)

⑥ EFI → Partition

⑦ Load the complete OS

32-bit Vs 64-bit

32 bit

Register Size

64 bit

Register Size

2^{32} Unique Address ⇒ 4 GB RAM

2^{64} Unique Address ⇒ Hexabyte RAM

Why can't we run 64 bit OS in 32 bit Systems

Registers are different (RAX, RBX, REX on x86-64
EAX, EBX, ESP

Memory addressing - different.

Different Storage in Computer

Register → Most closest to CPU
(Array kind of structure)
(Instructions are performed

cache memory → Additional memory
(for temporary
storage to be
executed)
↳ very fast memory
↳ copies frequently used data & instructions.
↳ Goal is to reduce average time to
access memory.

RAM → volatile
↳ used to store the running programs

ROM (HDD/SSD → long term storage

Virtual memory → part of secondary storage
(HDD/SSD) to extend
the RAM.

ROM → Boot ROM (Inside SoC)
Firmware

RAM → can be external DRAM

Storage → eMMC (NAND/NOR) SSD
↓
OS, Kernel, App's reside

Process Management

I have an executable (a.out)

Process creation

① Load the program & static data to memory (RAM)
↓
used for initialisation

static data
char *p = "Sailesh"; init data
int a = 0; Init data

② Allocate run-time stack.
(Local variables, f'n args, return values)

③ Allocate Heap
↳ Dynamic allocation

④ I/O tasks ↳ i/p, o/p, error
↓ ↓ ↓
Handle Handle Handle

⑤ OS hand off control to main

Sailesh binary

./Sailesh
↓
calls fork()
↓
copies & creates a child process

Parent (Actual char)

Child (copy)

In child
shell calls execv

↓

kernel reads the ELF

↓

set up text, code, data

↓

kernel sets up the PC to the start

↓

control goes back to user space

↓

Execute code

↓

return 0 (terminate)

---

fork

Parent (Actual shell)

child (Duplicate of shell)

① Parent waits for child to finish using wait system call.

* ② The child is not supposed to be remaining in the shell, it's job is to replace itself with the program we typed

③ Child calls execv, replaces the child memory with our program

④ Kernel looks at the ELF header of the binary to understand (_start)

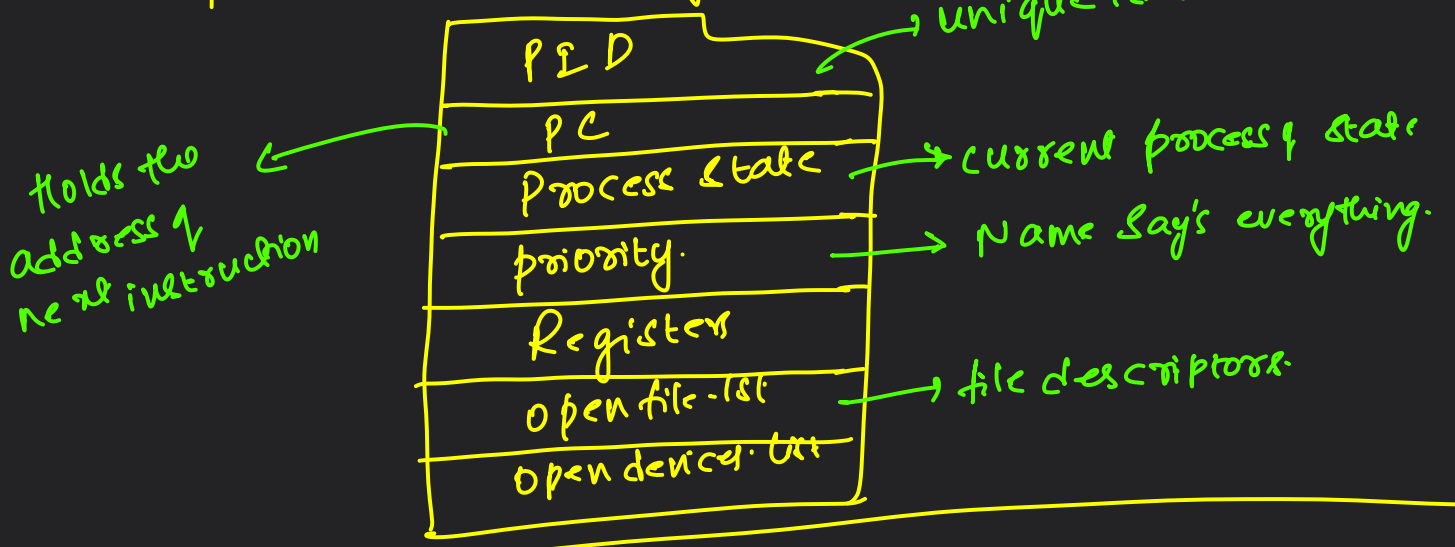
⑤ Set up code(text), data, bss sections.

⑤ PC points to _start

⑥ After start, jumps to main

Process control block has all the process attributes.

it is a "struct" type of data structure.



Process-states

Life-cycle of process

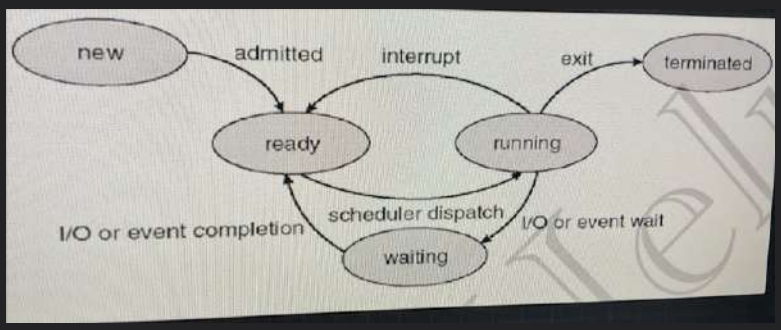
1) New state: Program converting to process

2) Ready state: Process in the memory (Ready Queue)

3) Running state: $P_1 \rightarrow$ CPU is allocated

4) Waiting state: Waiting for I/O completion

5) Terminated state: Process is finished



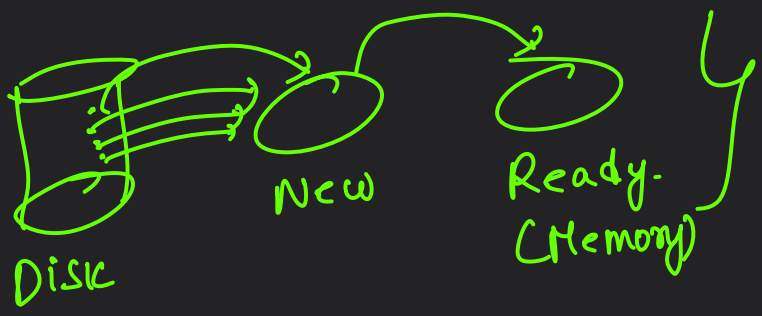
* Running to waiting
  ⇓
  I/O

* Running to ready
  Time slicing
  Again sent back to memory / Ready Queue

New state process are in Job queue

P1	P2	P3

So if any process goes to ready state i.e flow through job scheduler

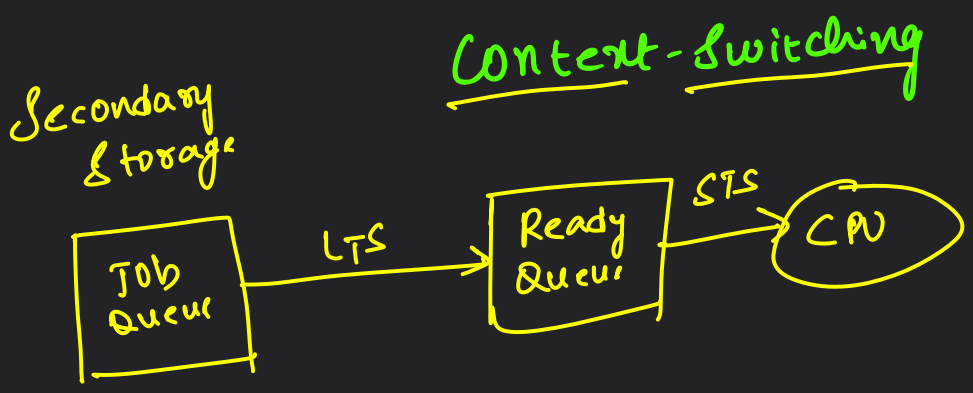


Long term scheduler
Job Scheduler

* Now the process is in memory in order to be executed (Ready Queue)
  it should be in Running state (CPU Execution)

Moving from Ready Queue to Running is done by CPU Scheduler.

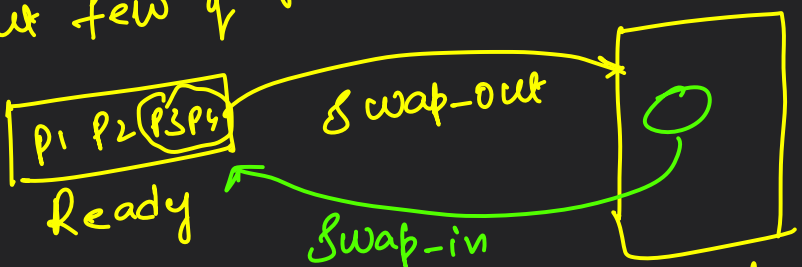
Context-Switching



LTS → Mix of Processes (I/O intensive, CPU intensive)

Medium Term Scheduler (MTS)

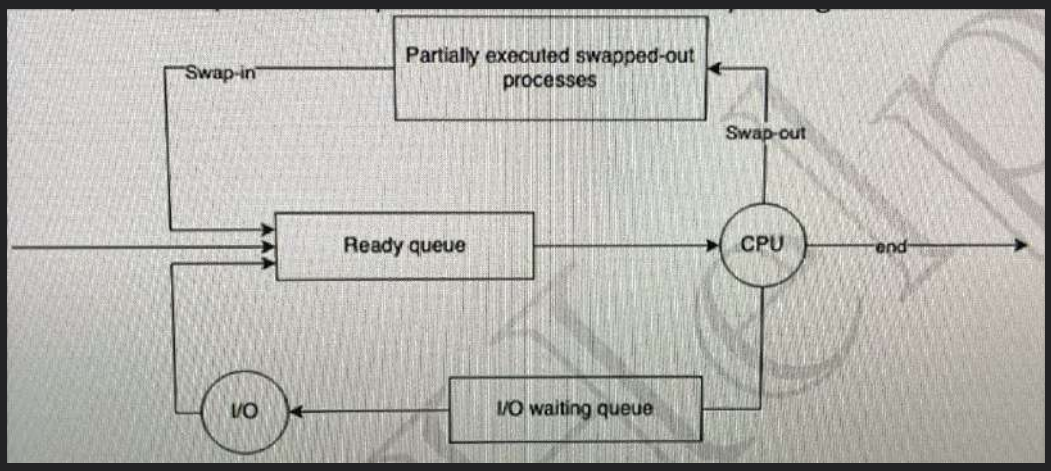
* Degree of multi-programming refers to No. of processes in ready Queue
* Now a condition arises that there are few processes in memory which are taking lot of memory, we (Ready Queue) can't store all of them right? So we need to swap out few of them.



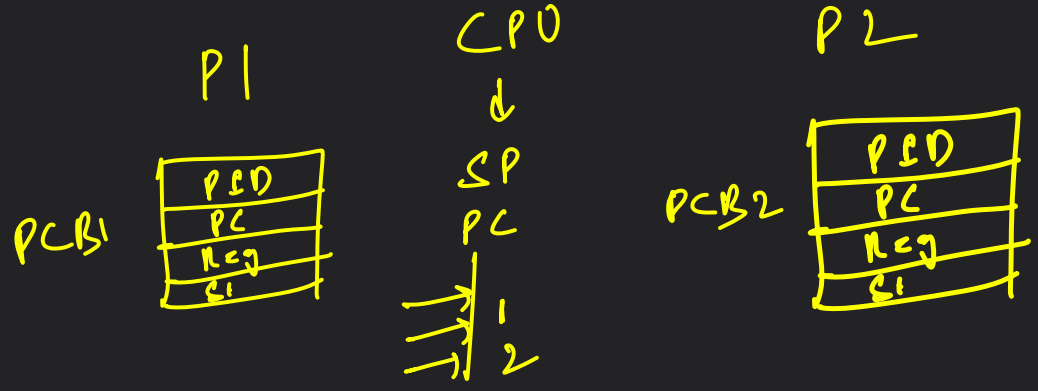
* We have understood swap-out, Let's see swap-in

Assume either P1 & P2 is terminated, Now ready Queue has memory at that time P3/P4 come back to ready Queue (swap-in)

The entire swap-out & swap-in called as Swapping.



Context-Switching



In Detail Analysis of Context Switching

* A is Running in CPU
* B is Ready (Probably created through fork() + exec())
  ↓
  Since Ready & created it has it's own PCB.
* Now 'A' is doing some read() operation from disk (Sys_call)
* Sys_call is triggered → CPU goes to Kernel mode.

* Kernel understands A will take time, so it can't make the CPU idle.
* So Kernel Scheduler will schedule "B".
* Before this it has to store 'A' CPU state
  - PC
  - SP
  - GPR's
  - CPSR / PSTATE
* Now "A" is marked as block.
* Scheduler → picks into Ready Queue → finds B → B is given to CPU
  - Kernel doesn't have any previous context of B
  - So it loads it's initial / primary context
* Now Kernel hit's eret
  - CPU ↳ PC = start of B
  - Switch from Kernel to User mode
* Now B starts running on CPU

Disadvantages → Overhead (Time-waste) during context switching.

Orphan process:

P1 Running task
| start
| P2 Running
| child

↳ P1 has suddenly hit an interrupt & it has terminated abruptly.

* Now P2's process has been linked to init.

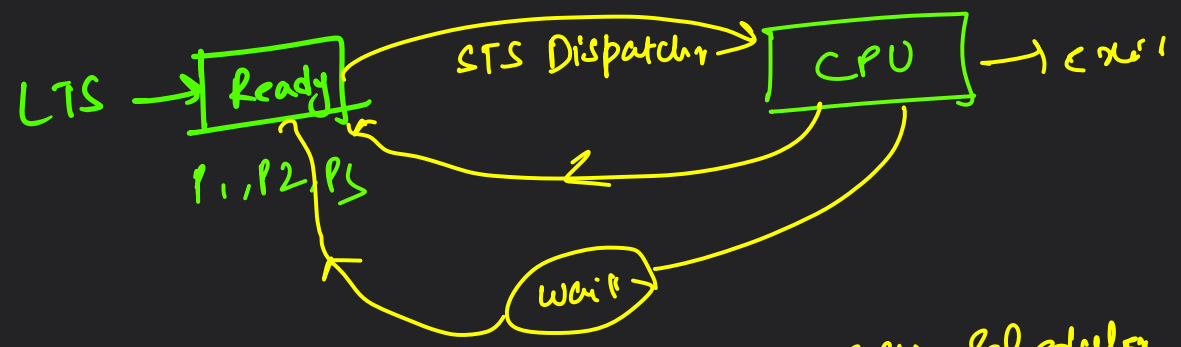
Orphan process is a process whose parent process has been terminated but child process is still running.

```
main()
{
    pid_t pid = fork();
    if (pid > 0)
    {
        printf("parent exiting");
        exit(0);
    }
    else if (pid == 0)
    {
        sleep(10); // give time for parent to exit
        printf("I am child, my parent is terminated, my
                new ppid is %d", getpid());
    }
```

Zombie-process

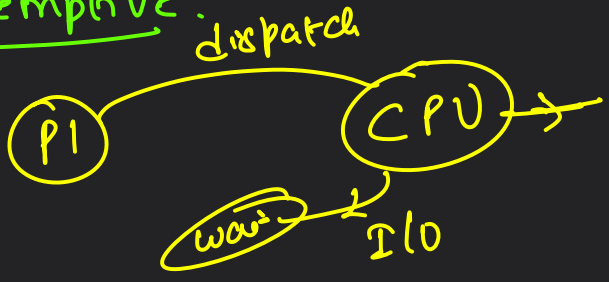
Suppose parent is waiting for 5 min but child is terminated in 2 min only

Process Scheduling



which process to pick is done by CPU Scheduler

1) Non-Preemptive:



Time slicing isn't there

2) Preemptive Scheduling : Time Quanting is followed

Starvation: CPU is not getting the process

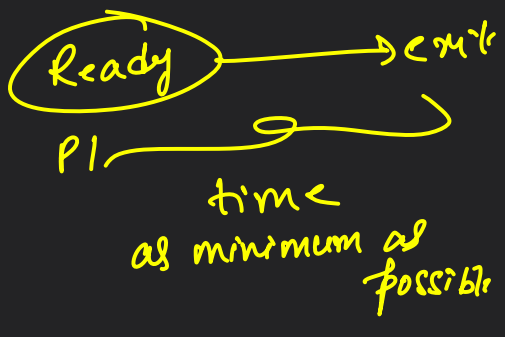
In non-preemptive starvation is more.

In preemptive : starvation is minimum

CPU Utilization is high, overhead is also more.

* Goals of CPU Scheduling

1) Max CPU Utilization
2) Minimum turn around time (TAT)
3) Minimum wait time (WT)

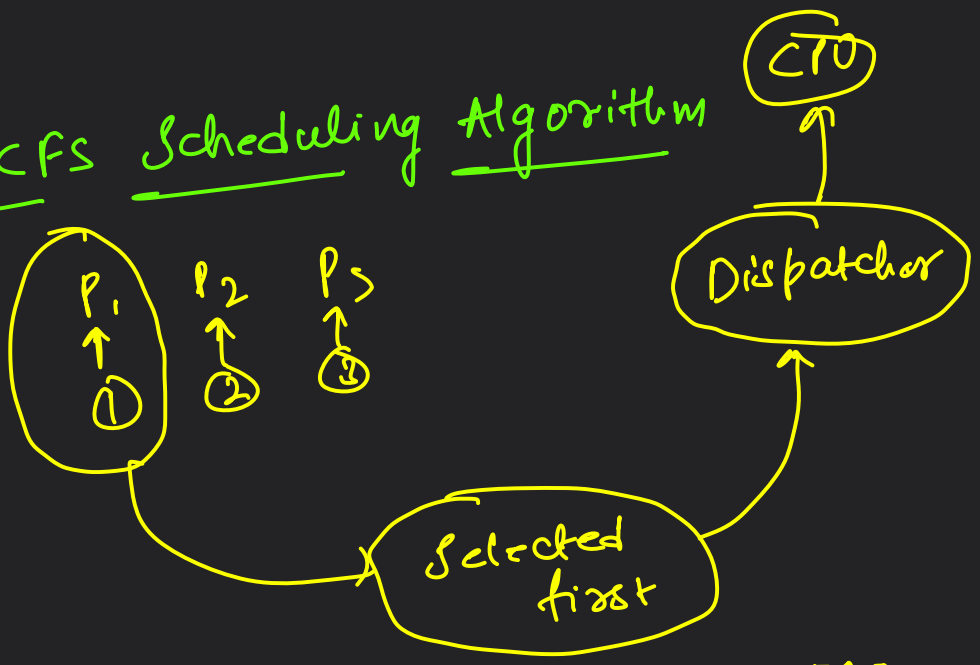


4) Minimum response time (Time b/w Ready Queue & CPU getting Assigned)

5) Max Throughput: No of process completed per Unit time

6) Burst-time: Time required for execution.

① FCFS Scheduling Algorithm



Pno	AT	BT	CT	TAT	WT
1	0	20			
2	1	2			
3	2	2			

FCFS → Non preemptive

first Arrived goes to Queue CPU taken it.

Like a Queue in Ticket Counter

SJF (Shortest job first)

Process with shortest burst time executed first

Burst time should be known.

Starvation for long jobs.

Round Robin → Fixed time Quantum
Issue overhead

Priority Scheduling:
CPU Allocated to the process to higher priority.

Scheduling in Detail will be covered later

Concurrency

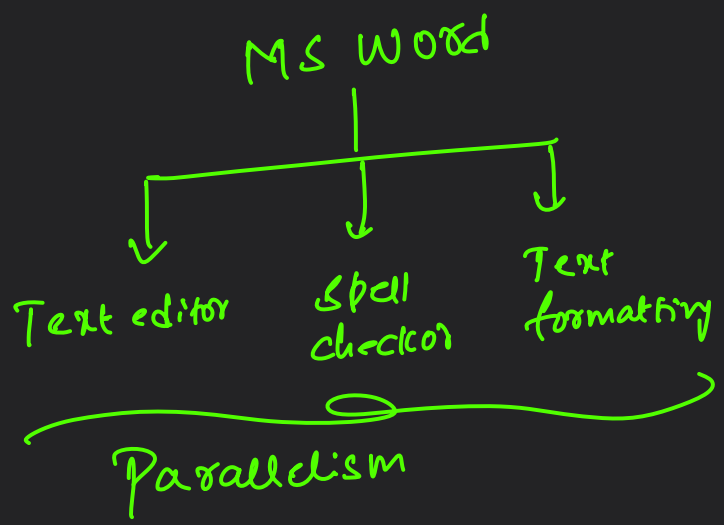
Ability to execute multiple instructions at a time

Process → T1, T2

Divide into threads (Sub-processes)

Independent Execution

Why are we dividing into threads?
So that we can divide the task & try to achieve parallelism.



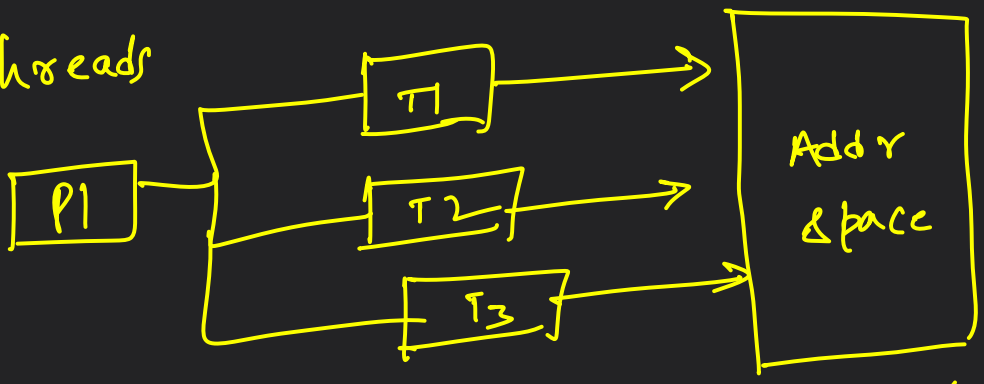
If not divided it would Execute Sequentially.

Memory mapping in Process & threads

Mapping is done through several steps

1) compiler / linker → Decides layout of code, data, BSS
2) Loader → Loads code & data into memory when process starts
3) OS Memory Manager → Provides virtual memory pages, manages stack/heap dynamically.
4) MMU → Maps virtual addresses to physical addresses

Threads

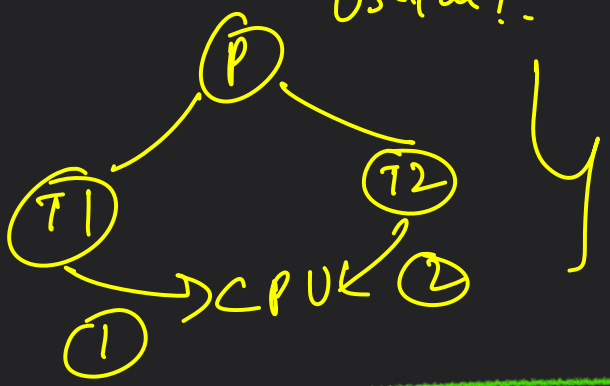


using same address "shared memory"

Just like processes, threads have their own Data structure "Thread Control block" TCB

In thread's context switching only the executing state (Register + SP) Address space remains same

Single CPU → Multithreading useful?!

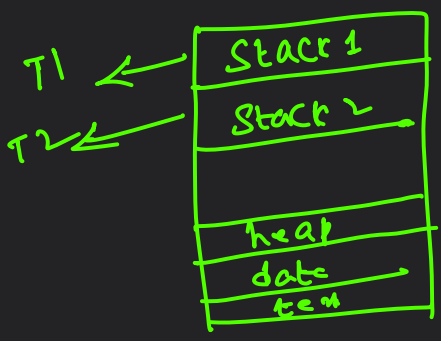


So Basically useless in case of single core CPU's

Multi-threading is benefit only for multi-core CPU's

Process

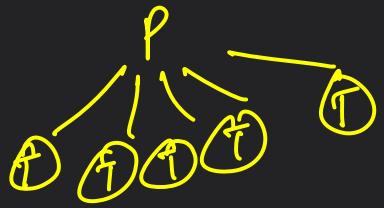
Thread



No of stacks depend on the no. of threads

Advantages of Multi-threading

1. Responsiveness & Interactive.
2. Resource sharing → address space is shared
   (No comm'n gap & minimal overhead
3. Thread Economy (context switching has less overhead)
4. Threads better utilization Multi-core CPU



Give it to multi-core CPU

C → Does not contain built in support for multithreading

It entirely relies on the OS

POSIX threads, Pthreads APIs are available

Linux, MacOS Systems.

// Library is #include <pthread.h>

```
main()
{
    pthread_t tid;
    Pf("Inside main before creation");
    pthread_create(&t_id, NULL, task_master, NULL);
    pthread_join(t_id, NULL);
    Pf("In main after join");
}
```

```
void * task_master(void * args)
{
    Pf("Inside task master");
    return NULL;
}
```

* Thread begins task_master
* main doesn't know whether completed & not
* Because task_master is not called by main, OS Scheduler starts new thread that eventually calls the task_master independently.
* Main & task_master are running independently. (Not the expected caller-calle function)

* So main moves to next line naturally.

* Here we see pthread_join (This lets main explicitly wait for the threads)

Synchronisation, OS clean up

Everything is done.

Critical Section

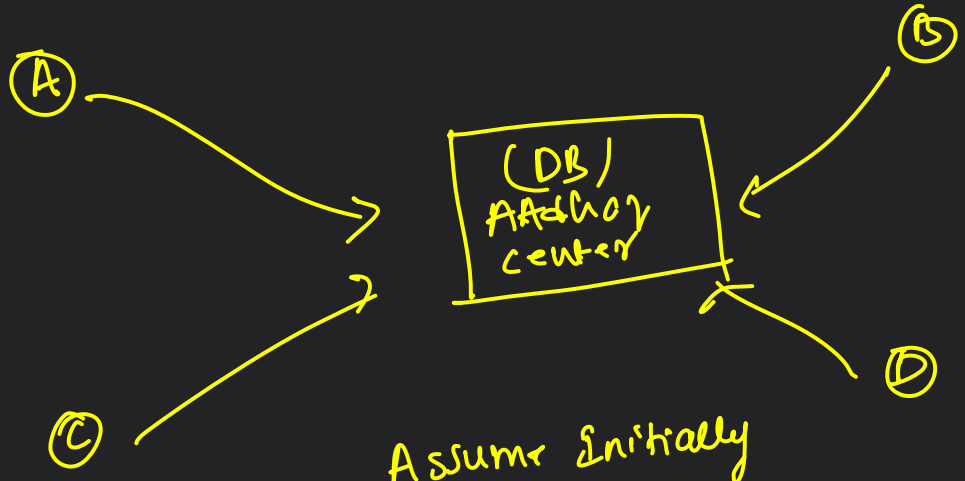
Eg `count++;` ⇒ `count = count + 1;`

How this works in CPU-level?

internally

```
tmp = count + 1;
count = tmp;
```

} Internally works like this



from A we got update request

`tmp = count + 1;`

`count = 11`

Assume Initially

count = 11

So we are updating tmp to 11 instead of count.

Now B is also updating request

So instead of count, the tmp will be

`tmp = count + 1`

`tmp = 12`

A → tmp 12

B → tmp 12

Even after 2 requests our final count is 12, instead of 13

This is race-condition

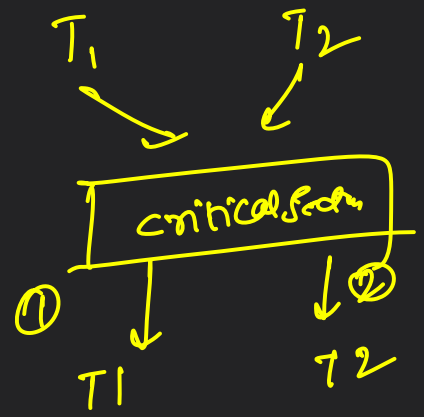
Here DB is critical section
(Multiple threads working on same section/shared memory)

Example code

```
int counter=0; // Global var.
void * incr (void *arg)
{
    for (int i = 0; i<500; i++)
        counter++;
}
pthread_t t1, t2;
pthread_create(&t1, NULL, incr, NULL);
pthread_create(&t2, NULL, incr, NULL);
// wait join
printf("%d", count);
```

T1	T2
tmp = count; 0	tmp = count; 0
tmp++; 1	tmp++; 1
count = tmp;	count = tmp;
(1)	(1)

Now we need to overcome this problem, this can be done by **mutual exclusion (mutex)**



T1 → T2
or
T2 → T1

Either
T1
or
T2

Locks (Mutex)

Let's understand working behind this

Init → Acquire → Hold → release → destroy.

1) Init:

pthread_mutex_init(&lock, NULL);

Create mutex object & set the internal state to be unlocked

Main creates → 2 threads

T1 T2

1) currently Lock is free
2) T1 calls pthread_mutex(&lock)
3) T1 gets lock
4) T1 does ++(counter part)
5) T1 calls unlock
   (Lock is free again)
   500
6) T2 is in waiting state, while T1 has the lock
7) once T1 is released T2 gets the lock
8) T2 does counter++
9) T2 releases lock
   500

Total exact 500+500 = 1000 times

Semaphores: A variable which can be used as a shared variable to prevent race condition.

* Implemented through wait() & Signal()
  ⇓
  Both are atomic in nature

wait()
↳ Gives the status of process to enter into critical Section

* If Semaphore value is >=1 then it can avail critical Section
* Semaphore value will be decremented.
* If Semaphore val is '0', then indicates some process is in critical Section. No other process is allowed

Signal() : It is an Increment operation
S = S+1

Semaphores
- Binary Semaphor (0, 1)
- Counting Semaphores (0, 1, ... n)

binary_semaphore

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

/*
 Simple program: two threads share one printer.
 Comments are written step-by-step (from main -> threads -> exit)
 and explain why each call is made and what the library does internally.
*/

/* Global semaphore representing the printer.
   We treat it as a binary semaphore by initializing to 1 (free).
   sem_t holds: an integer counter + kernel-managed wait-queue (handled by sem_* calls).
*/
sem_t printer;

/* Thread function: represents a user submitting a print job.
   Steps inside this function are numbered and explained.
*/
void* print_job(void* arg)
{
    char* name = (char*) arg;

    /* STEP A: Try to acquire the printer.
       sem_wait is an atomic library/kernel call:
         - If semaphore value > 0: sem_wait atomically decrements it (value--) and returns immediately.
         - If semaphore value == 0: sem_wait blocks this thread and the kernel puts it on the semaphore's wait-queue.
       Important: you do NOT write "printer--" yourself — the decrement and blocking logic is done inside sem_wait.
       Reason: manual decrement would cause race conditions; sem_wait is implemented to be atomic and safe.
    */
    sem_wait(&printer);

    /* STEP B: Now the thread has exclusive access to the printer (critical section).
       Only one thread can reach this point at a time because semaphore value was decremented to 0.
       We simulate the printing work with sleep(2).
    */
    printf("%s is using the printer...\n", name);
    sleep(2); // simulate printing time (critical section)
    printf("%s finished printing.\n", name);

    /* STEP C: Release the printer.
       sem_post is also an atomic call:
         - It increments the semaphore value (value++).
         - If there are threads blocked on this semaphore, the kernel wakes up (unblocks) one of them.
       Reason: sem_post signals "resource available" and transfers control to one waiting thread if present.
       Note: because semaphores don't enforce ownership, any thread can call sem_post (unlike a mutex).
    */
    sem_post(&printer);

    return NULL;
}

int main()
{
    pthread_t t1, t2;

    /* STEP 1: Initialize semaphore
       sem_init(&printer, pshared, value)
         - &printer : address of semaphore object
         - pshared = 0 : semaphore is shared between threads of the same process (not between processes)
         - value = 1 : initial count (1 means printer is free). Initializing to 1 makes this behave like a binary semaphore.
       Reason: we want exactly one user to use the printer at a time, so start with 1 available "slot".
    */
    sem_init(&printer, 0, 1);

    /* STEP 2: Create thread t1 (User1)
       pthread_create tells the OS to start a new thread that runs print_job("User1").
       Scheduling is up to the OS: thread may run immediately or later.
       Reason: we spawn two concurrent actors to demonstrate contention on the printer.
    */
    pthread_create(&t1, NULL, print_job, "User1");

    /* STEP 3: Create thread t2 (User2)
       Same as above. Now we have two threads that might race to acquire the semaphore.
       Reason: to show that only one thread can proceed while the other blocks inside sem_wait.
    */
    pthread_create(&t2, NULL, print_job, "User2");

    /* STEP 4: What happens now (runtime behavior — not explicit code)
       - Either t1 or t2 reaches sem_wait first (depends on scheduler).
       - First arriver sees value == 1 -> sem_wait decrements to 0 -> it proceeds into the critical section.
       - The other arrives and finds value == 0 -> sem_wait blocks that thread and it sleeps until sem_post.
       Reason: this ensures mutual exclusion on the printer resource.
    */

    /* STEP 5: Wait for threads to finish
       pthread_join(t1, NULL) blocks the main thread until t1 finishes.
       pthread_join(t2, NULL) blocks until t2 finishes.
       Reason: if main returned earlier, program might exit while threads are still running (bad).
       Also we must ensure semaphore is not destroyed while a thread could still use it.
    */
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    /* STEP 6: Cleanup
       sem_destroy releases any resources associated with the semaphore object.
       Reason: good practice to destroy what you initialize. Only do this after all threads are done.
    */
    sem_destroy(&printer);

    return 0;
}
```

Binary_Semaphore

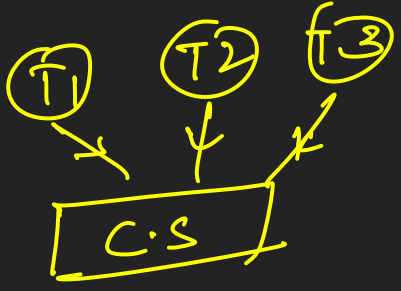
Example Snippet

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

/*
 Example: 6 students share 3 library tables.
 Counting semaphore initialized with 3 (for 3 tables).
 We will trace how the semaphore value changes with each sem_wait / sem_post.

 Compile: gcc -pthread semaphore_tables.c -o semaphore_tables
 Run:     ./semaphore_tables
*/

#define NUM_TABLES 3
#define NUM_STUDENTS 6

sem_t tables;  // counting semaphore

void* student(void* arg)
{
    int id = *(int*)arg;

    /* STEP A: Try to acquire a table
       sem_wait:
         - If tables > 0 → decrement tables-- and continue
         - If tables == 0 → block until someone calls sem_post
       Example value changes:
         Start = 3
         First student arrives → sem_wait → 3 → 2
         Second student → sem_wait → 2 → 1
         Third student → sem_wait → 1 → 0
         Fourth student → finds 0 → BLOCKED
    */
    sem_wait(&tables);
    printf("Student %d got a table (semaphore decremented).\n", id);

    sleep(2); // simulate study time

    /* STEP B: Release the table
       sem_post:
         - Increment tables++
         - If any students are waiting, wake one of them
       Example value changes:
         When first student leaves → 0 → 1
         Wakes up Student 4 → immediately sem_wait inside that student → 1 → 0
    */
    printf("Student %d left the table (semaphore incremented).\n", id);
    sem_post(&tables);

    return NULL;
}

int main() {
    pthread_t tid[NUM_STUDENTS];
    int ids[NUM_STUDENTS];

    /* STEP 1: Initialize semaphore
       sem_init(&tables, 0, 3)
       Current value = 3 (3 tables free)
    */
    sem_init(&tables, 0, NUM_TABLES);

    /* STEP 2: Create 6 student threads
       Each will try sem_wait:
         - First 3 succeed immediately (3→2→1→0)
         - Remaining students block until tables free
    */
    for (int i = 0; i < NUM_STUDENTS; i++)
    {
        ids[i] = i + 1;
        pthread_create(&tid[i], NULL, student, &ids[i]);
    }

    /* STEP 3: Join all threads
       Main thread waits until all students finish.
       Semaphore value will return to 3 at the end:
         - Every sem_wait (--) is matched by a sem_post (++).
    */
    for (int i = 0; i < NUM_STUDENTS; i++)
    {
        pthread_join(tid[i], NULL);
    }

    /* STEP 4: Destroy semaphore (cleanup) */
    sem_destroy(&tables);

    return 0;
}
```

Counting Semaphore Example

Disadvantages of Locks

① Contention



T1 is under execution

① Assuming T1 has encountered an exception & it got terminated / dead, T2 & T3 will be in infinite wait state.

② Deadlock:

T1 has R1, but needs R2

T2 has R2, but needs R1

Each thread is waiting for other thread to unlock/release

Conclusion: To manage critical section problem, we are introducing locking mechanism.

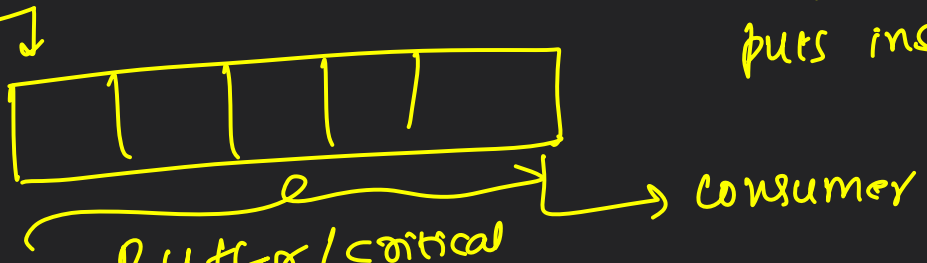
If locking mechanisms are not implemented correctly it will lead to deadlock conditions.

Producer - Consumer problem

① Thread 1 → Producer

② Thread 2 → consumer

Producer, produces data & puts inside buffer



Producer, produces data & fills in the empty slot

consumer, consumes data from the available slots

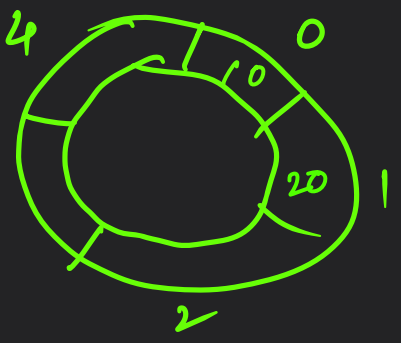
* Now i need to Sync b/w P & C
* If n slots are full, we can't tell P to fill the slots (Producer must not insert data when buffer is full)
* Consumer must not pick (remove data when buffer is empty.

Let Solve this through Semaphore

① mutex m → Binary Semaphore, used to acquire lock on buffer

② counting Semaphore → Initial val is 'n' (empty)

③ Full → Tracks filled slot
initial = 0



count = 5

circular buffer ⟹

Producer process → In
Consumer process → Out

Code for producer problem

```
while(true)
{
while(count == Buffer_size);
buffer[in] = next_produced;
in = (in+1) % Buffer_size;
count++;
}
```

Iterations

count = 0, in = 0

0 == 5 ✗

buffer[0] = 10;

Suppose in = 4, next it should point to 0
5 % 5 = 0 ✓

count = 1

buffer[1] = 20;
in = 2
Now count = 2

Last count is 5

5 == 5 ✓

(Infinite while loop)
Producer full / buffer full

Code for Consumer process

```
while (true)
{
    while (count == 0); // Buffer empty
    next_consumed = buffer[out];
    out = (out + 1) % Buffer_size;
    count--;
    // consume the item to be produced
}
```

5 == 0 ×
out = 0;
next consumed = 10;
incr out
c--;

4 == 0 ×
out = 1
next cons = 20;
incr out; // 2
c--;

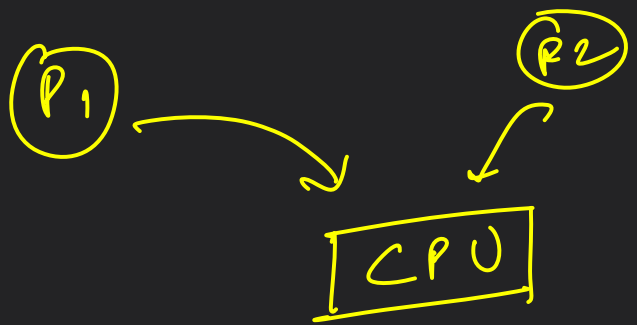
Last iteration
c = 0
0 == 0 ✓
(infinite while)

① Producer problem buffer full
(Producer must wait)

② Consumer problem buffer empty
(consumer must wait)

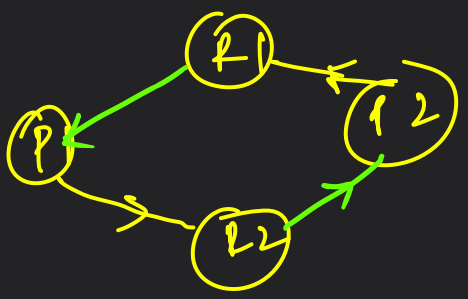
③ Critical Section (If producer & consumer try to access the shared buffer at same time)

Deadlocks



→ finite no. of resources are there

→ Multiple process/threads



P1 → R1 & R2

P2 → R1 & R2

Both are required

For acquiring resource, we use locking mechanism

P1 → R1 lock — Now P1 is waiting for R2

P2 → R2 lock — Now P2 is waiting for R1

* This waiting mechanism can lead to infinite wait time.
* Now during this situation, the system is in deadlock.

How a process/thread utilize a resource?

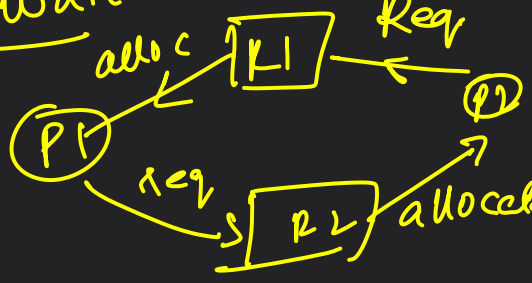
① Request ② Use ③ Release.

Necessary condition for deadlock

① Mutual exclusion

1 Resource → 1 Process / 1 Thread

② Hold & wait

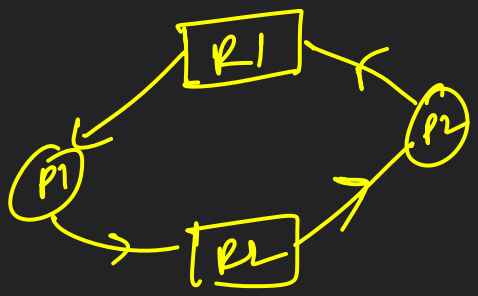


P1 is holding R1 also
P1 is waiting for R

③ No preemption:

P1 → Execution complete

④ circular wait



Resource allocation graph:

① Vertex → ① Process Vertex
② Resource vertex

② Edges → ① Assign Edge 

② Request

Methods to Handle Deadlock:

① Prevent mutual exclusion

Eg Replicate / Duplicate the resource

Multiple UART ports, tasks can use different ones.

② Make the resource read only

Eg If no state change happens multiple processes can read at the same time

(Files opened in read only mode)

Dead Lock Avoidance

① current state of System

↳ No. of processes

↳ Need of Resource of each process

↳ currently allocated amount of R

↳ Max amount of each Resource

OS Scheduler makes decisions dynamically at run-time ensure that System never reaches an unsafe state

Unsafe state ↳ These states refers to no. of resources required to complete.

Total Resource = 10
current available = 3

P1: Max need = 7
current alloc = 4
Need 3 to complete

P2: Max need = 4
current alloc = 2
Need 2 more to complete

Safe-State:

Give P2, 2 resources (current available 3 - 2 = 1)

P2 finish and release

So total available (4 + 1 = 5)

Give P1 → 3 [So total available 5 - 3 = 2]

and P1 = 4 + 3 = 7

P1 completed

Case 2: Unsafe

P1 requesting 3, and we give it directly 3 to it

But P2 is requesting 2 to complete

Now either P1 & P2 will finish

Deadlock

Deadlock detection:

Assuming our System is not having the capability of doing prevention & avoidance, so during these case we manually implement a deadlock detection mechanism (OS use resource allocation graph to check the cycles)

Deadlock Recovery

① Process Termination:

In System we are having 10 process 3 are suffering with deadlock

kill -9 3 processes

CPU time wastage, Memory wastage I/O device depended on this also wastage.

② Abort 1 process

check deadlock detection algorithm

Repeat

overhead of detection algorithm

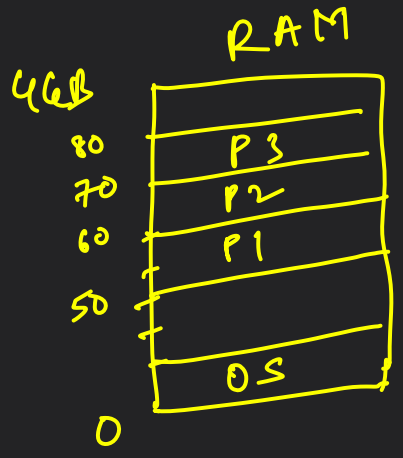
③ Resource-preemption : Forcefully release the resource

a) Selecting a resource

b) Rollback (execute from scratch)

c) Starvation (Resource starvation)

Memory Management

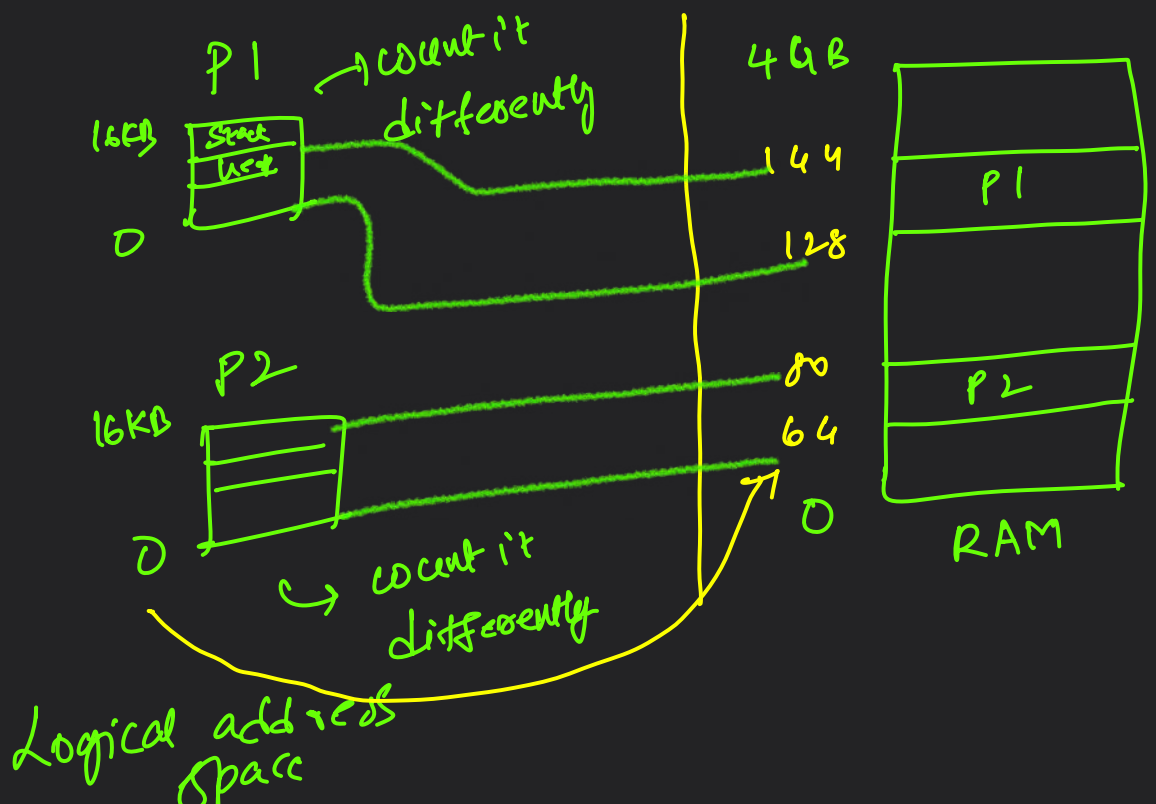


P1 Address space
50 - 60

Suppose P1 is trying to do
P1 address + 5
60 + 5 = 65

So P1 is trying to access P2

(This happens when OS is not having protection & memory isolation)

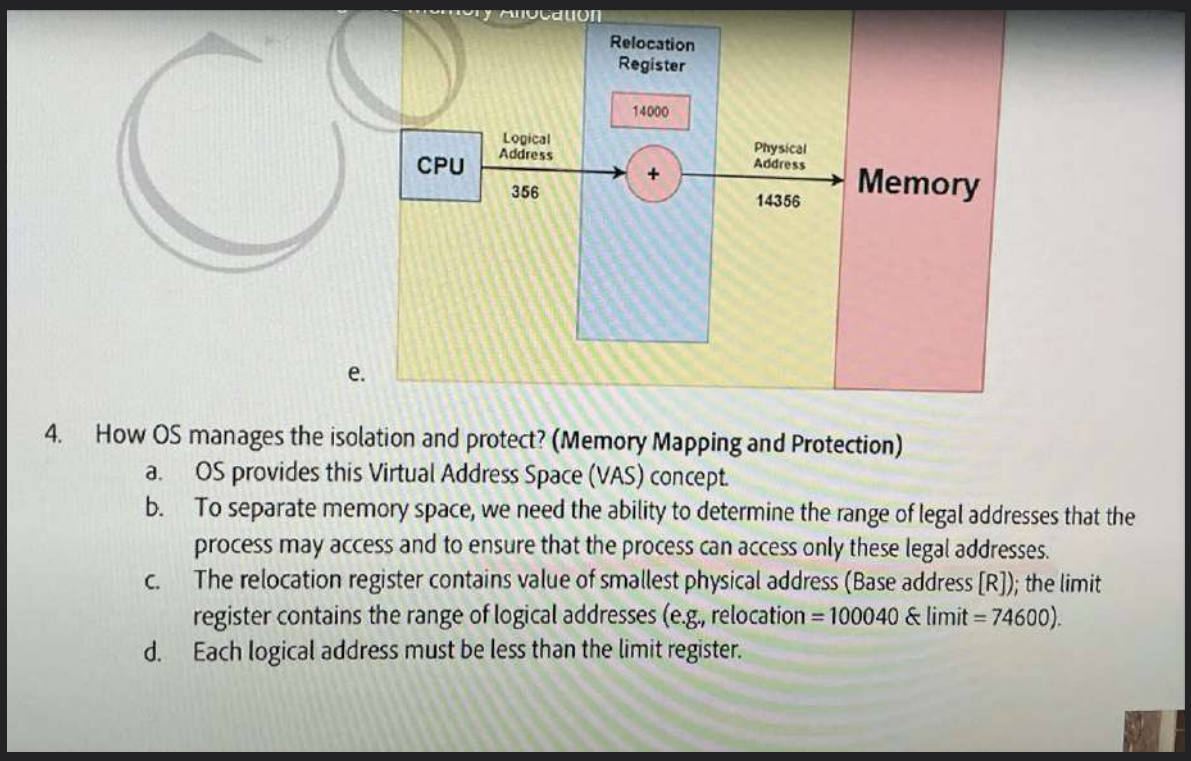


OS → Base & offset (P1)
↓ 128 ↓ 16

OS does the mapping of logical to physical address
(Address translation)

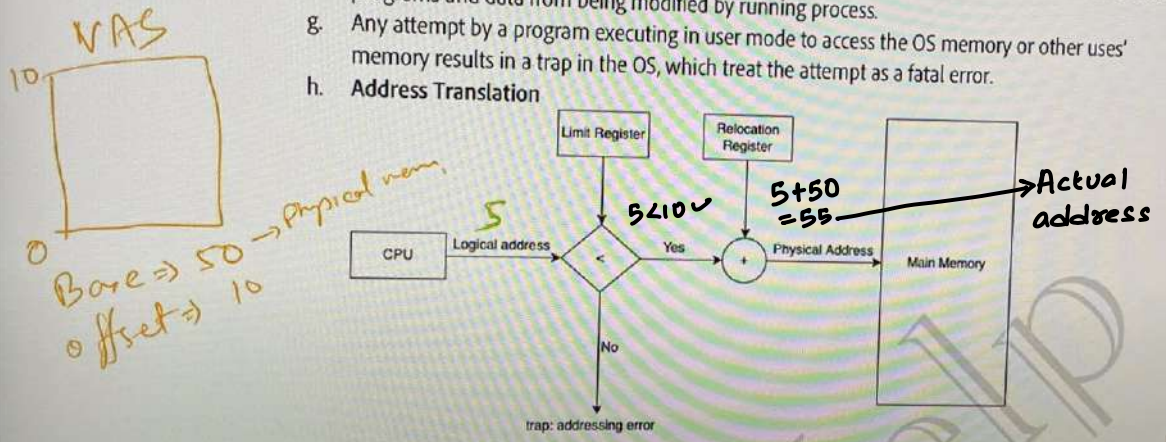
Logical addresses are generated by CPU

* OS & MMU compute the physical address
* MMU is a hardware device.
* Program needs physical address in order to execute.



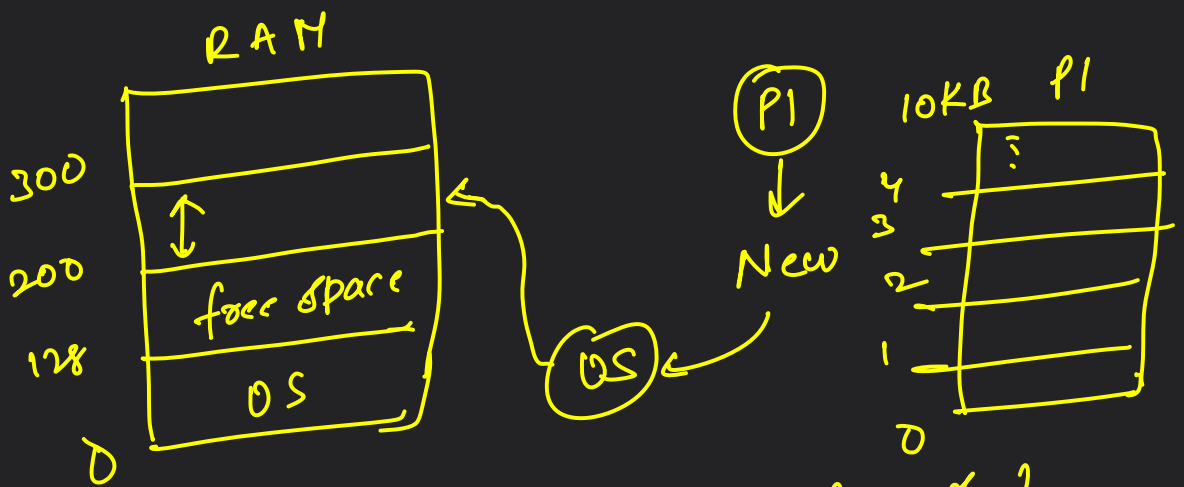
4. How OS manages the isolation and protect? **(Memory Mapping and Protection)**
   a. OS provides this Virtual Address Space (VAS) concept.
   b. To separate memory space, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
   c. The relocation register contains value of smallest physical address (Base address [R]); the limit register contains the range of logical addresses (e.g., relocation = 100040 & limit = 74600).
   d. Each logical address must be less than the limit register.

e. MMU maps the logical address dynamically by adding the value in the relocation register.
f. When CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Since every address generated by the CPU (Logical address) is checked against these registers, we can protect both OS and other users' programs and data from being modified by running process.
g. Any attempt by a program executing in user mode to access the OS memory or other uses' memory results in a trap in the OS, which treat the attempt as a fatal error.
h. **Address Translation**



Allocation Methods in Physical Memory

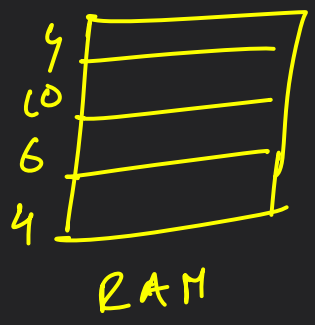
- contiguos mem Alloc
- Non-contigou mem Alloc

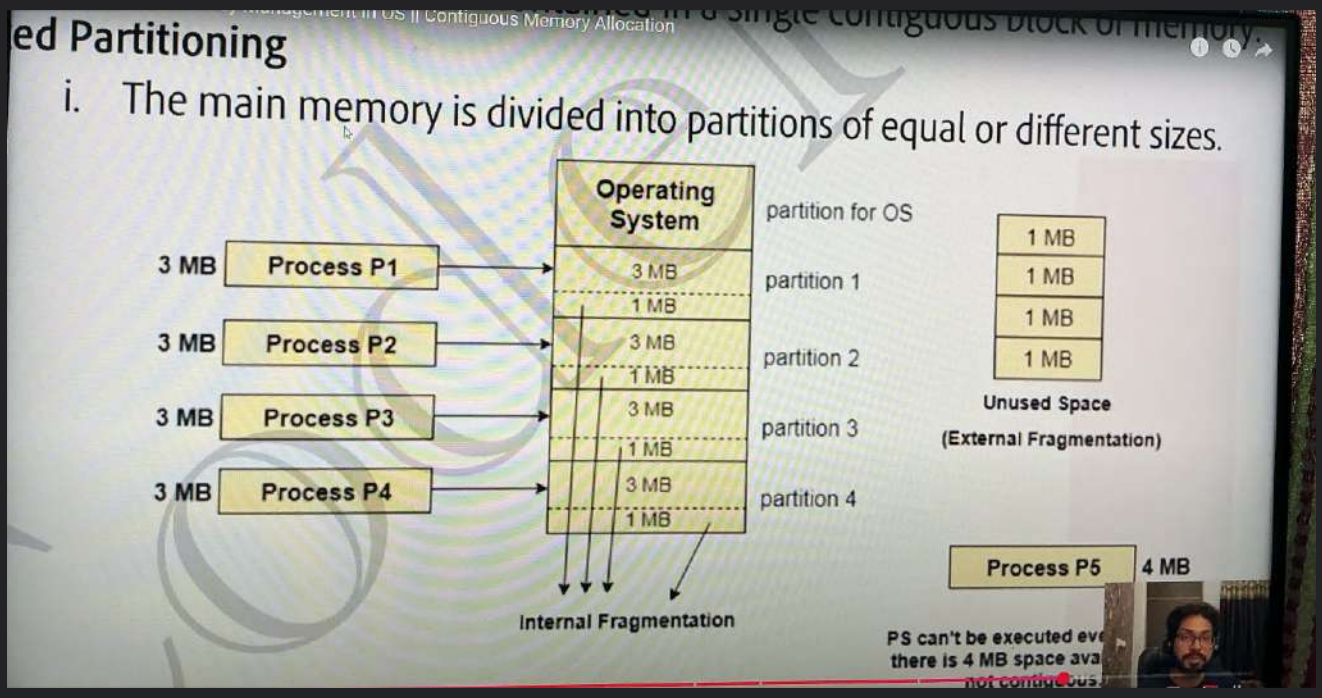


continuous (contiguous)
Put P1 to memory (RAM)

① Fixed partitioning of RAM

② Variable Size

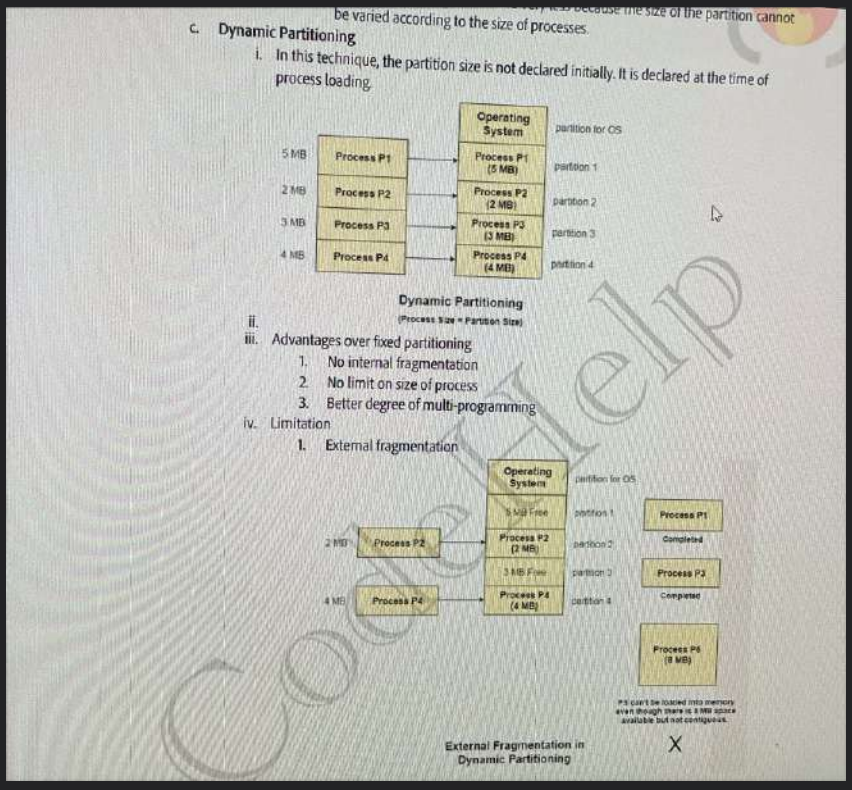




iii. Limitations:

1. **Internal Fragmentation:** if the size of the process is lesser then the total size of the partition then some size of the partition gets wasted and remain unused. This is wastage of the memory and called internal fragmentation.
2. **External Fragmentation:** The total unused space of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form.
3. Limitation on process size: If the process size is larger than the size of maximum sized partition then that process cannot be loaded into the memory. Therefore, a limitation can be imposed on the process size that is it cannot be larger than the size of the largest partition.

Next we will be discussing on dynamic partioning.



be varied according to the size of processes.

c. **Dynamic Partitioning**

i. In this technique, the partition size is not declared initially. It is declared at the time of process loading.

Dynamic Partitioning

(Process Size = Partition Size)

ii.

iii. Advantages over fixed partitioning

1. No internal fragmentation
2. No limit on size of process
3. Better degree of multi-programming

iv. Limitation

1. External fragmentation

External Fragmentation in Dynamic Partitioning

X

P1 & P3 are completed

P1 = 5MB

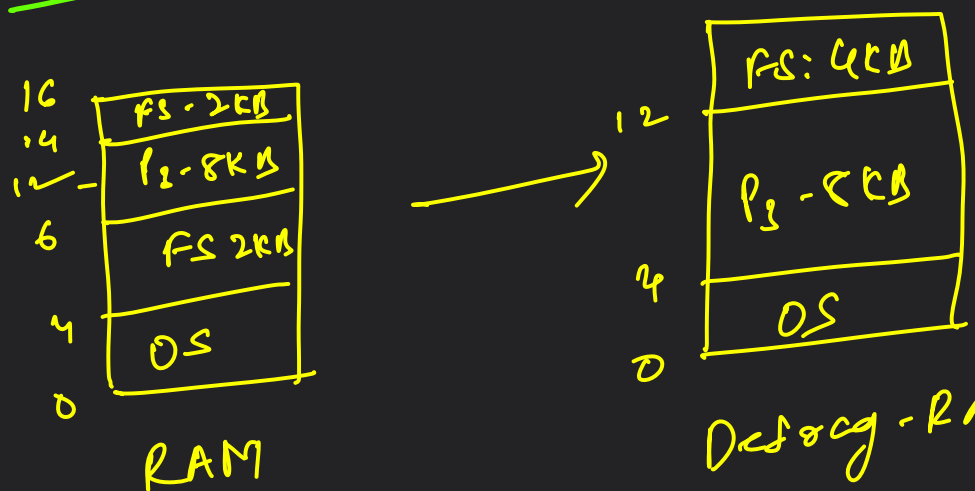
P3 = 3MB

we have 8MB free space

So Now P5 comes with 8MB, it can't utilese the above 8MB

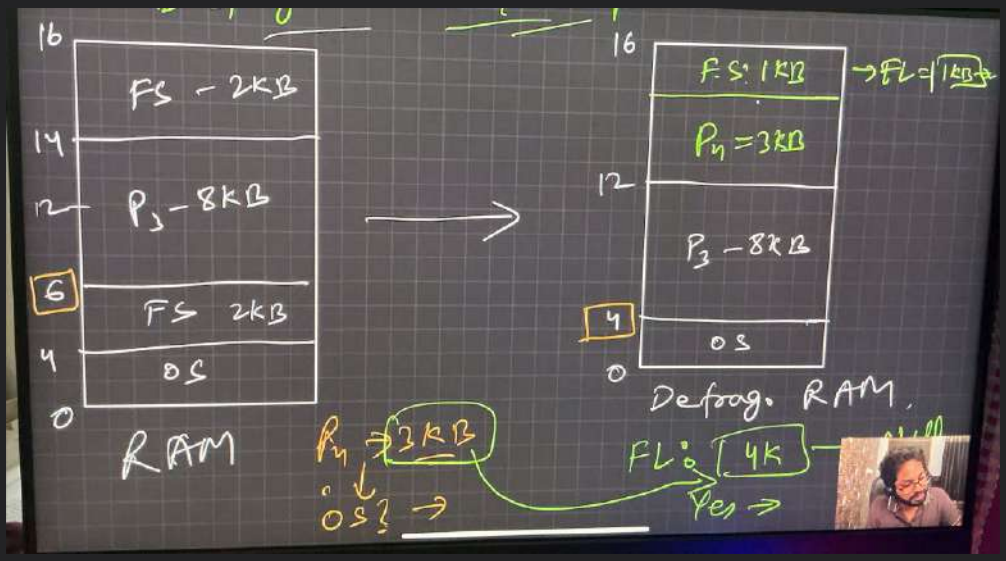
* For free space memory mgmt we are using a free list
* It is a type of linked list

Defragmentation / compaction:



Re-arranges scattered files/blocks into continuous memory locations (once files are deleted, free space gets scattered into

Small chunks

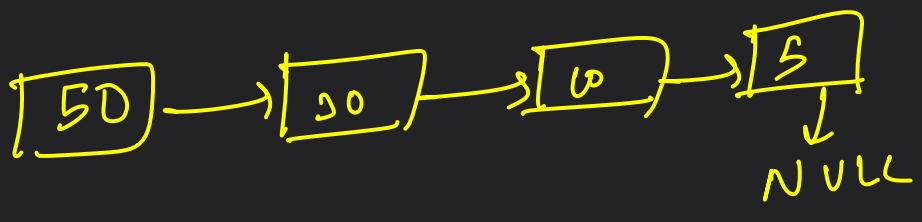


Take the chunks, combine them and do contiguous location

* Time consuming (Defragmentation)

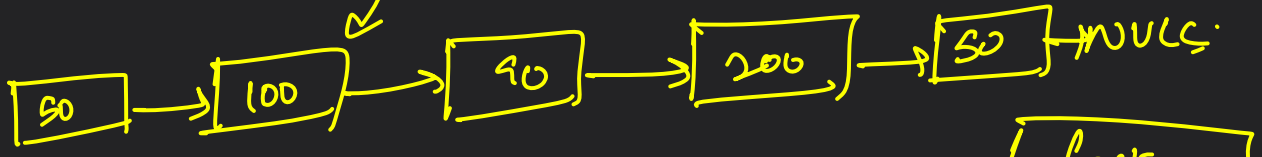
Just like RAM, Disk can also be defragmented.

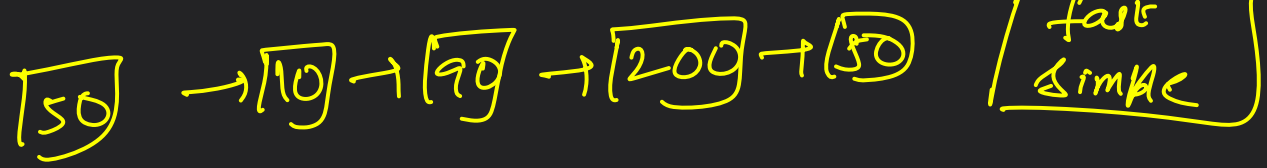
Free-space called as holes represented through Linked List



P4 20KB How to fit in free list:

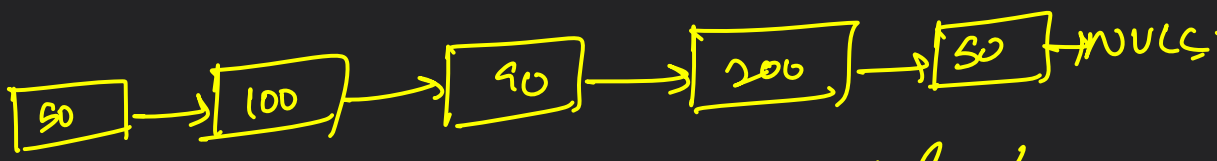
① First fit (90KB (Big enough first))





fast & simple

② Next fit



$P_2 \rightarrow 10KB$ (Previous saved next loc)

③ Best fit (Smallest hole that is big enough)

50 → 100 → 90 → 200 → 50Ks

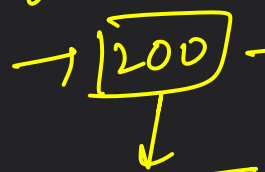
↓

40KB

④ Worst fit: Allocate longest hole

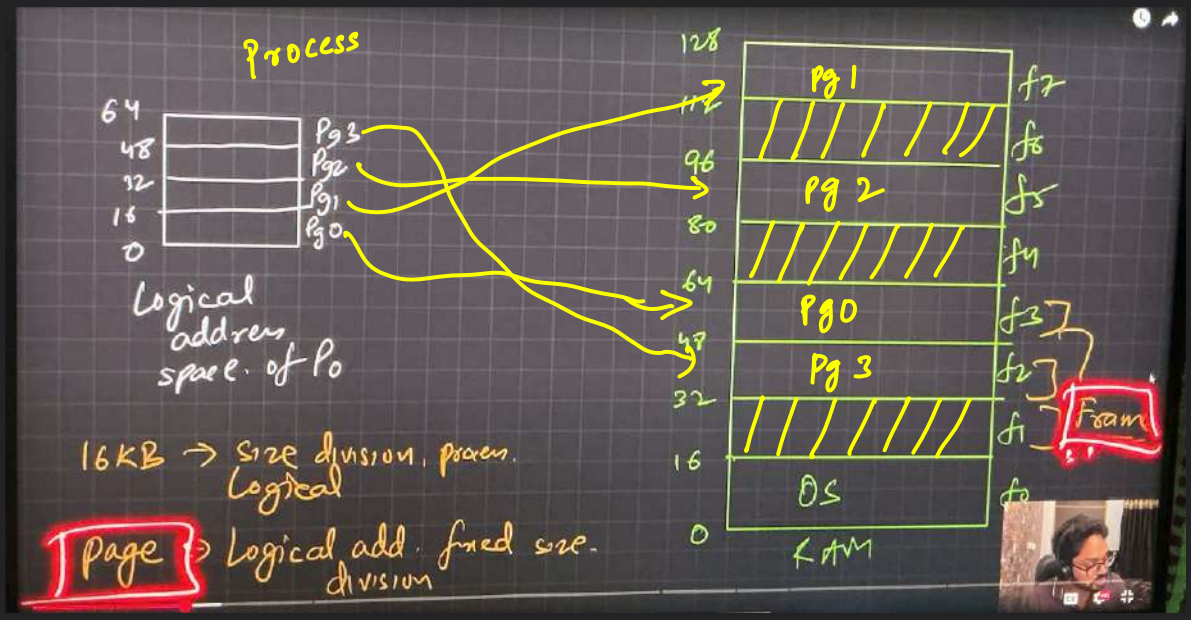
40KB

50 → 100 → 90 → 200 → 50Ks



← 200 →

Paging (Non-contiguous)



Page size = frame size

Page table

Logical pageno		Physical frame
00	Pg 0	frame 3
01	1	7
10	2	5
11	3	2

L.A → P0 → 64 bytes

$a^b = 64$

6 bits ⇒ 25 = 01 | 1001 (p | d, offset)

Logical address 25

01 → Pageno.

Pageno 1 Base → 16 + 9 (Offset) ⇒ 25

Paging in Operating Systems - Memory Management

Paging Overview -

1. In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory.
2. In this scheme, the operating system retrieves data from secondary storage in same-size blocks called pages.
3. Paging is an important part of virtual memory implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.
4. Non-contiguous memory allocation
5. Helps prevent external fragmentation
6. Logical address space is divided into equal size pages
7. physical address space is divided into equal size frames
8. Page Size = Frame Size

>> Logical Address or Virtual Address (represented in bits) - An address generated by the CPU

>> Physical Address (represented in bits) - An address actually available on memory unit

>> The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.