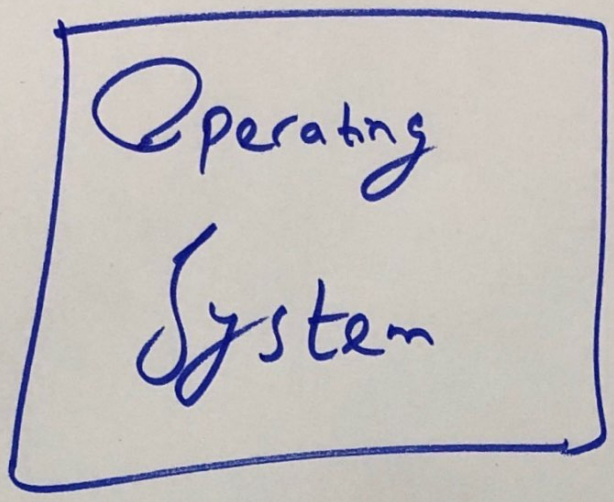


*) OS :-

- Manage Resources
- Execute programs in effective manner
  - Act as resource manager
  - Hide underlying complexity

*) If No OS :-

- Resource exploitation by 1 App
- No memory protection

*) OS Functions :-

- Execution of programs with isolation & protection
- Resource management
- Act as interface b/w resources and app
- Hide underlying complexity

*) Goals of OS :-

1. Max CPU Utilization
2. Less Process starvation
3. High Priority Job Execution

*) Type of OS :-

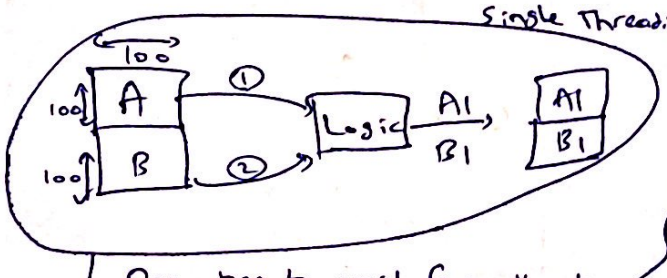
- SBMMMDR
  - ① Single Process OS
    - ① × ② × ③ ×
    - ms DOS
  - ② Batch Process OS
    - Processes Large amount of data in batches
    - ① × ② × ③ ×
    - Punch cards
  - ③ multiprogramming OS
    - Single CPU
    - Context Switching involve
    - PCB (Process control Block) involve
    - eg system → THE
  - ④ multitasking OS
    - Single CPU
    - Context Switching
    - Time sharing
      - Time Quantum
    - eg system → CTSS
  - ⑤ multi-Processing OS
    - multiple CPU
    - Context switching
    - Time sharing
    - eg system → Windows

6) Distributed OS
- multiple CPU
- multiple Users
- multiple Computers
- Loosely coupled
  - independent systems involved
- eg LAN connected CPUs

7) Real Time OS
- Used where error chances needs to be minimal
- Flight systems

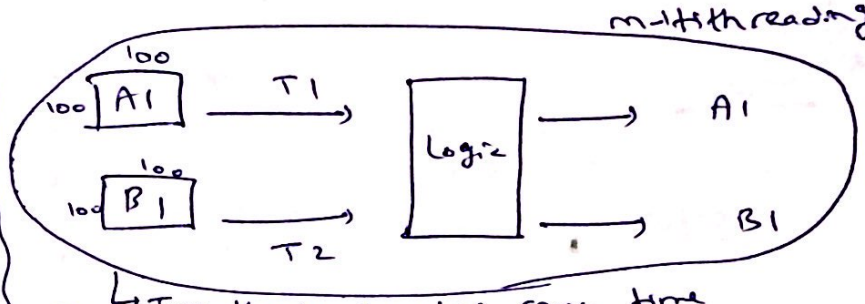
*) Multi Threading :-
- Process
  - Program Under execution
- Thread
  - Lightweighted process
  - Independently executable program
- eg JPG to PNG

Single Threading



↳ One has to wait for other to finish

Multithreading



↳ Two threads executing same time

*) Multi tasking :-
- More than one processes being executing simultaneously
  - Time sharing phenomenon
- No of CPU = 1
- Context switching b/w processes
- Isolation & memory protection

*) Thread Scheduling :-

Thread Context Switching	Process Context Switching
① OS save current state of thread and switch to another thread ↳ stored in Thread control Block	① OS save current state of process & switch to other process ↳ Stored in Process Control Block
② No switching of memory address	② Switching of memory address
③ Fast due to ②	③ Slow due to ②
④ CPU cache state preserved	④ CPU cache state flushed

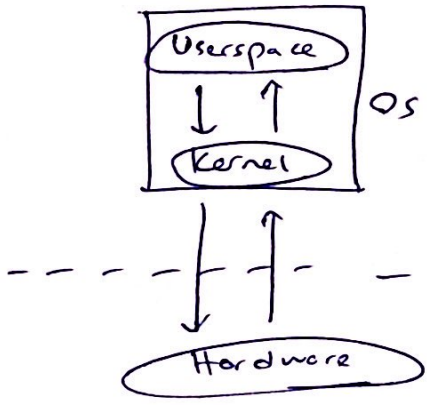
*) Components of OS :-
- ① Usersspace
- ② Kernel

Userspace

- No hardware access
  - Access hardware using kernel
- Convenient environment user apps
  - GUI
  - CLI
- Interact with kernel

*) Kernel :-

- Head of Os
- Interact with hardware



*) Multi Threading :-

- Subprocess called threads being executing simultaneously
- No of CPU ≥ 1
- Context Switching b/w threads
- No Isolation & protection

Imp → MultiThreading Vs Multi Tasking

unlike multitasking where OS must allocate seperate resources and memory to each program.

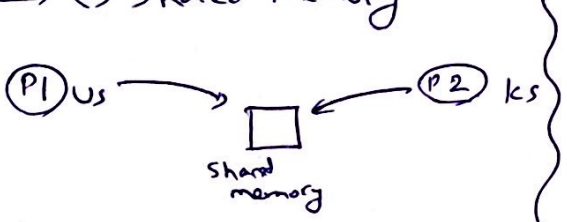
Here, Resources and memory is being shared among threads so no memory Protection and isolation

*) Functions of Kernel :-

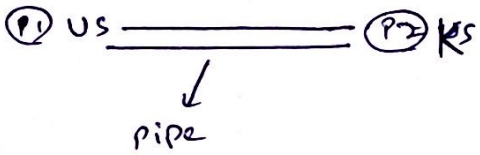
1. Process management
   - create, terminate, sync, communicate
2. Memory management
   - (Allocate / Deallocate, Free space mgmt)
3. File management
   - create, delete, directory management
4. I/O management
   - Imp works include
     1. Spooling
        - Printer common Print 20 pages
     2. Buffering
     3. Caching

* US and KS communication is done via IPC (Inter Process Communication) mechanism

- IPC is done via
  - ① Shared memory



  - ② message Passing



* Types of Kernel :-

- ① monolithic Kernel
- ② micro Kernel
- ③ Hybrid Kernel

Monolithic Kernel

- ① ✓ ② ✓ ③ ✓ ④ ✓
  All of the 4 functions are done in this kernel itself
- Fast communication b/w components of kernel
- Disadvantage
  - Kernel Bulky
- eg MSDOS

Micro Kernel

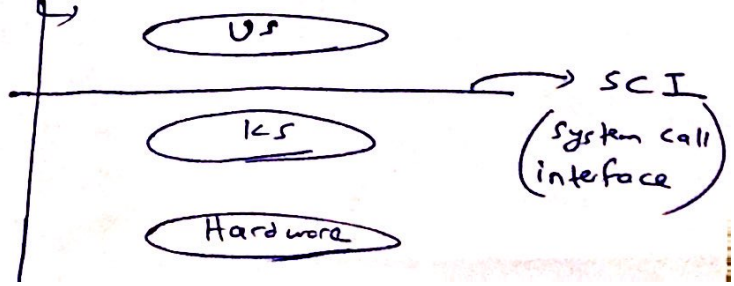
- ① ✓ ② ✓ — Done in kernel
- ③ ✓ ④ ✓ — Done in Userspace
- Less Bulky, more reliable, stable
- Disadvantage
  - Less performance
    - ↓ bcz
      Overhead switching b/w usermode and Kernel mode
- L4 Linux eg

Hybrid Kernel

- Combination of monolithic and micro Kernel
- ① ✓ ② ✓ ④ ✓ — Done in Kernel space
- ③ ✓ — Done in Userspace
- eg macOS, Windows

* System Calls :-

- Mechanism for userspace to communicate with Kernel space to further communicate with hardware

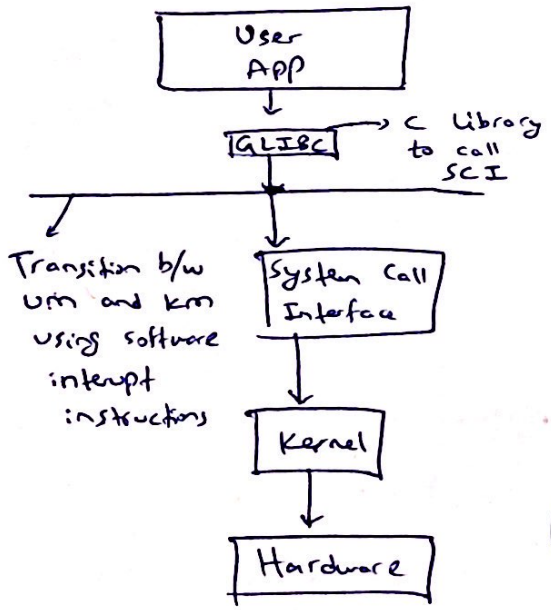


imp

- Switching of process from Usermode to Kernel mode
- System Calls are implemented in C language
- eg → GUI (New folder)
  - CLI → (mkdir)

imp

- How do Apps interact with Kernel?
  - → System Calls



*) Software Interupt :-

→ Transition b/w Uuser mode to Kernel mode is done by software interrupt

*) Types of System Calls

→ FCPDI

1. File management
   - → create, delete, open, close, read, write etc
2. Communication management
   - → send, recive msgs
   - → create delete information communication channels
3. Process Control
   - → end, abort, load, wait
4. Device management
   - → request device release device read, write
5. Information maintenance
   - → get time, set time set attributes

*) What happens when you turn on computer? :-

1. Power ON
2. CPU Loads BIOS (Basic I/O system) or UEFI (Unified Extensible Firware Interface)
   - → CPU initiate
   - → Goes to a chip BIOS chip
3. BIOS or UEFI run tests & initiate hardware

- i.e Load some settings from memory area
  - Backed by CMOS battery
- Power ON Safe Test (POST)

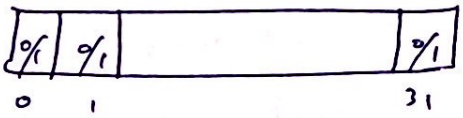
④ BIOS or UEFI hand off to Boot Device
- Disk (HDD or SSD)
- USB
- CD

⑤ Boot loade program run and On the OS
- It is present at (Boot loader)
  - MBR (Master Boot Record)
  - Old system
  - BIOS used it
- It is ~~(also)~~ present at disk Partitions in new systems
  - UEFI used it
    - ↓
    - new version of BIOS

⑥ BIOS Loader Loads full OS
- Windows
  - bootmgr.exe

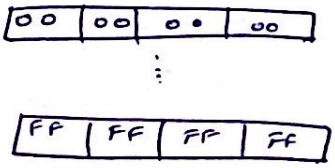
*) 32 bit Vs 64 bit :-

- Registers are implemented using transistors or data circuit chips
- 32 bit processor means it can hold 32 bits (4 bytes) of data at a time

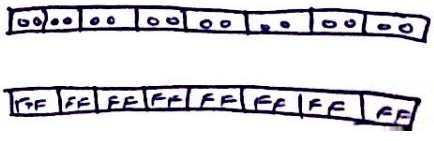


- CPU Locates memory address
- It can Locate 2^{32} unique addresses
- It means at 4 byte 32 bit processor can support 4 GB (2^{32}) of data

Advantage
- Addressable space
  - 32 bit CPU
    - 2^{32} diff memory locations



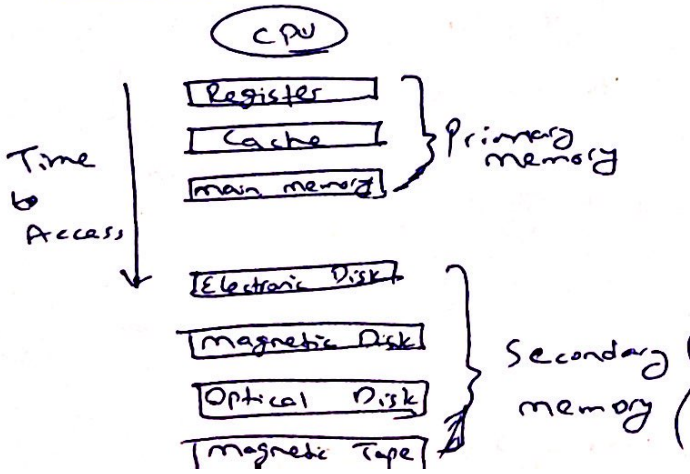
- 64 bit CPU
  - 2^{64} diff memory locations



- Resource Usage
  - 32 bit → 4 GB RAM can be installed thats it
  - 64 bit → 17 179 869 184 GB can be installed
- Performance
  - Data Process at once
  - 32 bit → 1 instruction cycle. ↓ 32 bit process
  - 64 bit → 1 instruction cycle ↓ 64 bit process
- Compatibility
  - 64 bit processor can handle 32 bits and not vice versa
- Best graphic performance
  - 64 bit

- Register
  - Smallest Unit of Storage
  - o/p
  - Fast Access
- ① Cost
  - Register high
  - Cache lower
  - main memory Lowest
- ② Access speed
  - Register high
  - Cache lower
  - main memory lowest
- ③ Size
  - Register → Bytes
  - Cache → Kbs
  - MM → GB
- ④ Volatility
  - Data saved when computer off or not
  - In secondary → Saved
  - In Primary → Not saved

*) Types of Storage :-



*) Process :-

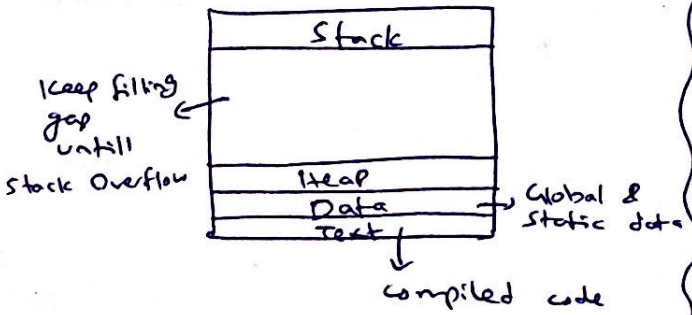
- Program under execution
- Why Process?
  - User work → way

Imp

→ How OS creates Process / Program to process conversion

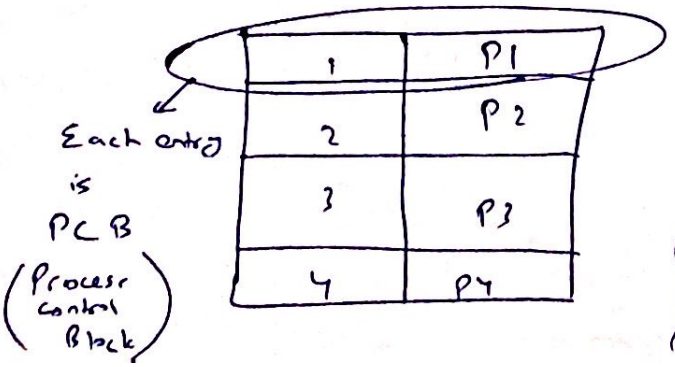
1. OS Loads Program & static data to memory
2. Allocate Runtime Stack
3. Allocate Heap
4. I/O tasks
   - Input handle allocate
   - Output handle allocate
   - Error handle
5. OS handoff control to main()

→ Architecture of Process in memory



→ Thats it ☺

*) Process Table :-



*) PCB :-

→ A data structure to store info about process

→ Present in main memory

Process ID	→ Unique identifier
Program Counter	→ Count the sev of instructions in CPU
Process State	→ New, wait, Run
Priority	→ Number
Register	→ save state for context switching
Open file List	→ Save from which file to read/write
Open Device List	→ Save from which device to I/O

*) Process States :-

1. New State
   → Program to process conversion is being done
2. Ready State
   → Process is in memory Ready Queue
3. Running State
   → CPU allocations
4. Waiting State
   → I/O operations wait is done
5. Terminated State
   → Process finished

*) Job scheduler / Long Term Scheduler (LTS) :-

↳ Process or work of passing the program from main memory to new state and then to ready state is done by LTS or Job scheduler

*) CPU scheduler / Short term Scheduler (STS) :-

↳ A scheduler which takes the process from ready queue and dispatch to CPU or to running state

*) Degree of multiprogramming :-

↳ At a time how many processes can stay at ready queue

↓

LTS handle it

*) How LTS ? :-

↳ After some time, Job scheduler checks whether there is any other program to be sent to new state and then to ready state.

↳ This (some time) delay is why its called LTS.

*) How STS ? :-

↳ Always looking for, as soon as the running state is free another process from ready queue is sent to it.

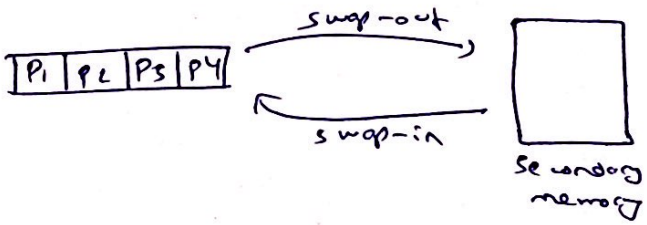
↳ NO delay so STS

*) Swapping :-

↳ MTS (Medium Term Scheduler)

↳ Swap-out, Swap-in is done in MTS

↳ If there are multiple processes open simultaneously lets say



Ready Queue dont have space to store P3, P4 so they will get swap-out to secondary memory and when some process from P1, P2 is done then memory is available in ready Queue then P3, P4 gets swap-in to ready Queue.

↓

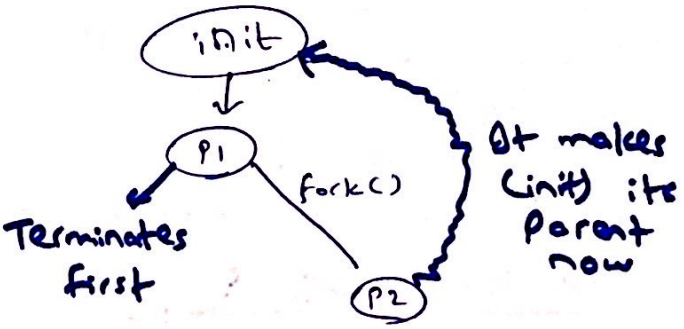
Done by kernel

*) Context Switching :-

- Done by kernel
- Speed depends on
  - Speed of RAM
  - Quality of registers
- Context switching is **pure overhead**
  - means system does no useful work (code execution) while switching
- Context switching is when a high priority work comes in then CPU stores the current state of process in PCB. Lets say P1 was going on, it gets stored in PCB of P1 and P2 gets loaded in CPU and work goes on.
- Process loading needs some information like program counter and stuff and all of it is present in PCB of that process so PCB plays vital role in context switching.

*) Orphan Process :-

- Parent process is "init" with Process ID (PID) = 1
- when createProcess() i-e fork() is called then a new process gets created with a parent that called the fork().
  But if parent has called a fork() to create child process and parent terminates first then child process is left with no parent (orphan). So now its parent will be "init".
- Kernel does this
- **Rule**
  - A parent should wait for child process to finish first, before execution but in this case its being overlooked by programmer



*) Zombie Process :-

- Lets say process P1 waited for child process before its own execution
- Child process completes its execution and exits, but the parent process does not immediately checks its exit status.
- During the delay before the parent process checks, the child process remains in the process table. The time duration of child process when it stays in process table even after completing its execution for that time it is called zombie process.
- Once the parent process checks the exit status using **wait** or **waitpid** the child process is removed from process table. This is called **reaping of zombie process**
- **Rule** → Same Orphan process rule

*) Process Scheduling / CPU Scheduling :-

- Process of taking a process from ready queue and dispatch to CPU is called CPU scheduling
- Done by STS
- **Types**
  - ① **Non-Preemptive Scheduling**
    - Once CPU is allocated to a process then it will only release it when it gets terminated or gets to wait when go for I/O
    - Low CPU utilization
    - Time Quantum phenomenon is missing
  - ② **Preemptive Scheduling**
    - Same as Non-preemptive but Time Quantum when gets expired it releases CPU too
- **Starvation**
  - When a process is not allocated time to be dispatched to CPU in running state then it means that process is starved
  - More in Non-preemptive

- CPU Utilization
  - more in Preemptive
  - less in Non-Preemptive
- Overhead
  - more in Preemptive
  - less in Non-Preemptive

*) Goals of CPU Scheduling :-

- Max CPU Utilization
- Min TurnAround Time
  - (CT - AT)
    - Completion Time − Arrival Time
  - Time b/w when a process comes in Ready Queue and when a process exit
- Min Wait Time
  - (TAT - BT)
    - TurnAround Time − Burst Time
  - Process spend time to wait for CPU allocation

Burst Time ⟶ Time required by process to complete its execution

- Min Response Time
  - Time duration b/w when process gets into Ready Queue and when process gets CPU for first time
- Max Throughput
  - No. of processes per unit time

*) Scheduling Algorithms :-

1. FCFS (First come first served)
2. SJF (Shortest Job First)
   Non-Preemptive version
3. Priority Scheduling (Non-Preemptive)
4. Priority Scheduling (Preemptive)
5. Round-Robin Algorithm
6. Multi-Level Queue Scheduling
7. Multi-Level Feedback Queue Scheduling
8. SJF (Preemptive)

*) 1. FCFS (First Come First Served) :-

- Process getting into ready queue first gets dispatched to CPU first.
- **Convoy Effect**
  - If a process with Large Burst Time comes first then Average Waiting Time gets high.

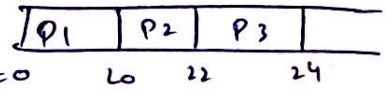
→ eg1

First make gantt chart

P.No.	Arrival Time	Burst Time	Completion Time	TAT = CT-AT	wait Time = TAT-BT
1	0	20	20	20	0
2	1	2	22	21	19
3	2	2	24	22	20

Avg WT = 13

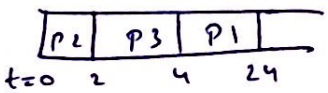
Gantt chart



Example 2
→ P.No gets reversed

PNo	AT	BT	CT	TAT	WT
2	0	2	2	2	0
3	1	2	4	3	1
1	2	20	24	22	2

Avg WT = 1



→ Poor Resource management

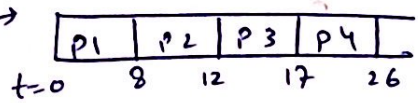
*) SJF (Shortest Job First)
Non-Preemptive version

→ Process with Least Burst Time will get CPU first
→ Fact is that before execution we cant know exact burst time it is just imagined value
→ To decrease Avg Time
→ eg

PNo.	AT	BT	CT	TAT	WT
P1	0	8	8	8	0
P2	1	4	12	11	7
P3	2	9	26	24	15
P4	3	5	17	14	9

Avg WT Time = 7.75

Gantt chat



Convoy Effect present here

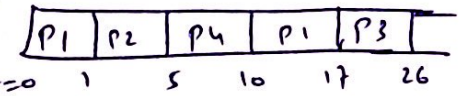
→ This is bcz its non-preemptive so thats why P1 is written first and not P2 (least burst time)

*) SJF (Preemptive)
→ eg

PNo	AT	BT	CT	TAT	WT
P1	0	8	17	17	9
P2	1	4	5	4	0
P3	2	9	26	24	15
P4	3	5	10	7	2

Avg WT = 6.5

Gantt chart



Convoy Effect can be avoided here using this technique

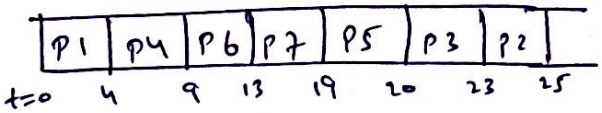
*) Priority Scheduling (Non-Preemptive) :-

↳ Priority is given to Process when they enter in ready queue

↳ eg

PNo	Priority	AT	BT	CT	TAT	WT
1	2	0	4	4	4	0
2	4	1	2	25	24	22
3	6	2	3	23	22	19
4	10	3	5	9	6	1
5	8	4	1	20	16	15
6	12	5	4	13	8	4
7	9	6	6	19	13	7

Avg WT = 9.719

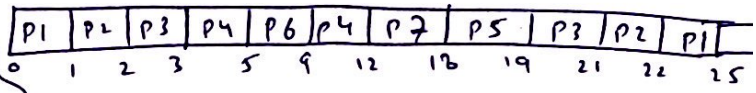


Convoy Effect is present but at small extent

*) Priority Scheduling (Preemptive) :-

PNo.	Priority	AT	BT	CT	TAT	WT
1	2	0	4	25	25	21
2	4	1	2	22	21	19
3	6	2	3	21	19	16
4	10	3	5	12	9	4
5	8	4	1	19	15	14
6	12	5	4	9	4	0
7	9	6	6	18	12	6

Avg TWT = 11.42



more overhead is present
Convoy Effect is Present

*) Biggest Drawback in Priority Scheduling (Preemptive, Non-Preemptive) :-

↳ Indefinite waiting is Present

↳ when a high priority jobs keeps on coming in the ready queue then the low priority jobs already present in ready queue never gets the CPU

*) Solution to Indefinite waiting :-

↳ Ageing

↳ we gradually increase the Priority of low-Priority Processes

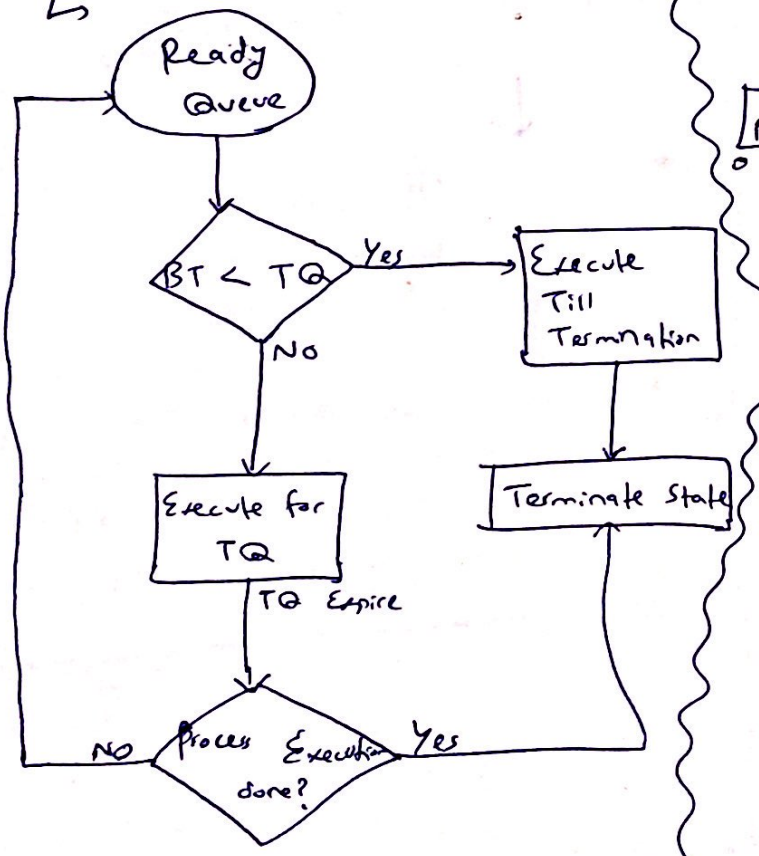
*) Round Robin Algorithm :-

↳ Response Time is low

↳ most Popular Algo

↳ FCFS (Preemptive version)

↳ Criteria (AT + Time Quantum (TQ))



↳ Does not depend on BT

↳ No Process will wait forever so no starvation

↳ No convoy Effect

↳ If TQ is less then more overhead (Disadvantage)

↳ eg

P	AT	BT	CT	TAT	WT
1	0	4	8	8	4
2	1	5	18	17	12
3	2	2	6	4	2
4	3	2	9	6	5
5	4	6	21	17	11
6	6	3	19	13	10

Avg WT = 7.3

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6	P5

0 2 4 6 8 9 11 13 15 17 18 19 21

TQ = 2 seconds

Note

↳ Each one of the above Algorithms are implemented in some optimised ways which are

↳ ① Multi-level Queue Scheduling

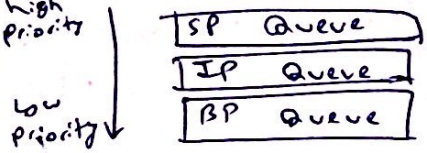
↳ ② Multilevel Feedback Queue Scheduling

① Multi-Level Queue Scheduling :-

- Process is divided into
  - System Process (SP)
    - Created by OS
  - Interactive Process (IP)
    - User input req
  - Batch Process (BP)
    - Background Process
- Now when a process comes, it goes into one of these sub-queues based on its nature, memory, size etc. And stays in that and get scheduled
- Scheduling in these 3 sub-queues is based on **Preemptive Priority** scheduling algorithm.
- **Problem** → Indefinite waiting
  - Only after completing the process of high priority sub-queue it goes to low-priority sub-queues
- Convoy Effect present
- No inter-queue movement of processes

Note

- Ready Queue is divided into 3 logical queues based on process type



- Every Queue has its own scheduling algorithm implementation

eg

SP → RR, IP → RR, BP → FCFS

② Multi-Level Feedback Queue :-

- Multiple sub-queues
- Allowed inter-queue movement of processes
- High Burst Time ⟶ Less Priority

  I/O, IP Queue ⟶ High Priority
- Thus does not cause starvation, we use Ageing to increase the priority overtime
- Less Starvation than MLQ

Design of MLFQs :- (we must know and fig out)

1. No of Queues
2. Which Algo to schedule each queue
3. method to upgrade process to higher queue (Ageing)
4. method to demote queue
5. Process will go to which queue at first

Note :-

- Out of all 8 scheduling Algorithms
  - ★ Convoy Effect present in
    - Round Robin
    - SJF Preemptive
  - ★ Overhead not present in
    - FCFS
    - Non-Preemptive SJF
    - Non-Preemptive Priority

*) Concurrency

- Execution of multiple instructions at the same time

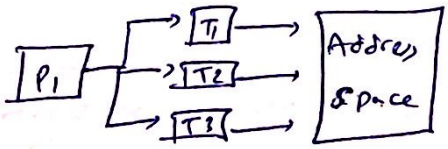
Threads

- Light weight Process or sub process
- Independent path of execution of a process
- Multithreading is only beneficial if there is multiprocessing system bcz in that way context switching b/w threads can be avoided.
- eg multiple tabs in chrome, ms word (Text editor, spell checker, Text format)

Difference

- Process
  - Every Process has its own address space

- Threads
  - Multiple threads can have same Address space



Thread Scheduling

- Threads are scheduled based on Priority
- OS assign time slices at runtime to threads

Threads Context switching

- OS saves current state of thread and switch to another thread of same process
- Doesn't include switching of memory address
- Include switching of Program counter, register & stacks
- CPU's cache is preserved as its same process
- Fast switching than process switching

- Threads have **TCB** (Threads control block)

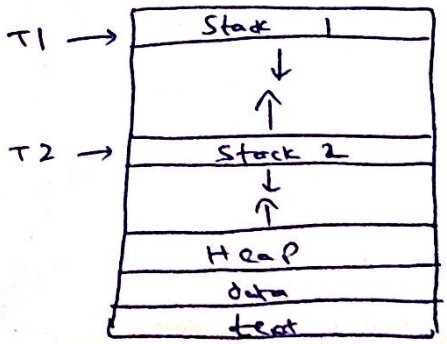
> Program Counter help thread to get access to CPU & Thread scheduling also helps.

- Benefit of multithreading is only in multi-processing CPU

Benefits of multi-Threading

- Responsiveness
  - One thread will I/O
  - One will backup
  - One will cloud sync etc simultaneous behavior
- Resource sharing
  - Memory Addresses are shared
  - Cache is shared
    - Low overhead
    - communication gap low
- Economy
  - Economical in context switching
  - Economical in allocate memory & resources as sharing is involved

Threads Architecture in Memory



*) Critical Section :-

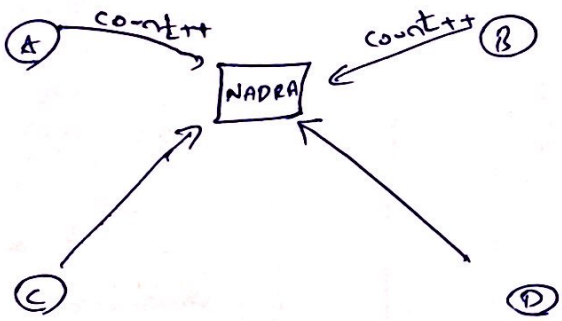
→ A shared resource in which multiple threads are coming in to work.
→ Its a segment of code where multiple threads work at same time.
eg common variables, files. Threads execute concurrently.

→ **Race Condition**

→ Count++ occurs in kernel as

→ count = 11
count++
→ temp = count
temp = count + 1
= 11 + 1
= 12
count = temp

→ Eg



Here, A, B, C, D are threads or centers which are upgrading NADRA people count.
If A and B request count++ simultaneously then two steps will occur

temp = 11 + 1	temp = 11 + 1
= 12	= 12
count = temp	count = temp
= 12	= 12

But it should have been 13
This is called race condition where two threads or more threads are racing to access/change the data. Value will be inconsistent due to thread scheduling algorithms.

→ **Solution**

→ ① Make process ATomic
Rather than doing
temp = count + 1
count = temp
make it a one step process
C++ has implementation of :
→ atomic <int>

→ ② Mutual Exclusion
→ when one Thread ~~Process~~ is inside critical section the other must wait
→ Programming Languages implement Mutex/Locks to achieve this

→ ③ Semaphores
→ Using flag/Boolean to restrict one thread and allow other to go in critical section

→ **Solution of Critical section must have 3 conditions :-**

→ ① Mutual Exclusion
→ ② Progress
→ If there are two threads scheduled to go into CS then they must have equal opportunity to go
→ T1 can go into it first or T2 can go into it first
→ They must not be

Scheduled like T1 will go first then T2 or vice versa

③ Bounded waiting
- Some threads are going into CS while other are not getting opportunity to do so.
- Infinite waiting
- There must be like after a particular time every thread must go into CS

Can we have a single flag?

```
T1                          T2
while(1){                   while(1){
  while(turn != 0){};         while(turn != 1){};
  [C.S]                       [C.S]
  turn = 1                    turn = 0
  Return statement;           Return statement;
}                           }
```

Here mutual exclusion is achieved but Progress is not achieved as which thread will go into C.S depends on the value of turn and the other will get scheduled after it

Solution to the flag problem

Peterson's Solution

- Uses flag[2]
- turn = 0/1
- flag[2] indicates if thread is ready to go to the C.S. flag[i] = true indicates it can go
- turn = 0/1 → whose turn to enter CS
- Initially flag[0] = False, flag[1] = false

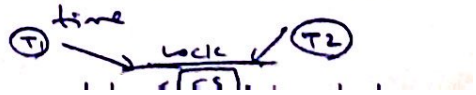
```
T1                                  T2
while(1){                           while(1){
  flag[0] = T                         flag[1] = T
  turn = 1                            turn = 0
  while(turn==1 && flag[1]==T){};     while(turn==0 && flag[0]==T){};
  [C.S]                               [C.S]
  flag[0] = F                         flag[1] = F
}                                   }
```

→ Here C.S will only get executed if while loop turns false otherwise control will stay in while

Here mutual Exclusion & Progress both are achieved.
- Here which thread will go into CS does not depend on turn variable much as whichever thread will get scheduled will go into C.S first. OS determines it in runtime using time slice.

Limitation of Peterson
- Safe for 2 Threads not more

mutex / Locks
- Allows only one thread to go into CS at a time



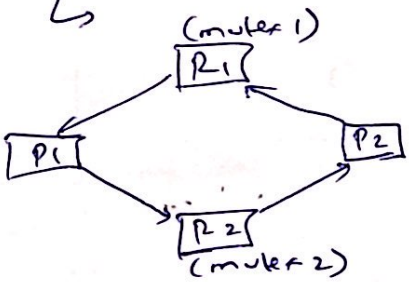
↳ Disadvantages

↳ ① Contention

↳ One thread has locked C.S and its inside. Others are waiting outside. The inside thread has given exception and has never been unblocked C.S and went out. Other threads on infinite waiting

↳ ② DeadLocks

↳



↳ ③ Debugging

↳ ④ Starvation

↳ Low priority thread is inside C.S preventing high priority thread to go in

↳ Advantage

↳ Handles more than 2 threads

*) Thread Synchronization Techniques

↳ Conditional Variable
↳ Semaphores (data structure)
↳ Peterson's solution
↳ mutex / Locks
↳ Atomicity

*) Conditional Variable :-

↳ Implemented using mutex/locks

↳ Locks/mutex has a problem called **busy waiting**

↳ When T1 is in C.S then it puts wait lock for T2. So T2 is in wait for T1 to finish. T2 is acquiring CPU cycle thus wasting it

↳ Here busy waiting is present bcz t2 is continuously checking if C.S is lock free and thus its acquiring CPU cycles

CPU cycle

↳ Time req for execution of one single processor operation eg Add

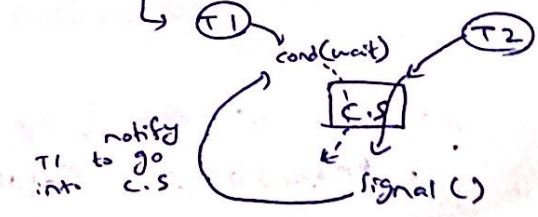
↳ Solution

↳ Conditional Variable

↳ It has two methods

↳ wait
↳ signal

↳ T1 thread will wait for a signal from T2 to go into C.S. And while waiting for signal, it executes other processes or works or it will not hold the CPU, it just block the T1



Here T1 is in wait and while its in wait it is waiting for a signal to notify it to go to C.S. Here CPU is not holding and no busy waiting is present.

T2 will go into C.S executes and after execution notify a signal to T1 wait(), it will then go to C.S and executes.

→ Threads are blocked means it is not holding CPU rather just waiting so no busy waiting present

```
wait(s){
    s → value--;
    if (s → value < 0)
        ↳ block();
          add to s blocklist
}
```

```
signal(s){
    s → value++;
    if (s → value <= 0)
        ↳ remove thread from blocklist
        ↳ wakeup(thread)
}
}
```

→ Types

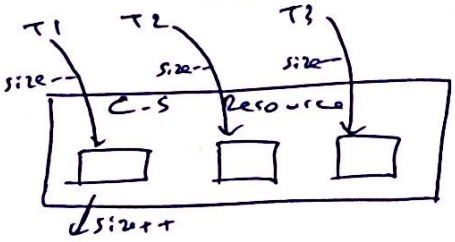
→ Binary Semaphore
- → Only two states 0 & 1
- → Control access to single instance of resource
- → 1 → available 0 → not available

→ Counting Semaphores
- → no. of instances > 1
- → Above if s = 5 then 5 instances available

*) Semaphores

→ Conditional Variable and mutex/ locks were using single instance of resource (C.S). while semaphores uses multiple instance of resource (C.S)

→ T1 ⟶ T10



3 instances of C.S

size = 3
↳ This size cant be < 0

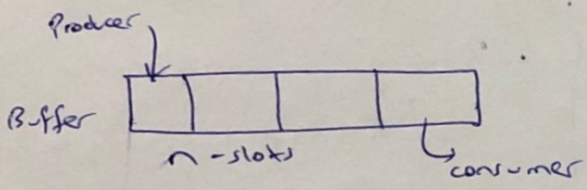
Here when one Thread goes into one instance then size of instances gets reduced

So at a time 3 threads can go into 3 instances and others have to wait. And when one comes out then size ++ then other can go in. Otherwise it will stay block

Binary Semaphore is actually Mutex because single instance with only 2 states 0 & 1 in both cases

*) Producer–Consumer Problem / Bounded Buffer Problem

- It has two threads
  - Producer Thread
  - Consumer Thread
- ~~Basically it has~~ a finite buffer is present in this scenario where Producer comes to fill and consumer comes to choose and consume.



- **Problems**
  - Buffer is basically a critical section (shared resource) so synchronization must be done b/w producer & consumer
  - If buffer is full then producer cant insert data into it.
  - if buffer is empty then consumer must not consume data
- **Solution**
  - Semaphore
    - As only two threads (producer, consumer) present so binary semaphore will be used to acquire lock on buffer
    - m, mutex
    - empty semaphore
      - Initial value n (no. of slots)
      - Tracks empty slots
    - full semaphore
      - Initial value 0
      - Tracks filled slots

Producer

```
do{
wait(empty); // wait untill
                empty > 0
                then empty → value

wait(mutex); // To provide
                mutual exclusion

[CS → Add data to buffer]

signal(mutex);
signal(full); // To increment
                full value
}while(1);
```

Consumer

```
do{
wait(full); // wait untill
               full > 0
               then full--

wait(mutex);

[C.S → Remove data from buffer]

signal(mutex);
signal(empty); // To increment
                  empty value
}while(1);
```

*) Reader - Writer Problem

- Threads
  - Reader Thread
  - writer Thread
    - write
    - update
- A database present
- Assumptions
  - if readers > 1 then no issue
  - If writers > 1 or if 1 writer with some reader threads then race condition and data inconsistency present
    eg if one writer writes abc and reader reads abc and writer updates it to a2f then reader read inconsistent
- Semaphore solution to R-W problem
  - ① mutex
    - To achieve mutual exclusion wher read count (rc) is upgraded.
    - No two or more readers can modify rc at a time
  - ② wrt semaphore
    - Binary semaphore common for both reader, writer
  - ③ readercount → To count readers

Writer Solution

```
do{
    wait (writ);   // No two writers write at a time

C.S → Do write operation

    signal (wrt);  // To allow other writers or readers to come in
}while(true);
```

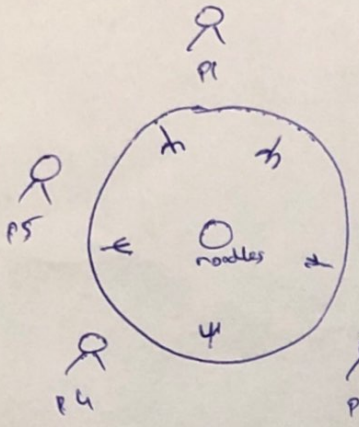
Reader Solution

```
do{
~~wait(mutex)~~
wait(mutex);   // To mutex readcount variable so avoid race condition
rc++;
if(rc==1){
    wait(wrt);   // ensure no writer can enter if there is one reader
}
signal(mutex);   // for rc

C.S → read operation

wait(mutex);   // for rc--
rc--;
if(rc==0){   // no reader left
    signal(wrt);   // writer can enter
}
signal(mutex);   // for rc
}while(1);
```

*) Dining Philosophers Problem



- 5 philosophers
- spend their life in two states
  - Thinking
  - Eating
- Sitting on a circular table, noodles present in middle, 5 forks are present (2 req for one philosopher to eat)
- Thinking State
  - Philosopher do not eat when thinking
- Eating State:
  - Ph pick 2 forks adjacent to him to eat
  - Cant pick fork already taken by another Ph
- **Solution**
  - Each fork is a semaphore (binary) fork[5]
  - wait() → Ph has acquired fork[i]
  - release() → Ph has released fork[i]

```
do{
    wait (fork[i]);
    wait (fork[(i+1)%5]);

    C.S ⟶ eat

    signal (fork[i]);
    signal (fork[(i+1)%5]);

    // think
} while(1);
```

- **Problem in above solution**
  - Semaphore ensure two members do not eat simultaneously but it creates deadlock
    - If every ph picks its left fork at the same time waiting for right one to get released
- **Solution to avoid deadlock**
  1. Allow at most 4 ph to sit simultaneously
  2. Allow a ph to pick fork only if both available otherwise do not pick single fork.

```
wait (fork[i])
wait (fork[(i+1) % 5])
```

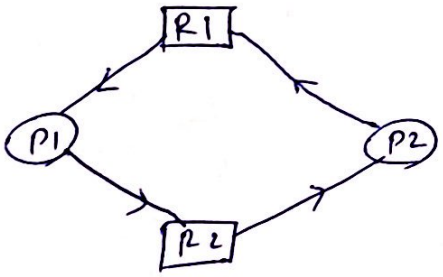
This will be made atomic and C.S by doing

```
mutex:
wait (fork[i])
wait [(fork[i+1)%5]
release mutex
```

→ ③ Odd-Even rule
→ odd ph picks his left fork then right fork
Even ph picks his right fork then left fork

*) Deadlock :-

→ System has resources like memory space, files, I/O, locks CPU instance
Process/Threads are used interchangably below

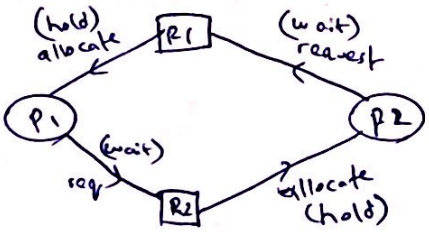


→ Process/Thread request resource and if its not available then it waits for it. Sometimes this wait is forever i-e resource is busy forever → Deadlock

→ Process acquire resource as it first requests to check if no locker on it then it uses while putting lock on it and after using it releases it.

→ **Necessary Conditions for DL**
4 conditions must exist simultaneously
→ ① Mutual Exclusion
→ if R1 is allocated to P1 then if P2 requests for R1 then it wont get R1 until P1 releaser it

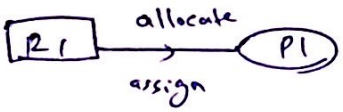
→ ② Hold and wait



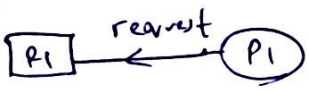
→ ③ No Preemption
→ Any resource allocated to process will be voluntarily released by process after using it
Any other process cannot take it from the process holding it.

→ ④ Circular wait
→ DeadLock is present in a circular fasion
→ Similar to hold & wait

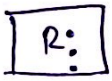
→ **Resource Allocation Graph (RAG)**
→ Used for system Representation
→ ~~Used~~ It has
→ ① Vertex
→ Process Vertex
P
→ Resource Vertex
R
→ ② Edges
→ Assign



↳ Request

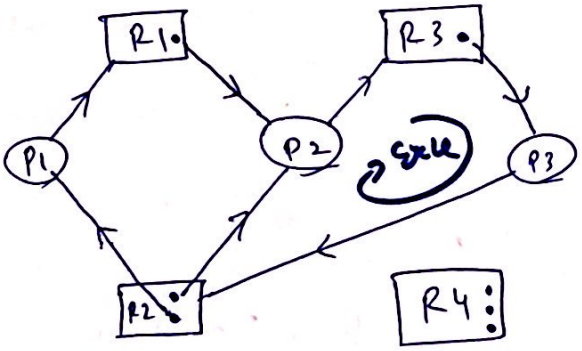


③ Multiple Instance of Resource

↳ 

Here it has 3 instances or 3 cores of CPU

↳ Diagram Example

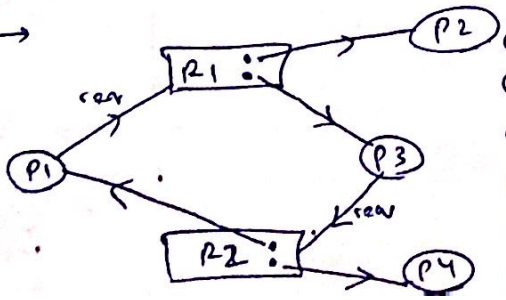


RAG Conditions

↳ ① If cycle
↳ maybe DL

↳ ② If no cycle
↳ No DL

↳ maybe DL because



↳ Here cycle is present but deadlock is not present but RAG Algorithm will return True for DL for this example

Methods to Handle DL

↳ ① Prevent or Avoid DL

↳ ② Allow system to go to DL then Detect DL and then Recover

↳ ③ Ostrich Algorithm
↳ Deadlock ignorance
↳ Assumption that app programmer has written code that will never cause DL

① Deadlock Prevention

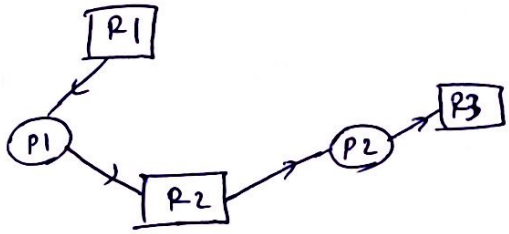
↳ We know 4 conditions cause DL exist simultaneously

↳ If any condition gets handled then no DL

↳ ① Handling Mutual Exclusion
↳ Separate sharable and non-sharable resources
↳ Non-sharable resources like read only files can be accessed by multiple Process/Threads
↳ Locks applied on sharable resources
↳ Helps to prevent DL to some extent

(Case 2)

→ Process P1, requests R1 then we check if its available if yes we allocate.
If No, then we check if R1 is being allocated to some other process P2 that is requesting some other resource.
Then ~~we~~ if this condition true we get R1 released for P1 and when work is done we get R1 again allocated to P2



Here we get R2 released for P1 then when P1 finishes we get R2 again allocated to P2

→ ② Hold & Wait

→ Ensure that whenever a process req for a resource, it must not hold any other resource

→ Protocol (A)

→ Allocate all resource before execution

→ CPU cycle can get waste

→ Protocol (B)

→ It must release all resources to request any resource

similar

(Case 1)

→ ③ No Preemption

→ If a process P1, has been allocated resource R1 and is requesting for resource R2 which is already allocated to some other process. Then P1 must release R1. Process will restart only if it can get both R1, R2.

→ **Live Lock** can occur

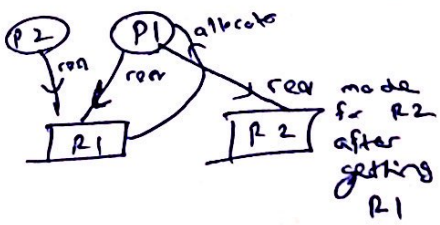
→ R1, R2 both free
P1 req R1, R2 the same time
P2 req R1, R2
Both get rejected

→ eg two people call each other at same time

→ ④ Circular Wait

→ Allocations are done in ordering

→ Both P1 and P2 require R1 and R2 simultaneously. So ordering will be done that whoever gets R1 first will be allowed to request for R2, other will wait



*) Deadlock Avoidance :-

- Kernel is provided with advance info of current state
  - Current state includes
    - ① No. of processes
    - ② Max no. of resources required for each process
    - ③ Currently Allocated resources to each process
    - ④ Max or total resources available
- We schedule processes & resources to ensure that our system stays in safe state
  - A state in which resources gets allocated to processes in such a way that at the end it avoid DL
- **Banker's Algorithm** is used to schedule
  - 3 instances of resource → A, B, C

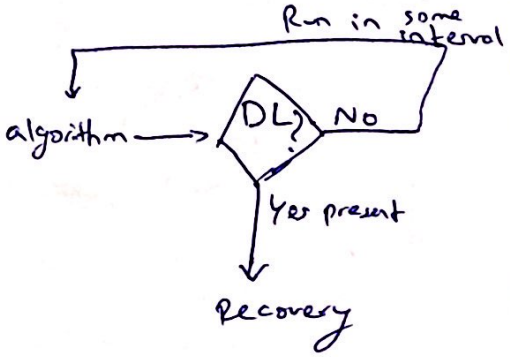
- Safe Sequence
  - Sequence in which processes gets executed ensuring safe state.
  - Example → Here in table looking at (3, 3, 2) tells only P2 can get executed first
    - Remaining Available resources will be
      3+2 3+0 2+0
      (5 3 2)
    - Then P4 can get executed using (5, 3, 2)
      And so on
  - Resultant safe sequence
    - P2 → P4 → P5 → P3 → P1
  - This is a safe sequence to execute all Processes and avoid DL

Remains = Max need − Allocated
Available = Total − Allocated total
Total resources → A → 10, B → 5, C → 7

Process	Allocated Resource			Max Need			Available			Remaining Need			
P	A	B	C	A	B	C	A	B	C	A	B	C	
P1	0	1	0	7	5	3	3	3	2	7	4	3	P1
P2	2	0	0	3	2	2	5	3	2	1	2	2	P2
P3	3	0	2	9	0	2	7	4	3	6	0	0	P3
P4	2	1	1	4	2	2	7	4	5	2	1	1	P4
P5	0	0	2	5	3	3	10	4	7	5	3	1	P5
	Total						10	5	7				
	7	2	5										

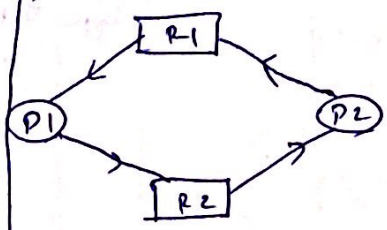
*) Deadlock Detection :-

- If (DL handling way) says to go to deadlock and the detect it and then recover it.

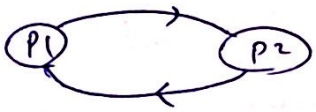


- This basically says to keep checking after some time interval for DL in system. If present then go to recovery otherwise keep on working And again check after some time interval

- Two Cases
  - (1) Single instance of each resource in system



  - This Resource Allocation Graph is converted to wait-for-graph.
  - Wait-for-graph is by removing resources



  - If cycle present after converting RAG to WFG then DL is detected true
  - (2) multiple instance of resources
    - Banker Algorithm is used to detect DL
      - Safe Sequence present ?
        - No DL
      - Safe Sequence cant be made?
        - DL detected true

*) Deadlock Recovery :-

- (1) Process Termination
  - Either terminate or kill all processes involved in DL (way 1)
  - Either kill one process. to release resource for other (way 2)
- (2) Resources Preemption
  - we preempt or release resources of Low priority process and allocate to high priority process

*) Memory Management Techniques :-

→ In multiprogramming environment, several processes can coexist in main memory. As a result, management is needed for main memory

→ Logical Vs Physical Address

→ ① Stored/Presence

→ Logical data is temporarily generated by CPU
So when a process gets in execution its been allocated some address out of virtual addresses present in virtual address space. This is called Logical address. Gets not stored Anywhere

→ Physical Address is stored in main memory.

→ ② Existence

→ Logical Address gets deallocated when the process ends

→ Physical address still contains data produced by the process after termination and if it has to be persistent data then it is stored in secondary storage somewhere and then only the physical address gets deallocated and free to reuse

→ ③ Access

→ Logical Address can be accessed by user
eg in C++ we use & (Ampersand) to access Logical Address

→ OS provides abstraction so physical address cannot be accessed by user its OS thing.

Note

Virtual Address space is a set of range of virtual addresses that an OS makes available to a process

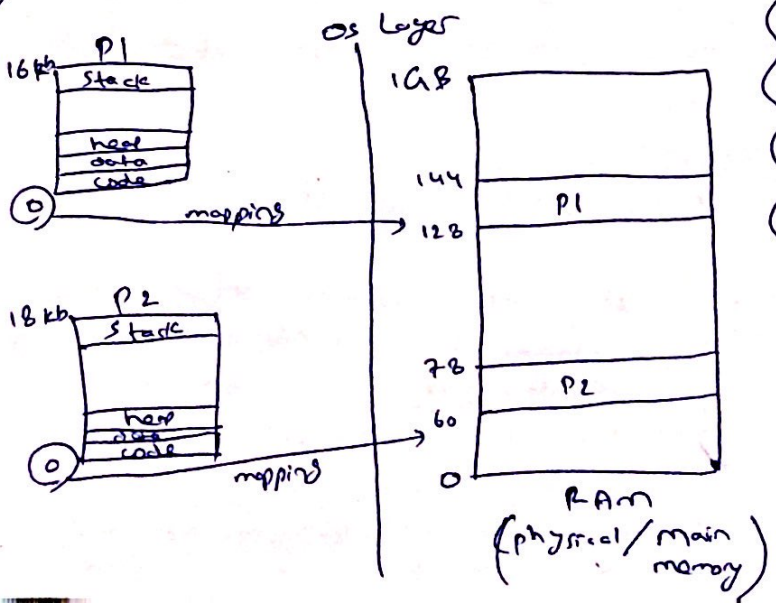
Note

Memory Management Unit
→ It is a hardware device
→ Manages mapping b/w Logical & physical address i-e os finds and allocate physical address using logical address via MMU's mapping through a process called **Address Translation**

Runtime process of mapping is done by it

Page Table
→ Data structure to store the mapping b/w virtual & physical address
→ Contains info like base/physical address, access control
→ MMU uses page table to perform address translation

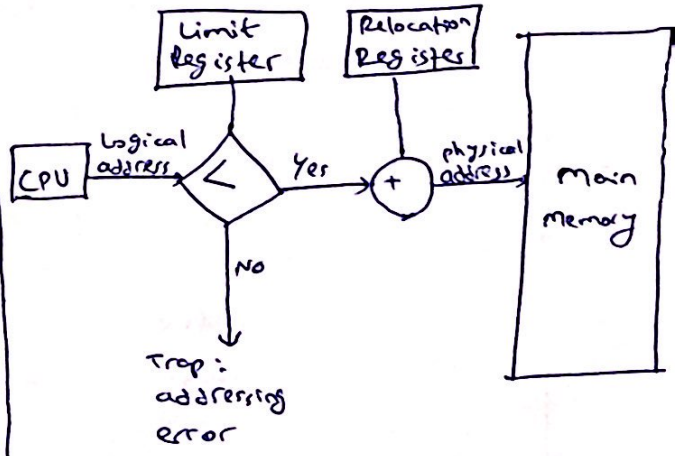
- Computation
  - MMU computes physical address during execution
  - CPU computes logical address during execution
- Range
  - Range of logical Address is from 0 to max address space rev by process
  - Range of physical address is eg p1 has range 0 to 16 Kb and in physical address this p1 is mapped from 128 lets say then range in physical address will be from 128 — 128+16 i-e 128 — 144
- Diagram



- So OS will say to every process to get a logical address starting from 0 to whatever space it requires

Address Translation

- A process of converting Logical address to physical address
- MMU uses page table data structure to do so
- Diagram



- Explanation
  - Lets say I have a process that has virtual address space from 0 to 10. That means OS allocated that this particular process can get any virtual/logical address in the range from 0 to 10 (just like we were saying 0 to 16 kb earlier)

Now

↳ Explanation

→ When a program is written and compiled then it gets allocated a virtual address space or some **PAGES** required for its execution. This allocation is done based on programming requirements of that program and management policies by OS. This virtual address space represents the memory addresses that, that particular program can use during its execution.

eg → Virtual Address space of 0—10 allocated for program P1.

Here it has 11 addresses or ~~pagaga~~ pages, from 0—10

(Pagination not used here so no pages etc! Its for understanding)

→ During the execution of program, the logical addresses generated by CPU keeps on changing dynamically based on program execution's needs and memory access pattern.

eg → lets say at first its 5 that means its accessing 5th address from virtual address space (0—10) that has been allocated to it.

After some time lets say its accessing 8th. depending upon execution needs

→ Work of Limit Register

→ Limit register defines maximum addressable range from memory segment for particular process

eg → (0—10) in this case.

→ If 5 generated logical address is beyond this range (which here is not 5<10) then it will throw addressing error

→ work of Relocation Register

→ It contains base address for that particular process

→ **Base Address** → Its the address that's been mapped for that particular Logical address, in the main memory

→ Its the physical address against the logical address

→ Its the starting point/address in main memory for that particular process

→ **Offset** → Its the max distance i-e lets say virtual address space was from 0 — 10 so the offset here is 10
→ Used to calculate the address locations that can be used for a process, in main memory
→ Done

eg Relocation register will now get the base address from main memory lets say its 50

Address locations that can be used = 50 + offset
= 50 + 10
= 60

Range = 50 — 60

And then it will add the logical address into it i-e 50 + 5 = 55 in this case
Now this 55 is the address in main memory where the data gets stored for process p1

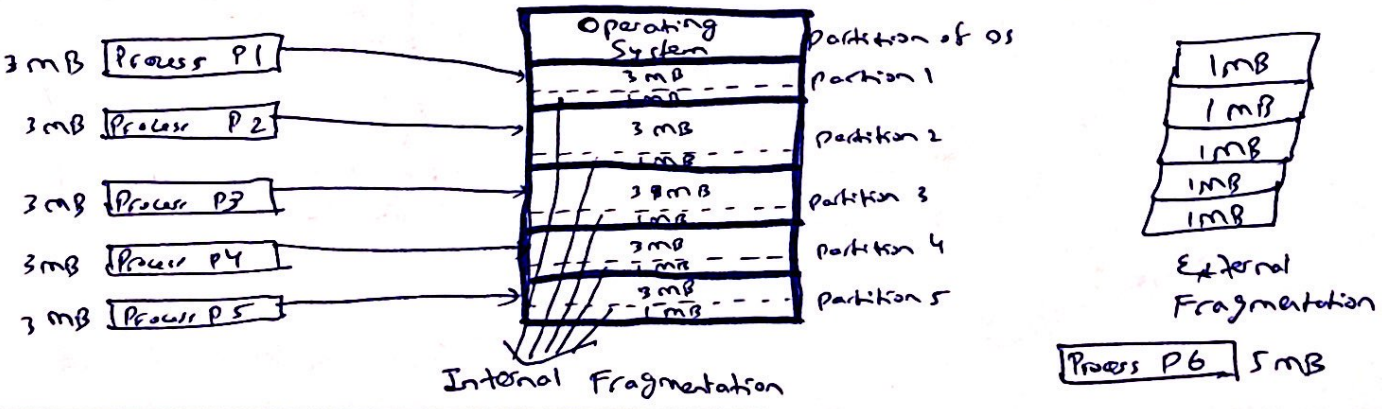
Note
Pages
↳ Chunks of address spaces in virtual or physical memory

*) Allocation methods in Physical/main memory :-

→ Two methods
→ ① Contiguous memory Allocation
→ ② Non-Contiguous memory Allocation

① Contiguous memory Allocation

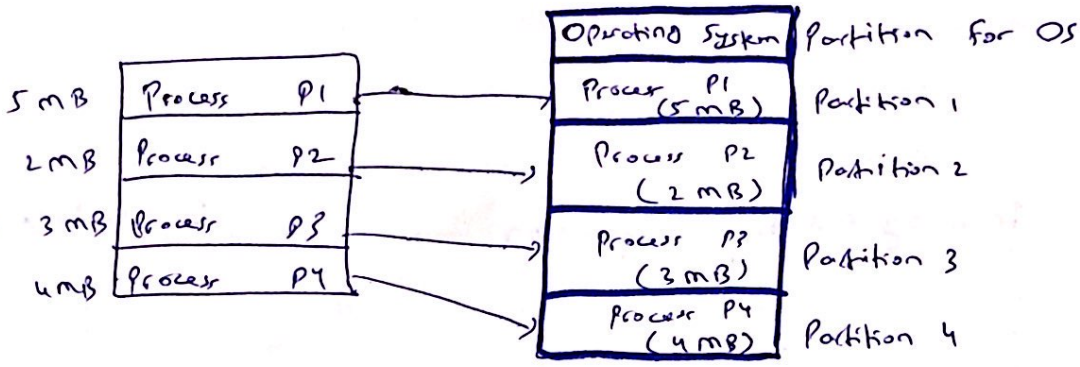
→ Each process is contained in a single contiguous block of memory
→ **Fixed partition** → main memory is divided into partitions of equal sizes



- ~~Explanation~~ **Limitations**
  - Internal Fragmentation
    - If the size of process is lesser than partition size of main memory then some size of partition gets unused. This wastage of memory is called internal fragmentation
  - External Fragmentation
    - Space available but not in contiguous form is sufficient to fulfill another process and execute it but couldn't be able to do so because its non-contiguous
      - eg 1MB + 1 + 1 + 1 + 1 = 5 MB is sufficient to execute process P6 but cant do so because its non-contiguous in main memory
  - Limitation of process size
    - If a process Larger than the size of partition comes in then it ~~cant get processed or executed and~~ cant be loaded into memory
      - eg → if process of 8 MB comes in then it cant get loaded as partitions sizes are of 4 MB
  - Low Degree of multi-programming
    - No. of partitions = No. of processes Not more

Dynamic Partition (Contiguous memory Allocation)

→ Partition size is declared at runtime.



→ Advantages

① No Internal Fragmentation
(bcz Partion size = Process need can be declared

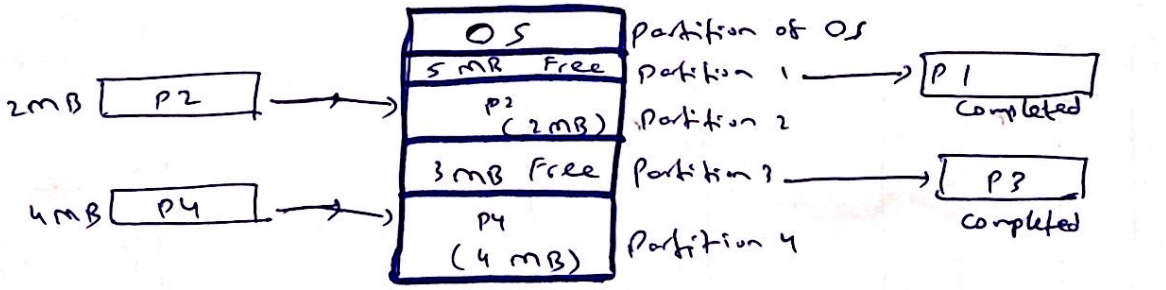
② No limit on size of process

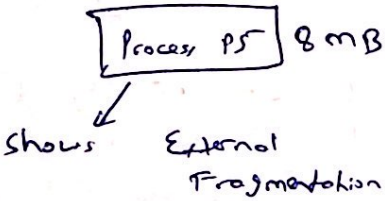
③ Better degree of multiprogramming

→ Limitation

→ External Fragmentation

→ Diagram Explanation





*) Free Space management :-
(Continuous Memory Allocation)

↳ The problem of external fragmentation in dynamic partition is handled using

Defragmentation/Compaction

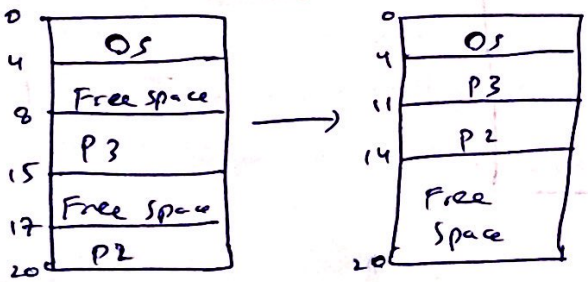
↳ All the free partitions are made contiguous by shifting and all the loaded partitions are brought together.

↳ Free partitions are merged

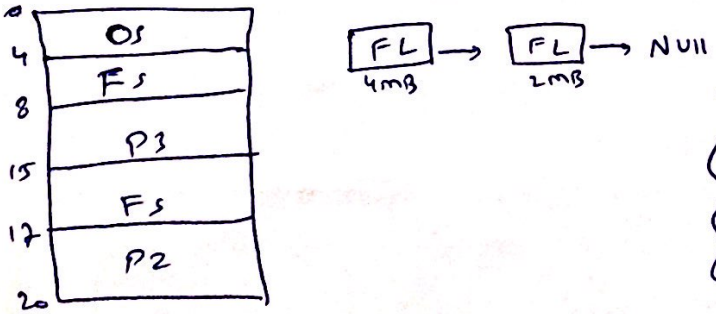
↳ **Drawback**

↳ CPU cycles will get wasted as pure overhead in doing this merge

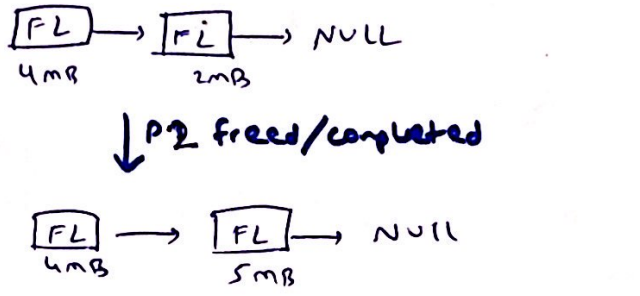
↳ Diagram of Defragmentation (Abstracted view)



↳ In reality it is not like this rather it is implemented using Linked list as (Free List FL)



↳ If contiguous memory gets freed then it gets merged ex if P2 gets executed then it will be like



↳ **How to satisfy request of Process in Free holes**

↳ Lets say we have 4 free holes

50 → 30 → 10 → 5 → Null

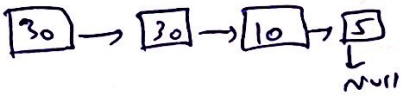
And a process of 20 comes then which hole should be used to fulfill this process

4 mechanisms used for this

↳ ① **First fit**

↳ Allocate the first hole big enough to complete the process eg 50 in this case

↳ Free list will be like

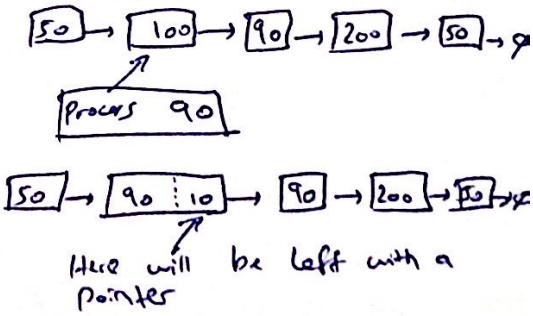


↳ Simple / Easy

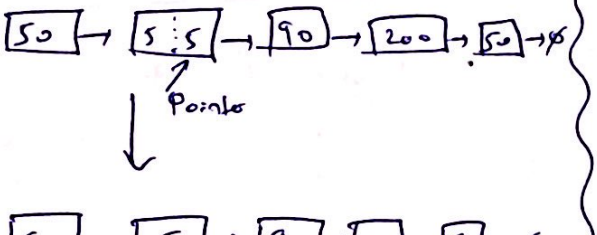
↳ Fast

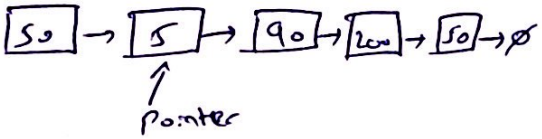
↳ ② **Next Fit**

↳ Lets say we have



↳ Now when next process of 15kb say 5kb comes then it will start searching from the pointer that has been left by previous process





And so on

↳ Same Advantages

① Best fit

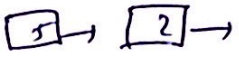
↳ Search for smallest hole in the whole list to fulfil i-e the smallest hole that can fulfill the process

↳ Slow due to iteration of whole list

↳ Can create problem of external fragmentation

Process 85

This process can be executed using 100 but will get executed using 90 here will leave a free hole of 5, Similarly many small free holes will remain at end like



so now if process of 7 comes then can't get executed thus

external fragmentation

↳ lesser Internal Fragmentation

↳ As it found 90 not 100 so less IF.

④ Worst Fit

↳ Search for Largest hole in the whole list to fulfil process

↳ Slow due to same reason

↳ Leave Larger holes thus avoid external fragmentation

*) Paging / Non-Contiguous memory Allocation :-

→ The disadvantage of Dynamic Partitioning is external Fragmentation

↓

Compaction / Defragmentation is used to solve this but with overhead

↓

You can see overhead in 4 ways described

→ We need more dynamic/optimal approach to load processes in partitions

↓

Paging

→ Paging is a memory management scheme that allows Physical address to be non-contiguous.

→ It avoids external fragmentation

→ Idea is to divide physical memory to fixed size blocks called **Frames** and along with it we divide Logical memory into blocks of fixed/same size called **Pages** (# Page size = Frame size)

Note:-
For one program when process gets generated then all processes and subprocesses are divided into pages and physical/main memory is divided into frames.

But when second program gets into execution then it's page sizes of processes and subprocesses may differ with page size of processes of program 1
And frame size may also differ but for each program their own
(page sizes = frame sizes)

→ **Page Table**

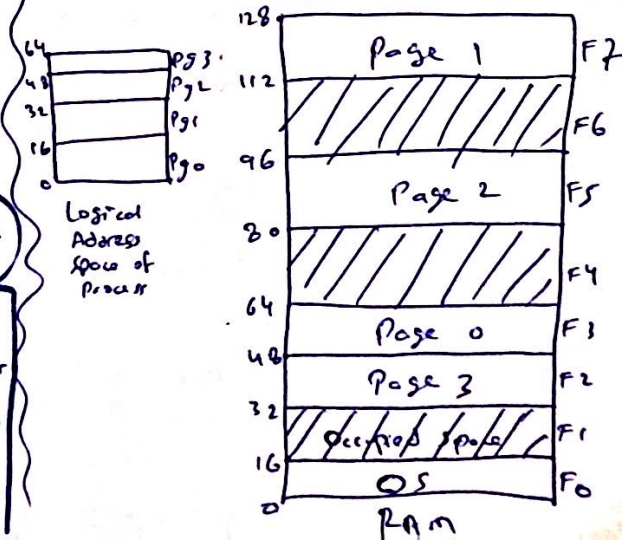
→ A data structure that shows which page is mapped to which frame in physical memory

→ It contains base address of each page in physical memory

→

Logical pg no	Physical frame no
0	3
1	7
2	5
3	2

for the following



→ Here the logical Address space of Process P0 is 64 byte

↓

To uniquely identify 64 byte we need how many bits?

↓

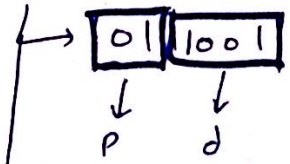
$\log_2(64) = 6$ bits means we need 6 bits to represent each PID that means lets say 25th byte of logical address space of P0 will be represented using 6 bits as 25 to binary with 6 bit length is

011001

→ So lets say 25th byte from Logical Address needs to get translated to physical Address

Address Translation from Logical to Physical

→ Step 1:- Convert it into bits of required length to uniquely identify it. In this case to 6 bits



→ p → page number

d → page offset

These are the logical address divisions

→ Step 2:- $(01)_2$ is binary representation of $(1)_{10}$. One (1)

Now go to page table and get the value of frame number against 1. In this case its 7

→ Step 3:- Now it will go to physical/main memory to frame 7. In this case frame 7 starts with 112 byte.

Here page offset (d) was $(1001)_2 \longrightarrow (9)_{10}$

so now.

$112 + 9 = 121$

So, 121st is the byte where the data of 25th byte of logical address of process P0 lies in main memory.

→ **Address Translation from physical to Logical address space**

→ Step 1:- We have 121 byte from physical address. we first have to know how many unique ways are these to represent whole RAM.

In this case Main/memory/RAM is of 128 byte so

$\log_2(128) = 7$

So each entry of Main memory must be represented with 7 bits. eg 0th byte will be represented as 0000000 and so on

→ Step 2:- Convert 121 to req Length bits

d (Page offset)

They tells frame no 7 i.e $(111)_2 \rightarrow (7)_{10}$

→ Page Table can be used to find page table against frame number. In this case its 1

→ Step 3:

Go to 1 page and add offset d so $16 + 9 = 25$

How Paging Avoids External Fragmentation:-

→ By doing non-contiguous memory allocations for each table of single process

→ In OS, In kernel the PCB is stored against each process.
Besides other things of PCB like Proces ID, registers, current state etc
It also stores the address of page table.
So, page table created by OS is present in memory and its address is present in PCB, ~~if there a register called Page Table base register~~ (PTBR).

→ During Context Switching

→ **Page Table Base Register (PTBR)**

→ PTBR is a special register present in CPU or MMU.

→ It contains the base address of page table of current process in execution

→ When context switching occurs then PTBR gets the value of new base address of page table of incoming process from PCB of incoming process

Why Paging is slow?

→ Paging is slow bc2 Address translation from Logical to physical address has to go to OS to get memory ref of page table

→ This is additional step as compared to Address translation traditional process

→ So many memory ref present

How to make paging fast?

→ Using **Transition Look-aside Buffer (TLB)**

↓

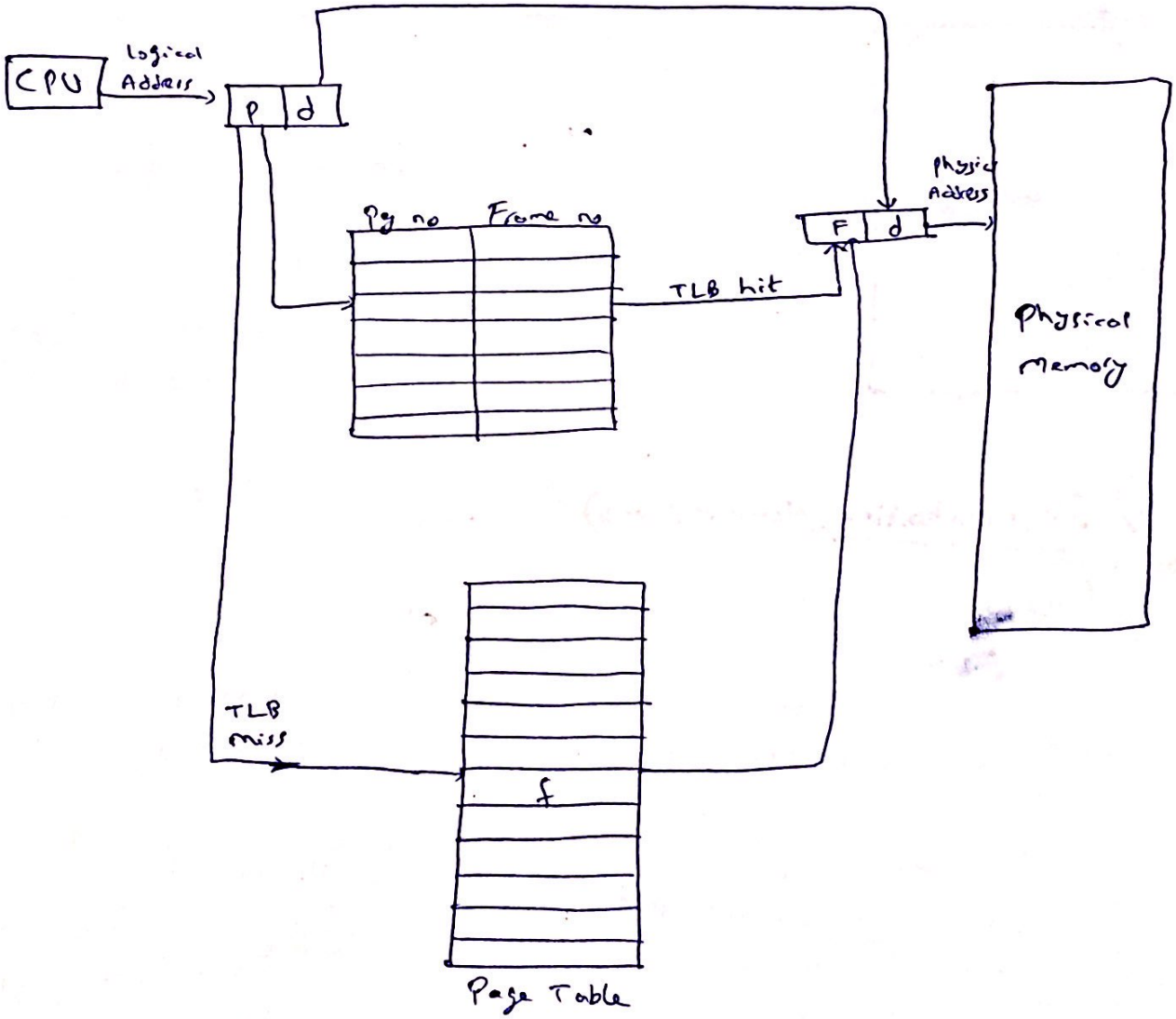
→ It's kind of a caching process that stores page table & frame ref after FIRST NEED OF THAT PAGE NUMBER

So second time if frame no. of same page number is needed its fast

↳ Paging Hardware with TLB

Note

TLB is a hardware, present in MMU
It has key (Pg no) and value (Frame no.



Explanation

↳ First time when CPU generates P and d then it checks TLB if any frame is present against Pg no. P. But since its first time so it can't finds that P in TLB so it goes to page table and gets frame against that P and process goes on and it also puts P and frame against it in TLB so next time when it searches it finds so process fast.

↳ **Problem here** → Each process has its ~~own~~ own page table so it has its life if context switching occurs then

Page table gets changed but TLB has the entries from previous page table this can cause issue

↳ Solution

↳ ① Flush TLB when context switching

↳ ② Add Address Space Identifier (ASID) into the TLB so TLB will look like

ASID	Pg no	Frame no
Process1	01	7
Process2	01	6

So now two entries with 01 are present but system will also check that it gets the frame no of the process in execution

Note

Paging has internal fragmentation. (Look at end of segmentation lecture)

Paging does not have/resolve external fragmentation

*) Segmentation (Non-contiguous)

↳ Problem in Paging

↓

eg there's one single function func1()

now paging will lets say make 2 pages of this single process func1() and it will have non-contiguous address allocations (2 addresses allocated for 2 pages)

Now when one page will finish executing then mmu will again do all address translations to get address of second page in main memory. This address translation second time although the process was single function

creates overhead and cpu cycle waste.

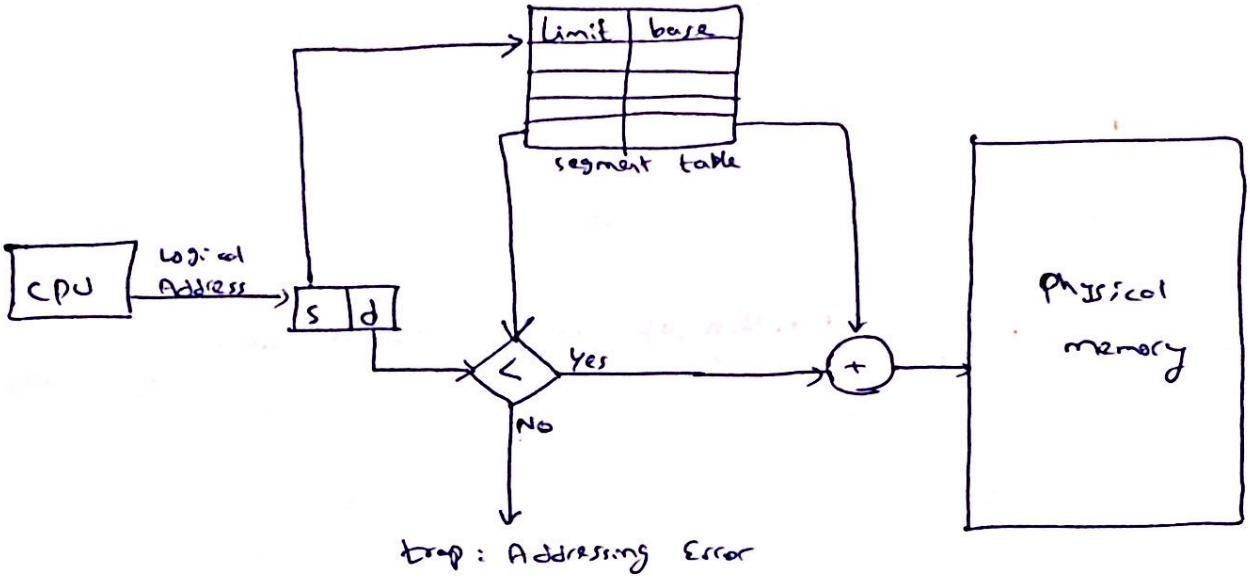
↳ Solution

↳ Segmentation solves this by creating segments

↓

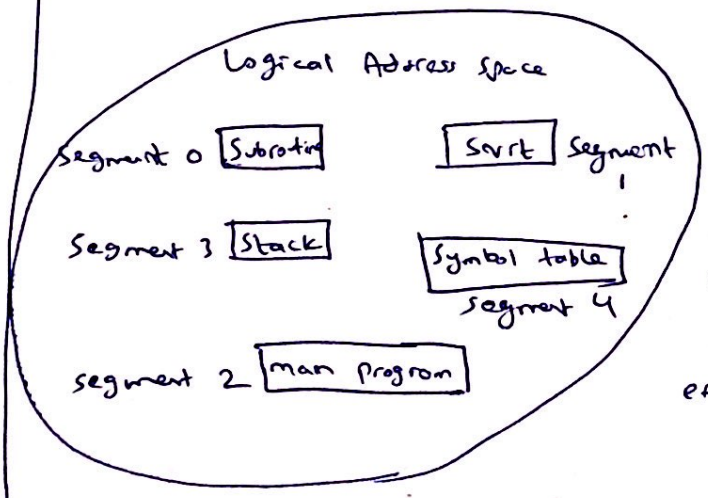
Unlike Paging where pages were of same size, segments are of variable sizes

Address Translation from Logical to Physical Address



Explanation

- Limit acts as Limit register of continuous memory allocation
- base act as relocation register of continuous memory allocation.
- Example of segments



Segment Table

	Limit	Base
4	1000	1400
3	400	6300
2	400	4300
1	1100	3200
0	1000	4700

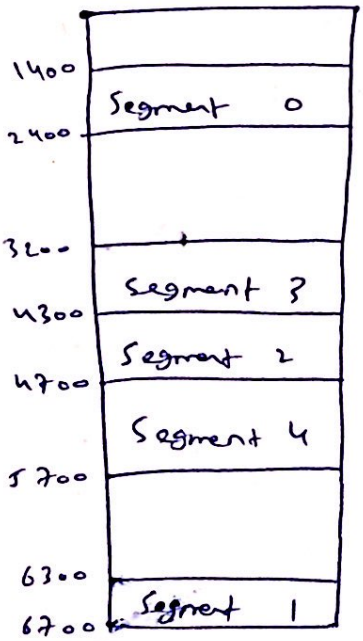
example → segment 2

↓

4300 + 53

= 4353

Logical Address generated is 53 which falls under limit 400 in entry 2 at segment table for segment 2



Advantages of Segmentation

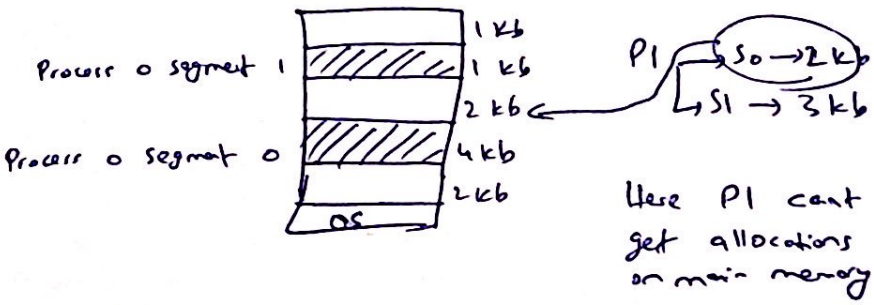
1. No Internal Fragmentation
   - Bcz whatever is the size of segment, that much size gets allocated to that segment in main memory so no space is waste due to variable sizes of segments

Problem of Paging ⟶ Solution

2. Each whole segment gets a continuous allocation, so like whole func1() will be one segment and will be allocated main memory so no overhead within one segment during address translation so more efficient.
3. Size of Segment Table < Size of Page table
   - Bcz a process of 100 kb may has 50 pages each of 2 kb but it will have lesser segments maybe 3, 4 depends. So this is fact
4. More Efficient
   - Compiler keeps same type of functions in one segment

Disadvantages of Segmentation

1. External Fragmentation
   - Since main memory will have variable spaces allocated so chances are like



Here P1 cant get allocations on main memory

2. Different size of segments ⟶ not good at time of swapping
   - Large size segment may need more time to swap-in swap-out to backing storage

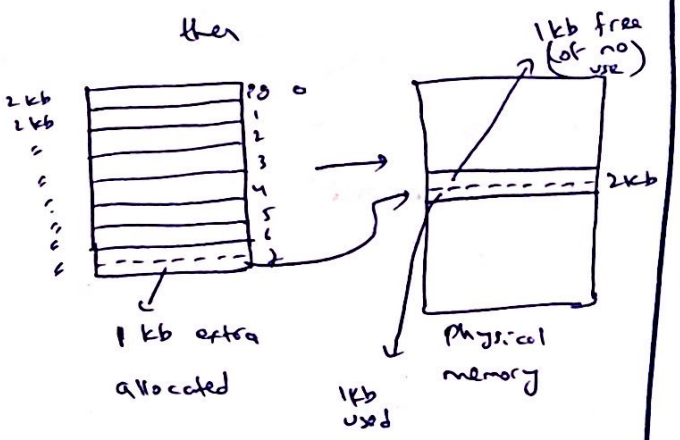
Why Internel fragmentation is said to be in Paging?

↳ Paging has internal fragmentation as eg

if Process P1 ⟶ 15 kb

page size ⟶ 2 kb

then



So internal fragmentation is presents

Modern systems architecture Provides both segmentation + Paging implementation in a hybrid manner

★ Virtual memory | Demand Paging

Page Faults :-

↳ Virtual memory is a technique that allows execution of a processes that are not completely in memory.

Creates an illusion of big main memory.

Swap-Space is used

↳ Part of secondary memory used as main memory.

→ Advantage of VM:-

↳ Programs larger than physical memory can run

eg GTA 5 lets say needs 40GB of RAM to run but it is run on 16GB of RAM by using 24 GB of Hard disk as swap-space.

Virtual memory = RAM + Swap Space

↳ Number of processes wanting to get into main memory and load but not all pages of each processes are needed every instance of time in main memory. Only some are required. The required ones are loaded into main memory and rest are pushed to swap space.

Then whenever a need arise the page from swap space gets replaced with pages in main memory.

↓

more programs can be run so CPU utilization is good and throughput.

Throughput is rate at which system can transfer data or process within a given period of time

Since more processes can be loaded in main memory simultaneously. This minimizes the amount of time spent in waiting by the processes. Thus increasing the throughput.

Demand Paging

- Method of virtual memory management
- In this process, the pages which will be least of use will get loaded to swap space and when a demand of that arises i-e when that particular page is demanded for execution in main memory then an interrupt called **Page fault** occurs. So this interrupt will go to OS and OS will use **page replacement algo.** to replace the page from swap space to main memory

Lazy Swapper

- Lazy swapper is used to swap the pages from disk (swap-space) to main memory on page fault
- Instead of "swapper" we use word "Pager".

How demand paging works?

- when process is to be swapped in to main memory then pager guesses which pages to be drawn to memory and which will not be used in near time. Pager then brings those to main memory

Valid-invalid bit scheme

- Used to determine which page in disk, which in memory
  - 0 ⟶ Page is either not valid or its currently on disk (swap-space)
  - 1 ⟶ Page is already in main memory
- If process never attempts to access same invalid bit page then execution gets completed smoothly
- If process tries to access a page not present in main memory

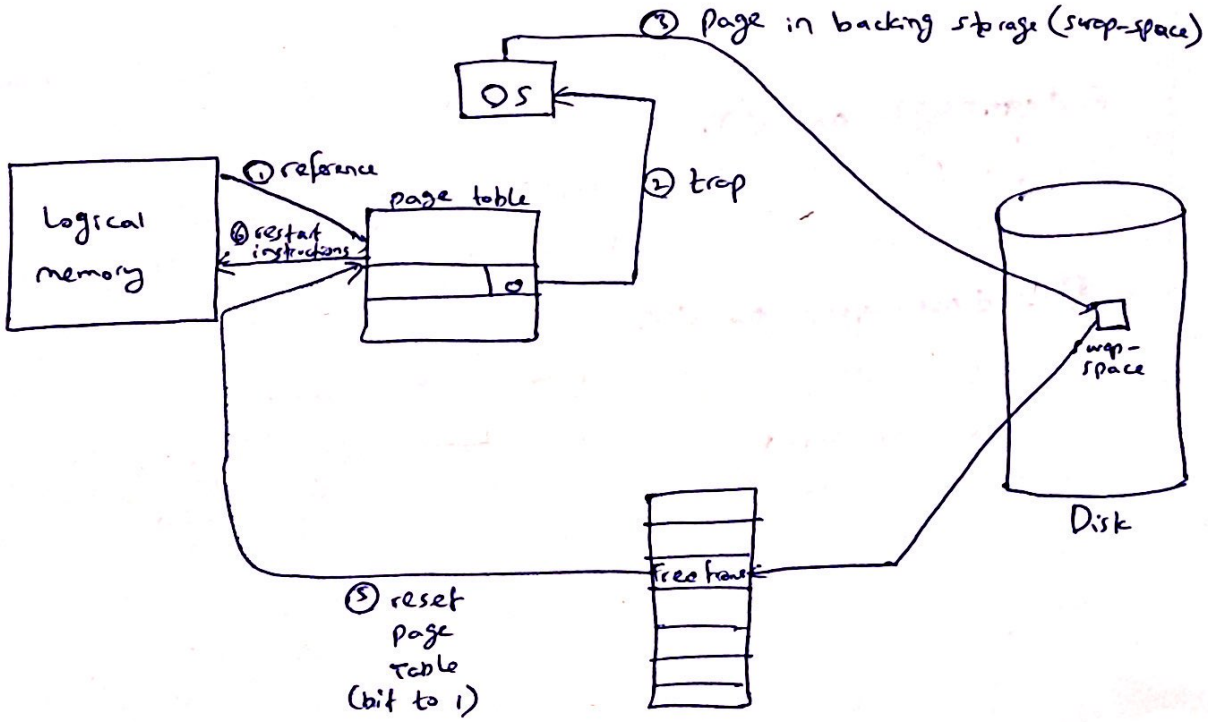
↓

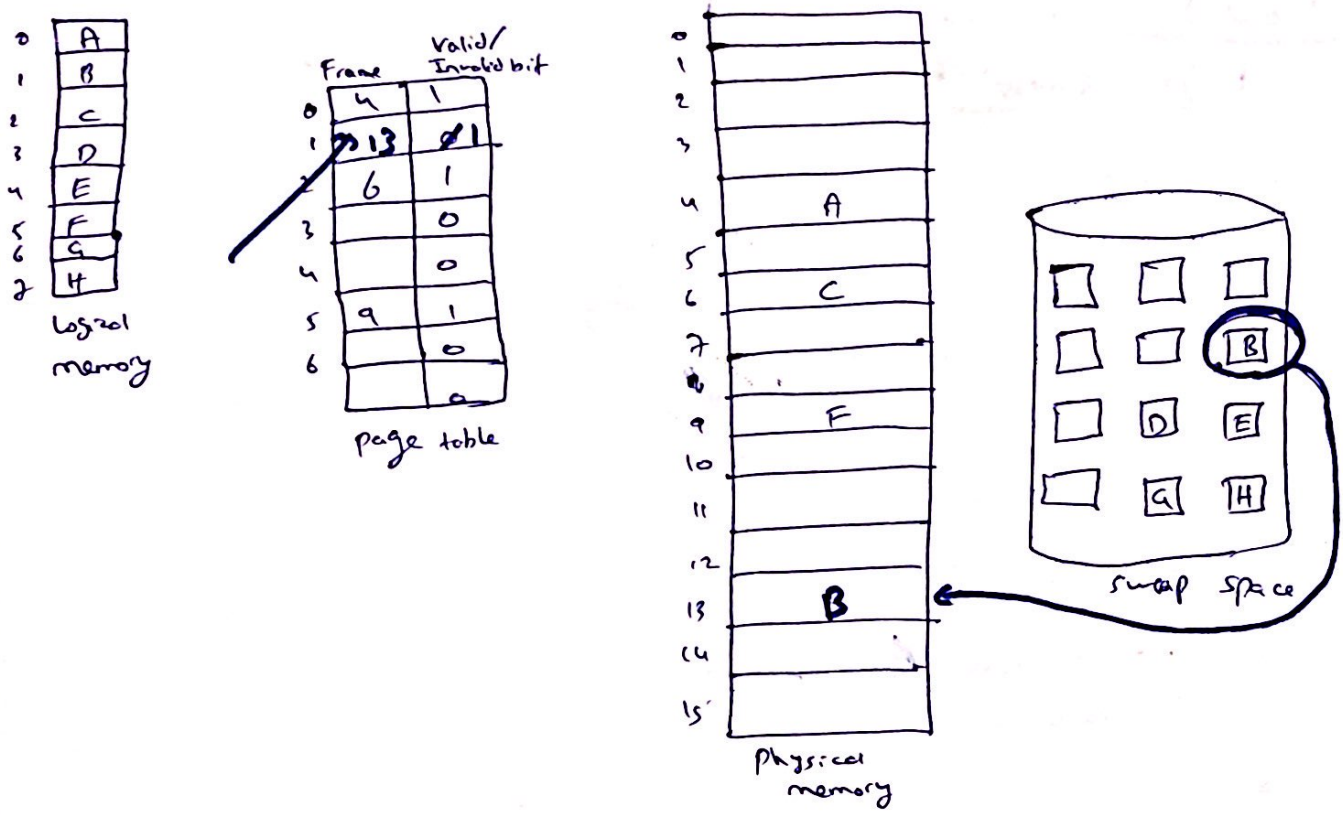
Page fault occurs. Paging hardware noticing invalid bit for demanded page will cause a trap to OS.

↓

Procedure to handle Page fault :-

- Checking PCB to check if reference generated falls under the range of virtual space address
  - if no → trap & exception throw
  - if Yes → Swap the page
- Find a free frame (from free-frame List)
- Schedule disk operation to read desired page into newly allocated frame
- Modify the page table
  - Make bit set to 1
- Restart the instructions that was interrupted by trap. Now second time they will not fall in trap.





Pure Demanding Page

- Putting whole process's pages to swap space
- Now, as the need arises then pages will get swapped to main memory
- Never bring page to memory until its required.

Advantage of VM

- Degree of multiprogramming increased
- User can run large apps with less physical memory

Disadvantages of VM

- Swapping takes time
  - System slower
  - Called "Page fault service time"
- Thrashing

*) Page Replacement Algo :-

→ If all frames of main memory are occupied then if page demand occurs then which page from main memory is to be swapped with demanding page is decided by Page replacement Algorithms.

→ **Aim** → To have minimum page faults

→ **Types of Algorithms**

↳ ① **FIFO Algorithm**

→ Replace the oldest page in main memory.

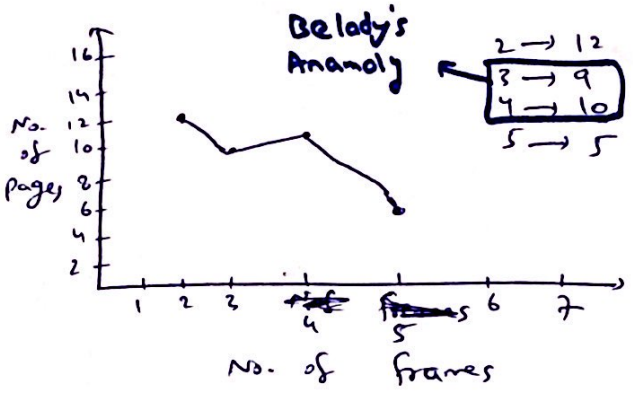
→ Easy to implement

→ Performance good if page replaced is needed after very long time (eg initialization page)

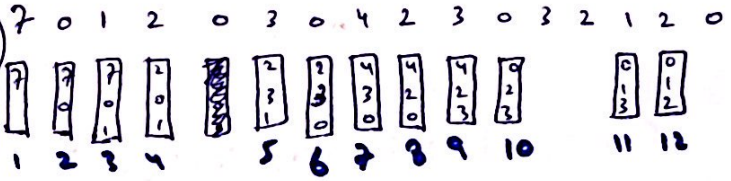
Performance bad if page just replaced is needed again

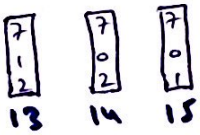
→ **Belady's Anamoly** present

↳ In case of LRU (least recently used) if no. of frames are increased then no. of page faults gets reduced but in FIFO sometimes the opposite of it happens. Its called Belady's Anamoly



→ Example





→ 15 page faults

→ Just replace the incoming page with oldest page (LRU) if not exist in memory. Here frame size is 3 i.e max frames can come in main memory are 3

② **OPR (Optimal Page Replacement)**

→ Find a page that will never be referenced in future and replace with that

→ Find a page that will be referenced Farthest in future and replace with that

↳ OPR has lowest Page fault rate

↳ Impossible to implement

↳ Bcz cant know earlier which page will be used when exactly

↳ As we couldn't know in Shortest Job First Algo that which process has least burst time we did that on assumptions basis

③ Least Recently Used (LRU) Algo

↳ Past is seen

↳ Replace the page with the page that has not been used for longest period

↳ can be implemented using

↳ ① Counter variable

↳ Keep counter of incoming variables and whose counter is less is the LRU page

↳ ② Stack

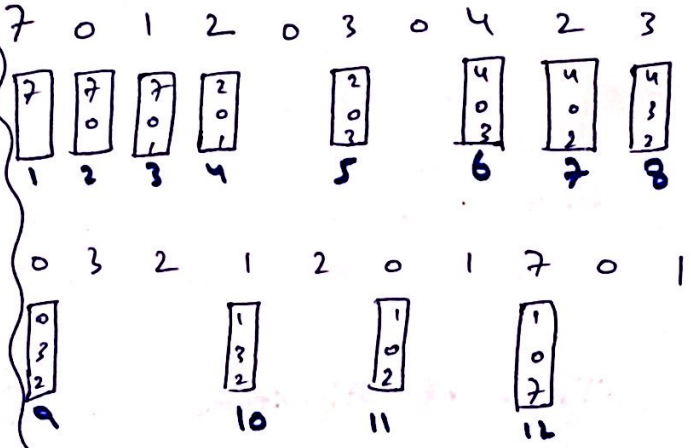
↳ Implemented using Doubly Linked List

↳ Keep a stack of pg no.

↳ whenever page is referenced it is removed from stack and put on top

↳ So, most recently used will always be on Top and LRU will be on bottom

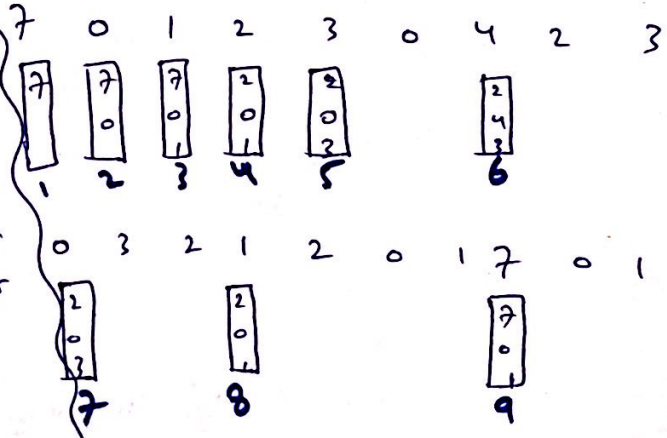
↳ Example



→ 12 page faults

→ whoever is the oldest and LRU will get replaced

Example of OPR



→ 9 page faults

→ whoever is farthest used will get replaced

① Counting based Page replacement Algo:-

- Whichever pages are frequently used counting are increased of them
- Types
  - ① Least Frequently used
    - Actively used pages will have large ref count
    - Replace page with smallest count
  - ② Most Frequently Used
    - Replace page with largest count
    - Opposite of LFU
    - Actively used pages will have lesser count

⋆ Thrashing :-

Situation 1

RAM 1
P_1 (1 Page)
P_1 (2nd Page)
P_1 (3rd Page)
P_2 (1st Page)
P_2 (2nd Page)
P_2 (3rd Page)

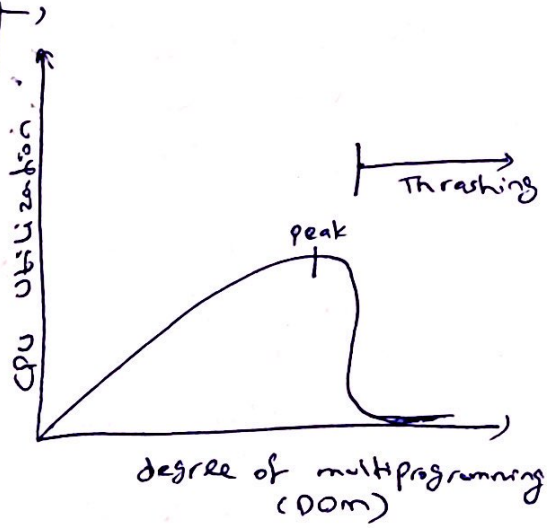
Situation 2

RAM 2
P_1 (1st Page)
P_2 (1st Page)
P_3 (1st Page)
P_4 (1st Page)
P_5 (1st Page)
P_6 (1st Page)

More no. of page faults → Situation 2

→ In situation 1, where the Locality of Process 1 is more then it will first handle 1st page then 2nd then 3rd and then if it has 4th then it will request for it, make a page fault and then from swap space 4th page will come and now whom will the 4th page will get swapped with? It has 3 free frames (1st) (2nd) (3rd) whom it can swap with.
Yes the degree of multiprogramming is low here as at a time only 2 process P1, P2 exist in RAM but page faults are less and will come after a good time.

- In situation 2, where Locality of each process is low and degree of multiprogramming is high, whenever one page processing is done then page fault will be sent for its second page so more page faults leads to less CPU utilization.
  - This high paging activity (handling high page faults, swapping and stuff) is called Thrashing



If we keep on increasing DOM then CPU utilization will keep on increasing as more no. of processes now exist in RAM but at some point if excessive no. of processes enters RAM then Locality will be less for each process so more page faults will occur and CPU utilization will get drastically reduced.

- Techniques to handle Thrashing
  - ① Working Set Model
    - Based on Locality concept
    - One process gets allocated enough frames to built good locality in RAM so less page faults
    - Each process has its size of locality eg 2, 3, 4 frames i.e pages. If allocated frames are lesser than size of locality then thrashing will occur
  - ② Page fault Frequency
    - Establish upper bound lower bound
    - if pg fault rate is above upper bound
      - allocate some frames to some Locality of that process
    - if pg fault rate is below lower limit
      - Remove a frame from the process