# Operating System

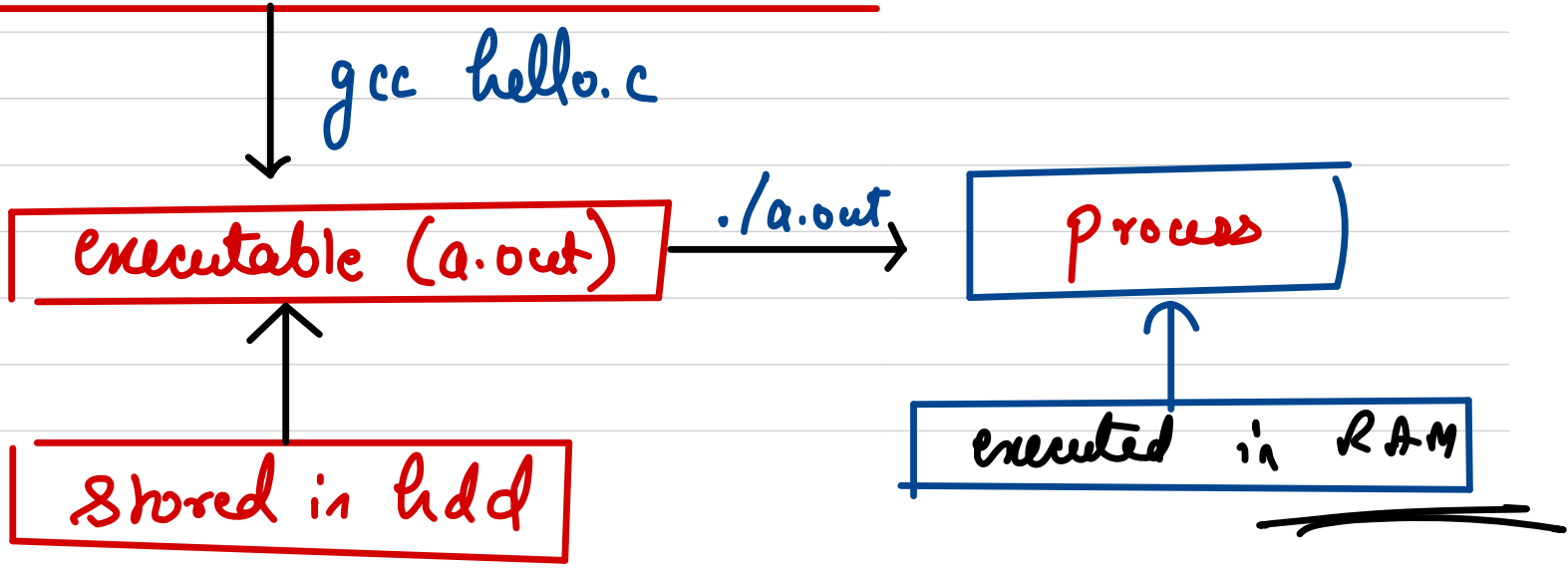
## * Memory Management →

Let's say we've a C program

```
#include <stdio.h>
int main() {
    char str[] = "hello";
    printf("%os", str);
}
```

```c
#include <stdio.h>
int main() {
    char str[] = "hello";
    printf("%s", str);
}
```

→ what happens when we compile and run a C program.



# Process →

→ A program in execution is called process.

→ Process is present inside RAM

→ few components of process are -

* Executable instructions
* Call stack
* Heap
* State of the process in OS

State : → list of open files, related process, registers data etc

all of this is managed by OS

What is an operating system? → software

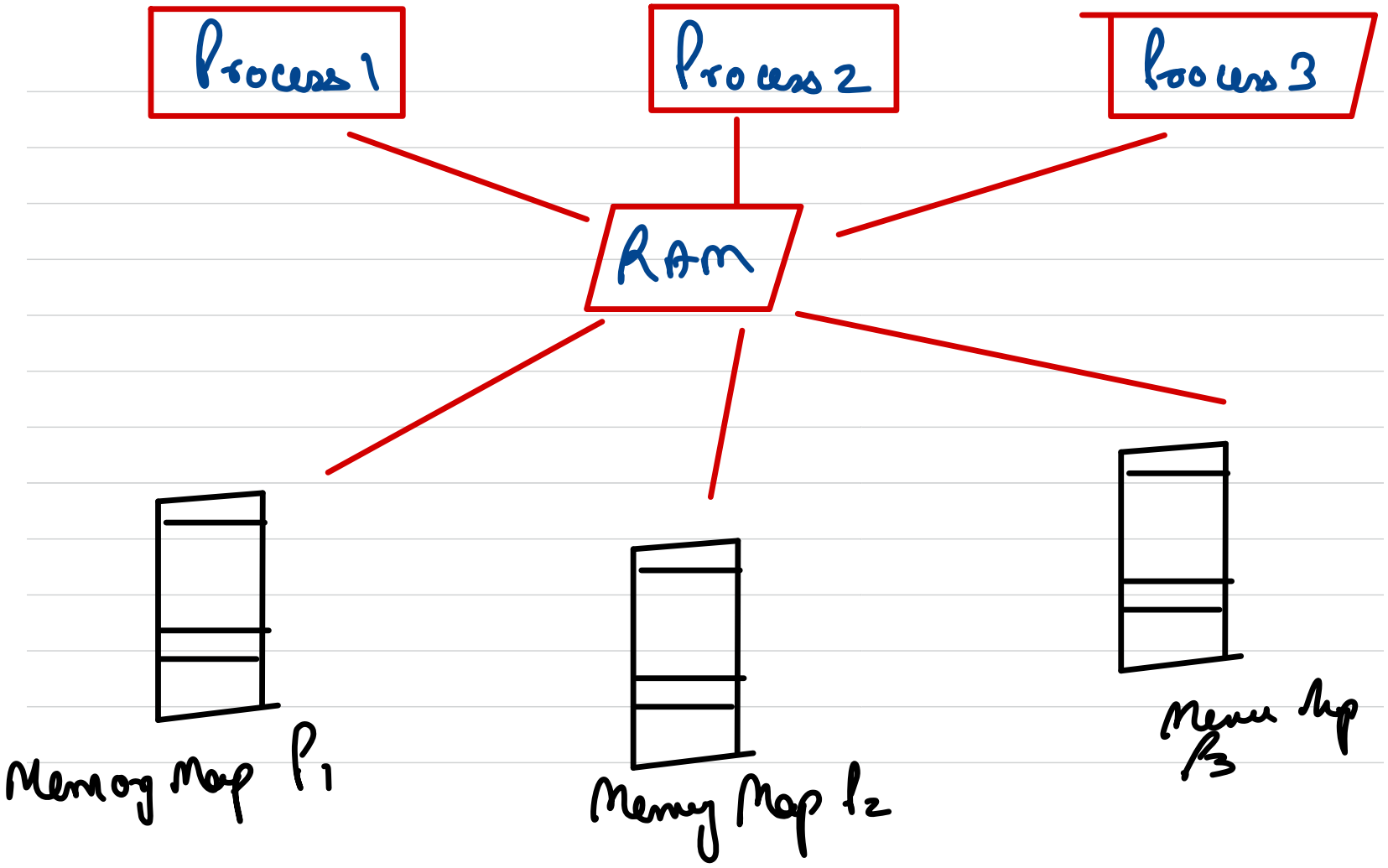
It is an intermediary between end user and computer hardware. It provides suitable environment to create and execute programs.

→ Resource Management
→ Process Management
→ memory & Storage Management
→ Security

## How RAM manages process?

We can have multiple process running simultaneously and sharing RAM. But Ram is a limited process.

Now, irrespective of the fact on how multiple process are managed by OS, their memory map should always be present in RAM.


Process 1
Process 2
Process 3
RAM
Memory Map P1
Memory Map P2
Memory Map P3

Let's see how RAM manages multiple process -

There are multiple modes of RAM management

(1) Single Continuous model -

→ No sharing

→ at one time only one process occupies RAM

→ when one process is completed then only other process starts

→ limitation

→ multiple process management is not very good.

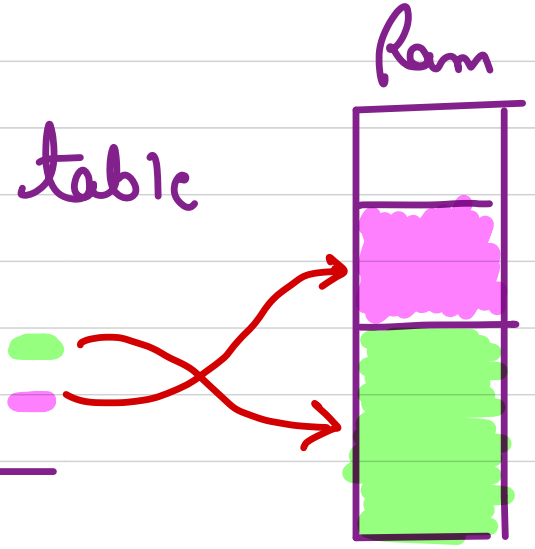
(VVI) → process memory size is restricted by RAM size..

## ② Partition Model

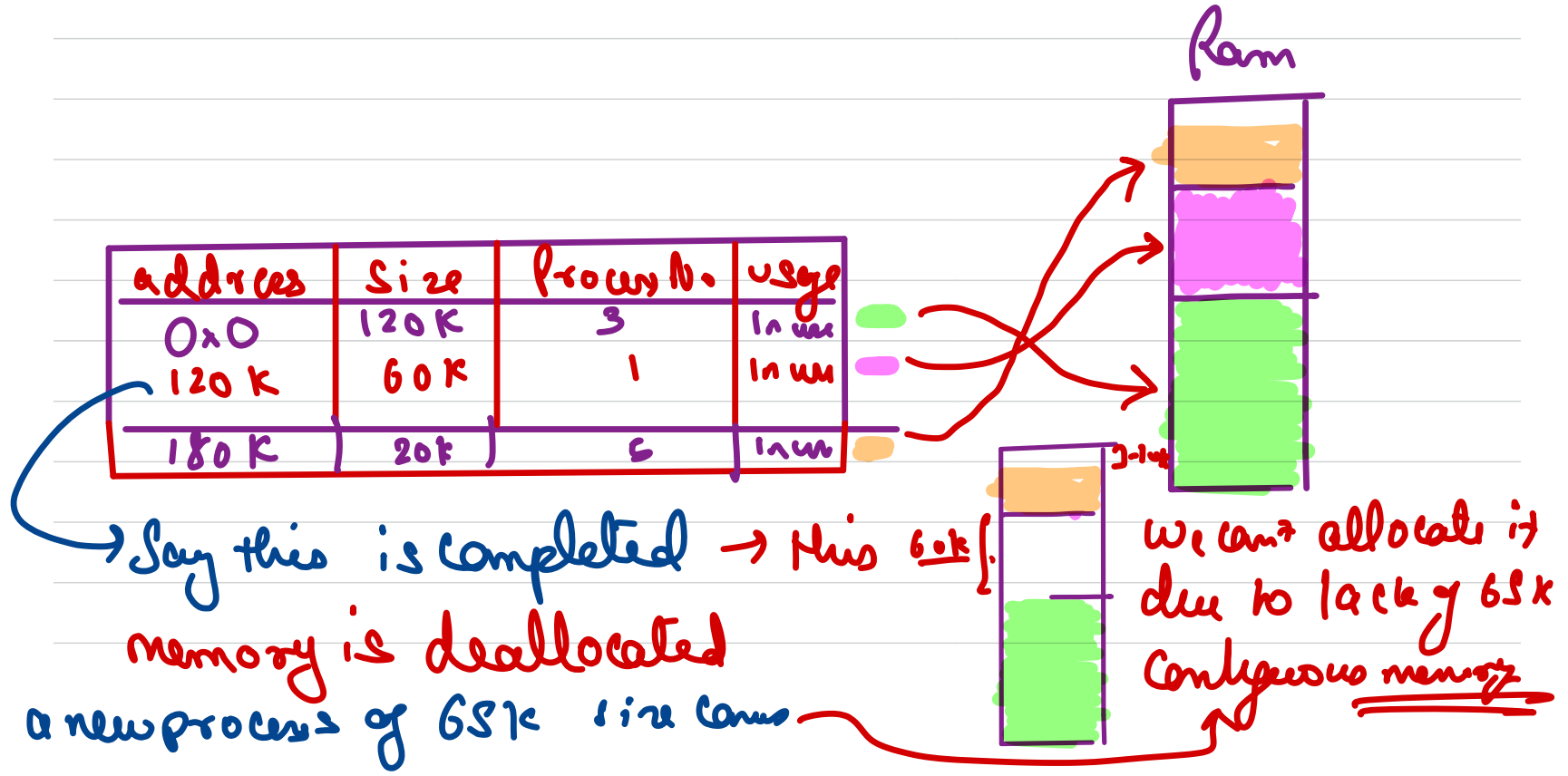
as long as sufficient contiguous space is available new process are allocated in the memory:

It maintains a partition table

| address | Size | Process No. | usage |
|---|---|---|---|
| 0x0 | 120K | 3 | In use |
| 120K | 60K | 1 | In use |



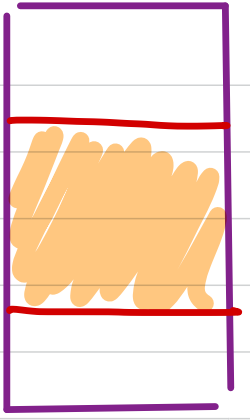
Let's say we get a new process of size 20k



This leads to a phenomenon called as fragmentation

fragmentation is a situation when free block are too small to satisfy an memory request. The small left over memory is getting wasted and if this small chunks inc, then amount of reusable memory decreases.

* Internal
* External



# How partition model handles fragmentation ??

① First fit → It may make fragmentation worse.

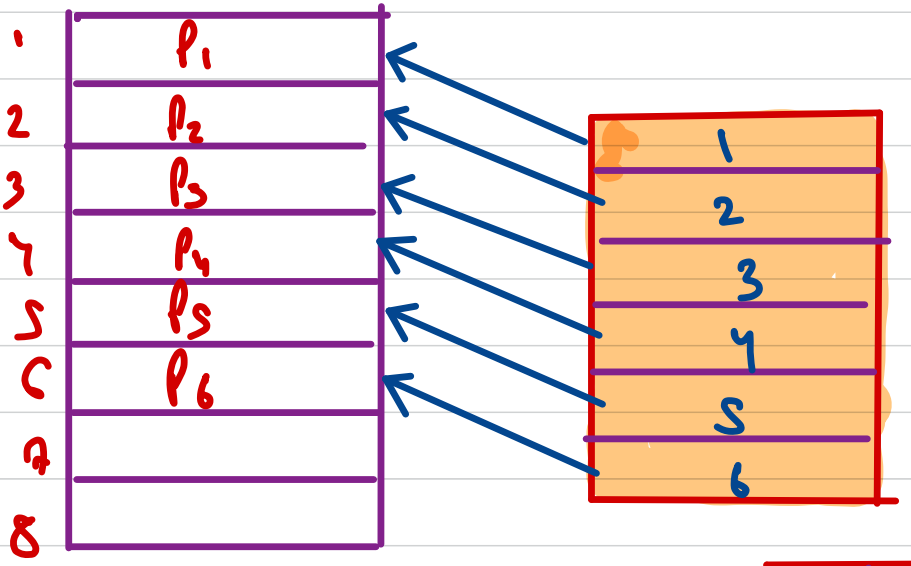
② Best fit → this may affect performance.

during deallocation of process we can merge free partitions, but it will take time.

Current OS does better job in memory management using virtual memory & Segmentation

# Virtual Memory

Ram is divided / split into fixed size partitions called as Page frames. and in abl processors it was typically about 4KB.




RAM

Process also splits into blocks of equal size when Block :: page frame size

| block | page frame |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

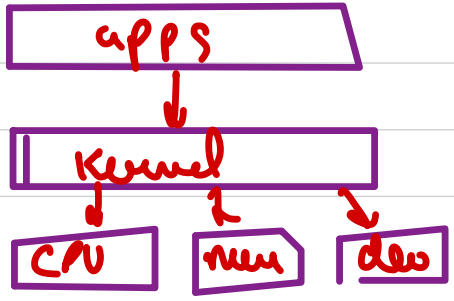
Page table

Then process block is allocated to frames

Because of per process page table, blocks of process need to be allocated in continuous frames. The page frames can be identified by page table.

So every memory access has an additional overhead of the lookup in the page table. This can be eliminated by partly a cache called as TLB (Translational lookaside buffer)

★ KERNEL → It is a central component of OS that manages operations of computer & hardware. It is a bridge between applications & data processing performed at hardware. using inter-process communication calls & system calls.



* Note → Memory associated with the process is in the user region of memory whereas the process page table is in the kernel region.

Depending on active process, the active page table will vary. So a process won't be able to access page frames of other process.

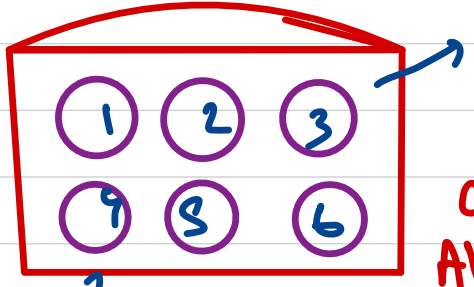
Q) Do we really need to load all blocks into memory before process starts executing ??

No

Not all parts of the program are accessed simultaneously. Infact some code may never be executed.

So this concept is called as Demand Paging.

Virtual memory is a feature of OS, that enables computer to be able to compensate shortage of physical mem. By transferring pages of data from RAM to Disk storage.




Swap Space on Disk

In our secondary storage, a space is allocated (swap space) All blocks of mem is present On demand blocks will be loaded in the ram.

| block | frame | present bit |
|---|---|---|
| 1 | 14 | 1 |
| 2 | - | 0 |
| 3 | - | 0 |
| 4 | - | 0 |
| 5 | - | 0 |
| 6 | 3 | 1 |

* Pages are loaded from disk to ram, only when needed.

* A present bit in table represents if block is in RAM or not.

Say, block 6 of executing process wants to access block 3 & block 3 is not loaded in RAM, then it will check if block 3 is loaded or not using present bit. If the bit is 0, processor issues a page fault interrupt, that triggers the OS to load the page.

#Situation → Say OS wants to load a block in a page frame, but no page frames are free for new block to be allocated. Then what happens??

# Page replacement policies

Note → Page table's for the processes under consideration are updated during page replacement

Few policies are-

1. First In First Out
2. Least recently used
3. Least frequently used

Based on the policy implemented in the OS, the OS swaps out a block from memory with a new block to be added in RAM

Process of loading a block in RAM is called Swap In

Process of pulling out a block from RAM is called Swap Out

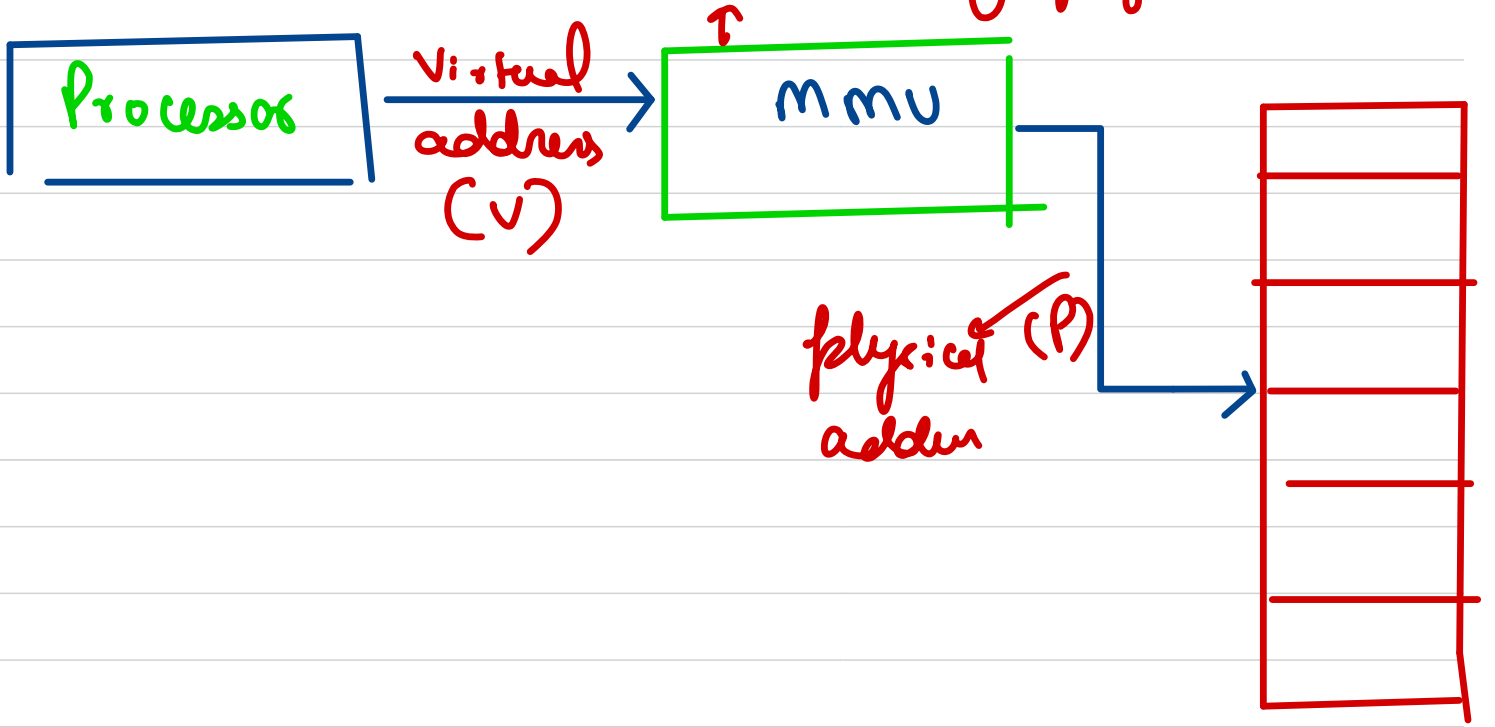
* Swap out process → During the swap out process the changes in the content of block to be swapped from RAM is copied to the disk so the disk has latest piece of changes. But if no changes has been done then no need to do a copy. To maintain this a dirty bit is used.

# Virtual address space of a process.

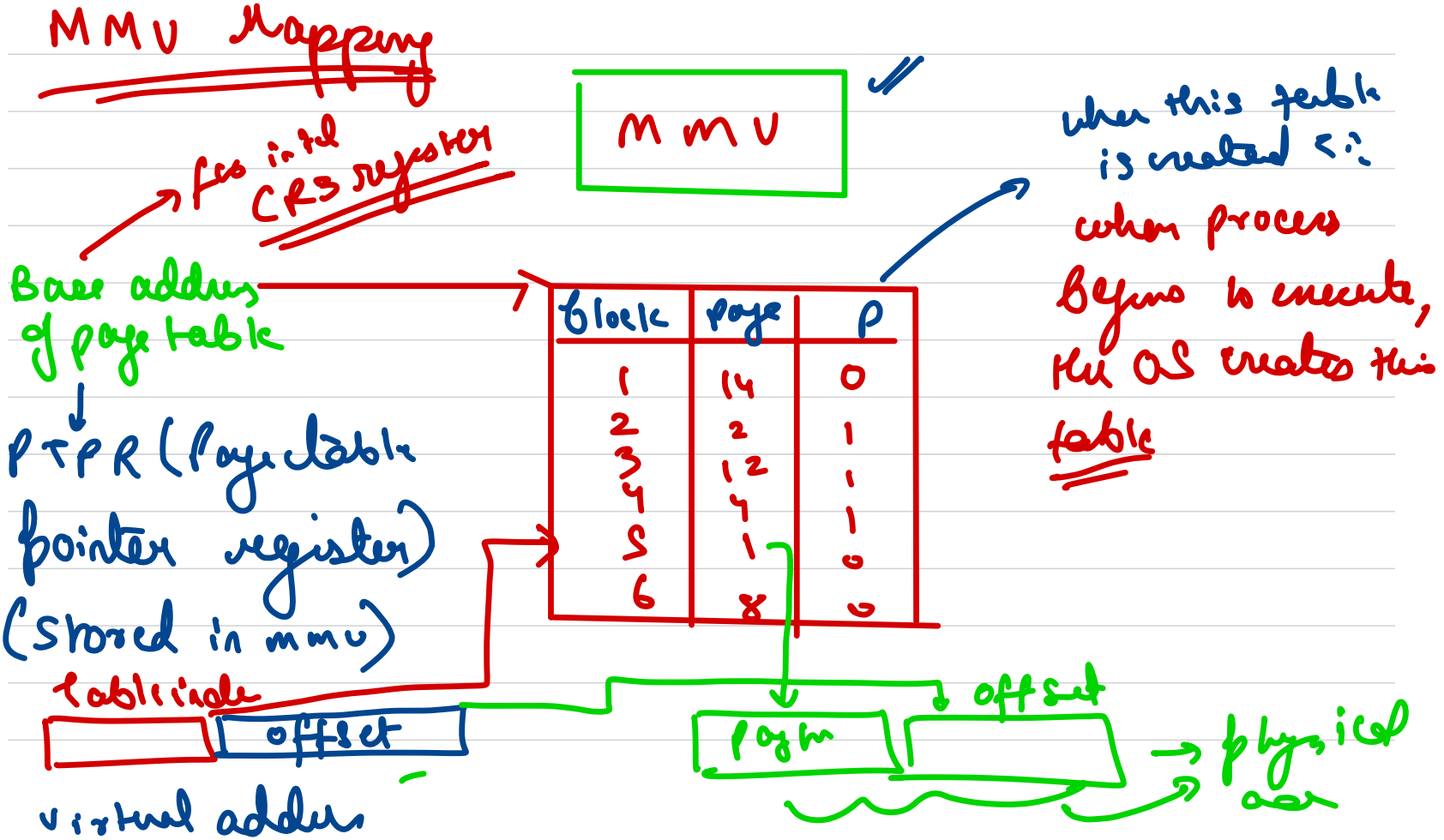
↳ is not the actual physical address

To access process memory the processor generates a corresponding virtual address.

mmu maps virtual address to the corresponding physical addres



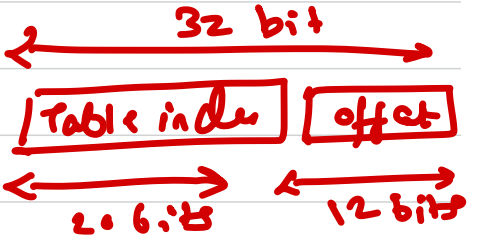
# MMU Mapping


MMU
Base address of page table
PTPR (Page table pointer register) (stored in mmu)
fed in the CR3 register
when this table is created ?:
when process begins to execute, the OS creates this table
Block
Page
P
1 14 0
2 2 1
3 12 1
4 4 1
5 1 0
6 8 0
Table index
offset
virtual address
page
offset
physical

# MMU mapping for 32 bit System

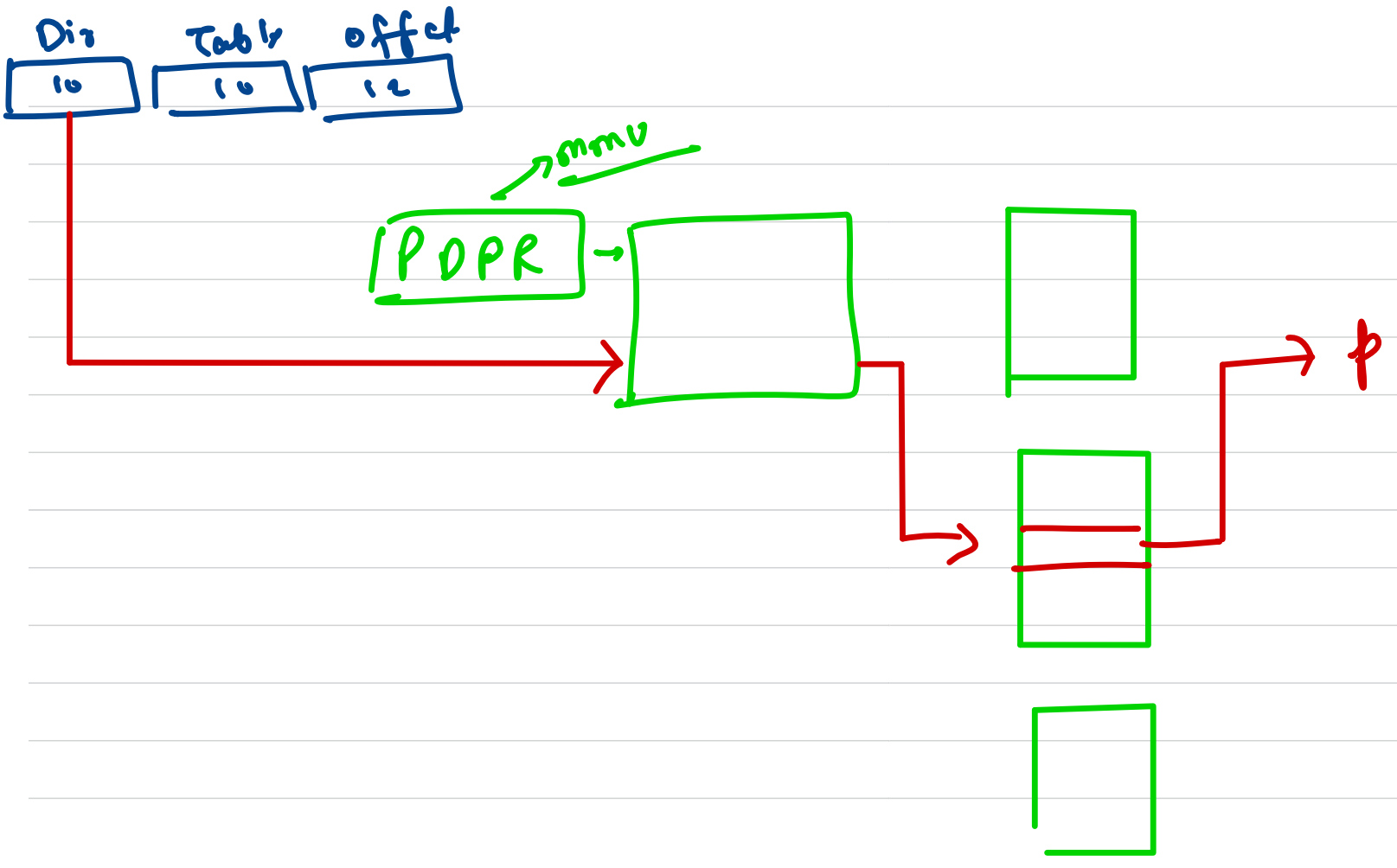
V address → 32 bit

Process size → $2^{32} \approx$ 4gb




if table index is of 20 bits then page table can have $2^{20}$ entries ≈ 4Mb (& this 4mb is reqd in contiguous order)

Enter 2 level page translation

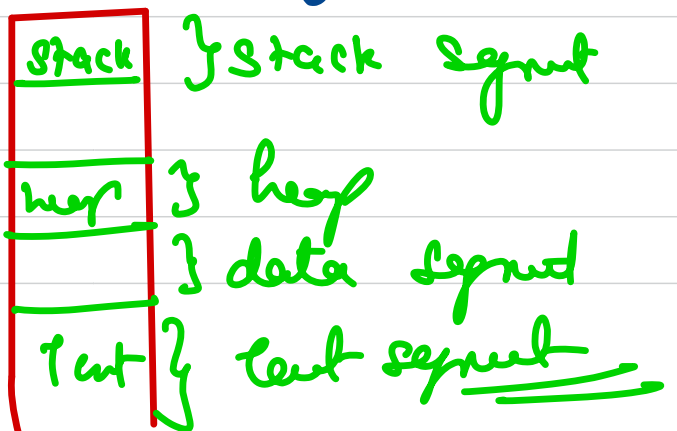

Dir
Table
offset
10
10
12
mmu
PDPR
p

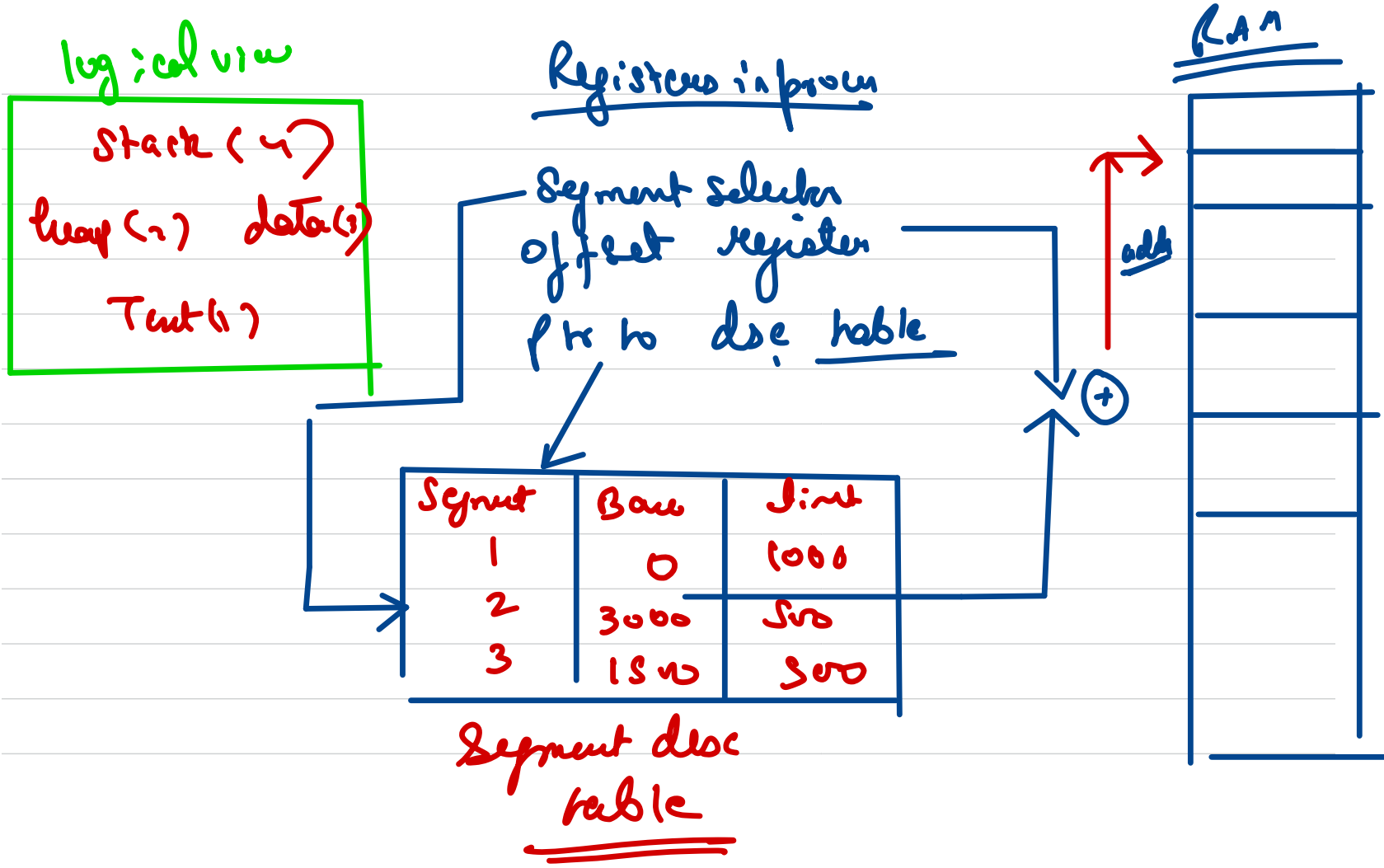
# Segmentation

Programs are collection of logical modules.

logical modules → global data, func°, class, namespace

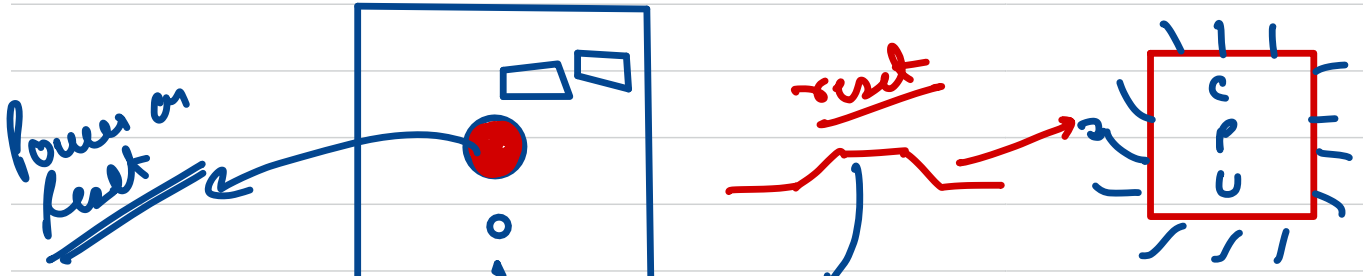
Segmentation splits programs into segment that are more logical




logical view
Stack (4)
heap (2) data (3)
Text (1)
Registers in proc
Segment selector
offset register
ptr to desc table
RAM
addr
+
Segment | Base | Limit
1 | 0 | 1000
2 | 3000 | 500
3 | 1500 | 300
Segment desc table

## Q. what happens when the PC boots ??

Intel processors maintain good backward compatibility.




→ this is sent to a special pin of CPU called as reset pin

The moment CPU receives the signal, CPU starts the process of turning the computer on (booting)

What happens during the booting step??

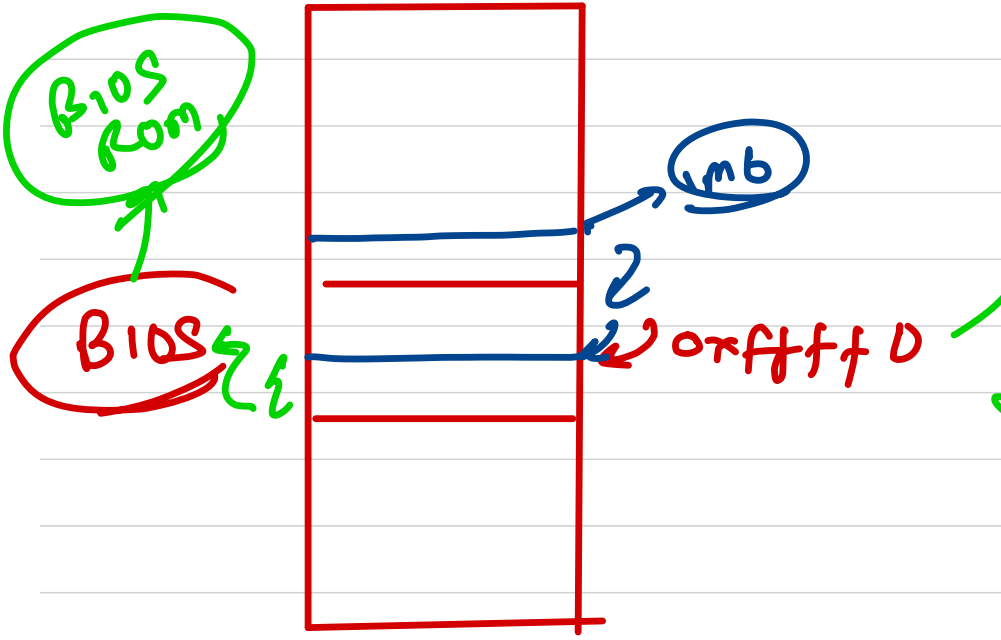
after power on reset, all the registers of the CPU are initialized with 0. except

CS → 0xf000

IP → 0xfff0

Calc the address of the first instruction to be execute

→ (CS << 4) + IP → 0xffffff0




What instruction is present here in

↳ this instruction makes us jump to the executable start of bios

Processor comes in REAL MODE (Backward compatible)

BIOS → Basic Input Output System

↳ It is a small chip connected to processor (ROM/flash)

# What Bios do ??

1) Power on self test
2) Initialize the video card & other devices
3) Display BIOS Screen
4) Perform memory test
5) Configure plug & play devices
6) Identify the boot device, read the sector 0 & copy it in 0x7C00, & jumps to 0x7C00

Sector 0 in the disk is called Master boot record.

↳ Total 512 bytes

↳ 446 bytes Bootable code

↳ 64 bytes is disk partition info

↳ 2 bytes Signature

Bootloader

↓

OS loads

# What happens inside bootloader ??

May give option to the user to select the OS to load (windows / linux)

→ Disable interrupts

→ Real Mode → Protected Mode ↗ we unlock the accessible memory beyond 1 MB.

→ Load the OS from disk

What happens when we power up OS??

Setup the virtual memory

Initiate
- to mouse, timer, HDD, FS etc

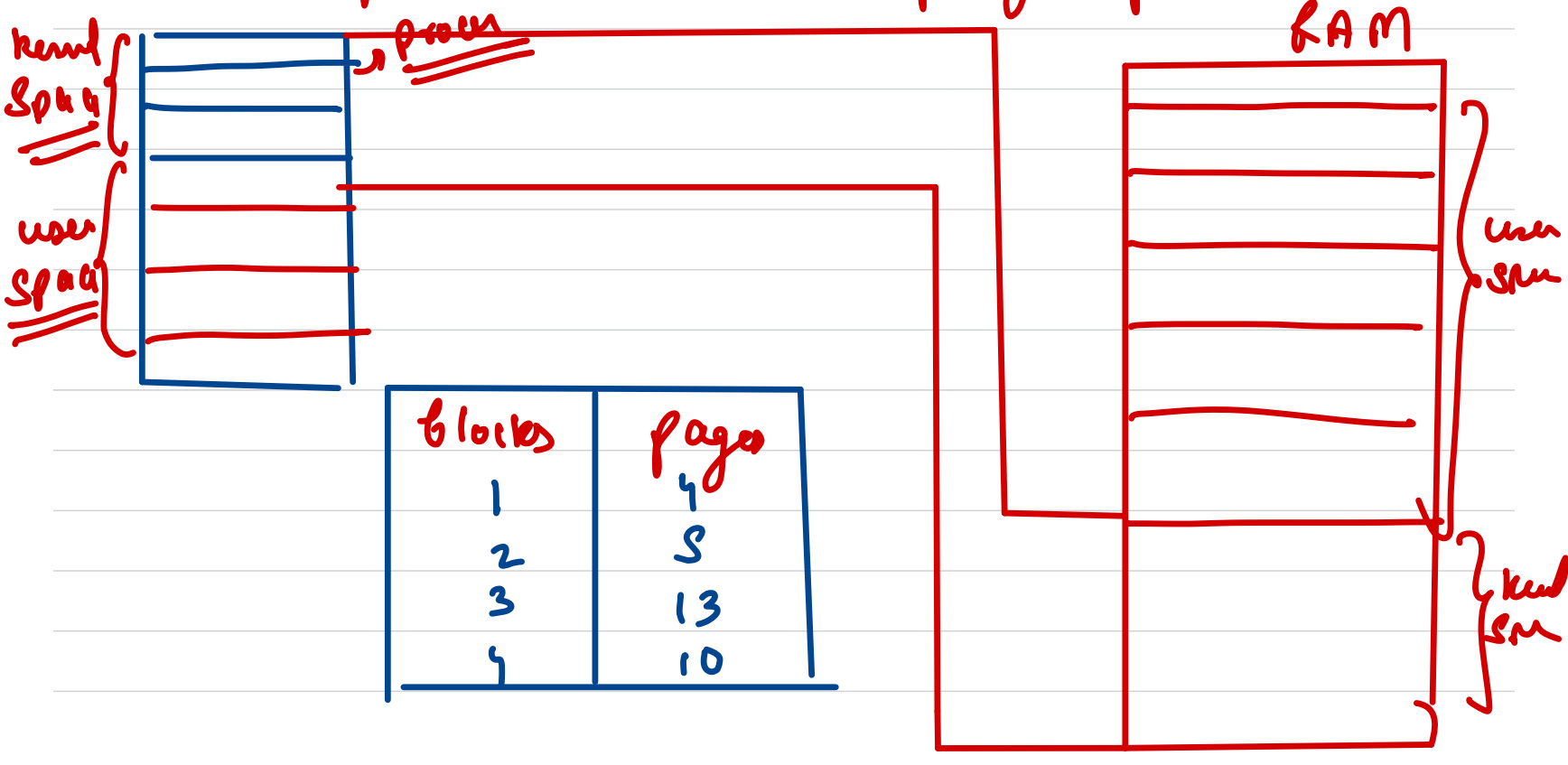
Starts user process

For multiprocessor booting, one processor is designated as "Boot Processor" (this info is stored in BIOS) all other processors are called app processors.

BIOS → Boot processor

↳ triggers booting of other app processors

# How is process created step by step ??



# Kernel data about a process

corresponding to each process, the kernel keeps some metadata

- PCB (process control block)
- kernel stack for each process to store context
- page tables for user process

# Process stacks

Every process has got two stacks:

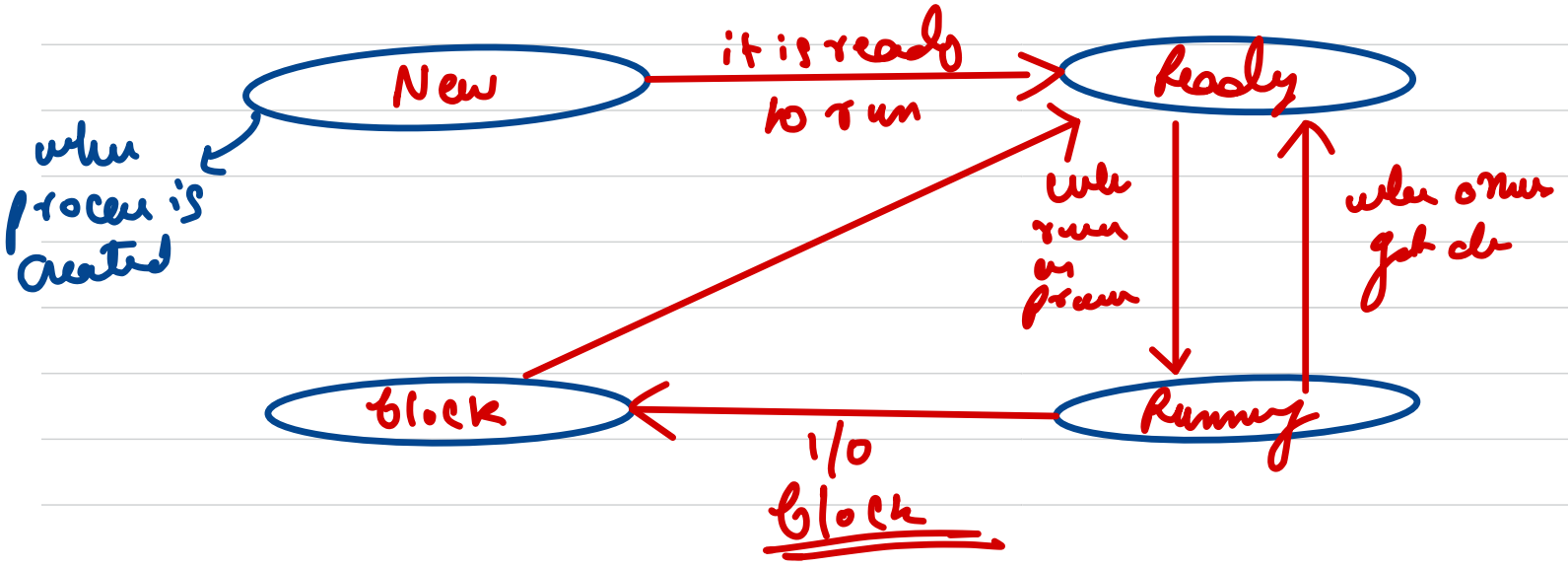
1. User stack → virtual call stack → what executing user code.
2. kernel stack
   ↳ Stores the content of a process

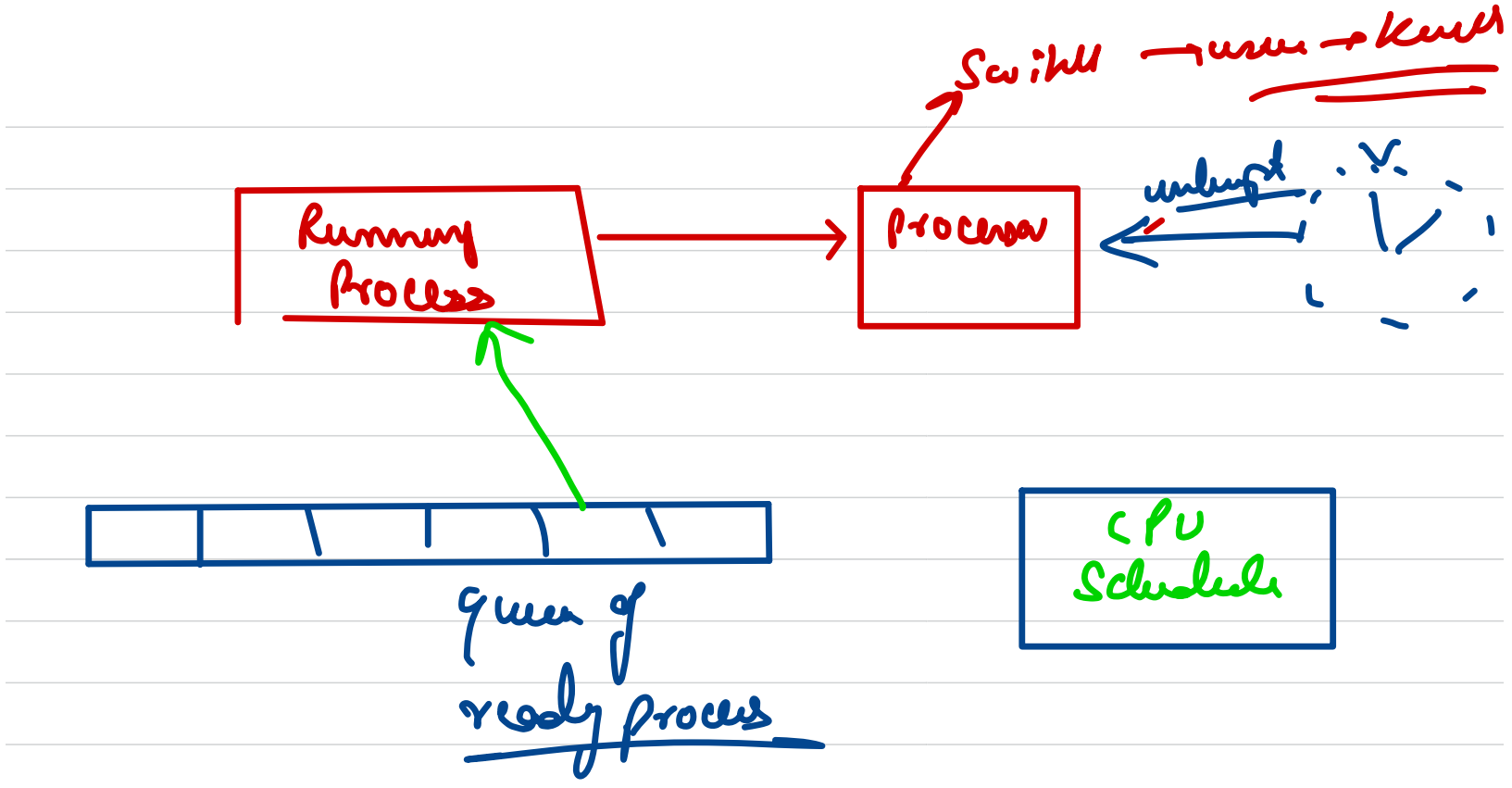
# PCB

Size of process memory
List of files opened
current working directory
Kernel stack pointer
process id
page directory
...

# pid → unique process identifier
generally sequential

# States of a process




Switch → user → kernel
Running Process
Processor
interrupt
CPU Scheduler
Queue of ready process

cpu Scheduler triggered to run when time / io / ome interrupt occurs or when runny process is blocked on: I/O

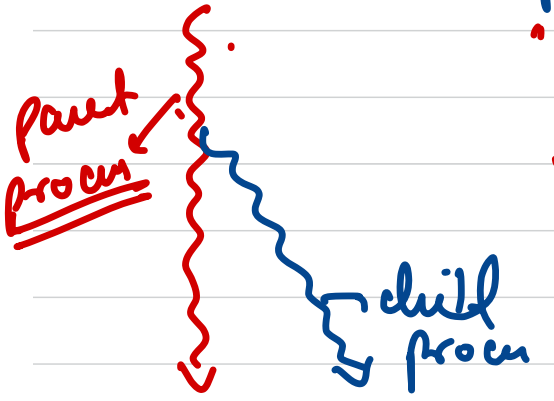
Scheduler picks another process from ready queue.

Performs a context switch.

# How actually process is created.

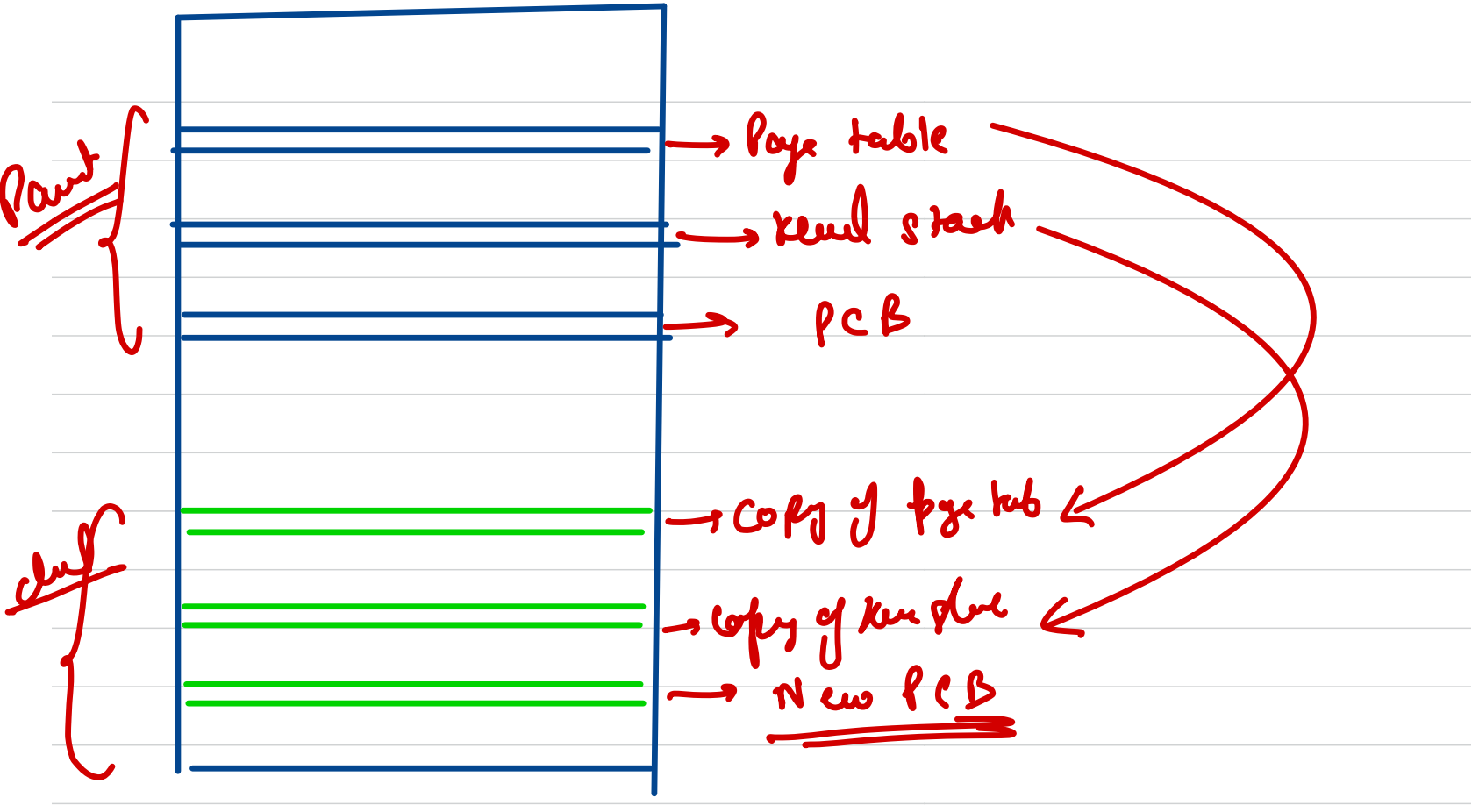
1) Cloning → child process is exact replica of parent process.

"fork" System call is executed




Note: → For accessing any kernel feature or resource, a OS process has to make a System call.


Process 1
system call
fork
Kernel
(wants fork)
child
Process 2


Parent
child
Page table
kernel stack
PCB
copy of page tab
copy of kernel
New PCB

→ find an unused PID

→ set the state of child process to new.

→ set pointers in the newly fork

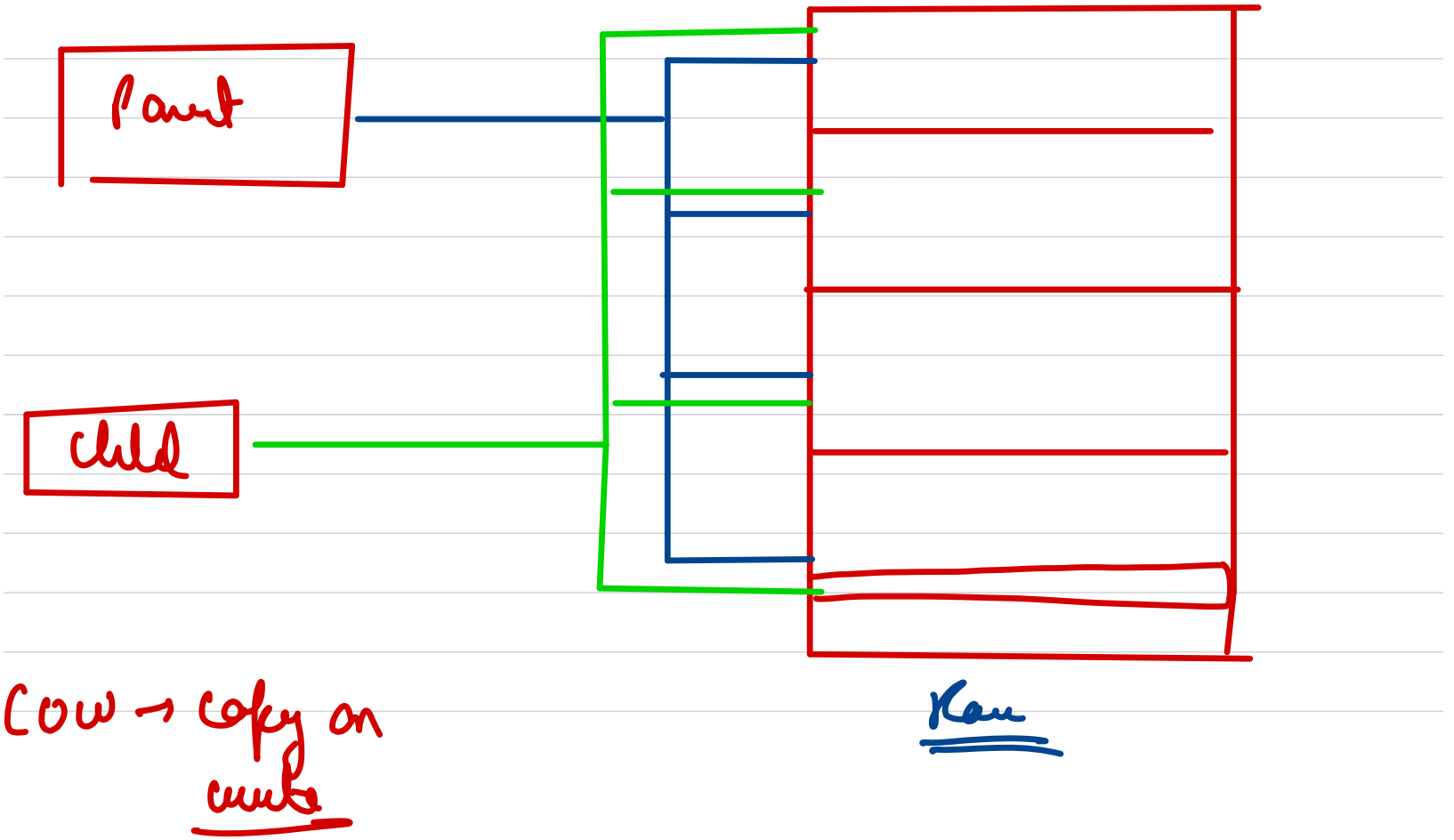
→ pagetable

→ mm_struct

→ Trapframe, set callee

→ copy info like env, files opened, etc from parent

→ set state to Ready before return

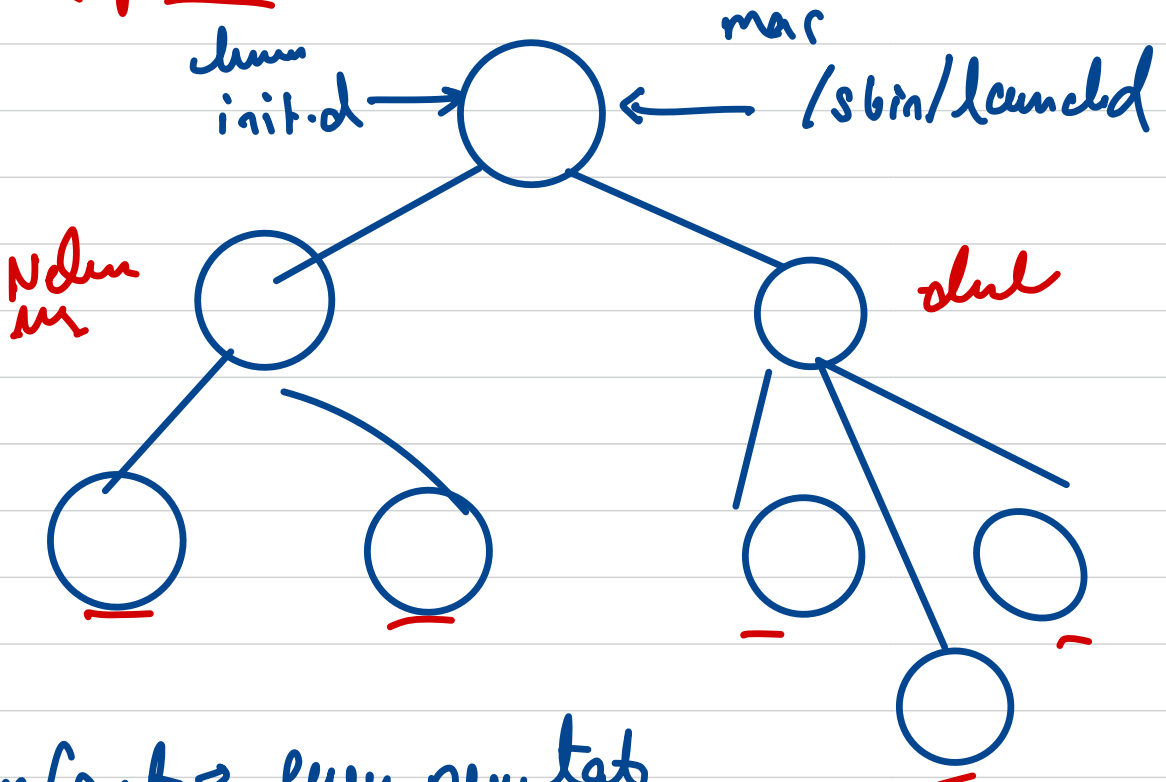

Parent
child
COW → copy on write
Ram

# exec system call → It is used to start execution of a process.

→ find or add the location of the executable

→ load on demand the page required to execute it.

# The process tree




#funfact → every new tab in chrome is a new process

# Exit System call :→

called generally inside a child process for voluntary termination

Terminates the process

The return status is 0 which returned to parent

# Kill system call →

this is a signal sent by other process or OS.

It also terminates the process.

What are zombie process ??

the moment a process is terminated at that moment it is not completely wiped from computer. So this is called zombie process

- PCB in OS still exists even though program is no longer executing.
  This is done so that parent process can read status of the killed child

when parent successfully reads the state the zombie is removed from OS using a process called as Reaper.

# Orphan process

When parent process terminates before it's child is adopted by root.

There are 2 type of orphan -

1. Unintentional orphan → when parent crashes.
2. Intentional orphan → Background process detach themselves from parent.

# Wait System call.

↳ this is enabled in kernel.

# Interrupt

OS and events → OS is event driven → it will come into the picture when an interrupt occurs.

Events / Interrupt:

→ (1) hardware interrupt

→ (2) Traps (Software interrupt)

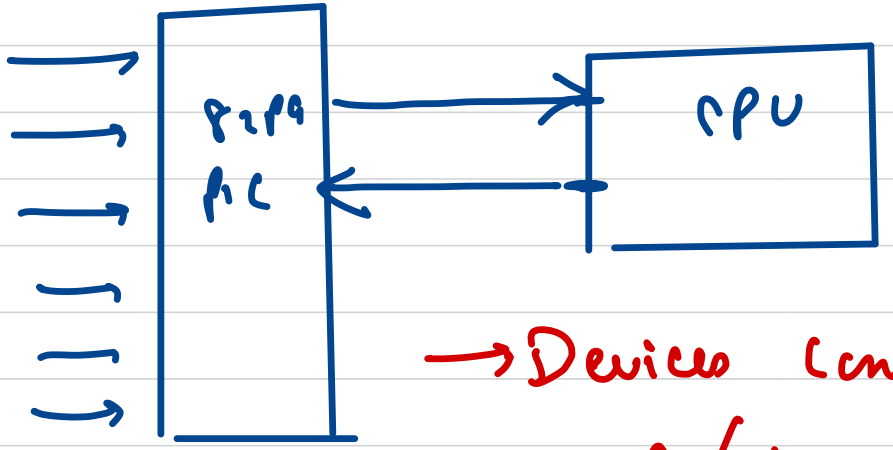
→ (3) Exceptions → faults → recoverable errors
→ aborts → difficult to recover

# hardware interrupt

Now a days every processor has a dedicated pin called as interrupt pin. Any hardware like keyboard, will be connected to this pin & whenever you press a key an interrupt occurs.

as soon as, the interrupt occurs the processor executes the interrupt routine handler. (ISR).

# PIC ( Programmable Interrupt Controller)




It relays upto 8 interrupt to CPU

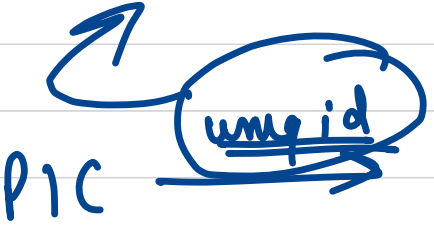
→ Devices connected raise a IRQ (interrupt req)

→ CPU acknowledges & requests 8259 to determine which device created interrupt

→ We can execute interrupts based on priority

Q. How to decide what Interrupt routin handler we need to execute for a hardware interrupt ??

→ IDTR (Interrupt Descriptor table)



# CPU context Switching

Let's consider we have a Single CPU in the system which is shared among multiple processes. So, this processor does not executes every thing parallel. OS by a feature called as Multi tasking enables that this CPU is fairly shared among processes.

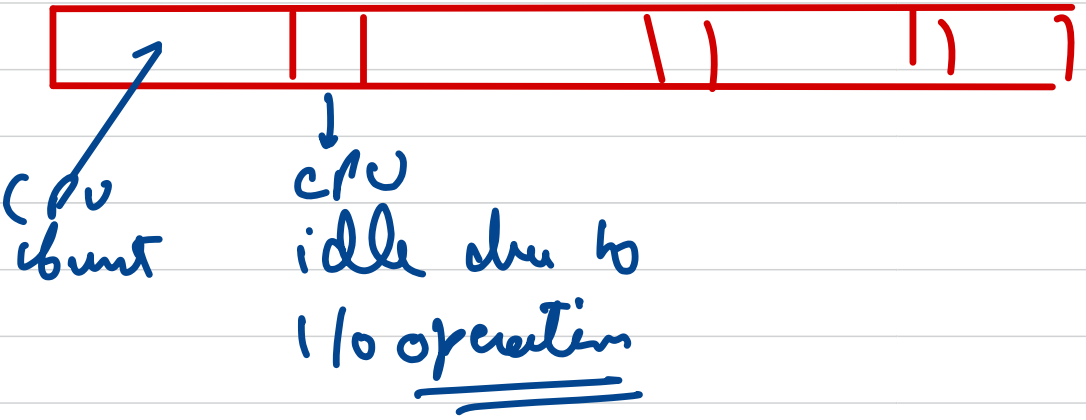
So, in a multi tasking env, or in a multi tasking enabled OS, the OS would allow one process to execute for some time and then there is a context switch. So during this context switch process 1 would stop running, & other process say process 2 starts execution.

The time delta for which a process run is called time slice.

# Context switch overhead

# CPU scheduling

which process the scheduler should choose??



Based on this we can say, we divide processing types -

1) I/O Bound → has small cpu burst & more I/O
2) CPU Bound → 3d rendering

# #scheduling Criteria

1) Max CPU utilisation → CPU should not be idle.
2) fairness → Given each process a fair share of CPU unless they are exception.
3) Max Throughput → Complete as many process as possible per unit time.
4) Arrival time → when process enters ready queue

5) Min turnaround time → for a process total time from start to end.

6) Min waiting time

7) Min response time

# FCFS (first come first serve)

The first job that comes to CPU get's the CPU

Non-preemptive → The process continues till the burst cycle ends.

Note → Burst time → It is the total time taken by the process for it's execution in CPU.

if the arrival time is same then we choose randomly.

- advantages → easy to implement.
- disadvantages → waiting time depends on order. short process stuck waiting for long process to complete.

★ convoy effect

# Shortest Job first (SJF) (Non-preemptive)

→ schedule process with the shortest burst time.
→ FCFS if same.

→ Non-preemption → the process continues to execute until it's CPU burst completion

| | Arrival | Burst |
|---|---|---|
| $P_1$ | 0 | 7 |
| $P_2$ | 0 | 4 |
| $P_3$ | 0 | 2 |
| $P_4$ | 0 | 1 |

$\frac{0+1+3+7}{4} \Rightarrow 2.75$ avg wait time

↳ $P_4 \rightarrow P_3 \rightarrow P_2 \rightarrow P_1$ →

| | Arrival | Burst |
|---|---|---|
| $P_1$ | 0 | 3 |
| $P_2$ | 2 | 4 |
| $P_3$ | 4 | 2 |
| $P_4$ | 3 | 1 |

$$\frac{0 + 0 + 4 + 8}{4} \rightarrow 3$$

$P_1 \rightarrow P_4 \rightarrow P_3 \rightarrow P_2$

- advantage → Optimal → min wait time, avg response time decrease
- disadvantage → Not practical
  May starve long job.

What if we introduce pre-emption ?^ Any thoughts

Preemption → if a new process arrives with a shorter burst time than remaining of current process will we start executing new process.

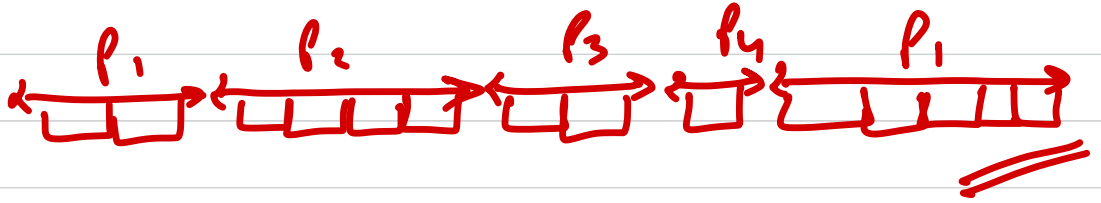
This further reduces avg wait time. & avg response time.

→ Not much practical, we can't predict burst time.

| | Arrival | Burst |
|---|---|---|
| $P_1$ | 0 | 7 |
| $P_2$ | 2 | 4 |
| $P_3$ | 4 | 2 |
| $P_4$ | 5 | 1 |

$$\frac{7 + 0 + 2 + 1}{4}$$

$$\rightarrow 2.5$$



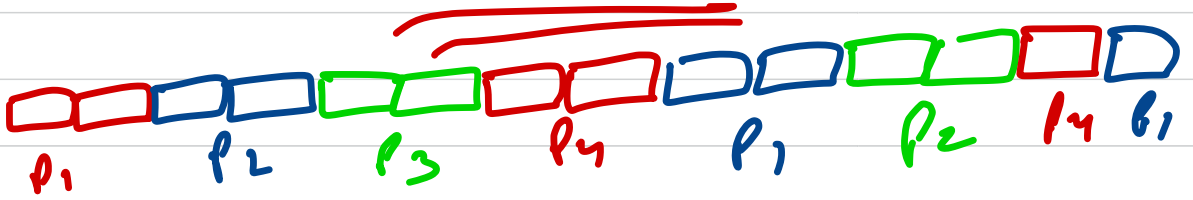
# Round Robin algo ??

Runs a process for a time slice then moves it back to ready queue & starts executing next process.

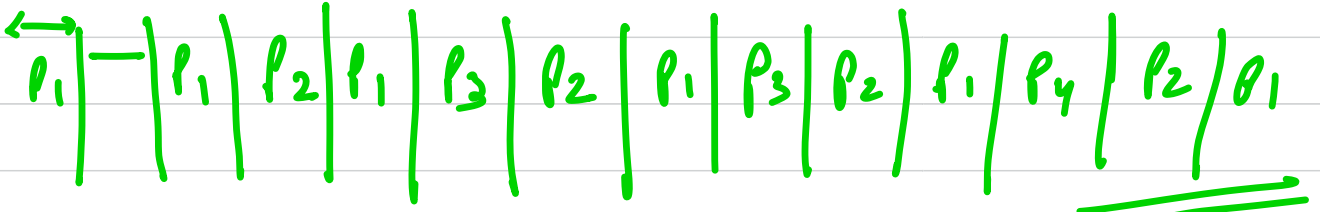
At every timer interrupt pre-empt current process.

Arrival Burst Time slice = 2

| | | |
|---|---|---|
| $P_1$ | 0 | 5 |
| $P_2$ | 0 | 4 |
| $P_3$ | 0 | 2 |
| $P_4$ | 0 | 3 |



| | Arrival | Burst | T.S = 1 |
|---|---|---|---|
| $P_1$ | 0 | 7 | |
| $P_2$ | 2 | 4 | |
| $P_3$ | 3 | 2 | |
| $P_4$ | 9 | 1 | |




Rules of time slice :

# a short quantum →

# a long quantum →

typically → 10ms — 100ms

#advantage

→ fair

→ low avg wait time

→ fast response time

#disadvantage → Inc context Switch

# Priority

Each process is assigned a priority (0-255)

a small priority no. means high priority.

Scheduler → from the ready queue, pick the process with highest priority.

# Multilevel Queue

→ Process is assigned to a priority class

→ Each class has it's own ready queue.

→ Scheduler picks the highest priority class, which has atleast one process

→ Selection of a process within the class could have it's policy -
  ↳ typically RR

# Scheduling in Linux:

Linux classifies process in 2 types →

Real time →
↳ have strict deadlly like RTC
↳ should never be blocked by Time critical task

Normal →
↳ generally interacts with user.
↳ Some are batch process that don't require I/O activities in background.

# History of linux scheduler

→ O(n) scheduler

→ O(1) scheduler

→ CFS scheduler

* O(n) scheduler → at every context switch

→ scan the list of process in ready queue

→ Compute priorities.

→ select best to run.

Q 11) → It used context time to pick next process.

Divides the process in 2 types.

1) Real time → (0, 99)

2) Normal → (100, 129)