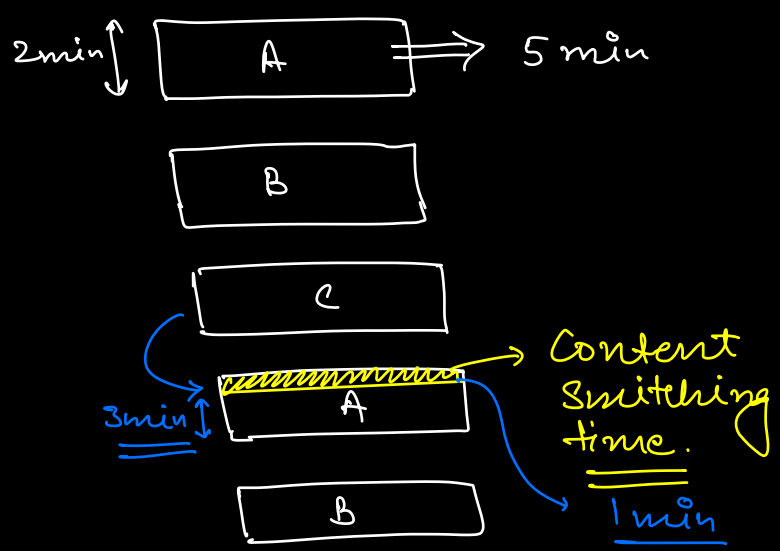
* If an OS is preemptive, the CPU can schedule other processes in b/w that will make OS a bit faster.

⇒ CPU Scheduling.

Why Scheduling ?

1. Efficient.
2. Iteractive.

CONTEXT SWITCHING



When CPU switches the process, it will have to store the state of previous process & fetch the state of current process time.

⇒ Context Switches.

⇒ A lot of context switches aren't good for the system.

CPU Scheduling Algo

Assume
- → 0 context switch time
- → Single core CPU (1 process at a time)

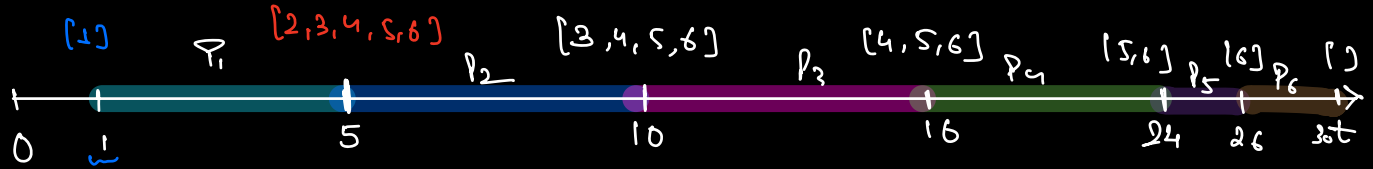
1. FCFS
2. SRTF
3. Round Robin

FCFS

Process.

Pid	arrival_time	timetocomplete.
1	1	4
2	2	5
3	3	6
4	4	8
5	4	2
6	5	4

→ Predicted time.



⇒ At any point of time CPU has to decide which process to run, it will the process with least arrival time. If more than 1 processes have same arrival time the CPU will pick the process with less process_id.

⇒ Non preemptive Algo.

* Implement FCFS.

② SRTF (Shortest Remaining Time first)

→ Pre Emptive Algo.

↳ Already running process can be paused to run a new process.

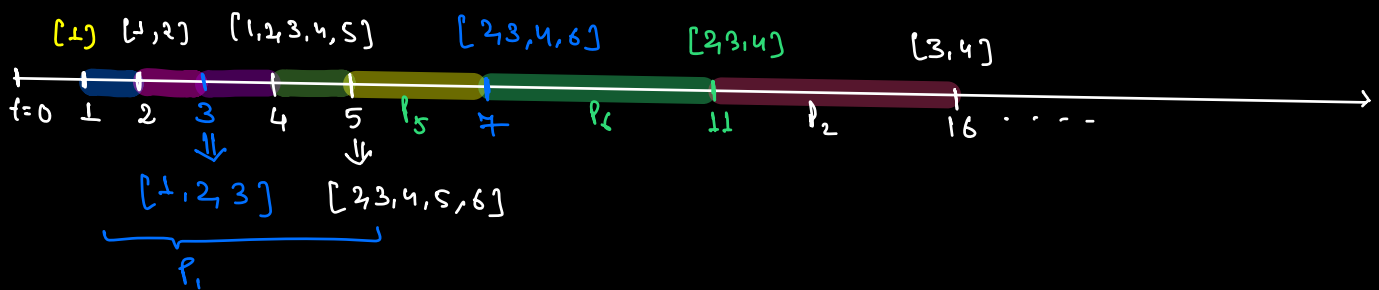
→ This algo will run

a) Whenever a new process comes in the system
b) Whenever earlier process completes.

→ Whenever the algo runs, it will calculate the remaining time of all the processes in the system & CPU will pick the process with least remaining time.

Burst time

Pid	arrival_time	timetocomplete	remaining time
1	1	4	4 3 2 1 0
2	2	5	5
3	3	6	6
4	4	8	8
5	4	2	2 0
6	5	4	4 0



STARVATION :

↳ A process with large time to complete then it will have to wait for longer time.

Round Robin CPU Scheduling

q = 2 sec.

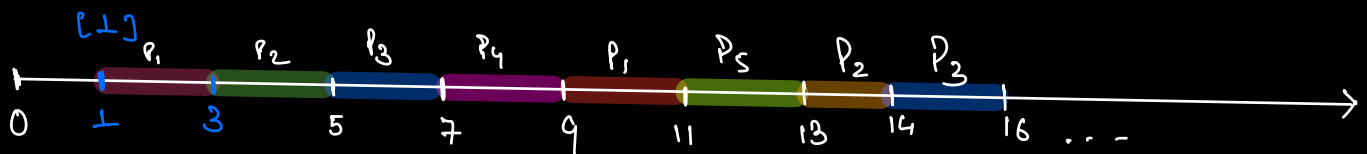
↳ time Quantum.

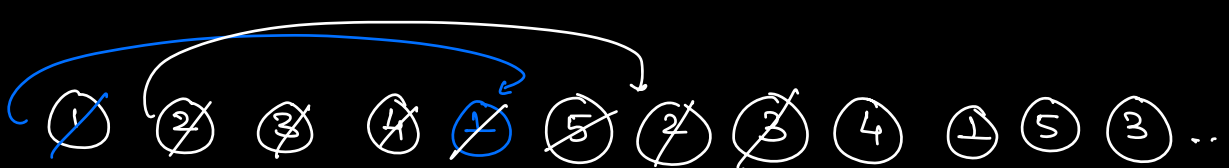
⇒ After every q seconds, CPU will pause the current process & it will pick the new process.

Processes

id	Arrival time	time to complete
1	1	~~6~~ ~~4~~ 2
2	2	~~3~~ ~~1~~ 0
3	3	~~9~~ ~~7~~ 5
4	3	~~4~~ 2
5	5	11

q = 2 sec





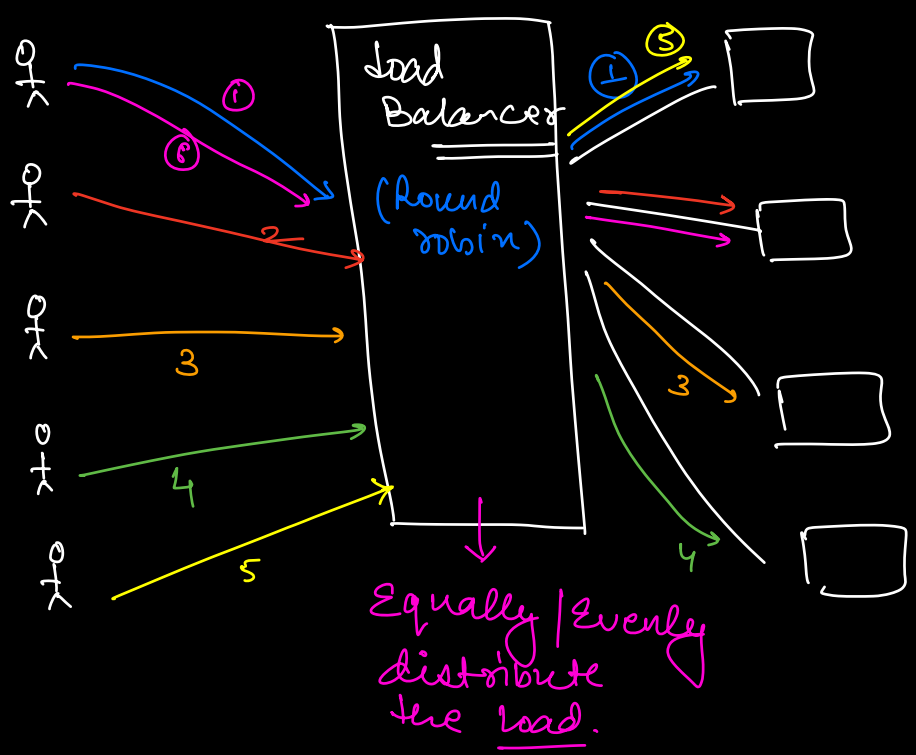
* q is very large ⇒ FCFS.
* q is very small ⇒ Too many context switches

→ Every time the process process completes or time quantum is elapsed, pause the current process move it to the back of the queue & run the process from the front of the queue.

Starvation ×

Partially ×

⇒ Load Balancer.



Throughput :

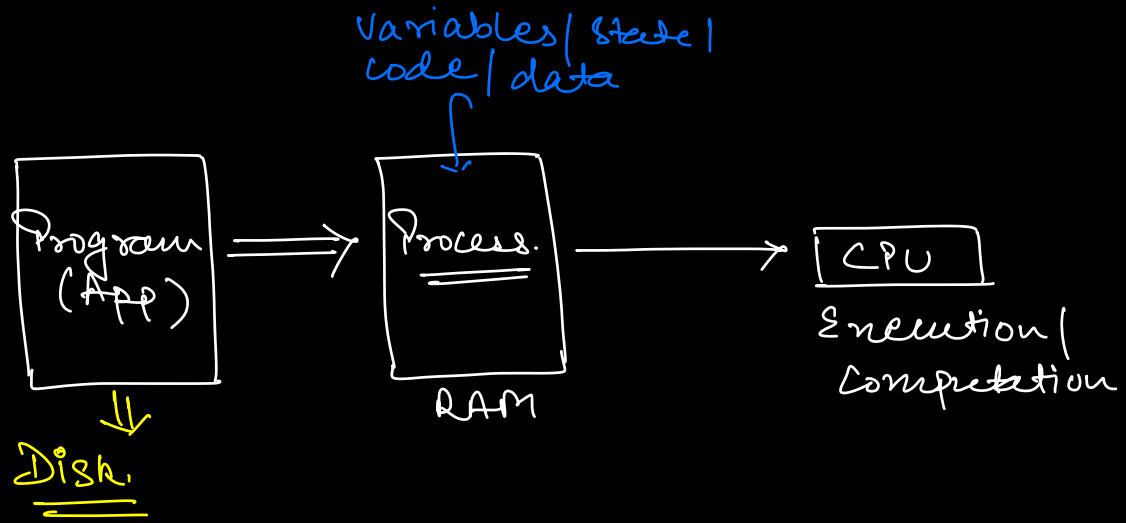
↳ # of processes completed per unit of time.

$$\frac{6}{30} = \frac{1}{5} = 0.2 \text{ processes/sec.}$$

Latency :-

↳ Avg time CPU takes for 1 process to complete from the time CPU knows about it.

Threads



Process = Program in Execution.

⇒ Speed gap.

CPU is very fast.

Word Processor (MS Word / Google Doc)

I will give a good party to my instructors once I get my dream job

- Spell checking
- Auto saved
- Grammar check
- Suggestion
- Auto updates
- Count of words/lines

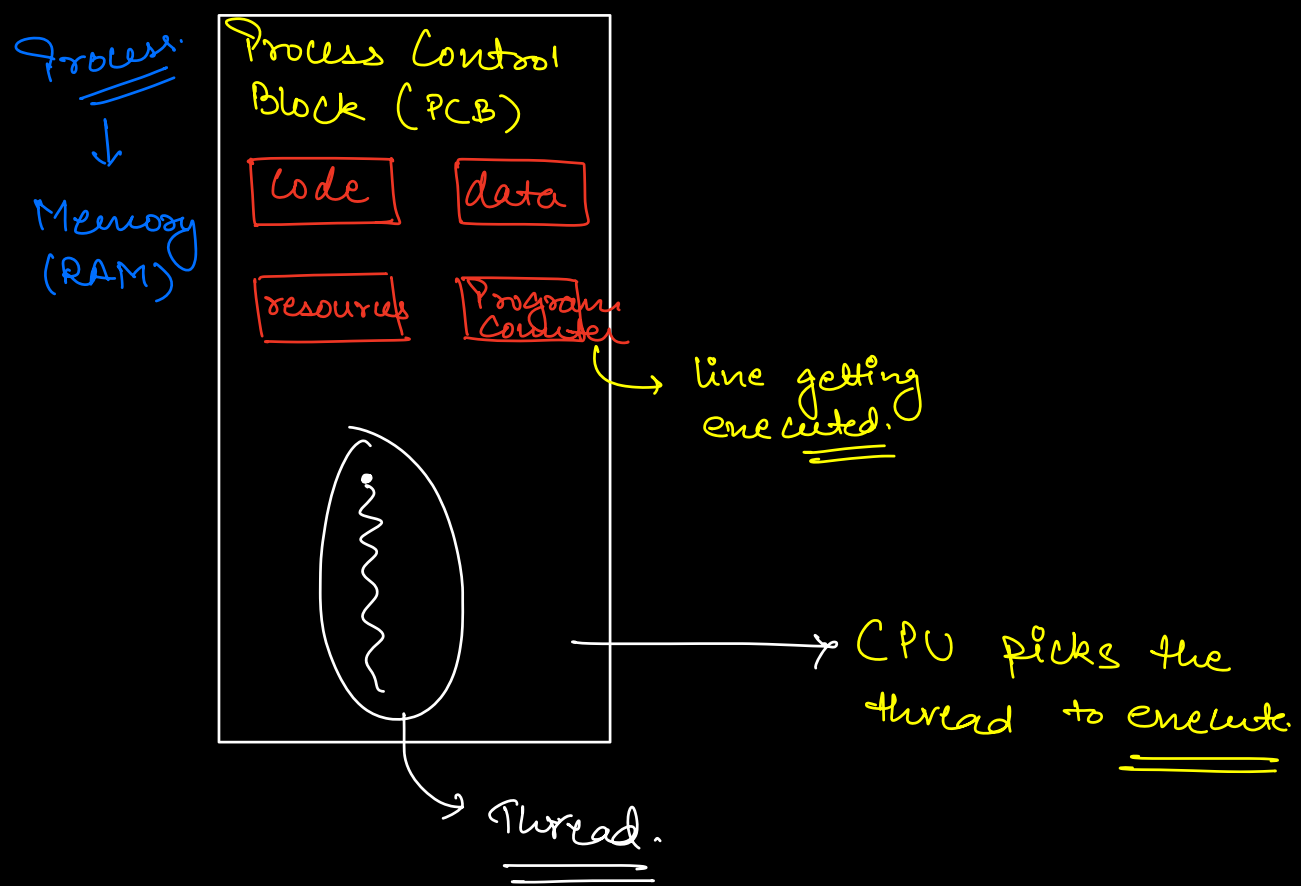
```
main() {
    → print(hello)
    → doSomething()
}

doSomething() {
    → print(hi)
}
```

Code executes line by line.

⇒ One task at a time.

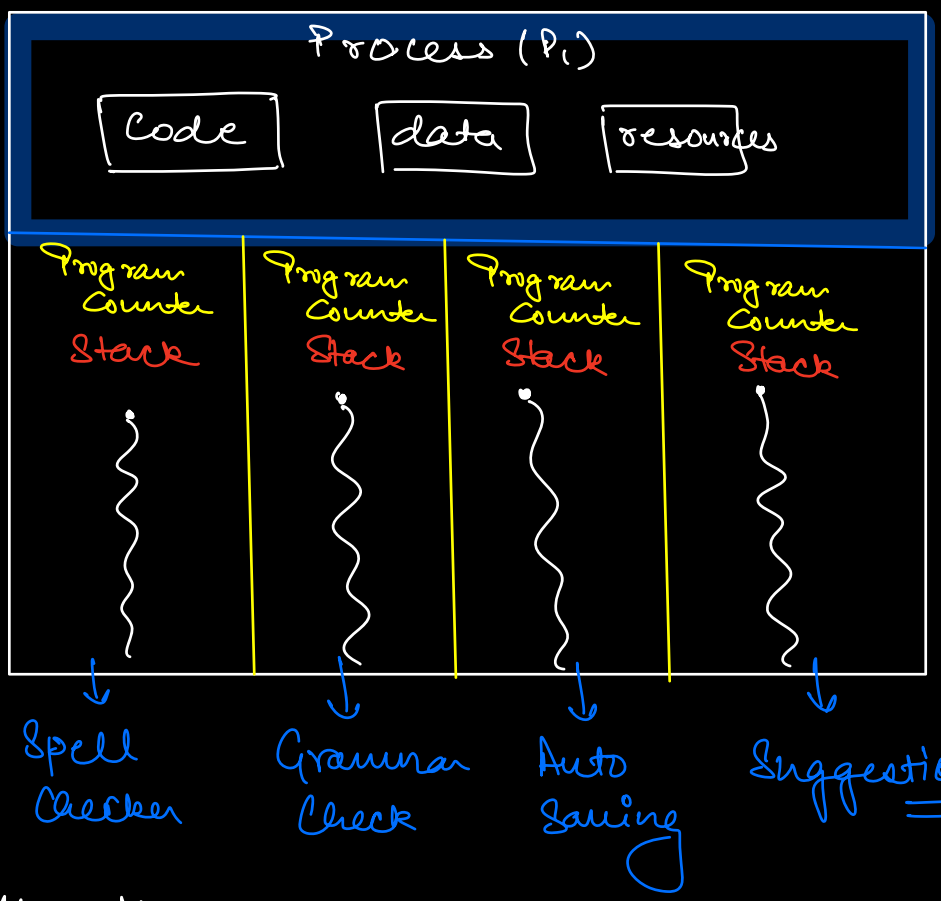
Thread: Unit of CPU execution.

↳ Whenever anything is running on our m/c there is a CPU running a thread which actually runs the source code.



MULTI THREADING

PCB



→ All the thread of the same process will have the access to data.

Thread : CPU executes the thread.

Process Vs Thread.

Data Sharing :

All the threads access the data from PCB but diff. process can't access each others data.

↳ IPC (Inter Process Communication)

→ Process takes more memory.
(code, data ---)

→ Process creation is time consuming than a thread.
