⇒ Recap → OS, Uni vs Multiprogramming OS

Process, PCB, Pre - Non Pre, CPU Scheduling

Non - preemptive ⇒ can't switch process

Preemptive ⇒ switching is possible

If we have a pre-emptive OS, we can schedule some other process that will make our system more efficient.

↓

CPU Scheduling

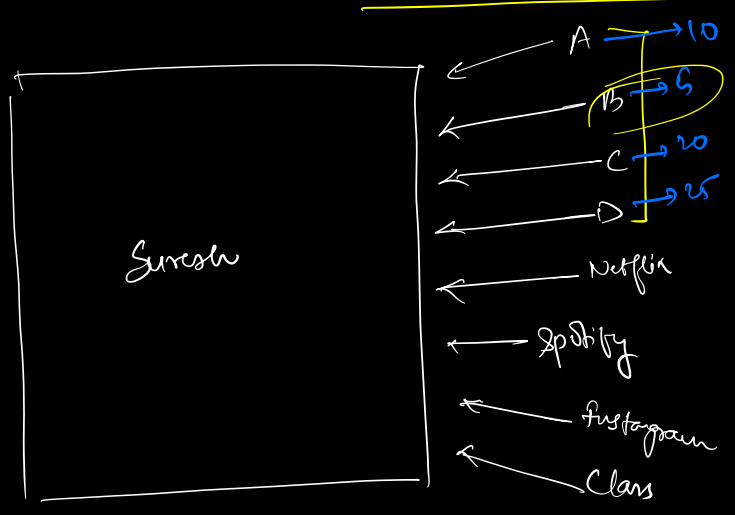
⇒ Why CPU Scheduling :-

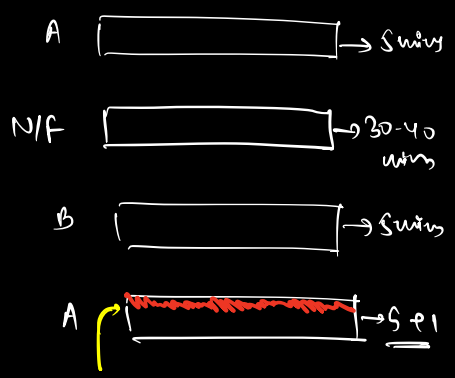
⇒ Efficiency

⇒ Interactive

⇒ Problems due to CPU Scheduling :

: CONTEXT SWITCHING

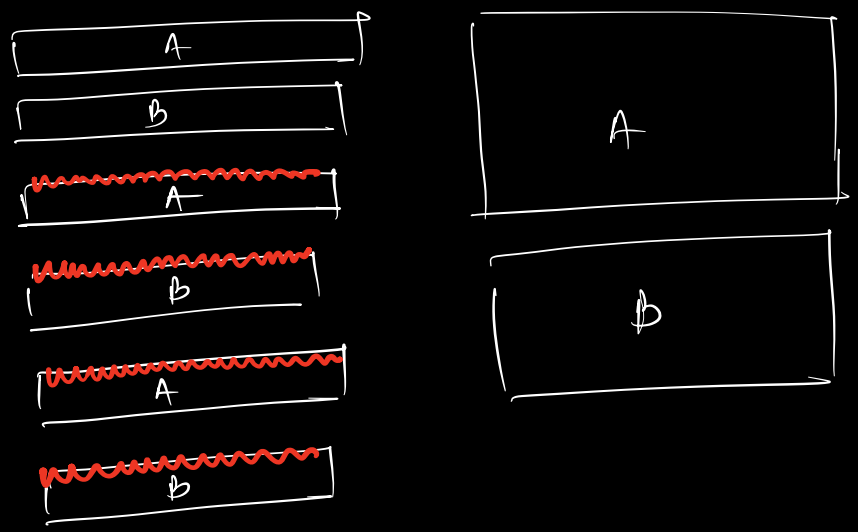



Some extra time spent here to get context (1 min)

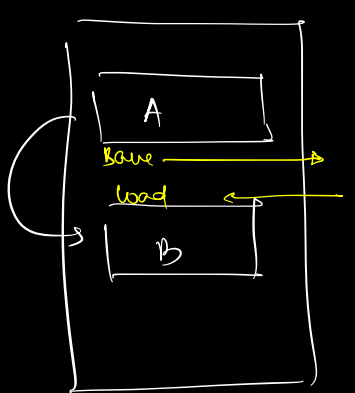
A ⇒ 5 + 5 + 1 = 11 secs

A B




CPU : when a CPU switches a task, it will have to store its previous state to the memory, and it will also have to load prev state of next task in the memory

* It save and loading takes extra time
* CONTEXT SWITCHING
* A lot of context switching can do more harm than good because they also take time



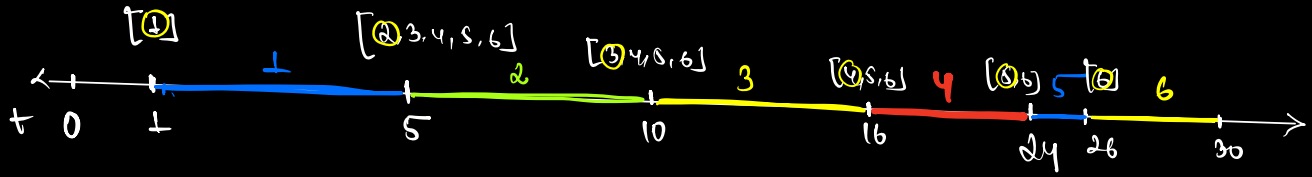
## : CPU scheduling algo :-

* 0 context switching time
* Single core CPU ( 1 task at a time)

### 1) FCFS ( First Come First Serve) :-

| PID | Arrival time | Time to complete |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 2 | 5 |
| 3 | 3 | 6 |
| 4 | 4 | 8 |
| 5 | 4 | 2 |
| 6 | 5 | 4 |

⇒ At any time CPU has to decide which process to run next, here it will choose the process that had come the earliest, if 2 or more processes come at the same time it will the process with lesser pid.




: Non-preemptive algo : No scheduling until a process finishes

H.W

Implement FCFS → Ip ⇒ List<Process>

↓

Process

→ pid

→ arrival time

→ time to complete

⇒ Op ⇒ List < pid, time >

: SRTF ⇒ Shortest Remaining Time First

a) Pre-emptive algo:

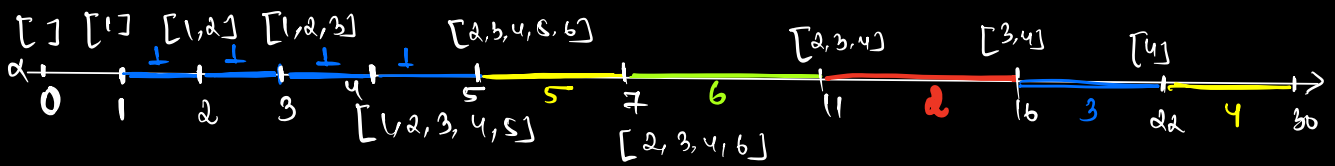
it can pause a process already running to schedule a new process.

b) this algo will run:-

- whenever a new process comes up
- whenever an earlier process completes.

c) Whenever this algo runs, it will calculate the remaining time of all processes that have come till now and will schedule one with the shortest remaining time.

| PID | Arrival time | Time to complete | Remaining time |
|---|---|---|---|
| 1 | 1 | 4 | 3 2 1 0 → 4 |
| 2 | 2 | 5 | 5 0 → 14 |
| 3 | 3 | 6 | 6 → 19 |
| 4 | 4 | 8 | 8 → 26 |
| 5 | 4 | 2 | 2 0 → 3 |
| 6 | 5 | 4 | 1 0 → 6 |




→ **Problem with SRTF :-**

: **STARVATION :**

A process with large time of completion will starve for CPU.

⇒ **Round Robin CPU scheduling algo :-**

: Only 1 process is picked up, and other processes make 0 progress.

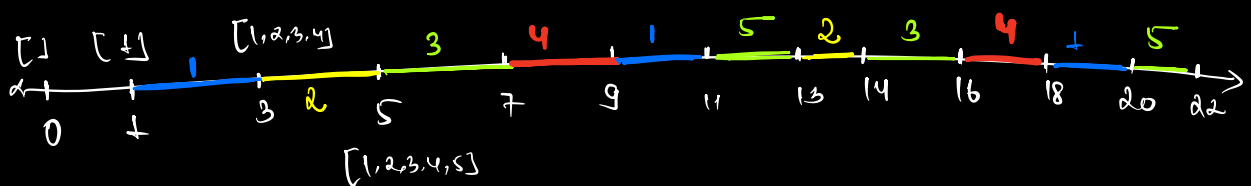
: take a quantum of time = q

assume q = 2s

Idea : after every quantum of time, pause the current process and pick up a new one.

Process

| Pid | Arrival time | Time to compl. |
|---|---|---|
| 1 | 1 | ~~6~~ ~~4~~ ~~2~~ 0 |
| 2 | 2 | ~~3~~ 1 0 |
| 3 | 3 | ~~9~~ ~~7~~ 5 |
| 4 | 3 | ~~4~~ ~~2~~ 0 |
| 5 | 5 | ~~11~~ 8 7 |




(5) (3)

(2) (4) (1)

⇒ Every time when a previous process ENDS OR a quantum of time elapses, pause the current process, move to the back of queue, and start the process at the front of queue.

→ partially → X

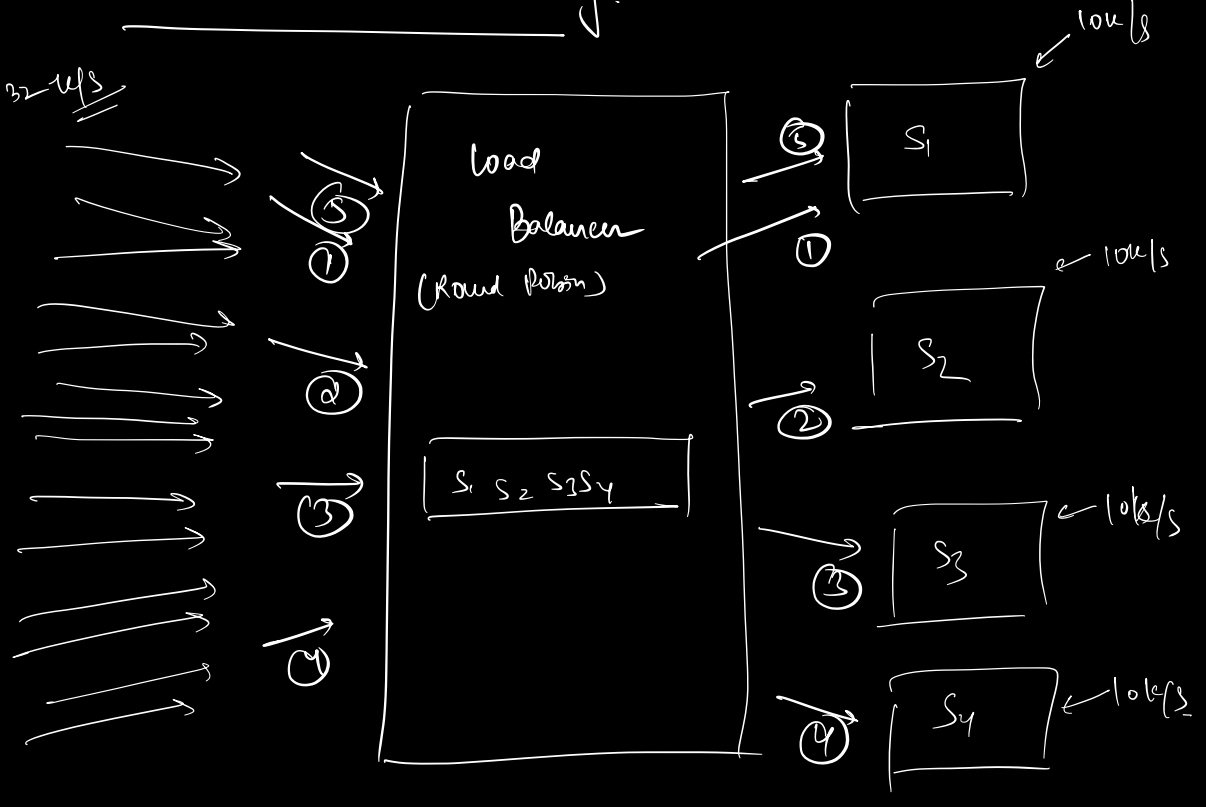
→ starvation → X

→ simple algo → ✓

if q is huge, then RR turns into FCFS

if q is too low, causes too many context switches

RR in load balancing :-




Equally distributing load

: Throughput :- No. of processes executed per second (unit of time) is called throughput

(↑)

eg ⇒ $\frac{6}{30} = \frac{1}{5} = 0.2$ p/s

: Latency :- Avg time it takes for a process to complete from the time CPU knows about it.

(↓)

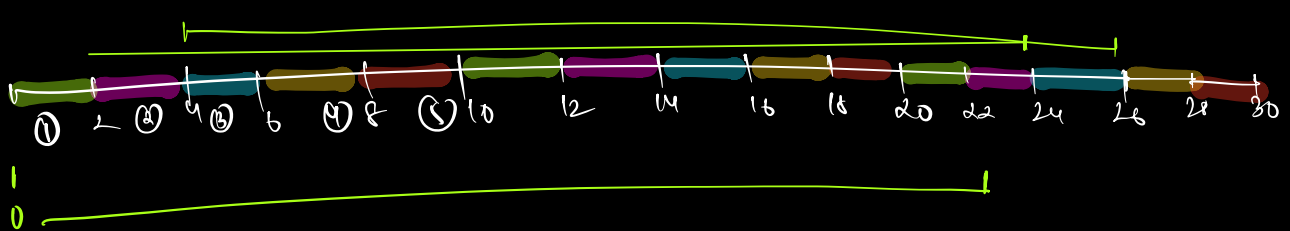
Pid

1 → 4
2 → 14
3 → 19
4 → 26
5 → 3
6 → 6

avg = $\frac{4+14+19+26+3+6}{6} = 12$

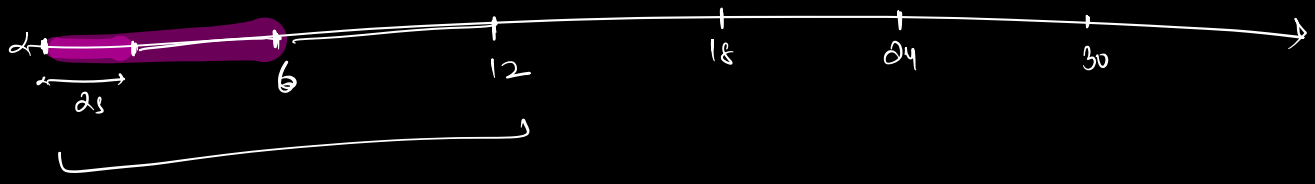
⇒ Round Robin

5 process → 6 secs → q = 2s




$\frac{22+22+22+22+22}{5}$

SRTF

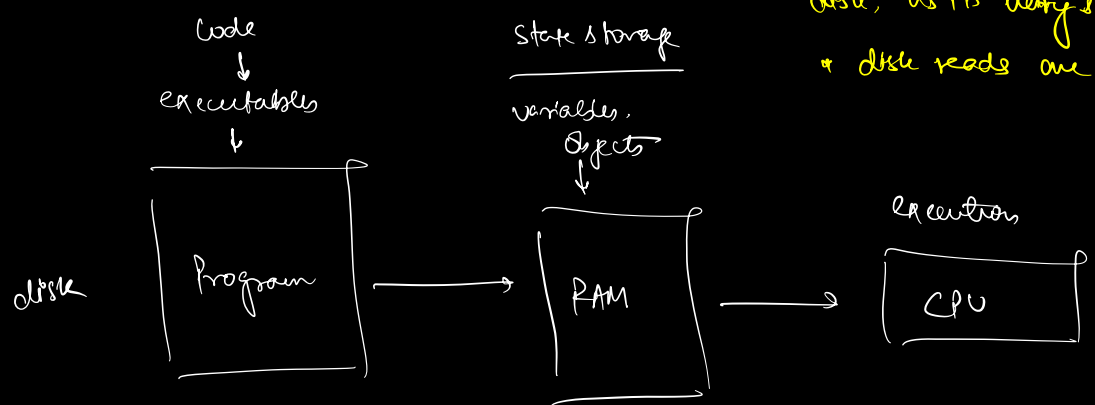




$$\frac{6 + 12 + 18 + 24 + 30}{5}$$

Latency → SRTF > RR

THREADS

* CPU doesn't talk to disk, as its very slow.
* disk reads are I/O




Word Processors ( MS. word, Google Doc) etc.

Hi, How are you?

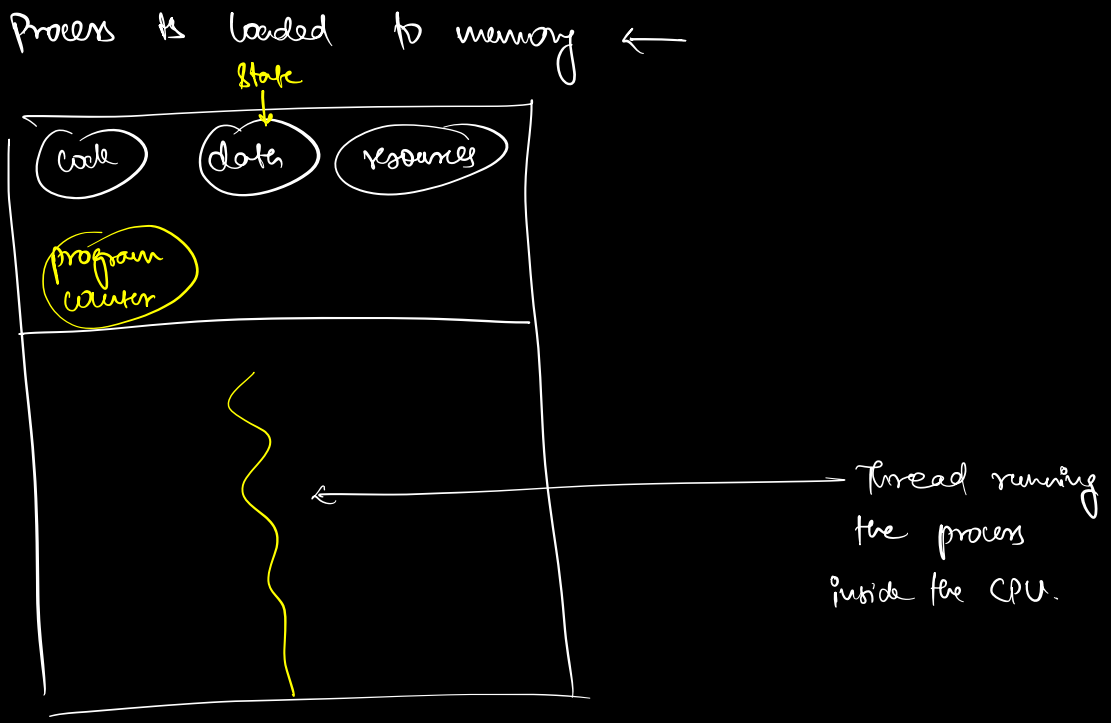
1) grammar check
2) spell check
3) display UI
4) Suggestions
5) Auto save
6) Checks for updates

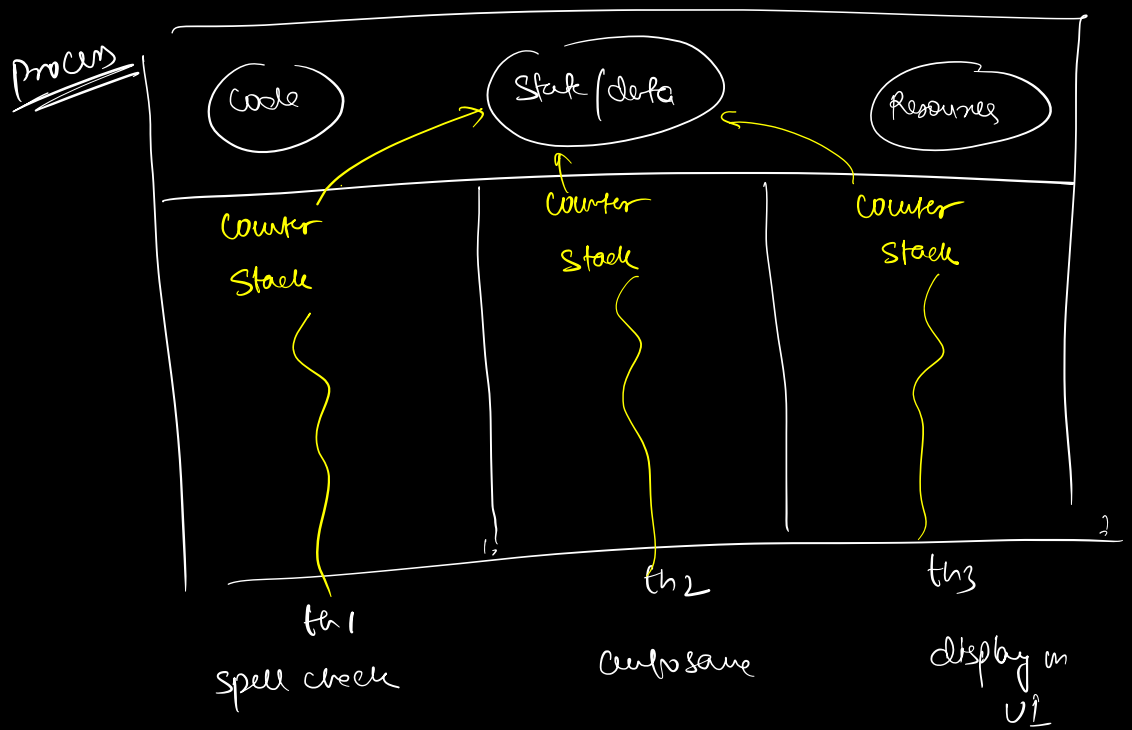
## Thread

→ A unit of CPU execution.

→ CPU executes thread

→ whenever anything is running on your machine, there is a CPU running a thread, which is running the code.






⇒ all threads have access to data

⇒ Process vs thread :-

1) threads can execute parallely multiple things

2) data sharing → threads can share data but processes can't share data.

Processes talk to each other, called IPC { Inter Process Communication }

3) Processes takes more memory than threads.

4) Creating a new process is slower than creating a new thread.