Operating Systems 2 : Threads

9:08 PM

180

Agenda

- → CPU Scheduling Algorithms
- → Threads
- → Concurrency v/s Parallelism
- → IPC
- → Producer Consumer Problem

→ HLD

→ Codes!!!

Agenda

- → Context Switch
- → Scheduling Algorithms

→ FCFS

→ SRTF

→ Round Robin ⇒ Starvation of Process

Threads

→ Motivation

→ Multicore

↳ Concurrency v/s Parallelism

→ Practical code of Threads

→ In Java

→ New way ⇒ Executor =

→ Thread Pools

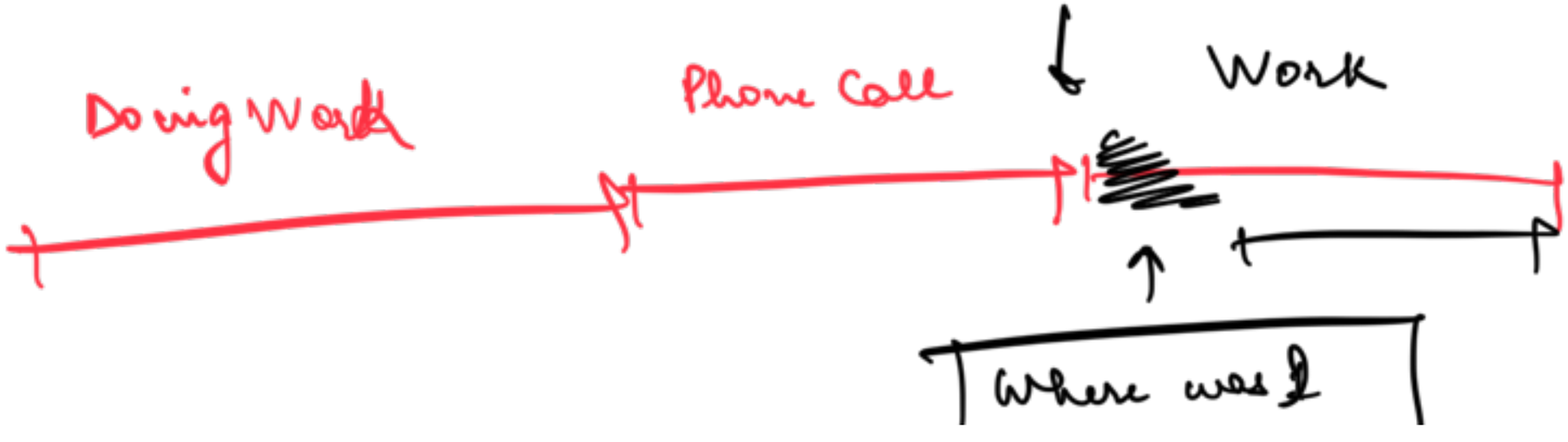
→ Callables (Futures) → Merge Sort using Threads

Cons of Threads

→ Synchronization

→ Mutex
→ Semaphores
→ Code

Context Switch



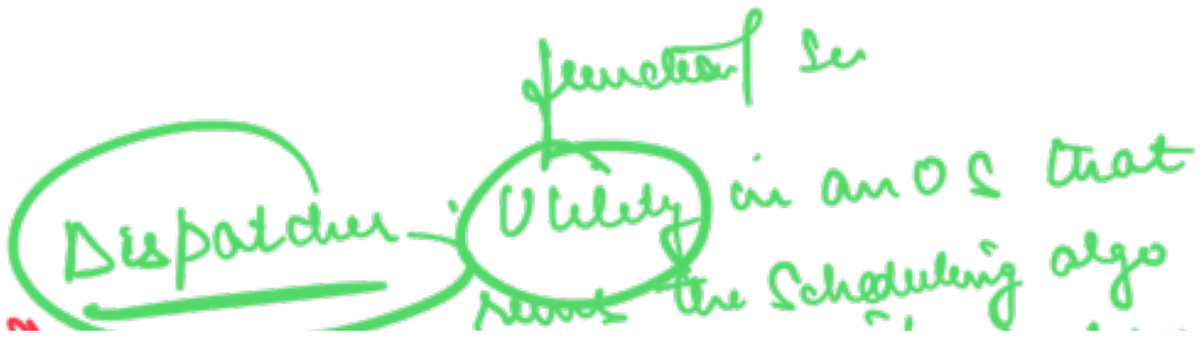
→ Change of work == change of context

→ Switch of work == Switch of context

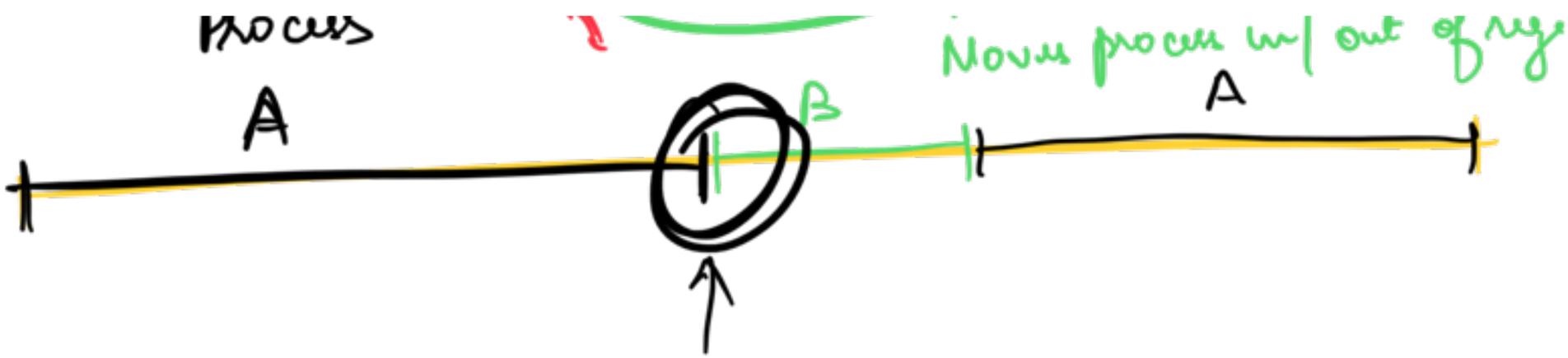
→ Context Switching

Spend some time
recollecting what I
was at to resume

In a CPU



process A

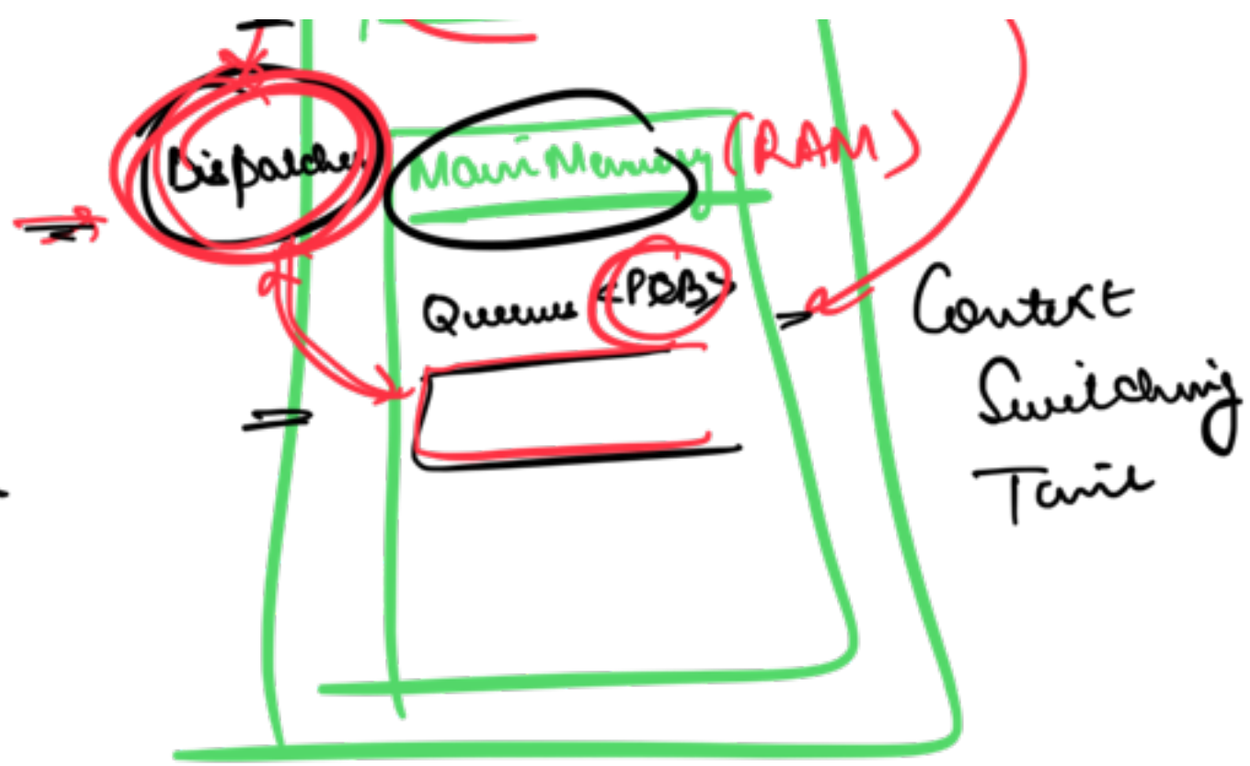


New process w/ out of reg

where a CPU switches the process ⇒ context switch

PCB

Process Control Block



If a CPU Scheduling algo ⇒ a lot of context switch

PCB {

program_counter ⇐ info about next line that has to execute

Main Memory ✓

Queue<PCB>

Register

currently running

A, B, C, D

FCFS

Dispatcher

Interrupt

Signal that is sent

by an OS
↳ when I process a
↳ wants I/O

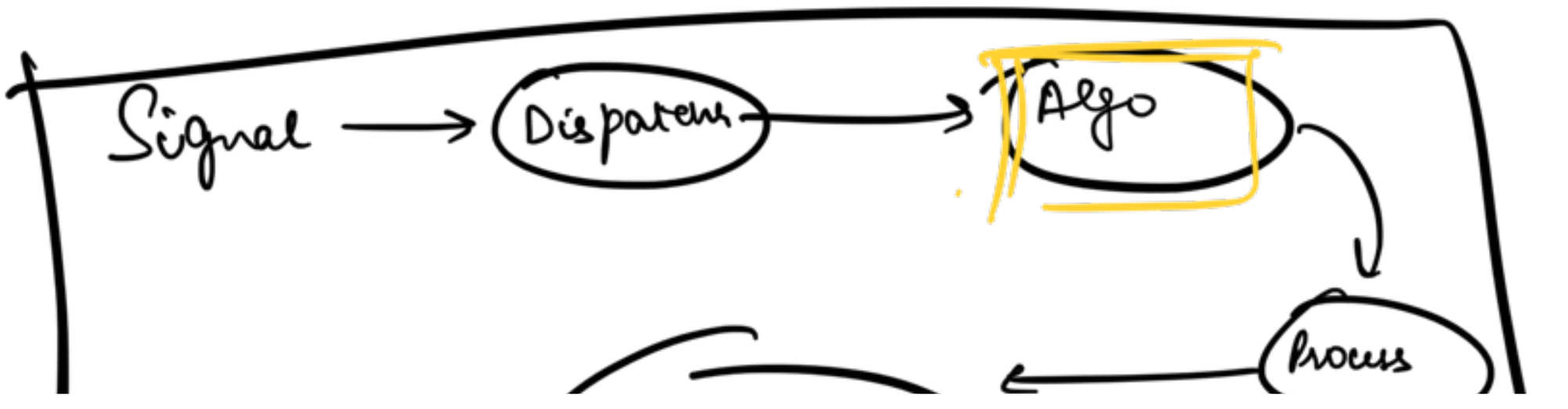
① runs CPU scheduling algo on the processes in the queue

Time taken to do this Context Switch Time

② removes the process in register

③ Move the next process into reg

↳ Context Switching



Moves process to waiting

CPU Scheduling Algos

Assumptions → No I/O in the processes

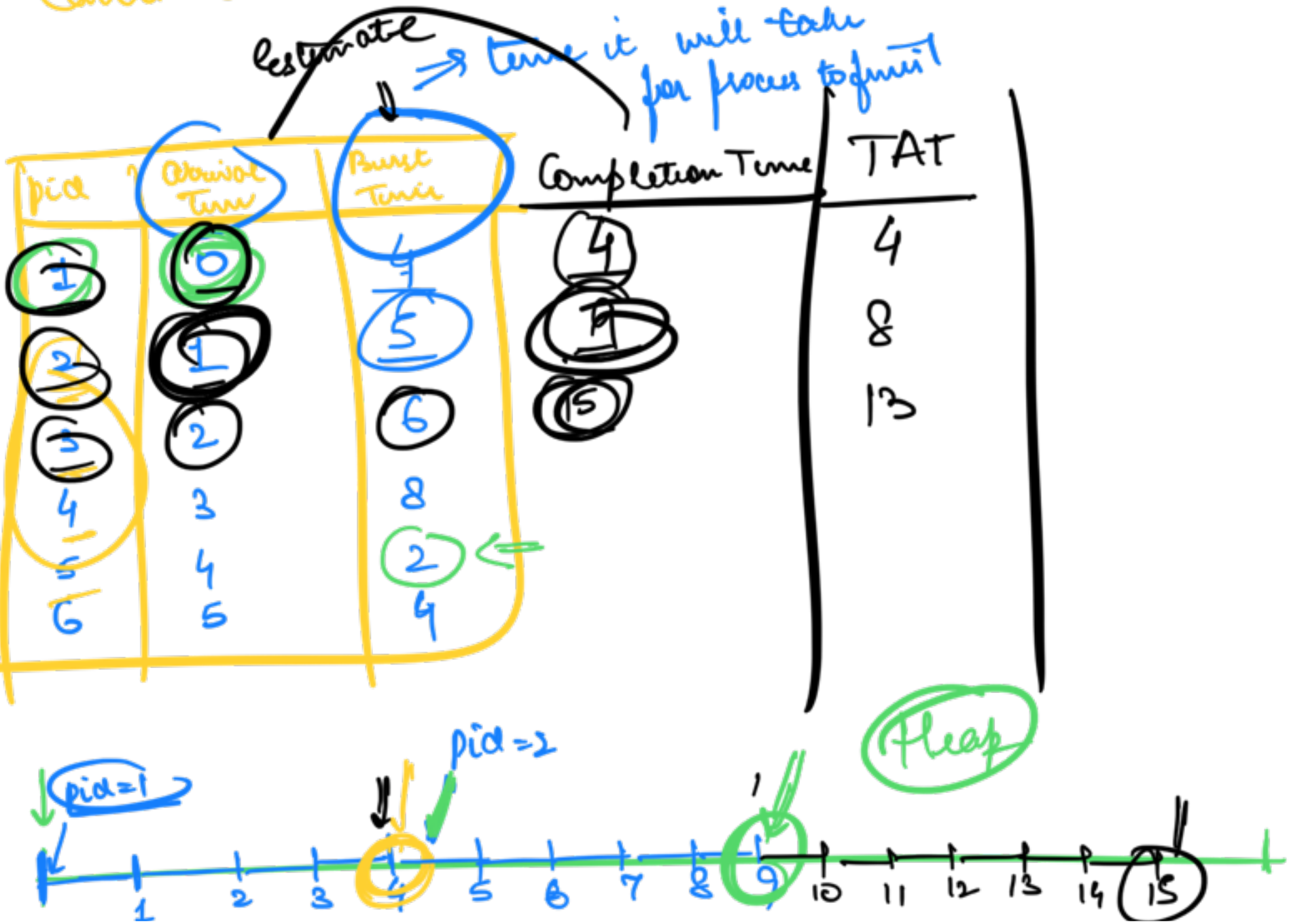
→ Context switch takes 0 time

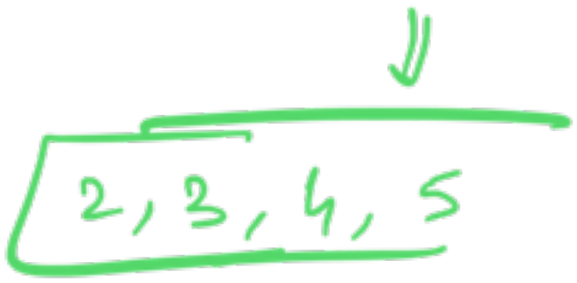
FCFS (First Come First Serve)

1. Non-Preemptive

↳ if a process is allowed to run CPU will never force stop it

Whenever dispatcher runs, out of the process in ready queue, it will pick the process which came the earliest





Boss gives you some work at t=0

deliver work at t=7

TAT

SJF : Shortest Job First (NP)

When dispatcher runs, it picks the process with shortest burst time

SRTF ((Shortest Remaining Time First))

① Preemptive (SJF is non Preempt)

② It May ask a process to give back control if Needed.

⇒ Will always schedule the process which has shortest remaining burst time

7 − 3 ⇒ 4 sec

pid	Arrival Time	Burst Time
1	0	8
2	1	6
3	2	4
4	3	2
5	4	6
6	5	1

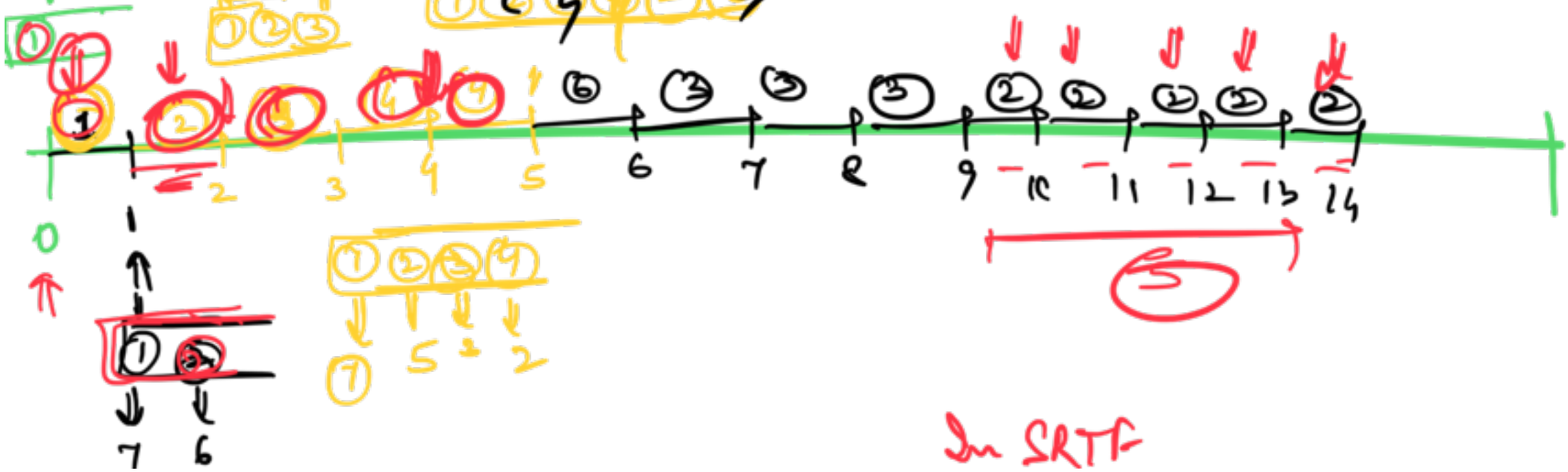
pid AT BT

SRTF

Dispatcher

1. When a process finishes
2. When a new process comes

Gantt Chart



In SRTF

RR + FCFS

Starvation : Some processes have to keep on waiting for long (starve for CPU time) because they had a higher burst time.

Round Robin

① Preemptive

② Time Quantum (q)

q = 1 sec ⇒ after every 1 sec it will change the proc

2 sec ⇒ ——————————— proc

q sec ⇒ runs the dispatcher

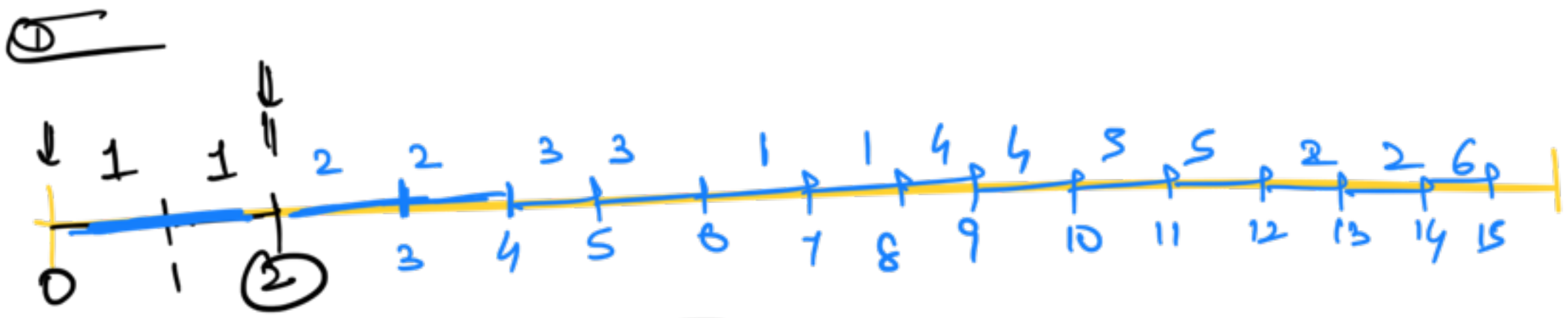
dispatcher works in FCFS

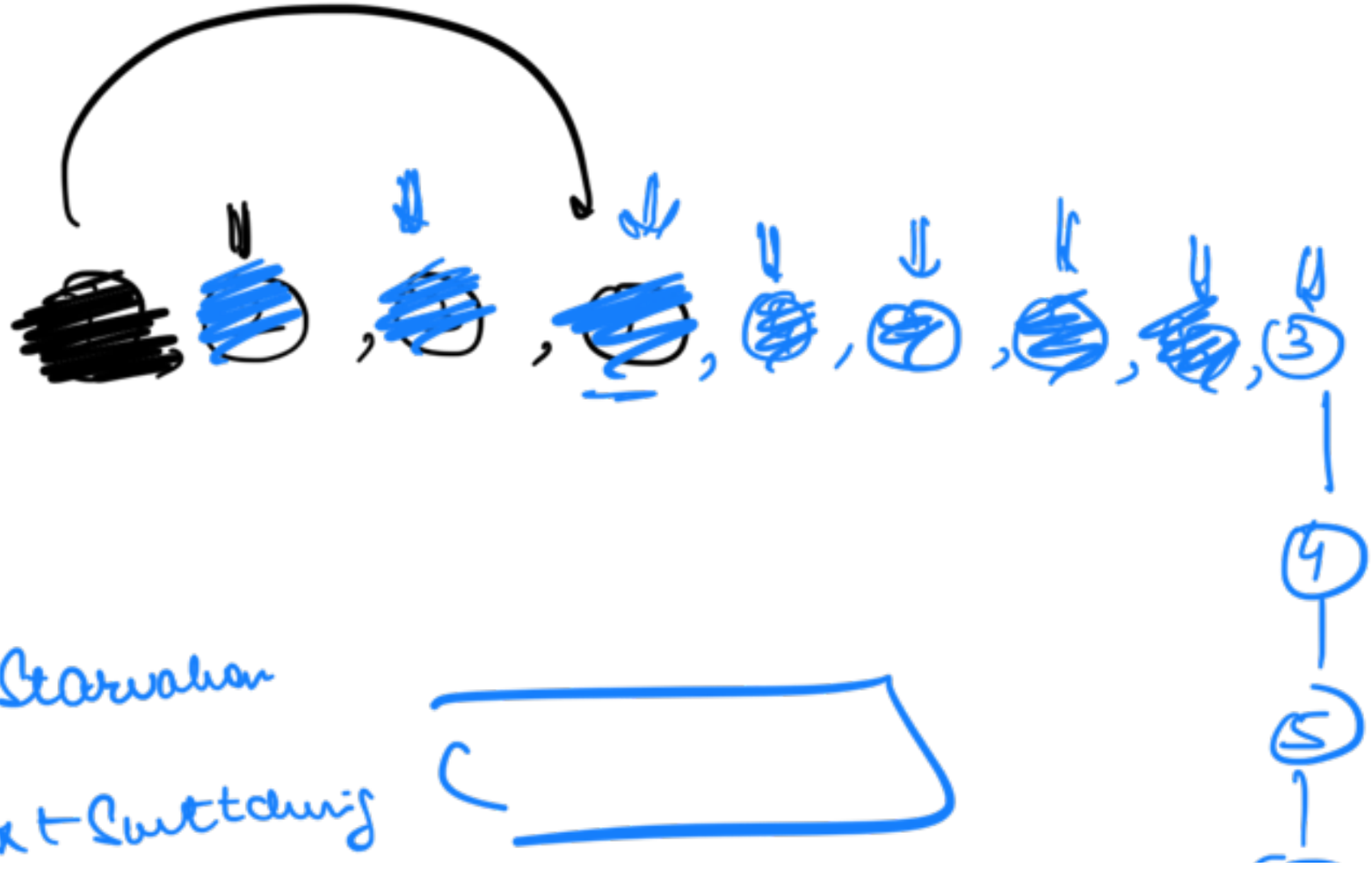
pid	arrival time	burst time
1	0	4
2	1	3
3	2	1
4	3	2

q = 2

0
1
4

Min (q.sec, remaining time)





Pro: No Starvation

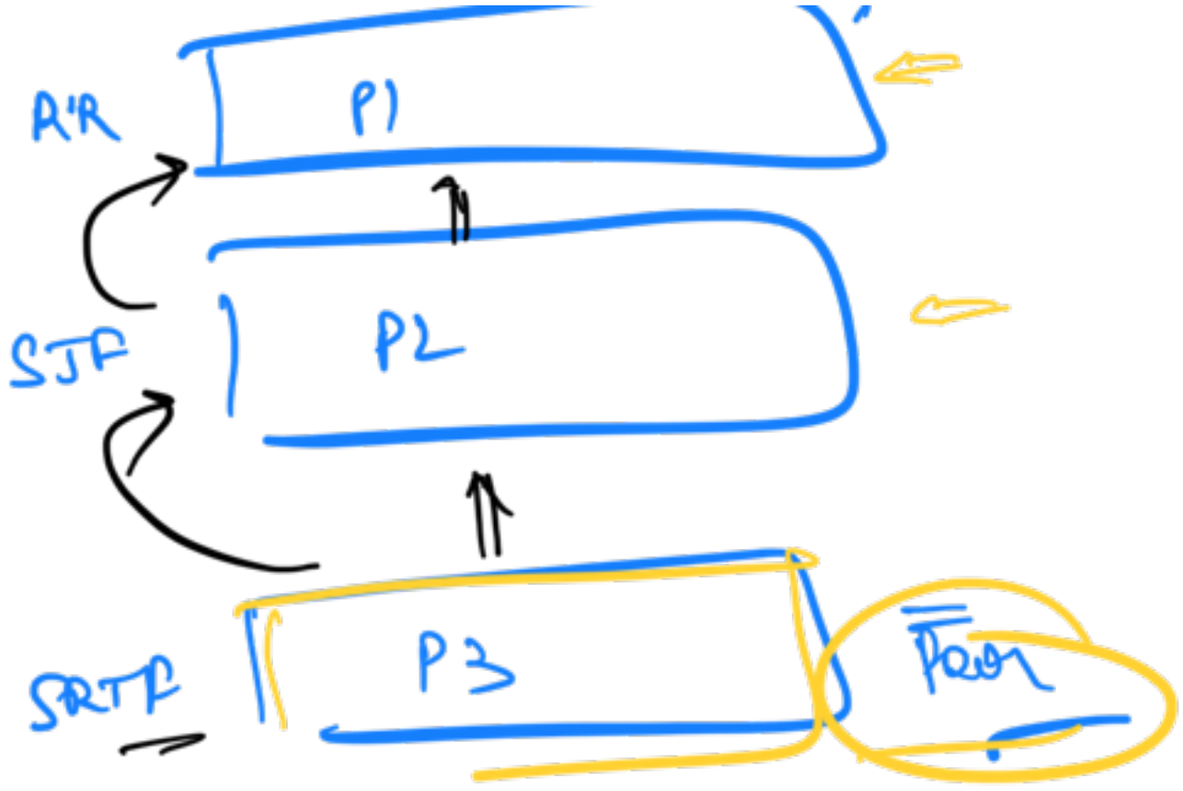
Con: Context Switching

2

Mouse Click ⇒ Priority

Shut Down ⇒ Highest Pri

JS loads ⇒ Low Pr



each q might have a diff scheduling algo

OS checks if P0 has any process → do that

go to P1 only if P0 has no p

go to P2

Multi Level Feedback Queue

Talk to ~~gf/bf~~

5 Tasks A, B, C, D, E

~~Result~~

Preparing for end S

2 Month

OS Kernel → You can change Scheduling algo

core of an OS

Schedules

---

① Linux Kernel Developer

② Interviews

---

① fork()

② CPU Scheduling Algs

③ Context Switch

THREADS

Program : Piece of code

Process : Running Program

Microsoft Word (Program)
↳ Process

→ Handling your input

→ Grammar Check

→ Checking for updates

→ Auto Completion

→ formatting

Web Browser

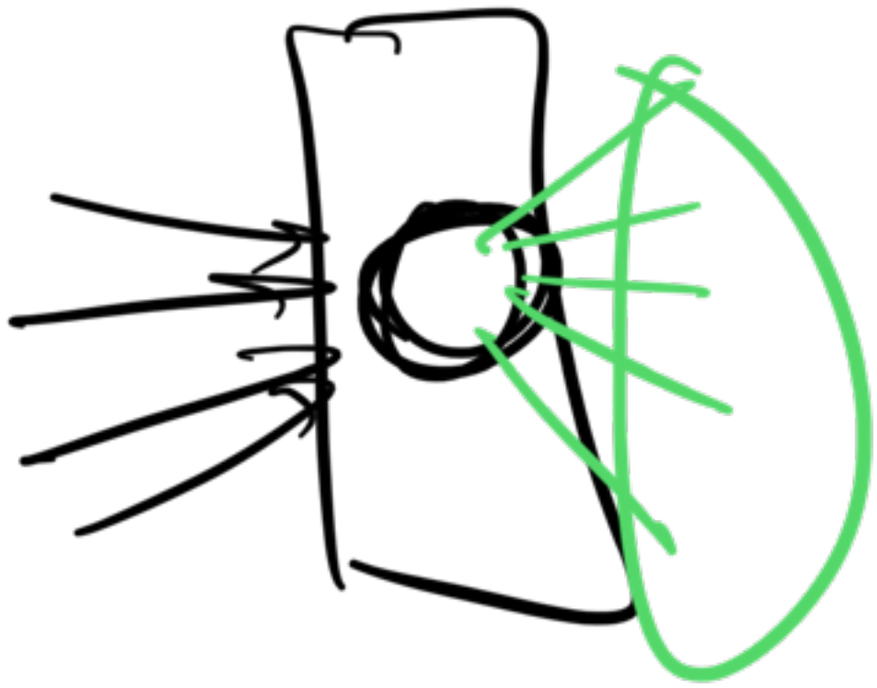
→ Handling Inputs

→ Network Call

Web Server

→ Process

→ Handling multiple requests



A single process might also want to do multiple things in parallel

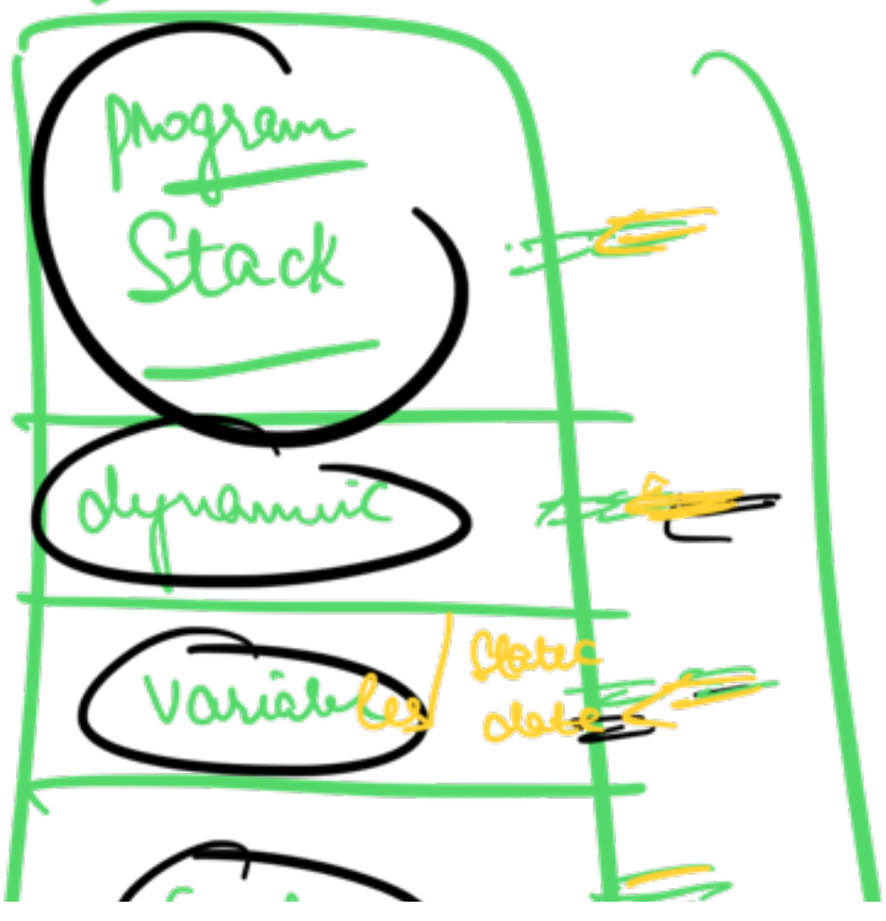
⇒ perform multiples tasks in parallel

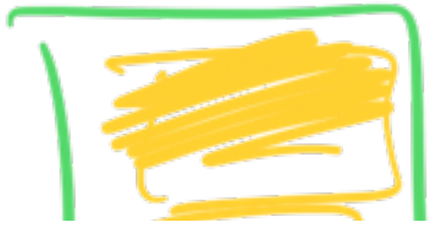
fork() Not a good idea

Why not create multiple processes

→ Creating a process is expensive

Every process has



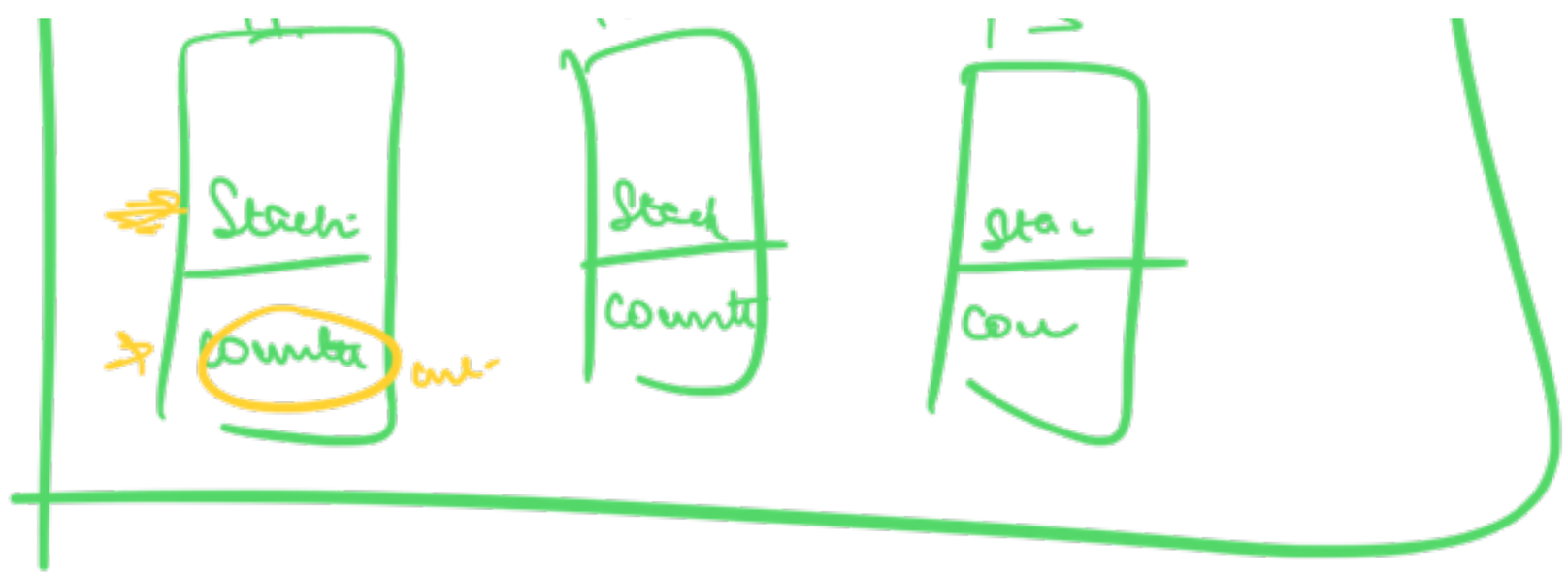


→ Every process has separate data

→ Data Sharing will be difficult

⇒ lightweight Process

⇒ Threads share their code and data with the parent process



① creating a thread is much faster than creating a process

② Make it easy to share data

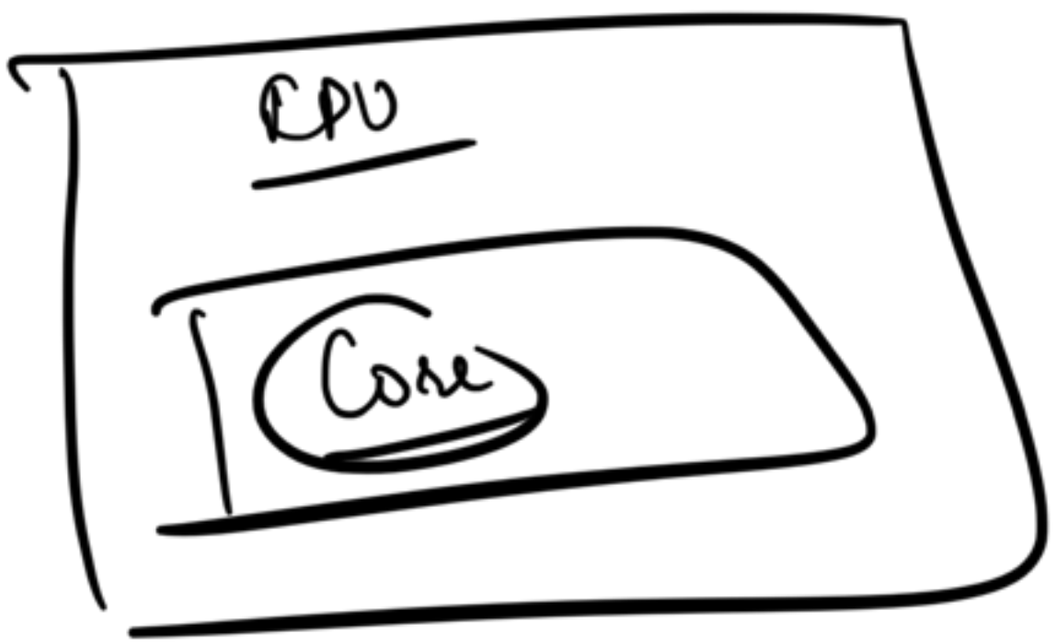
If a process has to do multiple tasks in parallel

→ create threads

Multicore Computer

Core: Computational Unit (Chip)

⇒ where calculations happen



Every process has atleast one thread

Main Thread

Thread is the fundamental unit of execution in

a computer system :

⇒ If there are multiple parallel independent tasks each sufficiently large to deserve being a separate thread

→ Semaphores

Cons of Threads

→ Synchronization

→ Debugging ⇒ Hard to debug

→ Deadlock

→ Race Cond^n

Concurrency v/s Parallelism

Parallelism: When multiple threads are making progress at the same

Time.

Multicore

Concurrency: where multiple threads are scheduled s/w scheduler

Parallelism ⇒ Concurrent

Concurrent ⇏ Parallelism

Concurrency: More than 1 threads are making progress.
(diff stage of exec^n)

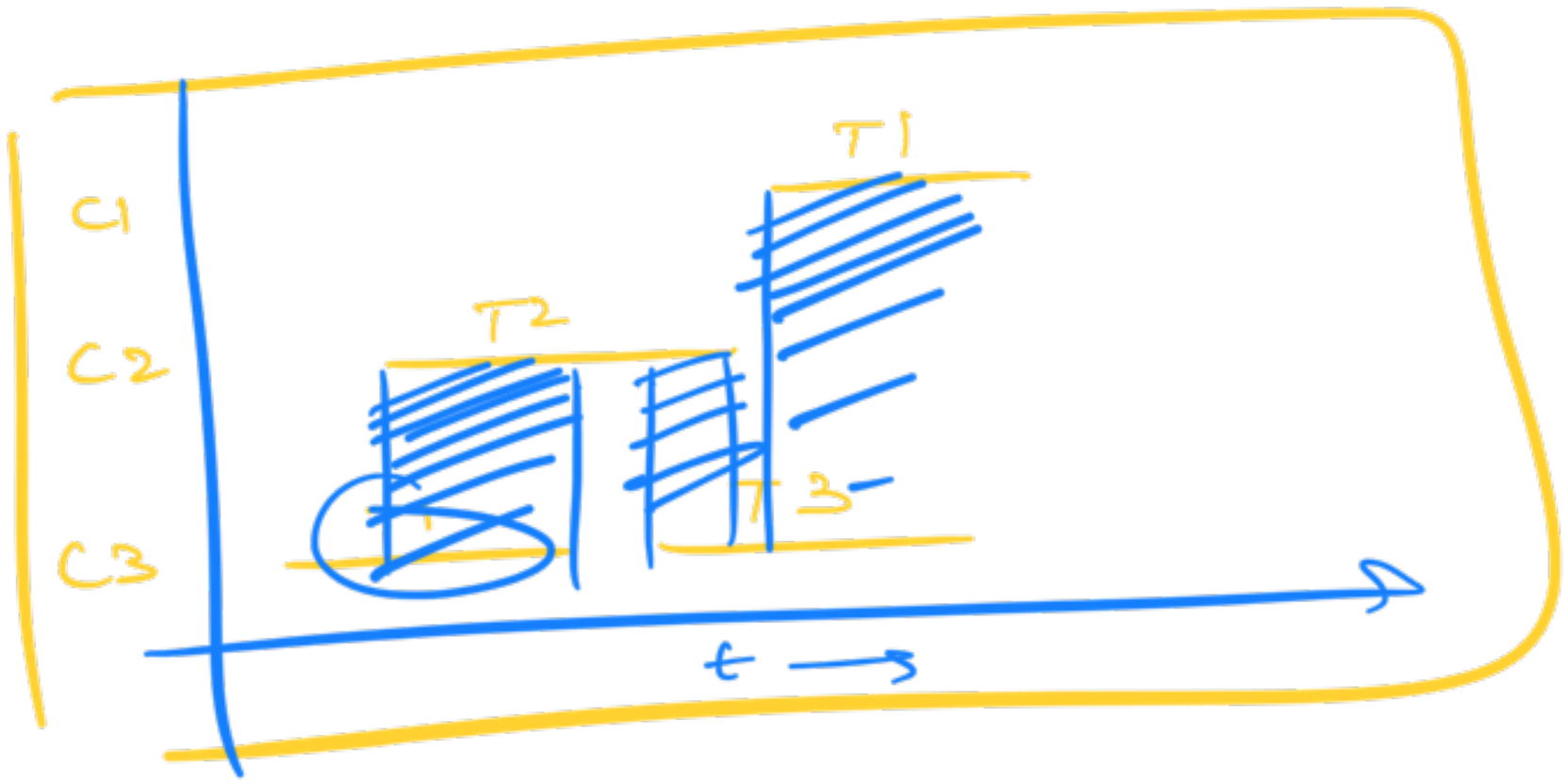
T1 → 2%.

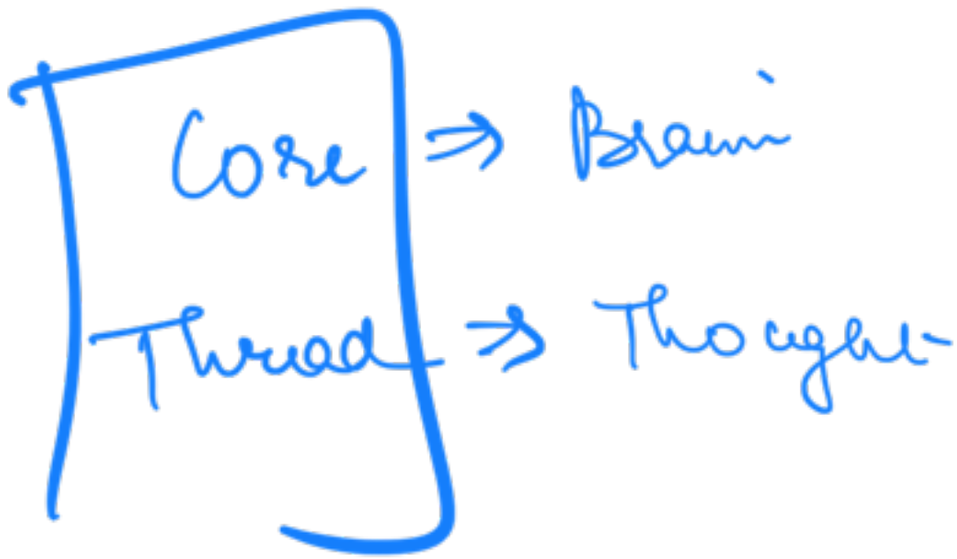
T2 → 5%.

T2

T1

Parallel





Parallelism : More than 1 thread can make progress at the same time

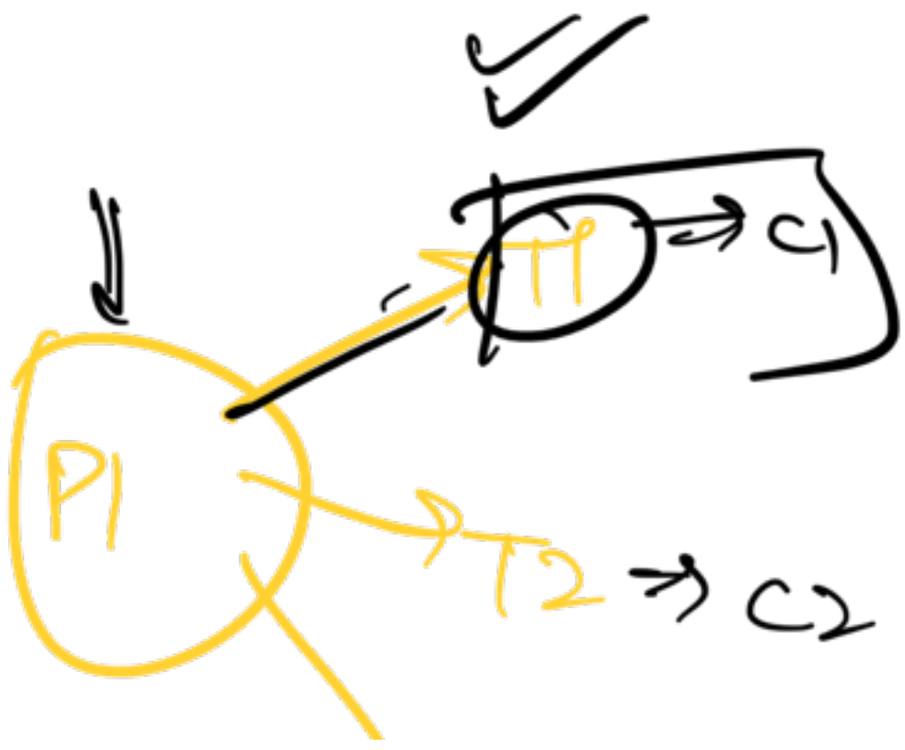
→ when multi core

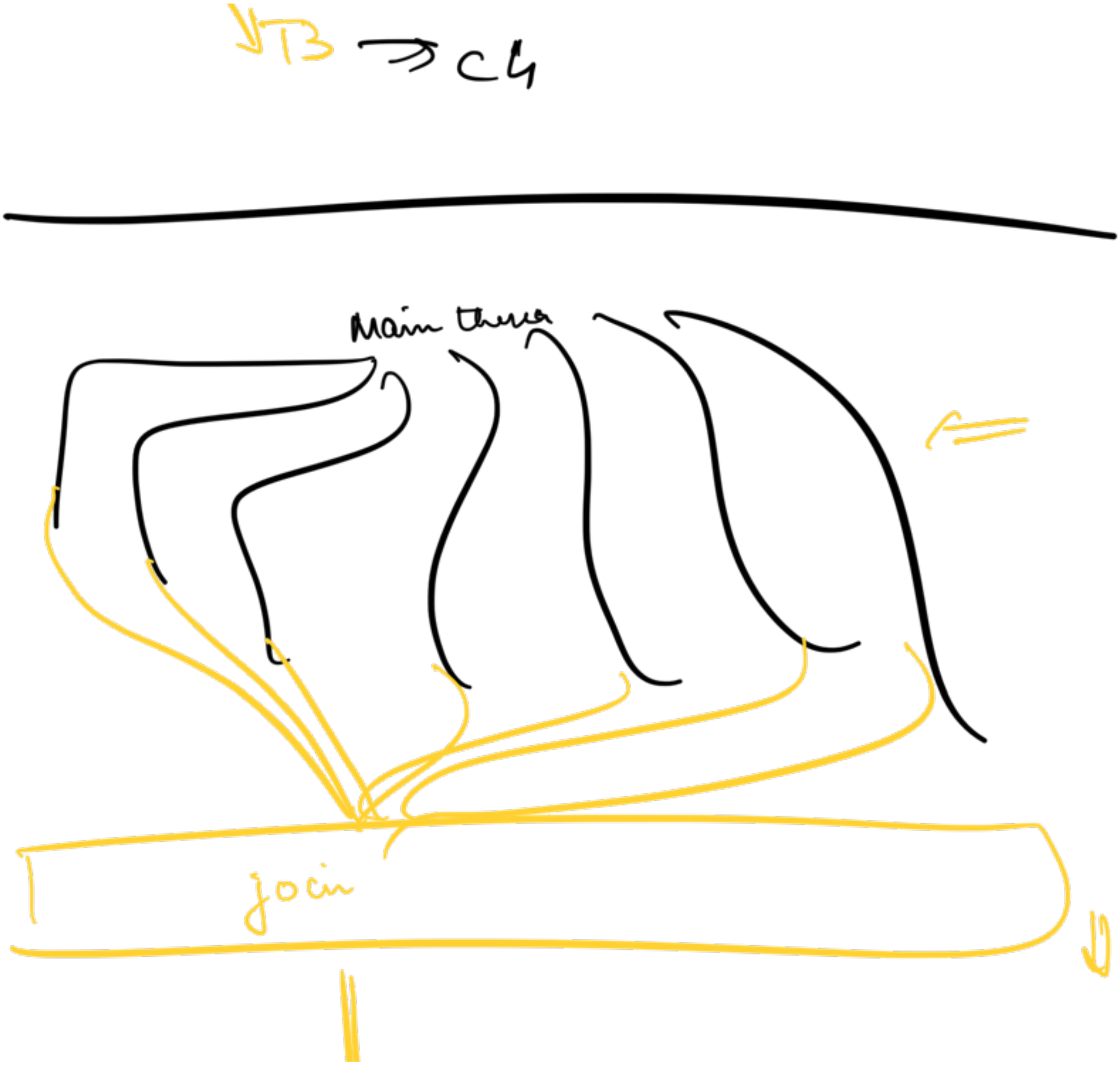
Concurrency : More than 1 thread can be on diff steps of exec^{n} at same time

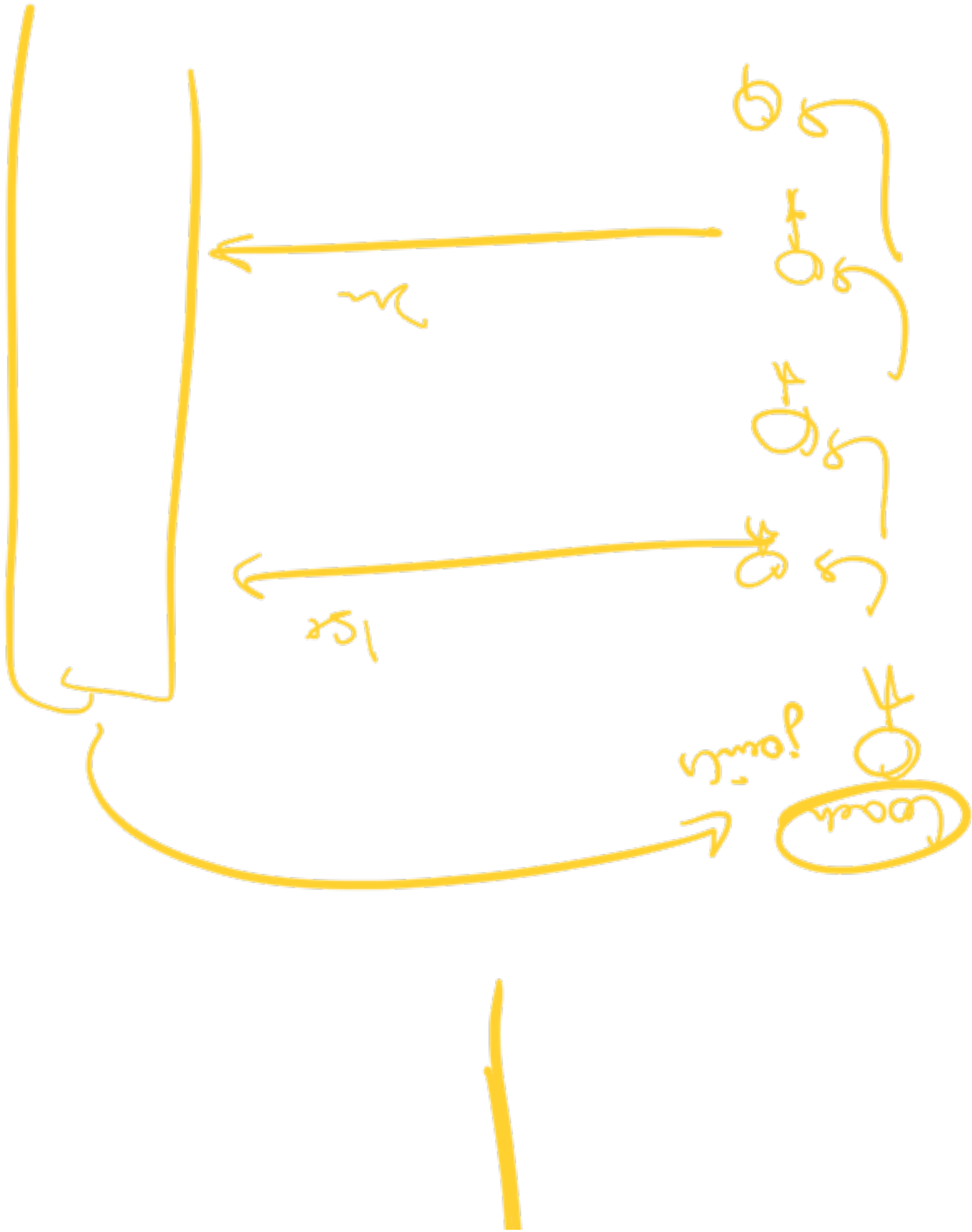
T1 ⇒ waiting for IO

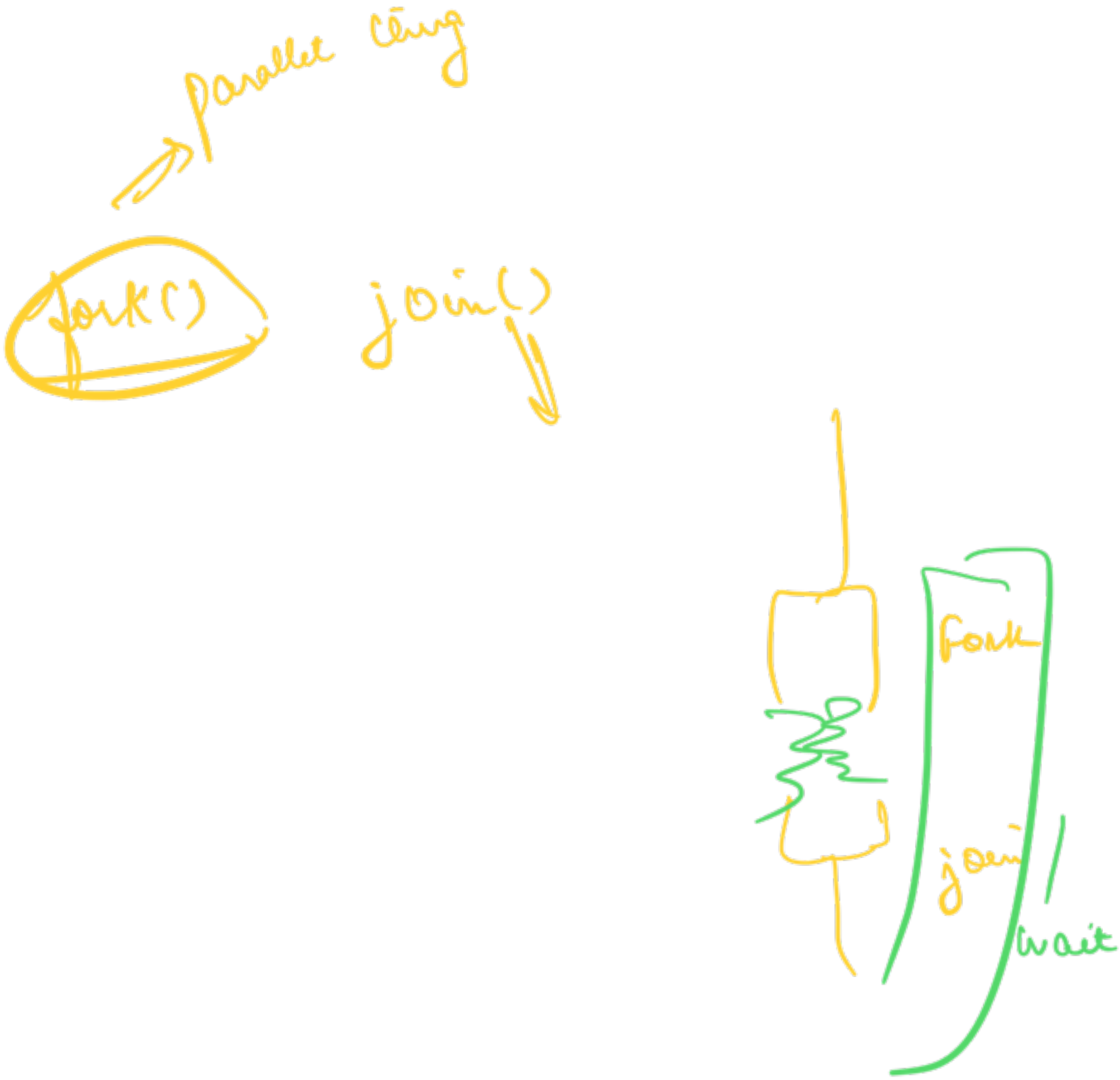
T2 ⇒ running

⇒ Only 1 making progress




Main thread
join




parallel cing
fork()
join()
fork
join
wait

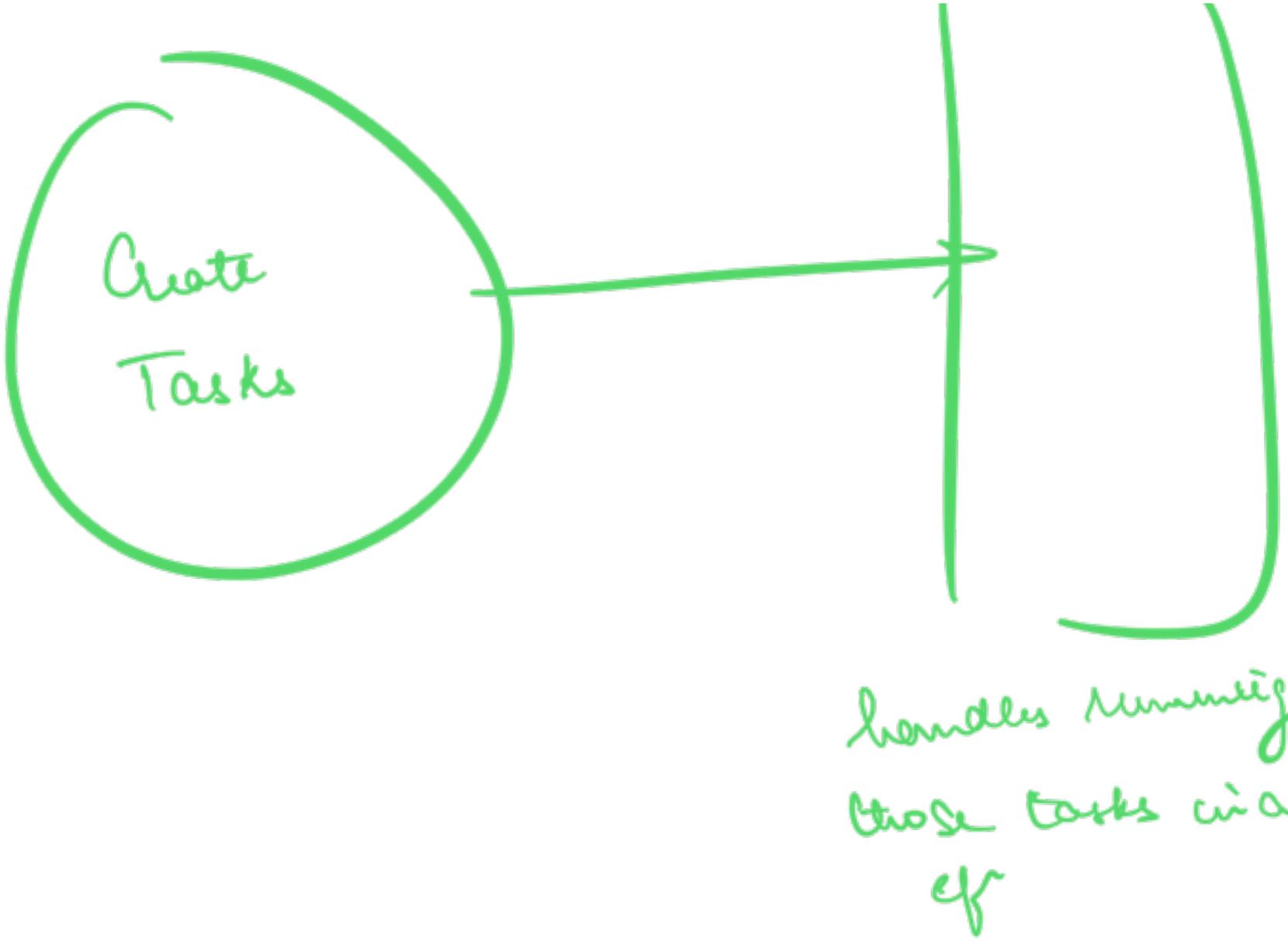
Thread Pool

Till Now

1. We created a thread
2. Started the thread

Executors

Separate the creation of a thread from exec^{n} of it

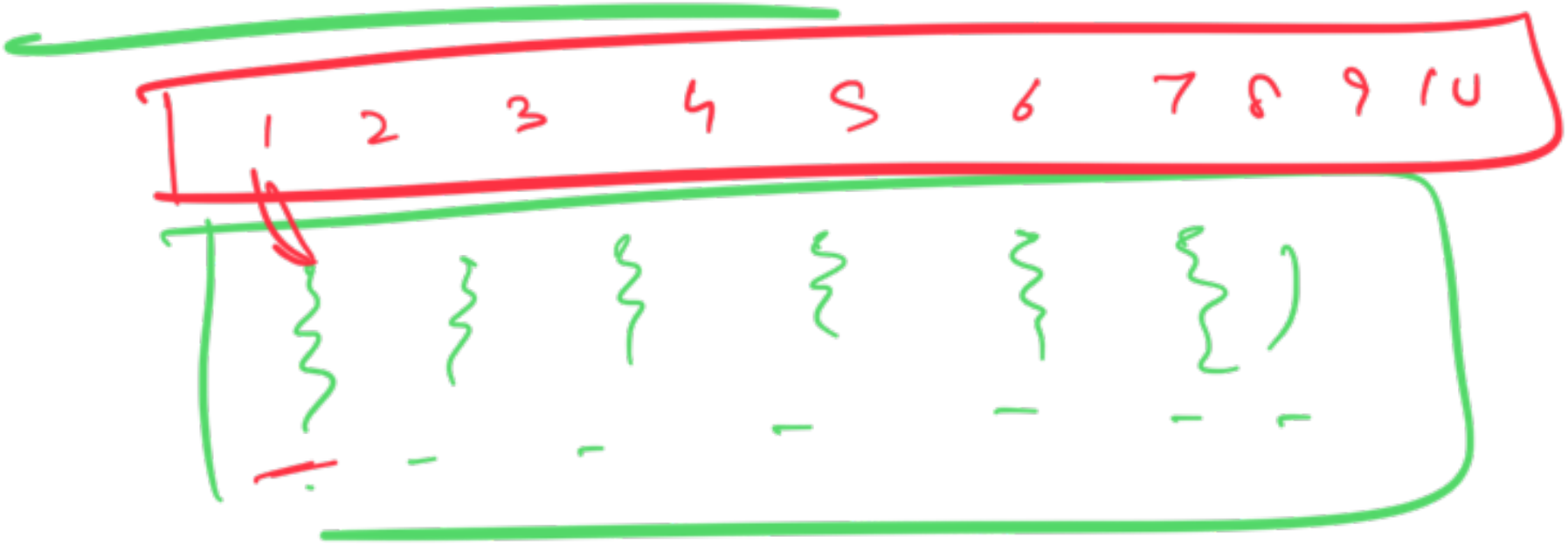


handles running those tasks in an ef

1. Create an executor
2. Give a task(Runnable in Java) to

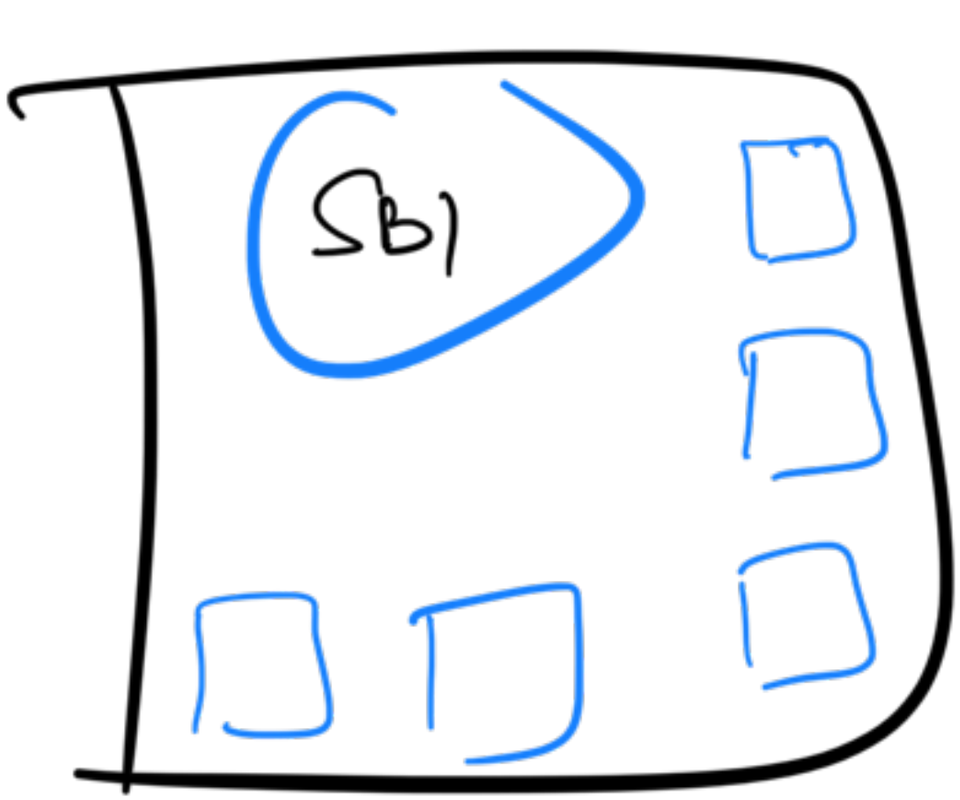
executor

Executor allow us Thread Pools



Create an executor with 5 threads


Executor
1
3
4
5


SB1

100

Executor 5

1 2 3 4 5

Query

95

CPU will Sch

parent

Pool
child 1
child 1


join
join
Only after all children are
done
(call join on every thread,