MWF → 1:30

TTS → 7:10 PM

OS → 4

CN → 4

LLD → 12

→ 20 days

→ 12 days

Java → OS1 = Process & CPU Sch

OS2: Threads & Sync

OS3: Memory Mg

OS4: File System

HLD → 16

→ 2 Months

Project Building

OSI : Introduction

→ Processes

fork()

→ CPU Scheduling

→ FCFS

→ SRTF

→ RR

Why learn OS

How app's we develop really work

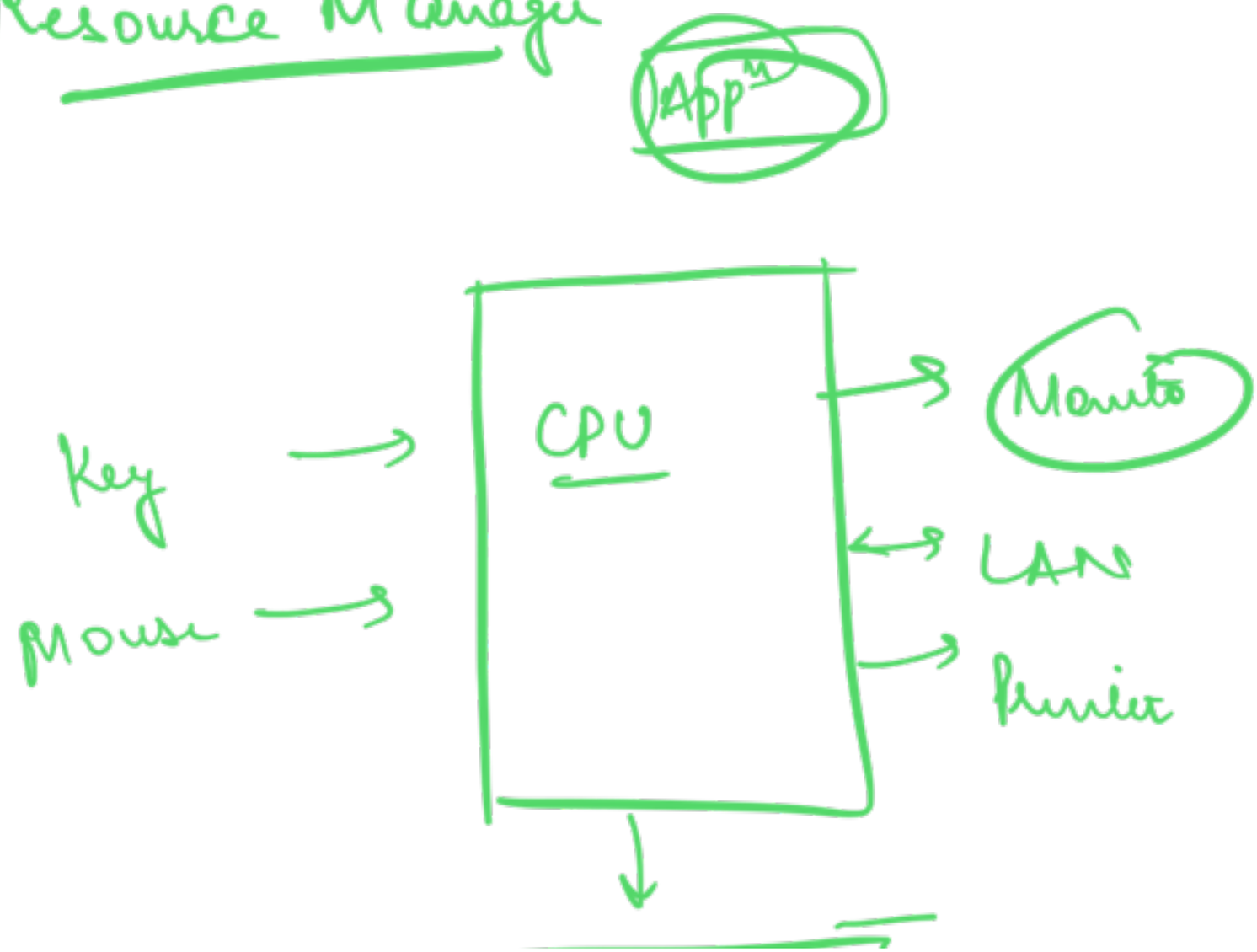
print (—)

Multithreaded

files

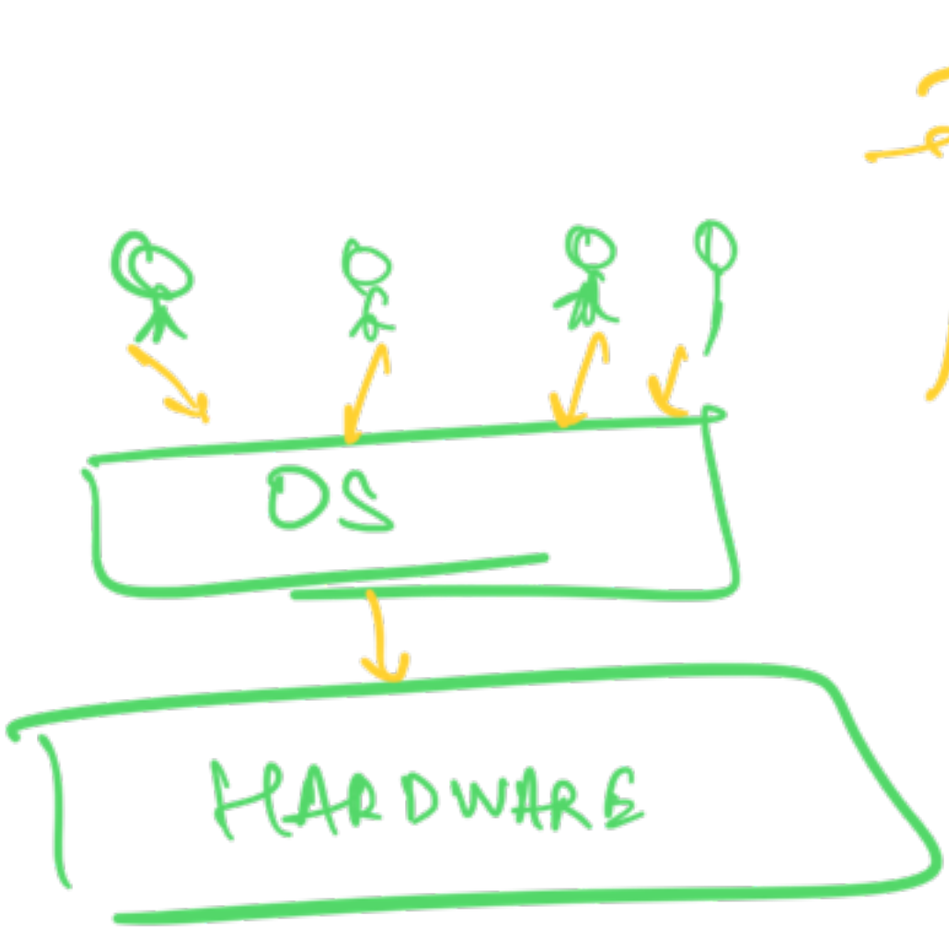
What is an OS

① Resource Manager



Memory

→ allow access to diff resources to diff appl^n
on an efficient manner



Functionalities to the appl^n developer

→ interact with hdw

→ how to interact with

→ I/O devices

→ APIs

fork()

SocketAPI

1. Resources Manager
2. Functionality to app^le develop
3. Enable users to interact with he do their day to day job

CN Client

FSAPI

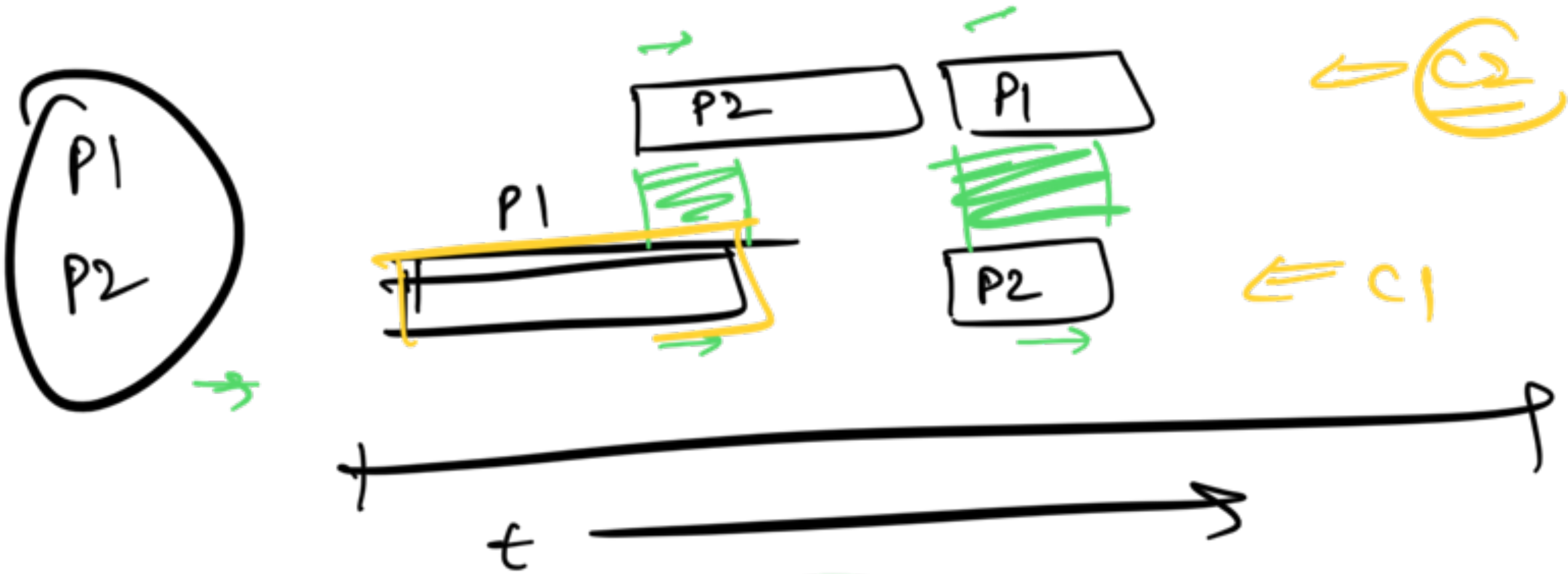
Unipro gramming v/s Multipro gramming Computer

More than one program is in execution at one time

(they might not be executing concurrently)

(MORE [illegible] phase of [illegible])

concurrency = 1



More than 1 prog making progress

Concurrency : More ... at same time

Multitasking V/S Multiprog

NO DIFFERENCE

Multprog → Unix → (Linux, MacOS)

Multitasking → Window

Multiprog → CPU Use at their fullest

P1 :
int a;

P2 :
int c;

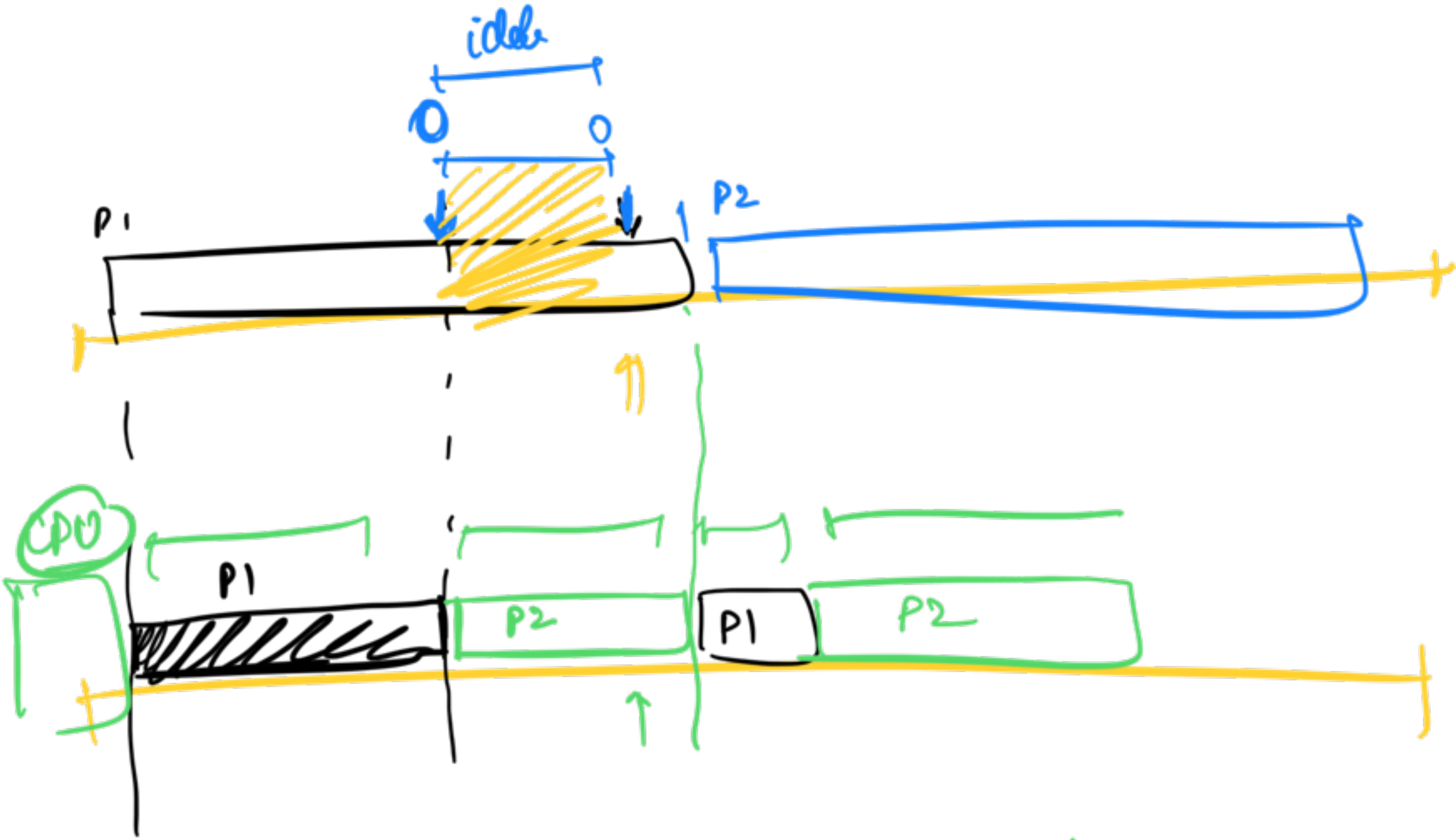
int b;

int c = a + b;

→ scanf("%d", &d)

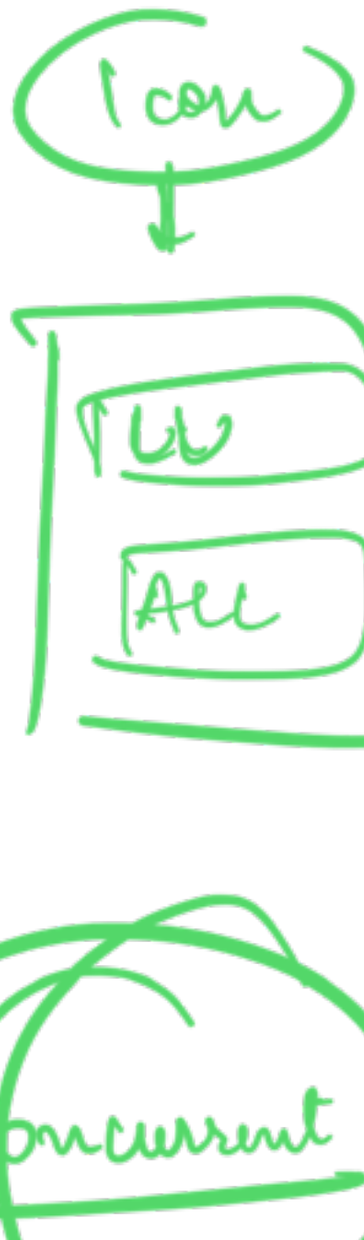
int d

int e = c + d;



My CPU is never idle

St CPU is never idle



P2

Multiprogramming

CPU core

ALU

place where all the exec happ

OS provides functionality to app to work

fork()

System Calls

Make CPU come in

Kernel Mode

SpringBoot

Python

JAVA

C++

JVM

OS

Hardware

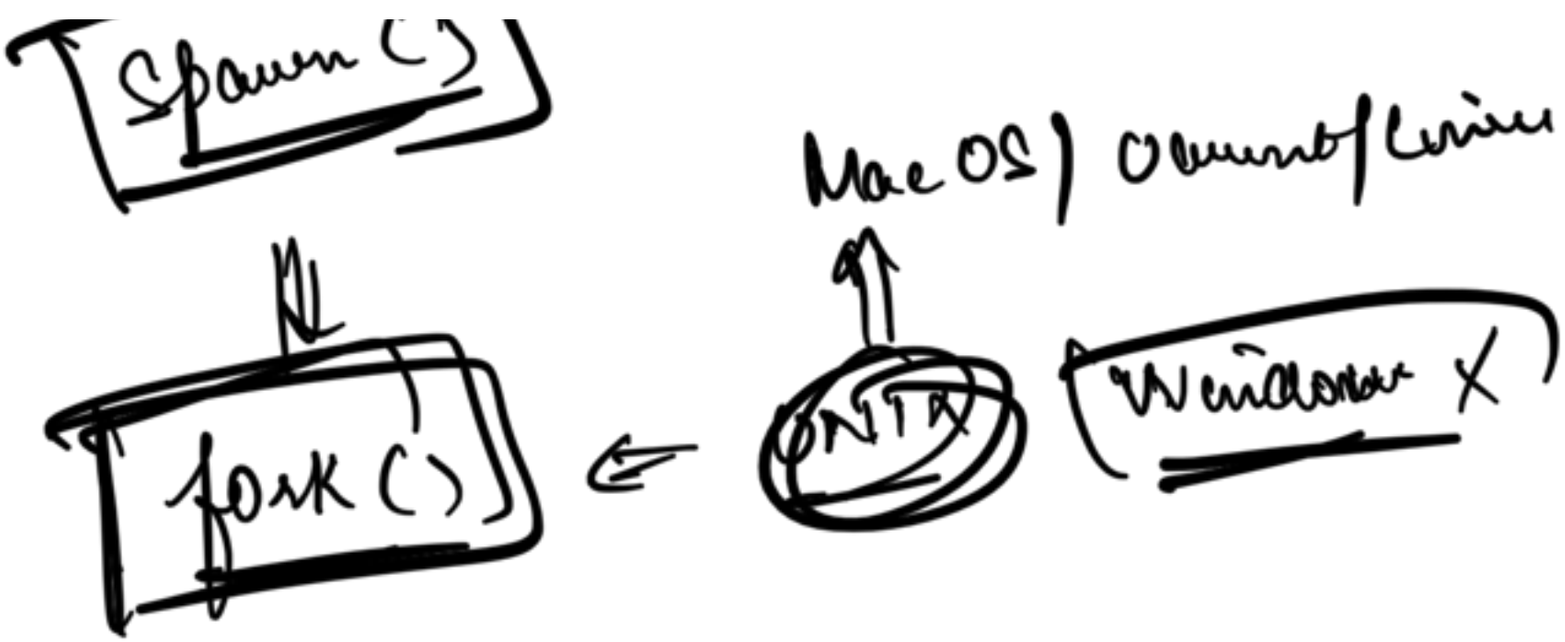
A program normally runs in one of the 2 Mode

① User Mode → Prempt that program
Appⁿ Developers / Users / Not Org

② Kernel Mode → Not Prempt
→ Key OS functionality
→ atomic execution

either the whole prog should run or nothing should run

fork() → Multithreading / Multiprocess / How OS treats



Method in unix API that allows to create a copy of a program / process

```
int main() {
    int a = 2
    int b = 3
    fork()
```

printf("Hi")

Ubuntu

Android

Mac

Linux

UNIX

```
int main(){
    int a=2
    int b=3
    fork()
    printf("Hi")
```

}

parent process

a=2
b=3

fork()

child process

a = 2

b = 3

hi hi

int main(){

fork()

fork()

printf(hi)

}

int Main()

fork()

fork()

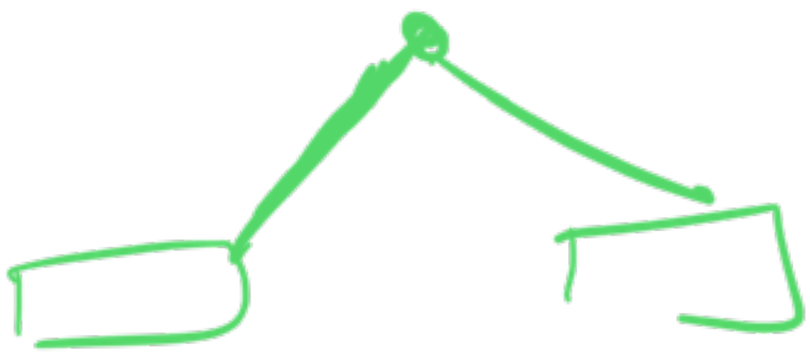
fork()

fork()

print (chi)

→ 8

fork()

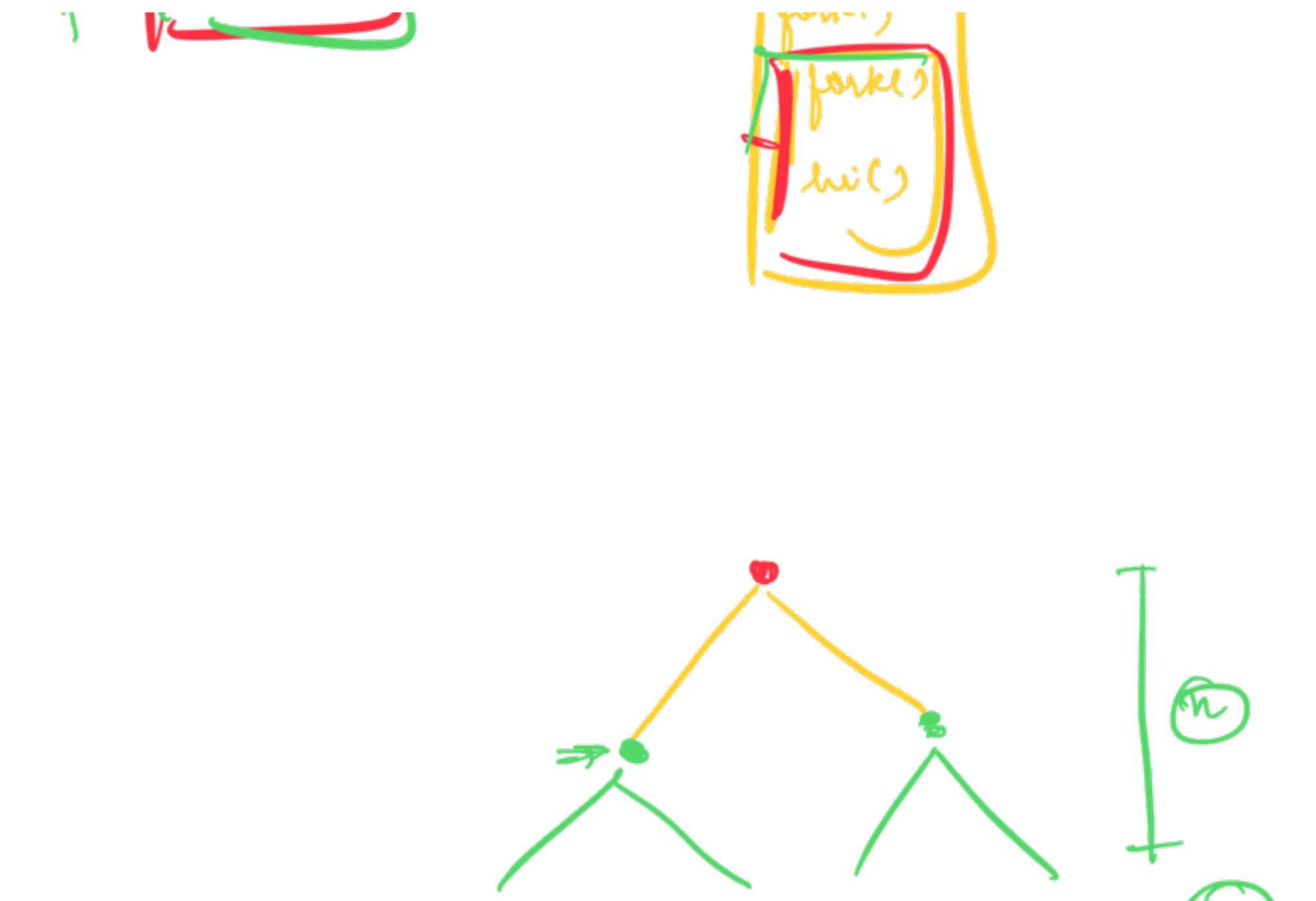


int main(){

fork()

fork()

hi

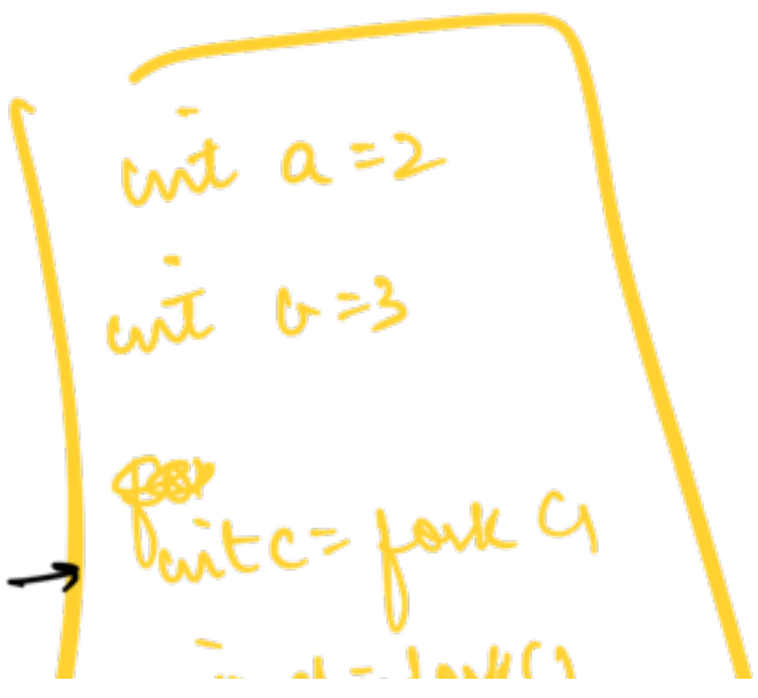

n
2^n
total

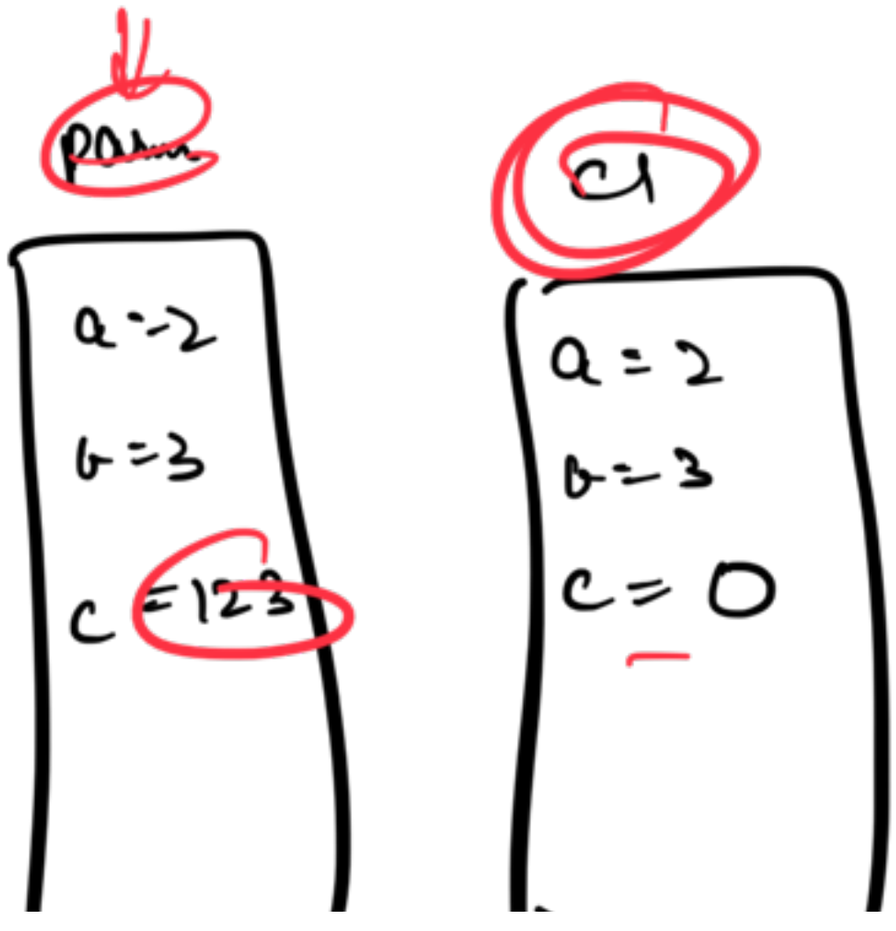
if your prog has n fork() how many child process will prod

fork() ⇒ int ⇒ process id of the new thread

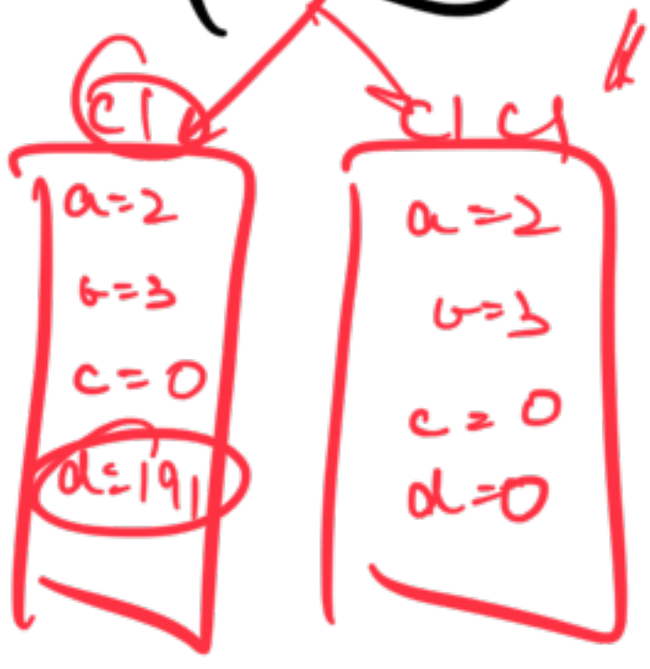
```
int main() {
    int a=2;
    int b=3;
```

```
int pid = fork();
if(pid > 0) {
    [
} else {
    [
    }
}
```



int a = fork()



int main() {

int a=2

int b=3

int c = fork();

if (c==0) {

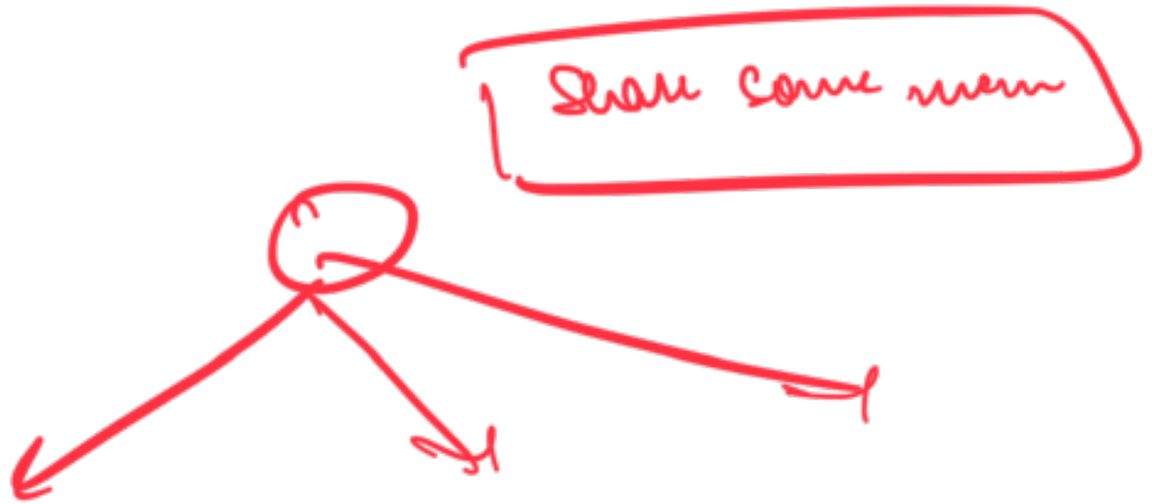
start loot

3.

if (c > 0) {

Shut down

}



parent

pid

C lang

```
while (true) {
    fork();
}
```

=

fork() ⇒ Kernel Mode

atomically create a New proc

⇒

fork()

⇒ Kernel Mode

CPU creates a new pr

copies the value

ends kernel mod

10:40PM → Process CPU Scheduling

fork() → NOT BLOCKING

wait(pid)

```
main() {
    int a = 2;
    int b = 3;
    int cid = fork();
```

1

wait (cid);

Shutdown

Process

⇒ App^{n} is stored in a disk

⇒ Process is a prog / app^{n} in execution

→ when the prog loads into mem

→ start taking resources

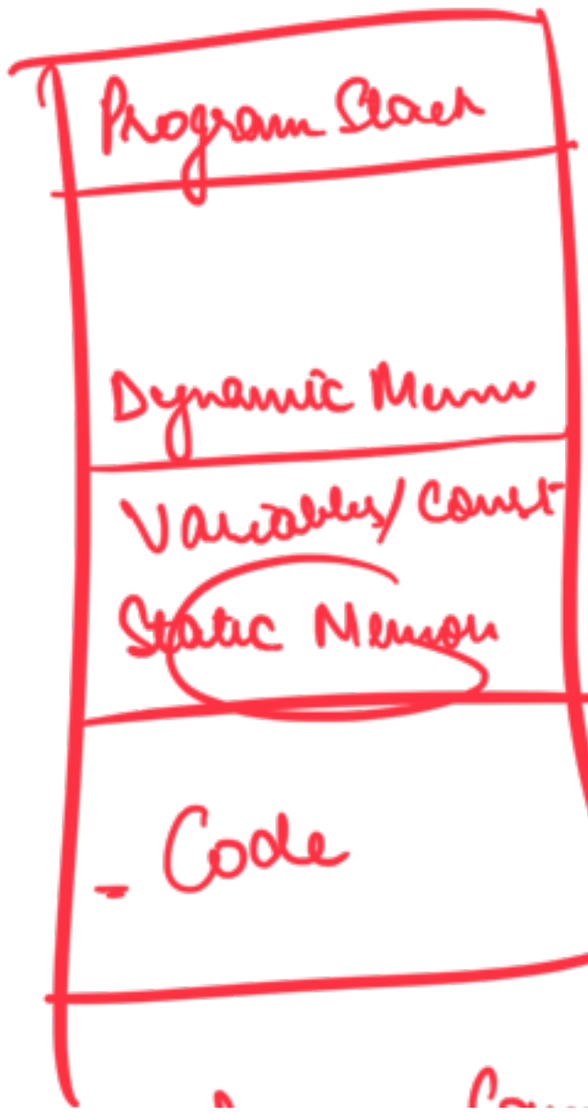
→ prog in execute

→ Volatile

Mem ⇒ RAM

Storage ⇒ SSD
HD
↳ Persistent

Process ⇒ Unit of execution in comp



When you create obj.
on heap

= new

malloc

Counter is a reference to

Program Counter

The next line of code that has
Change = 1 GB to be

⇒ Class Process {

int pid;
PC;
State
Memory info (#Stacks, Limit)
I/O Related info N/W)
files (
Protection (Mode, Access

}

→ Process Control Block
(PCB)

Struct

CPU Scheduling

CPU Bound

vast majority of work is using CPU

→ Calculations (Computations)

→ Scientific Computation

→ less waiting

VS

I/O Bound Process

→ Interactive Program

→ There will be a lot of waiting for I/O

→ I/O is ~~always slow~~ than CPU speed

I/O is always slower than CPU

CPU

GPU

processor ⇒ 2-8 GHz ⟶ 800MHz-1500

keyboard ⇒ 4

1000

How I/O Happens

input() Scanf() printf() print()

When a program needs to do I/O program raises an interrupt to the OS

(Java) print()

Signal that is emitted to the OS to tell the OS that some interruption is required by the proces

I/O → Proces raises an interrup
Connects with the resource

Starts I/O

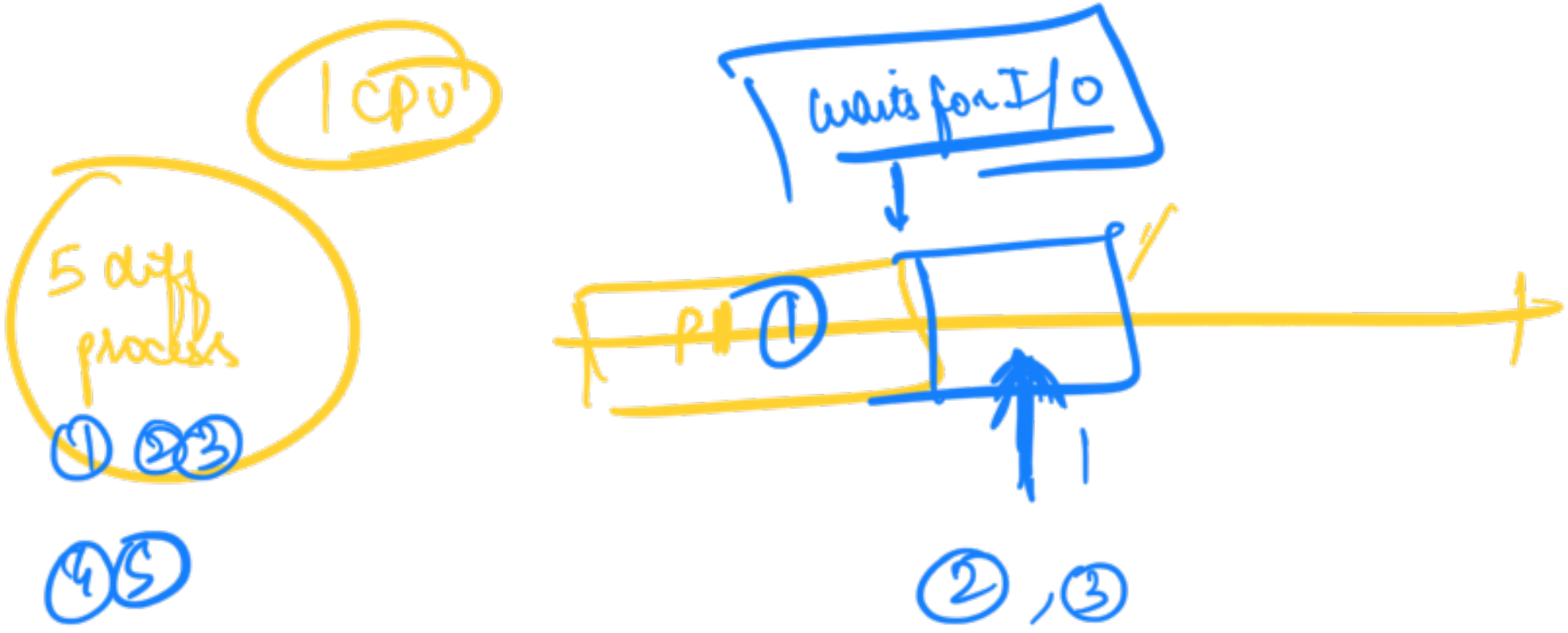
Purpose of CPU Scheduling

→ Make CPU efficient

→ Throughput : Number of process complete per unit Time

10 processes → 5 p/s

2 Sec



CPU Scheduling

Multiple ways to choose which should be the next process to be given CPU Turn

→ CPU Scheduling Algo
- → FCFS
- → SRTF
- → RR

Metrics wrt processes

① Arrival Time: The time when process came for execution

When request to run the progam can

(2) Wait Time : When the process is waiting to be run.

↳ due to i/o

↳ when no cpu

estimate the burst time

(1) length of prog

(2) past experience

(3) Burst Time : The time that will take to complete the proces

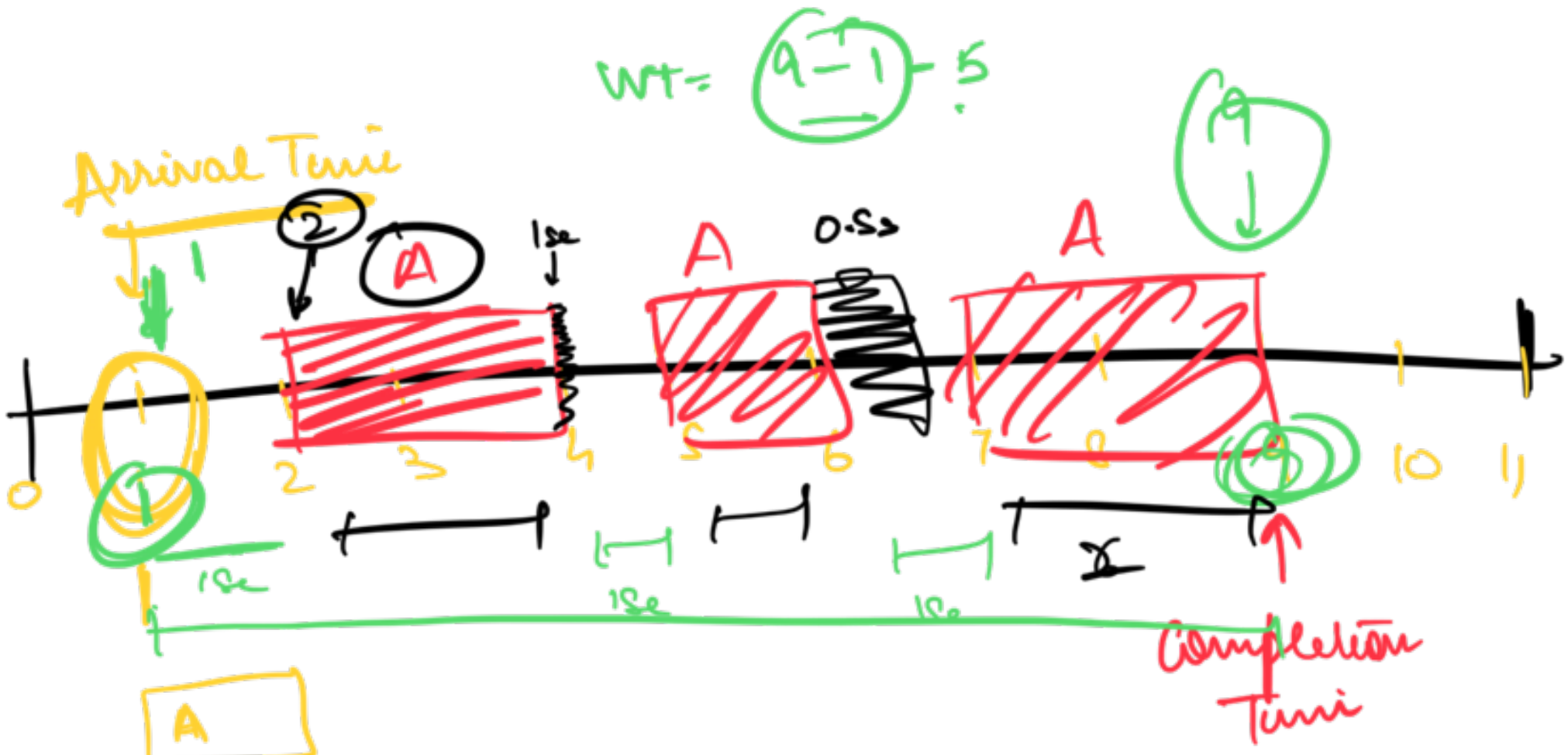
colour

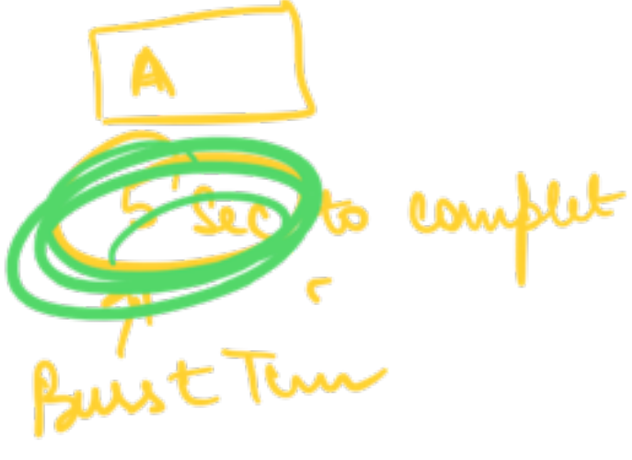
I/O

P A → 10sec

Completion Time: Time when process complete

I/O Wait Burst Time : How long will I/O take to complete





Wait Time = Completion − Arrival −

Arrival

Wait

Burst

Comput-

Burst

→ 9 - 1 - 5

→ (3)

I/O Burst Time : for I/O to finish

Turnaround Time

TAT = CT - AT

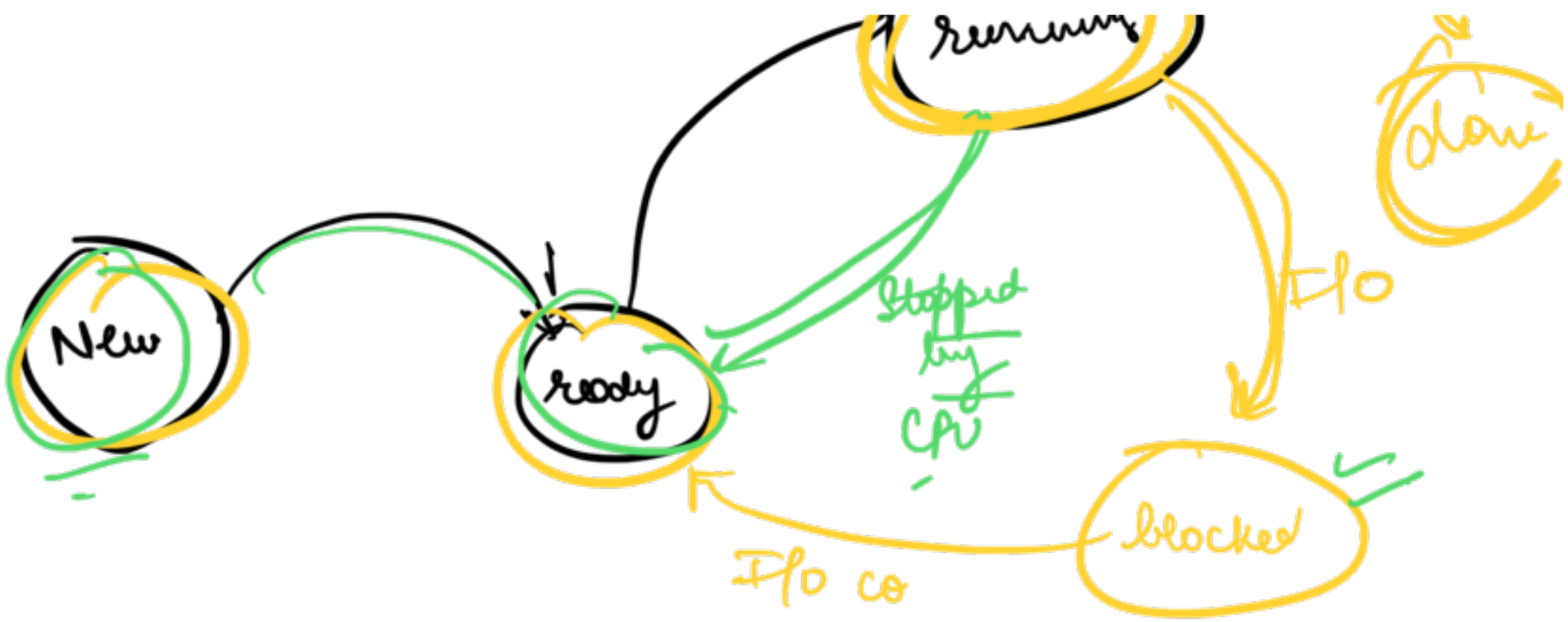
Response Time : When process gets CPU for the first time

Deadline : Time by which process is expected to finish

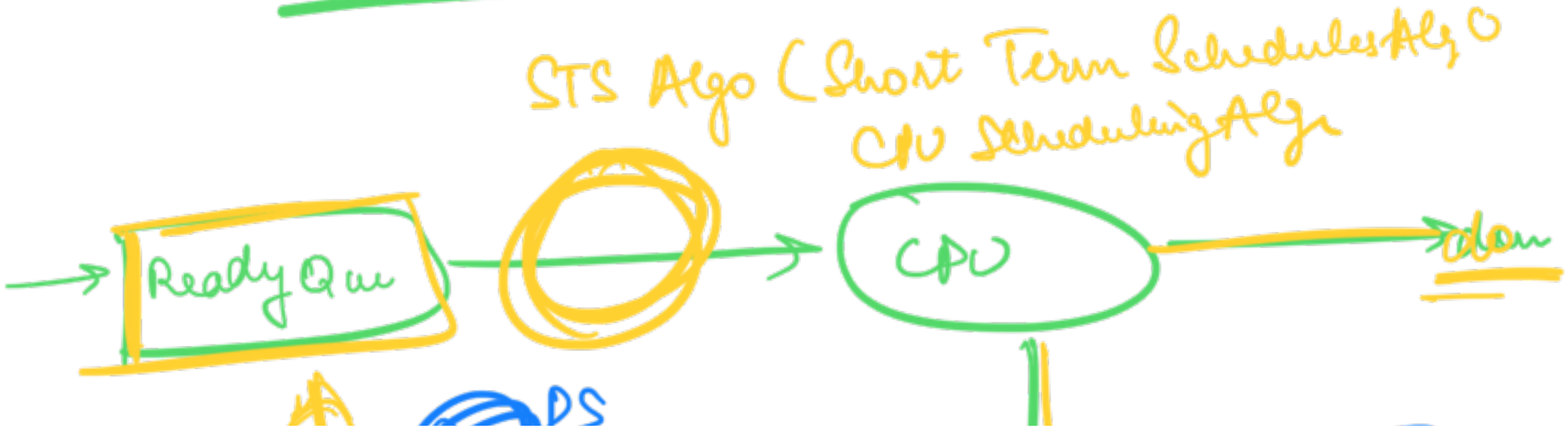
↳ Underrun : Complete before deadline

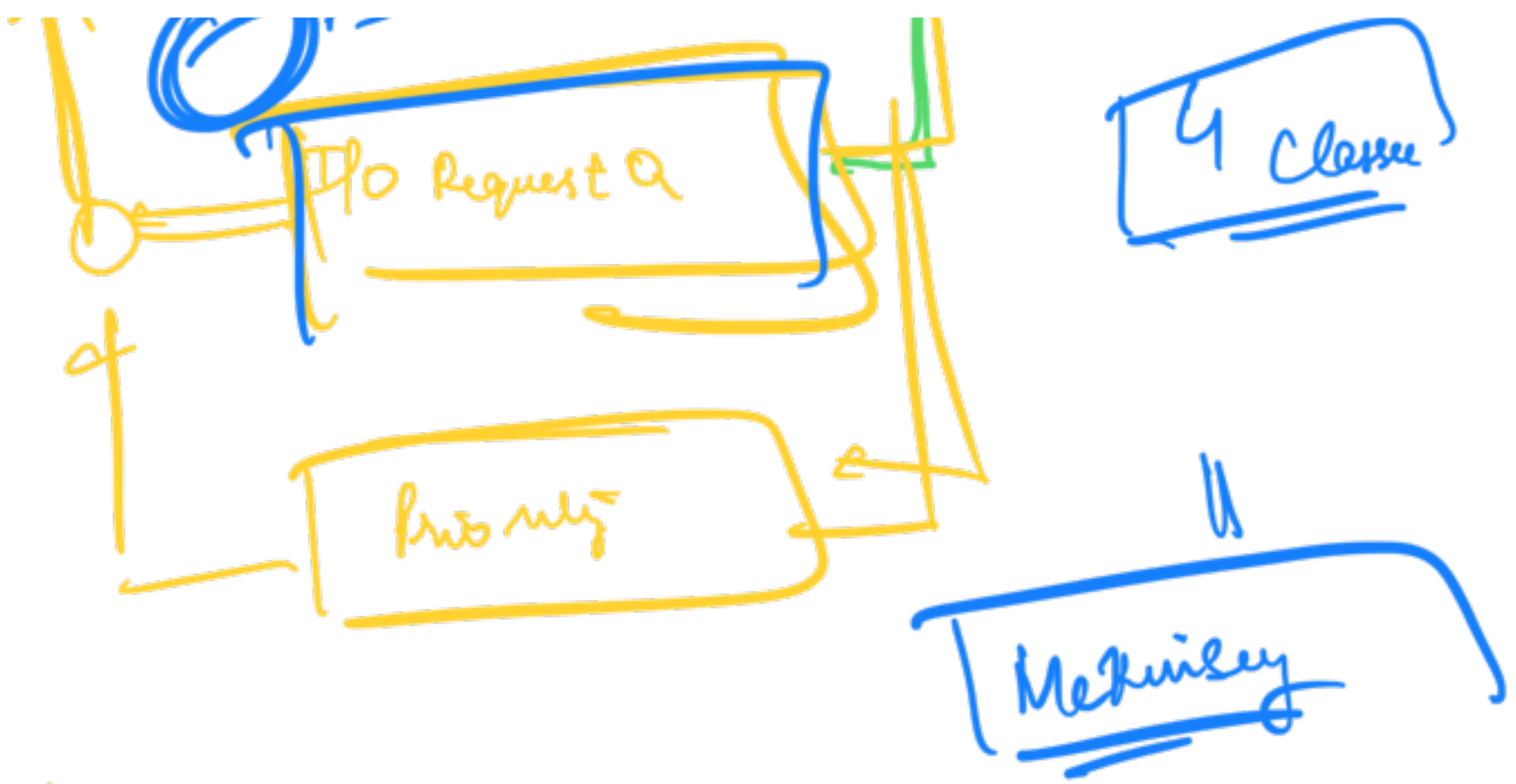
Overrun : Complete after deadline

Lifecycle of a Process



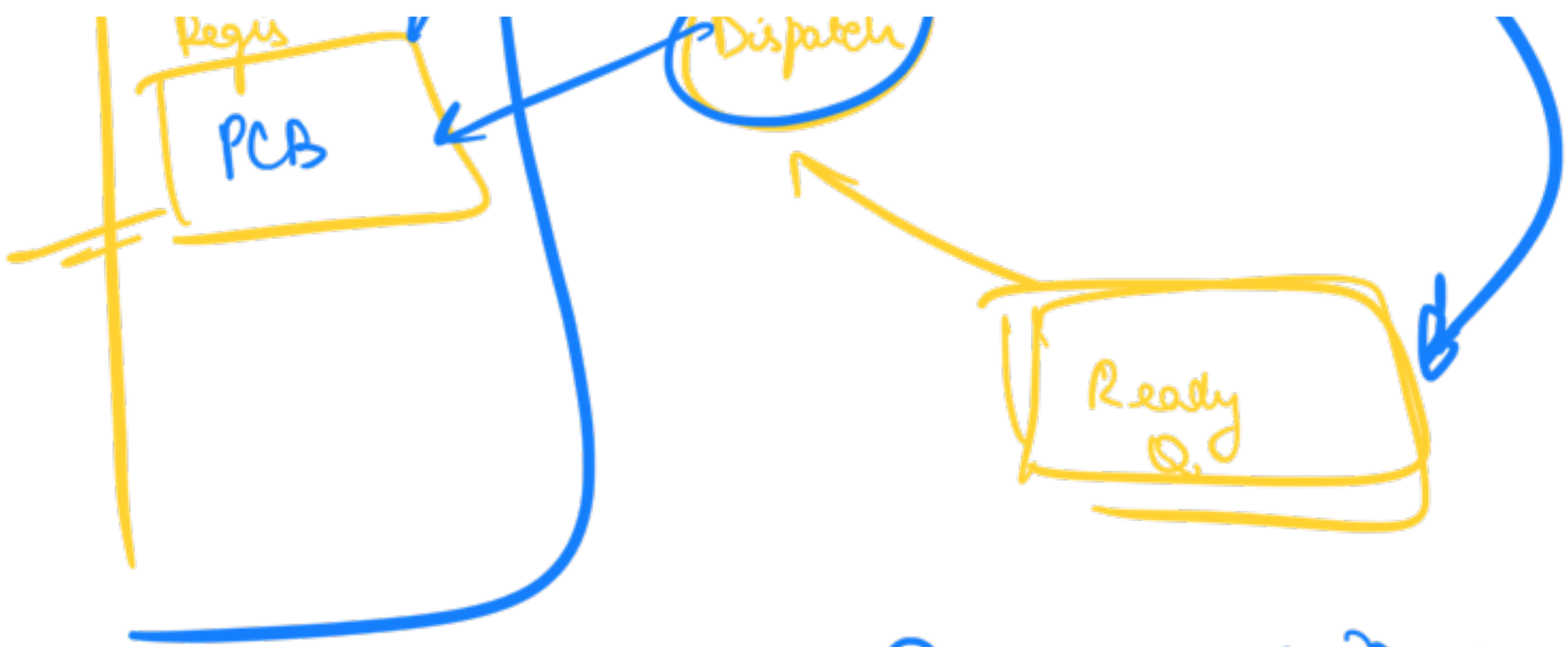
Process Scheduling Queues





Dispatcher

→ Picks a process from ready queue as per the algo and gives it to the CPU to run



Queue < PCB >

dispatcher takes the PCB from ready Q and puts it into register

S diff tw

"

FB, PT, Lelm

Context Switching

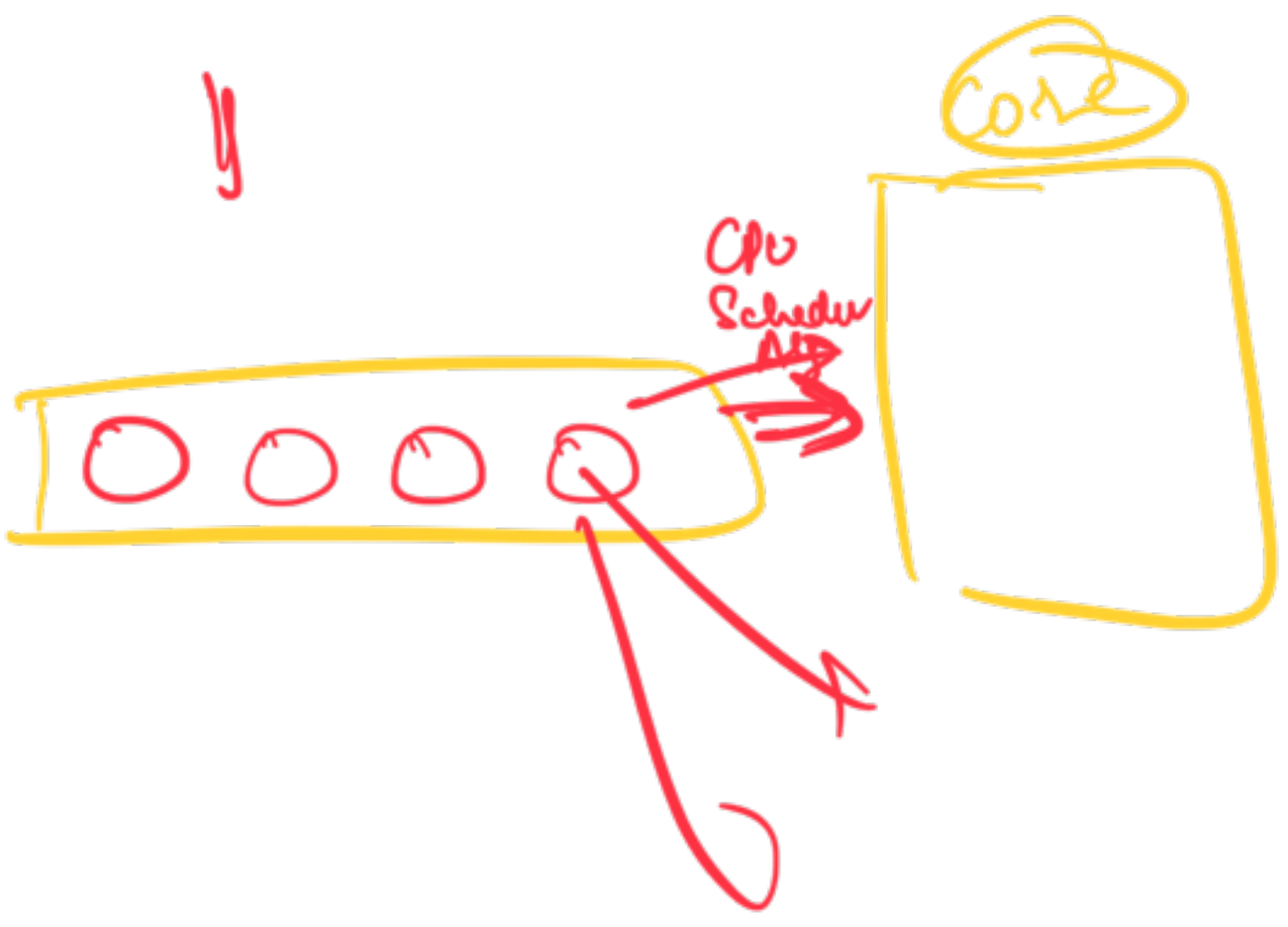
process of taking 1 task out of rg and replacing it with other task

Clear

FCFS, SRTF, RR

CPU

4 independent units of exec



CPU Scheduling Alg

FCFS

SRTF

RR

Threads & Concurrency

→ Threads

→ Core

→ Race Cond^n

→ Thread Pools

→ Mutex

→ Semaphore