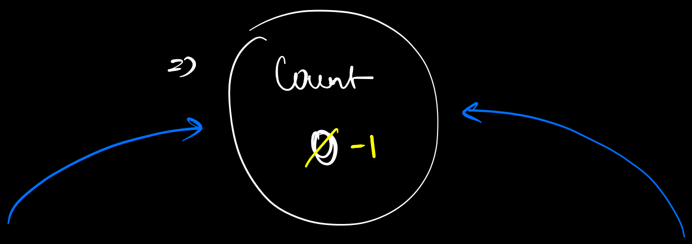
Synchronization Problem :-

Same resource being shared b/w two or more threads, and the threads are trying to modify it.

: ## Adder Subs tractor Problem

=>



(t1) Adder
(count = count +1)

i) $\overset{0}{x} \leftarrow$ count

v) x = x+1

vi) count ← x
①

Subs tractor (t2)
(count = count -1)

ii) $\overset{0}{x} \leftarrow$ count

iii) x = x -1

iv) count ← x

=> what is synch. problem ? ✓

=> when does synch. problem happen ? ✓

=> what is the ideal soln to the sync. problem ? ✓

4 solutions :- i) Mutex ✓
ii) Synchronised
iii) Semaphore
iv) Atomic Data types

⇒ When does sync problem happen?

Ans: when more than 1 thread is trying to work on a data, at the same time, it can lead to sync problem

* **Critical Section**: piece of code where potential issues might happen, related to data consistency, due to multi-threads trying to modify the data is called critical section.

There can be multiple critical sections in the code

Adder

i) print(Hi)

ii) count = count+1 ← CS

iii) print(hello)

Subtractor

i) print(Hey)

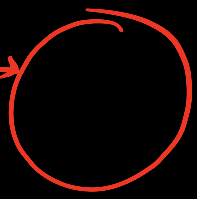
ii) count = count-1 ← CS

iii) print(Bye)

Squares

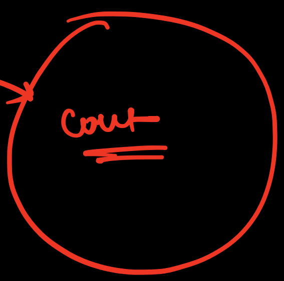
```
i)   print ("Hello  world")
ii)  count =  getCount();
iii) count =   square(count);
iv)  count =   randomise ( count);
v)   print ( Bye world ),
```

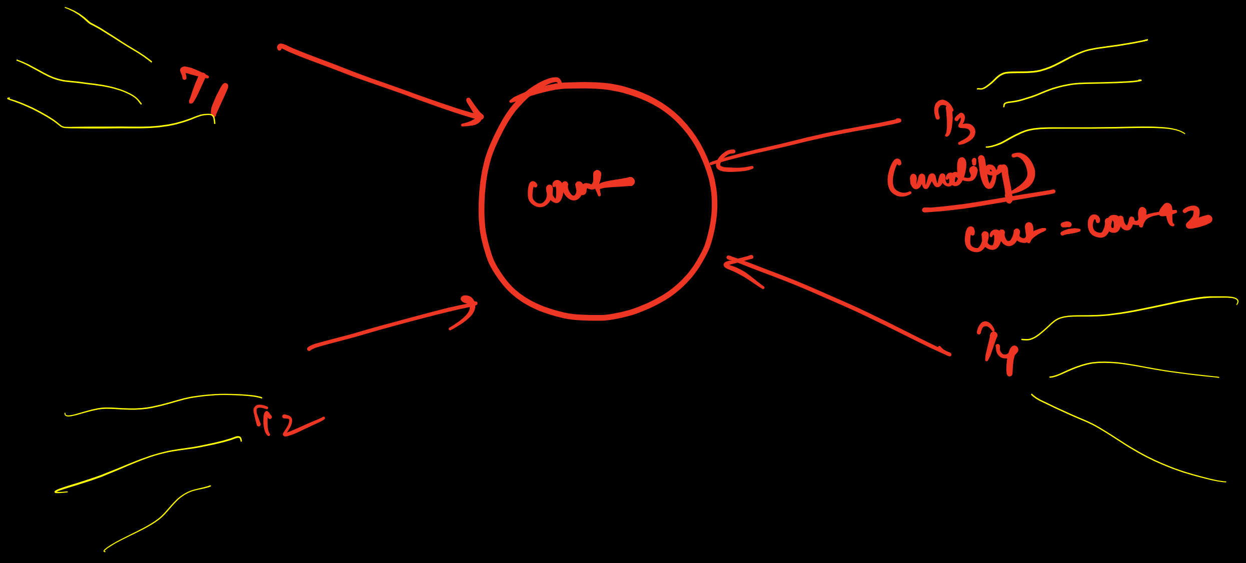
→ CS



Reader

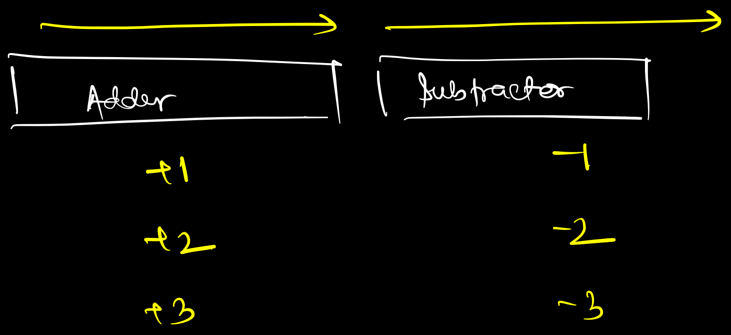
```
print ("Hi")
x = getCount( )
print (Hello)
```

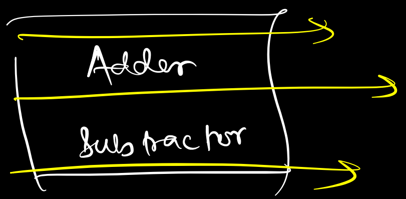




2) Race Condn :

More than one thread tries to enter/execute the critical section at the same time.





3) Preemption :

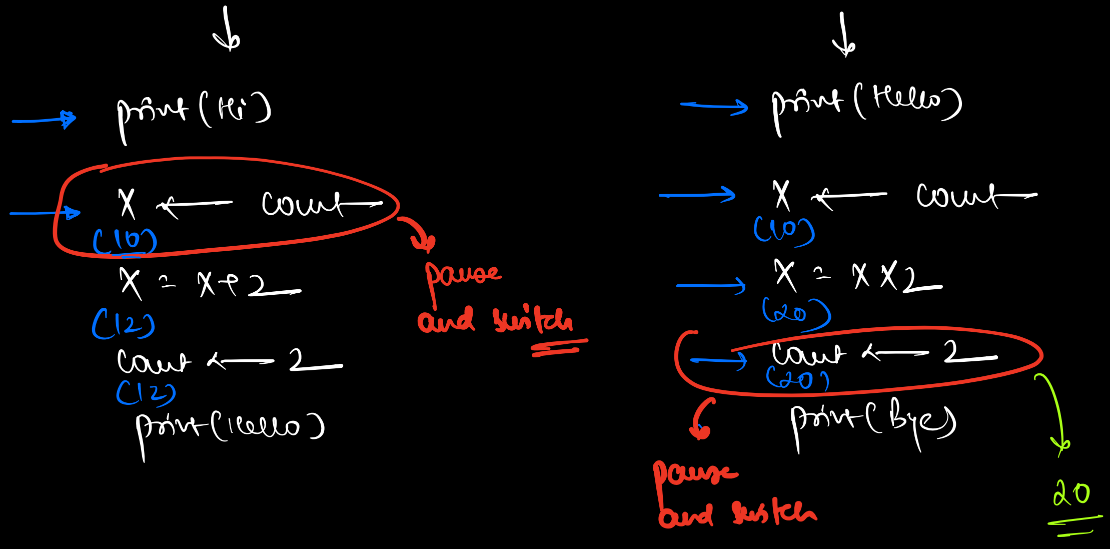
when a program, currently in its CS, and it gets preempted

=> Assume a single core CPU:-

Task 1	count	Task 2
print(Hi)		print(Hello)
count = count+2		count = count*2
print(Hello)		print(Bye)

count = 10

X → register/CPU



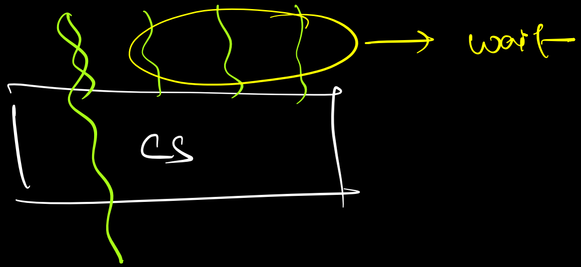
* if you get preempted by the CPU while executing (CS), then it might lead to inconsistent state

⇒ what is the ideal sol^n to the sync. problem?

: Properties of a good sol^n to sync problem :-

1) Mutual Exclusion :-

Only 1 thread allowed to enter the CS at a time



2) Progress

system, should keep on working, and make progress, entire system must not hault.

3) Bounded Waiting :-

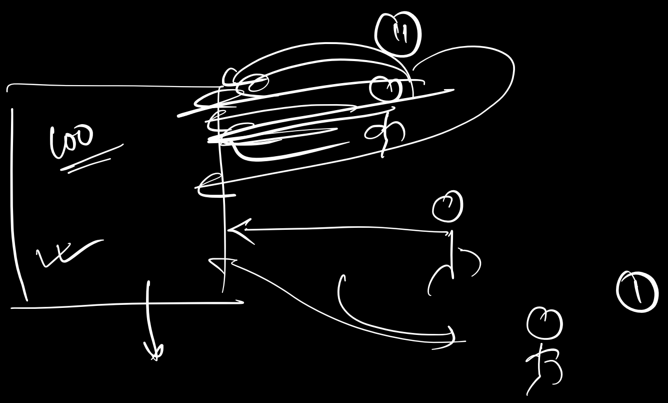
: No thread should have infinite waiting

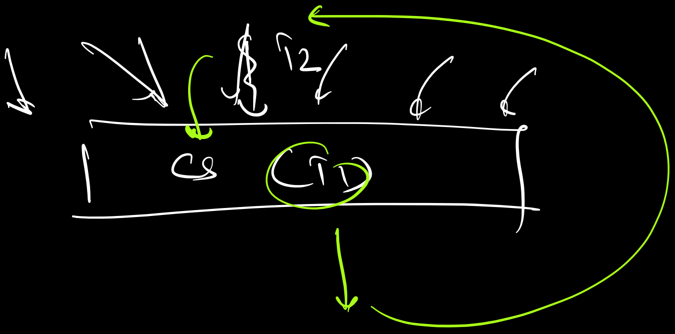
: Every thread should get progress after a finite wait.

* threads will get access to CS sequentially

4) No Busy waiting :-

=> when a thread comes to a CS, it should not keep checking to enter all the time, instead it should get some notification, once the CS is available





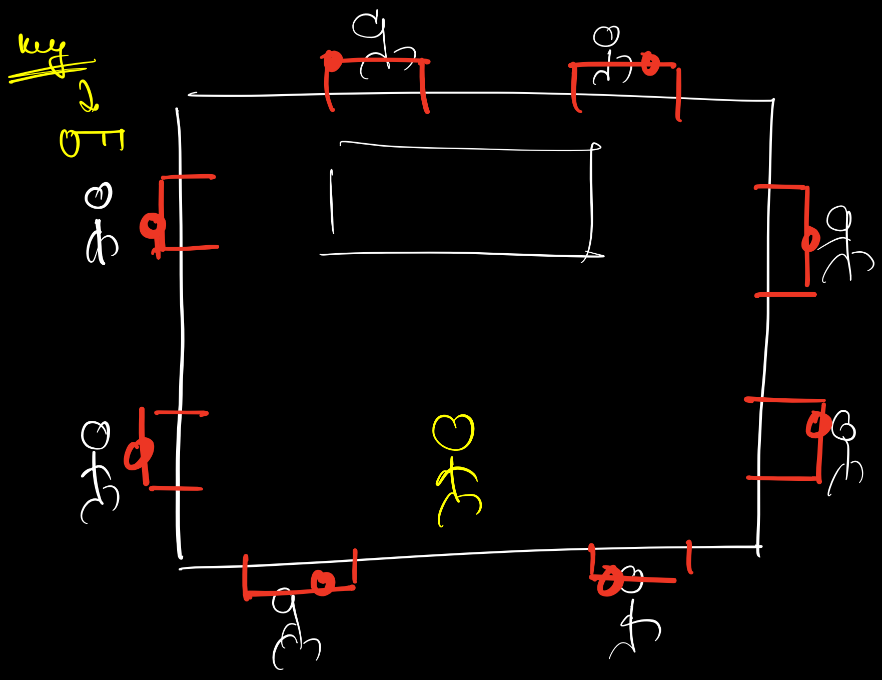
⇒ Soln to Sync problem :-

1) MUTEX ⇒ lock that enables mutual exclusion.

MUTEX → mutual exclusion

```
→ print(Hi)
      → lock.lock()
→ X ← count
  (10)
  X = X+2
  (12)
  count ← 2
  (12)
  print(Hello) → lock.unlock()
CS
```

```
→ print(Hello)
      → lock.lock()
→ X ← count
  (10)
→ X = X X2
  (20)
→ count ← 2
  (20)
  print(Bye) → lock.unlock()
CS
```



⇒ Only 1 person can enter the room at a time, so the room becomes mutually exclusive

⇒ Properties of lock :-

1) Only 1 thread can unlock the lock and enter the CS.

ii) Other threads have to wait, until the previous thread comes out

iii) lock will automatically notify the waiting thread, to run when the first threa unlocks / completes CS execution.

→ Mutual Exclusion → ✓

→ Progress → ✓

→ Bounded waiting → ✓

→ Busy waiting → X

Assgn ⇒ Thread → class

↓

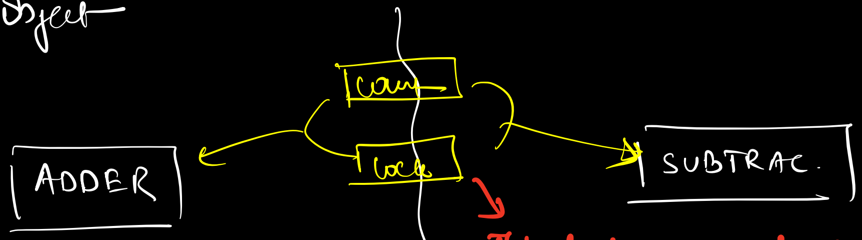
methods() → join()

↓

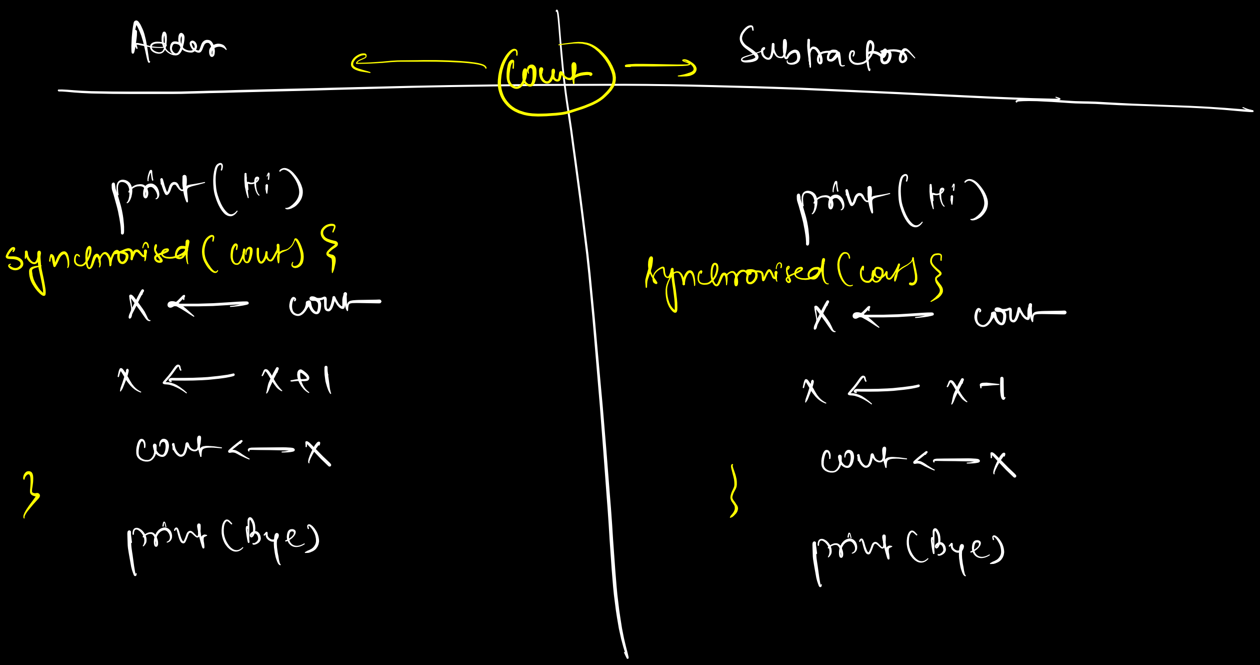
parameters

: Synchronised keyword :-

In Java, we have an implicit lock in every object



This lock exists to prevent sync problems for count



```
print(Hi)
synchronised(count) {
    X <— count
    x <— x+1
    count <— x
}
print(Bye)
```

```
print(Hi)
synchronised(count) {
    X <— count
    x <— x-1
    count <— x
}
print(Bye)
```

```
Synchronised ( obj ) {  ——→ lock
    ________
    ________  } CS
    ________
}  ——→ unlock
```

There should be only 1 shared variable/resource

⇒ {
- ⇒ synchronised method
- ⇒ semaphore (producer - consumer problem)

⇒ memory | atomic data