i) What is synchronisation issue?

ii) When synchro. issue happens?

iii) What is the ideal soln to syn. problem?

iv) Solution:

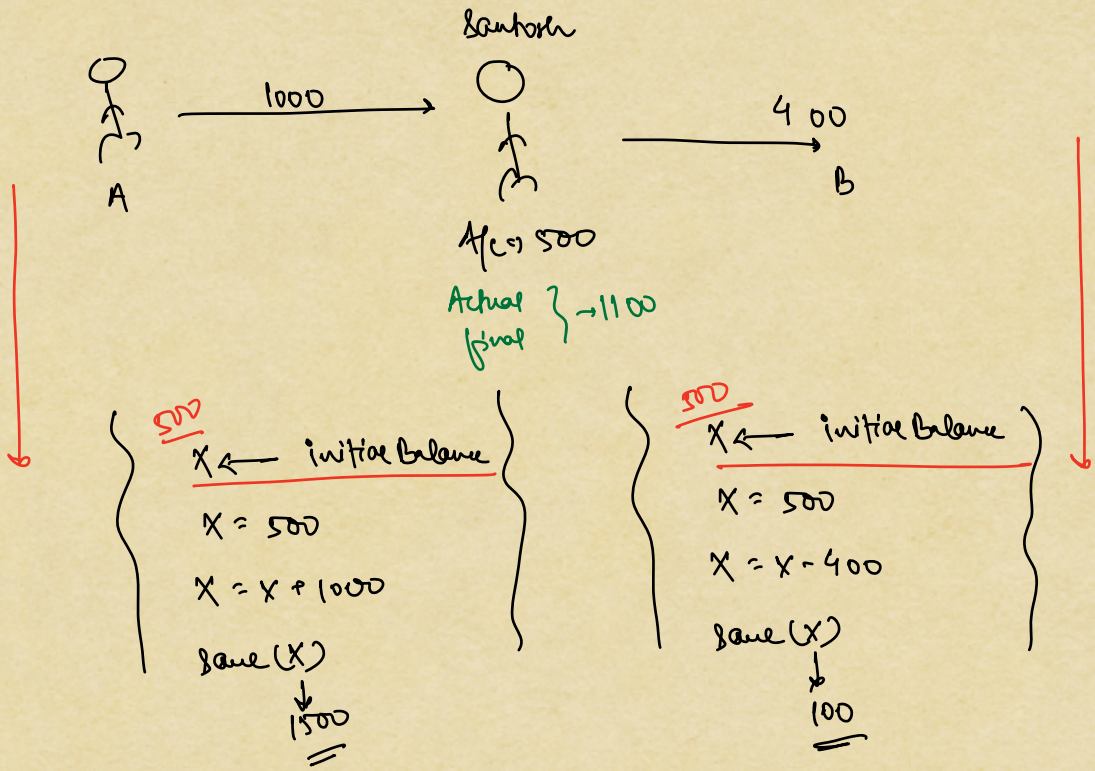
i) Mutex

ii) Synchronisation

iii) Semaphores.

: SYNCHRONIZATION PROBLEM

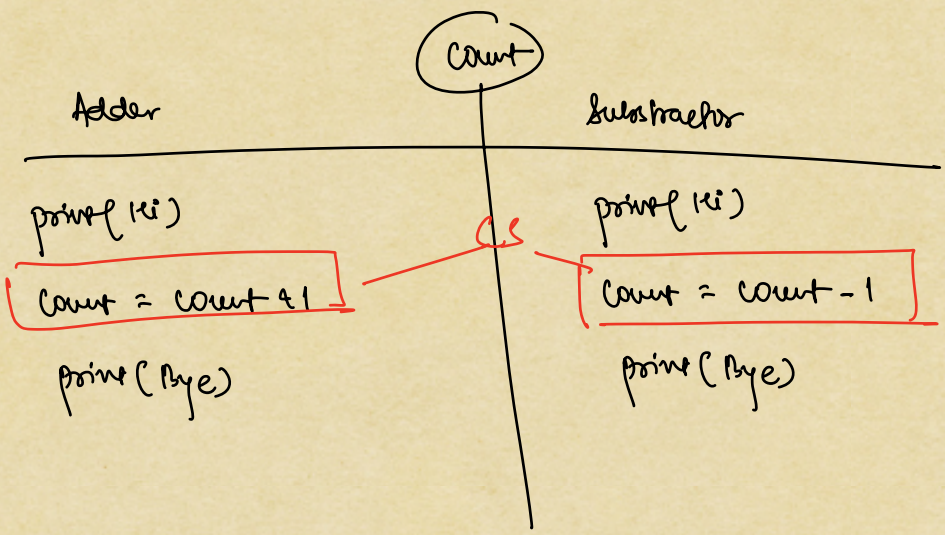
When more than 1 thread work on the same data at the same time, it can lead to inconsistent results.



⇒ When does a synchronization problem happen?

* **CRITICAL SECTION:-** It is the part of your code that is working on same piece of data

If 1 or more threads try to execute critical section at the same time, it can lead to synch issue



* **RACE CONDITION:-**

More than 1 thread tries to enter the CS at the same time.

* **PREMPTION:-**

CPU does preemption b/w threads ie, switches one thread from executing to another thread.

→ Assume single core CPU:

Adder

Count = 10

→ print(Hi)	
→ x ← read count	x = 10
→ x = x + 1	
→ count ← x	
→ print(Bye)	

⇒ Properties of a good solution of sync issue:

1) Mutual Exclusion:

* only one thread can enter the critical section at one time

* prevention of race condition

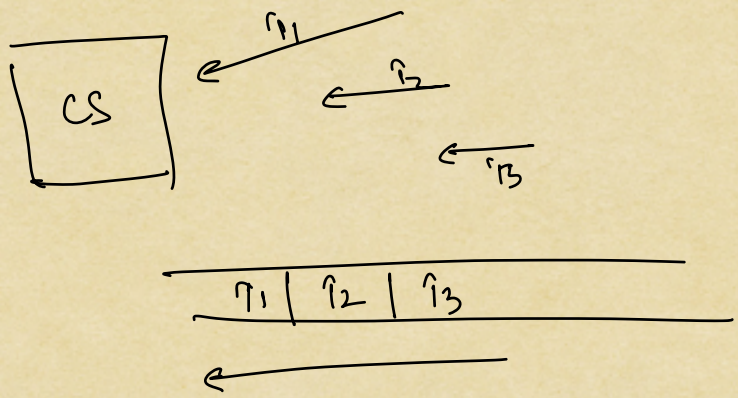
2) Progress: The overall system should keep working and making progress.

3) Bounded Waiting:

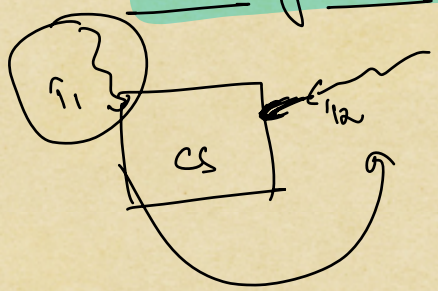
A thread should not be kept on waiting infinitely. There should be a bound of wait time for each thread.

4) **Maintain order of request:**

Threads will get access to the CS in the order of request



5) **No Busy waiting:**



Thread should not continuously check for entering the CS, instead it should get informed once CS is free.

(i) keep knocking

(ii) wait and you get notification one available

⇒ SOLN TO SYNCH PROBLEMS:-

1) MUTEX:-

↳ Mutually Exclusive

↳ lock that enables mutual exclusion.

Count

Adder	Substractor
1 → print(Hi)	→ print(Hi)
CS: 2 → X ← read count (lock.lock())	CS: → X ← read count (lock.lock())
3 → X = X + 1	→ X = X - 1
4 → Count ← X (lock.unlock())	→ Count ← X (lock.unlock())
5 → print(Bye)	→ print(Bye)

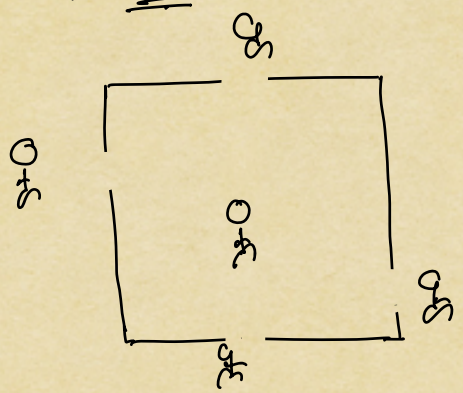
lock

→ A thread (person) must take a lock (close) before entering the critical section.

→ In the meanwhile other threads will wait.

→ As soon as a thread completes execution on CS, it should release the lock.

→ At any point of time only 1 thread can enter the CS.



→ Properties of lock:-

i) Only 1 thread enters the CS at one time ⇒ mutual exclusion

ii) Other threads wait until CS is unlocked.

iii) Lock notifies the next waiting thread as soon as CS becomes free.

iv) maintains queue and bounded waiting.

: SYNCHRONISED KEYWORD:-

→ In Java there is an implicit lock in every object.

ADDER

Count

↓

this object internally holds a lock to solve sync. issue

SUBSTRACTOR

Count

Adder	Substractor
1 → print(Hi)	→ print(Hi)
synchronised(count){	synchronised(count){
2 → X ← read count	→ X ← read count
3 → X = X + 1	→ X = X - 1
4 → Count ← X	→ Count ← X
}	}
5 → print(Bye)	→ print(Bye)

There is no need for an external lock as the count object itself contains a lock

synchronised (count) { ⟶ Count_lock.lock()

CS { ———

} ⟶ Count_lock.unlock()

notifies other thread

* Ideally we should use synchronised block for a single shared variable

→ Synchronised Method:

In Java, we can make the entire method as synchronised, and if a method is synchronised only 1 thread can execute it at any pt. of time

```
public void adder() {
        sout(Hi);
synchronised(count) {
        int x = count.value;
        x = x+1;
        count.value = x;
}
        sout(Bye)
}
```

```
public synchronised void adder() {
        sout(Hi);
        int x = count.value
        x = x+1;
        count.value = x;
        sout(Bye)
}
```

* Taking lock on entire method is more expensive

* locking the entire method will slow down the overall system

we should always prefer. synchronised block

2 mins → break