Agenda.

→ Intro to Synchronization.

→ MUTEX

→ SEMAPHORES.

Intro to Synchronization.

Assignment

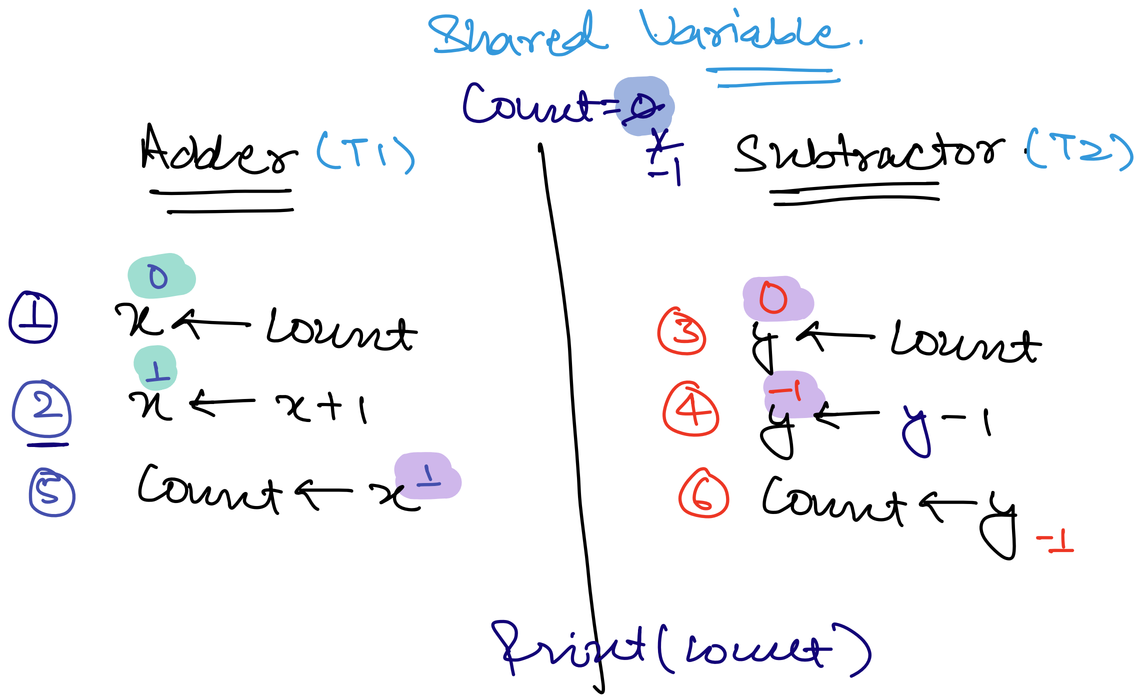
→ Shared Count object.

→ One thread will add no's from 1 to 100 in the Count object & other thread will subtract no's from 1 to 100 from the Count obj

→ Both of these threads will run parallely.

⇒ Synchronization.

Count += 1;



Shared Variable.

Count = ~~0~~ ~~1~~ -1

Adder (T1)

① x ← Count   0

② x ← x+1   1

⑤ Count ← x   1

Subtractor (T2)

③ y ← Count   0

④ y ← y-1   -1

⑥ Count ← y   -1

Print(count)

⇒ Count = -1 ×

Synchronization.

⇒ When more than one thread works on a shared variable at the same time, it can lead potentially wrong results.

① Critical Section (CS)

⇒ Section of code involves shared data.

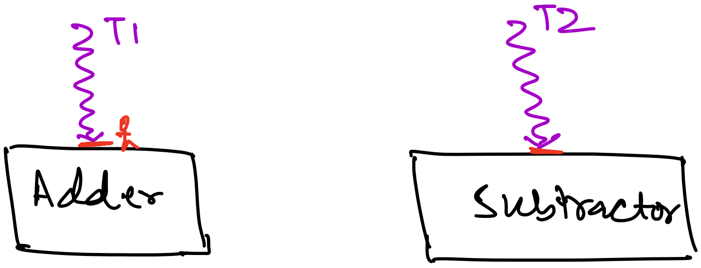
⇒ If there are more than one thread present inside the CS at the same time, it can lead to synch problem.

Count=0

Adder (T1)	Subtractor (T2)
1) Print (Hello)	4) Print (Hello)
2) Count+=1 ← T1	5) Count-=1 ← T2
3) print (Bye)	6) print (Bye)

② Race Condition.

⇒ When multiple threads are trying to access the shared resource at the same time.

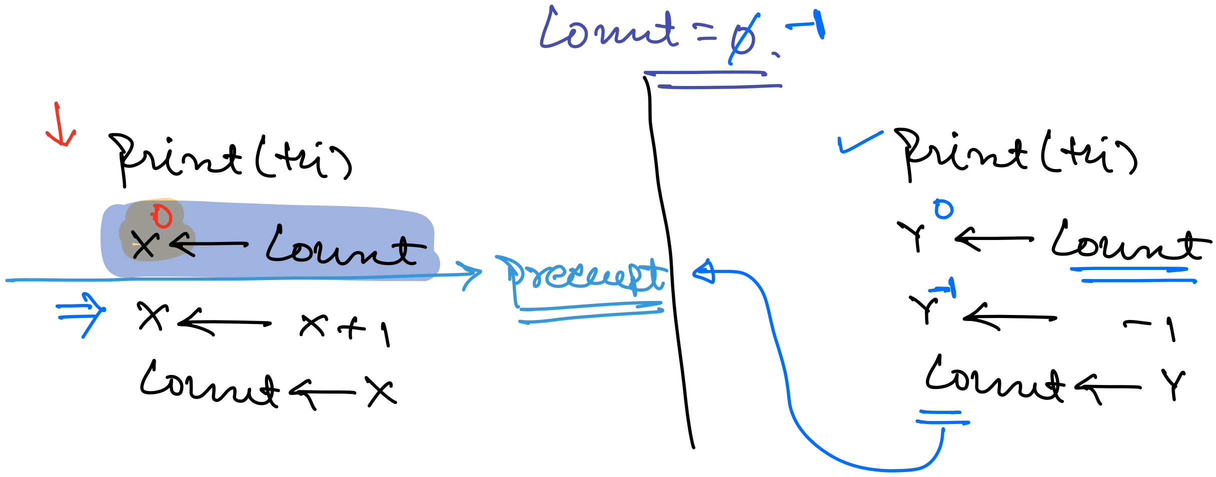


③ Preemption.

⇒ A program which in its critical section is preempted by CPU can lead to Synchroniz" problem.

halt | stop | abort

⇒ Assumption : Single Core CPU.



Count = ~~0~~. -1

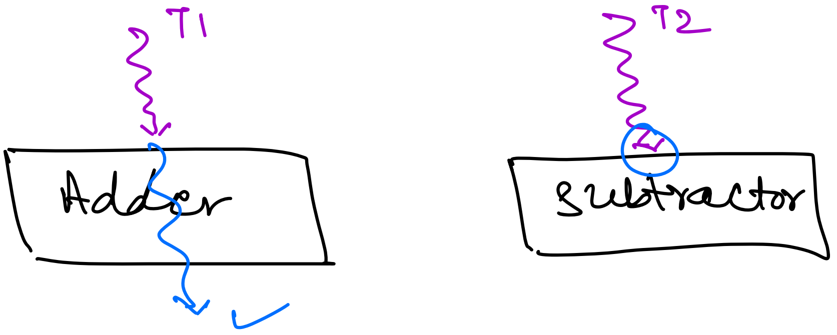
```
Print(hi)                 Print(hi)
X ← Count                 Y ← Count
X ← X+1                   Y ← -1
Count ← X                 Count ← Y
```

⇒ If a thread is preempted in their CS, it can lead inconsistent data.

Properties of an Ideal Soln. to Synchronization Problem.

1) MUTUAL EXCLUSION.

⇒ Only one threads should be present inside the CS at any point of time.



2) Progress.

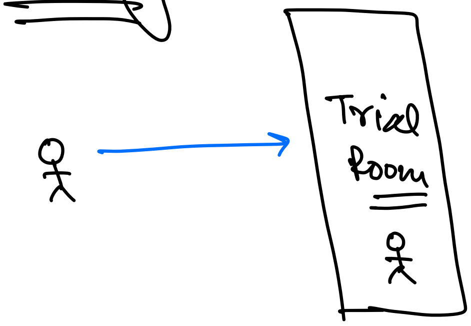
⇒ Overall system should keep making the progress.

3) Bounded Waiting

⇒ No thread should have to wait infinitely.

4) No Busy waiting

=> When a thread has to continously check if they can enter inside the CS, this is Busy Waiting.



=> Our code shouldn't have Busy Waiting

Sol^n

1) MUTEX => LOCK.

Mutually

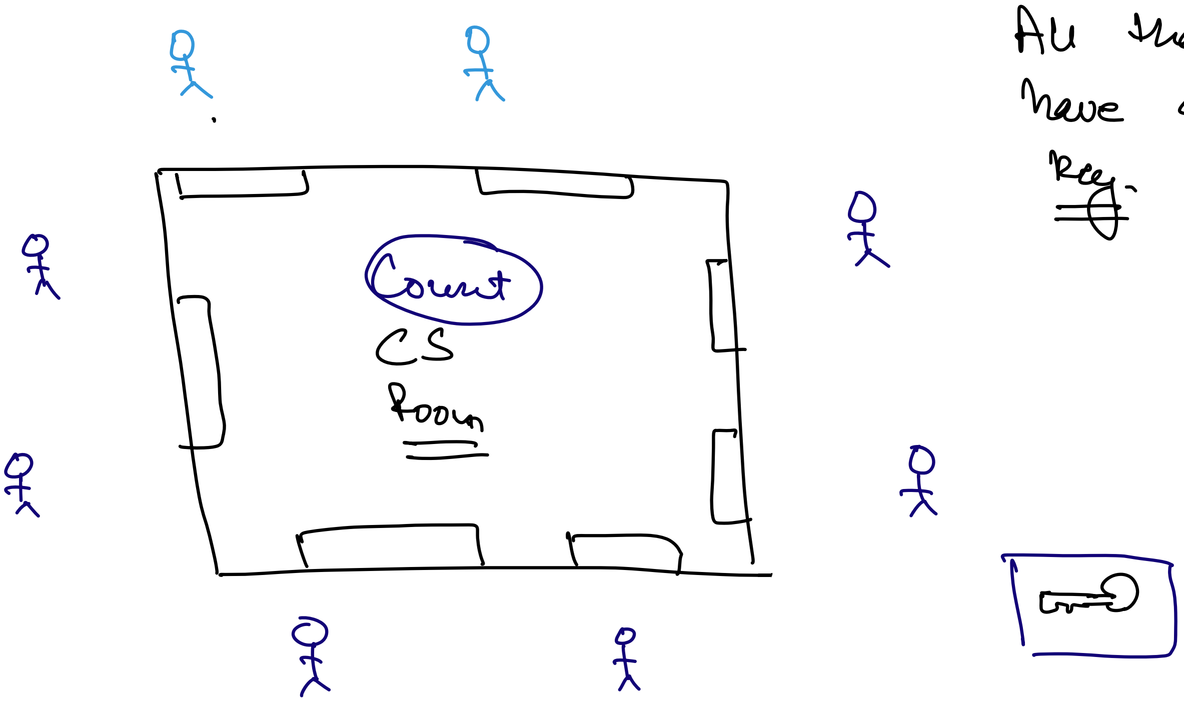
Exclusion.

T2
T3

Lock
Count=0

```
Adder (T1)                     Subtractor (T2)

lock.acquire()                 → lock.acquire()   →
x ← count                      y ← count
x ← x+1                        y ←  -1
count ← x^1                    count ← y
lock.release()                 lock.release()
```



All these doors have a common key.

Properties of lock.

1) Only one thread can acquire the lock at a time.

2) Other threads will have to wait till the first thread unlocks the lock.

3) Lock will automatically notify the waiting threads

⇒ lock outside for loop.

+1 +2 +3 - - - - - - +100 | -1 -2 -3 - - - - - -100

⇒ lock inside for loop.

+1 +2 -1 -2 -3 +3 -4 -5 +4 -6 - - - . .

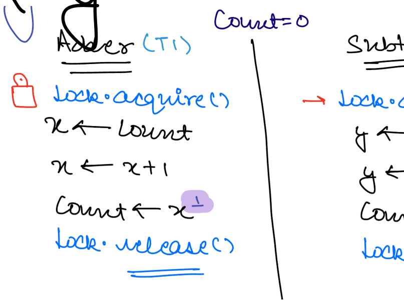
② Synchronized keyword.

In Java, Every object has implicit lock.

Count=0

```
Adder (T1)                              Subtractor (T2)

        ⇒ lock.lock()
synchronized (count) {                  synchronized (count) {
    x ← count                               y ← count
    x ← x+1                                 y ← -1
    count ← x                               count ← y
}                                       }
↳ lock.unlock()
```



Count=0

```
Adder (T1)                  Subt
lock.acquire()              → lock.a
x ← count                     y ←
x ← x+1                       y ←
count ← x                     Cou
lock.release()                lock
```

③ Synchronized Method.

If we declare a method of a class as synchronized then only one thread can be inside any sync method of that Object at a time.

```
Count {

    sync addValue(){ ≡ }

    Sync subtractValue(){ ≡ }

    getValue(){
          ≡
    }
}
```

```
Count c1 = new Count()
Count c2 = new Count()
```

T1	T2	Can they run in parallel?
C1.addValue()	C1.addValue()	✗
C1.addValue()	C1.subValue()	✗
C1.addValue()	C1.getValue()	✓
C1.addValue()	C2.addValue()	✓
C1.addValue()	C1.subValue()	✗

⇒ Only 1 thread can call one synchronized method on one object at any given point of time.

⇒ StringBuffer.

↳ Thread Safe.