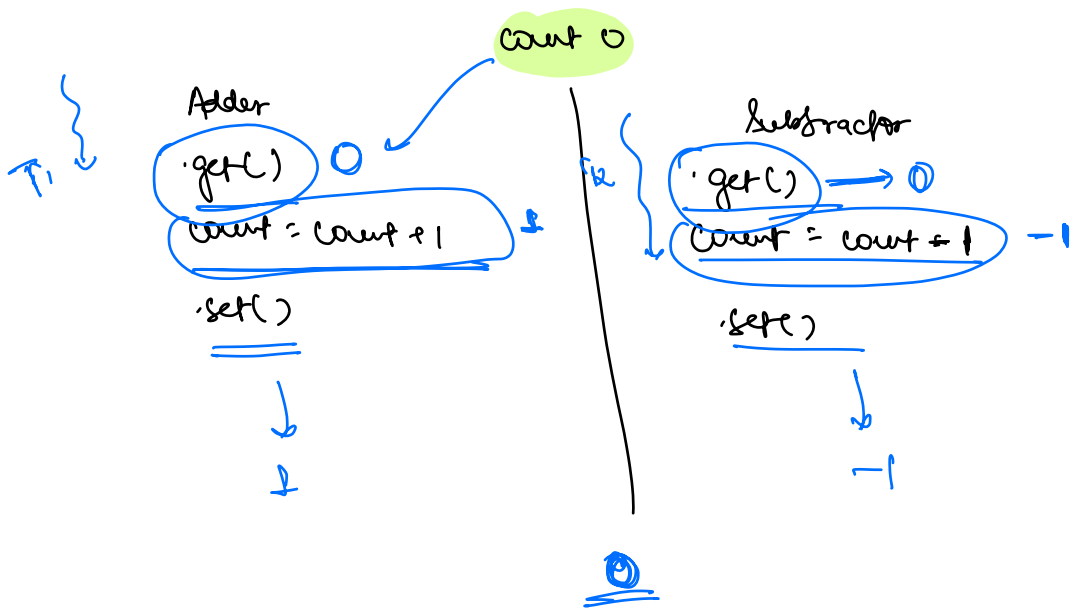
⇒ INTRO TO SYNCHRONISATION

1 thread ⇒ increments

1 thread ⇒ decrements the data

⇒ What is synchronization problem?

Ans When multiple threads are working on the same data at the same time, it can lead to inconsistent values, potentially wrong results



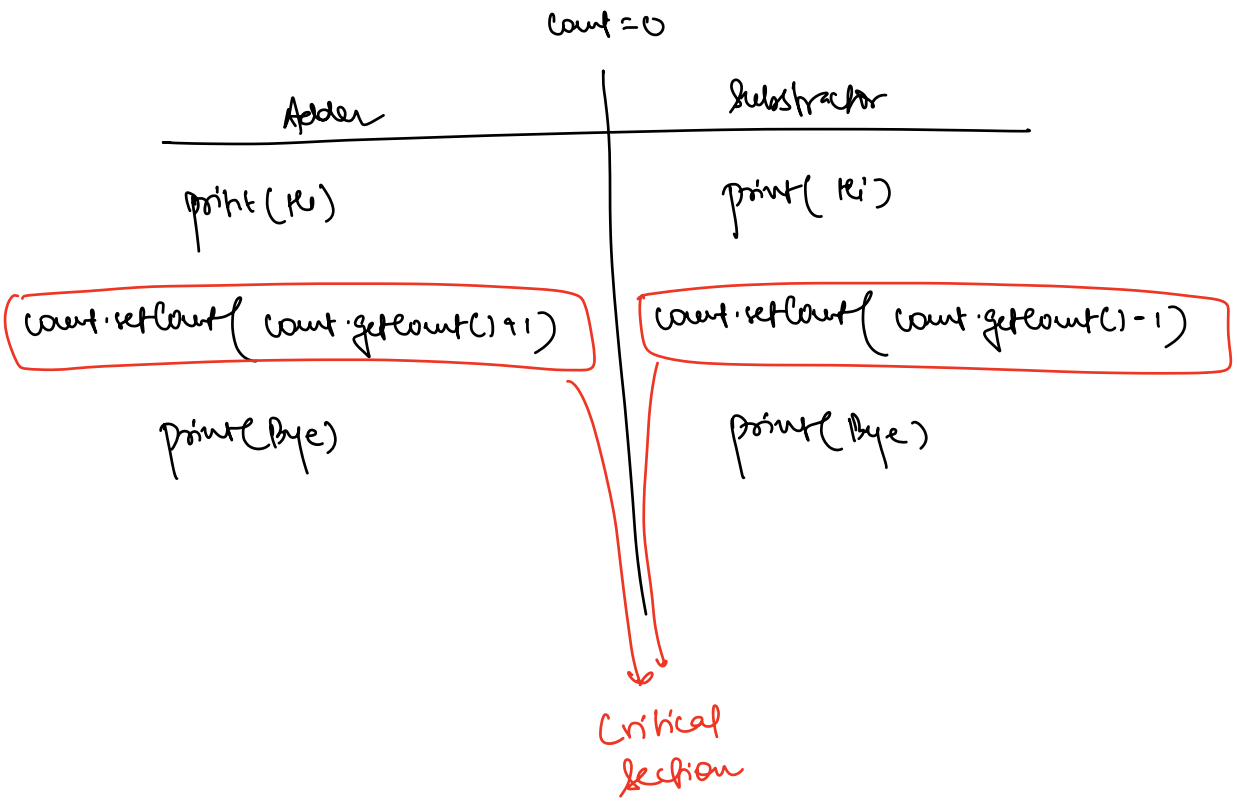
Count = 0

Adder → count = +1

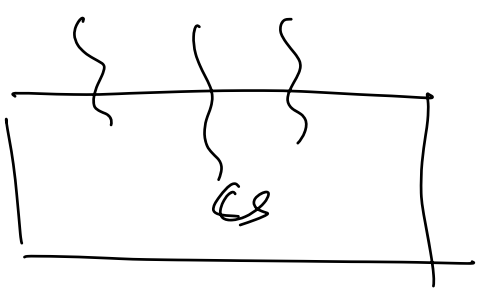
→ Subtractor ⇒ count -1 = 0

Count = 0

=> **Critical Section:-** Part of your code where potential issues might happen, so we need to be careful about that part of the code which is working on shared piece of data



Race Condition:- More than one thread trying enter the CS at the same time

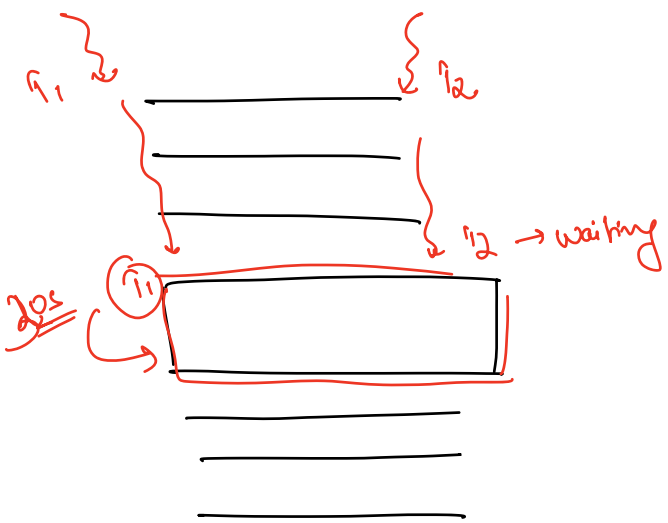


Solve => Even if we have CS in our code, and we are writing multi-threaded code, we want consistent results.

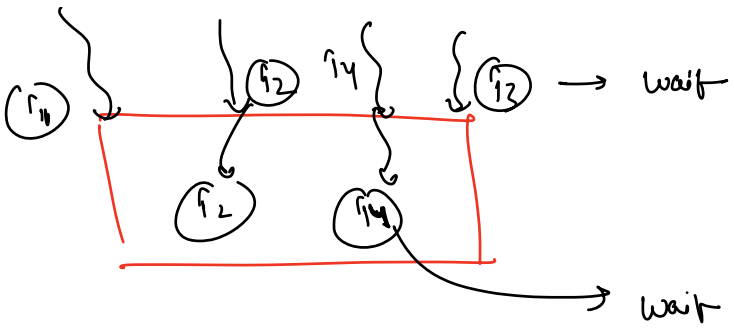
=> Properties of a good soln of synchronisation problem:-

i) **Mutual exclusion:** Make sure that only 1 thread enters the CS of code at one time.

ii) **Progress:-** The entire system should be making progress, there should not be any scenario where everything is in a wait state

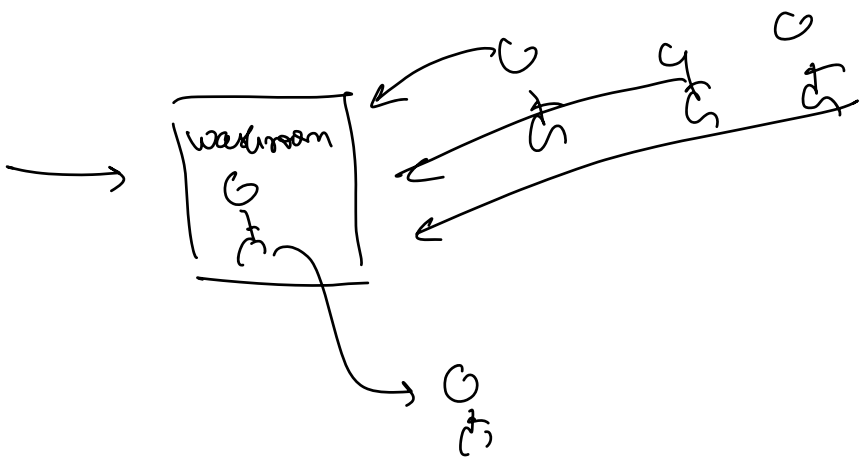


iii) **Bounded waiting:-** * No thread should wait infinitely
* There should be a bound for how long a thread will wait.



iv) No busy waiting :- Other threads should not keep on continously checking whether they can enter the CS.

The thread which is currently executing the CS, should notify all other threads once done.



⇒ Solutions to synchronisation problem :-

i) Mutex ii) Synchronisation iii) Semaphores

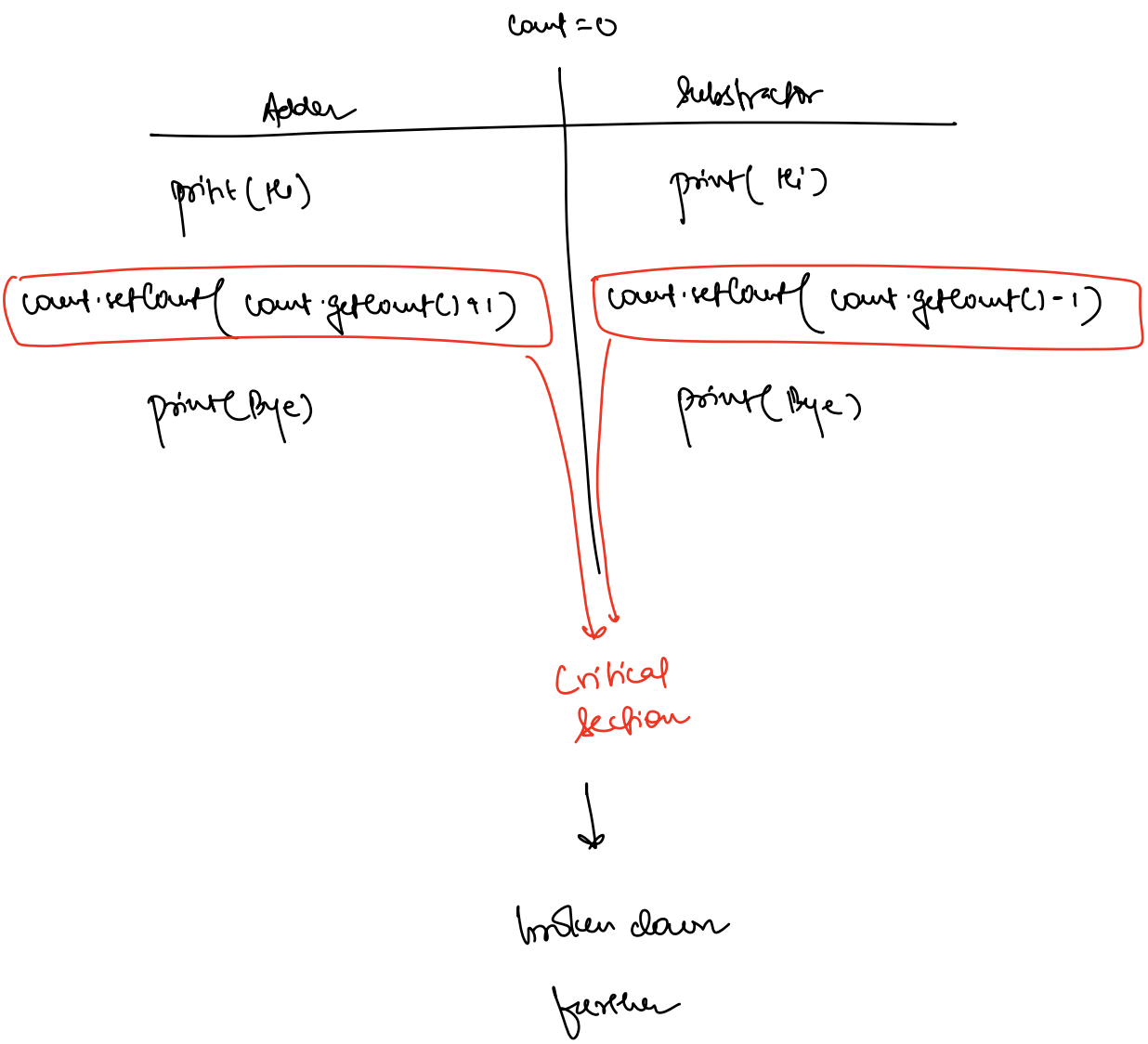
1) Mutex =>

MUT Ex

↓ ↓

Mutually Exclusive

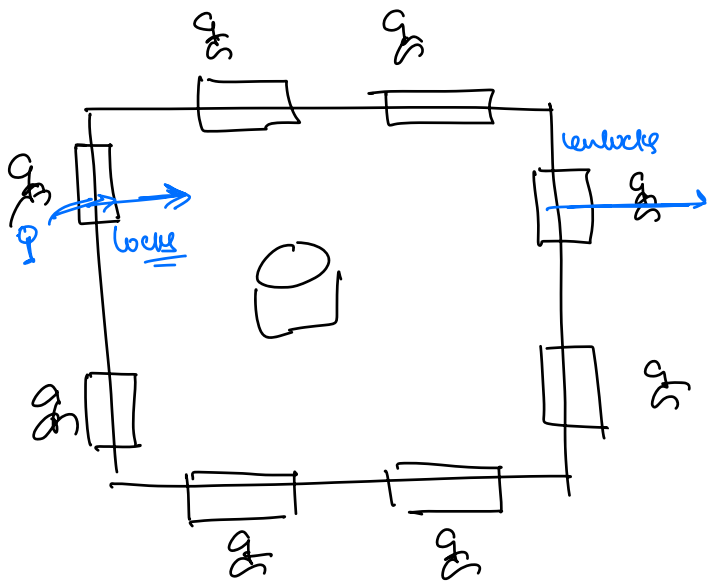
* Lock that allows mutual exclusion.



count = 0

	Adder	Subtractor
1	print(Hi)	print(Hi)
	lock.lock()	lock.lock()
2	x ← read(count)	x ← read(count)
3	x = x + 1	x = x - 1
CS		
4	count ← x	count ← x
	lock.unlock()	lock.unlock()
5	print(Bye)	print(Bye)

A thread must lock when it tries to enter a CS. and unlock when it leaves the CS.



* even though multiple people are trying to enter the room, only 1 person can obtain the key and enter the room at one time

* Properties of mutex lock :- [ME, progress, no busy waiting, bounded waiting]
→ mutual exclusion

i) Only 1 thread can unlock the lock at 1 time

ii) Other threads will wait until the initial thread unlocks

iii) mutex will notify the next thread to start execution when the initial thread completes → no busy waiting

iv) it supports bounded waiting. ✓

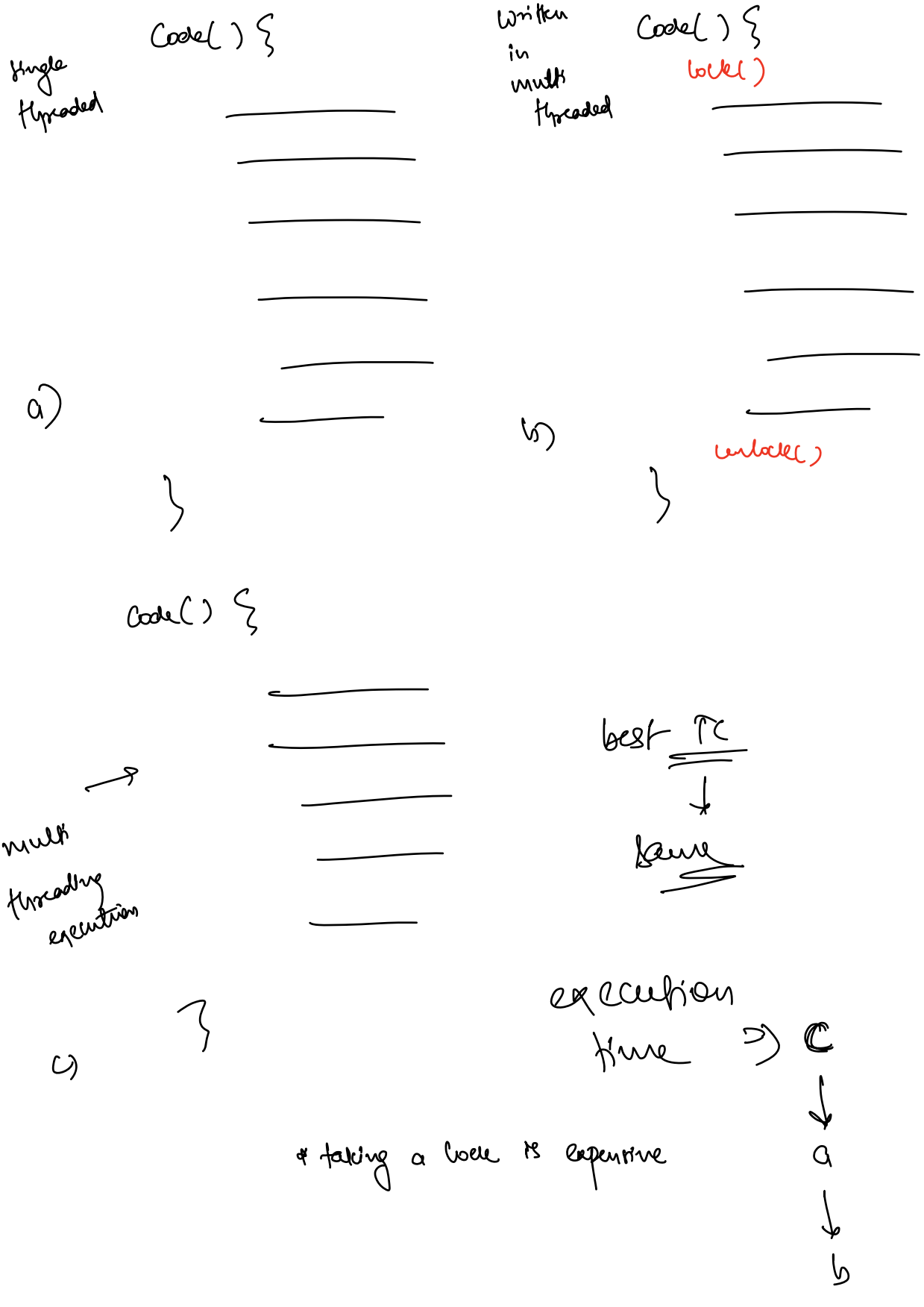
→ Mutex | Synchronisation | Semaphore

Reentrant

OS

* multi-threading ⇒ concurrent | parallel execution

* multi-threaded ⇒ lock everything


Code( ) {
single threaded
a)
}
written in multi threaded
Code( ) {
lock( )
b)
unlock( )
}
Code( ) {
multi threading execution
c)
}
best TC
same
execution time ⇒ C
a
b
* taking a lock is expensive

* lock ⇒ try to keep the locked section as small as possible

→ bigger the lock area, more expensive in terms of time & space

→ bigger the lock area, reduces the capabilities of multiple thread to execute parallely

lock → maintain consistency (✓)

reduce performance (X)