Agenda

1. Mutex
2. Synchronized
3. Semaphores
   - → Producer Consumer Problem
   - → leetcode Problem

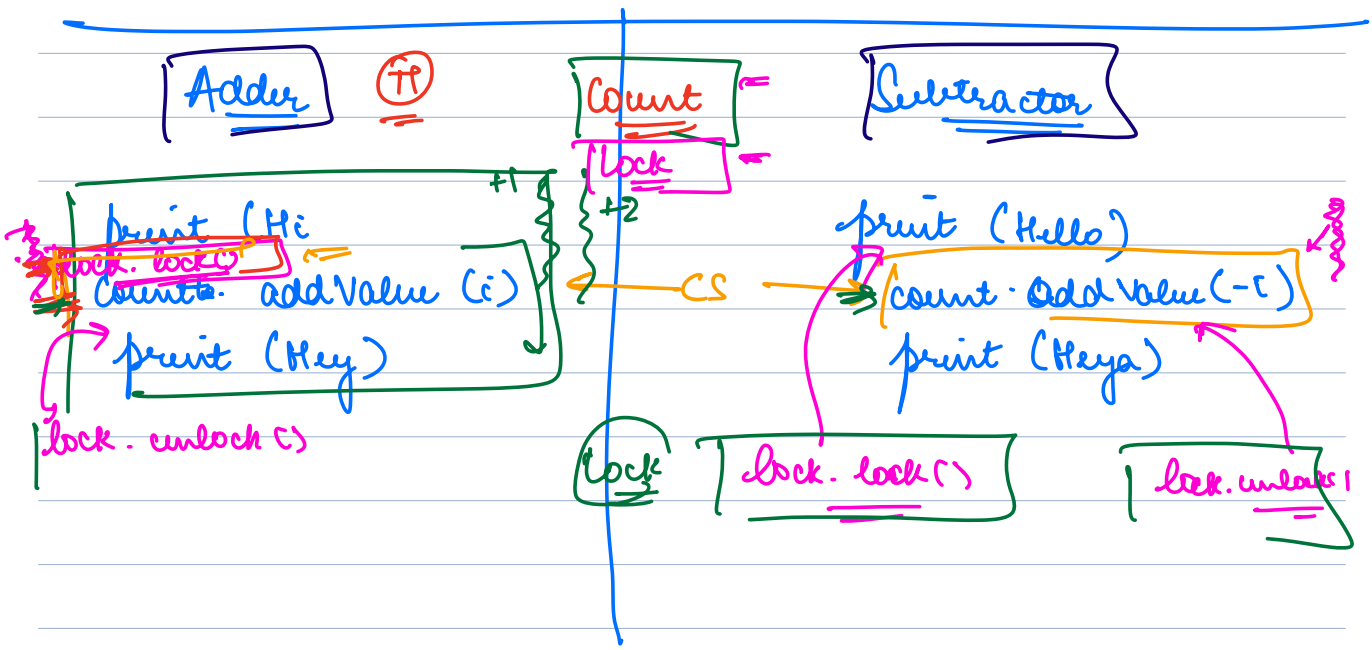
Assgn → {Few problems to Solve} =

MUTEX ← Exclusion

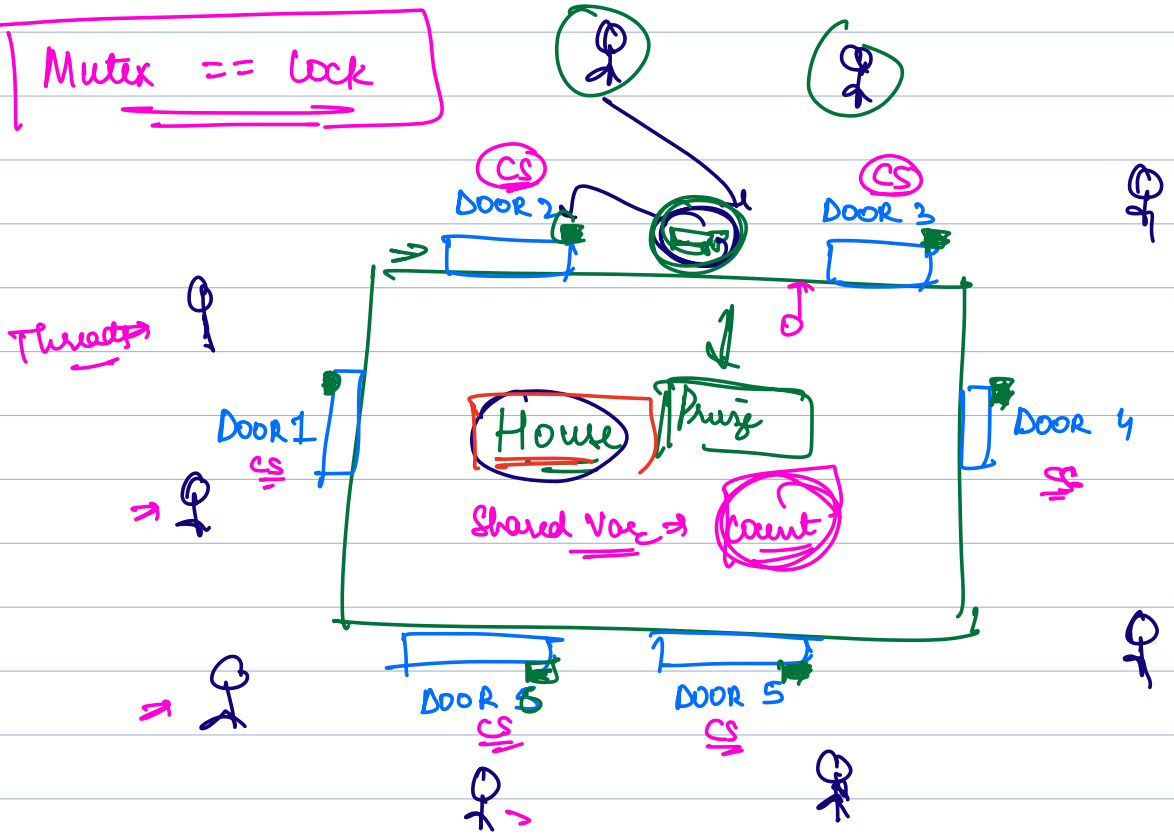
Mutual

Only one thread should be able to enter a critical section at a time

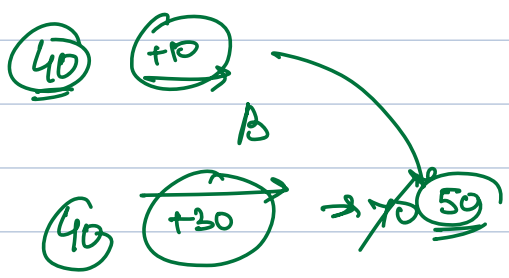
---



Mutex == Lock



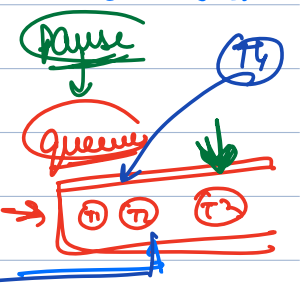
⇒ Only one person will be able to enter the house ⇒ who got the key



⇒ If there is a shared data that may lead to synchronization issues, we ask a thread to acquire a lock before entering CS

HOW LOCK WORKS

→ Shared lock obj
→ Before someone enters CS they must take lock



get lock()

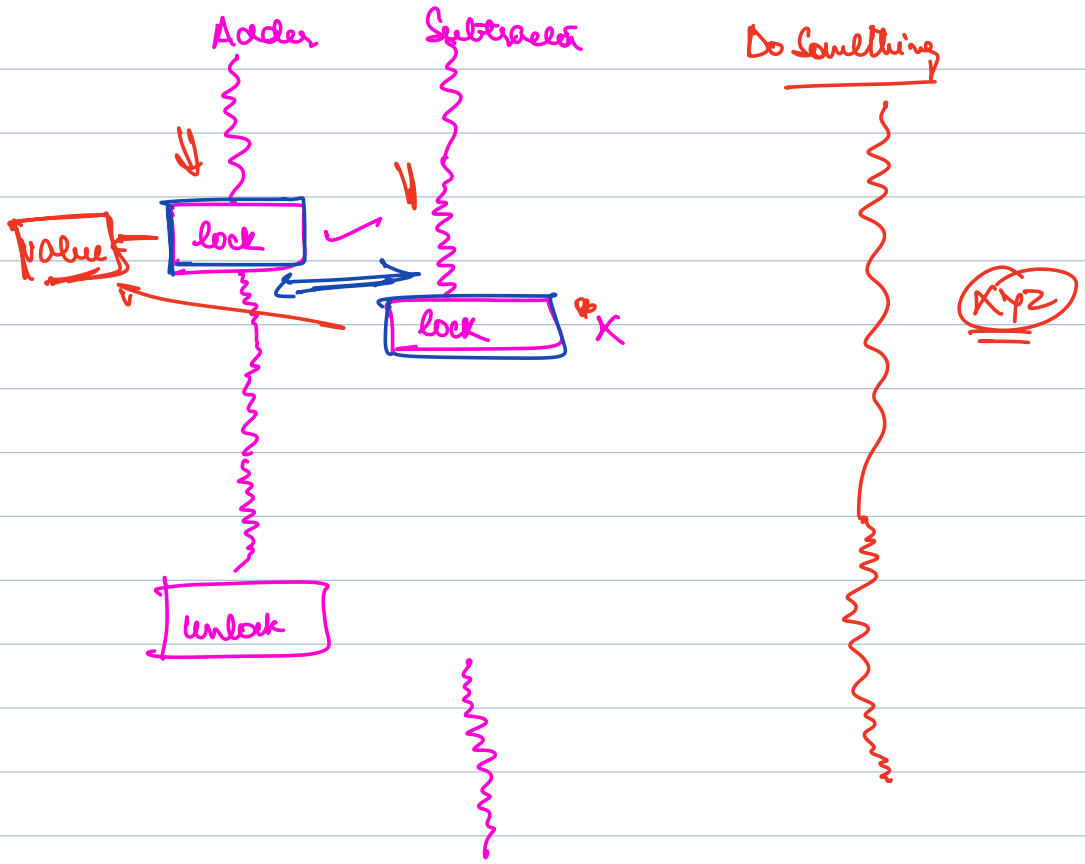
locked

already unlocked

thread will be able to go to next line of code

thread will be stopped
→ it will not be allowed to go to next line
→ put in a queue
→ Based on FIFO
when the thread that had the lock unlocks it, it gives access to the first thread in the queue

No Busy Wait ✓
Mutual Exclusion ✓
Bounded Waiting ✓
Progress ✓ → atleast one thread will have lock And keep running



value

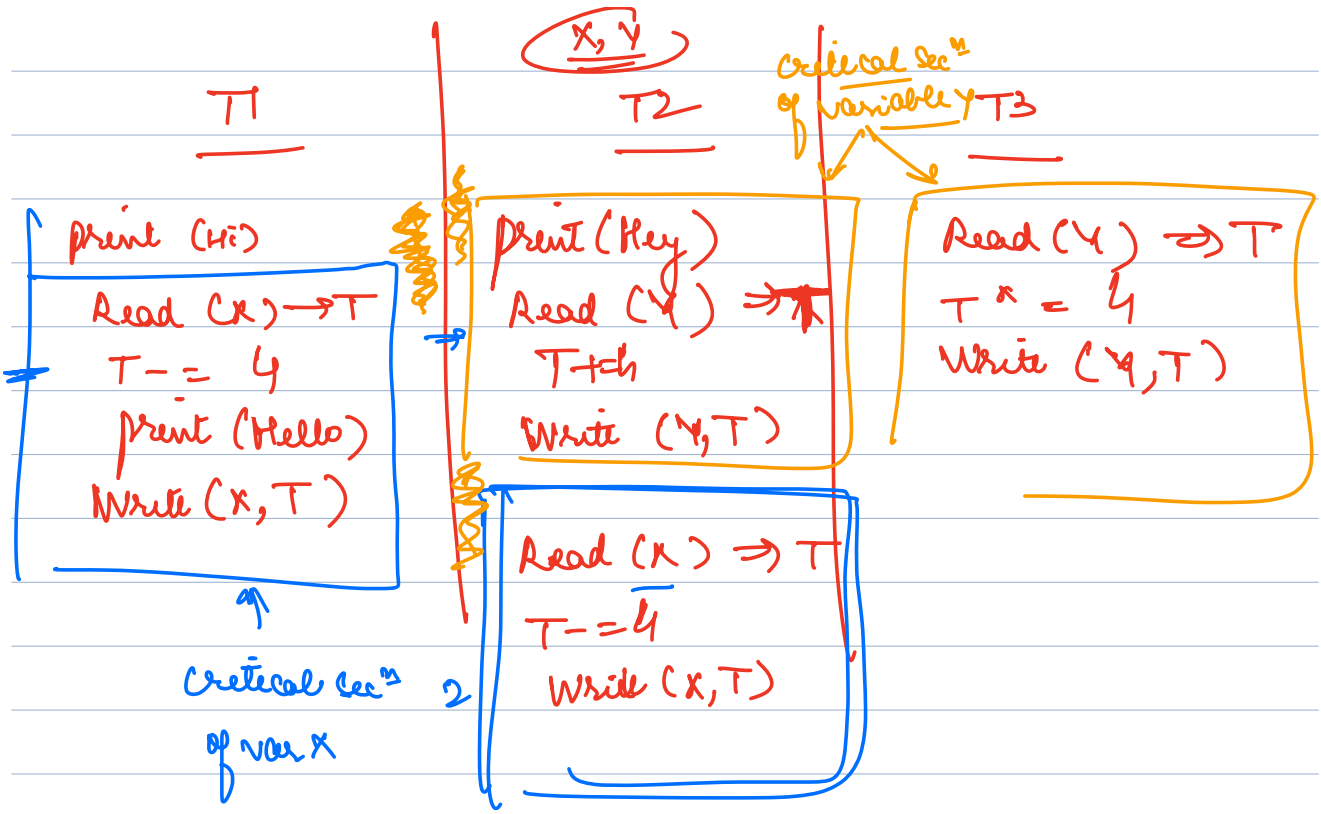
Adder

Subtractor

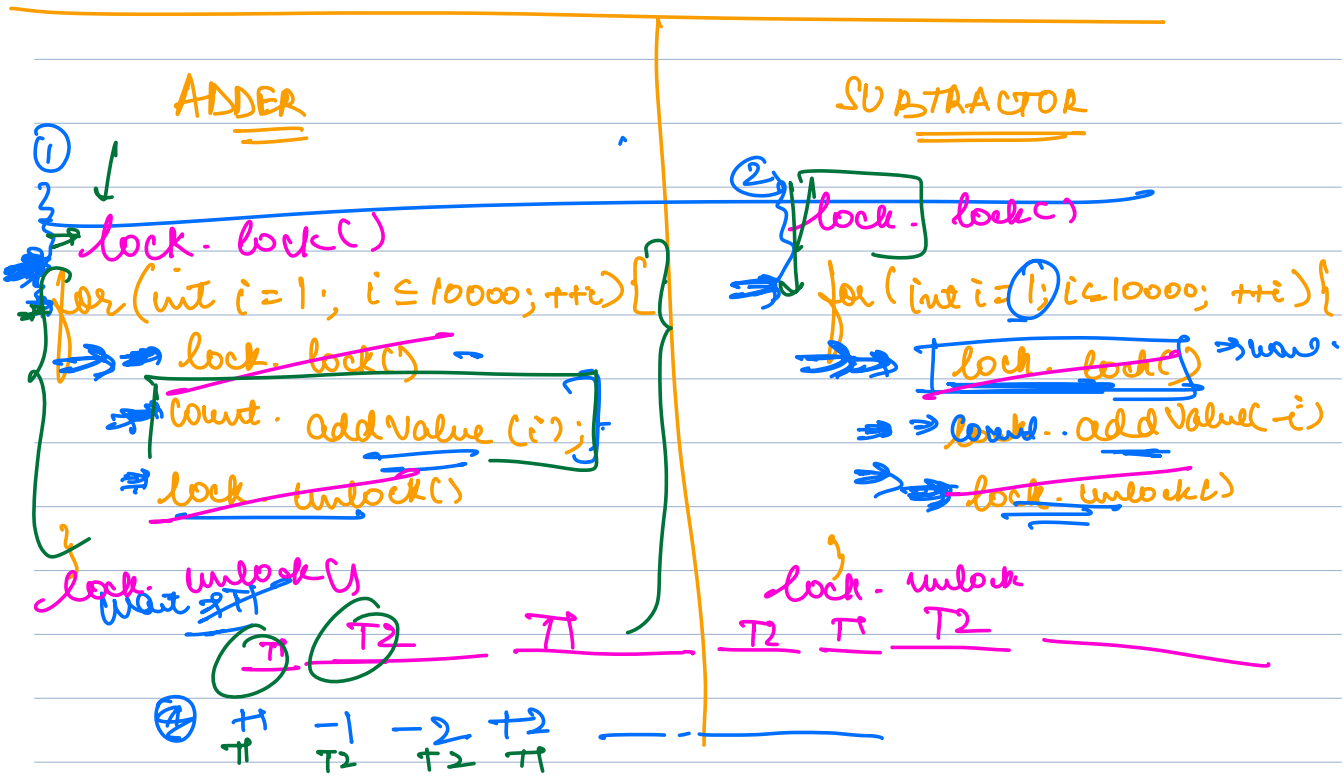
for i = 1 ; i <= n ; ++i

value += c

Square



CS is always associated with a variable



+1 +2 +3 +4 ——— +10000

T1 T2

lock.lock() { lock() {

H/W level

}

for (int i=1; i<=10000; ++i) {
    Read(value, temp);
    temp += i;
    Write(value, temp);
}

for (int i=1; i<=1000

count.addValue(i)

Synchronized Keyword (Java Utility)

⇒ In java every object has an implicit lock attached to it.

⇒ We can use that lock via Synchronized keyword.

Lock lockForCount = new ReEntrantLock()

Lock lock = new ReEntrantLock()

Count

Lock lockForXYZ = new Re

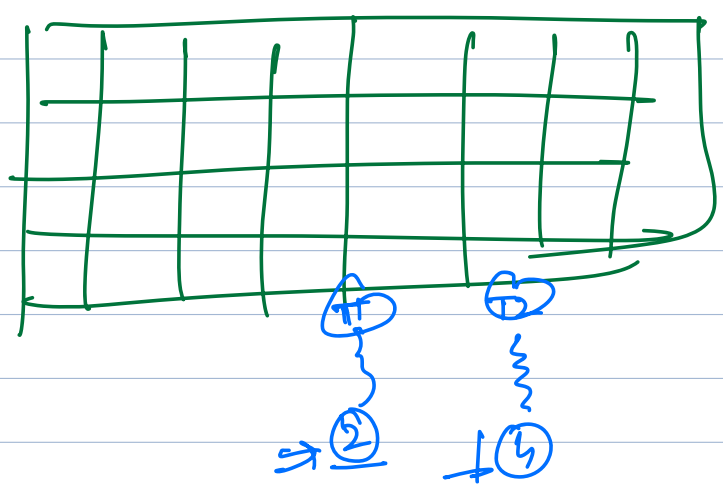
~~lock.lock()~~

⇒ XYZ

lock.lock()

(1)

lock



2

lock()

= lock.lock()

Count.lock()

lock.unlock()

⇒ In BMS, each seat will have an associated lock to it

lock (12, 14, 13, 15) {

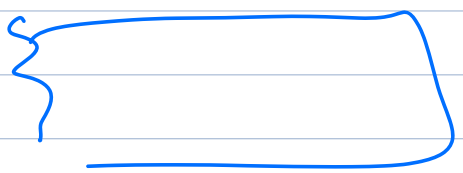
lock(2 . locks)

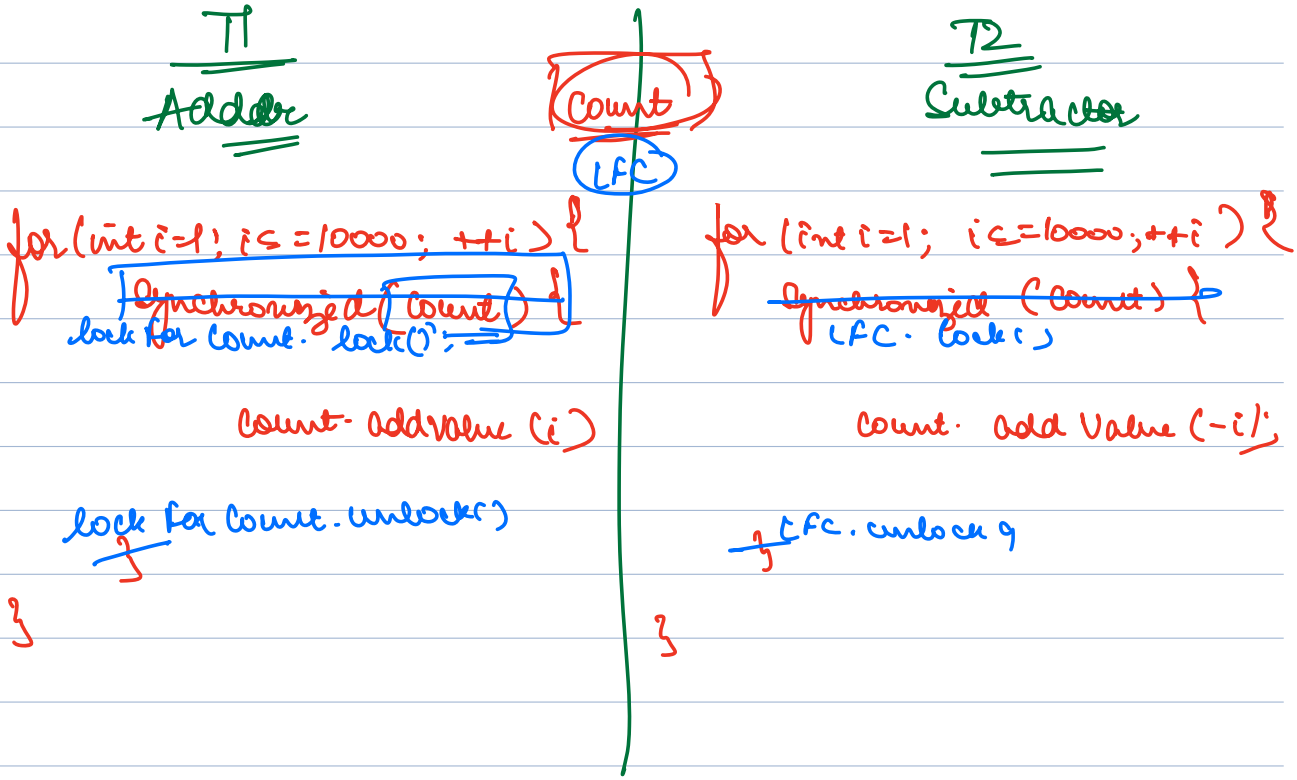
14 — —

13 — —

15 —

}





SYNCHRONIZED METHOD

⇒ lock on a method

⇒ you can declare a method to be Synchronized

Synchronized

```
void addValue (int i) {
    this.value += i;
}
```

```
class Count {
    value = 0;
    lock
    Sync  subValue (int i) {
            lock.lock()
            value -= i;
            lock.unlock
    Sync  addValue (int i) {
            lock.lock()
            value += i;
            lock.unlock()
    }
    -> mulValue (int i) {
    }
}
```

T1

T2

Count c1

Count c2 :

= c1 . subValue

- c2 subValue

T1

T2

⇒ All Sync methods of an obj share the same lock.

⇒ One thread in Sync and another in non Sync can run

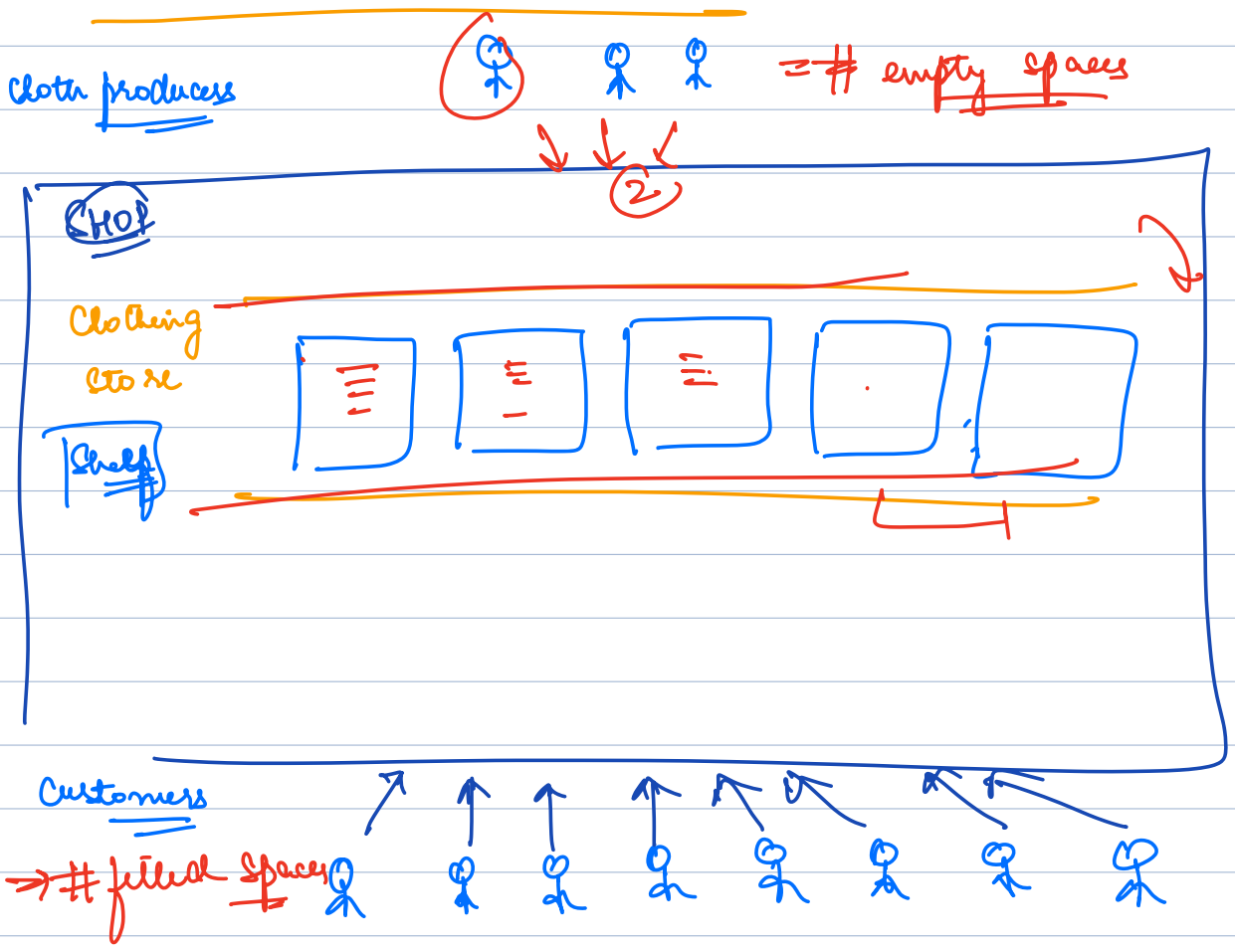
⇒ Lock is an object level not a class level lock

SEMAPHORES

→ Exactly 1 thread entering CS at one time

→ But, there are cases where we may be okay having more than one thread in the CS

PRODUCER CONSUMER PROBLEM



1 customer == 1 cloth

→ if a customer enters the shop, they shouldn't go empty handed.

→ producer to produce only if there is space for their product

Customers {

Store

Producers {

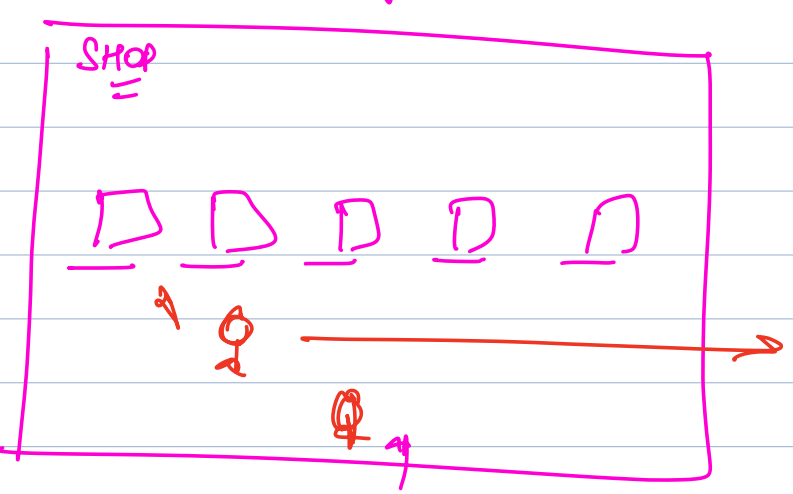
filled = 0

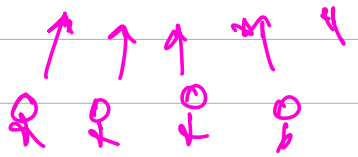
while (true) {
if (filled > 0) {
Store.buy()
}
}
}

while (true) {
if (filled < 5) {
Store.add()
}
}
}

T2
T1 T1 T2
T1 T1 → if (shop.size < 5)
shop.size += 1

size = 5

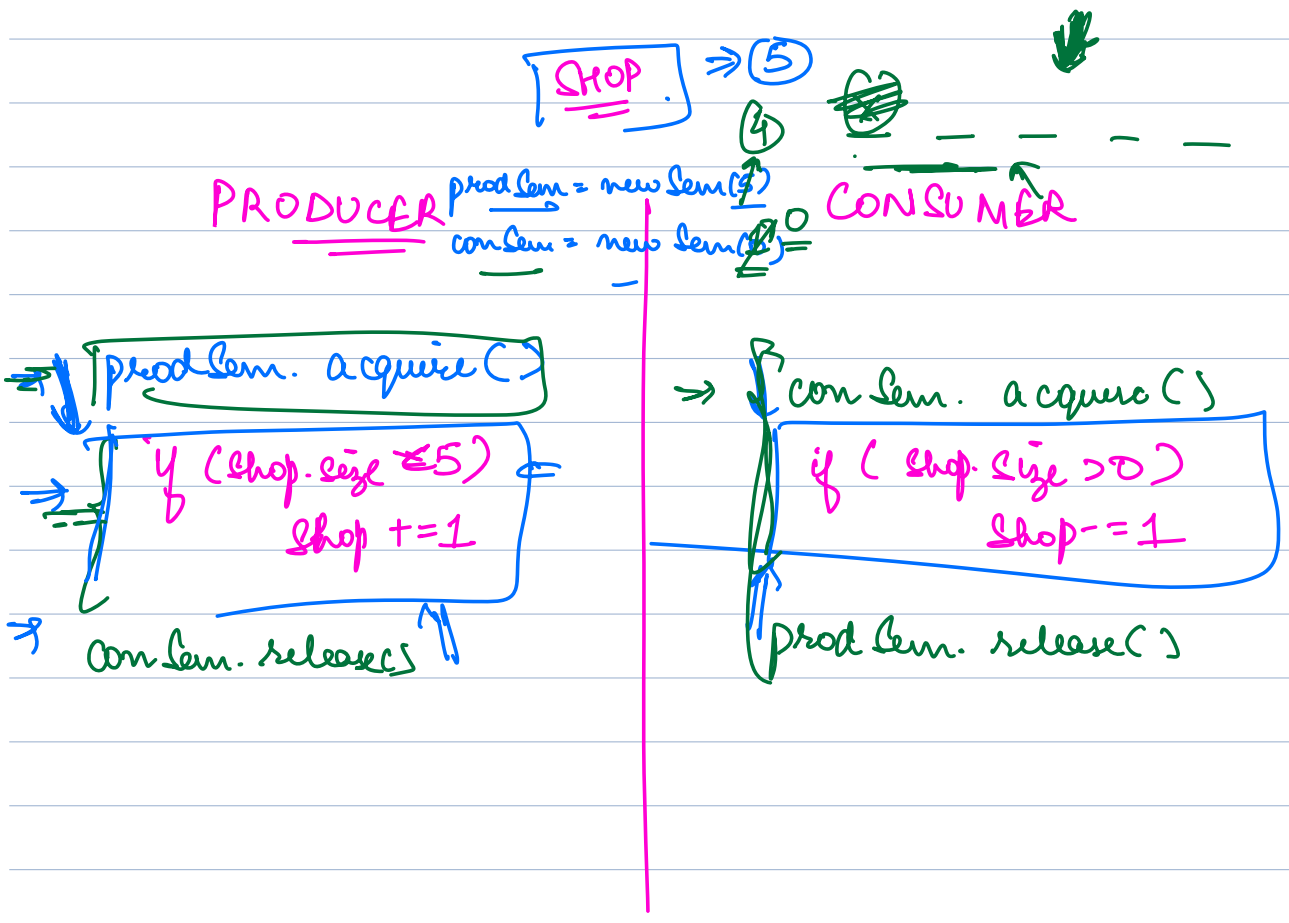




Semaphores ⇒ lock but where there are more than 1 key

Semaphore(x) ⇒ x ⇒ # of threads that can go inside the critical section

Mutex == Semaphore(1)



Semaphore s = new Semaphore (5); (count)

4 → 3 → 2 → 1

s.acquire() ⇒ if currently < i people have taken lock, allow else, wait

lock.lock()

s.release()

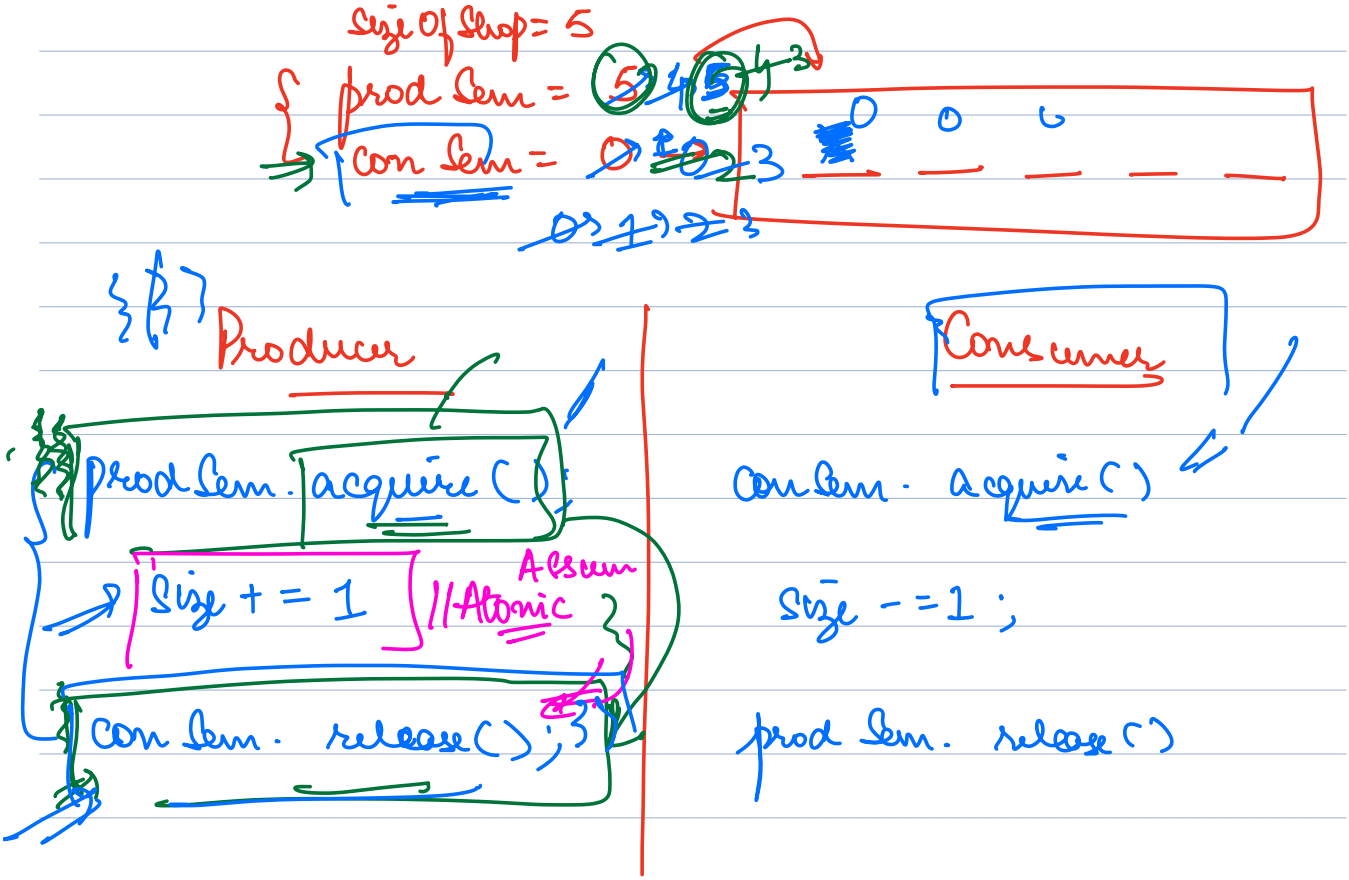
lock.unlock()

Semaphore ⇒ Mutex + Count

s.acquire() ⇒ if count > 0 ⇒ count -= 1 allow

how many threads can be allow

s.release() ⇒ count += 1



lock.lock() ⇦

Semaphore ⇒ Count + Lock

H/W ⇒ Revise Semaphores
→ Code P & C problem, Atomic data structure
→ Deadlock