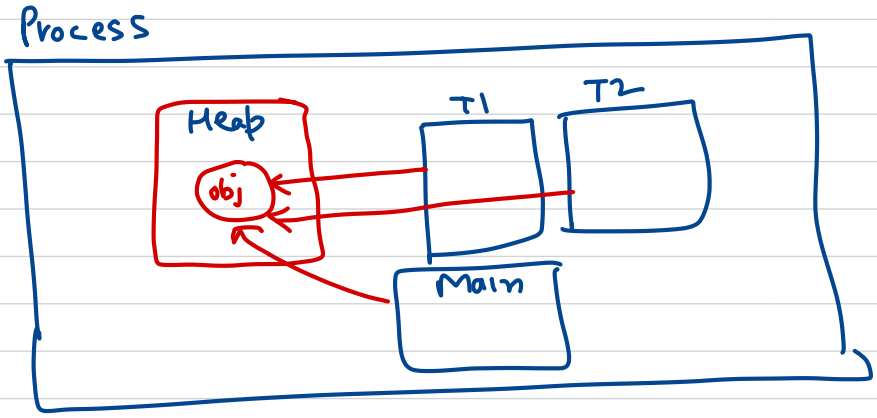
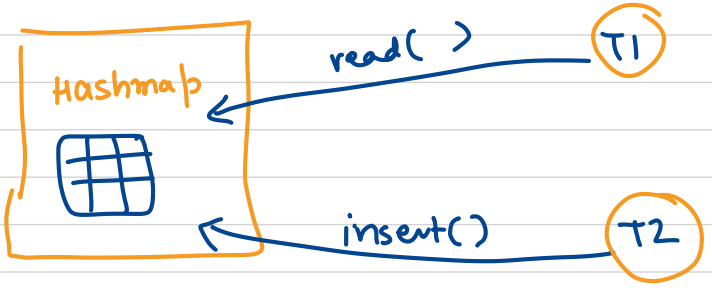
Concurrency 3:
Intro to Synchronisation



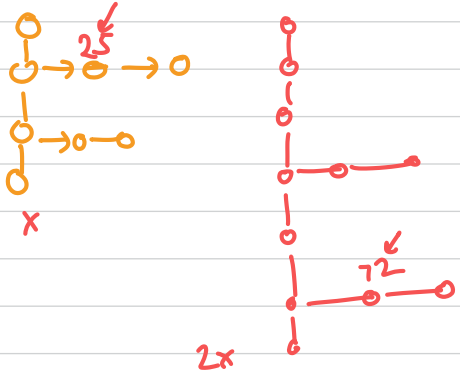
Objects in heap can be shared across different threads.

→ Concurrent execution can lead to modification of same object at same time, which can cause "inconsistent" state.

Advanced

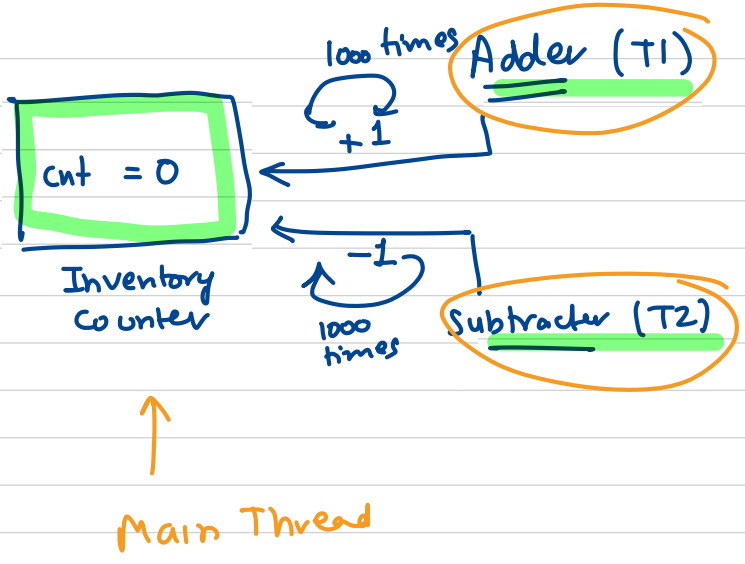


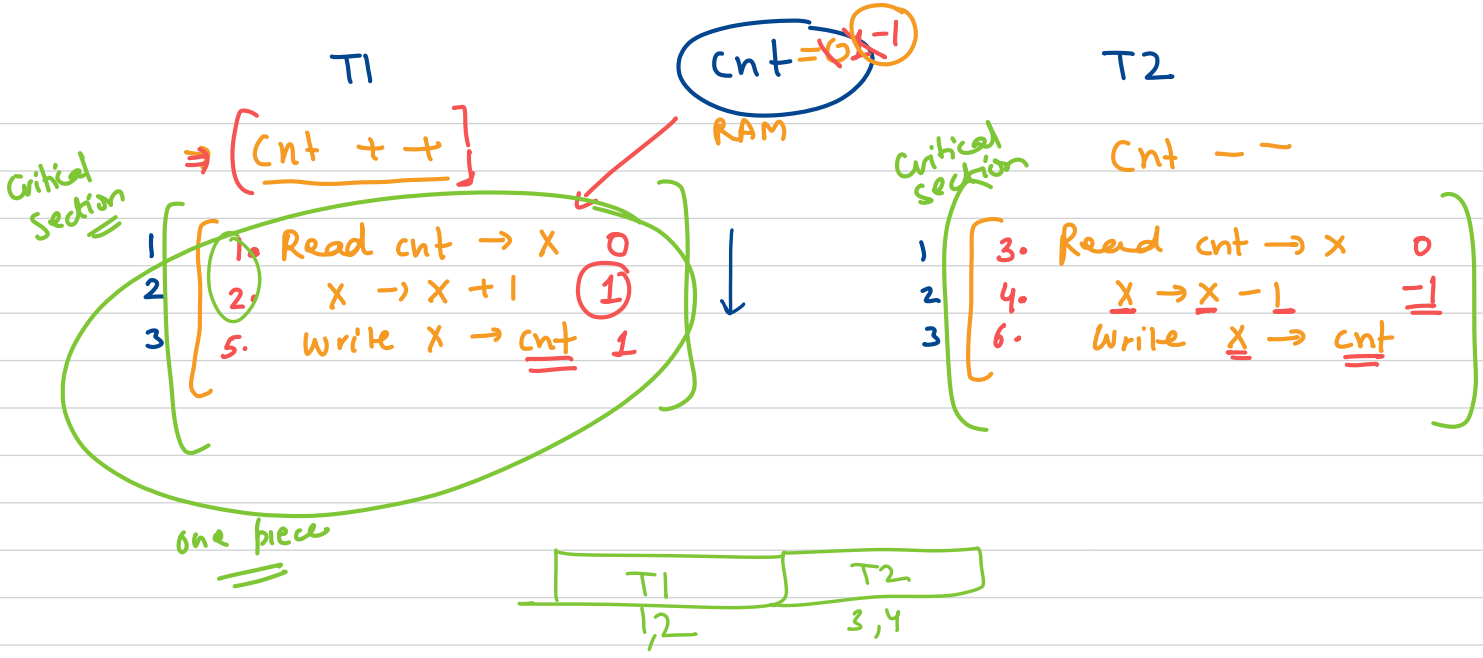

```
Read(key ){
→ idx = hashFn(...)
  go to bucket
  iterate
}
```




```
⇒ Insert( ){
→ check load factor
→ rehashing()
}
```

Simple Example

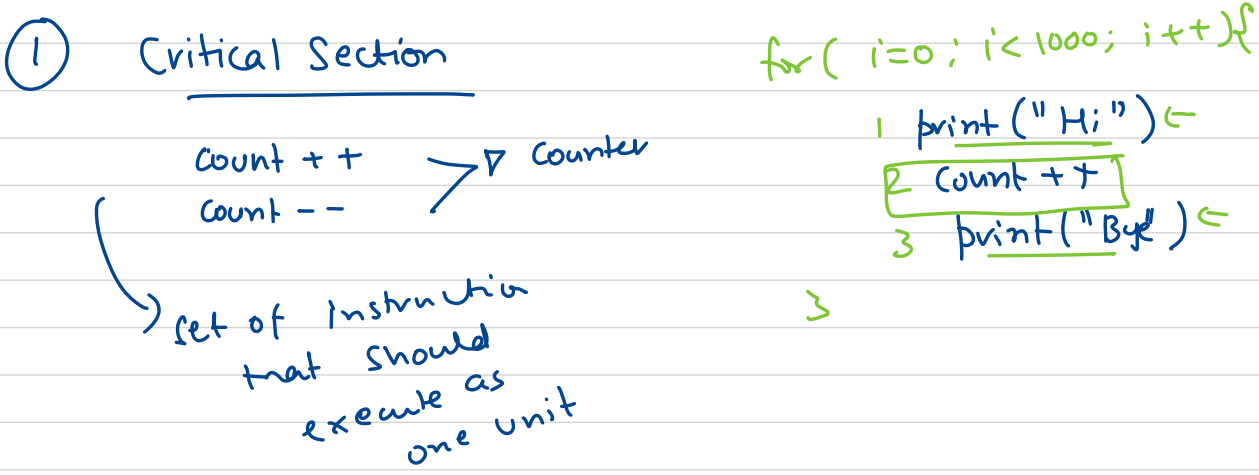



T1
cnt = 0 1 -1
RAM
T2
⇒ [Cnt ++]
Critical Section
Cnt --
Critical section
1 1. Read cnt → X 0
2 2. X → X + 1 1
3 5. Write X → cnt 1
1 3. Read cnt → X 0
2 4. X → X - 1 -1
3 6. Write X → cnt
one piece
T1
T2
1,2
3,4

Synchronisation Problem:

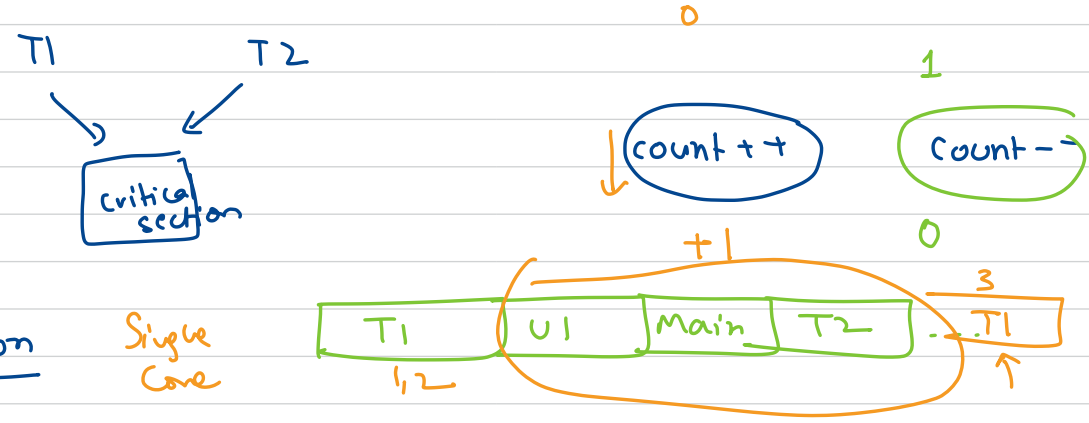
When we have more than one thread working on the same object at the same time, it can lead to inconsistent state and wrong results.

What condition can lead to synchronisation problems.



(2) Race Condition

when more than one thread tries to enter CS at same time.

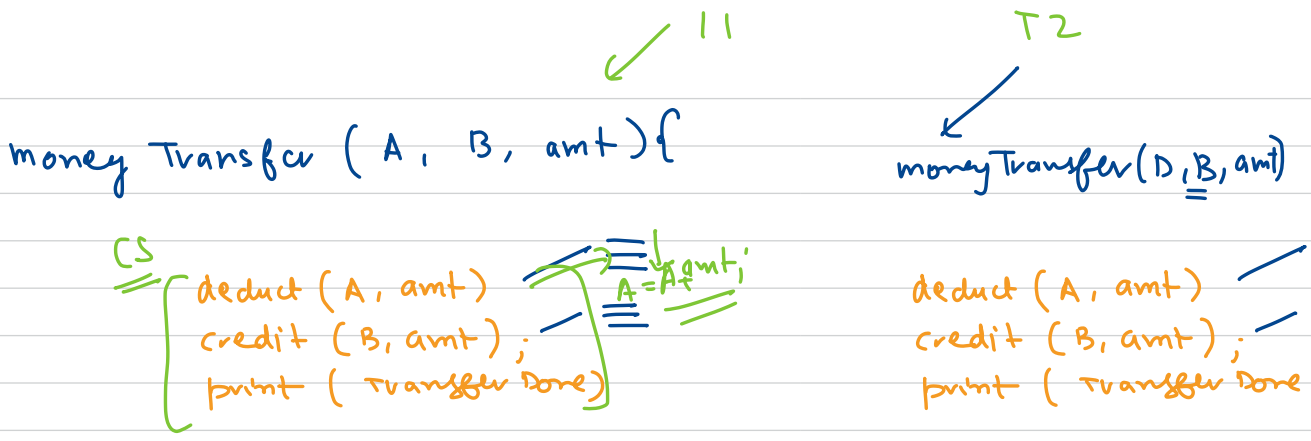


(3) Pre-emption

Single Core

Sync Problem occurs when a program in its critical section is preempted by CPU To do some other tasks, it can lead to inconsistent state.

Ideally: A CPU must complete all the tasks in the critical section and then allow other thread to enter the critical section.

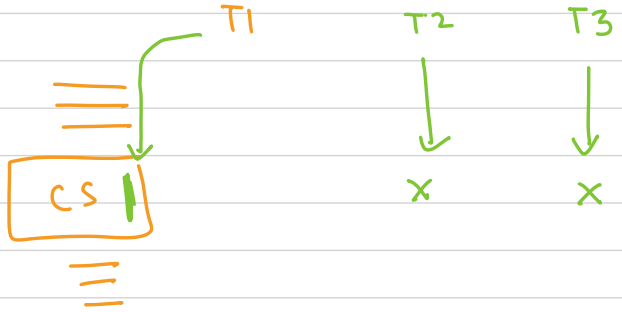

T1
T2
moneyTransfer(A, B, amt){
moneyTransfer(D, B, amt)
CS
deduct(A, amt)
credit(B, amt);
print(Transfer Done)
A=A-amt;
deduct(A, amt)
credit(B, amt);
print(Transfer Done

3

Ideal Solution for Sync Problem :

① Mutual Exclusion

A good solution should allow only one thread to enter its critical section at one time.

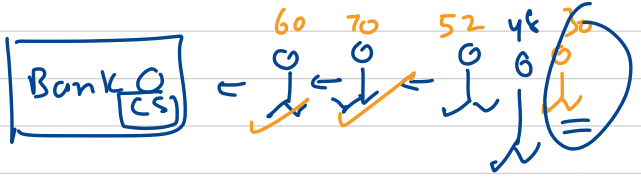


(2) Progress

Overall system should be moving or making progress.

(3) Bounded waiting

No thread should be waiting infinitely.



FIFO

(4) Busy waiting

opl → you continuously check (Busy waiting)

T2 opl → T1 comes out of CS and notifies T2()

Ideally soln must have some kind of notification system.

Should not have Busy waiting.

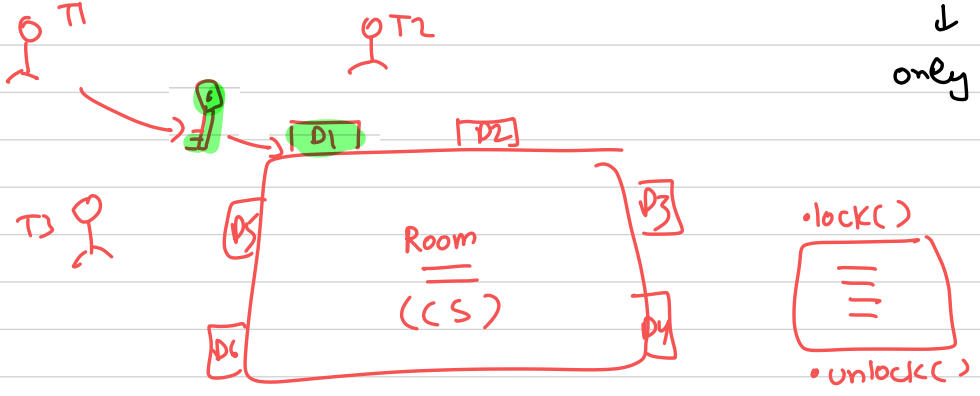
Sol: (1) MUTEX

↳ Mutual Exclusion, Mutex lock

↳ A lock enables ~~mutual~~ mutual exclusion.

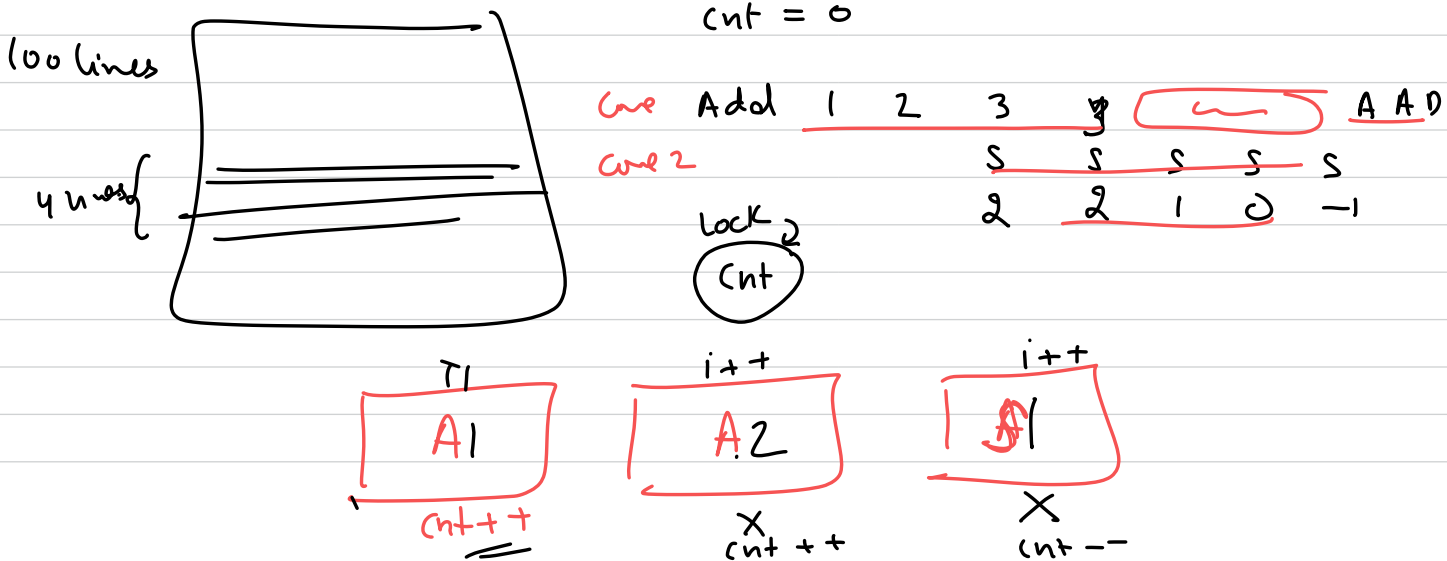
↓

only one thread can enter CS at one time.



- A thread must take a lock before it enters CS
- They must remove the lock as soon as they leave critical section.

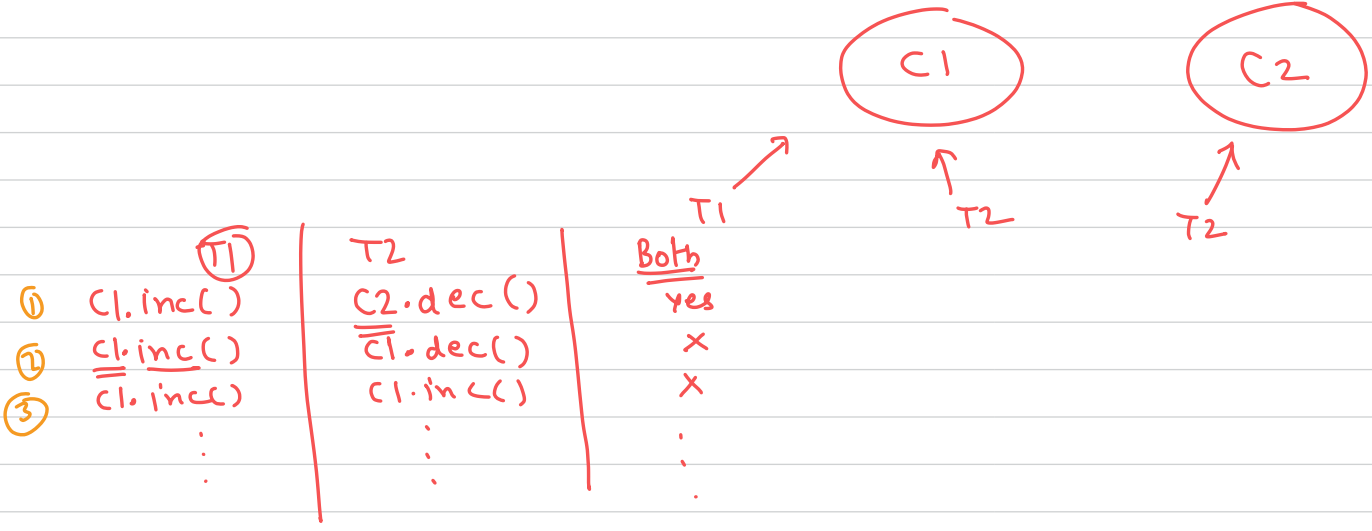
The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.



10:40

Synchronized method: If you declare a method of a class as synchronized, only one thread can be in any sync method of that object at one time.

It's a way to prevent concurrency without handling by the client.



Sync Problem