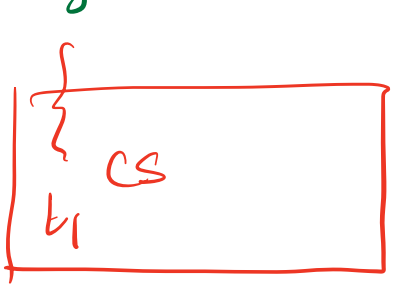
Agenda :

1. Properties of a good solution to data sync. problem.
2. Solution → Mutex.
3. Producer / Consumer Problem.
4. Semaphores.

Properties of Good Solution to Data Sync Problem

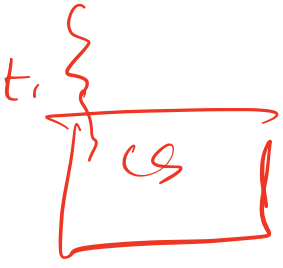
1. **Mutual Exclusion:**
   - One thread to work on the critical section of a variable at a time.
   - Restrict the number of thread to 1 for a critical section of a variable.

2. **Progress:**
   - Overall system must always make progress.



→ System is stuck ✗

```
addQuestion() {                    deleteQuestion() }
cs { List<Question> q;             cs { List<Question> q;
}                                  }
```



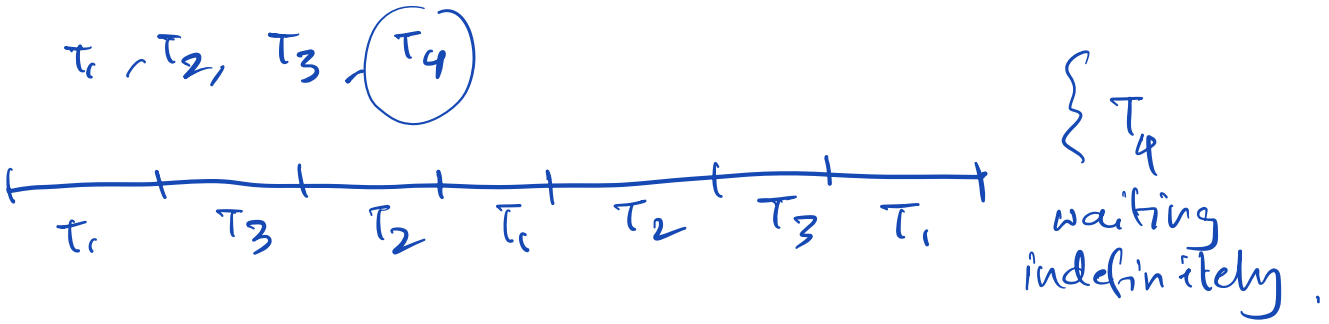
t_2 has to wait

But overall system shouldn't

wait or feel like stuck.

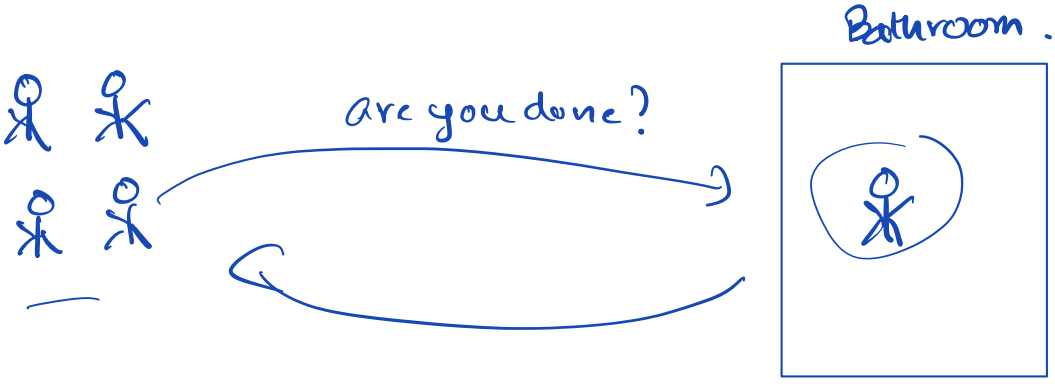
3. Bounded Waiting:

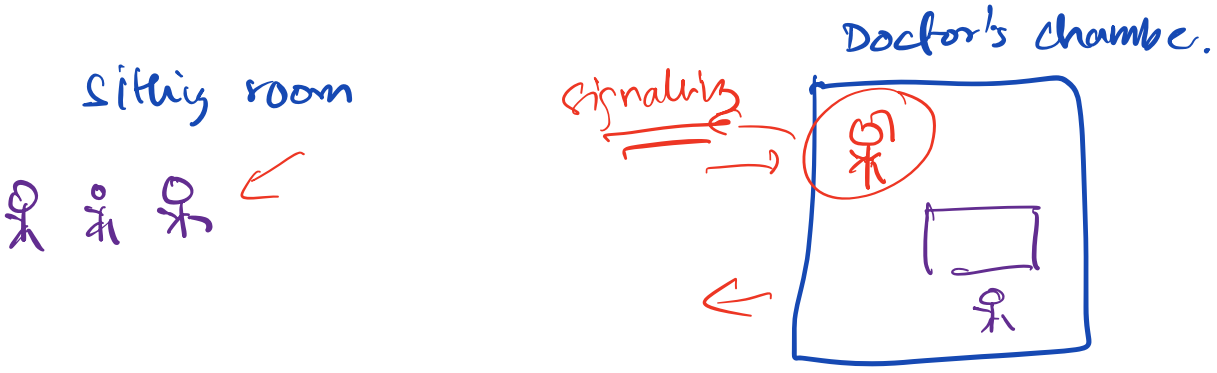
- No thread should wait indefinitely.



- Every thread should get equal amount of CPU time

4. No Busy waiting:





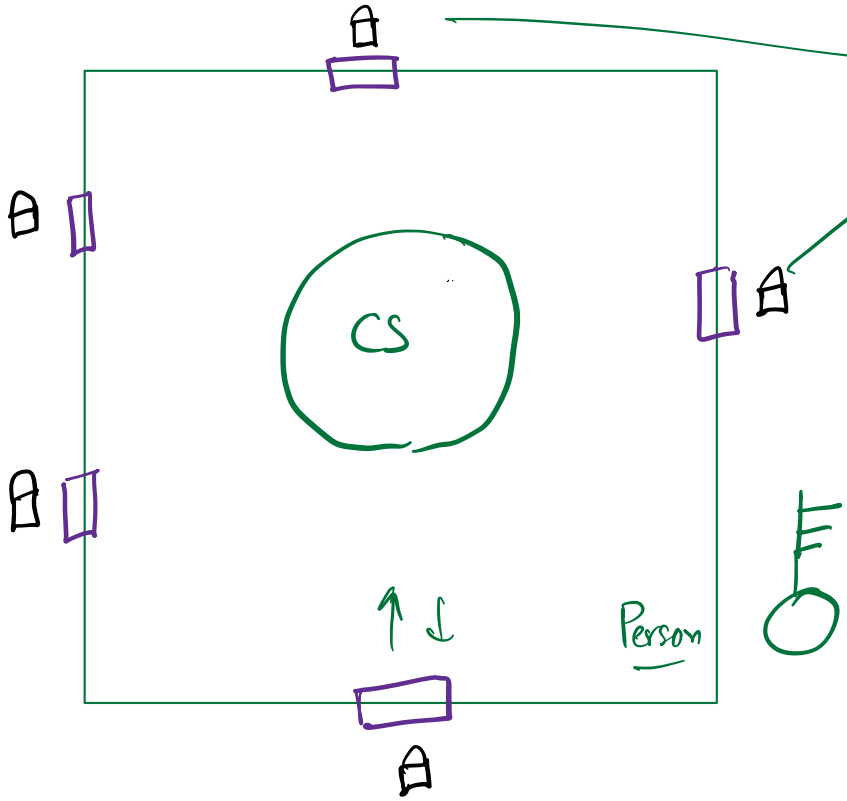
- Threads should not continuously check if CS is free. It is waste of resource.

```
while (true) {
    if (available) {
        ==
    }
}
```

→ Resource waisting

Mutex → Mutual Exclusion.

↳ Let's us restrict one thread to enter a critical section.



identical locks

One key opens all

Adder.

lock() t_1

v.val += 1

unlock()

Subtractor

lock() t_2 wait

v.val -= 1.

unlock().

8:10 → Break

Synchronized keyword

1. Synchronized method
2. Block synchronization.

} Mutual Exclusion

1. Synchronized method.
   - Implicit lock (Every object has an implicit lock)

```
synchronized public void add () {
    ==========
}
```

2. Block synchronization:
   - when you don't want to take lock over whole function

```
public void add () {
    ==========
    synchronized (this) {
        ==========
    }
}
```

→ Thread Safe / synchronised

Class Something {

synchronized void func1(){-}

void func2(){-}

synchronized void func3(){-}

}

Something A = new Something();

Something B = new Something();

{ A.func1() , { A.func1()

t_1 ↪ takes lock t_2 ↪ waits

A.func1() , A.func3()

t_1 → takes lock t_2 → wait.

A.func1() , A.func2()

t_1 → takes lock t_2 → start running func2 without any lock.

A.func1() , B.func1()

t_1 → takes lock t_2 → takes lock.

B.func3()

$t_3 \rightarrow$ wait

Class Account{

sync. phonePay();

sync GPay()

}

Acccount a = new Account();

a.phonePe()
$t_1 \rightarrow$ take lock

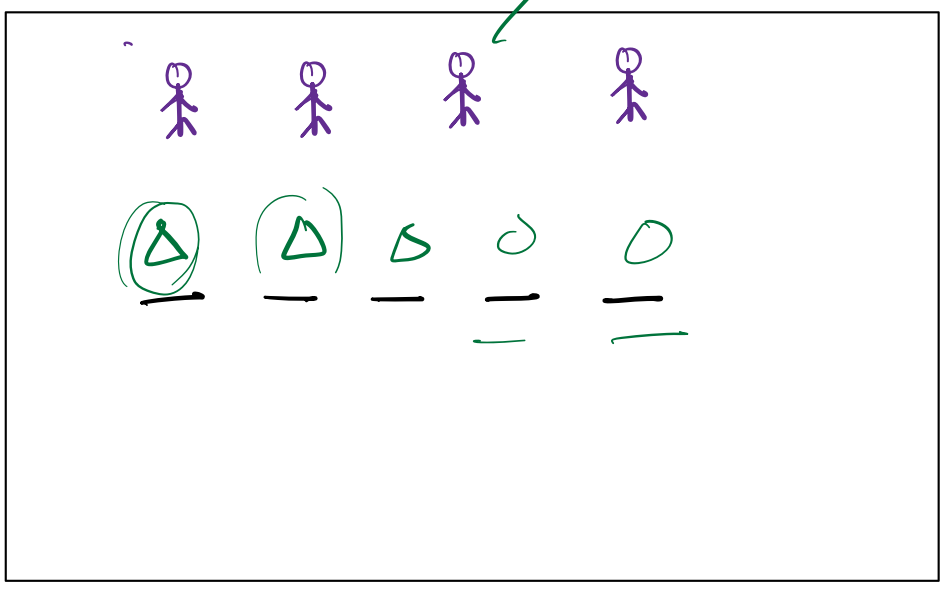
a.gpay();
$t_2 \rightarrow$ wait

Account b = new Account();

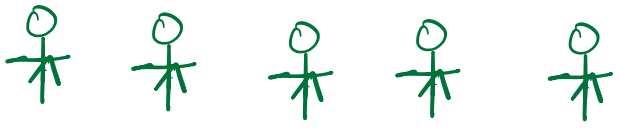
a: phonepe()
$t_1 \rightarrow$ tak

b. phonepe();

Producer / Consumer :



Producers: they will produce items when shelf with empty. store.



Consumers will consume item when it is available.

```
Class Store {
    int maxsize; → # of shelves.
    List<Object> itemsList;

    getSize() → current size
```

getMaxSize().

addItem()

removeItem().

2

addItem

P_2 P_1 = 4

$P_1 \rightarrow 4 < 5$

$P_2 \rightarrow 4 \not< 5$

```
if (items.currentSize < maxSize)
{
    item.addItem();
}
```

P_1 P_2

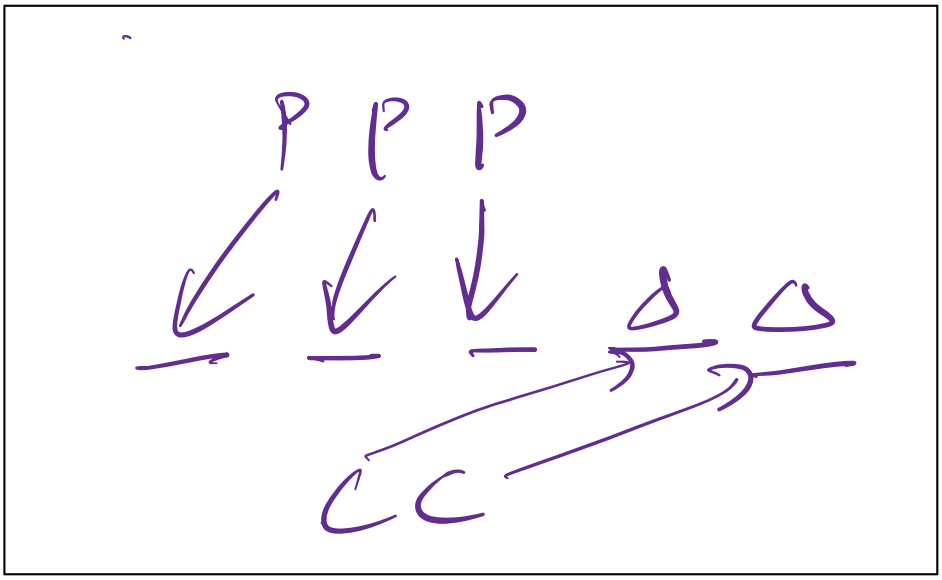
4, 5, 6

RemoveItem

C_1 C_2 → 1

```
if ( items.currentSize >= 0 ) {
    item.removeItem();
}
```

{ } C_1 C_2



no of producer + no. of consumers
= no. of shelves.

✓ 2P, 3C ✓
✓ 1P, 4C ✓
✓ 4P, 1C ✓
4P, 3C ✗

Need a capability of having fixed number of threads to work on a critical section.

↳ ? locking + count

Semaphores → It is a mutex lock with count