Agenda : ~~(1) Mutex on IDE~~

~~(2) Synchronized~~

~~(3) Producer & Consumer Prob.~~

~~(4) Semaphores~~

(5) Concurrent Data Structures

10-15mins

↳ (1) Atomic Datatypes

(2) Concurrent Datatypes

(6) Deadlocks ⟹ 20 mins

next class

Class starts at 9:05 PM

(1) Mutex : mutual exclusion lock

(a) Thread must acquire a lock before entering C.S.

(b) Thread must release the lock as soon as it exits the C.S.

(c) Allows only 1 thread at a time.

```
for (int i=1; i<=10000; i++){      | for (i=1; i<=10000; i++){
    lock.lock()      T1?           |     lock.lock()  T2?
```

lock.lock()

count.value += i;

lock.unlock()

}

lock.lock() T2

count.value -= i;

lock.unlock()

}

T1

X ← count

X = X + 1

count ← X

+1, +2, −1, −2, −3, +3, +4 . . . . . ⇒ 0

lock.lock()

for (int i=1; i<=1000; i++){

~~lock.lock()~~ T1

count.value += i;

~~lock.unlock()~~

}

lock.unlock()

lock.lock()

for (i=1 ; i<=10000; i++)

~~lock.lock()~~ T2

count.value -= i;

~~lock.unlock()~~

lock.Unlock()

+1, +2, +3, ... +10000

−1, −2, −3, ... −10000

for (int i = 1; i<=10000, i++) {

try to acquire a lock before the C.S.

count. value += i

}

T1 ~~~ false

T2 ~~~ true

(2) Synchronized => not an OS concept more of a Java concept.

Adder Sub prob => count }
lock

there is an implicit lock available for the count variable. we are not seeing the implicit lock but it is there.

Adder	Subtractor
print (Hi)	print (Hello)

~~lock.lock()~~
X = Read Count
X += 1
update count ⇒ x
~~lock.unlock()~~

~~lock.unlock~~
X = Read Count
X -= 1
update count ← X
~~lock.unlock()~~

```
synchronized ( count ){        → lock.lock()

}                              → lock.unlock()
```

⇒ It is a good approach compared to the earlier one.

⇒ Only use this when there is a single Shared variable ⇒ guideline as nested synchronized gets complex.

⇒ count {

```
private int value = 0;

int getValue() {
    return this.value;
    }

    void incrementValue (int offset) {

        this.value += offset;
    }
}
```

Synchronized ⇒ If we declare a method of a class as synchronized, then there can be only 1 thread in 1 synchronized method of that class for the same object

T1 ——

T2 ——

Count C1 = new Count();

Count C2 = new Count();

```
class Count {
    synchronized incrementValue () { = }
    synchronized decValue () { = }
    getValue ()

}
```

T1 | T2 |

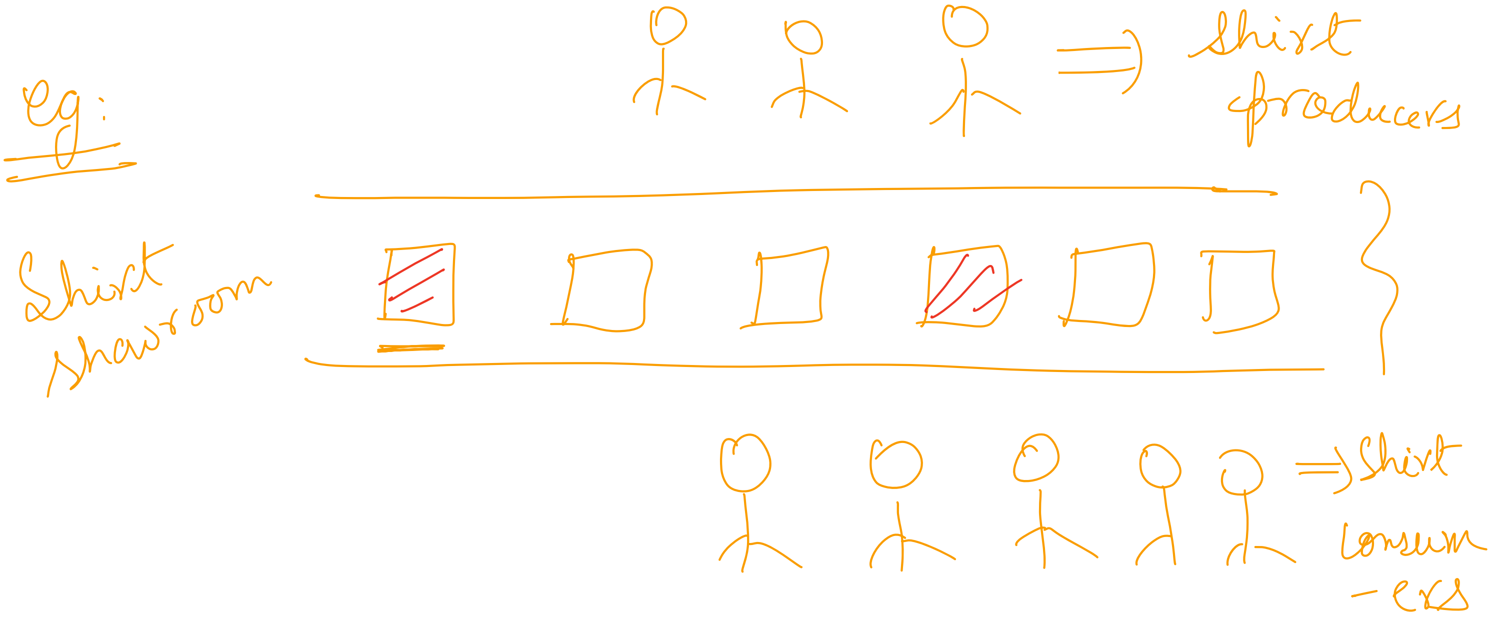
will it run parallely?

1	C1. incrValue()	C1. incValue()	✗
2	C1. incrValue() sync.	C1. decrValue() sync.	✗
3	C1. incrValue()	C1.getValue();	✓
4	C1. incrValue()	C2. incrValue()	✓

3 Producer Consumer

no. of threads to enter C.S. at a time ⟹ 1

eg:



1 I only want to allow a producer inside the showroom if there is an empty slot.

② I only want to allow a consumer when there is a shirt available in the showroom.

⇒ NO shirts, No consumers

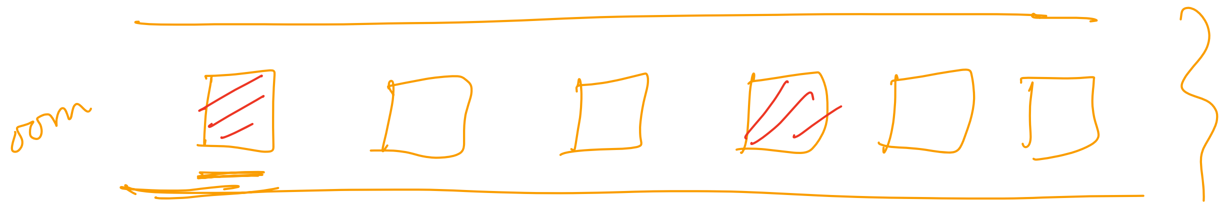
no. of producers ⇒ 6

⇒ If one slot is filled no. of producers ⇒ 5

⇒ If 2 slots are filled, I can allow ⇒ 2 consumers.

⇒ C.S. perspective

computer science.



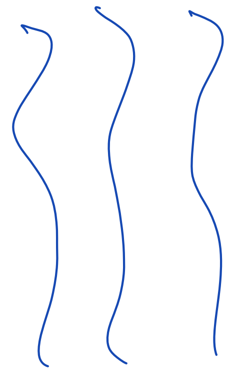
```
{
List<object> store;
maxSize = 6
}
```

Producer	Consumer

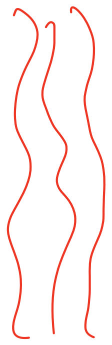
store.add( ) | store.remove()

⇒ I parallely run 100's of producers & consumers, would there be a problem?

[ ]



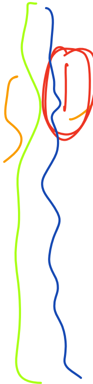
Producer	Consumer
store.add()	1 store.remove()



⇓

throw an error as there are no objects in the store.

Producer	T1 T2	Consumer
if (store.size() < maxSize) {	1	if (store.size() > 0) { ⇒ size was 1
store.add()		store.remove();
}		}

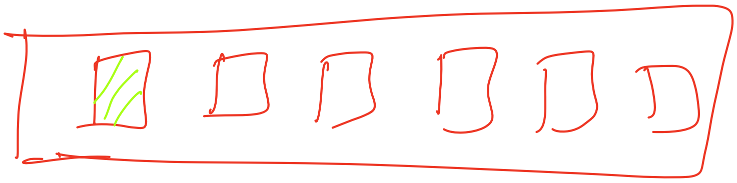


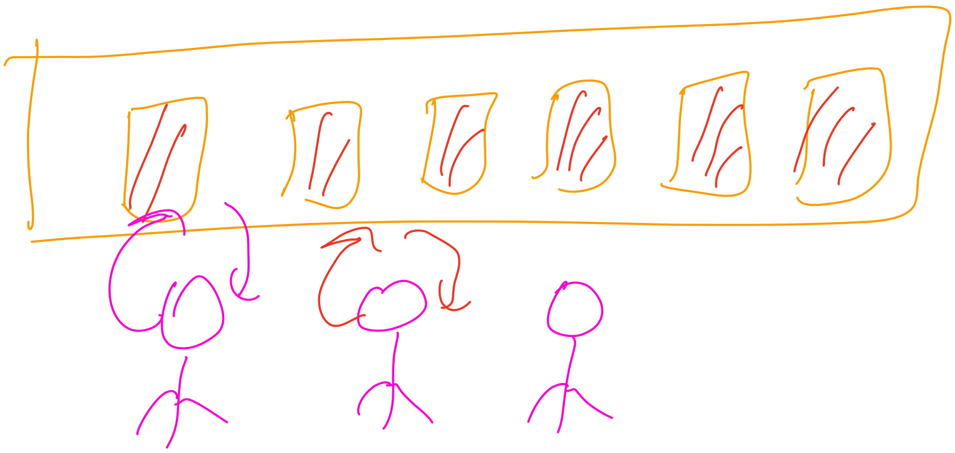
C.S.

Critical section

T1 → 1

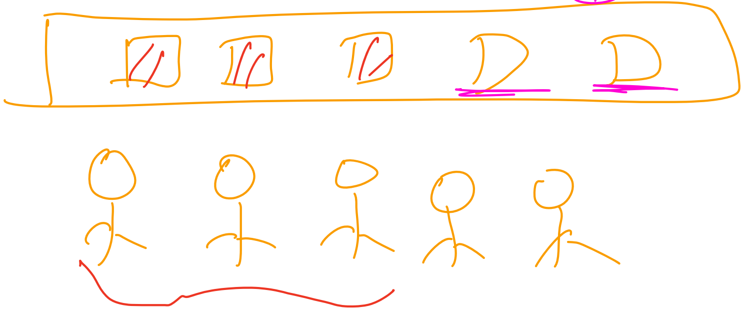
T2 → ①





Class starts at 10:45 PM

⟹ Semaphores

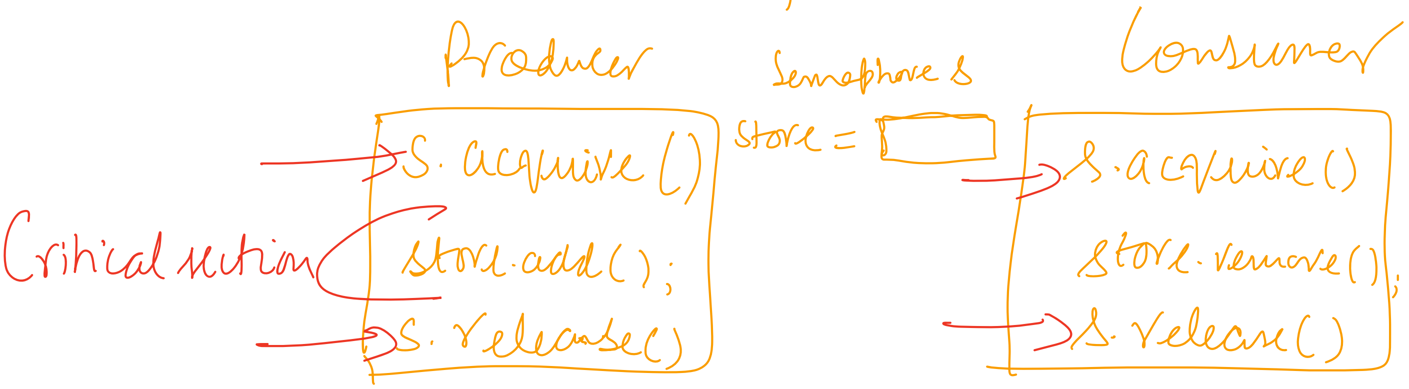


⟹ Mutex with an upper bound >= 1

```
Semaphore s = new Semaphore(2);
```

no. of threads that can enter the C.S ⟹ 2

$$\underbrace{\text{Semaphore s = new Semaphore}}_{\text{Mutex}}(1)$$



Producer

Semaphore s

Consumer

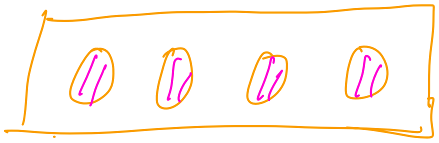
store =

```
s.acquire()
store.add();
s.release()
```

```
s.acquire()
store.remove();
s.release()
```

Critical section

Case 1: ⟹



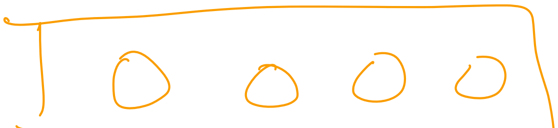
store has a size of 4
store is full
↳ 4 consumers can enter

Case 2: ⟹

⟹ producers = 2
consumers = 2

Can I say that I can create a semaphore of size 4 ⟹ no. of producers + no. of consumers that can enter at same size

store ⟹



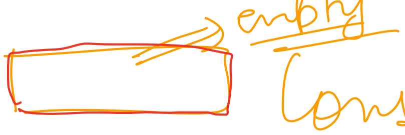
Semaphore s = new Semaphore(4)

X Case 3 ⇒ 0 producers

how many consumers can I allow ⇒ 4

this is not what we wanted

The reason for this is we are tracking 2 diff. type of tasks using a common Semaphore



Producer

```
→ p.acquire()
  store.add
→ c.release()
```

Semaphore
c = 0

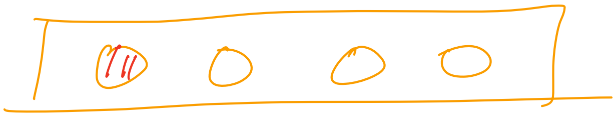
Semaphore
p = 4

Consumer

```
c.acquire()
store.remove()
p.release()
```

P → 4 → 3 → 4

C → ~~0~~ → 1



① first point to note in Semaphore, a thread can release a lock even if they didn't acquire it.

② How many people) ⇒ consumers ⇒ 1

Can enter the store after 1 shirt has been added } ⇒ producers ⇒ 3

eg: []

(4) P → 3 → 2 → 1

(0) C → 1 → 0

1 P1. acquire

2 P2. acquire

3 C1. acquire ⇒ nothing will happen, it will wait

4 P3. acquire

5 C. release()

6 C1. acquire ⇒ ~~nothing will happen, it will wait~~

will resume

X will call release for Y

Producer is signalling that I have done my job, now you can go & do your job.