1. Good Evening
2. We will begin at 9:08 pm
3. Topic – Semaphores, Atomic Types, Deadlocks.

Agenda

1. Recap ✓
2. Semaphores
   - → Producer - Consumer Problem ✓
   - → Solution using Semaphores ✓
   - → More problems ✓
3. Atomic Data types ✓
4. Deadlocks

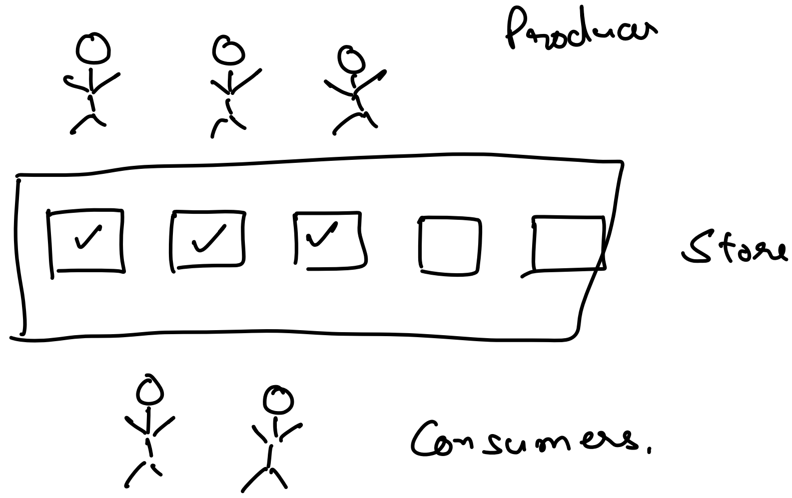
Recap

→ Adder Subtractor Problem
→ Synchronization Problems
→ $\underbrace{\text{Mutex \& Synchronized}}$

They only allow one thread to enter the critical section.

Semaphores

1. Producer Consumer Problem




* Allow a producer to go in the store if there are empty slots.

* Allow a consumer to go in the store if there are fill slots.

# producers allowed in the store = # of empty slots

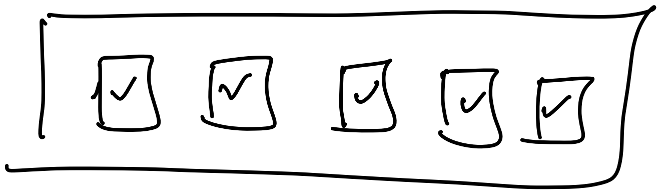
# consumers allowed in the store = # of filled slots

Code

```
List <object> store
int capacity
```

| Producer | Consumer |
|---|---|
| store.add(new Shirt()) | store.remove() |

Problem1



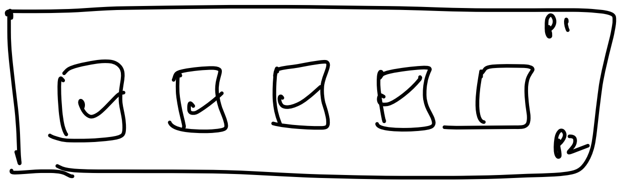
Soln 1.

| Producer | Consumer |
|---|---|
| if (store.size() < cap)<br>store.add(new Object()) | if (store.size() > 0)<br>store.remove() |

Problem2 → What happens in case of multi-threaded application.

| | t1 | | t2 |
|---|---|---|---|
| 1. | if(s.s < c) ✓ | 2. | if (s.s < c) ✓ |
| 3. | s.add ✓ | 4. | s.add ✓ |

P1 P2




Soln → Mutex Lock | Synchronized.

lock store

**Producer**

```
lock.lock()
    if (store.size() < cap)
        store.add(__)
lock.unlock()
```

**Consumer**

```
lock.lock()
    if (store.size() > 0)
        store.remove()
lock.unlock()
```

t1

```
✓ l.lock()
✓    if(s.s < c)
✓       s.add()
✓ l.unlock()
```

P1

t2

```
✓ → l.lock()
       if (s.s<c)
          s.add()
    l.unlock()
```

P2

| l.lock() | x | if | x | s.add | X |
|---|---|---|---|---|---|
| t1 | t2 | t1 | t2 | t1 | t2 |

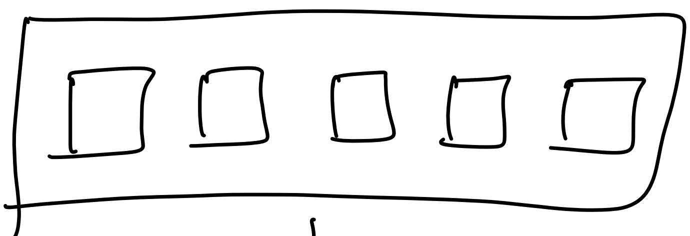
| l.unlock() | l.lock() |
|---|---|
| t1 | t2 |

Problem3 → $\boxed{\text{We have solved something else.}}$

Solution3 → Semaphores

Mutex vs Semaphores

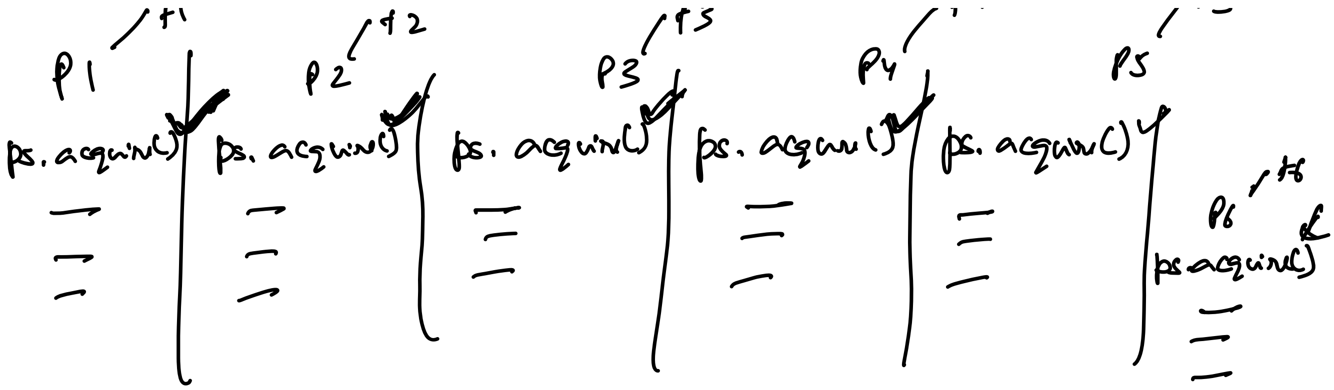
| Mutex | Semaphores |
|---|---|
| → Only one thread inside CS | → Can allow more than one inside CS |
| Lock l = new ReentrantLock() | Semaphore ps = new Semaphore(cap) |

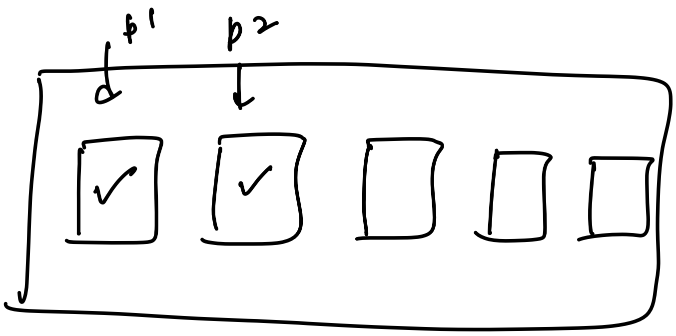


| Mutex | Semaphores |
|---|---|
| → l.lock() | → ps.acquire() → decrease the no' of threads by 1 |
| → l.unlock() | → ps.release() → Increase the no' of threads allowed by 1 |

count = ~~5~~ ~~4~~ ~~3~~ ~~2~~ ~~1~~ 0

ps = new Semaphore(5)




producers = ~~5~~ ~~4~~ 3

consumers = ~~0~~ ~~1~~ 2

ps = new Semaphore (cap)

cs = new Semaphore (0)   p = ~~5~~ 4

c = 0

ps, cs

**Producer**

l.lock()  ps.acquire()

if (s.s() < cap)

s.add()

cs.release()

l.unlock()

**Consumer**

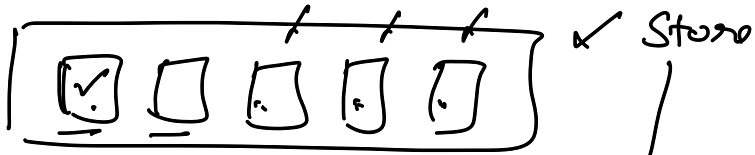
l.lock()  cs.acquire()

if (s.s() > 0) {

s.remove()

}

l.unlock()  ps.release()




a p → ~~5~~ ~~4~~ ~~3~~ 4

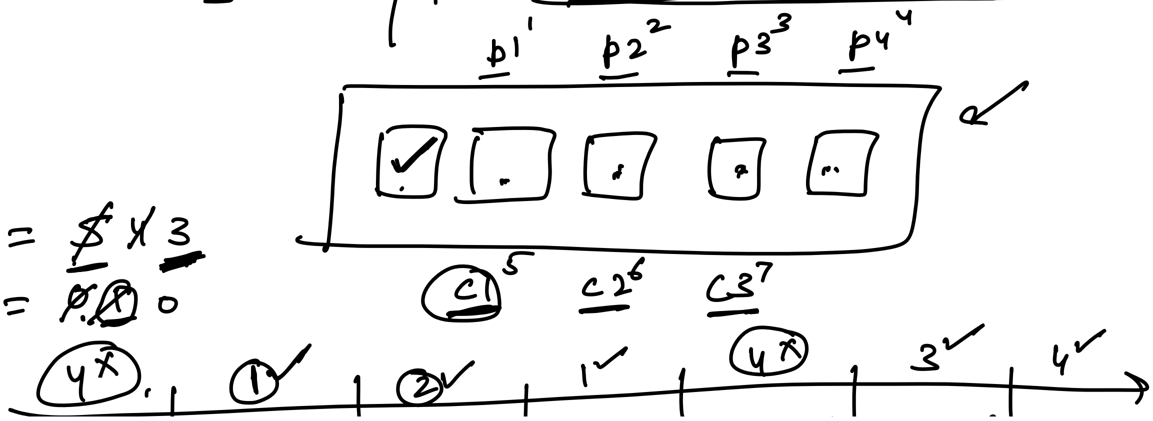
a c → ~~0~~ ~~1~~ ~~2~~ 1

S = Semaphore(3)          ~~3~~ ~~2~~ ~~1~~ 0

| t1 | t2 | t3 | t4 |
| --- | --- | --- | --- |
| ✓ s.acquire() | ✓ s.acquire() | ✓ s.acquire() | $\boxed{\text{s.acquire()}}$ ✗ |

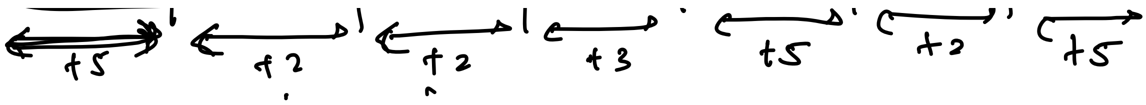
ps, cs

| Producer | Consumer |
| --- | --- |
| 1. ps.acquire() ✓ | 4. cs.acquire() ✓ |
| 2. $\boxed{\text{s.add()}}$ | 5. $\boxed{\text{s.remove()}}$ |
| 3. cs.release() ✓ | 6. $\boxed{\text{ps.release()}}$ ✓ |



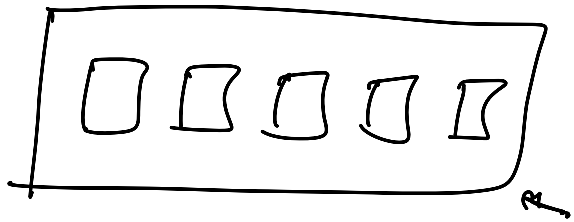

ps = ~~5~~ ~~4~~ 3

cs = ~~0~~ ~~1~~ 0




ps → 5

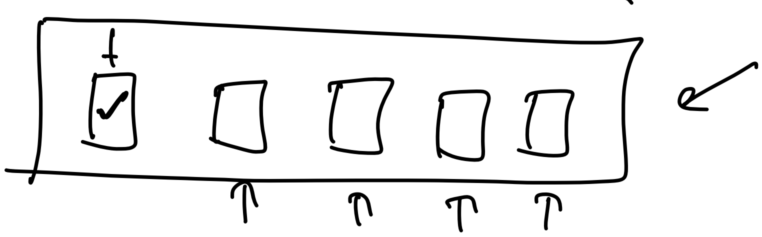
cs → 0



capacity

ps = new Semaphore(5)

cs = new Semaphore(0)

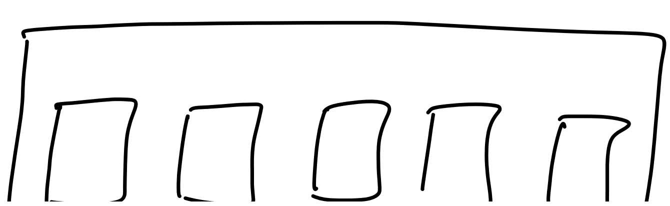



ps = [~~5~~ 4]

cs = [~~0~~ 1]

$$\left\{\begin{array}{l}\text{ps.acquire()} \longrightarrow \text{decrements} \\ \text{ps.release()} \longrightarrow \text{increments}\end{array}\right.$$

→ store Concurrent



ps = $\cancel{0}$ 4

cs = $\cancel{0}$ 1

ps. acquire()

cs. release()

cs. acquire()

ps. release()

Do you want a discussion? or code?

Break : $\boxed{\text{10:26 to 10:36}}$

---

Atomic Data Types

↳ Adder Subtractor

Adder

Count

Subtractor

count ++ | count --

Count = ~~10~~ ~~11~~ 9

t1 | t2
--- | ---
✓ Reg = Read Count | ✓ Reg = Read Count
✓ Reg = Reg + 1 | ✓ Reg = Reg - 1
✓ Count = Reg | ✓ Count = Reg

PCB^{t1}: Reg = ~~10~~ 11

PCB^{t2}: Reg = ~~10~~ 9

lock

lock.lock()
count ++
lock.unlock()

lock.lock()
count --
lock.unlock()

int → AtomicInteger ✓
bool → AtomicBoolean ✓
long → AtomicLong ✓

lock

Even without using lock your operations

count — will work fine.

Code → AtomicInteger.

| | |
|---|---|
| count++ ✓ | ++count ✓ |
| int x = count.getAndIncrement() | int x = count.incrementAndGet() |

count += i

count--

count = 10

int x = count++ ✓

x , count
10 , 11

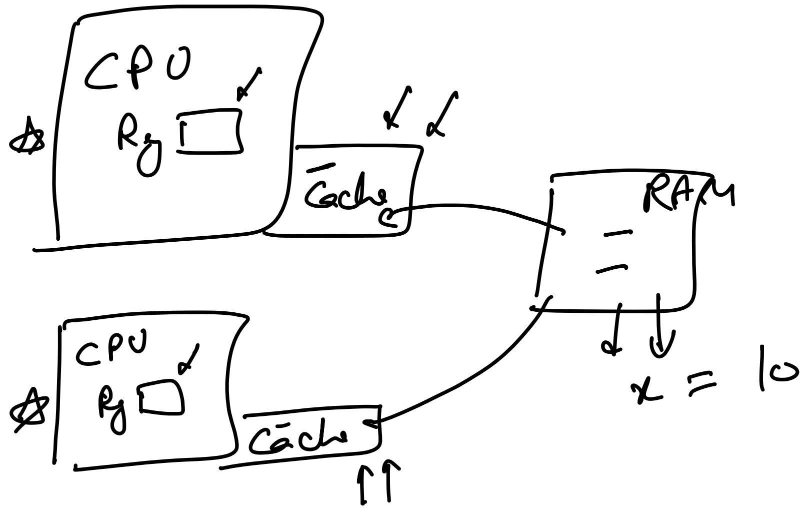
10

int x = ++count

x , count
11 , 11

---

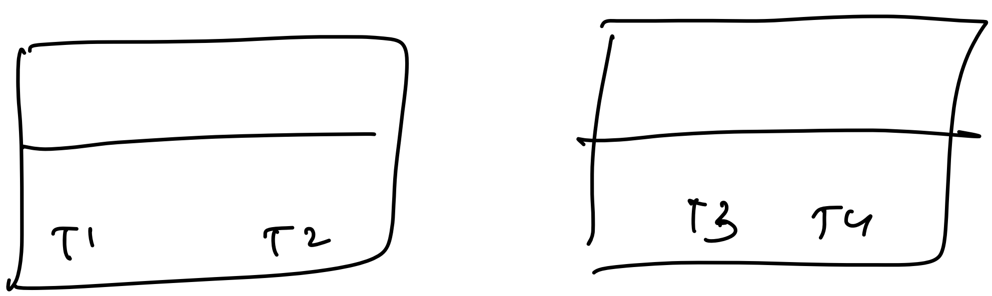
volatile → CPU will ~~g~~get the value of a variable from RAM always & not use the one in CPU cache.




t1 → CORE1

volatile

t2 → CORE2




P1 P2

Deadlocks : When no thread of an application is able to progress.

When can this happen?

ml1        ml2

{ t1

✓ ml1. lock()
✓ ml2. lock() (Block)

ml2. unlock()
ml1. unlock()

{ t2

✓ ml2. lock()
✓ ml1. lock() Block

ml1. unlock()
ml2. unlock()

| | | |
|---|---|---|
| t1 | ml1 ✓ | ml2 ✗ |
| t2 | ml2 ✓ | ml1 ✗ |

{ t1

✓ l. lock()

{ t2

✓ l. lock() Block()

✓ l.unlock()

---

ml1 ml2

t1

✓ ml1.lock()

☆ ml2.lock() [Block]

ml2.unlock()

ml1.unlock()

t2

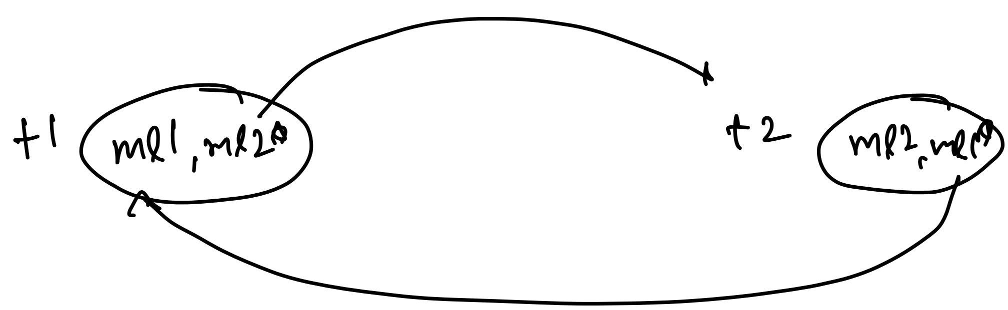
✓ ml2.lock()

☆ ml1.lock() Block

ml1.unlock()

ml2.unlock()

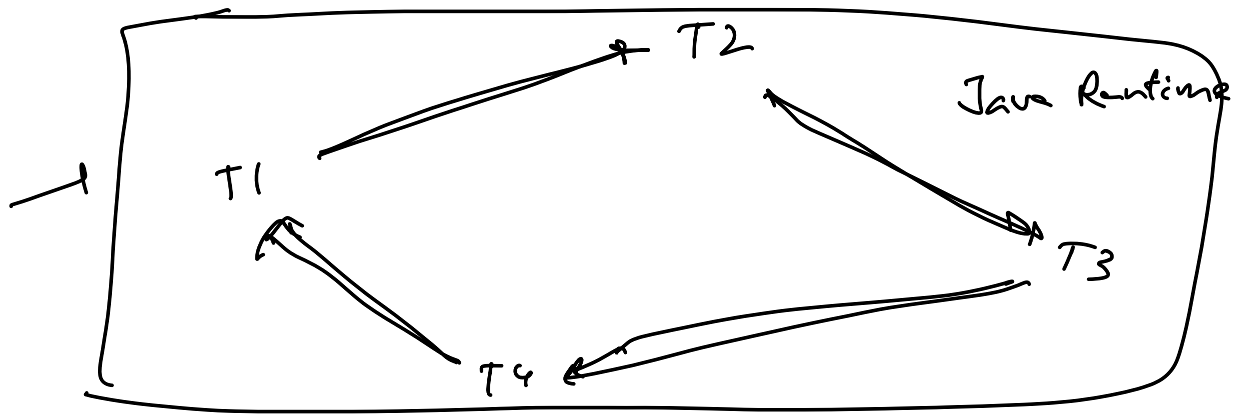
| | | |
|---|---|---|
| t1 | ml1 ✓ | ml2☆ ✓ |
| t2 | ml2 ✓ | ml1☆ ✓ |




→ Identify the deadlock

1. Timeout ✓ →

2. Graph Cycles. → Detection of deadlocks.




→ Resolve the deadlock

1. Avoid deadlocks
   - Always take locks in same order

t1 {
✓ ml1. lock()
ml2. lock()

t2 {
✓ ml1. lock()
ml2. lock()

2. $\boxed{\text{Ignore deadlock}}$ → User kill the process
   ↓
   OS

3. $\boxed{\text{Resolve dealock}}$ → By killing one of the threads

DBMS MySQL

Deals by killing/ rollbacking transaction

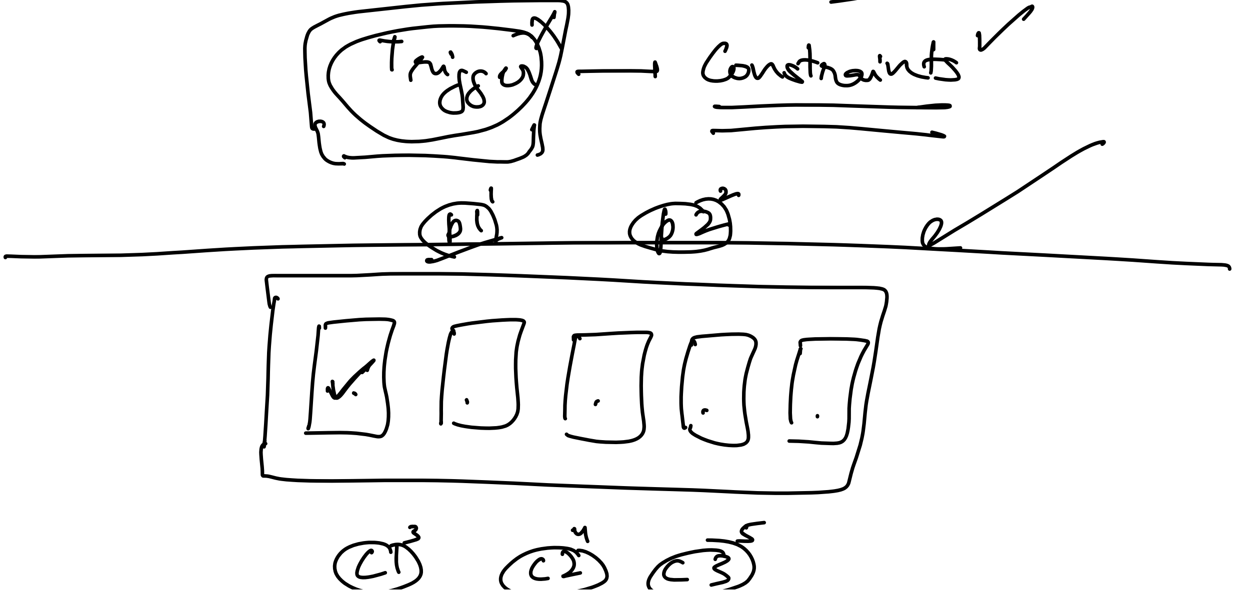
Deadlock Code → MySQL

→ Next class → Memory Management

→ Monday : DBMS

- Deadlock ✓ 5
- Full-text ✓ 5
- SP ✓ 1
- UDFs ✓ 30
- Doubts

Trigger → Constraints ✓



Producer

1. ps. acquire() ✓
2. $\boxed{\text{s.add()}}$
3. cs. release() ✓

Consumer

4. cs. acquire() ✓
5. s. remove()
6. ps. release()

Concurrent Queue
synchronized

ps : ~~5~~ 4
cs : ~~0~~ ~~1~~ 0