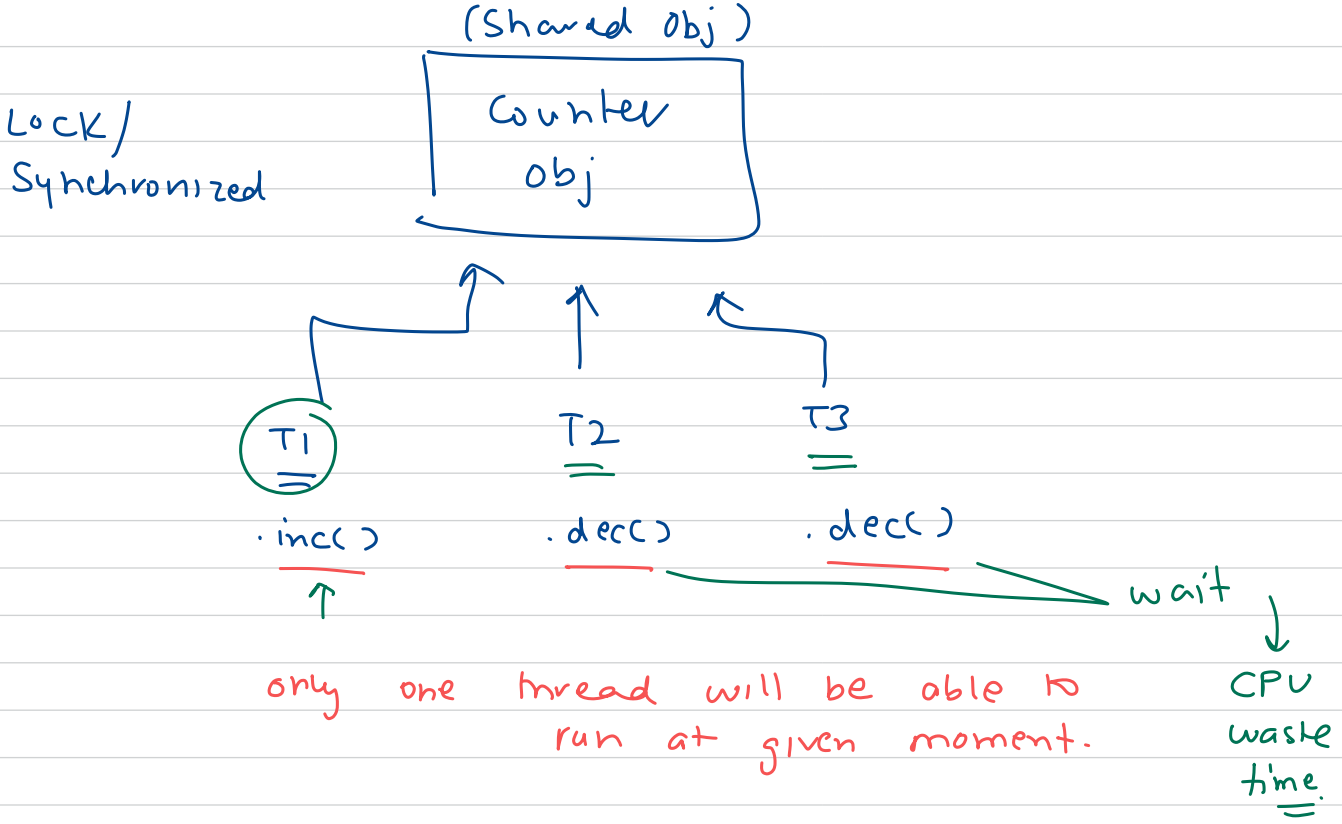
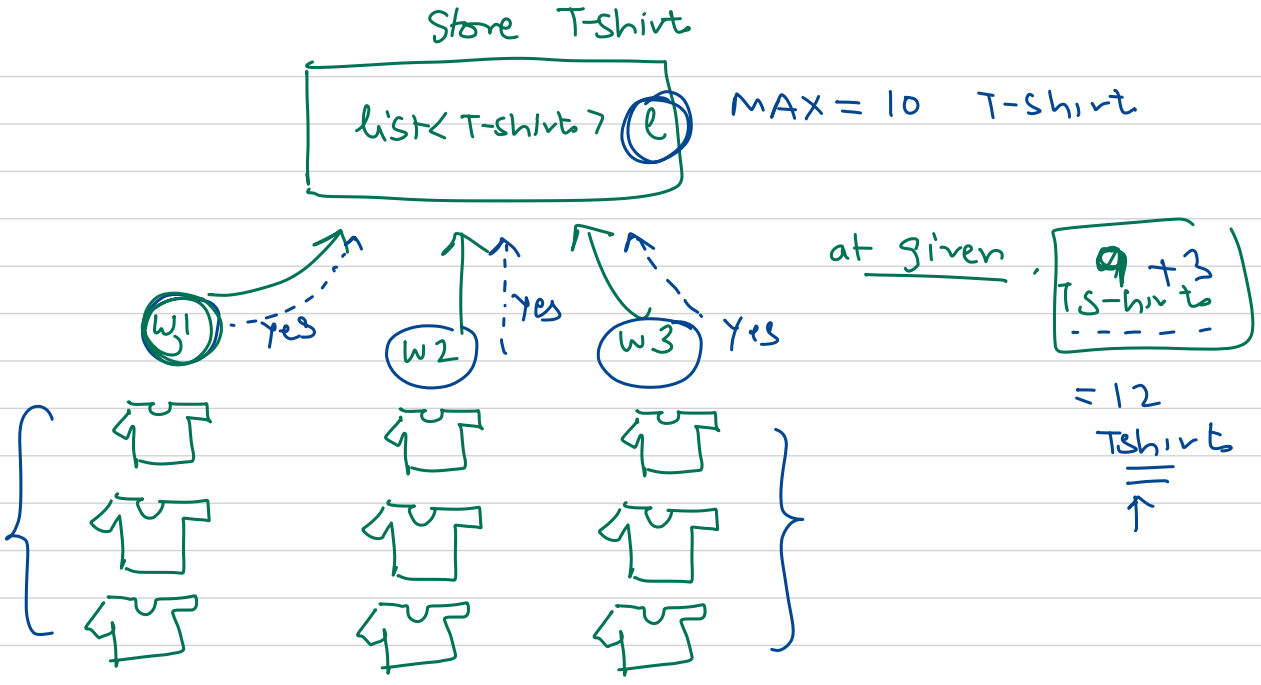
Synchronisation using Semaphores

1) Lock
2) Synchronized (Methods, Block)
3) Atomic Datatypes & Concurrent Data Structures
4) Semaphores (more powerful as you can allow multiple threads to enter critical section and acess shared Resource)

↓

another mechanism to implement synchronisation.


(Shared Obj)
Counter
Obj
Lock/
Synchronized
T1
T2
T3
.inc()
.dec()
.dec()
wait
only one thread will be able to run at given moment.
CPU waste time.



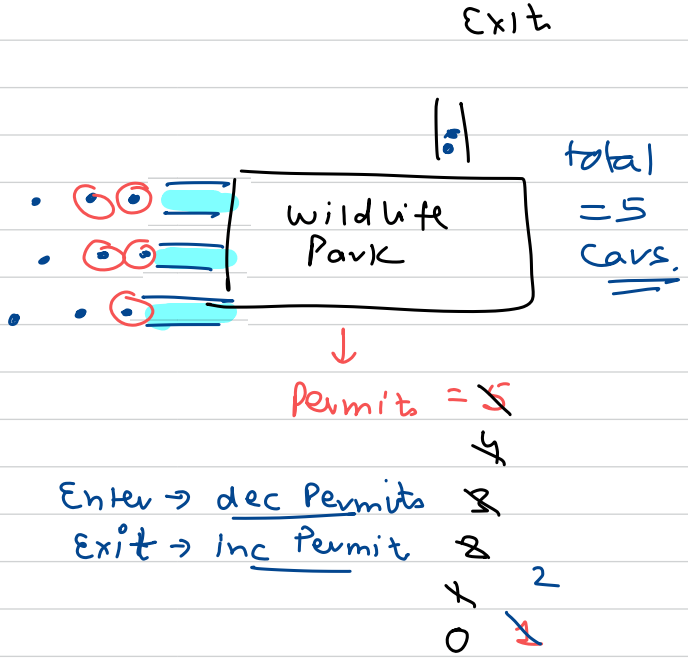
- No Synch, → More T-shirts that our store can handle.
- Sync → other workers will have to wait (sequential) more time.

Semaphores

It is mechanism that can be used to control access to a shared resources In concurrent programming.

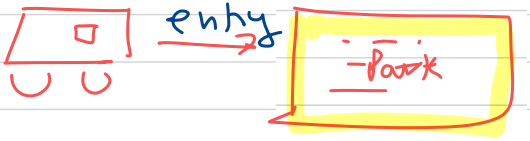
In Java,

- Binary Semaphore
- Counting Semaphore



```
int MAX_PERMITS = 5;

Semaphore semaphore = new Semaphore(MAX_PERMITS);
```



5

Park ⇒ CS

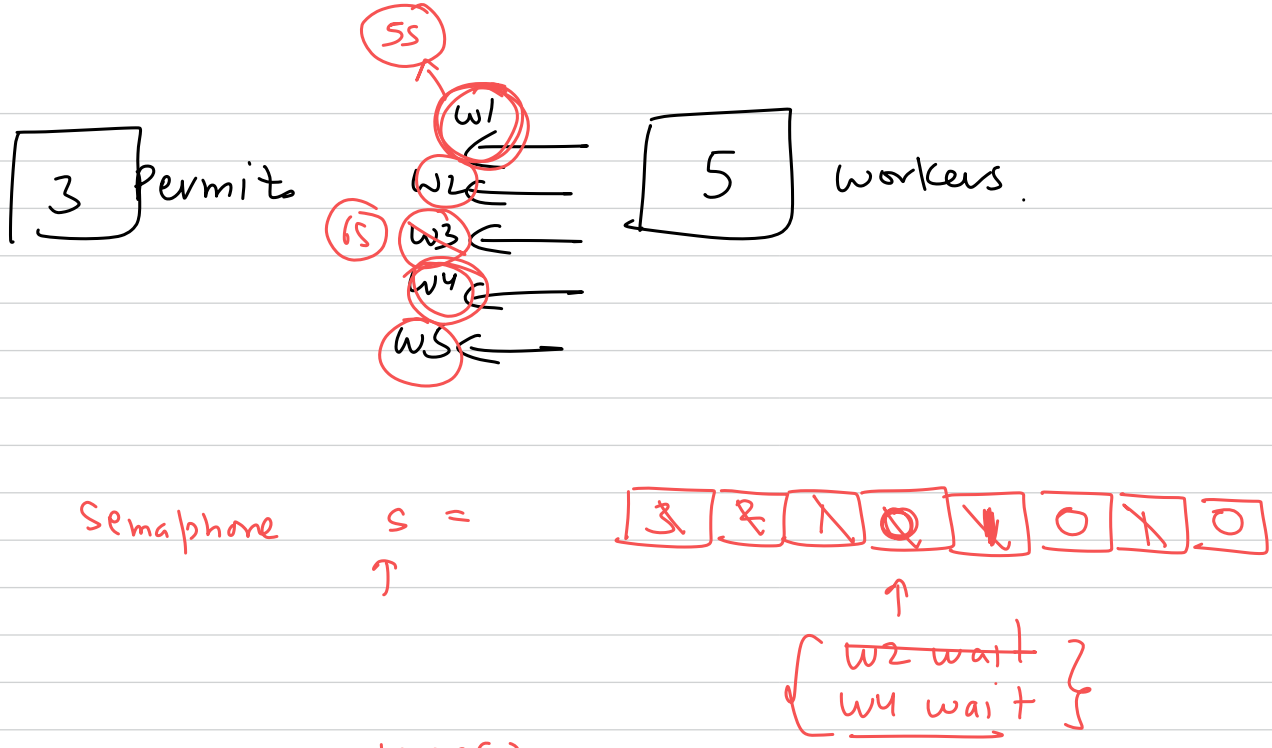
semaphore.acquire();

4

Car ⇒ Thread.

semaphore.release()

5

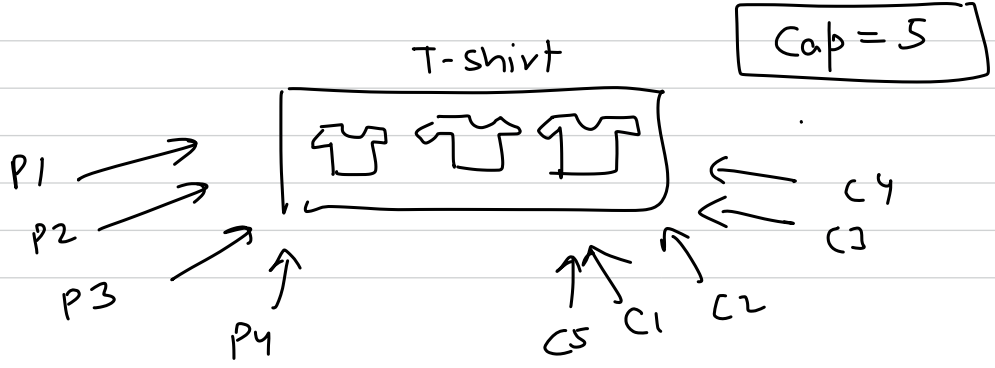

3 permits
5 workers
S5
W1
W2
W3
W4
W5
S
3 2 1 0 1 0 1 0
W2 wait
W4 wait
s.release()
final = 3

Locks | Sychronised vs Semaphores

↑

multiple threads (N) to access CS at same time.

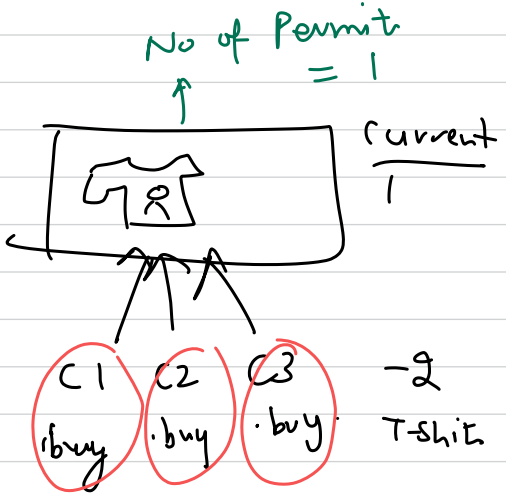
Producer-Consumer Problem

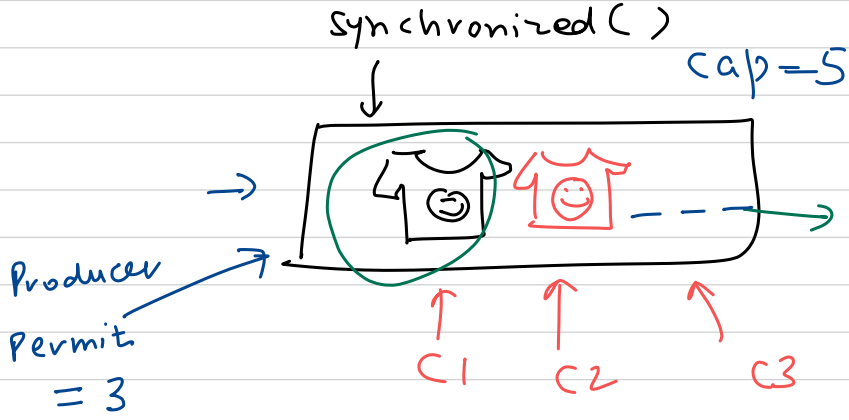


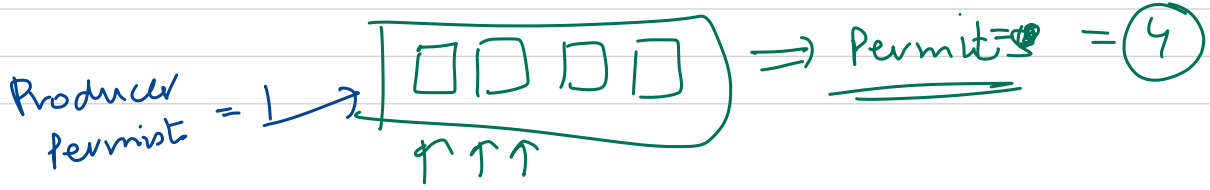
Each producer can produce many Tshirt

T1 T2 T3 - - -

- overflow ? 5 T-shirts
- underflow can happen.

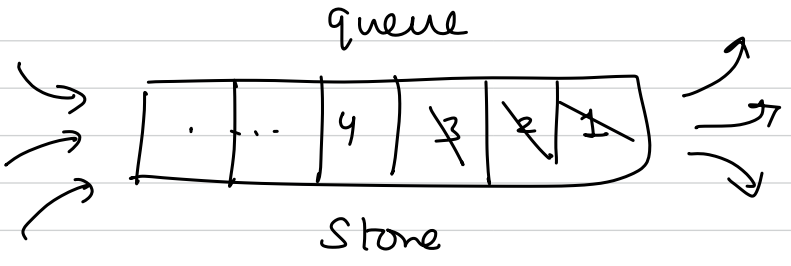


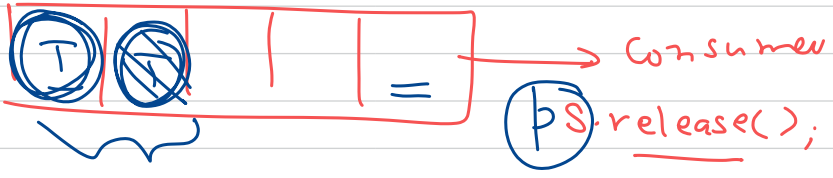




CODE

1. No sync (inconsistent output)
2. Synchronized (only one thread can access CS)
3. Semaphores. (multiple threads can access CS)

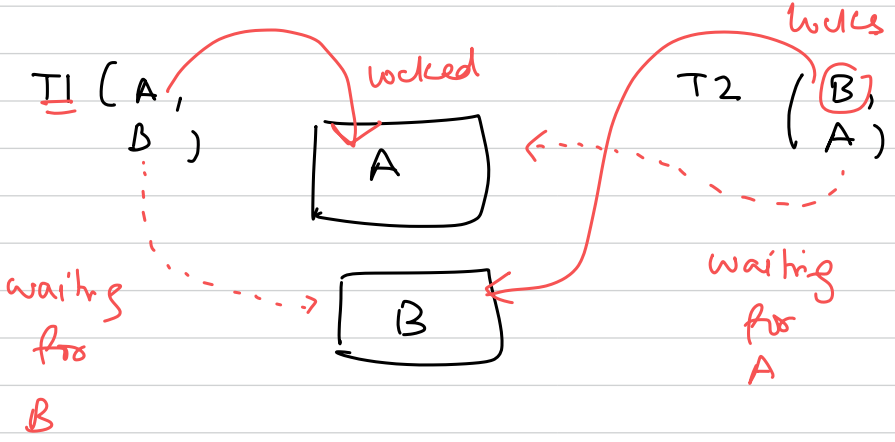




Producer semaphore → Producer
acquire() → dec permit for prod
release() → inc permit for consumer

Consumer Semaphore →
acquire() → dec permit for cons.
release() → inc permit for prod.

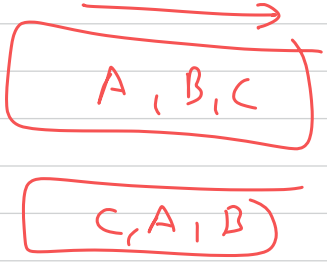
Deadlocks → a situation in which application is stuck because threads are waiting ∞ to acquire resources.



Solutions

① Avoid Deadlock

- Take locks in a fixed order
  (sort the shared objects ....)



T1 {

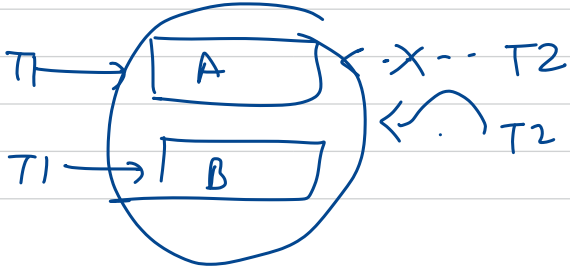
```
A.lock()
  :
B.lock()
  :
```

}

T2 {

```
A.lock()   A.lock();
  :        B.lock()
A.lock()
```

}



(2) Deadlock Recovery

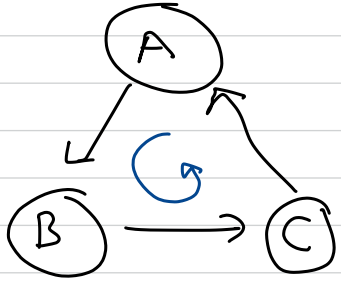
→ Deadlock Detection Algorithm

→ Timeout

→ Kill the process/thread

→ Restart

Threads
New
Runnable
Waiting



[Directed cyclic graph]

↑

Detect a cycle in directed graph.