Transactions

SERIAL

T1	**T2**
read(A)	
write(A)	
write(B)	
Commit T1	
	read(A)
	write(A)
	write(B)
	Commit T2

```
T1:
START TRANSACTION
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE

T2:
UPDATE inventory SET quantity = quantity + 1 WHERE product_id = 1
COMMIT;

T3:
START TRANSACTION
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE

T4:
UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1
COMMIT;
```

Conflict-Serializable Schedule:

1: Read Quantity in T1
2: Write Quantity in T2
3: Read Quantity in T3
4: Write Quantity in T4
5. Commit

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
write(B)	
Commit T1	
	write(B)
	Commit T2

In general, a schedule is a conflict serializable if it can be transformed into an equivalent schedule where all transactions are executed in serial order without creating any conflicts between them.

If we go by the precedence **T1-> T2 -> T3 -> T4**, then this will be an equivalent schedule as there will be no conflict, and we can swap instructions to make a serial schedule as we are first reading quantity and then adding it, then again reading quantity and sequentially decreasing from it.

Also, we are using shared lock FOR UPDATE in the transactions to avoid any inconsistencies and finally commit the changes.

Not Conflict Serializable Schedule:

1: Read Quantity in T1
2: Write Quantity in T4
3: Write Quantity in T2
4: Read Quantity in T3
5. Commit

Explanation: **W-W Conflict**

If we follow the precedence schedule **T1 -> T4 -> T2 -> T3**, then this is Not Conflict serializable.
The reason for the conflict is that T1 is followed by T4 instead of T2, which is causing conflicts and inconsistency in the database as we cannot swap the instructions in the above schedule to obtain a serial schedule like T1-> T2 -> T3 -> T4.
Also, we are using shared lock FOR UPDATE in the transactions to avoid any inconsistencies and finally commit the changes.

T1	**T2**
read(A)	
write(A)	
	read(A)
	write(A)
	write(B)
	Commit T2
write(B)	
Commit T1	