Deadline-6

Updating product stock and placing an order followed by payment:

Transaction1→ User A is placing an order for a product followed by payment

Transaction2→User B placing an order for that same product followed by payment

Conflicting case → User A and B tries to buy the same product at the same time. While B is placing an order for the product, A updates the product stock. If the stock updated by A is done before B places the order will lead to conflicting transactions. This is because for B, the product may appear in stock, whereas in reality the stock has been reduced (maybe finished). Thus, the order will not be processed for B because of A.

Example:

--Begin transaction 2: User B is placing an order for a product

```
BEGIN TRANSACTION;
SELECT productStock FROM product WHERE productID = 1; (quantity selected = 1)
IF product stock >= 1 THEN
  UPDATE product SET productStock = productStock - 1 WHERE productID=1;
  INSERT INTO order_detail (totalCost, orderStatus, customerID) VALUES ( , 'pending', 1);
INSERT INTO payment VALUES(    );
  COMMIT;
ELSE:
  -- Rollback the transaction if there is not enough stock
  ROLLBACK;
END IF;
```

-- Begin transaction 1: User A is updating the stock of the product

```
BEGIN TRANSACTION;
UPDATE product SET productStock = 0 WHERE product ID = 1; (Wants to buy everything in stock)
INSERT INTO order_detail (totalCost, orderStatus, customerID) VALUES ( , 'pending', 2);
INSERT INTO payment VALUES(    );
COMMIT;
```

In this example, User B checks the current stock of the product and attempts to place an order. However, if the stock of the product is zero, the transaction is rolled back. Meanwhile, User A is simultaneously updating the stock of the same product to zero. As a result, User B's transaction will be rolled back since there is not enough stock to fulfill the order. Payment is assumed to be done after some seconds the order is placed.

Solution:

This situation is known as a race condition, where multiple processes compete to access a shared resource at the same time, leading to inconsistent or unpredictable behavior. To avoid this, you can use a technique called concurrency control. One common approach is to use locking mechanisms, such as semaphores or mutexes, to ensure that only one process can access the shared resource at a time. In this case, you could use a lock to prevent B from updating the stock while A is placing an order, or vice versa.

Non-conflict serializable schedule: (WR conflict) Schedule 1

T1	T2
Read product stock	
	Read product stock→Reads a non-zero value and decides to order
Write product stock	
Place Order (Commit)	
	## Read product stock
	Write product stock
	Place Order
OrderStatus read	
Payment is done	
	OrderStatus read
	Payment is done

This schedule is non-conflict serializable schedule since no swapping can be done to convert the schedule into a serial schedule since a consecutive conflicting instruction is occurring

(Write product stock→Read product Stock)→WR conflict→Dirty Read

Solution:

One can use a concurrency control mechanism such as locking. Here one can use a read lock on the data item. This allows multiple transactions to read from the data item at the same time, but only one transaction can write to the data item while the lock is held. In this scenario, the transaction that writes to the data item will acquire a write lock and prevent any other transactions from reading or writing to the data item until the lock is released.

Conflict Serializable schedule: Schedule 2

T1	T2
Read product stock	
Write product stock	
Place order	
	Read product stock (Correct product stock read)
	Write product stock (May not be able to place order if stock=0)
	Place order
OrderStatus read (Pending)	
Payment is done	
	OrderStatus read (Pending)
	Payment is done

This schedule is conflict serializable since it can be converted into a serial schedule by swapping all the consecutive non-conflicting pair of instructions between T1 and T2 into:

T1	T2
Read product stock	
Write product stock	
Place order	
OrderStatus read (Pending)	
Payment is done	

Read product stock (Correct product stock read)

Write product stock (May not be able to place order if stock=0)

Place order

OrderStatus read (Pending)

Payment is done

R(A)→reading product stock

R(B) →reading the order status

W(A)→writing product stock→placing order

W(B)→Payment is done

Part 2:

Conflicting transaction occurs when a customer puts something to buy in the cart and the admin/seller removes it from the product list is that the customer may not be able to complete their purchase if the item is no longer available.

```
-- When a customer adds a product to their cart, check if it's still available
INSERT INTO cart_item (customerID, productID, quantity)
VALUES (10, 1, 1);
SELECT productID FROM product WHERE id = 1;
IF record fetched THEN:
    --Continue
 ELSE:
 -- If the product is no longer available, remove it from the cart
 DELETE FROM cart_item WHERE customer id = 10 AND product id = 1;
END IF;
```

Solution:

Use real-time updates: You can use real-time updates to check and update the product availability in real-time. This can be achieved by implementing a mechanism that allows the customer's cart to refresh automatically when there is a change in the product availability. This ensures that the customer is aware of the availability of the product before they attempt to complete their purchase.

Non-conflicting Transactions:

1)Add a New Seller and Assign Products to Them

```
START TRANSACTION;

   -- Step 2: Insert the new seller details
   INSERT INTO seller (first_name, last_name, phoneNumber) VALUES ('Lakshya', 'Agrawal', '123-456-7890');

   -- Step 3: Get the seller ID
   SELECT LAST_INSERT_ID() INTO @sellerID;

   -- Step 4: Assign products to the new seller
   INSERT INTO sells (sellerID, productID) VALUES (@sellerID,1), (@sellerID,2), (@sellerID,3);

-- Step 5: Commit the transaction
COMMIT;
```

2)Buying cart

```
START TRANSACTION;
TRY
        INSERT INTO order_detail (customerID,deliveryID,orderStatus ,totalCost)
        VALUES (13, 5, 'dispatching',
                        (SELECT total_Cost
```

```
                    FROM cart
                    WHERE cart.customerID =13 )
                );

        INSERT INTO payment (modePayment,dateTransaction,orderID)
        VALUES ('UPI', CURDATE(), last_insert_id());

        UPDATE product
        JOIN cart_item ON product.productID = cart_item.productID
        SET product.productStock = product.productStock - cart_item.quantity
        WHERE cart_item.customerID = 13;

        DELETE FROM cart_item WHERE customerID = 13;

        UPDATE cart
        SET Total_Cost =0;
COMMIT;

CATCH
        ROLLBACK;
END TRY;
```

3)Adding a new product in the database

```
START TRANSACTION;

   INSERT INTO product (productName, productPrice, productStock)
   VALUES ('Watermelon',60,35);

   SET @productID = LAST_INSERT_ID();

   INSERT INTO belongs (productID, categoryID)
```

```
    VALUES (@productID, 14);


COMMIT;
```