17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured.

17.15 Consider the following two transactions:

T13:

```
read(A);
read(B);
if A = 0 then B := B + 1;
write(B).
```

T14:

```
read(B);
read(A);
if B = 0 then A := A + 1;
write(A).
```

Let the consistency requirement be A = 0 ∨ B = 0, with A = B = 0 as the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

Case 1: $T13 \rightarrow T14$

1. **Initial state:** $A = 0, B = 0.$
2. $T13$:
   - Reads $A = 0$, then reads $B = 0.$
   - Since $A = 0, B := B + 1 = 1.$
   - Writes $B = 1.$
   - **State after** $T13$: $A = 0, B = 1.$
3. $T14$:
   - Reads $B = 1$, then reads $A = 0.$
   - Since $B \neq 0$, no changes to A.
   - Writes $A = 0.$
   - **State after** $T14$: $A = 0, B = 1.$
4. **Consistency Check:** $A = 0 \lor B = 0$ is satisfied ($A = 0$).

Case 2: $T14 \rightarrow T13$

1. **Initial state:** $A = 0, B = 0.$
2. $T14$:
   - Reads $B = 0$, then reads $A = 0.$
   - Since $B = 0, A := A + 1 = 1.$
   - Writes $A = 1.$
   - **State after** $T14$: $A = 1, B = 0.$
3. $T13$:
   - Reads $A = 1$, then reads $B = 0.$
   - Since $A \neq 0$, no changes to B.
   - Writes $B = 0.$
   - **State after** $T13$: $A = 1, B = 0.$
4. **Consistency Check:** $A = 0 \lor B = 0$ is satisfied ($B = 0$).

Conclusion

In both serial executions, the consistency requirement A=0∨B=0 is preserved. This shows that every serial execution involving these transactions maintains the consistency of the database.

b. Show a concurrent execution of T13 and T14 that produces a nonserializable schedule.

Let us consider the operations in $T13$ and $T14$:

- $T13$: `read(A), read(B), if A = 0 then B := B + 1, write(B)`
- $T14$: `read(B), read(A), if B = 0 then A := A + 1, write(A)`

Concurrent Schedule

We interleave the operations of $T13$ and $T14$ as follows:

1. $T13$: `read(A)` → Reads $A = 0$.
2. $T14$: `read(B)` → Reads $B = 0$.
3. $T13$: `read(B)` → Reads $B = 0$.
4. $T14$: `read(A)` → Reads $A = 0$.
5. $T13$: Since $A = 0$, sets $B := B + 1 = 1$.
6. $T14$: Since $B = 0$, sets $A := A + 1 = 1$.
7. $T13$: `write(B)` → Writes $B = 1$.
8. $T14$: `write(A)` → Writes $A = 1$.

Resulting State

- After the schedule completes: $A = 1, B = 1$.

Consistency Check

The consistency requirement is $A = 0 \lor B = 0$. However, the final state $A = 1, B = 1$ **violates the consistency condition**, as neither $A = 0$ nor $B = 0$ holds.

Nonserializability

This schedule is **nonserializable** because:

- In both possible serial schedules ($T13 \rightarrow T14$ or $T14 \rightarrow T13$), at least one of A or B would remain 0, ensuring the consistency condition.
- The interleaving above introduces a state ($A = 1, B = 1$) that cannot be obtained from any serial schedule.

c. Is there a concurrent execution of T13 and T14 that produces a serializable schedule?

For part **(c)**, the answer is **no**, there is no concurrent execution of T13 and T14 that produces a **serializable schedule,** There is no concurrent execution of T13 andT14 that pr**serializable schedule**.

17.16 Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.

Transactions

We define two transactions, $T1$ and $T2$, as follows:

- $T1$:
  `read(X)` → `write(X)` → `commit`.
- $T2$:
  `read(X)` → `write(X)` → `commit`.

Serializable Schedule

A **schedule** interleaving the operations of $T1$ and $T2$ is:

1. $T1$: `read(X)`
2. $T2$: `read(X)`
3. $T2$: `write(X)`
4. $T2$: `commit`
5. $T1$: `write(X)`
6. $T1$: `commit`.

Serialization Order

The **serialization order** of this schedule is $T2 \rightarrow T1$:

- $T2$'s `write(X)` logically occurs before $T1$'s `write(X)`.
- The final state of X is determined by $T1$, which overwrites $T2$'s changes.

Commit Order

The **commit order** is $T1 \rightarrow T2$:

- $T1$ commits after $T2$ finishes its `write(X)` but before $T2$ commits.

Why is this Serializable?

The schedule is serializable because:

1. The operations can be reordered into a valid serial execution ($T2 \rightarrow T1$) without affecting the final database state.
2. There is no cyclic dependency in the precedence graph of the transactions.