Atomic Transactions

Lecture-17

Introduction

- In a distributed system like online banking, many different actions (or processes) might need to work together at the same time. These processes should work smoothly without causing errors, even if something unexpected happens (like a power outage or a broken internet connection).
- An atomic transaction is like a safety mechanism that ensures all parts of a process are completed successfully, or none of them are applied at all. Think of it like a "package deal" — either everything happens, or nothing does.
- **Example:** Transferring Money Between Accounts
- Imagine you're moving money between two bank accounts:
- **Withdraw(amount, Account-1):** You take money from the first account.
- **Deposit(amount, Account-2):** You add that money to the second account.
- Now, if something goes wrong between these two steps — let's say your phone disconnects after the first step — there’s a problem. Money has been taken from the first account, but it hasn't been added to the second one. This leaves things in a messy, incomplete state.

- **How Atomic Transactions Solve This?**
- An atomic transaction would make sure that either:
- Both steps (withdraw and deposit) happen together, or
- Neither step happens at all (if something goes wrong in between).
- This means if your connection breaks, the system will automatically undo the withdrawal, so the money stays safe. It's like the transaction never started.

Cont..

- **The Transaction Model**:
- Stable Storage:
- Storage comes in three categories.
- First we have ordinary RAM memory, which is wiped out when the power fails or a machine crushes.
- Next we have disc storage, which survives CPU failures but which can be lost in disc head crashes.
- Finally, we have **stable storage**, which is designed to survive anything except major calamities such as floods and earthquakes.
- Stable storage can be implemented with a pair of ordinary discs as shown by Fig. 3-15(a) in the next slide.

Cont..

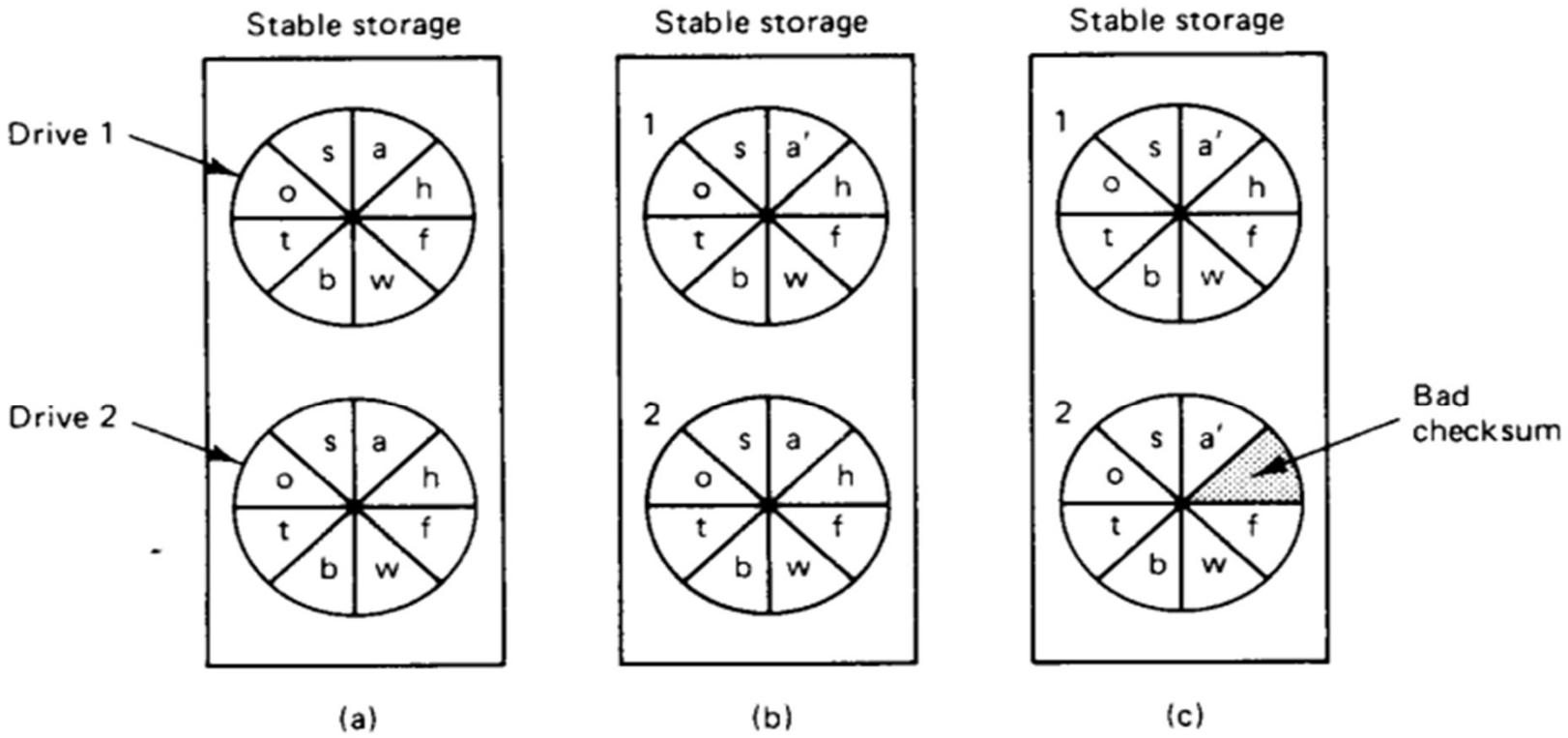

Fig. 3-15. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Cont..

- Each block on drive 2 is an exact copy of the corresponding block on drive 1.
- When a block is updated, first the block on drive 1 is updated and verified, then the same block on drive 2 is done.
- Suppose that the system crashes after drive 1 is updated, but before drive 2 is updated, as shown in figure 3–15(b).
- Upon recovery, this can be compared block for block.
- Whenever 2 corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2.
- When the recovery process is complete, both drive 1 and 2 will be identical.
- Another potential problem is the spontaneous decay of a block.
- Dust particles or general wear and tear can give a previously valid block a sudden checksum error, without cause or warning, as shown in figure 3–15(c).
- When such an error is detected, the bad block can be regenerated from the corresponding block on the other drive.

Cont..

- **Transaction Primitives**:
- Programming using transactions requires special primitives that must either be supplied by the operating system or by the language run-time system.

- **Examples are**:

1. BEGIN_TRANSACTION: Mark the start of a transaction.
2. END_TRANSACTION: Terminate the transaction and try to commit.
3. ABORT_TRANSACTION: Kill the transaction; restore the old values.
4. READ: Read data from a file (or other object).
5. WRITE: Write data to a file (or other object).

Cont..

- **Properties of transactions**:
- **Transactions have four essential properties. Transactions are**:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

- T
- **Atomicity**:
- By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
- — **Abort** : If a transaction aborts, changes made to the database are not visible.
- — **Commit** : If a transaction commits, changes made are visible.
- Atomicity is also known as the 'All or nothing rule'.

Cont..

- **Consider the following transaction T consisting of T1 and T2 : Transfer of 100 from account X to account Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

- If the transaction fails after completion of T1 but before completion of T2 .(say, after write(X) but before write(Y)), then the amount has been deducted from X but not added to Y.
- This results in an **inconsistent** database state.
- Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Cont..

- **Consistency**:
- This means that **integrity constraints** must be maintained so that the database is consistent before and after the transaction.
- It refers to the correctness of a database. Referring to the example above,
- The total amount before and after the transaction must be maintained.
- Total before T occurs = 500 + 200 = 700 .
- Total after T occurs = 400 + 300 = 700 .
- Therefore, the database is consistent . Inconsistency occurs in case T1 completes but T2 fails. As a result, T is incomplete.
- **(Integrity constraints** are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.)

Cont..

- **Isolated**:
- The third property says that transactions are **isolated** or **serializable**.
- What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as the all transactions ran sequentially in some order.
- In figure 3–17(a)–(c), we have three transactions that are executed simultaneously by 3 separate processes.
- If they were to be run sequentially the final value of x would be 1, 2, or 3, depending which one ran last.
- In figure 3– 17(d), we see various **orders**, called **schedules**, in which they might be interleaved.
- Schedule 1 is actually serialized. In other words, the transactions run strictly sequentially, so it meets the serializability condition by definition.

Cont..

```
BEGIN_TRANSACTION
    x = 0;
    x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
    x = 0;
    x = x + 3;
END_TRANSACTION
```

(c)

Time →

Schedule 1	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = 0;	x = x + 3;	Legal
Schedule 2	x = 0;	x = 0;	x = x + 1;	x = x + 2;	x = 0;	x = x + 3;	Legal
Schedule 3	x = 0;	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = x + 3;	Illegal

(d)

Fig. 3-17. (a)–(c) Three transactions. (d) Possible schedules.

Cont..

- **Schedule 1** is actually serialized. In other words, transactions run strictly sequentially, so it meets the serializability condition by definition.
- **Schedule 2** is not serialized, but it is still legal, because it results in a value for x that could have been achieved by running the transactions strictly sequentially.
- **Schedule 3** is illegal since it sets the value of x to 5, something that no sequential order ot the transactions could produce.

- **Durable:**
- The fourth property says that the transactions are durable.
- It refers to the fact that once a transaction commits, no matter, what happens, the transaction goes forward and the results become permanent.
- No failure after the commit can undo the results and cause them to be lost.