CS 439
Principles of Computer Systems
17: Transactions

Motivation

Two bank accounts A and B with balances A=50, B=50 USD

- Transfer 100 USD from A to B
- Get the sum of A and B

```plaintext
A -= 100;
// A = -50

B += 100;
// B = 150

Total balance: 100
Total balance: 0
```

Transactions

- $O = \langle x_1, x_2, ..., x_n \rangle$ a set of objects
- $OP = \{R, W\}$ the set of operations

- A transaction $T = \langle A_1, A_2, ..., A_n \rangle$ is a sequence of actions where $A_m$ is either $R(x_m)$ or $W(x_m)$.

ACID

- Atomicity
  - Updates are all or nothing
  - If something fails in the middle, roll back

- Consistency
  - System transitions from valid state to valid state
  - No partial effects

- Isolation
  - Transactions appear to operate on their own and do not interfere with other transactions
  - Effects are not visible to other transactions before committed

- Durability
  - Effect of committed transaction is never lost
Transactions

- How can we implement transactions?
  - Let's start with consistency

- Reads?

- Writing in place?
  - Memory?
  - Disk?
  - Distributed System?

Write-Ahead Logging

- First write transaction persistently into a log
- Then perform the modification
- Then the commit is logged

- Undo Logging
  - Keep enough information to roll back to the previous state
  - Old values
- Redo Logging
  - Keep enough information to redo the transactions
  - New values

- Databases typically do both
- File systems typically pick one

How much of ACID does Logging give us?

- Atomicity
  - Updates are all or nothing
  - If something fails in the middle, roll back

- Consistency
  - System transitions from valid state to valid state
  - No partial effects

- Isolation
  - Transactions appear to operate on their own and do not interfere with other transactions
  - Effects are not visible to other transactions before committed

- Durability
  - Effect of committed transaction is never lost

Isolation

- There is a temporal angle to transactions
  - If two transactions do not overlap, they do not interfere

- We talked about locking
  - Brute-force: Lock all objects
  - Limits concurrency
  - Fine granular locking: only objects involved

- We talked about deadlock

- In scalable systems isolation means concurrency control
We want parallel execution wherever possible
How can we decide where it is safe?
Schedule

A schedule is a triple \((T,A,<)\) with
- \(T\) being a transaction
- \(A\) the set of actions
- \(<\) a partial ordering on the set of actions \(A\) for each pair of actions \((a_m, a_m) \in A \times A\)

What do we want to achieve?
- Running two (or more) transactions in parallel
- Getting a correct result
  - Correct = equivalent to a serial execution

Transactions
- \(T_1: R_1(x) W_1(x) R_1(y) W_1(y)\)
- \(T_2: R_2(x) R_2(y)\)

Schedule
- \(R_1(x) W_1(x) R_2(x) R_2(y) R_1(y) W_1(y)\)

Conflict Pairs
- \(< W_1(x), R_2(x) >\)
- \(< R_2(y), W_1(y) >\)

Conflicts
- Two operations conflict if
  - They belong to different transactions,
  - touch the same data object,
  - and at least one of them is a write
Conflict-Preserving Serializability

- **Equivalency of schedules**
  - Two schedules are (conflict preserving) equivalent if they have the same transaction set action set and if the conflict pairs are ordered in the same way in both schedules.

- **Correctness of a schedule**
  - A schedule is correct, if it is equivalent to a serial schedule, such a schedule is said to be (conflict preserving) serializable (CPSR).

Dependency Graph

- **Nodes are transactions** \((T_i, T_j, \ldots)\)
- **Edge from** \(T_i\) **to** \(T_j\) **iff**
  - An operation in \(T_i\) conflicts with an operation in \(T_j\)
  - \(T_i\) appears before \(T_j\) in the schedule.

- **Example**
  - Schedule \(R_1(x), W_1(x), R_2(y), R_3(z), R_2(x), W_2(y), R_4(z), W_4(z), R_4(z)\)
  - Conflict pairs: \(R_1(x), R_2(x)\), \(R_2(x), W_2(y)\), \(R_2(x), R_4(z)\)
  - **Dependency Graph**

Two-Phase Locking

- **Rule 1**: an object has to be locked before it can participate in a transaction.
  - Growing phase
- **Rule 2**: as soon as a transaction unlocks one object it cannot lock any further objects.
  - Shrinking phase
- **2PL ensures serializability**

Durability and Atomicity

- **Are we okay on a single machine?**
  - Locking
  - Logging
- **What happens in a distributed system?**
  - Consider two identical replicas of the data objects
  - **Problem?**
- **Atomicity**: Either both replicas must commit, or both must abort the transaction
  - **We need consensus...**
Two Generals Problem
- Barbarians win if the two Roman generals cannot coordinate their attack.
- Communication through a messenger

Some Failure Scenarios
- Request message can be lost
- Reply message can be lost
- Recipient can crash
- Sender can crash

Ordering
- FIFO
- Causal Ordering

Two-Phase Commit (2PC)
- Voting Phase
  - Coordinator sends VOTE_REQ to all workers
  - Workers send VOTE_COMMIT or VOTE_ABORT back
Two-Phase Commit (2PC)

- Completion Phase
  - If all votes were VOTE_COMMIT, send GLOBAL_COMMIT to all workers, else send GLOBAL_ABORT
  - Workers commit or abort the transaction

Example: DHT

- Consistent hash function
- Nodes have finger tables
- Content is replicated
- If a node fails, another node can take over
- Consistency?
- Availability?
- Partition Resilience?

CAP Theorem

- Brewer’s conjecture:
  - A distributed computer system can only have two out of
    - Consistency
    - Availability
    - Partition Resilience

- Accepted by many, disputed by some
  - Formal proof exists