Atomic Transactions

- The Transaction Model / Primitives
- Implementation
- *Reading:*
  - *Coulouris: Distributed Systems, Addison Wesley, Chapters 14*
  - *A.S. Tanenbaum: Distributed Operating Systems, Prentice Hall, 1995, Chapter 3.4*

Atomic Transactions

- Example: online bank transaction:
  `withdraw(amount, account1)`
  `deposit(amount, account2)`
- What if network fails before deposit?
- Solution: Group operations in an atomic transaction.
- Volatile storage *vs.* stable storage.
- Primitives:
  - `BEGIN_TRANSACTION`
  - `END_TRANSACTION`
  - `ABORT_TRANSACTION`
  - `READ`
  - `WRITE`

ACID Properties

A	atomic:	transactions happen indivisibly
C	consistent:	no violation of system invariants
I	isolated:	no interference between concurrent transactions
D	durable:	after transaction commits, changes are permanent

Serializability

Schedule is serial if the steps of each transaction occur consecutively.
Schedule is serializable if its effect is “equivalent” to some serial schedule.

```
BEGIN TRANSACTION      BEGIN TRANSACTION      BEGIN TRANSACTION
  x := 0;                x := 0;                x := 0;
  x := x + 1;            x := x + 2;            x := x + 3;
END TRANSACTION        END TRANSACTION        END TRANSACTION
```

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

Testing for Serializability: Serialization Graphs

- Input: Schedule S for set of transactions $T_1, T_2, \ldots, T_k$.
- Output: Determination whether S is serializable.
- Method:
  - Create *serialization graph G*:
    - Nodes: correspond to transactions
    - Arcs: G has an arc from T_i to T_j if there is a T_i:*UNLOCK*(A_m) operation followed by a T_j:*LOCK*(A_m) operation in the schedule.
  - Perform topological sorting of the graph.
    - If graph has cycles, then S is not serializable.
    - If graph has no cycles, then topological order is a serial order for transactions.

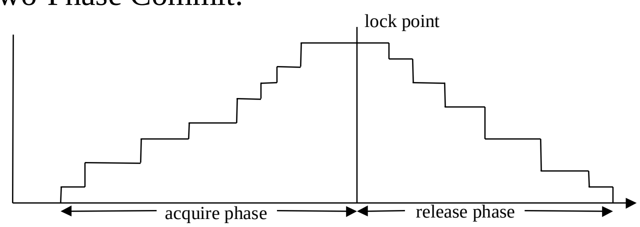
- Theorem: This algorithm correctly determines if a schedule is serializable.

Implementation

- How to maintain information from not-yet committed transactions: "Prepare for aborts"
  - private workspace
  - writeahead log / intention lists with rollback
- Commit protocol
  - 2-phase commit protocol.
- Concurrency control:
  - pessimistic -> lock-based: 2-phase locking
  - optimistic -> timestamp-based with rollback

Serializability through Two-Phase Locking

- read locks *vs.* write locks
- lock granularity
- arbitrary locking:
  - non-serializable schedules
  - deadlocks!
- Two-Phase Commit:



  - modify data items only after lock point
  - all schedules are serializable

Two-Phase Locking (cont)

- Theorem: If *S* is any schedule of two-phase transactions, then *S* is serializable.

- Proof:

  Suppose not. Then the serialization graph *G* for *S* has a cycle,

  $$T_{i1} \; -> \; T_{i2} \; -> \; \ldots \; -> \; T_{ip} \; -> \; T_{i1}$$

  Therefore, a lock by T_{i1} follows an unlock by T_{i1}, contradicting the assumption that T_{i1} is two-phase.

Transactions that Read "Dirty" Data

	T_1	T_2
(1)	LOCK A	
(2)	READ A	
(3)	A:=A-1	
(4)	WRITE A	
(5)	LOCK B	
(6)	UNLOCK A	
(7)		LOCK A
(8)		READ A
(9)		A:=A*2
(10)	READ B	
(11)		WRITE A
(12)		COMMIT
(13)		UNLOCK A
(14)	B:=B/A	

Assume that T_1 fails after (13).

1. T_1 still holds lock on B.
2. Value read by T_2 at step (8) is wrong.

T_2 must be rolled back and restarted.

3. Some transaction T_3 may have read value of A between steps (13) and (14)

Strict Two-Phase Locking

- Strict two-phase locking:
  - A transaction cannot write into the database until it has reached its commit point.
  - A transaction cannot release any locks until it has finished writing into the database; therefore locks are not released until after the commit point.
- pros:
  - transaction read only values of committed transactions
  - no cascaded aborts
- cons:
  - limited concurrency
  - deadlocks
- Models/protocols can be extended for READ/WRITE locks.

Optimistic Concurrency Control

"Forgiveness is easier to get than permission"

- Basic idea:
  - Process transaction without attention to serializability.
  - Keep track of accessed data items.
  - At commit point, check for conflicts with other transactions.
  - Abort if conflicts occurred.
- Problem:
  - would have to keep track of conflict graph and only allow additional access to take place if it does not cause a cycle in the graph.

Timestamp-based Optimistic Concurrency Control

- Data items are tagged with read-time and write-time.
- 1. Transaction cannot read value of item if that value has not been written until after the transaction executed.

  Transaction with T.S. t_1 cannot read item with write-time t_2 if $t_2 > t_1$.
  (abort and try with new timestamp)
- 2. Transaction cannot write item if item has value read at later time.

  Transaction with T.S. t_1 cannot write item with read-time t_2 if $t_2 > t_1$.
  (abort and try with new timestamp)
- Other possible conflicts:
  - Two transactions can read the same item at different times.
  - What about transaction with T.S. t_1 that wants to write to item with write-time t_2 and $t_2 > t_1$?

Timestamp-Based Conc. Control (cont)

Rules for preserving serial order using timestamps:

a) Perform the operation X if X=READ and $t >= t_w$ or if X=WRITE, $t >= t_r$, and $t >= t_w$.
   - X=READ: set read-time to t if $t > t_r$.
   - X=WRITE: set write-time to t if $t > t_w$.

b) Do nothing if X=WRITE and $t_r <= t < t_w$.

c) Abort transaction if X=READ and $t < t_w$ or X=WRITE and $t < t_r$.

Timestamp-based Optimistic Concurrency Control

- Accesses to data items are tagged with timestamp (e.g. Lamport)
- Examples: