

## CSE 486/586 Distributed Systems Concurrency Control --- 1

Steve Ko  
Computer Sciences and Engineering  
University at Buffalo

CSE 486/586

### Banking Example (Once Again)

- Banking transaction for a customer (e.g., at ATM or browser)
  - Transfer \$100 from saving to checking account
  - Transfer \$200 from money-market to checking account
  - Withdraw \$400 from checking account
- Transaction
  1. savings.deduct(100)
  2. checking.add(100)
  3. mnymkt.deduct(200)
  4. checking.add(200)
  5. checking.deduct(400)
  6. dispense(400)

CSE 486/586

2

### Transaction

- Abstraction for **grouping multiple operations into one**
- A transaction is **indivisible (atomic)** from the point of view of other transactions
  - No access to intermediate results/states
  - Free from interference by other operations
- Primitives
  - **begin()**: begins a transaction
  - **commit()**: tries completing the transaction
  - **abort()**: aborts the transaction & rolls back to the previous state (as if nothing happened)
- Why abort()?
  - A failure happens in the middle of execution.
  - A transaction is part of a bigger transaction (i.e., it's a sub-transaction), and the bigger transaction needs abort.
  - Etc.

CSE 486/586

3

### Properties of Transactions: ACID

- **A**tomicity: All or nothing
- **C**onsistency: if the server starts in a consistent state, the transaction ends with the server in a consistent state.
- **I**solation: Each transaction must be performed without interference from other transactions, i.e., the non-final effects of a transaction must not be visible to other transactions.
- **D**urability: After a transaction has completed successfully, all its effects are saved in permanent storage.

CSE 486/586

4

### This Week

- Question: How to support transactions?
  - Multiple transactions share data.
- What would be your first strategy (hint: locks)?
  - One transaction at a time with one big lock, i.e., complete serialization
- Two issues
  - Performance
  - Abort

CSE 486/586

5

### Performance?

#### • Process 1

```
lock(mutex);
savings.deduct(100);
checking.add(100);
mnymkt.deduct(200);
checking.add(200);
checking.deduct(400);
dispense(400);
unlock(mutex);
```

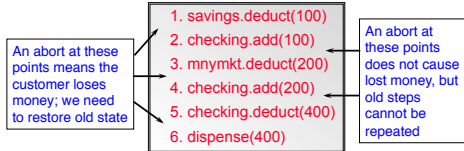
#### • Process 2

```
lock(mutex);
savings.deduct(200);
checking.add(200);
unlock(mutex);
```

CSE 486/586

6

## Abort?



CSE 486/586

7

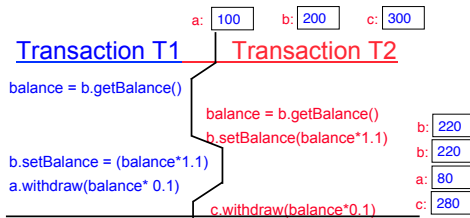
## This Week

- Question: How to support transactions?
  - Multiple transactions share data.
- What would be your first strategy (hint: locks)?
  - Complete serialization
  - One transaction at a time with one big lock
  - Two issues: Performance and abort
- First, let's see how we can improve performance.
  - By executing multiple transactions concurrently

CSE 486/586

8

## What Can Go Wrong?



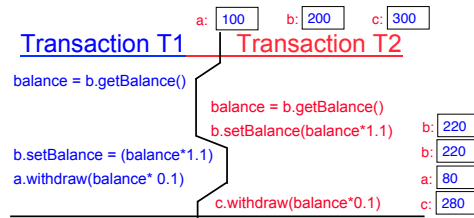
- T1/T2's update on the shared object, "b", is lost

CSE 486/586

9

## Lost Update Problem

- One transaction causes loss of info. for another: consider three account objects

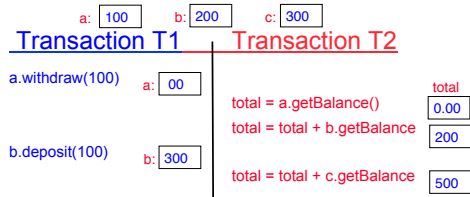


- T1/T2's update on the shared object, "b", is lost

CSE 486/586

10

## What Can Go Wrong?



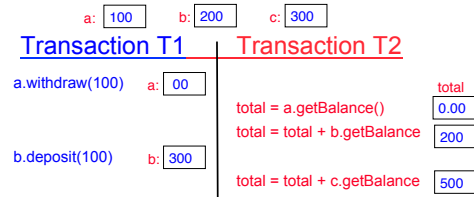
- T1's partial result is used by T2, giving the wrong result

CSE 486/586

11

## Inconsistent Retrieval Problem

- Partial, incomplete results of one transaction are retrieved by another transaction.



- T1's partial result is used by T2, giving the wrong result

CSE 486/586

12

## What This Means

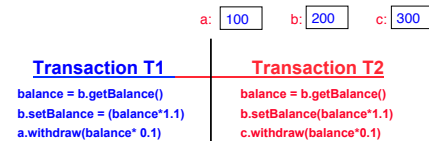
- Question: How to support transactions (with locks)?
  - Multiple transactions share data.
- Complete serialization is correct, but performance and abort are two issues.
- Executing transactions concurrently for performance
  - Problem: Not all current executions produce a correct outcome

CSE 486/586

13

## What is “Correct”?

- How would you define correctness?
- For example, two independent transactions made by me and my wife on our three accounts.
- What do we care about at the end of the day?
  - Correct final balance for each account

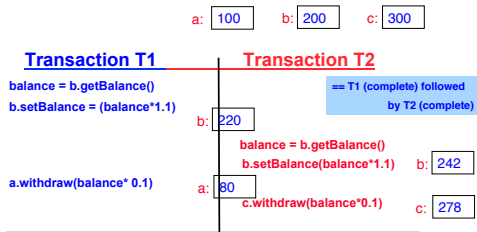


CSE 486/586

14

## Concurrency Control: Providing “Correct” Interleaving

- An interleaving of the operations of 2 or more transactions is said to be *serially equivalent* if the combined effect is the same as if these transactions had been performed *sequentially in some order*.



CSE 486/586

15

## CSE 486/586 Administrivia

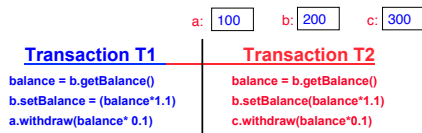
- Grading will be done this week.
- PA3 will be out this week.

CSE 486/586

16

## Providing Serial Equivalence

- What operations are we considering?
  - Read/write
- What operations matter for correctness?
  - When write is involved



CSE 486/586

17

## Conflicting Operations

- Two operations are said to be in conflict, if their *combined effect* depends on the *order* they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, not on different variables.

Operations of different transactions	Conflict	Reason
read    read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed
read    write	Yes	Because the effect of a read and a write operation depends on the order of their execution
write    write	Yes	Because the effect of a pair of write operations depends on the order of their execution

CSE 486/586

18

## Conditions for Correct Interleaving

- What should we need to do to guarantee serial equivalence with conflicting operations?
- Case 1
  - T1.1 -> T1.2 -> T2.1 -> T2.2 -> T1.3 -> T2.3
- Case 2
  - T1.1 -> T2.1 -> T2.2 -> T1.2 -> T1.3 -> T2.3
- Which one's correct and why?

Transaction T1	Transaction T2
1. balance = b.getBalance()	1. balance = b.getBalance()
2. b.setBalance = (balance*1.1)	2. b.setBalance(balance*1.1)
3. a.withdraw(balance*0.1)	3. c.withdraw(balance*0.1)

CSE 486/586

19

## Conflicting Operations

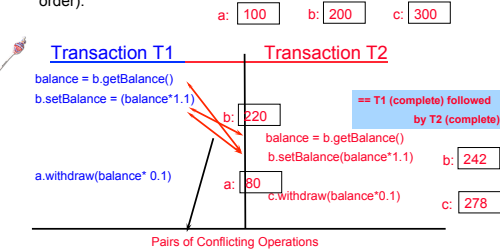
- **Insight for serial equivalence**
  - Outcomes of write operations in one transaction to all shared objects should be *either consistently visible to the other transaction or the other way round.*
- The effect of an operation refers to
  - The value of an object set by a write operation
  - The result returned by a read operation.
- Two **transactions** are **serially equivalent** if and only if **all pairs of conflicting operations** (pair containing one operation from each transaction) **are executed in the same order** (transaction order) **for all objects (data) they both access.**

CSE 486/586

20

## Example of Conflicting Operations

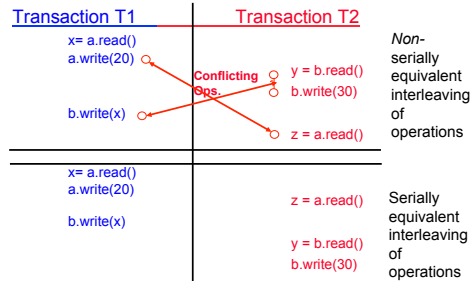
- An interleaving of the operations of 2 or more transactions is said to be **serially equivalent** if the combined effect is the same as if these transactions had been performed sequentially (in some order).



CSE 486/586

21

## Another Example



CSE 486/586

22

## Inconsistent Retrievals Problem

Transaction V:	Transaction W:
a.withdraw(100)	aBranch.branchTotal()
b.deposit(100)	
a.withdraw(100);	total = a.getBalance()    \$100
\$100	total = total+b.getBalance()    \$300
b.deposit(100)	total = total+c.getBalance()
\$300	•

Both withdraw and deposit contain a write operation

CSE 486/586

23

## Serially-Equivalent Ordering

Transaction V:	Transaction W:
a.withdraw(100);	aBranch.branchTotal()
b.deposit(100)	
a.withdraw(100);	\$100
	total = a.getBalance()    \$100
b.deposit(100)	\$300
	total = total+b.getBalance()    \$400
	total = total+c.getBalance()
	•

CSE 486/586

24

## Summary

- Transactions need to provide ACID
- Serial equivalence defines correctness of executing concurrent transactions
- It is handled by ordering conflicting operations

CSE 486/586

25

## Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).

CSE 486/586

26