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

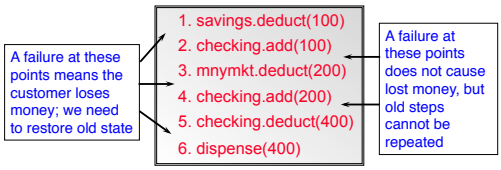
Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap: Concurrent Transactions

• Process 1	• Process 2

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

Why Not Satisfied?



Recap: Locks & Transactions

- What we discussed in mutual exclusion is one big lock.
  - Everyone else has to wait.
  - It does not necessarily deal with failures.
- Performance
  - Observation: we can interleave some operations from different processes.
- Failure
  - If a process crashes while holding a lock
- Let's go beyond simple locking!

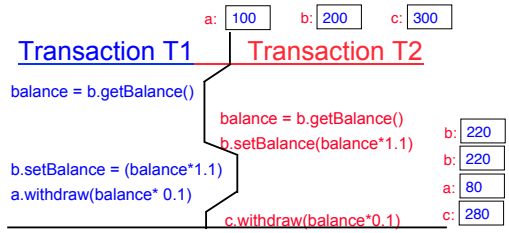
Transaction

- 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
- Implementing transactions
  - Performance: finding out what operations we can interleave
  - Failure: dealing with failures, rolling back changes if necessary

Properties of Transactions: ACID

- Atomicity: All or nothing
- Consistency: if the server starts in a consistent state, the transaction ends with the server in a consistent state.
- Isolation: 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.
- Durability: After a transaction has completed successfully, all its effects are saved in permanent storage.

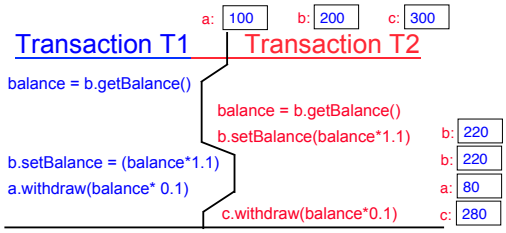
What Can Go Wrong?



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

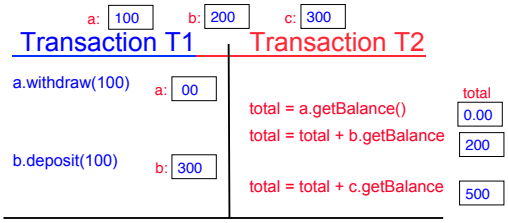
Lost Update Problem

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



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

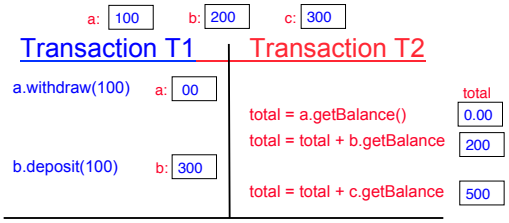
What Can Go Wrong?



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

Inconsistent Retrieval Problem

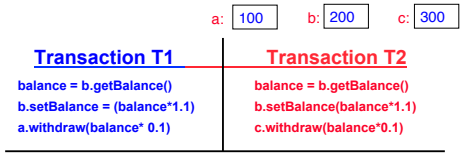
- Partial, incomplete results of one transaction are retrieved by another transaction.



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

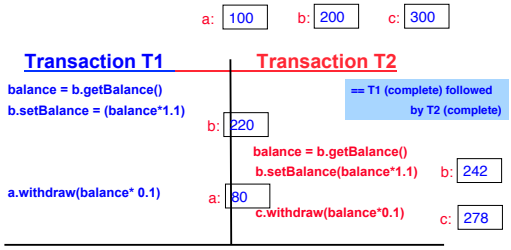
What is "Correct"?

- How would you define correctness?



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).

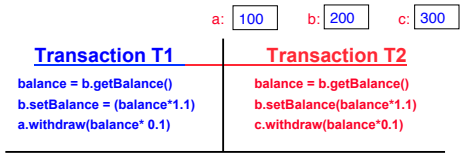


CSE 486/586 Administrivia

- Midterm: 3/6 (Wednesday) in class
  - 45 minutes
  - Everything up to leader election
  - 1-page cheat sheet is allowed.
- Tech Talk: Dave Parfitt (Basho) Tonight March 4 at 6PM in Davis 338A
- PA3 is out.
- No recitations this week
- Anonymous feedback form still available.
- Please come to me!

Providing Serial Equivalence

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



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

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?

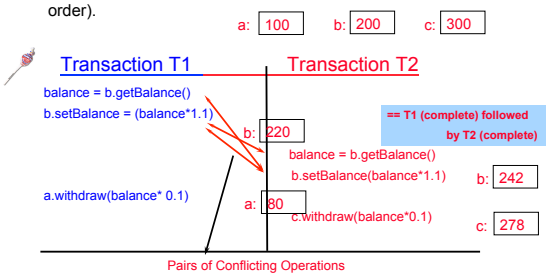
Transaction T1	Transaction T2
1. balance = b.getBalance()	1. balance = b.getBalance()
2. b.setBalance = (balance*1.1)	2. b.setBalance(balance*1.1)
3. a.withdraw(balance* 0.1)	3. c.withdraw(balance*0.1)

Conflicting Operations

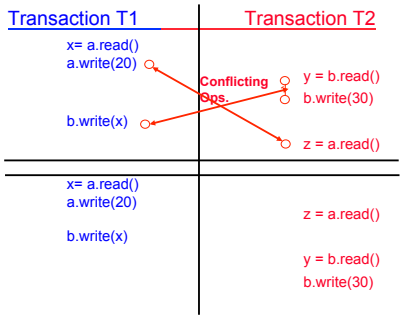
- Insight for serial equivalence
  - Outcomes of write operations in one transaction to all shared objects should be *either consistently visible to the other transaction or the other way round*.
- The effect of an operation refers to
  - The value of an object set by a write operation
  - The result 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.*

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).



Another Example



Non- serially equivalent interleaving of operations

Serially equivalent interleaving of operations

Inconsistent Retrievals Problem

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

Both withdraw and deposit contain a write operation

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()	
		...	

Summary

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

Acknowledgements

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