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)

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. (E.g., powering off the machine doesn’t mean the result is gone.)

This Week

- Question: How to support multiple transactions?
  - When multiple transactions share data.
  - Assume a single processor (one instruction at a time).
- 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

Performance?

- **Process 1**
  - lock(mutex);
  - savings.deduct(100);
  - checking.add(100);
  - mnymkt.deduct(200);
  - checking.deduct(200);
  - dispense(400);
  - unlock(mutex);

- **Process 2**
  - lock(mutex);
  - savings.deduct(200);
  - checking.add(200);
  - unlock(mutex);
Abort?

1. savings.deduct(100)
2. checking.add(100)
3. mnymkt.deduct(200)
4. checking.add(200)
5. checking.deduct(400)
6. dispense(400)

An abort at these points means the customer loses money; we need to restore old state.

An abort at these points does not cause lost money, but old steps cannot be repeated.

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.

Possibility: Interleaving Transactions for Performance

• Process 1
  savings.deduct(100);
  checking.add(100);
  mnymkt.deduct(200);
  checking.add(200);
  checking.deduct(400);
  dispense(400);

• Process 2
  savings.deduct(200);
  checking.add(200);

• P2 will not have to wait until P1 finishes.

What Can Go Wrong?

Transaction T1

a.withdraw(balance*0.1)
balance = b.getBalance()
b.setBalance(balance*1.1)
b.a.withdraw(balance*1.1)

Transaction T2

b.setBalance(balance*1.1)
b.c.withdraw(balance*0.1)

• 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

Transaction T1

balance = b.getBalance()
b.setBalance(balance*1.1)
a.withdraw(balance*0.1)

Transaction T2

balance = b.getBalance()
b.setBalance(balance*1.1)
c.withdraw(balance*0.1)

• T1/T2’s update on the shared object, “b”, is lost.

What Can Go Wrong?

Transaction T1

a.withdraw(100)

Transaction T2

b.deposit(100)
total = total + b.getBalance()
a.total = total + a.getBalance()
b.total = total + c.getBalance()

total = 500

• 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.

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: withdraw(100)</td>
<td>total = a.getBalance()</td>
</tr>
<tr>
<td></td>
<td>total = total + b.getBalance()</td>
</tr>
<tr>
<td>b: deposit(100)</td>
<td>total = total + c.getBalance()</td>
</tr>
</tbody>
</table>

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

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 interleavings produce a correct outcome

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

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance()</td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td>a.withdraw(balance*0.1)</td>
<td>c.withdraw(balance*0.1)</td>
</tr>
</tbody>
</table>

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.

Providing Serial Equivalence

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

CSE 486/586 Administrivia

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

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a read and a write operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of write operations depends on the order of their execution</td>
</tr>
</tbody>
</table>

Conditions for Correct Interleaving

- What do 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
  - 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)

  Case 2
  - T1.1 -> T2.1 -> T2.2 -> T1.2 -> T1.3 -> T2.3

Observation

- Case 1
  - T1.1 -> T1.2 -> T2.1 -> T2.2 -> T1.3 -> T2.3
  - Correct: for a shared object (b), T1 produces its final outcome, and then T2 accesses it.

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)

- Incorrect: for a shared object (b), T2 uses T1’s intermediate outcome, which T1 later modifies.

Serial Equivalence and Conflicting Operations

- 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.

Generalizing the Observation

- Insight for serial equivalence
  - Only the final outcome of a shared object from one transaction should be visible to another transaction.
  - The above should be the case for each and every shared object in the same order.
  - E.g., if T1’s final outcome on one shared object becomes visible to T2, then for each and every other shared object, T1 should produce the final outcome before T2 uses it.
  - The other way round is possible, i.e., T2 first then T1.
  - What is it when one transaction modifies a shared object and another transaction uses it?
    - Conflicting operations
Serial Equivalence Example

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).

Transaction T1
- balance = b.getBalance
- b.setBalance = (balance*1.1)
- a.withdraw(balance*0.1)
- c.withdraw(balance*0.1)

Transaction T2
- balance = b.getBalance
- b.setBalance(balance*1.1)
- a.withdraw(balance*0.1)
- c.withdraw(balance*0.1)

Pairs of Conflicting Operations

Inconsistent Retrievals Problem

Transaction V:
- a.withdraw(100)
- b.deposit(100)
- a.withdraw(100)
- b.deposit(100)

Transaction W:
- aBranch.branchTotal
- a.deposit(100)
- b.deposit(100)
- a.getBalance()
- total = a.getBalance()
- total = total + b.getBalance()
- total = total + c.getBalance

Serially-Equivalent Ordering

Transaction V:
- a.withdraw(100);
- b.deposit(100)
- a.withdraw(100);
- b.deposit(100)

Transaction W:
- aBranch.branchTotal()
- a.withdraw(100);
- b.deposit(100)
- a.getBalance()
- total = a.getBalance()
- total = total + b.getBalance()
- 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).