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)

Concurrent Transactions

- 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);
  - checking.deduct(400);
  - dispense(400);
  - unlock(mutex);

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?

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)

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

Lost Update Problem

Transaction T1
a.withdraw(100)
total = a.getBalance
b.deposit(100)
total = total + b.getBalance

Transaction T2
b.deposit(100)

• T1’s partial result is used by T2, giving the wrong result

Inconsistent Retrieval Problem

Transaction T1
a.withdraw(100)

Transaction T2
b.deposit(100)

• T1’s partial result is used by T2, giving the wrong result
What is “Correct”?  
• How would you define correctness?

Concurrent Transactions: 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

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.

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?

---

<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>
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

Transaction T1
- x = a.read()
- a.write(20)
- b.write(x)

Transaction T2
- y = b.read()
- b.write(30)
- z = a.read()

Inconsistent Retrievals Problem

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

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

Serially-Equivalent Ordering

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100); b.deposit(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>a.withdraw(100)</td>
<td>$100</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>$300</td>
</tr>
</tbody>
</table>

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).