Course 02158

Transactions

Hans Henrik Løvengreen

DTU Compute

Need for larger atomic actions

A

deposit( )

withdraw( )

balance()

B

deposit( )

withdraw( )

balance()

Transactions

- *Transaction* = atomic action spanning several concurrent objects
- Classic example: Transfer between bank accounts
- Transactions may be *interactive*
- Transaction may *fail* or be *given up* (abort).
- Often associated with persistent data objects (data bases)

Transaction Properties

Atomicity. All-or-nothing property.

Consistency. Should preserve system invariants.

Isolation. No interference (= atomicity).

Durability. Effect should remain.

Serializability

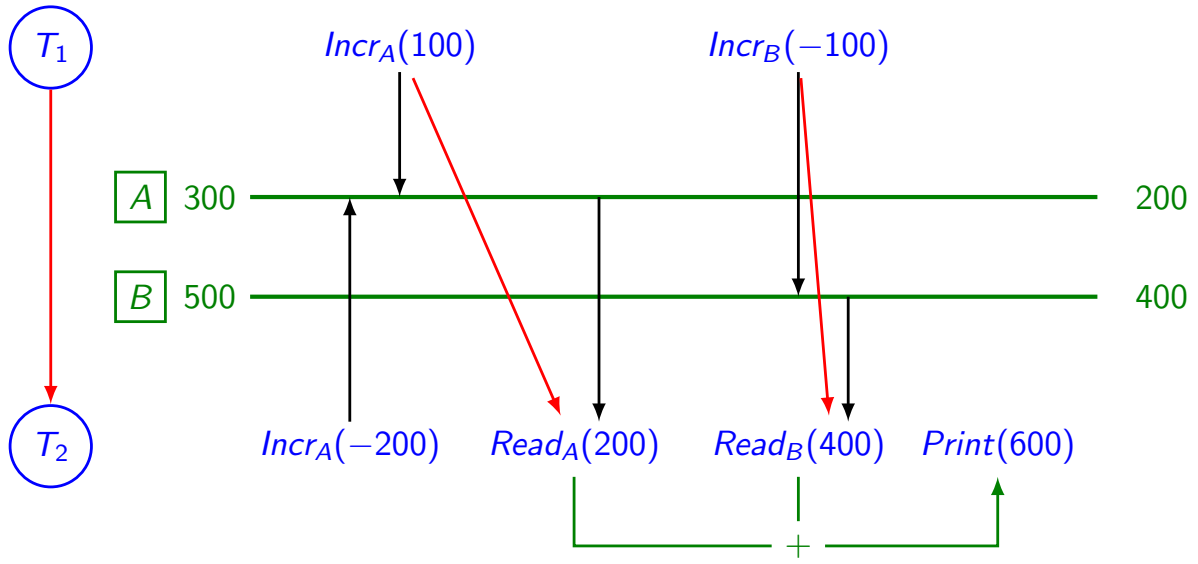
- A set of concurrently executed transactions are *serializable* if the operations on the underlying objects can be reordered such that:
  1. The sequence of operations for each transaction is preserved
  2. The transactions follow each other in some sequence
  3. The effect on the object states remains the same

Conflics

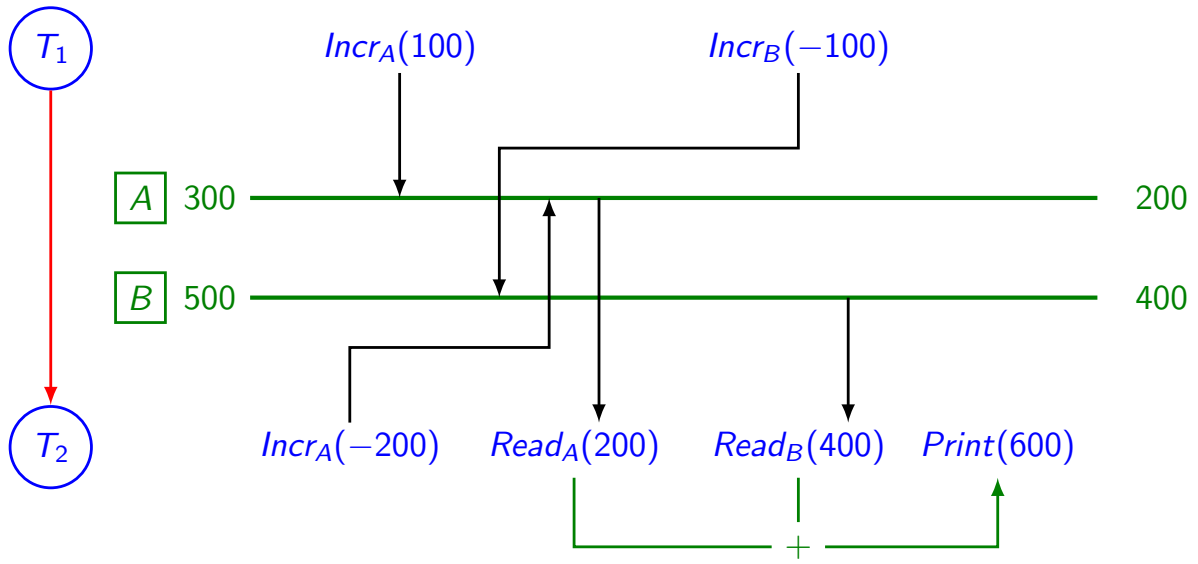
- Two operations are in *conflict* if they cannot always be swapped
- Operations on different objects do not conflict
- Single object conflicts may be characterized semantically, eg.

	Read	*Write*	*Incr*
Read		×	×
Write	×	×	×
Incr	×	×	

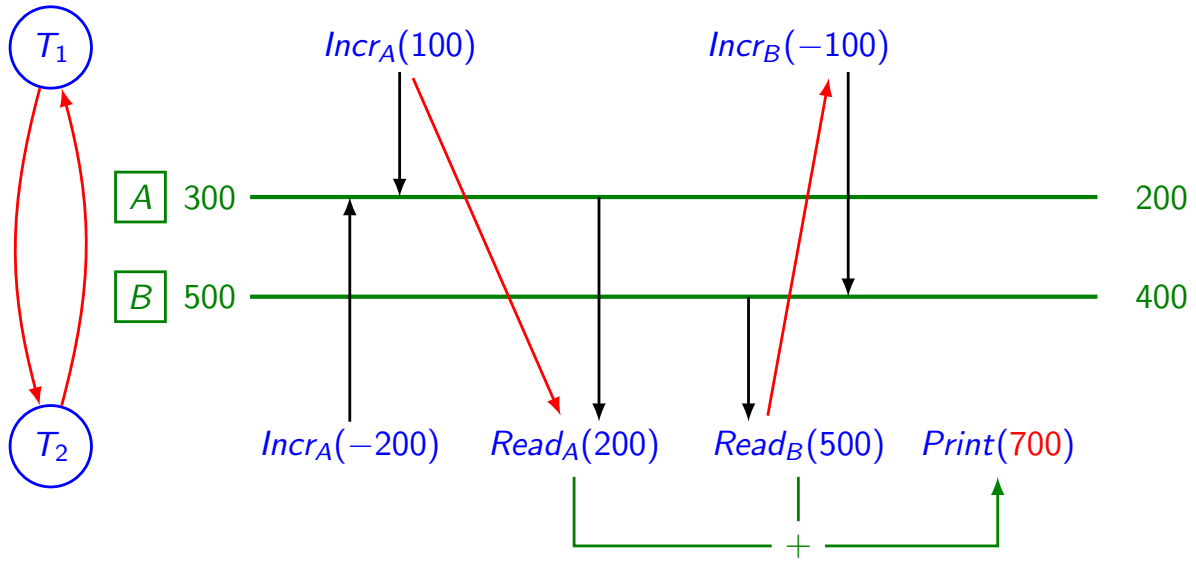
Serializability



Serializability — serialization



Serializability



Transaction Management

- $tid = StartTransaction()$

  $\vdots$

  $obj.op(\dots)_{tid}$

  $\vdots$

  $res = EndTransaction(tid)$ $\quad$ res : Commit, Abort
- If transaction may abort, operations must be *undone*

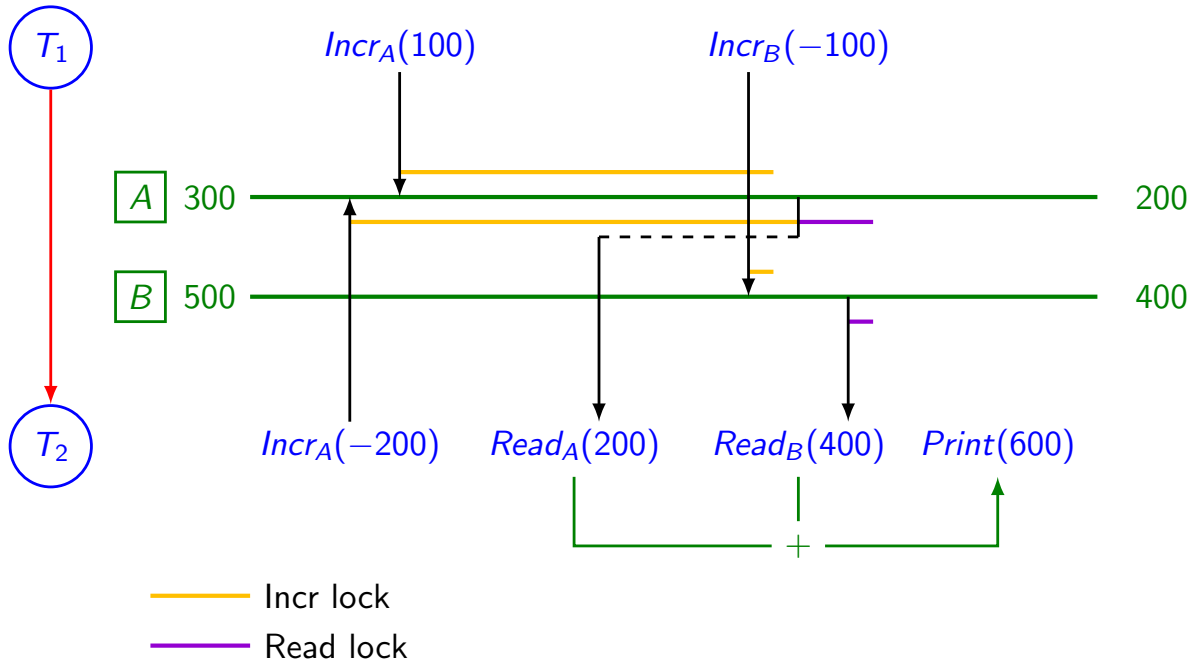
Pessimistic Solutions

- *Global lock* at transaction level
- Locking of *individual objects*, conflict locking, successive locking
- 2-phase locking: Lock until commit/abort decided, then unlock.

Optimistic Solutions

- Idea: Go ahead and validate transaction before commitment
- Many different techniques.

Serializability by Locking



MySQL Transactions

Storage Engine MyISAM

- Efficient, but only supports explicit locking at table level
- Example: `LOCK TABLE A READ, B WRITE, C WRITE`
  Use of tables `A`, `B`, and `C`
  `UNLOCK TABLES`

Storage Engine InnoDB

- Supports transactions by row-level locking
- Example:
  ```
  START TRANSACTION
    SELECT ... FROM ... WHERE ...
    :
    UPDATE ... SET ...
  COMMIT
  ```
- Optionally *weaker isolation levels* may be used

Software Transactional Memory (STM)

- Idea: To use transaction notions for in-memory program state

Principles

- Data must be stored in *transaction-aware objects* ("references")
- All use must be coordinated by a *transaction manager*
- Mostly optimistic techniques are used
- Transactions may have syntactic support:

```
atomic {
  ...
}
```

- Conditional operations may be achieved by explicit **retry**
- Transactions might be *composed*, eg. T_1 **and** T_2, T_1 **orElse** T_2
- Composing conditional operations may implement *selection*

An STM Framework: Multiverse for JVM

- Uses optimistic *multiversion concurrency control*
- Classes may be *transactionally instrumented* by annotations

Example

```
@TransactionalObject
public class Account {

    private IntRef balance = newIntRef(0);

    public int getBalance() { return balance; }

    public void addToBalance(int amount) {
        if (balance + amount < 0) throw new IllegalStateException();
        balance = balance + amount;
    }

    @TransactionalMethod
    public static void transfer (Account from, Account to, int amount) {
        from.addToBalance(-amount);
        to.addToBalance(amount);
    }
}
```

Another STM Framework: Scala STM for JVM

- Shared data must be accessed via *transactional references* (`Ref`)
- Transactions use embedded *transactional constructs*

Example

```
class ConcurrentIntList {
  private class Node(val elem: Int, prev0: Node, next0: Node) {
    val isHeader = prev0 == null
    val prev = Ref(if (isHeader) this else prev0)
    val next = Ref(if (isHeader) this else next0)
  }

  private val header = new Node(-1, null, null)
```

```
  def addLast(elem: Int) {
    atomic { implicit txn =>
      val p = header.prev()
      val newNode = new Node(elem, p, header)
      p.next() = newNode
      header.prev() = newNode
    }
  }
```

Another STM Framework: Scala STM for JVM II

Example (cont'd)

```
  def addLast(e1: Int, e2: Int, elems: Int*) {
    atomic { implicit txn =>
      addLast(e1)
      addLast(e2)
      elems foreach  addLast(_)
    }
  }
```

```
  def removeFirst(): Int = atomic { implicit txn =>
    val n = header.next()
    if (n == header)
      retry
    val nn = n.next()
    header.() = nn
    nn.prev() = header
    n.elem
  }
```

Distributed Transaction Management

- Objects handled by several distributed servers
- Roles:
  - *Client*: Defines transaction extent and contents
  - *Coordinator*: Controls the outcome of transaction
  - *Participants*: Distributed objects being operated upon
- Client creates transaction and issues operations to participants
- Overall *decision* whether to commit or abort must be made
- Standard approach: *Two-phase commit*

Transaction Summary

- The transaction notion originates from *data bases*
- Can be used on any system with shared data
- Takes over the *concurrency control* via a *transaction manager*
- Transaction management has an overhead
- May be based on *optimistic* og *pessimistic* (locking) approach