Transactions and Concurrency Control
Business and Logistics

☐ Term project proposals due
Acknowledgements

☐ Some material taken from author’s teaching slides based on Distributed Systems: Concepts and Design book

☐ Some figures taken from Distributed Systems: Concepts and Design book
Lecture Objectives

☐ To study the design of services whose long-lived objects are intended to be shared by multiple clients

☐ To establish a model of a single-process server of long-lived objects

☐ To set up the all-or-nothing and isolation properties of transactions within this model, so as to be able to study how they are maintained in the presence of concurrent clients and server failures

☐ To study the three main approaches to concurrency control, all of which maintain the isolation property of transactions in the presence of their concurrent execution at a single server
Introduction to Transactions

- Goal of transactions
  - Objects managed by a server must remain in a consistent state
    - when they are accessed by multiple transactions
    - in the presence of server crashes

- Recoverable objects
  - Can be recovered after their server crashes
  - Objects are stored in permanent storage

- Failure model
  - Transactions deal with crash failures of processes and omission failures of communication

- Designed for an asynchronous system
  - It is assumed that messages may be delayed
Operations of the *Account* Interface

- **deposit(amount)**
  - deposit amount in the account
- **withdraw(amount)**
  - withdraw amount from the account
- **getBalance() → amount**
  - return the balance of the account
- **setBalance(amount)**
  - set the balance of the account to amount

- **create(name) → account**
  - create a new account with a given name
- **lookUp(name) → account**
  - return a reference to the account with the given name
- **branchTotal() → amount**
  - return the total of all the balances at the branch
Atomic Operations at Server

- Synchronization of client operations without transactions
- When a server uses multiple threads it can perform several client operations concurrently
- If we allowed deposit and withdraw to run concurrently we could get inconsistent results
- Objects should be designed for safe concurrent access
  - In Java use synchronized methods
    ```java
    public synchronized void deposit(int amount) throws RemoteException
    ```
- Atomic operations are free from interference from concurrent operations in other threads.
- Use any available mutual exclusion mechanism on server
Client Cooperation by Synchronizing Servers

- Now consider clients share resources via a server
  - Some clients update server objects and others access them
- Servers with multiple threads require atomic objects
- In some applications, clients depend on one another to progress
  - Ex: one is a *producer* and another a *consumer*
  - Ex: one sets a lock and the other waits for it to be released
- Waiting client polls server to see if a resource is yet available
  - Potentially poor performance (why?) and unfair (why?)
- Java *wait* and *notify* methods allow threads to communicate with one another and to solve these problems
  - Ex: when a client requests a resource, the server thread waits until it is notified that the resource is available
Failure Model for Transactions

- Lampson’s failure model
  - Deals with failures of disks, servers, communication
  - Algorithms work correctly when predictable faults

- Predictable faults
  - Writes to permanent storage may fail
    - reads can detect bad blocks by checksum
  - Servers may crash occasionally
    - recovery procedure to get its objects’ state
    - faulty servers are made to crash
  - A message delayed, lost, duplicated, or corrupted
    - detect corrupt messages (by checksum)
Transactions

☐ Some applications require a sequence of client requests to a server to be atomic in the sense that:
  ☐ Free from interference other concurrent clients
  ☐ Either all operations completed successfully or no effect at all in the presence of server crashes

☐ Consider transactions originating from database management systems

☐ Transactional file servers were built in the 1980s

☐ Transactions on distributed objects late 80s and 90s
  ☐ Use middleware components

☐ Transactions apply to recoverable objects and are atomic
A Client’s Banking Transaction

Transaction T:
\[ a.\text{withdraw}(100); \]
\[ b.\text{deposit}(100); \]
\[ c.\text{withdraw}(200); \]
\[ b.\text{deposit}(200); \]

- Transaction specifies a sequence of related operations involving bank accounts named A, B and C
  - Referred to as \(a\), \(b\) and \(c\) in the program
- First two operations transfer $100 from A to B
- Second two operations transfer $200 from C to B
- B’s balance should be $300 more
Atomicity of Transactions - Two Aspects

- **All or nothing**
  - Either completes successfully and the effects of all of its operations are recorded in the objects
  - Or (if it fails or is aborted) it has no effect at all
  - *Failure atomicity*: effects are atomic even with crashes
  - *Durability*: effects are saved in permanent storage

- **Isolation**
  - Each transaction performed without interference
    - no observation of a transaction's intermediate effects
  - Concurrency control ensures isolation
Operations in the *Coordinator* Interface

- Transaction capabilities may be added to a server of recoverable objects
  - Each transaction is created and managed by a *Coordinator* object whose interface follows:

  \[\text{openTransaction()} \rightarrow \text{trans};\]
  - starts a new transaction and delivers a unique TID to be used in the other operations in the transaction

  \[\text{closeTransaction(} \text{trans} \text{)} \rightarrow (\text{commit, abort});\]
  - ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted

  \[\text{abortTransaction(} \text{trans} \text{)};\]
  - aborts the transaction
## Transaction Life Histories

<table>
<thead>
<tr>
<th>Successful</th>
<th>Aborted by client</th>
<th>Aborted by server</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>openTransaction</td>
<td>openTransaction</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>operation</td>
</tr>
<tr>
<td>operation</td>
<td>operation</td>
<td>operation</td>
</tr>
</tbody>
</table>

- server aborts transaction

- operation ERROR reported to client

- closeTransaction
- abortTransaction

- A transaction is either successful (it commits)
  - Coordinator objects saved in permanent storage

- Or it is aborted by the client or the server
  - All temporary effects invisible to other transactions
  - How will client know when its transaction aborted?
    - next time it tries to access an object at the server
Concurrency Control

- Problems without appropriate concurrency control
  - Lost update
    - occurs when two transactions both read the old value of a variable and use it to calculate a new value
  - Inconsistent retrievals
    - occur when a retrieval transaction observes values that are involved in an ongoing updating transaction

- Show how serial equivalent executions of transactions can avoid these problems

- Assume that the operations are synchronized
  - deposit, withdraw, getBalance, setBalance
  - Their effect on the account balance is atomic
# The Lost Update Problem

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>balance = b.getBalance();</code></td>
<td><code>balance = b.getBalance();</code></td>
</tr>
<tr>
<td><code>b.setBalance(balance*1.1);</code></td>
<td><code>b.setBalance(balance*1.1);</code></td>
</tr>
<tr>
<td><code>a.withdraw(balance/10)</code></td>
<td><code>c.withdraw(balance/10)</code></td>
</tr>
</tbody>
</table>

| | | Problems? |
| `balance = b.getBalance();` | $200 | |
| `b.setBalance(balance*1.1);` | $220 | |
| `a.withdraw(balance/10)` | $80 | |
| `c.withdraw(balance/10)` | $280 | |

- **Initial balances:**
  - $A - $100, $B - $200, $C - $300

- Both transactions increase $B$’s balance by 10%
The Inconsistent Retrievals Problem

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

Transaction W:
- aBranch.branchTotal()

<table>
<thead>
<tr>
<th>Transaction V</th>
<th>Transaction W</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>$100 total = a.getBalance() $100</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>total = total + b.getBalance() $300</td>
</tr>
<tr>
<td></td>
<td>total = total + c.getBalance()</td>
</tr>
<tr>
<td>$300</td>
<td></td>
</tr>
</tbody>
</table>

- \( V \) transfers $100 from \( A \) to \( B \) while \( W \) calculates branch total (which should be $600)
Serial Equivalence

☐ If each one of a set of transactions has the correct effect when done on its own

☐ Then if they are done one at a time in some order the effect will be correct

☐ A *serially equivalent interleaving* is one in which the combined effect is the same as if the transactions had been done one at a time in some order

☐ The same effect means
  - ☐ Read operations return the same values
  - ☐ Instance variables of objects have same values at end

☐ Transactions are scheduled to avoid access overlap
Serially Equivalent Interleaving of T and U

Transaction T:
- balance = b.getBalance()
- b.setBalance(balance*1.1)
- a.withdraw(balance/10)

Transaction U:
- balance = b.getBalance()
- b.setBalance(balance*1.1)
- c.withdraw(balance/10)

- balance = b.getBalance() $200
- b.setBalance(balance*1.1) $220
- a.withdraw(balance/10) $80
- c.withdraw(balance/10) $278
- balance = b.getBalance() $220
- b.setBalance(balance*1.1) $242

- If T or U runs before the other, can’t get lost update
- Also true if they are run in serially equivalent ordering
Seriously Equivalent Interleaving of V and W

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

- If \( W \) run before or after \( V \), the problem will not occur
- Will not occur in a serially equivalent ordering
- Illustration is serial, but it need not be (How?)

\[
\begin{align*}
\text{total} &= \text{a.getBalance()} + 100 \\
\text{total} &= \text{total} + \text{b.getBalance()} + 400 \\
\text{total} &= \text{total} + \text{c.getBalance()} + \ldots
\end{align*}
\]
### read and write Operation Conflict Rules

<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**
  - Pair of operations conflicts if their combined effect depends on the order in which they were performed
Serial Equivalence and Conflicting Operations

- For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

- Consider
  - $T$: $x = \text{read}(i); \text{write}(i, 10); \text{write}(j, 20)$;
  - $U$: $y = \text{read}(j); \text{write}(j, 30); z = \text{read}(i)$;
  - Serial equivalence requires that either
    - $T$ accesses $i$ before $U$ and $T$ accesses $j$ before $U$, or
    - $U$ accesses $i$ before $T$ and $U$ accesses $j$ before $T$

- Serial equivalence is used as a criterion for designing concurrency control schemes.
Non-Serially Equivalent Interleaving of T and U

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{read}(i) )</td>
<td>( y = \text{read}(j) )</td>
</tr>
<tr>
<td>( \text{write}(i, 10) )</td>
<td>( \text{write}(j, 30) )</td>
</tr>
<tr>
<td>( \text{write}(j, 20) )</td>
<td>( z = \text{read}(i) )</td>
</tr>
</tbody>
</table>

- Each transaction’s access to \( i \) and \( j \) is serialised with respect to one another, but
- \( T \) makes all accesses to \( i \) before \( U \) does
- \( U \) makes all accesses to \( j \) before \( T \) does
- Therefore this interleaving is not serially equivalent
Recoverability from Aborts

- If a transaction aborts, server must make sure other concurrent transactions do not see any of its effects

- Two problems:
  - **Dirty reads**
    - an interaction between a read operation in one transaction and an earlier write operation on the same object
    - a transaction that committed with a dirty read is not recoverable
  - **Premature writes**
    - interactions between write operations on the same object by different transactions
    - one of which aborts
A Dirty Read when Transaction T Aborts

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| ```
  a.getBalance()
  a.setBalance(balance + 10)
``` |
| ```
  a.getBalance()
  a.setBalance(balance + 20)
``` |

<table>
<thead>
<tr>
<th>balance = a.getBalance()</th>
<th>$100</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.setBalance(balance + 10)</td>
<td>$110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>balance = a.getBalance()</th>
<th>$110</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.setBalance(balance + 20)</td>
<td>$130</td>
</tr>
</tbody>
</table>

commit transaction

- **U** has committed, so it cannot be undone
Recoverability of Transactions

- If a transaction (like U) commits after seeing the effects of a transaction that subsequently aborted, it is not recoverable.

- For recoverability, a commit is delayed until after the commitment of any other transaction whose state has been observed:
  - U waits until T commits or aborts.
  - If T aborts, then U must also abort.
Cascading Aborts

- Suppose that $U$ delays committing until after $T$ aborts
  - Then, $U$ must abort as well
  - If any other transactions have seen the effects due to $U$, they too must be aborted
  - Aborting of these latter transactions may cause still further transactions to be aborted

- Such situations are called *cascading aborts*

- For recoverability, delay the commits
  - $U$ waits to perform *getBalance* until $T$ commits or aborts
Avoiding Cascading Aborts

☐ To avoid cascading aborts
   Transactions are only allowed to read objects written by committed transactions
   To ensure this, any read operation must be delayed until other transactions that applied a write operation to the same object have committed or aborted

☐ Avoidance of cascading aborts is a stronger condition than recoverability
Overwriting Uncommitted Values

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.setBalance(105)</td>
<td>a.setBalance(110)</td>
</tr>
<tr>
<td>$100</td>
<td>$110</td>
</tr>
<tr>
<td>a.setBalance(105)</td>
<td>$105</td>
</tr>
<tr>
<td>a.setBalance(110)</td>
<td>$110</td>
</tr>
</tbody>
</table>

- **Premature writes**
- Some database systems keep *before images* and restore them after aborts
Strict Executions of Transactions

- Curing premature writes
  - If a recovery scheme uses before images
    - Write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted

- Strict executions of transactions
  - To avoid both dirty reads and premature writes
    - Delay both read and write operations
  - Strict executions of transactions
    - Both read and write operations are delayed until all transactions that previously wrote that object have either committed or aborted
  - Strict execution of transactions enforces isolation property

- Tentative versions are used during progress of a transaction
Nested Transactions

Transactions may be composed of other transactions

- Several transactions may be started from within a transaction
- We have a top-level transaction and sub-transactions which may have their own subtransactions
Nested Transactions

- To a parent, a sub-transaction is atomic with respect to failures and concurrent access
- Transactions at the same level (e.g., $T1$ and $T2$) can run concurrently but access to common objects is serialized
- A sub-transaction can fail independently of its parent and other sub-transactions
  - When it aborts, its parent decides what to do
    - start another subtransaction or give up
- The CORBA transaction service supports both flat and nested transactions
Advantages of Nested Transactions (over flat)

- Sub-transactions may run concurrently with other subtransactions at the same level
  - Allows additional concurrency within a transaction
  - Sub-transactions can work in parallel
    - Ex: \textit{branchTotal} operation can be implemented by invoking \textit{getBalance} at every account in the branch

- Sub-transactions can commit or abort independently
  - This is potentially more robust
  - A parent can decide on different actions according to whether a subtransaction has aborted or not
Commitment of Nested Transactions

- A transaction may commit or abort only after its child transactions have completed
- A sub-transaction decides independently to commit provisionally or to abort
  - Abort decision is final
- When a parent aborts, all sub-transactions are aborted
- When a sub-transaction aborts, the parent can decide whether to abort or not
- If the top-level transaction commits, then all of the sub-transactions that have provisionally committed can commit too, provided that none of their ancestors has aborted
Summary on Transactions

- We consider only transactions at a single server
  - Atomic in the presence of concurrent transactions
    - achieved by serially equivalent executions
  - Atomic in the presence of server crashes
    - they save committed state in permanent storage
    - they use strict executions to allow for aborts
    - they use tentative versions to allow for commit/abort
  - Nested transactions structured from sub-transactions
    - they allow concurrent execution of sub-transactions
    - they allow independent recovery of sub-transactions
Introduction to Concurrency Control

- Transactions must be scheduled so that their effect on shared objects is serially equivalent
- For serial equivalence
  - All access by a transaction to an object must be serialized with respect to another transaction’s access
  - All pairs of conflicting operations of two transactions should be executed in the same order
- A server can achieve serial equivalence by serialising access to objects (for instance) by the use of locks
  - To ensure a transaction is not allowed any new locks after it has released a lock
- “Two-phase” locking
Concurrency Control Protocols

- Applicable to servers whose operations can be modelled in terms of Read and Write operations
  - Essential that each operation be atomic
- Protocols designed to cope with conflicts
  - Each pair of operations conflicts when their combined effect depends on the order in which they are executed
  - Effect of an operation refers to the value of a data item set by a Write operation and the result returned by a Read operation
- Conflict rules
Concurrency Control Alternatives

- Locking
  - Server sets a lock with transaction ID on each data item just before it is accessed
  - Removes locks when transaction has completed
  - While locked, other transactions must wait or shared the lock
  - Can lead to deadlock

- Optimistic concurrency control
  - Conflicts are checked for before commits

- Timestamp ordering
Transactions T and U with Exclusive Locks

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$balance = b.getBalance()$</td>
</tr>
<tr>
<td>$b.setBalance(bal*1.1)$</td>
<td>$b.setBalance(bal*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td>$c.withdraw(bal/10)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td></td>
<td>openTransaction</td>
<td></td>
</tr>
<tr>
<td>$bal = b.getBalance()$</td>
<td>lock $B$</td>
<td>$bal = b.getBalance()$</td>
<td>waits for $T$’s lock on $B$</td>
</tr>
<tr>
<td>$b.setBalance(bal*1.1)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a.withdraw(bal/10)$</td>
<td>lock $A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlock $A$, $B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Initially the balances of $A$, $B$, and $C$ unlocked
Strict Two-Phase Locking

- Procedure
  - Delay a transaction reading/writing until other transactions that wrote the same object have committed or aborted
  - To enforce this, any locks applied during transaction progress are held until transaction commits or aborts
  - For recovery purposes, locks are held until updated objects have been written to permanent storage

- Prevent dirty reads and premature writes

- No new locks allowed after it has released a lock
  - “growing phase”: locks are acquired
  - “shrinking phase”: locks are released
Read-Write Conflict Rules

- Concurrency control protocols are designed to deal with conflicts between operations in different transactions on the same object.
- We describe the protocols in terms of read and write operations, which we assume are atomic.
- Read operations of different transactions do not conflict.
- Therefore, exclusive locks reduce concurrency more than necessary.
- The “many reader/ single writer” scheme allows several transactions to read an object or a single transaction to write it (but not both).
- It uses read locks and write locks.
  - Read locks are sometimes called shared locks.
## Lock Compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock already set</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
</tbody>
</table>

- If a transaction $T$ has already performed a read operation on a particular object, then a concurrent transaction $U$ must not write that object until $T$ commits or aborts.

- If a transaction $T$ has already performed a write operation on a particular object, then a concurrent transaction $U$ must not read or write that object until $T$ commits or aborts.
Lock Promotion

- Lost updates
  - Two transactions read an object and use to calculate a new value
- Lost updates are prevented by making later transactions delay their reads until the earlier ones have completed
- Each transaction sets a read lock when it reads and then promotes it to a write lock when it writes the same object
- When another transaction requires a read lock it will be delayed
- Lock promotion
  - Conversion of a lock to a stronger lock (more exclusive)
  - Demotion of locks (making them weaker) is not allowed
Use of Locks in Strict Two-Phase Locking

1. When an operation accesses an object within a transaction:
   - (a) If the object is not already locked, it is locked and the operation proceeds
   - (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked
   - (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds
   - (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds
     - Where promotion is prevented by a conflicting lock, (b) is used

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction
Lock Implementation

- The granting of locks will be implemented by the lock manager in the server
  - Holds a set of locks
  - Each lock is an instance of the class Lock
  - Variables refer to the object
    - holder(s) of the lock and its type
- Lock manager code uses wait (when an object is locked) and notify when the lock is released
- Lock manager provides setLock and unLock operations for use by the server
public class Lock {
    private Object object; // the object being protected by the lock
    private Vector holders; // the TIDs of current holders
    private LockType lockType; // the current type

    public synchronized void acquire(TransID trans, LockType aLockType) {
        while(/*another transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            } catch (InterruptedException e) { /*...*/ }
        }
        if(holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans);
            lockType = aLockType;
        } else if( /*another transaction holds the lock, share it*/ ) {
            if( /* this transaction not a holder*/ )
                holders.addElement(trans);
        } else if( /* this transaction is a holder but needs a more exclusive lock*/
            lockType.promote();
        }
    }
}
public synchronized void release(TransID trans)
{
    holders.removeElement(trans);   // remove this holder
    // set locktype to none
    notifyAll();
}
}
public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType){
        Lock foundLock;
        synchronized(this){
            // find the lock associated with object
            // if there isn’t one, create it and add to the hashtable

            foundLock.acquire(trans, lockType);
        }

        // synchronize this one because we want to remove all entries
        public synchronized void unLock(TransID trans) {
            Enumeration e = theLocks.elements();
            while(e.hasMoreElements()){
                Lock aLock = (Lock)(e.nextElement());
                if(/* trans is a holder of this lock*/) aLock.release(trans);
            }
        }
    }
}
Deadlock with Write Locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction $U$</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a$.deposit(100);</td>
<td>write lock $A$</td>
<td></td>
<td>$b$.deposit(200);</td>
<td>write lock $B$</td>
</tr>
<tr>
<td></td>
<td>$b$.withdraw(100)</td>
<td>waits for $U$'s lock on $B$</td>
<td></td>
<td>$a$.withdraw(200);</td>
<td>waits for $T$'s lock on $A$</td>
</tr>
<tr>
<td></td>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td></td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

- The *deposit* and *withdraw* methods are atomic
  - Although they read as well as write, they acquire write locks
The Wait-For Graph for the Previous Case

- Definition of deadlock
  - Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
  - A *wait-for graph* can be used to represent the waiting relationships between current transactions.
A Cycle in a Wait-For Graph

- Wait-for graph contains a cycle $T \rightarrow U \rightarrow \ldots \rightarrow V \rightarrow T$
  - Each transaction waits for the next in the cycle
  - All transactions are blocked waiting for locks
  - None of the locks can ever be released (deadlocked)
  - If one transaction is aborted, then its locks are released
  - Cycle is broken
Another Wait-For Graph

- $T$, $U$, and $V$ share a read lock on $C$
- $W$ holds write lock on $B$ (which $V$ is waiting for)
- $T$ and $W$ then request write locks on $C$
- Deadlock occurs (Why?)
Deadlock Prevention is Unrealistic

- We could lock all objects used by a transaction
  - When it starts and then wait until all available
  - Unnecessarily restricts access to shared resources
  - Might be impossible to predict at the start of a transaction which objects will be used

- Suppose we requesting locks in some predefined order
  - Can result in premature locking
  - Can lead to reduction in concurrency

- Trade off of allowing the possibility of deadlocks and greater concurrency and performance
Deadlock Detection

- How do we know deadlocks have occurred?
  - By finding cycles in the wait-for graph (*detection*)
  - Transaction must be selected to abort (breaks cycle)
  - Deadlock detection can be part of the lock manager
  - Use wait-for graph to check for cycles (when?)
  - Edges added / removed by the lock manager
    - *setLock* and *unLock* operations
  - When a cycle is detected, choose a transaction to abort
    - remove all the edges belonging to it from the graph
  - Hard to choose a victim
    - what happens if we choose the oldest transaction?
Timeouts on Locks

- Lock timeouts can be used to resolve deadlocks
  - Each lock is invulnerable for a given a limited period
  - After this time, a lock becomes vulnerable
  - Vulnerable lock remains if no other transaction waiting
  - If any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
    - object is unlocked and the waiting transaction resumes
  - Transaction whose lock broken is normally aborted

- Problems?
  - Break locks without deadlocks
  - Long transactions are penalized
  - Hard to set timeout values
Increasing Concurrency in Locking Schemes

- **Two-version locking**
  - Allows writing of tentative versions with reading of committed versions

- **Hierarchical locks**
  - Ex: `branchTotal` operation locks all the accounts with one lock whereas the other operations lock individual accounts (reduces the number of locks needed)
## Resolution of the Deadlock

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction U</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a.deposit(100)$</td>
<td>write lock $A$</td>
<td></td>
<td>$b.deposit(200)$</td>
<td>write lock $B$</td>
</tr>
<tr>
<td></td>
<td>$b.withdraw(100)$</td>
<td>waits for $U$’s lock on $B$</td>
<td></td>
<td>$a.withdraw(200)$</td>
<td>waits for $T$’s lock on $A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait (timeout elapses)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$T$’s lock on $A$ becomes vulnerable, unlock $A$, abort $T$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$a.withdraw(200)$</td>
<td>write locks $A$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unlock $A$, $B$</td>
</tr>
</tbody>
</table>
Lock Compatibility (*read*, *write*, *commit* locks)

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock to be set</th>
<th>read</th>
<th>write</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>OK</td>
<td>wait</td>
<td></td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>wait</td>
<td>wait</td>
<td></td>
</tr>
</tbody>
</table>
Drawbacks of Locking

- Overhead of lock maintenance and need to use locking even for read-only transactions
- Use of locks can result in deadlock and prevention or detection and recovery reduces concurrency
- To avoid cascading aborts, locks cannot be released until the end of the transaction
- Consider an optimistic approach
  - Likelihood of conflicting transactions is low
  - Transactions proceed as if no conflict will occur
  - Conflicts cause transactions to be aborted/restarted
Optimistic Concurrency Control

- Optimistic because likelihood of conflict is low
- Transaction proceeds without restriction until the `closeTransaction` (no waiting, therefore no deadlock)
- Then checked to see whether it has a conflict
  - When a conflict arises, a transaction is aborted
- Each transaction has three phases:
  - *Working phase* - uses tentative versions of objects, recording *read* and *write sets* for each transaction
  - *Validation phase* - look for conflicts at close, validate or resolve conflict with choice of which to abort
  - *Update phase* - tentative versions are made permanent, read only can commit immediately
Validation of Transactions

Transaction being validated

Later active transactions

Earlier committed transactions
Validation of Transactions

- To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase
  - Retained if successful; released if failure
- Monotonically increasing transaction numbers
  - Transaction number defines position in time
  - A transaction always finishes its read phase after all transactions with lower numbers
- Validation test is based on conflicts between operations in pairs of transactions $T_i$ and $T_j$
- Use read-write conflict rules
Validation of Transactions (continued)

- $Ti / Tj$
  - Read/Write: $Ti$ must not read items written by $Tj$
  - Write/Read: $Tj$ must not read items written by $Ti$
  - Write/Write: $Ti$ must not write items written by $Tj$ and $Tj$ must not write items written by $Tk$

- Make simplification that only one transaction may be in the validation and write phase at once

- Validation must test for overlaps

- Two forms of validation
  - Backward validation
  - Forward validation
Backward Validation

- Check with preceding overlapping transactions
  - Preceding reads not affected by writes
  - Check if read set overlaps with any preceding overlapping write sets
- Only way to resolve any conflicts is to abort the transaction undergoing validation
- Write sets of old committed versions of data items corresponding to recently committed transactions are retained until there are no unvalidated overlapping transactions with which they might conflict
Forward Validation

- Write set of $T_j$ compared with the read sets of all overlapping active transactions, those still in their read phase

- Alternatives to resolve conflicts
  - Defer validation until conflicting transactions have finished, but further conflicting ones may start
  - Abort all conflicting active transactions and commit the transaction being validated
  - Abort transaction being validated
Backward vs. Forward Validation

- Forward validation allows flexibility in resolution of conflicts where backward has only one choice.
- Read sets of transactions are larger than write sets.
  - Backward validation checks possibly large read set with old write sets.
  - Forward validation checks small write set with read sets of active transactions.
- Backward validation stores old write sets.
- Forward validation has to allow for new transactions.
Timestamp Ordering Concurrency Control

- Each operation in a transaction validated when carried out
  - If an operation cannot be validated, the transaction is aborted
  - Each transaction is given a unique timestamp when it starts
    - Defines its position in the time sequence of transactions
  - Transaction requests can be totally ordered by their timestamps

- Basic timestamp ordering rule (based on operation conflicts)
  - A request to write an object is valid only if that object was last read and written by earlier transactions
  - A request to read an object is valid only if that object was last written by an earlier transaction

- This rule assumes only one version of each object

- Refine the rule to make use of the tentative versions
  - To allow concurrent access by transactions to objects
Operation Conflicts for Timestamp Ordering

☐ Refined rule

☐ Tentative versions are committed in order of their timestamps (may wait) but no need for client to wait

☐ Read operations wait for earlier transactions to finish
  ➢ only wait for earlier ones (no deadlock)

☐ Each read/write operation is checked with conflict rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
<th>$T_c$ must not write an object that has been read by any $T_i$ where $T_i&gt;T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>write</td>
<td>read</td>
<td>$T_c$ must not write an object that has been read by any $T_i$ where $T_i&gt;T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.</td>
</tr>
<tr>
<td>2</td>
<td>write</td>
<td>write</td>
<td>$T_c$ must not write an object that has been written by any $T_i$ where $T_i&gt;T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
<tr>
<td>3</td>
<td>read</td>
<td>write</td>
<td>$T_c$ must not read an object that has been written by any $T_i$ where $T_i&gt;T_c$ this requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
</tbody>
</table>
Write Operations and Timestamps

- Versions and timestamps when we do $T_3$ write
- To be allowed, $T_3 \geq$ maximum read timestamp

(a) $T_3$ write

Before

$T_2$

After

$T_2$, $T_3$

(b) $T_3$ write

Before

$T_1$, $T_2$

After

$T_1$, $T_2$, $T_3$

(c) $T_3$ write

Before

$T_1$, $T_4$

After

$T_1$, $T_3$, $T_4$

(d) $T_3$ write

Before

$T_4$

After

$T_4$

Transaction aborts

Key:

- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$)

$T_1 < T_2 < T_3 < T_4$
Timestamp Ordering Write Rule

By combining rules 1 (write/read) and 2 (write/write) we have the following rule for deciding whether to accept a write operation requested by transaction $T_c$ on object D

○ Rule 3 does not apply to writes

if ($T_c \geq$ maximum read timestamp on $D$ \&\&
   $T_c >$ write timestamp on committed version of $D$)
   perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
   Abort transaction $T_c$
Using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction \( Tc \) on object \( D \):

- Accept it immediately, wait, or reject it if \( (T_c > \text{write timestamp on committed version of } D) \) {
  - let \( D_{\text{selected}} \) be the version of \( D \) with the maximum write timestamp \( \leq T_c \)
  - if \( (D_{\text{selected}} \text{ is committed}) \)
    - perform read operation on the version \( D_{\text{selected}} \)
  - else
    - Wait until the transaction that made version \( D_{\text{selected}} \) commits or aborts
      then reapply the read rule
} else
  - Abort transaction \( T_c \)
Read Operations and Timestamps

- Illustrates the timestamp, ordering read rule, in each case we have $T_3$ read
- A version whose write timestamp is $\leq T_3$ is selected

Key:
- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$)
$T_1 < T_2 < T_3 < T_4$
Transaction Commits with Timestamp Ordering

- When a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - Committed versions of an object must be created in timestamp order
  - Server may sometimes need to wait, but the client need not wait
  - To ensure recoverability, the server will save the “waiting to be committed versions” in permanent storage

- Timestamp ordering algorithm is strict because
  - The read rule delays each read operation until previous transactions that had written the object had committed or aborted
  - Writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted
Timestamp Ordering Concurrency Control

- Method avoids deadlocks
- But is likely to suffer from restarts
- *Ignore obsolete write*
  - If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway
  - However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item
- *Multiversion timestamp ordering*
  - Allows more concurrency by keeping multiple committed versions
  - Late read operations need not be aborted
# Timestamps in Transactions $T$ and $U$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$U$</th>
<th>Timestamps and versions of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$A$</td>
</tr>
<tr>
<td>$\text{openTransaction}$</td>
<td>$\text{openTransaction}$</td>
<td>$\text{RTS}$</td>
</tr>
<tr>
<td>$\text{bal} = b.\text{getBalance}()$</td>
<td>$\text{bal} = b.\text{getBalance}()$</td>
<td>${}$</td>
</tr>
<tr>
<td>$\text{b.setBalance(bal*1.1)}$</td>
<td>$\text{wait for } T$</td>
<td>$\text{S}, \text{T}$</td>
</tr>
<tr>
<td>$\text{a.withdraw(bal/10)}$</td>
<td>$\text{commit}$</td>
<td>$\text{S}, \text{T}$</td>
</tr>
</tbody>
</table>
Late *write* Operation Would Invalidate a *read*

Key:
- **Committed**: object produced by transaction $T_i$ (with write timestamp $T_i$ and read timestamp $T_k$)
- **Tentative**: object produced by transaction $T_i$ (with write timestamp $T_i$ and read timestamp $T_k$)

$t_1 < t_2 < t_3 < t_4 < t_5$
Comparing Methods for Concurrency Control

- **Pessimistic approach** (detect conflicts as they arise)
  - Timestamp ordering
    - serialisation order decided statically
    - better for transactions where reads >> writes
    - strategy for aborts - immediate
  - Locking
    - serialisation order decided dynamically
    - better for transactions where writes >> reads
    - strategy for aborts - waits but can get deadlock

- **Optimistic methods**
  - all transactions proceed, but may need to abort at the end
  - efficient operations when few conflicts
  - aborts lead to repeating work

- Above methods are not always adequate
  - in cooperative work may need user notification and involvement
Summary

- Operation conflicts form a basis for the derivation of concurrency control protocols
  - Protocols ensure serializability and allow for recovery by using strict executions (e.g., to avoid cascading aborts)
- Three alternative strategies are possible in scheduling an operation in a transaction:
  - (1) to execute it immediately, (2) to delay it, or (3) to abort it
  - Strict two-phase locking uses (1) and (2)
    - aborting in the case of deadlock
    - ordering according to when transactions access common objects
  - Timestamp ordering uses all three - no deadlocks
    - ordering according to the time transactions start
  - Optimistic concurrency control allows transactions to proceed without any form of checking until they are completed.
    - validation is carried out and starvation can occur