Transactions and Concurrency

Dr. Xiaobo Zhou

Introduction to Transactions

- **Transaction**: specified by a client as a set of operations on objects to be performed as an indivisible unit by the server managing the objects

- **The goal of transactions (all success or nothing)**
  - Goal: the objects managed by a server must remain in a consistent state
    - when they are accessed by multiple transactions and
    - 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; storage; and omission failures of communication

- **Designed for asynchronous systems: messages may be delayed**
Introduction to Concurrency Control

- All concurrency control protocols are based on the criteria of serial equivalence
  - **Locks** are used to order transactions that access the same objects according to the order of arrival of their operations at the objects
  - **Optimistic concurrency control** allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on objects
  - **Timestamp ordering** uses timestamps to order transactions that access the same objects according to their starting times

---

A Banking Example

**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

**Operations of the Branch interface**

- `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

- first we consider the 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
  - e.g. in Java use synchronized methods:
    - 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 (e.g. mutex)

Enhancing Client Cooperation by Synchronizing Server Operations

- Clients share resources via a server
  - e.g. some clients update server objects and others access them
  - servers with multiple threads require atomic objects
- but in some applications, clients depend on one another to progress
  - e.g. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- Java wait and notify methods allow threads to communicate with one another and to solve these problems
  - e.g. when a client requests a resource, the server thread waits until it is notified that the resource is available
- Wait-notify vs. rejection-retry
  - it would not be a good idea for a waiting client to poll the server to see whether a resource is yet available
  - it would also be unfair (later clients might get earlier turns)
Failure Model for Transactions

- Lampson’s failure model deals with failures of disks, servers and communication.
  - algorithms work correctly when predictable faults occur.
  - but if a disaster occurs, we cannot say what will happen
- Writes to permanent storage may fail
  - e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
  - reads can detect bad blocks by checksum
- Servers may crash occasionally.
  - when a crashed server is replaced by a new process its memory is cleared and then it carries out a recovery procedure to get its objects’ state
  - faulty servers are made to crash so that they do not produce arbitrary failures
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted.
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that:
  1. they are free from interference by operations being performed on behalf of other concurrent clients; and
  2. either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.
- Transactions originate from database management systems
- Transactional file servers were built in the 1980s
- Transactions on distributed objects late 80s and 90s
- Middleware components e.g. CORBA Transaction service.
- Transactions apply to recoverable objects and are intended to be atomic.
  
  Servers 'recover' - they are restated and get their objects from permanent storage
A Client's Banking Transaction

*Transaction T:*
- `a.withdraw(100);`
- `b.deposit(100);`
- `c.withdraw(200);`
- `b.deposit(200);`

- This transaction specifies a sequence of related operations involving bank accounts named A, B and C and referred to as `a, b` and `c` in the program.
- The first two operations transfer $100 from A to B.
- The second two operations transfer $200 from C to B.

Atomicity of Transactions

- The atomicity has two aspects:
  1. All or nothing:
     - It 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. This all-or-nothing effect has two further aspects of its own:
       - Failure atomicity:
         - The effects are atomic even when the server crashes;
       - Durability:
         - After a transaction has completed successfully, all its effects are saved in permanent storage.
  2. Isolation:
     - Each transaction must be performed without interference from other transactions - there must be no observation by other transactions 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 \( \text{trans} \). This identifier will be used in the other operations in the transaction.

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

    \[
    \text{abortTransaction}({\text{trans}});
    \]
    - aborts the transaction.

Operations in the Coordinator Interface (cont.)

- the client uses OpenTransaction to get TID from the coordinator
  - the client passes the TID with each request in the transaction
  - e.g. as an extra argument
    - deposit(\text{trans, amount})
    - or transparently if provided as middleware; The CORBA transaction service does uses 'context' to do this.

- To commit
  - the client uses closeTransaction and the coordinator ensures that the recoverable objects are saved in permanent storage

- To abort
  - the client asks either to commit or abort
  - the client uses abortTransaction and the recoverable objects and the coordinator must ensure that all temporary effects are invisible to other transactions
### 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>
<tr>
<td>operation</td>
<td>server aborts transaction</td>
<td>operation ERROR reported to client</td>
</tr>
<tr>
<td>closeTransaction</td>
<td>abortTransaction</td>
<td></td>
</tr>
</tbody>
</table>

- A transaction is either successful (it commits)
  - the coordinator sees that all objects are saved in permanent storage or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions
  - how will the server know when the client crashes?
  - how will the client know when the server has aborted its transaction?

  the client finds out next time it tries to access an object at the server.

### Concurrency Control

- We will illustrate the ‘lost update’ and the ‘inconsistent retrievals’ problems which can occur in the absence of appropriate concurrency control
  - a 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

- we show how serial equivalent executions of concurrent transactions can avoid these problems

- we assume that the operations deposit, withdraw, getBalance and setBalance are synchronized operations - that is, their effect on the account balance is atomic.
### The Lost Update Problem

**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)`

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Initial Balance</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td>$200</td>
<td>$220</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td>$200</td>
<td>$220</td>
</tr>
</tbody>
</table>

- the initial balances of accounts A, B, C are $100, $200, $300
- both transfer transactions increase B’s balance by 10%

### The Inconsistent Retrievals Problem

**Transaction V:**
- `a.withdraw(100)` // initial $200
- `b.deposit(100)` // initial $200

**Transaction W:**
- `aBranch.branchTotal()`

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Initial Balance</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>$200</td>
<td>$100</td>
</tr>
<tr>
<td>W</td>
<td>$300</td>
<td></td>
</tr>
</tbody>
</table>

we see an inconsistent retrieval because V has only done the withdraw part when W sums balances of A and B
Serial Equivalence Interleaving

• 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
  – the read operations return the same values
  – the instance variables of the objects have the same values at the end

The transactions are scheduled to avoid overlapping access to the accounts accessed by both of them

A Serially Equivalent Interleaving of T and U

Transaction T:
\[
\begin{align*}
\text{balance} &= \text{b.getBalance()} \\
\text{b.setBalance(balance*1.1)} \\
\text{a.withdraw(balance/10)}
\end{align*}
\]

Transaction U:
\[
\begin{align*}
\text{balance} &= \text{b.getBalance()} \\
\text{b.setBalance(balance*1.1)} \\
\text{c.withdraw(balance/10)}
\end{align*}
\]

their access to B is serial, the other part can overlap

• if one of T and U runs before the other, they can’t get a lost update,
  – the same is true if they are run in a serially equivalent ordering

(lost updates cured – but how this can be assured?)
A Serially Equivalent Interleaving of $V$ and $W$

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

**Transaction $W$:**
- $aBranch$.branchTotal()

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw 100</td>
<td>$100</td>
<td>withdraw 100</td>
<td>$100</td>
</tr>
<tr>
<td>deposit 100</td>
<td>$300</td>
<td>$100</td>
<td>$400</td>
</tr>
</tbody>
</table>

- we could overlap the first line of $W$ with the second line of $V$

- if $W$ is run before or after $V$, the problem will not occur
  - therefore it will not occur in a serially equivalent ordering of $V$ and $W$
  - the illustration is serial, but it need not be

**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
- a pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. read and write (whose effects are the result returned by read and the value set by write)
Serial Equivalence Defined by 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 criteria for designing schemes for concurrency control.

A Non-serially Equivalent Interleaving

- Each transaction’s access to \( i \) and \( j \) is serialised w.r.t 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

- Servers must record the effects of all committed transactions and none of the effects of aborted transactions
  - if a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects
- we study two problems:
  - ‘dirty reads’
    - an interaction between a read operation in one transaction and an earlier write operation on the same object (by a transaction that then aborts)
    - 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
- (getBalance is a read operation; setBalance a write operation)

A Dirty Read When Transaction T Aborts

<table>
<thead>
<tr>
<th>Transaction T:</th>
<th>Transaction U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.getBalance()</td>
<td>a.getBalance()</td>
</tr>
<tr>
<td>a.setBalance(balance + 10)</td>
<td>a.setBalance(balance + 20)</td>
</tr>
<tr>
<td>balance = a.getBalance()</td>
<td>$100</td>
</tr>
<tr>
<td>a.setBalance(balance + 10)</td>
<td>$110</td>
</tr>
<tr>
<td>$T$ subsequently aborts.</td>
<td>•$U$ reads $A$’s balance (which was set by $T$) and then commits</td>
</tr>
<tr>
<td>abort transaction</td>
<td>balance = a.getBalance()</td>
</tr>
<tr>
<td></td>
<td>$110</td>
</tr>
<tr>
<td></td>
<td>a.setBalance(balance + 20)</td>
</tr>
<tr>
<td></td>
<td>$130</td>
</tr>
<tr>
<td></td>
<td>commit transaction</td>
</tr>
</tbody>
</table>

What is the problem? $U$ has performed a dirty read
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

- e.g. $U$ waits until $T$ commits or aborts
- if $T$ aborts then $U$ must also abort

So what is the potential problem?

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.
  - the aborting of these latter transactions may cause still further transactions to be aborted.

- Such situations are called *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.

e.g. $U$ waits to perform `getBalance` until $T$ commits or aborts

Avoidance of cascading aborts is a stronger condition than recoverability

**For recoverability - delay commits**
## Premature Writes - Overwriting Uncommitted Values

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>before $T$ and $U$ the balance of $A$ was $100$</td>
<td>$100$</td>
<td>$105$</td>
<td>Interaction between write operations when a transaction aborts</td>
<td>$105$</td>
</tr>
</tbody>
</table>

- Some database systems keep ‘before images’ and restore them after aborts.
  - E.g. $100$ is before image of $T$’s write, $105$ is before image of $U$’s write
  - If $U$ aborts and $T$ commits, we get the correct balance of $105$,
  - But if $U$ commits and then $T$ aborts, we get $100$ instead of $110$
  - What happens if $T$ aborts and then $U$ 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
    - Executions of transactions are called **strict** if both read and write operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
    - The strict execution of transactions enforces the desired property of isolation

So, where is the concurrency?
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 subtransactions which may have their own subtransactions

Summary on Transactions

- We consider only transactions at a single server, they are:
  - atomic in the presence of concurrent transactions
    - which can be 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 are 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,
  (a) all access by a transaction to a particular object must be serialized with respect to another transaction’s access.
  (b) 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, e.g. by the use of locks
  - to ensure (b), a transaction is not allowed any new locks after it has released a lock (also useful to avoid cascading aborts)

- **Two-phase locking** - has a ‘growing’ and a ‘shrinking’ phase

---

**Recap: A Serially Equivalent Interleaving of T and U**

<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(balance*1.1)</td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>c.withdraw(balance/10)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>balance = b.getBalance()</th>
<th>$200</th>
</tr>
</thead>
<tbody>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>$220</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>$80</td>
</tr>
</tbody>
</table>

- If T was allowed to access A, then unlock it, then access B then access A again, another transaction U might access A while it was unlocked so we have a serially equivalent interleaving of T and U

---
Transactions T and U with Exclusive Locks (two-phase)

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

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

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

when T is about to use B, it is locked for T
when U is about to use B, it is still locked for T and U waits

when T commits, it unlocks B
the use of the lock on B effectively serialises access to B

- initially the balances of A, B and C unlocked

Strict Two-phase Locking

- strict executions prevent dirty reads and premature writes (if transactions abort).

What are dirty reads?

How can they be prevented?
- a transaction that reads or writes an object must be delayed until other transactions that wrote the same object have committed or aborted.
- to enforce this, any locks applied during the progress of a transaction are held until the transaction commits or aborts.
- this is called strict two-phase locking
- For recovery purposes, locks are held until updated objects have been written to permanent storage

- granularity - apply locks to small things e.g. bank balances
- there are no assumptions as to granularity in the schemes we present
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.

What decides whether a pair of operations conflict?

Lock Compatibility

- The operation conflict rules tell us that:
  1. 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.
  2. 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.

How inconsistent retrievals and lost updates are avoided?

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

- Lost updates – two transactions read an object and then use it 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.
- Lock promotion: the conversion of a lock to a stronger lock – that is, a lock that is more exclusive.
  - Demotion of locks (making them weaker) is not allowed.
- When another transaction requires a read lock it will be delayed (can anyone see a potential danger which does not exist when exclusive locks are used?)

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, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

- The server applies locks when the read/write operations are about to be executed
- The server releases a transaction’s locks when it commits or aborts
Lock Implementation

• The granting of locks will be implemented by a separate object in the server that we call the *lock manager*.
  – The client program has no access to operations for locking/unlocking, but the coordinator.

• the lock manager holds a set of locks, for example in a hash table.

• each lock is an instance of the class *Lock* and is associated with a particular object.
  – its variables refer to the object, the holder(s) of the lock and its type.

• the lock manager code uses *wait* (when an object is locked) and *notify* when the lock is released.

• the lock manager provides *setLock* and *unLock* operations for use by the server.

---

**Lock Class**

```java
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) {
        try {
            while(/* another transaction holds the lock in conflicting mode */) {
                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, LockType aLockType) {
        try {
            wait();
        } catch (InterruptedException e) {/* ... */ }

        if(holders.isEmpty()) { // no TIDs hold lock
            holders.removeElement(trans);
            lockType = aLockType;
        } else {
            if(/* another transaction holds the lock */) {
                if(/* this transaction is a holder */) holders.removeElement(trans);
            }
        }
    }
}
```

Continues on next slide
**Lock Class (continued)**

```java
public synchronized void release(TransID trans) {
    holders.removeElement(trans); // remove this holder
    // set locktype to none
    notifyAll();
}
```

**LockManager Class**

```java
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>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T: deposit(100)</td>
<td>write lock A</td>
<td>U: deposit(200)</td>
<td>write lock B</td>
</tr>
<tr>
<td>T accesses A → B</td>
<td></td>
<td>U accesses B → A</td>
<td></td>
</tr>
<tr>
<td>• • • waits for U’s</td>
<td></td>
<td>• • • waits for T’s</td>
<td></td>
</tr>
<tr>
<td>• • • lock on B</td>
<td></td>
<td>• • • lock on A</td>
<td></td>
</tr>
</tbody>
</table>

Then what must we do with T and U?

Is this serially equivalent? Can both T and U be allowed to commit?

The deposit and withdraw methods are atomic. Although in practice they read as well as write, they acquire write locks.

When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked. We have a ‘deadlock’. The lock manager must be designed to deal with deadlocks.

What can a lock manager do about deadlocks?

The Wait-for Graph for the Previous Figure

- 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

- Suppose a wait-for graph contains a cycle $T \rightarrow U \rightarrow \cdots \rightarrow V \rightarrow T$
  - each transaction waits for the next transaction in the cycle
  - all of these transactions are blocked waiting for locks
  - none of the locks can ever be released (the transactions are deadlocked)
  - If one transaction is aborted, then its locks are released and that cycle is broken

Another Wait-For Graph

- $T$, $U$ and $V$ share a read lock on $C$ and
- $W$ holds write lock on $B$ (which $V$ is waiting for)
- $T$ and $W$ then request write locks on $C$ and deadlock occurs
e.g. $V$ is in two cycles - look on the left

How to prevent deadlocks?
Deadlock Prevention vs. Deadlock Detection

- Is deadlock prevention realistic? How about lock-all-at-once?
- How about deadlock detection with abortion?

Timeouts on Locks

- Lock timeouts can be used to resolve deadlocks
  - each lock is given a limited period in which it is invulnerable.
  - after this time, a lock becomes vulnerable.
  - provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain.
  - but if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
    - (that is, the object is unlocked) and the waiting transaction resumes.
  - The transaction whose lock has been broken is normally aborted

What are the problems with lock timeouts?
Resolution of the Deadlock (w/ timeout)

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>a.deposit(100);</td>
<td>write lock A</td>
<td>U</td>
<td>b.deposit(200)</td>
<td>write lock B</td>
</tr>
<tr>
<td></td>
<td>b.withdraw(100)</td>
<td></td>
<td></td>
<td>a.withdraw(200);</td>
<td>waits for T's lock on A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>waits for U's lock on B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Timeout elapses</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T's lock on A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>becomes vulnerable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock A, abort T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a.withdraw(200);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write locks A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unblok A,B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increasing Concurrency in Locking Schemes

Transaction T:

\[ \text{sum} = \text{sum} + a.\text{getBalance}() \]
\[ \text{sum} = \text{sum} + b.\text{getBalance}() \]
.....

Transaction U:

\[ c.\text{withdraw}(100) \]
\[ a.\text{deposit}(100) \]
.....

\[ \text{sum} = \text{sum} + a.\text{getBalance}() \]
\[ \text{sum} = \text{sum} + b.\text{getBalance}() \]
..... no more read/write of \( a \) commit

In the beginning: \( \text{sum} = 0; \ a = 100; \ b = 100; \ c = 100. \)
Is it a serial equivalent interleaving?
What if a two-phase lock applied?
How to achieve higher concurrency?
Increasing Concurrency in Locking Schemes

**two-version locking**

- An optimistic scheme, using three types of lock: a read lock, a write lock, and a commit lock (only commit is exclusive)
- allows writing of tentative versions with reading of committed versions
- *read* operations only wait if another transaction is committing the same object
- Writing transactions risk waiting or even rejection when they attempt to commit

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>Lock requested read</th>
<th>Lock requested write</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>wait</td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
<td>wait</td>
<td>--</td>
</tr>
<tr>
<td>commit</td>
<td>wait</td>
<td>wait</td>
<td>--</td>
</tr>
</tbody>
</table>

Increasing Concurrency – Two-version Locking

- Can transactions commit their *write* operation immediately if other uncompleted transactions have read the same object?
- When the transaction coordinator receives a request to commit a transaction, it attempts to convert all that transaction's write locks to commit locks. What if any of the objects have outstanding read locks?
- Can deadlock occur when transactions are waiting to commit?

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>Lock requested read</th>
<th>Lock requested write</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>wait</td>
</tr>
<tr>
<td>write</td>
<td>OK</td>
<td>wait</td>
<td>--</td>
</tr>
<tr>
<td>commit</td>
<td>wait</td>
<td>wait</td>
<td>--</td>
</tr>
</tbody>
</table>
Increasing Concurrency – Two-version Locking

- What are the two main differences between the two-version locking scheme and an ordinary read-write locking scheme?
  - Clue: think about “Read” operations

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

Increasing Concurrency – Hierarchic Locks

- hierarchic locks
  - e.g. the branchTotal operation locks all the accounts with one lock whereas the other operations lock individual accounts (reduces the number of locks needed)
Optimistic Concurrency Control

- Why not using locking?
- Optimistic approach to the serialization of transactions
  - the scheme is called optimistic because the likelihood of two transactions conflicting is low (consider T: \(a[i] += 1\) and U: \(a[i]+=2\))
- a transaction proceeds without restriction until the closeTransaction (no waiting, therefore no deadlock)
- it is then checked to see whether it has come into conflict with other transactions
- when a conflict arises, a transaction is aborted; and need to be restarted by the client

Optimistic Concurrency Control (cont.)

- Each transaction has three phases:
  
  **Working phase**
  - the transaction uses a tentative version of the objects it accesses (allows the transaction to abort w/o effect on the objects.)
  - the coordinator records the readset and writeset of each transaction
  
  **Validation phase**
  - at closeTransaction the coordinator validates the transaction (looks for conflicts)
  - if the validation is successful the transaction can commit.
  - if it fails, either the current transaction, or one it conflicts with is aborted
  
  **Update phase**
  - If validated, the changes in its tentative versions are made permanent.
  - read-only transactions can commit immediately after passing validation.

With locks we had deadlock T \(\rightarrow\) U at i and U \(\rightarrow\) T at j.
What would happen with the optimistic scheme?
Validation of Transactions

- We use the read-write conflict rules
  - to ensure a particular transaction is serially equivalent with respect to all other overlapping transactions
- each transaction is given a transaction number when it starts validation (working phase finished; the number is kept if it commits)
- the rules ensure serializability of transaction $T_v$ (transaction being validated) with respect to transaction $T_i$

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$ forward</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$ backward</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

*Validation can be simplified by omitting rule 3 (if no overlapping of validate and update phases – make sure only one in the validation and update)*

Validation of Transactions (cont.)

The earlier committed transactions are $T_1$, $T_2$ and $T_3$. $T_1$ committed before $T_v$ started. (earlier means they started validation earlier)

Earlier committed transactions

Rule 1 ($T_v$’s write vs $T_i$’s read) is satisfied because reads of earlier transactions were done before $T_v$ entered validation (and possible updates)

Rule 2 - check if $T_v$’s read set overlaps with write sets of earlier $T_i$ (fails if overlap) $T_2$ and $T_3$ committed before $T_v$ finished its working phase.

Rule 3 - (write vs write) assume no overlap of validate and commit.
Backward Validation of Transactions

Backward validation of transaction $T_v$

boolean valid = true;
for (int $T_i = startTn+1; T_i <= finishTn; T_i++$) {
    if (read set of $T_i$ intersects write set of $T_i$) valid = false;
}

In validating $T_v$, which transactions need to be examined?
Which to abort in case of false validation?
Need to be checked if no read in $T_v$?
Write-only transactions always pass validation? Y

to carry out this algorithm, we must keep write sets of recently committed transactions. What is the problem?

4/4/2011
Forward Validation

- Rule 1. the write set of $T_v$ is compared with the read sets of all overlapping active transactions
  - The write set of $T_v$ must be compared with the read sets of $active1$ and $active2$ ($T1,T2,T3$ already be compared with $Tv$)
- Rule 2. (read $T_v$ vs write $T_i$) is automatically fulfilled because the active transactions do not write until after $T_v$ has completed.

```java
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++) {
    if (write set of $Tv$ intersects read set of $Tid$) valid = false;
}
```

Forward validation of transaction $Tv$

```java
read only transactions always pass validation? Yes.
```

Forward Validation Discussion

```java
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++) {
    if (write set of $Tv$ intersects read set of $Tid$) valid = false;
}
```

the scheme must allow for the fact that read sets of active transactions may change during validation, increasing risk of false validation

What to do in case of false validation?
Comparison of Forward and Backward Validation

- In conflict, choice of transaction to abort
  - forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
- In general read sets > than write sets.
  - backward validation
    - compares a possibly large read set against the old write sets
    - overhead of storing old write sets
  - forward validation
    - checks a small write set against the read sets of active transactions
    - need to allow for new transactions starting during validation

Is forward validation deterministic about “no conflict”?

Does Tv need to have both backward and forward validation?

Starvation vs. Deadlock

- Starvation
  - after a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed

Starvation vs deadlock?

What you propose to handle Starvation?
Timestamp Ordering Concurrency Control

- each operation in a transaction is validated when it is carried out
  - if an operation cannot be validated, the transaction is aborted
  - each transaction is given a unique timestamp when it starts.
    - The timestamp defines its position in the time sequence of transactions.
  - requests from transactions 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 the order of their timestamps (wait if necessary) but there is no need for the client to wait
  - but read operations wait for earlier transactions to finish
    - only wait for earlier ones (no deadlock)
  - each read or write operation is checked with the conflict rules

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

(a) $T_3$ write

Before

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 $T_4$

Key:

- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$). In cases (a), (b) and (c) $T_3 < \text{w.t.s}$ on committed version and a tentative version with w.t.s $T_3$ is inserted at an appropriate place in the list of versions.

+ this illustrates the versions and timestamps, when we do $T_3$ write. For write to be allowed, $T_3 \geq$ maximum read timestamp (not shown)

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 $T_c$
**Timestamp Ordering Read Rule**

- by using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction $T_c$ on object $D$. That is, whether to
  - accept it immediately,
  - wait or
  - reject it

```
if ($T_c >$ write timestamp on committed version of $D$) {
    let $D_{selected}$ be the version of $D$ with the maximum write timestamp $\leq T_c$
    if ($D_{selected}$ is committed)
      perform read operation on the version $D_{selected}$
    else
      Wait until the transaction that made version $D_{selected}$ commits or aborts
      then reapply the read rule
  }
else
  Abort transaction $T_c$
```

---

**Read Operations and Timestamps**

- (a) $T_3$ read
  
  read proceeds

  in case (c) the read operation is directed to a tentative version and the transaction must wait until the maker of the tentative version commits or aborts

- (b) $T_3$ read
  
  in case (d) there is no suitable version and $T_3$ must abort

- (c) $T_1$ read
  
  read waits

- (d) $T_4$ read
  
  Transaction aborts

* illustrates the timestamp, ordering read rule, in each case we have $T_3read$. In each case, a version whose write timestamp is $\leq T_3$ is selected

4/4/2011 70
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
  - The 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
- The 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

Remarks on Timestamp Ordering Concurrency Control

- The method avoids deadlocks, but is likely to suffer from restarts
  - Modification known as 'ignore obsolete write' rule is an improvement
    - 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 (page 506)
    - Allows more concurrency by keeping multiple committed versions
      - Late read operations need not be aborted
      - There is not time to discuss the method now
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>openTransaction</td>
<td>bal = b.getBalance()</td>
<td>${T}$</td>
</tr>
<tr>
<td></td>
<td>b.setBalance(bal*1.1)</td>
<td>$S$</td>
</tr>
<tr>
<td></td>
<td>wait for $T$</td>
<td>$T$</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>bal = b.getBalance()</td>
<td>$T$</td>
</tr>
<tr>
<td></td>
<td>b.setBalance(bal*1.1)</td>
<td>$U$</td>
</tr>
<tr>
<td></td>
<td>c.withdraw(bal/10)</td>
<td>$T , U$</td>
</tr>
</tbody>
</table>

Late write Operation Would Invalidate a Read

4/4/2011

73

4/4/2011

74
Comparison of Methods for Concurrency Control

• **optimistic methods**
  – all transactions proceed, but may need to abort at the end
  – efficient operations when there are few conflicts, but aborts lead to repeating work

• **pessimistic approach (detect conflicts as they arise)**
  – timestamp ordering: serialisation order decided statically
  – locking: serialisation order decided dynamically
  – timestamp ordering is better for transactions where reads >> writes,
  – locking is better for transactions where writes >> reads
  – strategy for aborts
    • timestamp ordering – immediate
    • locking– waits but can get deadlock

• **the above methods are not always adequate e.g.**
  – in cooperative work there is a need for user notification
  – applications such as cooperative CAD need user involvement in conflict resolution

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. Starvation can occur.