DISTRIBUTD COMPUTING MODULE 5

TRANSACTIONS AND CONCURRENCY CONTROL

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

SIMPLE SYNCHRONIZATION (WITHOUT TRANSACTIONS)

Atomic operations at the server

- Use of multiple threads is beneficial to **performance**
- Threads allow **operations** from multiple clients to run **concurrently**
- Clients can possibly **access** the same objects **concurrently**
- Methods of objects should be designed for multi-threaded context
- If *deposit & withdraw* are not designed for multi-threaded program:
  - actions of two or more concurrent executions of the method could be arbitrarily interleaved
  - have strange effects on the instance variables of the account objects

- If one thread invokes a **synchronized method** on an object:
  - then that object is effectively locked
  - Another thread that invokes its synchronized methods: blocked until the lock is released
  - Forces the execution of threads to be separated in time and
  - Ensures that instance variables are accessed in a consistent manner
- Without synchronization:
  - 2 separate *deposit* invocations might read the balance before either has incremented
  - resulting in an incorrect value
- Should synchronize any method that accesses an instance variable that can vary
- *Atomic operations* - Free from interference from concurrent operations

TRANSACTIONS

- Clients require a sequence of separate requests to a server to be **atomic** :

  1. free from interference by operations

  2. In the presence of server crashes:

     Either all the operations completed successfully or they must have no effect at all

- A client's banking transaction

  Transaction T:

  a.withdraw(100);

  b.deposit(100);

  c.withdraw(200);

  b.deposit(200);

- **Transaction:** execution of a sequence of client requests

There are two aspects to atomicity:

- **All or nothing**: This all-or-nothing effect has two further aspects:
  - ***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.

- **Isolation:** Each transaction must be performed without interference from other transactions

- **Properties of Transaction:**
  - **A: Atomicity**
  - **C: Consistency**
  - **I: Isolation**
  - **D: Durability**

- Each transaction is created and managed by a coordinator
- *Coordinator* interface

Operations in the *Coordinator* interface

openTransaction( ) → trans;
Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) → (commit, abort);
Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
Aborts the transaction.

- A transaction is achieved by cooperation among
  - a client program
  - some recoverable objects
  - a coordinator

- *A*n extra argument in each operation of a recoverable object

  deposit(trans, amount)

  Deposits *amount in the account for transaction with TID trans*

- A transaction completes when the client makes a *closeTransaction* request.
  - If transaction has progressed normally-reply states that the transaction is *committed*
  - Transaction **may have to *abort***
  - When a transaction is aborted the recoverable objects and the coordinator must ensure that **none of its effects are visible to future transactions**

Transaction life histories

Successful	*Aborted by client*	*Aborted by server*	
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	*transaction* →	•
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Service actions related to process crashes

- **If a server process crashes unexpectedly, it** is replaced

- The new server process
  - aborts any uncommitted transactions and
  - restores the values of the objects to that of most recently committed transaction

- To deal with a **client that crashes** unexpectedly during a transaction
  - servers can give each transaction an expiry time
  - abort any transaction that has not completed before its expiry time

Client actions related to server process crashes

- **If a server crashes while a transaction** is in progress:
  - Client becomes aware when one of the operations returns an exception after a timeout

- If a **server crashes and is then replaced:**
  - the transaction will no longer be valid
  - the client must be informed via an exception

- In either case:
  - the client must then formulate a plan
  - possibly in consultation with the human user
  - for the completion or abandonment of the task

CONCURRENCY CONTROL

The lost update problem:

- bank accounts *A, B and C, whose initial balances are $100, $200 and* $300, respectively

The lost update problem

Transaction *T*:		Transaction *U*:	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

- **Inconsistent retrievals**

The inconsistent retrievals problem

Transaction *V*:		Transaction *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance( )	$100
		total = total + b.getBalance( )	$300
		total = total + c.getBalance( )	
b.deposit(100)	$300	•	
		•	

Serial equivalence

An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order is a *serially equivalent interleaving*.

A serially equivalent interleaving of *T* and *U*

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

A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal( )*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance( )	$100
		total = total + b.getBalance( )	$400
		total = total + c.getBalance( )	
		...	

Conflicting operations

- combined effect depends on the order of execution of the 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.

Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

A non–serially-equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Consider as an example the transactions *T* and *U*, defined as follows:

T: x = *read(i)*; *write(i, 10)*; *write(j, 20)*;
U: y = *read(j)*; *write(j, 30)*; *z = read (i)*;

- Each transaction's access to i and j is serialized with respect to one another
  - T makes all of its accesses to i before U does
  - U makes all of its accesses to j before T does

- Serially equivalent orderings require one of the following two conditions:
  1. T accesses i before U and T accesses j before U
  2. U accesses i before T and U accesses j before T

- Serial equivalence is a criterion for the derivation of concurrency control protocols

- Three alternative approaches to concurrency control are commonly used:
  - Locking
  - optimistic concurrency control
  - timestamp ordering

Recoverability from aborts

- Servers must record
  - all the effects of committed transactions
  - none of the effects of aborted transactions

- a transaction may abort but not affect other concurrent transactions if it does so

Dirty reads

- Caused by the interaction between
  - a *read operation in one transaction*
  - *an earlier write operation in* another transaction
  - on the same object

A dirty read when transaction *T* aborts

Transaction *T*:		Transaction *U*:	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

Recoverability of transactions

- A transaction like *U* is not recoverable
- any transaction *is in danger of* a dirty read delays its commit operation
- until after the commitment of the transaction whose uncommitted state has been observed
- In the example, *U delays its commit until after T commits*
- *In* the case that *T aborts, then U must abort as well*

Cascading aborts

To avoid cascading aborts

- Transactions are only allowed to read objects that were written by committed transactions

- *Read delayed until other transactions' write to the same object have* ***committed or aborted***

- *Avoidance of* cascading aborts is a stronger condition than recoverability

Premature writes

Overwriting uncommitted values

Transaction *T*:		Transaction *U*:	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

- Systems may implement the action of *abort by restoring* ***'before images'*** of all the *writes of a transaction.*

- Case 1: **Transaction *U aborts and T commits***
  - *A is $100 initially, which is* the 'before image' of *T's write*
  - *$105 is the 'before image' of U's write*
  - *Thus* if *U aborts, we get the correct balance of $105*

Overwriting uncommitted values

Transaction *T*:		Transaction *U*:	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

- Case 2: ***U commits and then T aborts***.
  - the 'before image' of *T's write is $100*
  - *we get the wrong balance of $100*
  - *The balance should be* $110

- Case 3: ***T aborts and then U aborts***
  - *the 'before image' of U's write is $105*
  - *we* get the wrong balance of $105
  - the balance should revert to $100

Recovery scheme that uses before images:

- *write* operations delayed until earlier transactions that updated the same objects have either committed or aborted

Strict executions of transactions

- *Both read and write delayed* until all transactions that previously wrote that object have either committed or aborted.

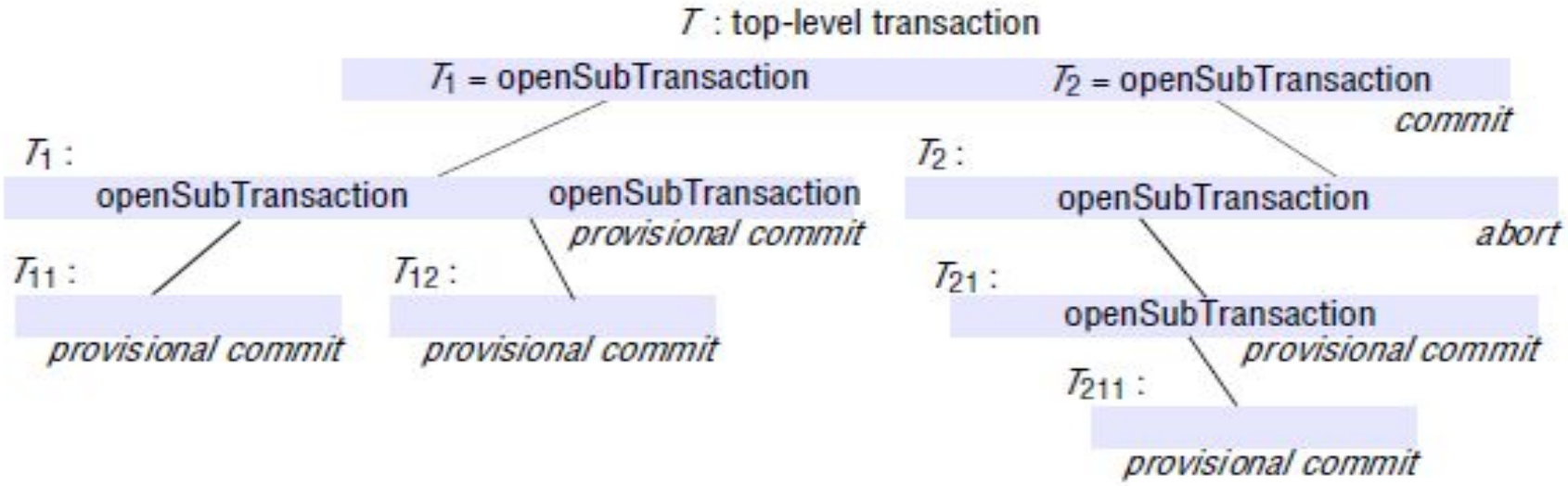
- The strict execution of transactions enforces the desired property of isolation

Flat transaction

- *All of its work at the same level* between *openTransaction and a commit or abort*
- *It is not possible to commit or* abort parts of it

Nested transactions

- Transactions can be composed of other transactions
- Transactions may be regarded as modules that are composed as required
- *Top-level* transaction - Outermost transaction
- *Subtransactions*- Transactions other than the top-level transaction



- Subtransaction fail independently of its parent & other subtransactions

- When a subtransaction aborts, parent may choose an alternative subtransaction to complete its task

Nested transactions have the following main advantages:

1. Subtransactions at the same level may run concurrently

- This can allow additional concurrency in a transaction.
- Subtransactions running in different servers can work in parallel
- Eg, *branchTotal operation in* banking - invoking *getBalance at every account* in the branch

2. Subtransactions can commit or abort independently

- Set of nested subtransactions is potentially more robust.
- Flat transaction- one transaction failure causes the whole transaction to be restarted
- Nested- Parent can decide on different actions when a subtransaction has aborted or not

RULES FOR COMMITTING OF NESTED TRANSACTIONS

- A transaction may commit or abort only after its child transactions have completed

- When a subtransaction completes
  - it makes an independent decision either to commit provisionally or to abort
  - Its decision to abort is final

- When a parent aborts, all of its subtransactions are aborted.
  - For example, if *T2* aborts then *T21 and T211 must also abort, even though they may have provisionally* committed

- When a subtransaction aborts, the parent can decide whether to abort or not.
  - In the example, *T decides to commit although T2 has aborted*

- If the top-level transaction commits, then
  - all of the subtransactions that have provisionally committed can commit too
  - provided that none of their ancestors has aborted

  In our example, *T's commitment*
  - *allows T1, T11 and T12 to commit,*
  - *but* not *T21 and T211 since their parent, T2, aborted.*

- *The effects of a* subtransaction are not permanent until the top-level transaction commits

- In some cases, the top-level transaction may decide to abort because one or more of its subtransactions have aborted.

  - Eg. Consider the following *Transfer* transaction:

    Transfer $100 from *B to A*

    a.deposit(100)

    b.withdraw(100)

LOCKS

- Transactions must be scheduled so that their effect on shared data is serially equivalent.
- A server can serialize access to the objects
- A simple example of a serializing mechanism: **exclusive locks**

In this locking scheme:

- lock object about to be used by operation of client's transaction
- If a client requests access to a locked object :
  - the request is suspended
  - client must wait until the object is unlocked

Transactions *T* and *U* with exclusive locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

- Assumption- when *T and U* start, balances of *A, B and C are not yet locked*
- *When transaction T is* about to use account *B, it is locked for T*
- *When transaction U is about to use B it is still* locked for *T, so U waits*
- *When T is committed, B is unlocked,* whereupon transaction *U is resumed*
- *The use of the* ***lock on B effectively serializes the*** **access to** ***B***
- Serial equivalence:

  All of a transaction's accesses to a particular object must be serialized with respect to accesses by other transactions

Two Phase Locking

- All pairs of conflicting operations of two transactions should be executed in the same order

- A transaction is not allowed any new locks after it has released a lock

Two phases-

- **Growing phase**: first phase during which new locks are acquired

- **Shrinking phase**: second phase during which the locks are released

Strict Two Phase Locking

- Since transactions may abort
- Strict executions needed to prevent dirty reads & premature writes
- Under a strict execution regime:
  - a transaction that needs to read or write an object delayed
  - other transactions that wrote the same object must commit or abort first
- To enforce this rule:
  - any locks applied during the progress of a transaction are held until the transaction commits or aborts

- Locks prevent other transactions reading or writing the objects
- When a transaction commits- locks are held till updates are written to permanent storage
- It ensures recoverability
- ***Granularity*** *of application of concurrency control to objects is an important* issue

- Concurrency control applied to all objects at once- Limits scope for concurrent access to objects in a server severely

- Portion of objects to which access must be serialized should be as small as possible

- Just that part involved in each operation requested by transactions

- Concurrency control protocols cope with *conflicts between* operations in different transactions on the same object

  Eg

  – Pairs of *read operations* from different transactions on the same object do not conflict
  – **Simple exclusive lock that is used for both *read and write operations reduces concurrency more* than is necessary**

MANY READERS/SINGLE WRITER SCHEME

- Two types of locks are used: *read locks and write locks*
  - Before a transaction's *read operation- read lock is set on the* object
  - Before a *write operation is performed- write lock is set on* the object

- Pairs of *read operations from different transactions do not conflict*
- *Attempt* to set read lock on an object with a read lock is **always successful**
- All the transactions reading the same object share its read lock
- Read locks are sometimes called ***shared locks***

Lock compatibility

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

- To prevent-
  - **Inconsistent retrievals-**

    Perform retrieval transaction before or after the update transaction
  - **Lost updates-**

    Make later transactions delay their reads until the earlier ones have completed

- To achieve this-
  - Set a **read lock** when it reads an object
  - Then ***promoting it to a write lock*** *when it* writes the same object
  - Subsequent transactions' read lock request delayed until any current transaction has completed

- Transaction with a **shared read lock**
  - **Cannot promote its read lock to a write lock**
  - Must request a write lock and **wait for other read locks to be released**

- Lock promotion - the **conversion of a lock to a stronger lock**

- **Stronger lock** –a lock that is more exclusive

- It is unsafe to demote a lock held by a transaction before it commits

- Client has no access to operations for locking or unlocking items of data
  - Locking performed when *read and write operations are about to be applied*
  - Unlocking is performed by *commit or abort operations of the transaction* coordinator

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.

CORBA Concurrency Control Service

- May be used to apply concurrency control on behalf of transactions
- Means of associating a **collection of locks** (***lockset****) with a resource*
- *A lockset allows locks to be* acquired or released.
- A lockset's *lock method will acquire a lock or block until the lock* is free
- Other methods allow locks to be promoted or released
- Transactional locksets require transaction identifiers as arguments
- CORBA transaction service tags all client requests in a transaction with the **transaction identifier**
- The transaction coordinator is responsible for releasing the locks when a transaction commits or aborts

Lock implementation

- ***Lock manager:*** Separate object in the server for granting of locks

- *Lock manager holds a set of locks, for* example in a hash table

- A lock is
  - an instance of the class '*Lock*'
  - *associated with* a particular object

- E*ach instance of Lock* maintains the following information in its instance variables:
  - the identifier of the locked object
  - the transaction identifiers of the transactions that currently hold the lock (shared locks can have several holders)
  - a lock type

Lock class

```
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();
    }
}
```

- The methods of *Lock are synchronized:*
  - threads attempting to acquire or release a lock will not interfere with one another

- ***Wait*** *method used to wait for another thread to release the lock*

- The ***acquire*** *method carries out the rules:*
  - Its arguments specify a transaction identifier and the type of lock required
  - It tests whether the request can be granted
  - If another transaction holds the lock in a conflicting mode, it invokes *wait*
  - *Wait causes the caller's thread to be* suspended until a corresponding *notify*
  - *Wait is enclosed in 'while', as* all waiters are notified and some may not be able to proceed
  - When, eventually, the condition is satisfied, the method sets the lock appropriately

- The ***release*** *method's*
  - *arguments specify the transaction identifier of the transaction that* is releasing the lock
  - It removes the transaction identifier from the holders
  - sets the lock type to *none and calls notifyAll*
  - *The method notifies all waiting threads*

Lock Manager

All requests to set locks and to release them on behalf of transactions are sent to an instance of *LockManager:*

- **The *setLock method***
  - It finds a lock for that object in its hashtable or, if necessary, creates one
  - It then invokes the *acquire method of that lock*

- **The *unLock method***
  - It finds all of the locks in the hashtable that have the given transaction as a holder
  - For each one, it calls the *release method.*

LockManager class

```
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 it 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);
       }
   }
}
```

Locking rules for nested transactions

The aim of a locking scheme is to serialize access to objects so that:

1:

Each set of nested transactions is a single entity that must be prevented from observing the partial effects of any other set of nested transactions.

2:

Each transaction within a set of nested transactions must be prevented from observing the partial effects of the other transactions in the set.

To enforce **Rule 1**:

- **Every lock acquired** by a successful subtransaction is ***inherited by its parent when it completes***
  - *Inherited locks are also* inherited by ancestors
  - This form of inheritance passes from child to parent!

- **Top-level transaction eventually inherits all of the locks**
  - Ensures that locks can be held until the top-level transaction has committed or aborted
  - Prevents members of different sets of nested transactions observing one another's partial effects

To enforce **Rule 2**:

- **Parent transactions are not allowed to run concurrently with their child**
  - If a parent transaction has a lock, it *retains the lock* when its child is executing
  - Child transaction temporarily acquires the lock from its parent for its duration

- **Subtransactions at the same level are allowed to run concurrently**
  - **when they** access the same objects, the locking scheme must serialize their access

The following rules describe lock acquisition and release:

- For a subtransaction to acquire a read lock on an object
  - no other active transaction can have a write lock on that object
  - the only retainers of a write lock are its ancestors

- For a subtransaction to acquire a write lock on an object
  - no other active transaction can have a read or write lock on that object
  - the only retainers of read and write locks on that object are its ancestors

- When a subtransaction commits
  - its locks are inherited by its parent
  - the parent retain the locks in the same mode as the child

- When a subtransaction aborts
  - its locks are discarded
  - If the parent already retains the locks, it can continue to do so

DEADLOCKS

The use of locks can lead to deadlock.

Deadlock with write locks

Transaction *T*		Transaction *U*	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
		•••	
•••		•••	
•••		•••	

- This is a deadlock situation – two transactions are waiting, and each is dependent on the other to release a lock so it can resume.

- Deadlock is particularly common when clients are involved in an interactive program

- Result in many objects being locked and remaining so, thus preventing other clients using them

- **Locking of subitems** in structured objects **avoid conflicts and possible deadlock situations**

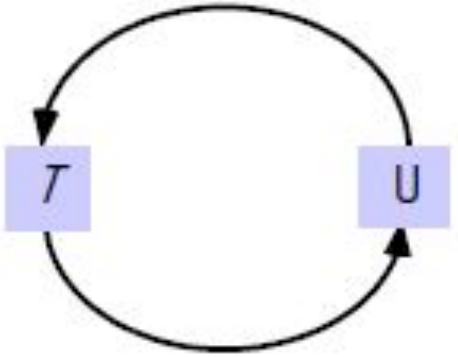
  For example,

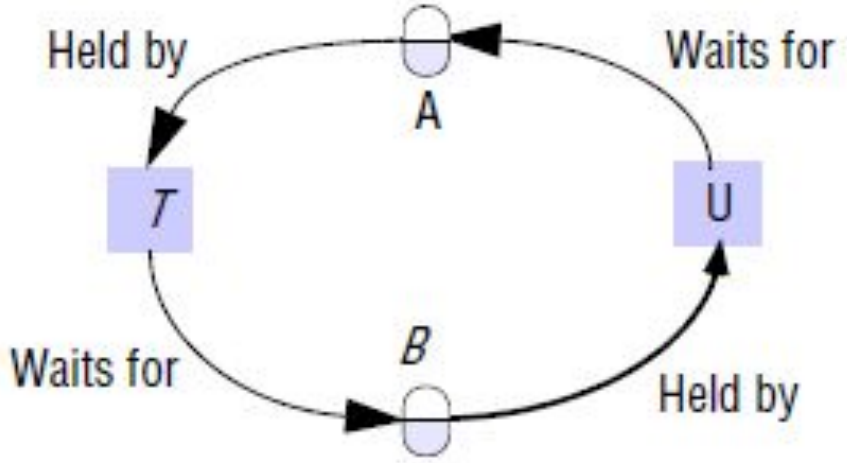
  – a day in a diary could be structured as a set of timeslots, each of which can be locked independently for updating.

- **Hierarchic locking schemes** are useful if the application requires a different granularity of locking for different operations,

DEFINITION OF DEADLOCK

- State in which each member of a group of transactions is waiting for some other member to release a lock
- ***Wait-for graph*** represents waiting relationships b/w current transactions
- In a wait-for graph the
  - nodes represent transactions
  - edges represent wait-for relationships between transactions
  - Edge from node *T to node U - T is* waiting for *U to release a lock*
- Deadlock - *T and U both attempted to acquire an object held by* the other
- Therefore *T waits for U and U waits for T*

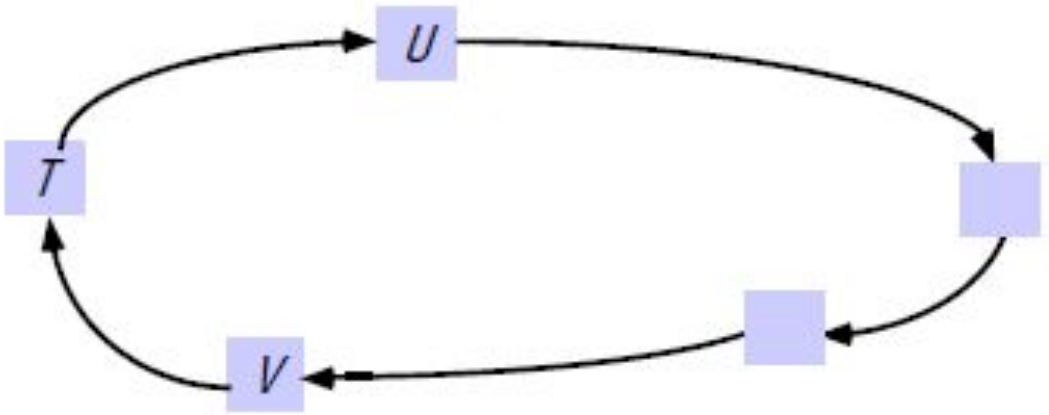




A wait-for graph containing a cycle *T -> U->V->T* .

- *Each transaction is waiting for the next in the cycle*
- *All of these* transactions are blocked waiting for locks
- None of the locks can ever be released, hence, deadlocked
- If one of the transactions in a cycle is aborted, then its locks are released and that cycle is broken

A cycle in a wait-for graph

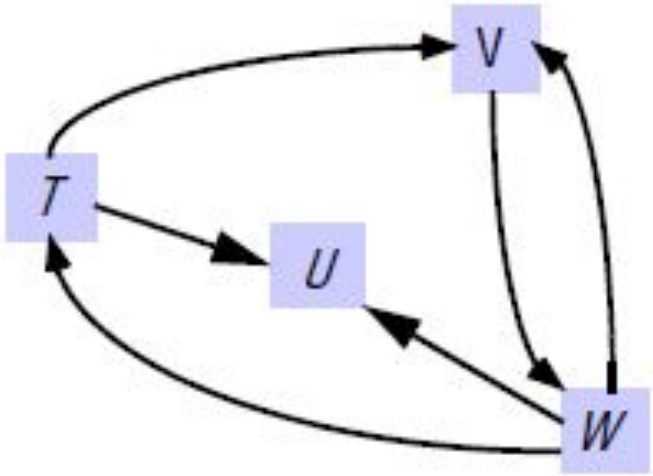


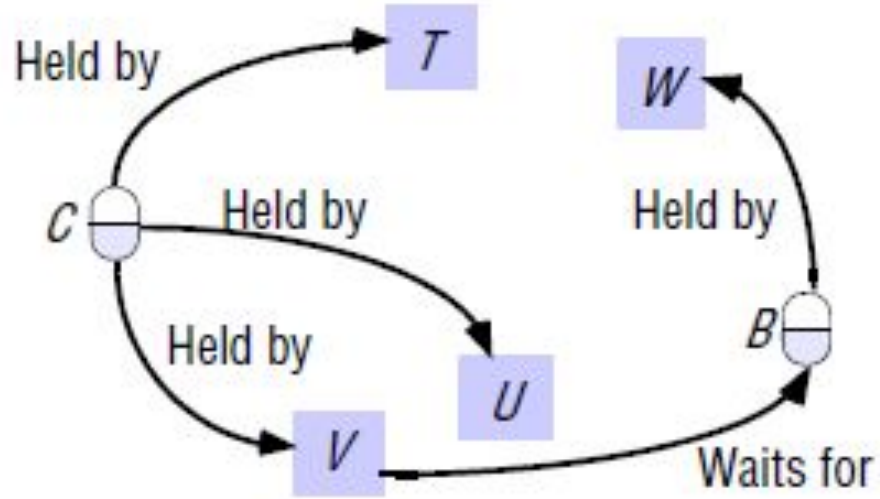
State:

- Transactions *T, U and V share a read* lock on object *C*
- *Transaction W holds a write lock on object B*
- Transaction *V is waiting to obtain a lock on B*

Scenario:

- *The* transactions *T and W then request* ***write locks*** *on object C*
- *A deadlock situation arises* in which *T waits for U and V, V waits for W, and W waits for T, U and V.*





- Although each transaction can wait for only one object at a time, it may be involved in several cycles

- Transaction *V is* involved in two cycles: *V ->W->T ->V and*
  V->W-> V

- Suppose that transaction *V is aborted. This will release V's lock* on *C and the two cycles involving V will be broken*

DEADLOCK PREVENTION

- Lock all of the objects used by a transaction when it starts
- Apparently simple but not very good way to overcome the deadlock problem
- Needs to be done as a single atomic step so as to avoid deadlock at this stage
- Such a transaction cannot run into deadlocks with other transactions
- But this approach unnecessarily restricts access to shared resources.
- At times impossible to predict at the start of a transaction which objects will be used
  - Eg. interactive applications, browsing-style applications
- Request locks on objects in a predefined order
- Could result in premature locking and a reduction in concurrency

UPGRADE LOCKS

- CORBA's Concurrency Control Service introduces a third type of lock, called 'upgrade'
- Intended to avoid deadlocks
- Deadlocks are often caused by two conflicting transactions
- Taking read locks and then attempting to promote them to write locks
- A transaction with an upgrade lock on a data item
  - is permitted to read that data item
  - but this lock conflicts with upgrade locks set by other transactions on the same item
- Cannot be set implicitly by the use of a read operation
- Must be requested by the client.

DEADLOCK DETECTION

- Deadlocks detected by finding cycles in the wait-for graph
- A transaction must be selected for abortion to break the cycle
- Deadlock detection software can be **part of the lock manager**
- It must hold a **representation of the wait-for graph**
- It can check it for cycles from time to time
- Edges are added to the graph and removed from the graph by the lock manager's **setLock and unLock operations**

Transaction	*Waits for transaction*
T	*U, V*
V	*W*
W	*T, U, V*

- Edge T->U is added when T requests for a lock on an object already locked on behalf of transaction U
- When lock is shared, several edges may be added.
- Edge **T ->U is deleted** whenever **U releases a lock** that T is waiting for and allows T to proceed

Exception:

- If a transaction **shares a released lock**, the lock is **not released**
- The **edges leading to a particular transaction are removed**

- **Presence of cycles** may be **checked**
  - each time an edge is added
  - Or less frequently to avoid unnecessary overhead

- If deadlock- One of the transactions in the cycle is chosen and **aborted**

- The **corresponding node and the edges** involving it are **removed** from wait-for graph

- Happens when the aborted transaction has its **locks removed**

- The choice of the transaction to abort is **not simple**

- Some **factors** that may be taken into account are
  - the **age of the transaction**
  - **the number of cycles** in which it is involved

Timeouts

- Lock timeouts- commonly used method for resolution of deadlocks
- Each lock is given a limited period in which it is **invulnerable**
- After this time, a lock becomes **vulnerable**
- No other transaction is competing for the locked object->
  - object with a vulnerable lock remains locked
- If any other transaction is waiting to access the object with a vulnerable lock->
  - the lock is broken (the object is unlocked)
  - And the waiting transaction resumes
- The transaction whose lock has been broken is normally aborted

Problems with the use of timeouts as a remedy for deadlocks:

- Transactions with vulnerable locks aborted when other transactions are waiting, but there is **actually no deadlock**

- In an overloaded system,
  - Number of transactions timing out will increase
  - **Penalizes lengthy transactions**

- Hard to decide on an **appropriate length for a timeout**

DEADLOCK RESOLUTION

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T	•••	
		a.withdraw(200);	write lock *A*
			unlock *A*, *B*

- Deadlock detection is used with timeouts
- Using lock timeouts, we can resolve the deadlock as shown
  - The write lock for T on A becomes vulnerable after its timeout period
  - Transaction U is waiting to acquire a write lock on A
  - Therefore, T is aborted and it releases its lock on A
  - U can resume and complete the transaction
- When **transactions access objects located in several different servers**, the possibility of **distributed deadlocks** arises.
- In a distributed deadlock, the wait-for graph can involve objects at multiple locations.

INCREASING CONCURRENCY IN LOCKING SCHEMES

- Locking rules based on conflicts between read and write operations
- Granularity at which locks are applied is as small as possible
- Still some scope for increasing concurrency

- Two approaches used to deal with this issue:

1. **Two-version locking:**
   setting of exclusive locks is delayed until a transaction commits

2. **Hierarchic locks**: mixed-granularity locks are used

TWO-VERSION LOCKING

- This is an optimistic scheme that allows
  - one transaction to **write tentative versions of objects** while
  - other transactions **read from the committed versions** of the same objects
- **Read operations only wait if object is being currently commited**
- **More concurrency than read-write locks**
- Write transactions risk waiting/rejection when they attempt to commit
- Cannot commit writes immediately if other uncompleted transactions have read the same objects
- Therefore, transactions wait until reading transactions have completed

- Deadlocks may occur when transactions are waiting to commit.
- **May need to be aborted** while waiting to commit, **to resolve deadlocks**
- This variation on strict two-phase locking uses **three types of lock**:
  a read lock, a write lock and a commit lock
- Before a transaction's **read operation** is performed- read lock must be set
- Attempt to set a read lock is successful unless the object has a **commit lock**
- Before a transaction's **write operation** is performed- write lock must be set
- Attempt to set a write lock is successful unless the object has
  - a **write lock**
  - **a commit lock**

- When transaction coordinator receives a **request to commit-**
  - it attempts to **convert all that transaction's write locks to commit locks**
  - If any of the objects have **outstanding read locks**, the transaction must **wait**
- The compatibility of read, write and commit locks is shown

Lock compatibility (*read*, *write* and *commit* locks)

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	–
	commit	wait	wait	–

Differences In Performance

Two Version Locking	Ordinary Read-write Locking
1. Read operations delayed only while the transactions are being committed (commit protocol takes only a small fraction of the time required to perform an entire transaction)	1. Read operations delayed during the entire execution of transactions
2. Read operations of one transaction can cause delays in committing other transactions	2. **Read operations of one transaction** do not cause delays **delays in committing** other transactions

HIERARCHIC LOCKS

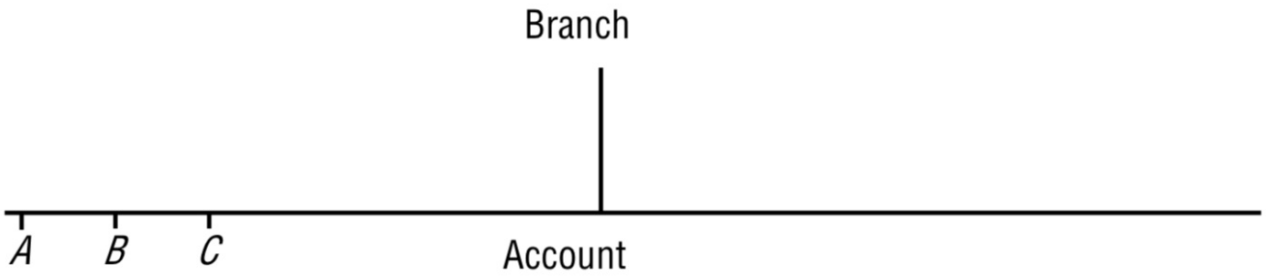
- Sometimes granularity suitable for one operation is not appropriate for another

Banking example:

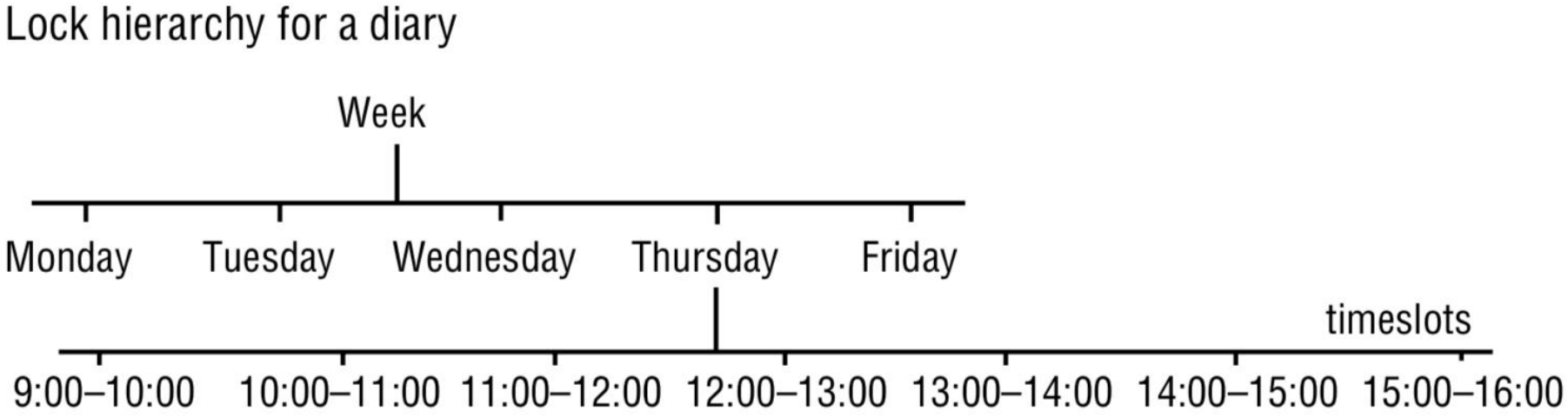
  - Majority of the operations require locking at the granularity of an account
  - BranchTotal operation is different – reads the values of all account balances

- To reduce locking overhead, **allow locks of mixed granularity to coexist**
- **Setting of a parent lock** - same effect as **setting all the equivalent child locks**
- This economizes on the number of locks to be set

Lock hierarchy for the banking example



Banking example, the branch is the parent and the accounts are children



- Operation to view a week - read lock set at the top of this hierarchy
- Operation to enter an appointment - write lock set on a given time slot
- Read lock on a week prevents write operations on any of the substructures

- **Each node** in the hierarchy can be **locked**
  - Owner of the lock has **explicit access to the node**
  - Owners children have **implicit access**
  - Eg. A read-write lock on the branch implicitly read-write locks all the accounts
- Intention to read-write lock set on parent before granting lock to child node
- The intention lock
  - compatible with other intention locks
  - conflicts with read and write locks according to the usual rules

Lock compatibility table for hierarchic locks

For one object		*Lock to be set*			
		read	*write*	*I-read*	*I-write*
Lock already set	*none*	OK	OK	OK	OK
	read	OK	wait	OK	wait
	write	wait	wait	wait	wait
	I-read	OK	wait	OK	OK
	I-write	wait	wait	OK	OK

- A deposit operation needs to set a write lock on a balance
- But first it attempts to set an intention to write lock on the branch
- These rules prevent these operations running concurrently

- Hierarchic locks reduce the number of locks when mixed granularity locking is required
- Mixed granularity allows each transaction to lock a portion
- Size is chosen according to the needs
- CORBA Concurrency Control Service supports variable-granularity locking with intention to read and intention to write lock types

OPTIMISTIC CONCURRENCY CONTROL

Drawbacks of locking:

- Lock maintenance represents an **overhead**
  - Even read only operations require locks
- The use of locks can result in deadlock
  - **Deadlock prevention reduces concurrency severely**
  - Deadlocks must be resolved either by timeouts or by deadlock detection
  - Neither of these is wholly satisfactory for use in interactive programs.
- **To avoid cascading aborts, locks held till the end of the transaction**
  - Significantly reduces the potential for concurrency

- The alternative approach is ‘optimistic’
- Based on the observation:
  - Mostly, likelihood of two transactions accessing the same object is low
- Transactions are allowed to proceed as though there were no possibility of conflict
- When a conflict arises, some transaction is generally aborted and will need to be restarted by the client

PHASES OF A TRANSACTION

1. Working phase:

- Each transaction has a tentative version of each of the objects that it updates.
- Tentative version- most recently committed value of the object
- Use of tentative versions allows the transaction to abort (with no effect on the objects)
  - either during the working phase or
  - if it fails validation due to other conflicting transactions
- Read operations are performed immediately

- Write operations record the new values of the objects as tentative values
- Several concurrent transactions-> several different tentative values of the same object may coexist
- Two records are kept of the objects accessed within a transaction:
  - a read set containing the objects read by the transaction
  - a write set containing the objects written by the transaction.
- All read operations are performed on committed versions- hence no dirty reads

2. Validation phase:

- Validation done when the closeTransaction request is received
- Check whether its operations on objects conflict with other transactions
- If the validation is successful, then the transaction can commit
- If the validation fails, then some form of conflict resolution must be used
- Either the current transaction orthose with which it conflicts is aborted

3. Update phase:

- If validated transaction:
  - All of the changes recorded in its tentative versions are made permanent

- **Read-only** transactions commit immediately after passing validation

- **Write** transactions commit once the tentative

VALIDATION OF TRANSACTIONS

- Validation **uses the read-write conflict rules**
- Ensure that transaction is serially equivalent with other **overlapping transactions**
- Each transaction is assigned a transaction number when it enters the validation phase (client issues a **closeTransaction**)
- If transaction is **validated** & completes successfully- number retained
- The number is released for reassignment if
  - transaction **fails** the validation checks and is **aborted**
  - the **transaction is read only**

- Transaction numbers are
  - **integers** assigned in **ascending sequence**
  - number of a transaction therefore defines its position in time
  - a transaction always finishes its working phase after all transactions with lower numbers

- Transaction Ti always precedes a transaction Tj if i < j

- If transaction number were assigned at the beginning of working phase:
  - transaction that reached the end of the working phase before one with a lower number would have to wait to be validated until the earlier one had completed

Rules for serializability

For a transaction Tv to be serializable with respect to an overlapping transaction Ti:

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v.
read	*write*	2.	T_v must not read objects written by T_i.
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i.

- V**alidation and update phases** are generally **short** compared to working phase

- Rule: Only one transaction may be in the validation and update phase at one time

- Thus rule 3 is satisfied
  - Restriction on *write operations and no dirty reads can occur*

- Entire validation and update phases can be implemented as a critical section

- Prevents overlapping

- In order to increase concurrency:
  - part of the validation and updating may be implemented outside the critical section
  - essential that the **assignment of transaction numbers is performed sequentially**

- Current transaction number is like a pseudo-clock –
  that ticks whenever a transaction completes successfully

- Ensure that rules 1 and 2 are obeyed by testing for overlaps between the objects of transactions *Tv and Ti*

- There are two forms of validation – backward and forward

- **Backward validation**
  - checks the transaction undergoing validation with other preceding overlapping transactions
  - those that entered the validation phase before it

- **Forward validation**
  - checks the transaction undergoing validation with other later transactions
  - those which are still active

Backward validation

Rule 1

- *Read operations of earlier overlapping transactions* were performed before the validation of *Tv started*
- *Hence they cannot be affected by the writes* of the current transaction

Rule 2

- Validation of transaction *Tv checks* whether its read set *overlaps with any* of the write sets of earlier overlapping transactions, *Ti*
- *If there is any overlap,* the validation fails

- ***startTn*** - *biggest transaction number assigned (to some other committed* transaction) at the time when transaction *Tv started its* ***working phase***
- ***finishTn*** - biggest transaction number assigned at the time when *Tv entered the* ***validation phase***

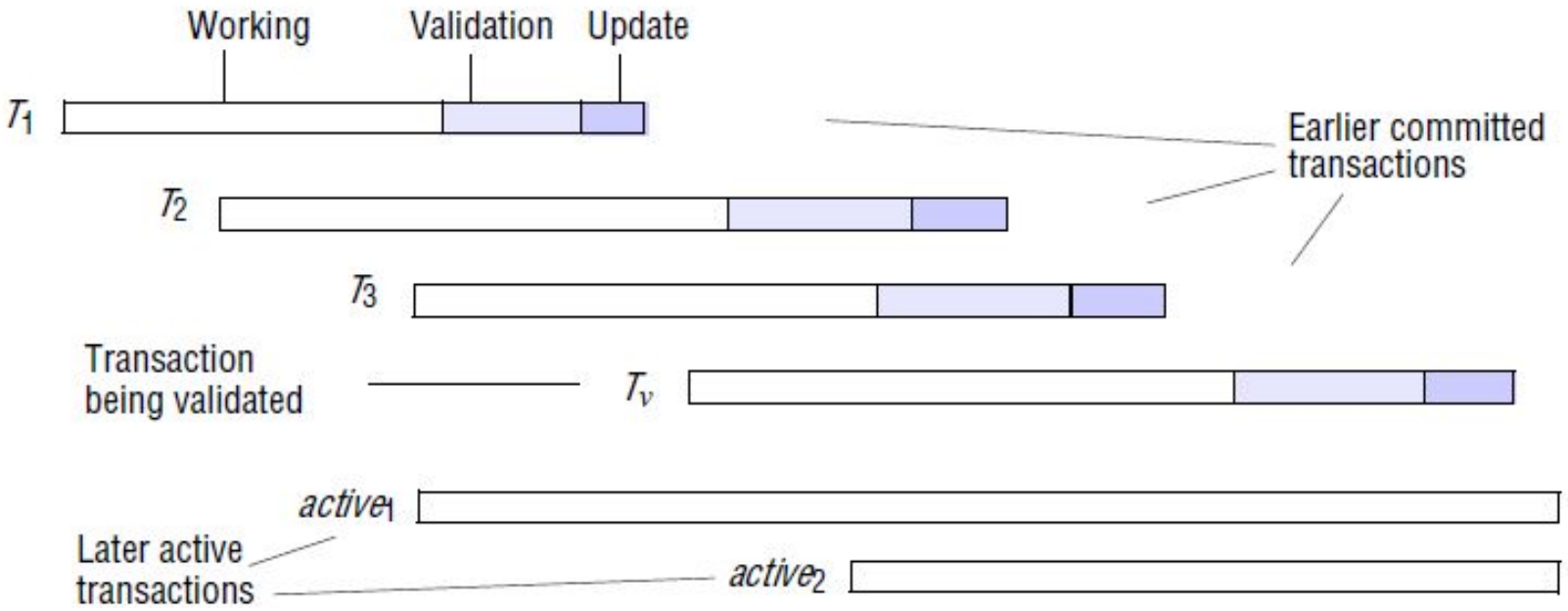
- Description of the algorithm for the validation of *Tv:*

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

Overlapping transactions considered in the validation of a transaction *Tv.*

- *Time increases from left to right*
- *The earlier committed transactions* are *T1, T2 and T3*
- *T1 committed before Tv started*
- *T2 and T3 committed before Tv* finished its working phase
- *StartTn + 1 = T2 and finishTn = T3*
- *Backward validation:* read set of *Tv compared with write sets of T2 & T3*

16.28 Validation of transactions



- To resolve any conflicts - abort the transaction undergoing validation
- Here transactions that have no *read operations (only write* operations) need not be checked
- Write sets of old committed versions of objects retained until there are no unvalidated overlapping transactions
- Preceding transaction list by Transaction service
  - Transaction number, *startTn and write set of validated* transaction *recorded*
- This list is ordered by transaction number

- In an environment with long transactions, the retention of old write sets of objects may be a problem

- For example, write sets of *T1, T2, T3 and Tv must be retained until the active transaction active1 completes*

- *Although* the active transactions have transaction identifiers, they do not yet have transaction numbers

Forward validation

- **In forward validation of the transaction *Tv, the write set of Tv is*** compared with the read sets of all overlapping active transactions – those that are still in their working phase (rule 1).

- Rule 2 is automatically fulfilled because the active transactions do not write until after *Tv has completed.*

- *Let the active transactions have* (consecutive) transaction identifiers *active1 to activeN.*

- *The following program describes* the algorithm for the forward validation of *Tv:*

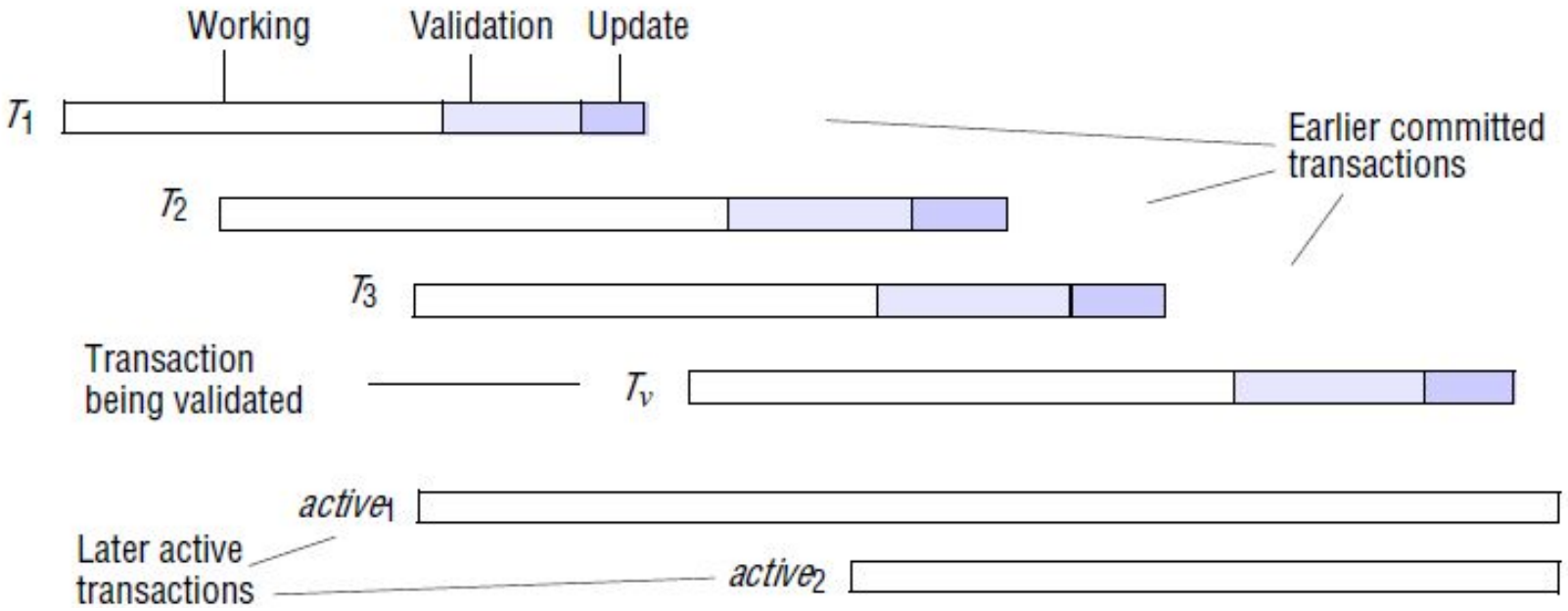
```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
if (write set of Tv intersects read set of Tid) valid = false;
}
```

- the write set of transaction *Tv must be compared with the read sets of* the transactions with identifiers *active1 and active2.*

- read sets of active transactions may change during validation and writing

- Read-only transactions always pass the validation check

16.28 Validation of transactions



During a Conflict :

- Choice of whether to abort the validating transaction or to pursue some alternative way of resolving the conflict

- Strategies-

  - Defer the validation until a later time when the conflicting transactions have finished

  - Abort all the conflicting active transactions and commit the transaction being validated

  - Abort the transaction being validated

    (disadvantage that future conflicting transactions may abort, in which case the transaction under validation has aborted unnecessarily)

THANK YOU.