Chapter 13

Transaction and Concurrency Control

Sequence of operations as a transaction

- Sometimes clients want a sequence of separate requests to a server to be treated as a single unit.
- A *transaction* defines a sequence of server operations that is guaranteed by the server to be atomic:
  - It is free from interference by operations of concurrent clients.
  - Either all operations must be completed successfully or they must have no effect at all in the presence of server crash.

Transaction T involving 3 Account objects:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

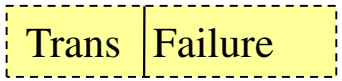
Properties of Atomic Transaction

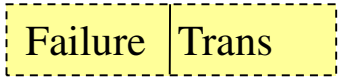
rewind or undo

inconsistency

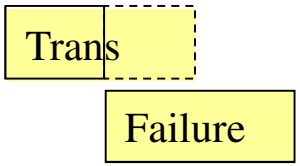
ACID

- *All or nothing*: a transaction either completes successfully and effects of all operations are recorded in the objects, or it has no effect at all if it fails or is aborted.

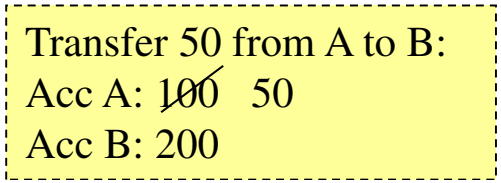




  - *Failure atomicity*: effects are atomic even when the server crashes (effects are not a result of a server that fails half-way).



  - *Durability*: after a transaction has completed successfully, all its effects are saved in permanent storage.
    - So objects are recoverable even though a server later crashes.
- *Isolation*: intermediate effects of a transaction must not be visible to other transactions.



Defining a transaction

- The server of the objects or a separate process can be a coordinator, implementing the *Coordinator* interface.

openTransaction() -> trans;
starts a new transaction and delivers a unique TID *trans*. This identifier will accompany each method invocation to identify the call as part of this transaction, e.g. *deposit(trans, amount).*

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.

```
coord.openTransaction() ->TID
a.withdraw(TID, 50)
b.deposit(TID, 50)
coord.closeTransaction(TID) -> commit/abort
```

- Middleware, e.g. CORBA, provides a transaction service as a coordinator, and objects that can be involved in a transaction have to extend interface *TransactionalObject*.

Java Transaction API, WS-Coordination, WS-AtomicTransaction

Problems of concurrent transactions

- Concurrent transactions may interfere with each other if they are not controlled properly and intermediate effects can be observed.
  - Lost update
  - Inconsistent retrieval
- The following *Account* example assumes all its methods are synchronized operations.

Only one thread at a time can access an object to change the instance variables:

public synchronized void deposit(int amount) throws RemoteException {…}

This makes it an *atomic operation.*

T -> U then a=80 b=242 c=278

U -> T then a=78 b=242 c=280

Lost Update

initial balances: a=100, b=200, c=300

openTransaction() is omitted here

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

Interrupts

$20

$20

Separate execution: b's balance = 242 $200+20+22

Concurrent execution: b's balance = 220; U's update is overwritten by T

Inconsistent Retrieval

initial balances: a=200, b=200

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100)	$100		
		total = a.getBalance() (new value)	$100
		total = total+b.getBalance() (old value)	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

Separate execution: a's balance + b's balance = 400

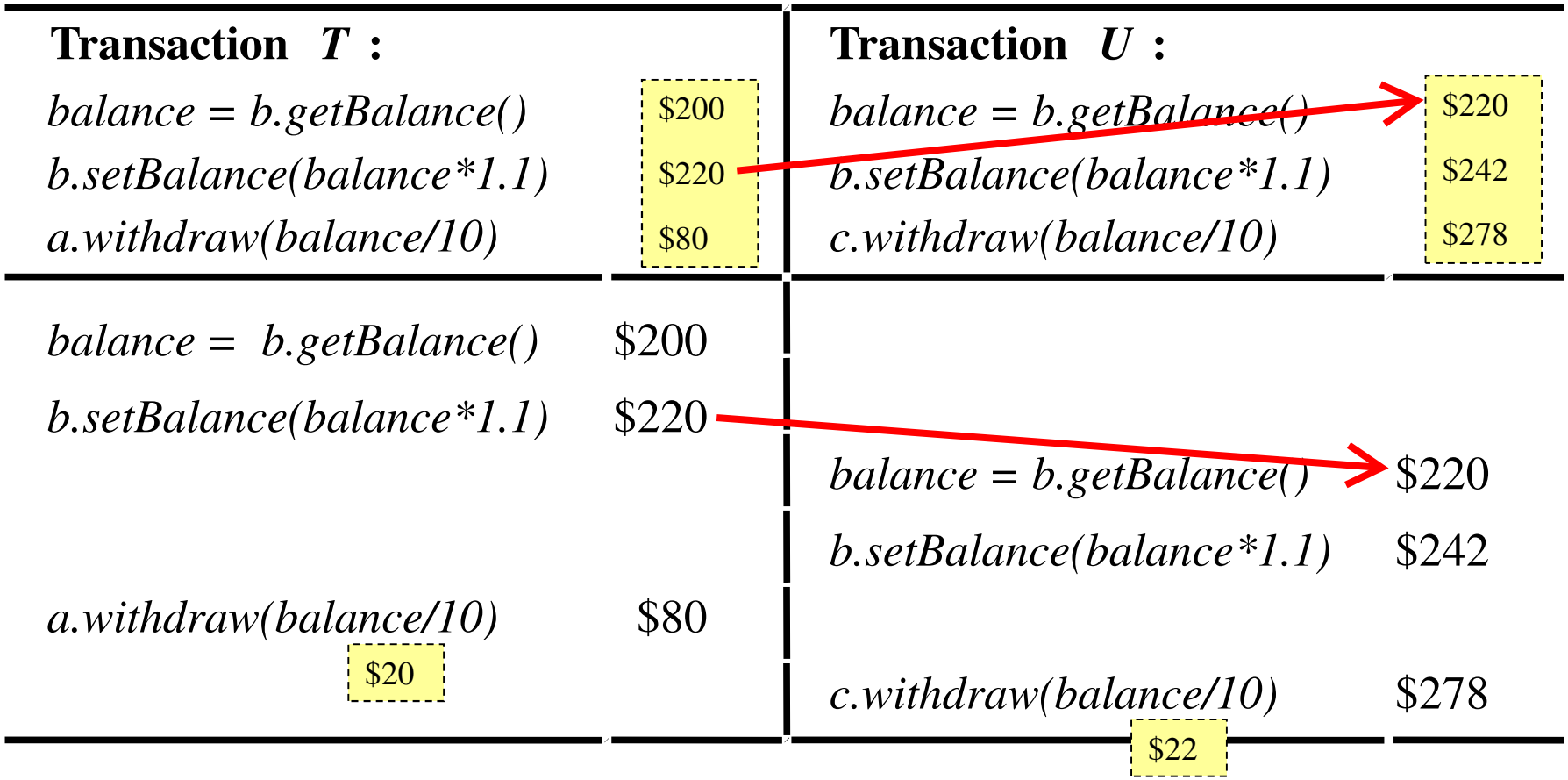
Concurrent execution: a's balance + b's balance = 300; W's retrievals of a's and b's balances are inconsistent

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 *serially equivalent interleaving*.
  - Read operations return the same values.
  - Instance variables of the objects have the same values at the end.
- Serial equivalence prevents lost update and inconsistent retrieval.
- Concurrency control algorithms (we will later see) will control concurrent transactions so that they interleave in a way that is equivalent to serial execution.

A serially equivalent interleaving for the lost update problem

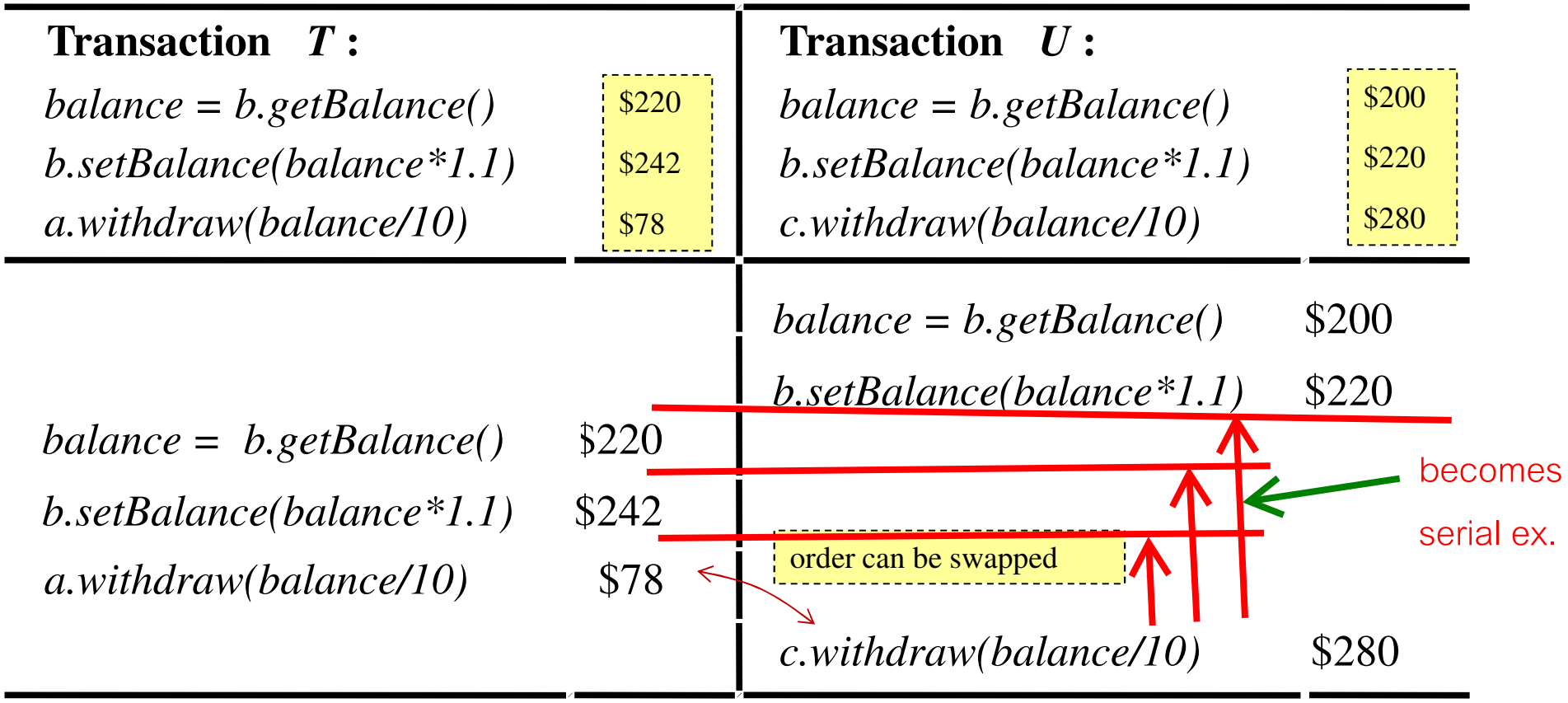
initial balances: a=100, b=200, c=300



The same effect as the serial execution of T before U

Another serially equivalent interleaving for the lost update problem

initial balances: a=100, b=200, c=300



The same effect as the serial execution of U before T

A serially equivalent interleaving for the inconsistent retrieval problem

initial balances: a=200, b=200

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

The same effect as the serial execution of V before W

V	W	
	total=b.getBalance()	$200
a.withdraw(100)	$100	
	total=total+a.getBalance()	$300
	total=total+c.getBalance()	

Another serially equivalent interleaving for the inconsistent retrieval problem

initial balances: a=200, b=200

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
		total = a.getBalance() (old value)	$200
a.withdraw(100)	$100		
		total = total+b.getBalance() (old value)	$400
b.deposit(100)	$300		
		total = total+c.getBalance()	
		...	

The same effect as the serial execution of W before V

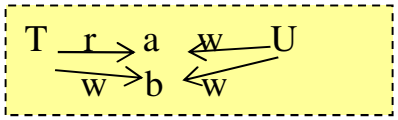
out-of-date ≠ incorrect -> if b.deposit(100) is removed, a+b=400 vs. a+b=300 (first 2 lines are swapped)

Conflicting Operations (1)

- Conflicting operations are those whose combined effect depends on the order of their execution.

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

- For two transactions to be serially equivalent, all pairs of conflicting operations are executed in the same order at all of the objects they both access.



Conflicting Operations (2)

- The ordering below is not serially equivalent as T accesses i before U but U accesses j before T.

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

- Serially equivalent ordering requires either
  - T accesses i before U and T accesses j before U, or
  - U accesses i before T and U accesses j before T.

Effect of abort on concurrent transactions

- An aborting transaction may affect another concurrent transaction even though their interleaving execution is serially equivalent.
  - Dirty read
  - Premature write

Dirty Read

already T before U

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

At T's abort, a's balance is restored to 100.

U has observed a value that never exists, and commit cannot be undone.

U's

As U is in danger of having a dirty read, its commit must be delayed until T has committed.

But if T aborts, U has to aborts too (and also other transactions that have seen the effects of T and U); this is called *cascading abort*.

U

To avoid cascading abort, transactions can read only from the objects written by committed transactions.

Premature Write

already T before U

Transaction *T*: intially a=100		**Transaction *U*:** intially a=105		
a.setBalance(105)		*a.setBalance(110)*		
	$100			
a.setBalance(105)	$105			
		a.setBalance(110)	$110	should delay

Suppose U commits and then T aborts, a's balance is restored to 100.

Suppose T aborts and then U aborts, a's balance is restored to 105.

Either case, we get the wrong balance.

Write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.

Strict execution – Both read and write on an object are delayed until all transactions that previously wrote that object have committed or aborted.

Concurrency Control Algorithms

- Concurrency control algorithms schedule concurrent transactions so that their effects are serially equivalent.
- A server can achieve serial equivalence by serializing access to objects by
  - Locking
  - Optimistic concurrency control
  - Timestamp ordering

Acc a has committed version = 200
T has tentative version of a = 200
U has tentative version of a = 200

- Concurrency is controlled with the use of private *tentative versions* of the objects that a transaction has changed; they are in volatile memory.
  - Any updates can then be removed easily if the transaction is aborted; tentative versions are deleted.
  - Tentative versions are transferred to the objects in permanent storage when the transaction is committed.

committed version

Locking

- The server attempts to lock any object that is to be used by any operation of the client's transaction.
  - If the object is already locked by other transaction, the client must wait until it is unlocked.
- Locking serializes access to the same objects according to the order of arrival of the operations at the objects.
- Granularity – portion of the objects to which access must be serialized – should be as small as possible for better concurrency.

Strict two-phase locking

- Locking has two non-overlapping phases : growing phase and shrinking phase; this is called *two-phase locking*.
- Any locks are held until the transaction commits or aborts; locking is *strict*.

Many readers/single writer locking scheme

- Since two reads are not conflicting operations, read locks are shared locks.
  - This scheme provides better concurrency than exclusive locks.

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

- Read lock set previously on an object has to be *promoted* to write lock first in order to write the object.

Strict two-phase locking using many readers/single writer scheme

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

lock promotion

Lock that is associated with each shared object

```
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();     //thread is put in ready queue when notified; recheck loop when running
                        }catch ( InterruptedException e){/*...*/ }
                }
                if(holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if(/*another transaction holds the lock, share it*/ ) ){    //someone holds read lock
                        if(/* this transaction not a holder*/) holders.addElement(trans);   //want read lock too
                } else if (/* this transaction is a holder but needs a more exclusive lock*/)    //already hold read lock
                                lockType.promote();
                }
        }

        public synchronized void release(TransID trans ){
                        holders.removeElement(trans);          // remove this holder
                        // set locktype to none
                        notifyAll();            //notify all waiters on this lock object
                }
        }
```

such as Account

if it is the only holder

Lock manager of the server

```
public class LockManager {
   private Hashtable theLocks;
                                        Account a
   public  void setLock(Object object, TransID trans,  LockType lockType){
           Lock foundLock;
           synchronized(this){
                   // find the lock associated with the object in the hashtable
                   // 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 lock entries of the trans
   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);
      }
   }
}
```

setLock() is called when read or write operation is about to be applied on a shared object.

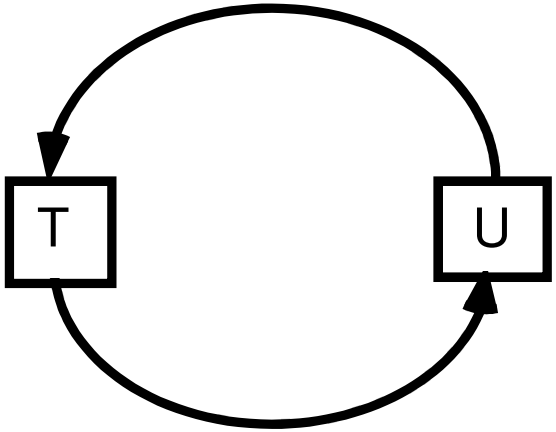
unLock() is called by commit or abort operation of the transaction coordinator.

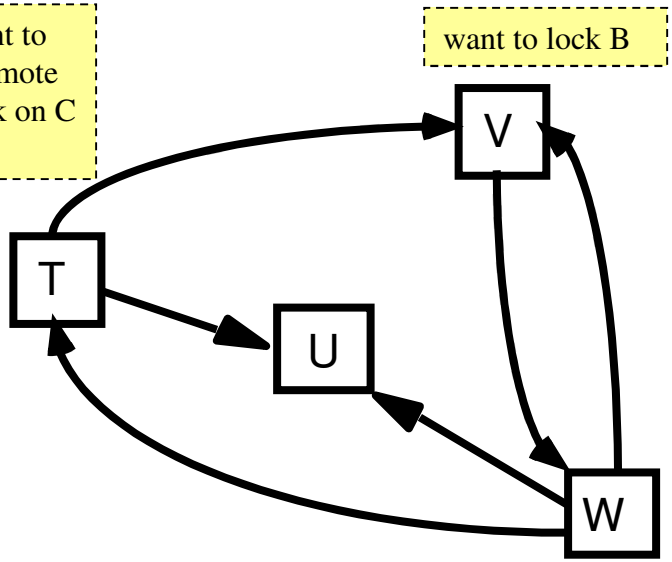
Deadlocks

- Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.

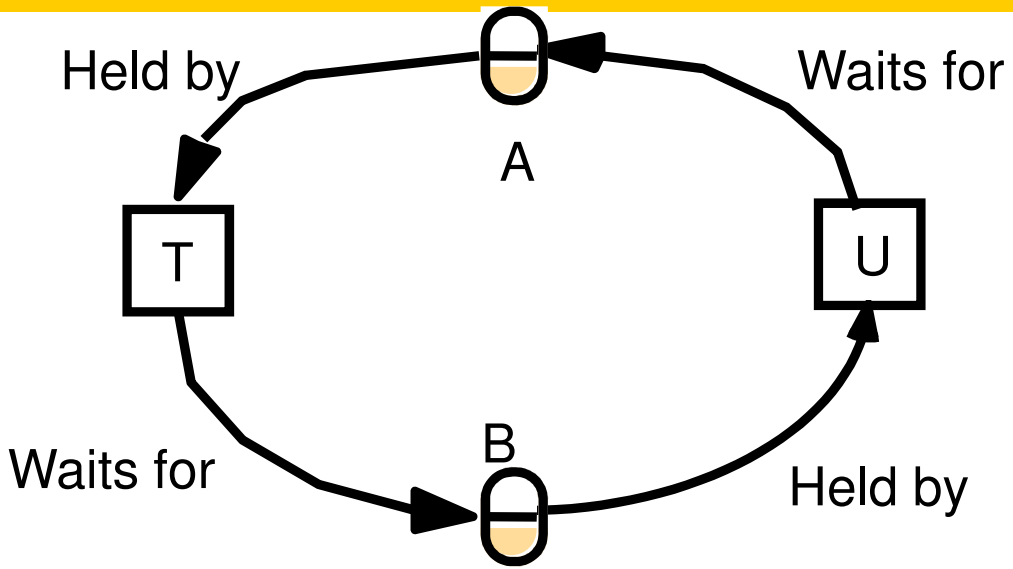
Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)	waits for *U*'s		
•••	lock on *B*	*a.withdraw(200);*	waits for *T*'s
		•••	lock on *A*
•••		•••	
•••		•••	

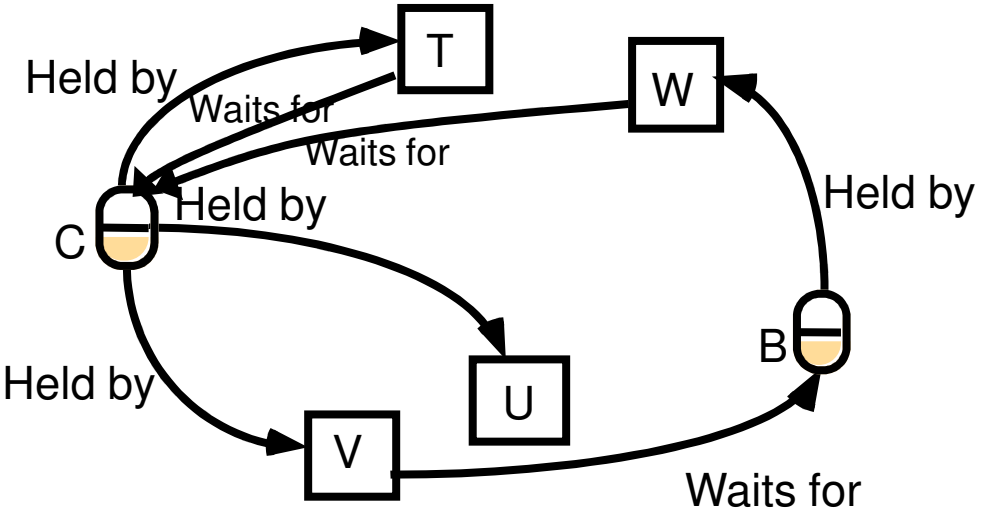
Wait-for graph





Resource Allocation Graph





Deadlock prevention is not realistic

- Lock all objects used by the transaction when it starts
  - This results in premature locking and a reduction in concurrency.
  - It is sometimes impossible to predict at the start which objects will be used, e.g. in interactive applications.
- Request locks on objects in a predefined order
  - This can also result in premature locking.

such as meeting scheduler

Deadlock detection

- Lock manager maintains a wait-for graph on blocking by *setLock()* and on *unLock().*
- Cycle may be checked each time an edge is added, or less frequently to avoid overhead.
- When a cycle is detected, choose a transaction to be aborted and remove the node and edges involving it.
  - Choose the oldest one or the one in the most cycles.

Lock Timeouts

- Each lock is given a limited time in which it is invulnerable.
- If no other transaction is waiting for the locked object, the vulnerable lock remains with the holder transaction.
- Otherwise, the lock is broken.
  - The waiting transaction acquires the lock and resumes.
  - The previous holder transaction is normally aborted.
- Problems:
  - Transactions are aborted even though the transactions that are waiting do not cause deadlock.
  - Lock timeouts will occur often in overloaded system, and long transactions are penalized.
  - It is hard to select a suitable length for a timeout.

Lock timeout resolves deadlock

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 locks *A*
			unlock *A*, *B*

If here is c.withdraw(100) which waits for V's lock on C, then U->T->V. Unnecessary abort.

Optimistic Concurrency Control

- Optimistic approach argues that
  - The likelihood of two transactions accessing the same object at the same time is low.
  - Locking causes deadlock.
  - Strict two-phase locking reduces the potential of concurrency.
- In this approach, a transaction proceeds without restriction until *closeTransaction().*
- It is then validated whether it has come into conflict with other transactions.
  - If so, a transaction is aborted.
- Optimistic approach serializes access to the same objects according to the order of validation of the transactions.

Each transaction has three phases

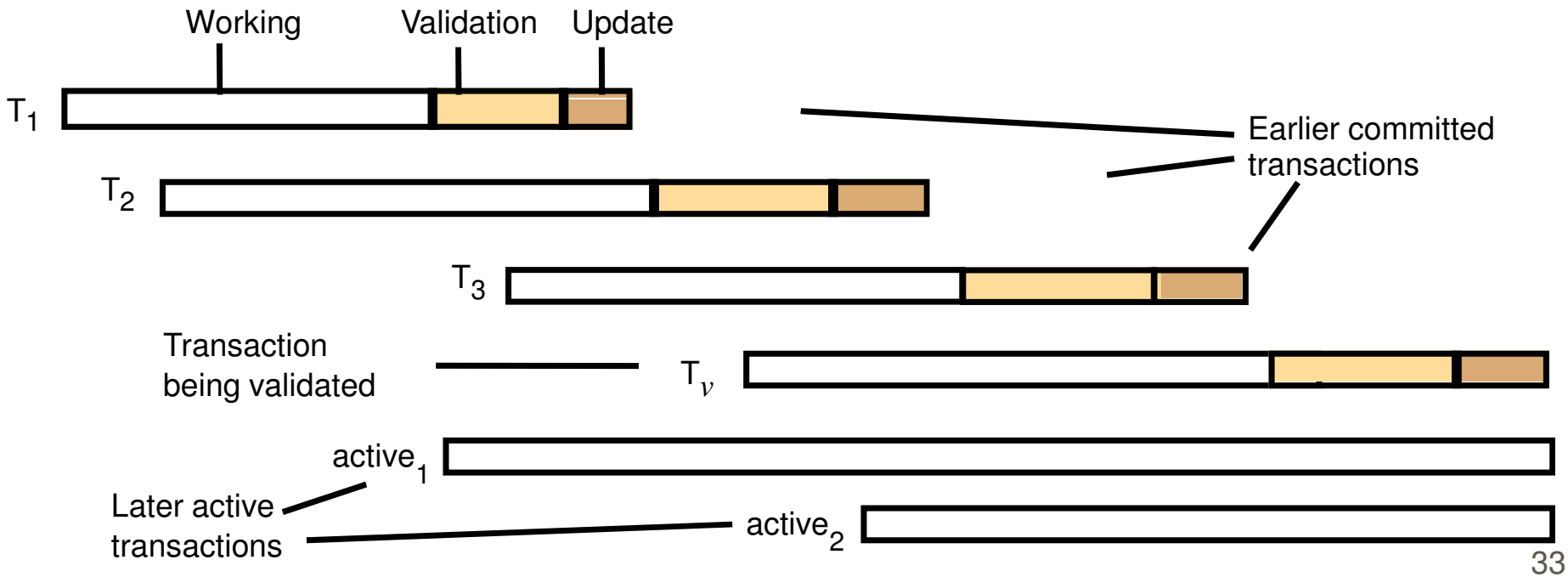
- Working phase
  - Write and read perform immediately.
  - Write on private tentative versions
  - Read from committed version; no dirty read
  - Read set and write set containing objects read and written respectively by the transaction are maintained.

committed version of Acc a = 200
T: tentative of a = 200, write 300, closeTrans()
U: tentative of a = 200, write 250

- Validation phase
  - On *closeTransaction(),* the transaction is validated if its operations on objects conflict with operations of other transactions on the same objects.
  - If successful, the transaction can commit and go to update phase.
  - If failed, the transaction or one that it conflicts with is aborted.
- Update phase
  - All of the changes recorded in the tentative versions are recorded in permanent storage.

Validation of Transaction (1)

- A transaction is validated against *overlapping* transactions – those not yet committed at the time this transaction starts.
- Transaction will be assigned a transaction number (or validation number); the number is in ascending order.
  - If validation fails, this transaction number can be reassigned to the next validating transaction.



Validation of Transaction (2)

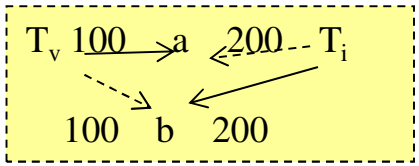
- Validation is still based on read-write conflict rules.

validating trans | overlapping trans

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

If these rules are satisfied, we can commit Tv.

write and commit

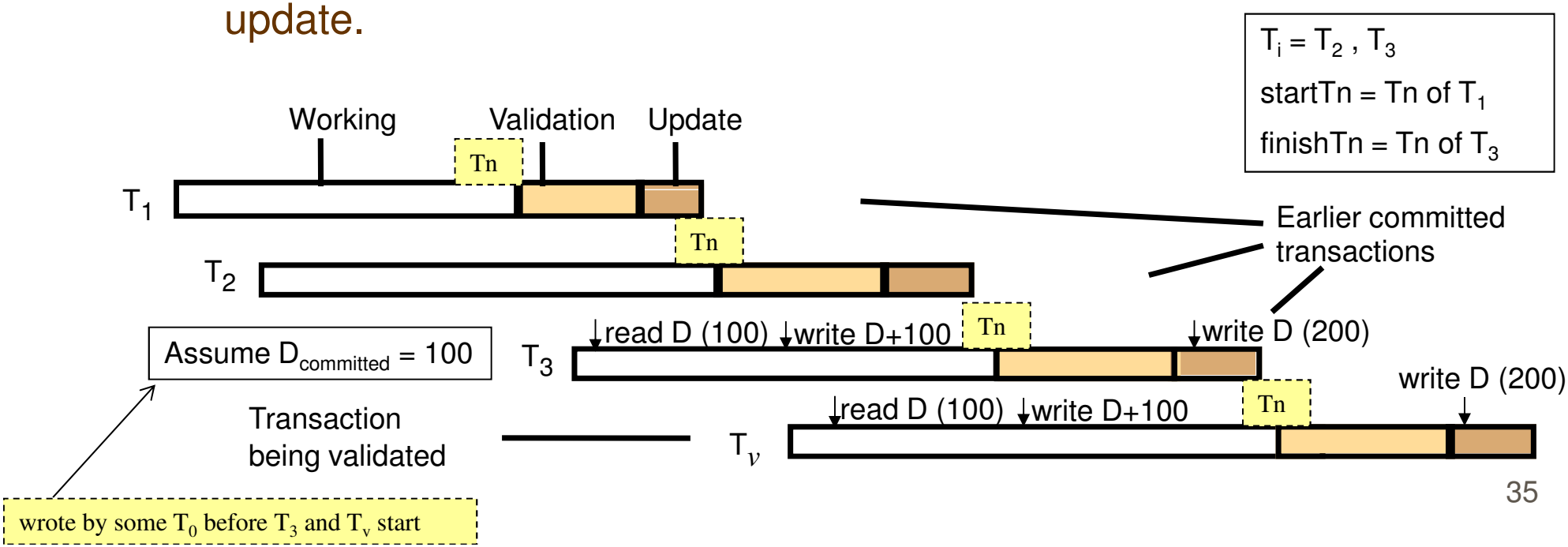


Finally a = 200, b = 100 which is wrong

- Validation phase and update phase take very short time, so their execution can be made atomic (implemented as a critical section).
  - Only one transaction can be in validation and update phases at a time.
  - When no two transactions may overlap in update phase, rule 3 is satisfied.

Backward validation against committed overlapping transactions (1)

- Rule 1 (T_v's write vs. T_i's read) is satisfied because read of earlier committed transactions were performed before T_v enters validation (and possible writes).
- Rule 2 (T_v's read vs. T_i's write) has to be checked.
  - If T_v reads objects that were written already by T_i, we may have lost update.



Backward validation (2)

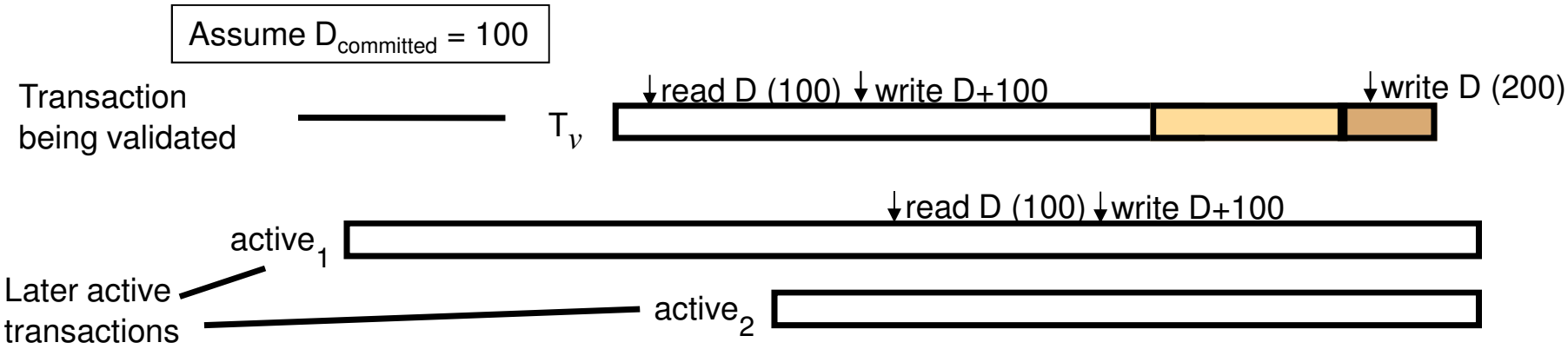
```
boolean valid = true;
for (int Ti = startTn+1; Ti ≤ finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```

- *startTn* is the biggest transaction number assigned to some other committed transaction when T_v started its working phase.
- *finishTn* is the biggest transaction number assigned to some other committed transaction when T_v started its validation phase.
- In the previous figure, *StartTn* + 1 = T_2 and *finishTn* = T_3.
- The only way to resolve a conflict is to abort T_v.
- We must keep the write set and transaction number of old committed transactions until there are no unvalidated overlapping transactions with which they might conflict.

other active transactions at the time this transaction finishes update phase.

Forward validation against active overlapping transactions (1)

- Rule 2 (T_v's read vs. T_i's write) is satisfied because possible writes of later active transactions will not be performed until after T_v has completed.
- Rule 1 (T_v's write vs. T_i's read) has to be checked.
  - If T_i has read objects that are about to be written by T_v, we may have lost update.



Forward validation (2)

```
boolean valid = true;
for (int T_id = active_1; T_id ≤ active_N; T_id++){
        if (write set of T_v intersects read set of T_id)  valid = false;
}
```

- If validation fails,
  - Abort T_v
  - Abort all conflicting active transactions and commit T_v
  - Defer validation until later time
    - Conflicting active transactions may themselves abort, but
    - Further conflicting transactions may start as well.

Comparison between backward and forward validation

- Choice of transaction to abort
  - Forward validation is more flexible, whereas backward validation allows only one choice (the one being validated).
- In general, read sets are larger 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
    - needs to allow for new transactions starting during validation
- Both may suffer from starvation.
  - After a transaction is aborted, the client must restart it; there is no guarantee it will ever succeed.
  - The transaction that has been aborted several times should be given exclusive access.

Comparison between locking and optimistic approach

- Locking
  - Pessimistic approach (detect conflicts as they arise)
  - When conflicts are detected, wait but can get a deadlock which needs to abort transaction later.
- Optimistic approach
  - All transactions proceed, but may need to abort at the end.
  - Efficient operations when there are few conflicts, but aborts lead to repeating work.