13.4 Locks

- Transactions must be scheduled so that their effect on shared objects is serially equivalent
- A server can achieve serial equivalence by serialising access to objects, e.g. by the use of locks
- 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.
- to ensure (b), a transaction is not allowed any new locks after it has released a lock
  - **Two-phase locking** - has a 'growing' and a 'shrinking' phase

Transactions T and U with exclusive locks (Fig. 13.14, same as 13.7)

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*

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

when *T* is about to use *B*, it is locked for *T*

when *T* commits, it unlocks *B*

U can now continue

- initially the balances of A, B and C unlocked
- the use of the lock on B effectively serialises access to B

Strict two-phase locking

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

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.

to enforce 1, a request for a write lock is delayed by the presence of a read lock belonging to another transaction

to enforce 2, a request for a read lock or write lock is delayed by the presence of a write lock belonging to another transaction

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

Figure 13.15

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
- when another transaction requires a read lock it will be delayed until any current transaction has completed
- 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
  - The result will be more permissive than the previous one and may allow executions by other transactions that are inconsistent with serial equivalence

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.

Figure 13.16

- The sever 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 lock manager holds a set of locks, for example in a hash table
- each lock is an instance of the class *Lock* (Fig 13.17) 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

Figure 13.17 *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 conflicing 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();
                }
        }
```

Continues on next slide

Figure 13.17 continued

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

Figure 13.18 *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 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

T accesses A → B
U accesses B → A

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	*a.withdraw(200);*	waits for *T*'s
	lock on *B*	• • •	lock on *A*
• • •		• • •	
• • •		• • •	

Figure 13.19

- The *deposit* and *withdraw* methods are atomic. Although they read as well as write, they acquire write locks.
- The lock manager must be designed to deal with 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
- In a wait-for graph the nodes represent transactions and the edges represent wait-for relationships between transactions

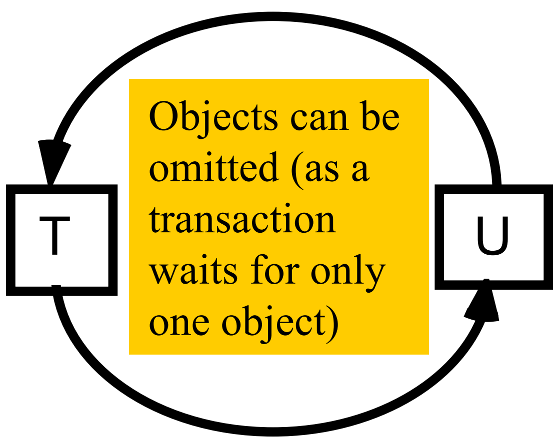
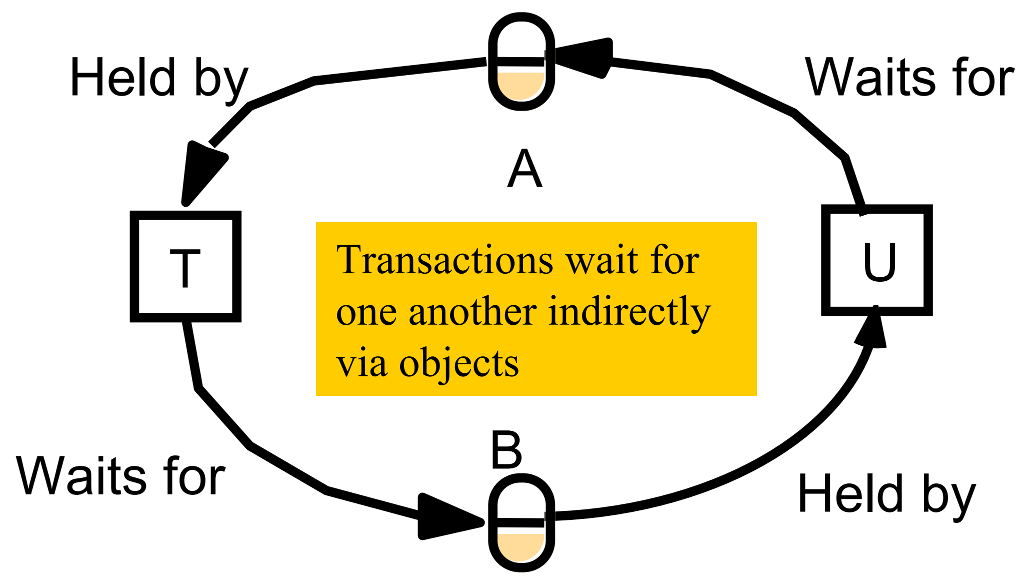


Figure 13.20



A cycle in a wait-for graph

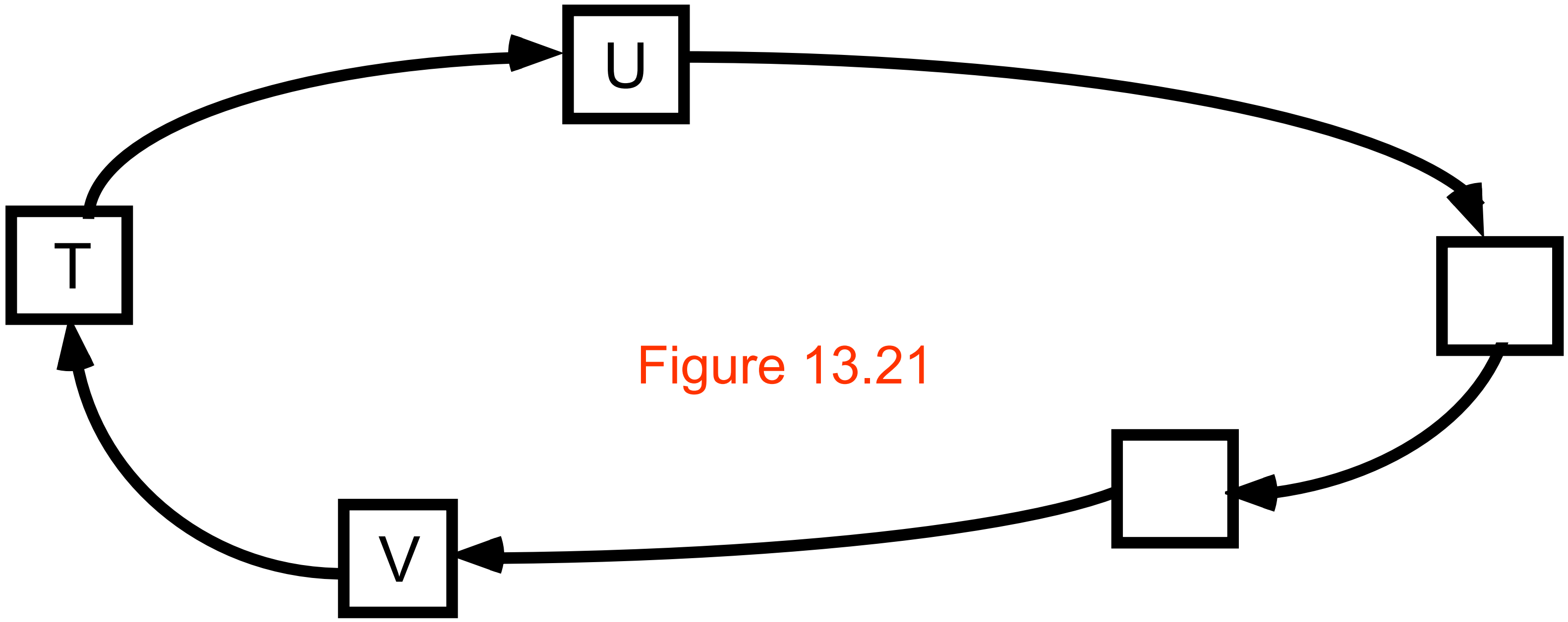


Figure 13.21

- Suppose a wait-for graph contains a cycle T …→ U → … → V → 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

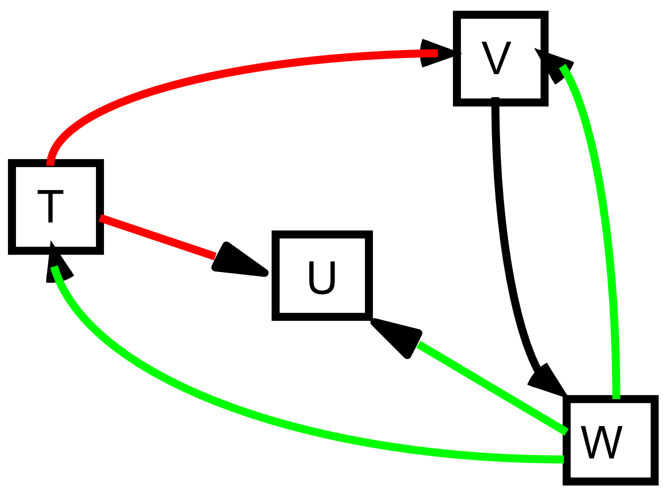
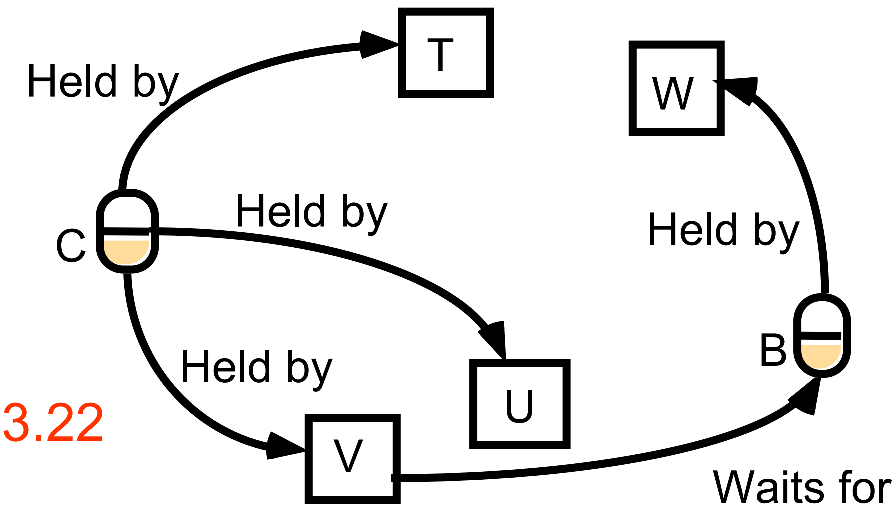


Figure 13.22



- *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

Deadlock prevention is unrealistic

- e.g. lock all of the objects used by a transaction when it starts
  - unnecessarily restricts access to shared resources.
  - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- Deadlock can also be prevented by requesting locks on objects in a predefined order
  - but this can result in premature locking and a reduction in concurrency

Deadlock detection

- by finding cycles in the wait-for graph
  - after detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  - the software for deadlock detection can be part of the lock manager
  - it holds a representation of the wait-for graph so that 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
  - when a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
  - it is hard to choose a victim - e.g. choose the oldest or the one in the most cycles

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

- problems with lock timeouts
  - locks may be broken when there is no deadlock
  - if the system is overloaded, lock timeouts will happen more often and long transactions will be penalised
  - it is hard to select a suitable length for a timeout

13.5 Optimistic concurrency control

- the scheme is called optimistic because the likelihood of two transactions conflicting is low
- 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
- each transaction has three phases

three phases

Working phase

- the transaction uses a tentative version of the objects it accesses (dirty reads can't occur as we read from a committed version or a copy of it)
- 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.

13.6 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

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

Comparison of methods for concurrency control

- 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
- 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
- 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.