Concurrency Control Techniques

406.426 Design & Analysis of Database Systems

Jonghun Park

jonghun@snu.ac.kr

Dept. of Industrial Engineering

Seoul National University

lock

- a variable associated with a data item that describes the **status** of the item w.r.t. **possible operations** that can be applied to it
- **one lock for each data item** in the DB
- used as a means of **synchronizing the access** by concurrent transactions to the DB items
- types
  - binary lock
  - shared/exclusive (read/write) lock
  - certify lock

binary locks

- a binary lock can have two states: locked and unlocked
  - *X* is **locked** (i.e., value = 1): item *X* cannot be accessed by a DB operation that requests the item
  - *X* is **unlocked** (i.e., value = 0): the item can be accessed when requested
  - LOCK(*X*): the current value of the lock associated with item *X*
- operations
  - lock_item
    - a transaction requests access to an item *X* by first issuing a lock_item(*X*) operation
    - if LOCK(X) = 1, the transaction is forced to wait
    - if LOCK(X) = 0, it is set to 1, and the transaction is allowed to access *X*
  - unlock_item
    - when the transaction is through using the item, it issues an unlock_item(*X*) operation so that *X* may be accessed by other transactions
- binary lock enforces **mutual exclusion** on the data item

lock_item (*X*):

```
B: if LOCK (X)=0  (* item is unlocked *)
   then LOCK (X)←1  (* lock the item *)
   else begin
     wait  (until lock (X)=0  and
       the lock manager wakes up the transaction);
     go to B
     end;
```

unlock_item (*X*):

```
LOCK (X)←0;  (* unlock the item *)
if any transactions are waiting
  then wakeup one of the waiting transactions;
```

lock manager

- each lock can be a record with 3 fields: <**data item name**, **LOCK**, **locking transaction**> plus a **queue** for transactions that are waiting to access the item
- lock manager needs to maintain only these records for the items that are currently locked in a **lock table**, that can be **organized as a hash file**
- rules
  - a transaction *T* must issue the operation lock_item(*X*) **before** any read_item(*X*) or write_item(*X*) operations are performed in *T*
  - a transaction *T* must issue the operation unlock_item(*X*) **after** all read_item(*X*) and write_item(*X*) operations are completed in *T*
  - a transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*
  - a transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*

shared/exclusive locks

- problem with the binary locks
  - at most one transaction can hold a lock on a data item
  - what if all transactions only need to **read** the data item?
- 3 locking operations: read_lock(*X*), write_lock(*X*), and unlock(*X*)
- 3 possible states
  - read-locked: other transactions are allowed to read the item
  - write-locked: only a single transaction exclusively holds the lock on the item
  - unlocked

```
read_lock (X):
  B: if LOCK (X)="unlocked"
     then begin LOCK (X)← "read-locked";
                no_of_reads(X)← 1
          end
     else if LOCK(X)="read-locked"
           then no_of_reads(X)← no_of _reads(X) + 1
           else begin wait (until LOCK (X)="unlocked" and
                     the lock manager wakes up the transaction);
                go to B
                end;
```

```
write_lock (X):
  B: if LOCK (X)="unlocked"
     then LOCK (X)← "write-locked"
     else begin
          wait (until LOCK(X)="unlocked" and
            the lock manager wakes up the transaction);
          go to B
          end;
```

```
unlock_item (X):
  if LOCK (X)="write-locked"
     then begin LOCK (X)← "unlocked;"
          wakeup one of the waiting transactions, if any
          end
     else if LOCK(X)="read-locked"
           then begin
                no_of_reads(X)← no_of_reads(X) – 1;
                if no_of_reads(X)=0
                  then begin LOCK (X)="unlocked";
                       wakeup one of the waiting transactions, if any
                       end
                end;
```

lock manager for shared/exclusive locks

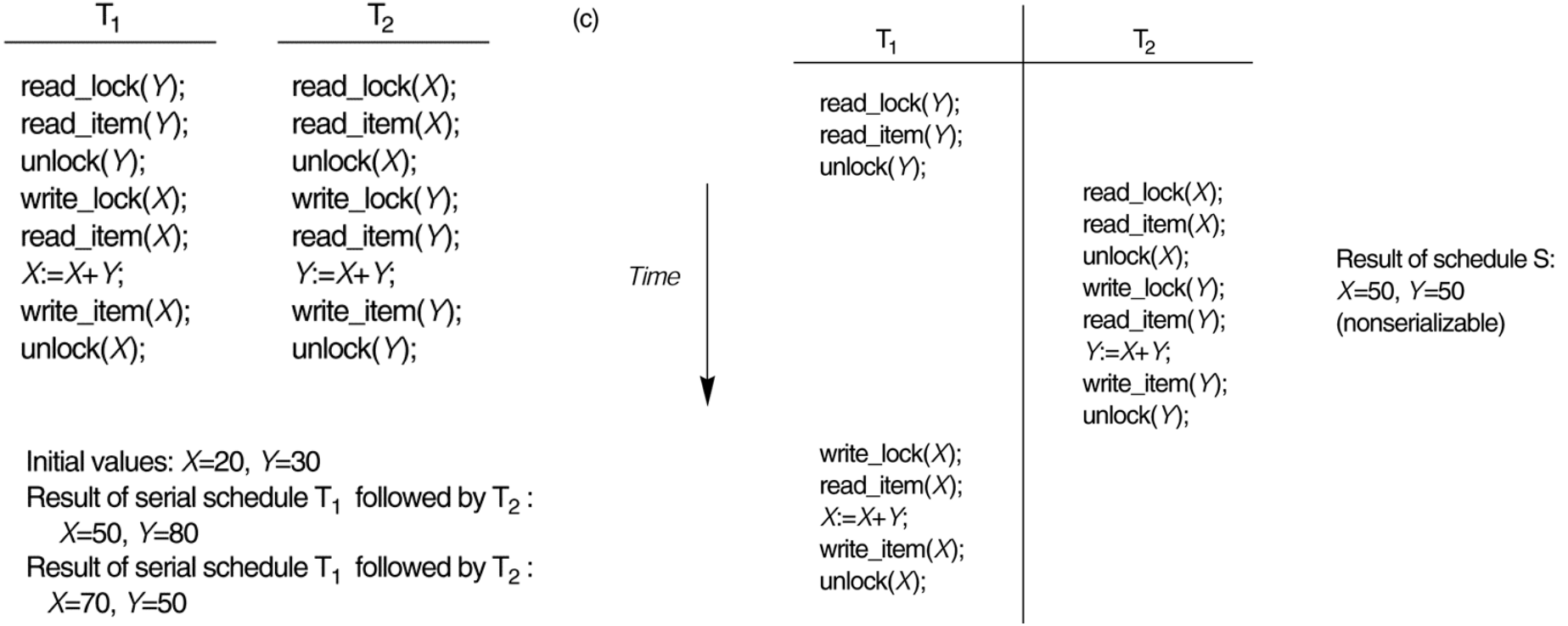
- each record in the lock table will have 4 fields
  - <data item name, LOCK, no_of_reads, locking_transactions>
  - LOCK: read-locked or write-locked
- rules
  - a transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*
  - a transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in T
  - a transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in T
  - a transaction *T* will not issue a read_lock(*X*) operation if it already holds a read lock or a write lock on item *X*
  - a transaction *T* will not issue a write_lock(*X*) operation if it already holds a read lock or a write lock on item *X*
  - a transaction *T* will not issue a unlock(*X*) operation unless it already holds a read lock or a write lock on item *X*

conversion of locks

- relaxes the 4th and 5th rules
- a transaction that already holds a lock on *X* may be allowed to convert the lock from one locked state to another
  - upgrade: read_lock(*X*) -> write_lock(*X*)
  - downgrade: write_lock(*X*) -> read_lock(*X*)

serializability and the locking

- using binary locks and read/write locks in transactions does not guarantee serializability of schedules



2PL (Two Phase Locking)

- a transaction is said to follow the two-phase locking protocol if **all** locking operations (read_lock, write_lock) **precede** the **first unlock operation** in the **transaction**
  - **growing** phase: new locks on items can be acquired but none can be released
  - **shrinking** phase: existing locks can be released but no new locks can be acquired
  - if lock conversion is allowed, then upgrading of locks must be done during the growing phase, and downgrading locks must be done during the shrinking phase
- if every transaction in a schedule follows the 2PL, the schedule is serializable => obviates the need to test for serializability

T_1	T_2
read_lock(*Y*);	read_lock(*X*);
read_item(*Y*);	read_item(*X*);
unlock(*Y*);	unlock(*X*);
write_lock(*X*);	write_lock(*Y*);
read_item(*X*);	read_item(*Y*);
X:=*X*+*Y*;	*Y*:=*X*+*Y*;
write_item(*X*);	write_item(*Y*);
unlock(*X*);	unlock(*Y*);

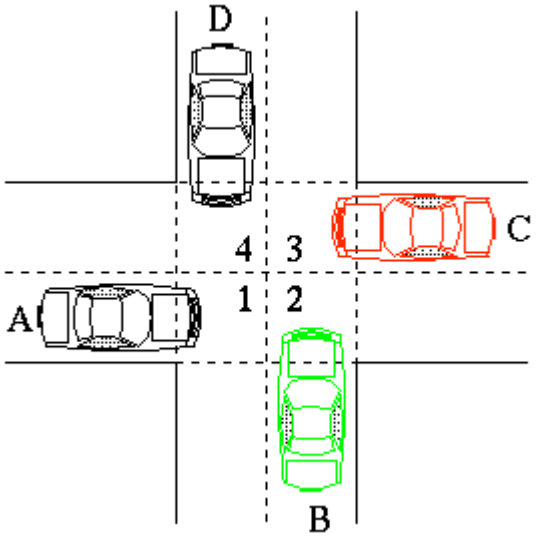
T_1'	T_2'
read_lock (*Y*);	read_lock (*X*);
read_item (*Y*);	read_item (*X*);
write_lock (*X*);	write_lock (*Y*);
unlock (*Y*);	unlock (*X*);
read_item (*X*);	read_item (*Y*);
X:=*X*+*Y*;	*Y*:=*X*+*Y*;
write_item (*X*);	write_item (*Y*);
unlock (*X*);	unlock (*Y*);

limitations of 2PL

- 2PL limits the amount of concurrency that can occur in a schedule
  - a transaction *T* may not be able to release an item *X* after it is through using it if *T* must lock an additional item *Y* later
  - *X* must remain locked by *T* until all items that the transaction needs to read or write have been locked
  - meanwhile, another transaction seeking to access *X* may be forced to wait, even though *T* is done with *X*
- variations of 2PL
  - conservative 2PL
  - strict 2PL
  - rigorous 2PL

conservative 2PL

- requires a transaction to lock **all the items** it accesses before the transaction begins execution by predeclaring its read-set (data items it reads) and write-set (data items it writes)
- if any of the predeclared items needed cannot be locked, the transaction does not lock any item -> it waits until all the items are available for locking
- once the transaction starts, it is in its shrinking phase
- deadlock-free protocol
- difficult to use in practice



strict 2PL

- most popular variation of 2PL
- guarantees **strict schedules**
- a transaction *T* does not release any of its **write locks** until after it commits or aborts -> no other transaction can read or write an item that is written by *T* unless *T* has commited
- deadlock prone

rigorous 2PL

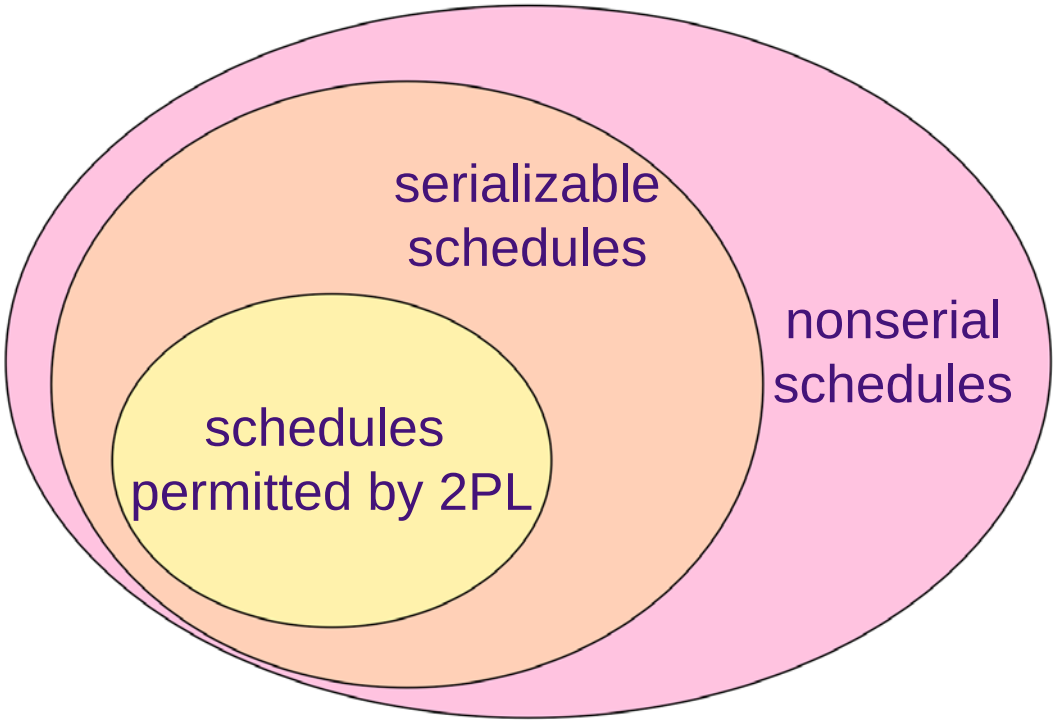
- a more restrictive variation of strict 2PL
- guarantees strict schedules
- a transaction *T* does not release any of its locks (**write or read**) until after it commits or aborts
- easier to implement than strict 2PL
- transaction is in its growing phase until it ends

role of concurrency control subsystem

- assuming that the system is to enforce the strict 2PL protocol,
- whenever transaction *T* issues a read_item(*X*), the system calls the read_lock(*X*) operation on behalf of *T*
  - if the state of LOCK(*X*) is write_locked by some other transaction *T'*, the system places *T* on the waiting queue for *X*
  - o.w. it grants the read_lock(*X*) request and permits the read_item(*X*) operation of *T* to execute
- if *T* issues a write_item(*X*), the system calls the write_lock(*X*) operation on behalf of *T*
  - if the state of LOCK(*X*) is write_locked or read_locked by some other transaction *T'*, the system places *T* on the waiting queue for *X*
  - if the state of LOCK(*X*) is read_locked and *T* itself is the only transaction holding the read lock on *X*, the system upgrades the lock to write_locked and permits the write_item(*X*) operation by *T*
  - if the state of LOCK(*X*) is unlocked, the system grants the write_lock(*X*) requests and permits the write_item(*X*) operation to execute

problems with 2PL

- does not permit all possible serializable schedules
- use of locks can cause deadlock and starvation



deadlocks

- deadlock occurs when each transaction *T* in a set of two or more transactions is waiting for some item that is locked by some other transaction *T'* in the set

T_1'	T_2'
read_lock(*Y*);	
read_item(*Y*);	
	read_lock(*X*);
	read_item(*X*);
write_lock(*X*);	
	write_lock(*Y*);

deadlock prevention protocols

- **advance reservation**
  - deadlock prevention protocol used in conservative 2PL
  - requires that every transaction **lock all the items** it needs in advance
  - if any of the items cannot be obtained, none of the items are locked -> the transaction waits and then tries again to lock all the items it needs
  - obviously limits concurrency
  - livelock prone
- **resource ordering**
  - order all the items in the DB and make sure that a transaction requiring several items will **lock them according to the order**
  - requires the system be aware of the chosen order of the items, which is **not practical**

other deadlock prevention protocols: timestamp

- transaction timestamp, TS(T)
  - a unique identifier assigned to each transaction (e.g., when the transaction is started)
  - if T_1 starts before T_2, then TS(T_1) < TS(T_2)
    - **older** transaction has the **smaller** timestamp value

- 2 schemes for preventing deadlock when **T_i tries to lock an item X** but is not able to because **X is locked by T_j**
  - **wait-die**: if TS(T_i) < TS(T_j), then T_i is allowed to wait; o.w. abort T_i and restart T_i later with the same timestamp
  - **wound-wait**: if TS(T_i) < TS(T_j), then abort T_j and restart T_j later with the same timestamp; o.w. T_i is allowed to wait
- both schemes end up **aborting the younger** of the two transactions that may be involved in a deadlock
- deadlock-free since,
  - in wait-die, transactions only wait on younger transactions => no cycle is created
  - in wound-wait, transactions only wait on older transactions => no cycle is created

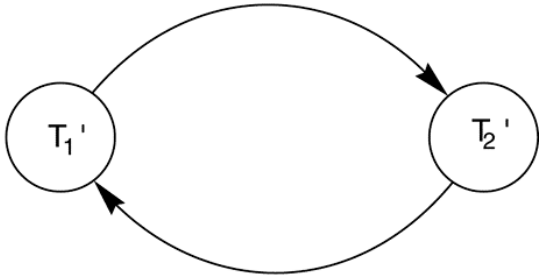
other deadlock prevention protocols: NW & CW

- no waiting (NW) algorithm
  - if a transaction is unable to obtain a lock, it is **immediately aborted** and then restarted after a certain time delay without checking whether a deadlock will actually occur or not
  - livelock prone
- cautious waiting (CS) algorithm
  - tries to reduce the number of needless aborts / restarts
  - suppose that T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock
    - if T_j is not blocked (i.e., not waiting for some other locked item), then T_i is blocked and allowed to wait; o.w. abort T_i

deadlock detection

- more **practical** approach
- system checks if a state of deadlock actually exists
- useful when the transactions are short and each transaction **locks only a few items**, or if the transaction **load is light**
- wait-for graph
  - one node is created in the wait-for graph for each transaction that is currently executing
  - whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge (T_i -> T_j) is created in the wait-for graph
  - when T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph
- a deadlock state **iff the wait-for graph has a cycle**

T_1'	T_2'
read_lock(*Y*);	
read_item(*Y*);	
	read_lock(*X*);
	read_item(*X*);
write_lock(*X*);	
	write_lock(*Y*);



issues in deadlock detection

- criteria on when the system should check for a deadlock
  - # of concurrently executing transactions
  - period of time several transactions have been waiting to lock items
- victim selection
  - if the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted
  - generally avoids selecting transactions that have been running for a long time and that have performed many updates
- timeouts
  - if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it
  - practical due to the low overhead and simplicity

starvation

- occurs when a transaction **cannot proceed for an indefinite period of time** while other transactions in the system continue normally
- may occur if the waiting scheme for locked items is **unfair**, giving priority to some transactions over others
- can also occur because of the **victim selection** if the algorithm selects the same as victim repeatedly, thus causing it to abort and never finish execution
- possible solutions
  - FCFS
  - prioritization, but increasing the priority of a transaction the longer it waits until it eventually gets the highest priority and proceeds