

COP 4710: Database Systems Spring 2004

-Day 24 – April 5, 2004 –
Transaction Processing - Concurrency Protocols

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
<http://www.cs.ucf.edu/courses/cop4710/spr2004>

School of Electrical Engineering and Computer Science
University of Central Florida



Types of Locks

- There are different types of locks that locking protocols may utilize.
- The most restrictive systems use only **exclusive-locks** (X-lock also called a binary lock).
- An exclusive lock permits the transaction which holds the lock exclusive access to the object of the lock.

If transaction T_x holds an X-lock on object A then no distinct transaction T_y can obtain an X-lock on object A until transaction T_x releases the X-lock on object A. T_y is blocked awaiting the X-lock on object A.

- The process of locking and un-locking objects must be indivisible operations within a critical section. There can be no interleaving of issuing and releasing locks.



X-Lock Protocol

Before any transaction T_x can read or write an object A, it must first acquire an X-lock on object A. If the request is granted T_x will proceed with execution. If the request is denied, T_x will be placed into a queue of transactions awaiting the X-lock on object A, until the lock can be granted. After T_x finishes with object A, it must release the X-lock.

- When the lock manager grants a transaction's request for a particular lock, the transaction is said to “hold the lock” on the object.
- Under the X-lock protocol a transaction must obtain, for every object required by the transaction, an X-lock on the object. This applies to both reading and writing operations.



Serializability Under X-Lock Protocol

Algorithm TestSerializabilityXLock

//input: a concurrent schedule S under X-lock protocol

//output: if S is serializable, then a serially equivalent schedule S' is produced, otherwise, no.

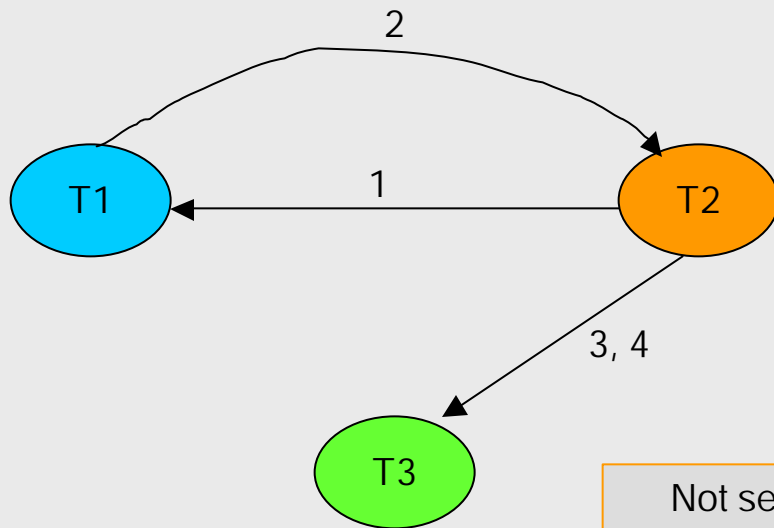
TestSerializabilityXLock(S)

1. let $S = (a_1, a_2, \dots, a_n)$ where "action" a_i is either $(T_x: \text{Xlock } A)$ or $(T_x: \text{Unlock } A)$
2. construct a precedence graph of n nodes where n is the number of distinct transactions in S .
3. proceed through S as follows:
 - if $a_r = (T_x: \text{Unlock } A)$ then look for the next action a_s of the form $(T_y: \text{Xlock } A)$. If one exists, draw an edge in the graph from T_x to T_y . The meaning of this edge is that in any serially equivalent schedule T_x must precede T_y .
4. if the graph constructed in step 3 contains a cycle, then S is not equivalent to any serial schedule (i.e., S is not serializable). If no cycle exists, then any topological sort of the graph will yield a serial schedule equivalent to S .



Example - X-Lock Protocol and Serializability

Let $S = [(T1: Xlock A), (T2: Xlock B), (T2: Xlock C), (T2: Unlock B),$
 $(T1: Xlock B)$, $(T1: Unlock A)$, $(T2: Xlock A)$, $(T2: Unlock C)$,
 $(T2: Unlock A)$, $(T3: Xlock A)$, $(T3: Xlock C)$, $(T1: Unlock B),$
 $(T3: Unlock C), (T3: Unlock A)]$



Edge #1: $(T2: Unlock B) \dots (T1: Xlock B)$
Edge #2: $(T1: Unlock A) \dots (T2: Xlock A)$
Edge #3: $(T2: Unlock C) \dots (T3: Xlock C)$
Edge #4: $(T2: Unlock A) \dots (T3: Xlock A)$

Not serializable, cycle exists



Problems with X-Lock Protocol

- The X-lock protocol is too restrictive.
- Several transactions that need only to read an object must all wait in turn to gain an X-lock on the object, which unnecessarily delays each of the transactions.
- One solution is to issue different types of locks, called **shared-locks** (S-locks or read-locks) and write-locks (X-locks).
- The lock manager can grant any number of shared locks to concurrent transactions that need only to read an object, so multiple reading is possible. Exclusive locks are issued to transactions needing to write an object.
- If an X-lock has been issued on an object to transaction T_x , then no other distinct transaction T_y can be granted either an S-lock or an X-lock until T_x releases the X-lock. If any transaction T_x holds an S-lock on an object, then no other distinct transaction T_y can be granted an X-lock on the object until all S-locks have been released.



Serializability Under X/S-Lock Protocol

Algorithm TestSerializabilityX/SLock

//input: a concurrent schedule S under X/S-lock protocol

//output: if S is serializable, then a serially equivalent schedule S' is produced, otherwise, no.

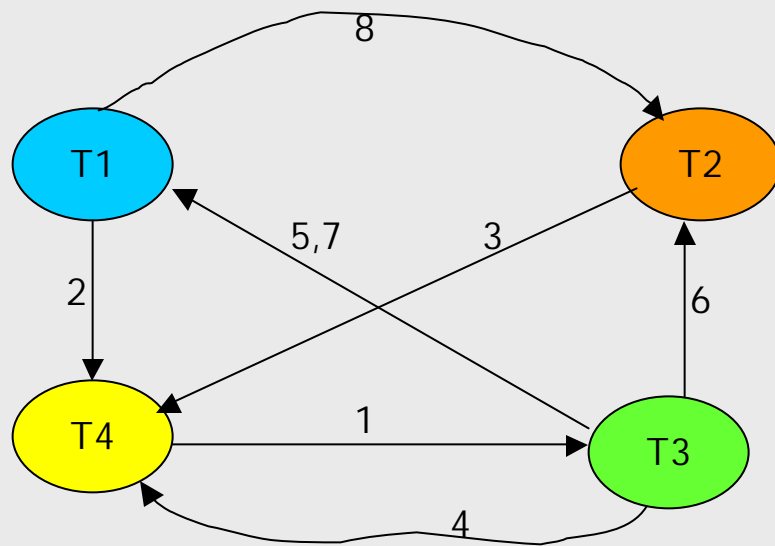
TestSerializabilityXLock(S)

1. let $S = (a_1, a_2, \dots, a_n)$ where "action" a_i is one of $(T_x: \text{Slock } A)$, $(T_x: \text{Xlock } A)$ or $(T_x: \text{Unlock } A)$.
2. construct a precedence graph of n nodes where n is the number of distinct transactions in S .
3. proceed through S as follows:
 - if $a_x = (T_x: \text{Slock } A)$ and a_y is the next action (if it exists) of the form $(T_y: \text{Xlock } A)$ then draw an edge from T_x to T_y .
 - if $a_x = (T_x: \text{Xlock } A)$ and there exists an action $a_z = (T_z: \text{Xlock } A)$ then draw an edge in the graph from T_x to T_z . Also, for each action a_y of the form $(T_y: \text{Slock } A)$ where a_y occurs after a_x ($T_x: \text{Unlock } A$) but before a_z ($T_z: \text{Xlock } A$) draw an edge from T_x to T_y . If a_z does not exist, then T_y is any transaction to perform $(T_y: \text{Slock } A)$ after $(T_x: \text{Unlock } A)$.
4. if the graph constructed in step 3 contains a cycle, then S is not equivalent to any serial schedule (i.e., S is not serializable). If no cycle exists, then any topological sort of the graph will yield a serial schedule equivalent to S .



Example – X/S-Lock Protocol and Serializability

Let $S = [(\underline{T3: Xlock\ A}), (\underline{T4: Slock\ B}), (\underline{\underline{T3: Unlock\ A}}), (\underline{\underline{T1: Slock\ A}}),$
 $(T4: Unlock\ B), (\underline{T3: Xlock\ B}), (\underline{T2: Slock\ A}), (\underline{\underline{T3: Unlock\ B}}),$
 $(\underline{T1: Xlock\ B}), (T2: Unlock\ A), (T1: Unlock\ A), (\underline{T4: Xlock\ A}),$
 $(T1: Unlock\ B), (\underline{T2: Xlock\ B}), (T4: Unlock\ A), (T2: Unlock\ B)]$

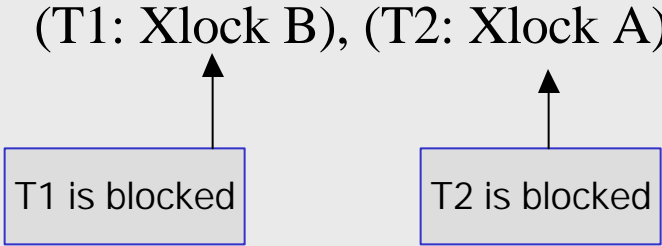


Edge #1: (T4: Slock B)...(T3: Xlock B)
Edge #2: (T1: Slock A)...(T4: Xlock A)
Edge #3: (T2: Slock A)...(T4: Xlock A)
Edge #4: (T3: Xlock A)...(T4: Xlock A)
Edge #5: (T3: Unlock A)...(T1: Slock A)
Edge #6: (T3: Unlock A)...(T2: Slock A)
Edge #7: (T3: Xlock B)...(T1: Xlock B)
Edge #8: (T1: Xlock B)...(T2: Xlock B)

Not serializable, cycle exists



Problems Locking Protocols

- The X-lock protocol can lead to deadlock.
 - For example consider the schedule $S = [(T1:Xlock\ A), (T2:Xlock\ B), (T1:Xlock\ B), (T2:Xlock\ A)]$ 
- While there are many different techniques that can be used to avoid deadlock, most are not suitable to the database environment.



Deadlock Avoidance - Problems Locking Protocols (cont.)

- Impose a total ordering on the objects.
 - Problem is the set of lockable objects is very large and changes dynamically.
 - Many database transactions determine the lockable object based on content and not name.
 - The locking scope of a transaction is typically determined dynamically.
- Two-phase locking protocols.
 - All locks are granted at the beginning of a transaction's processing or no locks are granted. Transactions which cannot acquire all of the locks they need are suspended without being granted any locks.
 - Leads to low data utilization, low-levels of concurrency and livelock.
 - Livelock occurs when a transaction that needs several "popular" items is consistently blocked by transactions which need only one of the popular items.



Deadlock Avoidance - Problems Locking Protocols (cont.)

- There is also a timestamp based protocol (under locking – don't confuse this with timestamp based concurrency controls we'll see later) to prevent deadlock under locking protocols.
- A timestamp is a unique identifier assigned to each transaction based upon the time a transaction begins.
 - if $ts(T_X) < ts(T_Y)$ then T_X is the older transaction and T_Y is the younger transaction.
 - In resolving deadlock issues, the system uses the value of the timestamp to determine if a transaction should wait or rollback. Locking is still used to control concurrency.
 - Under rollback a transaction retains its original timestamp.



Deadlock Resolution – Wait or Die

- Assume that T_X requests an object whose lock is held by T_Y .
- This is a non-preemptive strategy where if $ts(T_X) < ts(T_Y)$ (T_X is older than T_Y) then T_X is allowed to wait on T_Y , otherwise T_X dies (is rolled back). T_Y continues to hold the lock and T_X subsequently restarts with its original timestamp.
 - if request is made by older transaction – it waits on the younger transaction.
 - if request is made by younger transaction – it dies.
- Example: let $ts(T1) = 5$, $ts(T2) = 10$, $ts(T3) = 15$

Suppose T2 requests object held by T1. T2 is younger than T1, T2 dies.

Suppose T1 requests object held by T2. T1 is older than T2, T1 waits.



Deadlock Resolution – Wound or Wait

- Assume that T_X requests an object whose lock is held by T_Y .
- This is a preemptive strategy where if $ts(T_X) < ts(T_Y)$ (T_X is older than T_Y) then T_Y is aborted (T_X wounds T_Y). T_X preempts the lock and continues. Otherwise, T_X waits on T_Y .
 - if request is made by the younger transaction – it waits on the older transaction.
 - if request is made by older transaction – it preempts the lock and the younger transaction dies.
- Example: let $ts(T1) = 5$, $ts(T2) = 10$, $ts(T3) = 15$

Suppose $T2$ requests object held by $T1$. $T2$ is younger than $T1$, $T2$ waits.

Suppose $T1$ requests object held by $T2$. $T1$ is older than $T2$, $T1$ gets lock and $T2$ dies.



Timestamp Deadlock Resolution

- Both wait or die and wound or wait protocols avoid starvation. At any point in time there is a transaction with the smallest timestamp (i.e., oldest transaction) and it will not be rolled back in either scheme.

Operational Differences

- In wait or die, the older transaction waits for the younger one to release its locks, thus, the older a transaction gets, the more it will wait. In wound or wait, the older transaction never waits.
- In wait or die protocol if transaction T1 dies and is rolled back it will in probably be re-issued and generate the same set of requests as before. It is possible for T1 to die several times before it will be granted the lock it is requesting as the older transaction is still using the lock. Whereas, in wound or wait, it would restart once and then be blocked. Typically, the wound or wait protocol will result in fewer roll backs than does the wait or die protocol.



Deadlock Avoidance vs. Detection and Resolution

- If the deadlock prevention or avoidance mechanism is not 100% effective, then it is possible for a set of transactions to become deadlocked.
- Handling this problem can be achieved in one of two basic manners: optimistically or pessimistically.
- Optimistic approaches tend to wait for deadlock to occur before doing anything about it, while pessimistic approaches tend to make sure that deadlock cannot occur.
- Optimistic approaches use detection and resolution schemes while pessimistic approaches use avoidance mechanisms.



Deadlock Detection and Resolution

- Deadlock detection and resolution involves two phases: detection of deadlock and its resolution.
- Deadlock detection is commonly done with wait-for graphs (a form of a precedence graph). Each node in the graph represents a transaction in the system. An edge from transaction T_X to transaction T_Y indicates that T_X is waiting on an object currently held by T_Y . A deadlock is detected if the graph contains a cycle.
- The resolution phase or the recovery from the deadlock, essentially amounts to selecting a victim of the deadlock to be rolled back, thus breaking the deadlock.



Deadlock Detection and Resolution (cont.)

- Selection of a victim to resolve the deadlock can be based upon many different things:
 - how long has the transactions been processing?
 - how much longer does the transaction require to complete?
 - how much data has been read/written?
 - how many data items are still needed?
 - how many transactions will need to be rolled back?
- Once a victim has been selected you can decide how far back to roll it. It is not always necessary for a complete restart.
- Deadlock detection and resolution requires some mechanism to prevent starvation from occurring. Typically this is done by limiting the number of times a single transaction can be identified as the “victim”.



Timestamping Concurrency Control

- No locking is used with timestamp concurrency control. Do not confuse this topic with the timestamped method for avoiding deadlock under locking.
- As before, each transaction is issued a unique timestamp indicating the time it arrived in the system.
- The size of the timestamp varies from system to system, but must be sufficiently large to cover transactions processing over long periods of time.
- Assignment of the timestamp is typically handled by the long-term scheduler as transactions are removed from some sort of input queue.



Timestamping Concurrency Control (cont.)

- In addition to the transaction's timestamp, each object in the database has associated with it two timestamps:
 - read timestamp – denoted $rts(object)$, and it represents the highest timestamp of any transaction which has successfully read this object.
 - write timestamp – denoted $wts(object)$, and it represents the highest timestamp of any transaction to successfully write this object.
- As with locking the granularity of an “object” in the database becomes a concern here, since the overhead of the timestamps can be considerable if the granularity is too fine.



Timestamp Ordering Protocol

READ – transaction T_x performs read(object)

if $ts(T_x) < wts(object)$

then rollback T_x // implies that the value of the object has been written by a
// transaction T_y which is younger than T_x

else // $ts(T_x) \geq wts(object)$

execute read(object)

set $rts(object) = \max\{rts(object), ts(T_x)\}$

WRITE – transaction T_x performs write(object)

if $ts(T_x) < rts(object)$

then rollback T_x //implies that the value of the object being produced by T_x was
//read by a transaction T_y which is younger than T_x and T_y
//assumed the value of the object was valid.

else if $ts(T_x) < wts(object)$

then ignore write(object) //implies that T_x is attempting to write an “old”
//value which has been updated by a younger
//transaction.

else

execute write(object)

set $wts(object) = \max\{wts(object), ts(T_x)\}$



Explanation of the Ignore Write Rule

- In the timestamp ordering protocol, when the timestamp of the transaction attempting to write an object is less than the write timestamp of the object of concern, the write is simply ignored.
- This is known as **Thomas's write rule**.
- Suppose that we have two transactions T1 and T2 where T1 is the older transaction. T1 attempts to write object X. If $ts(T1) < wts(X)$ then if T2 was the last transaction to write X, $wts(X) = ts(T2)$ and between the time T2 wrote X and T1 attempted to write X, no other transaction Tn read X or otherwise $rts(X) > ts(T1)$ and T1 would have aborted when attempting to write X. Thus T1 and T2 have read the same value of X and since T2 is younger, the value that would have been written by T1 would simply have been overwritten by T2, so T1's write can be ignored.



Example - Timestamp Ordering Protocol

	Transactions				Objects		
time	T1	T2	T3	Action	A	B	C
initial	ts = 200	ts = 150	ts = 175		rts = 0 wts = 0	rts = 0 wts = 0	rts = 0 wts = 0
1	read B			allowed		rts = 200	
2		read A		allowed	rts = 150		
3			read C	allowed			rts = 175
4	write B			allowed		wts = 200	
5	write A			allowed	wts = 200		
6		write C		abort T2 ts(T2) < rts(C)			
7			write A	ignore			
final					rts = 150 wts = 200	wts = 200 rts = 200	rts = 175 wts = 0

