Transaction Management in Concurrency Control

Michael Yu

CSCI3170 2024T1

Concurrency Control

Recall: We now categorized schedules that produces a correct results

Additionally:

1. Transactions are submitted to the system
2. Fixed schedules cannot be created in advance
3. We need concurrency control, and do not tolerate serial execution

How to control the interleaving of transactions systematically to produce correct results?

Recap:

1. Non-serial schedules
2. Serializability
3. Conflict serializability

Concept of Locks

A **lock** is a mechanism to control concurrent **access to a data item**

To **use/access** a data item, you must acquire the relevant locks, which each transaction will Transaction can only after request is granted.

Data items can be locked in two ways, as described in the two modes:

1. ***Exclusive Lock/ X Lock / Write Lock***
   - The data item can be read
   - The data item can be written

2. ***Shared Lock/ S Lock/ Read Lock***
   - The data item can be read (only)

Schedules plus Lock-Based Protocols

T_1	T_2	concurrency-control manager
write_lock(B)		
		grant-write_lock(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	read_lock(A)	
		grant-read_lock(A, T_2)
	read(A)	
	unlock(A)	
	read_lock(B)	
		grant-read_lock(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
write_lock(A)		
		grant-write_lock(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Note:

- Grants omitted in rest of slides
- Assume grant happens just before the next instruction following lock request

Lock Requests

Lock requests are made to **concurrency-control manager**

1. An exclusive lock is requested using **write_lock()** instruction
2. A shared lock is requested using **read_lock()** instruction

What happens next?

- A transaction may be granted a lock on an item if the **requested lock** is **compatible** with locks already held on the item by other transactions

- If a lock cannot be granted, the ***requesting transaction*** is made to wait till all **incompatible** locks held by other transactions have been **released**. The lock is then granted.

Lock-Based Protocols

"Lock compatibility becomes an issue when one application holds a lock on an object and another application requests a lock on the same object."

Presenting the **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

Hint: Perhaps you now begin to notice how locking protocols enforce serializability by restricting the set of possible schedules?

Lock-Based Protocols

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocol 1: A simple locking protocol

If *T* has only one operation manipulating an item *X*:

- *if read:* obtain a read lock on *X* before reading
- *if write:* obtain a write lock on *X* before writing
- unlock *X after this operation on X*

Even if *T* has several operations manipulating *X,* we obtain **one** lock still:

- if all operations on *X* are reads, obtain read lock
- if at least **one** operation on *X* is a write, obtain write lock
- unlock *X* after the **last** operation on *X*

Lock-Based Protocols

Example (with slight variation to lecture notation)

T_1

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

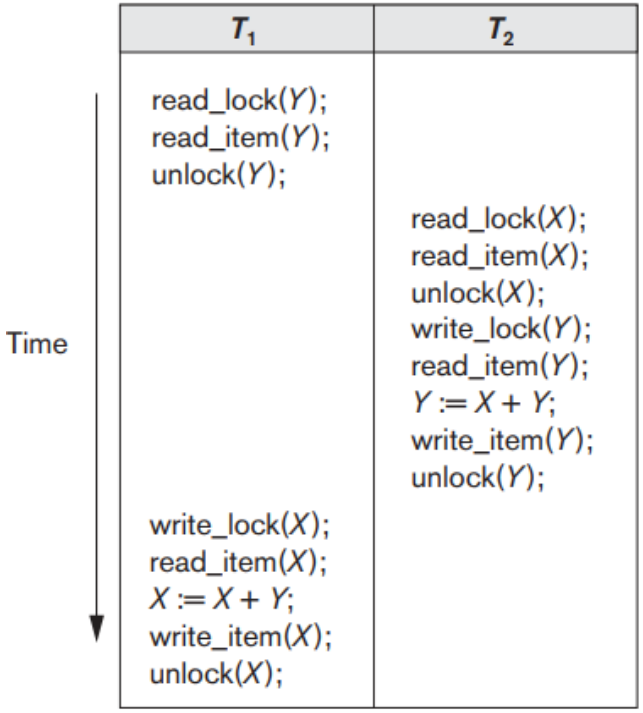
T_2

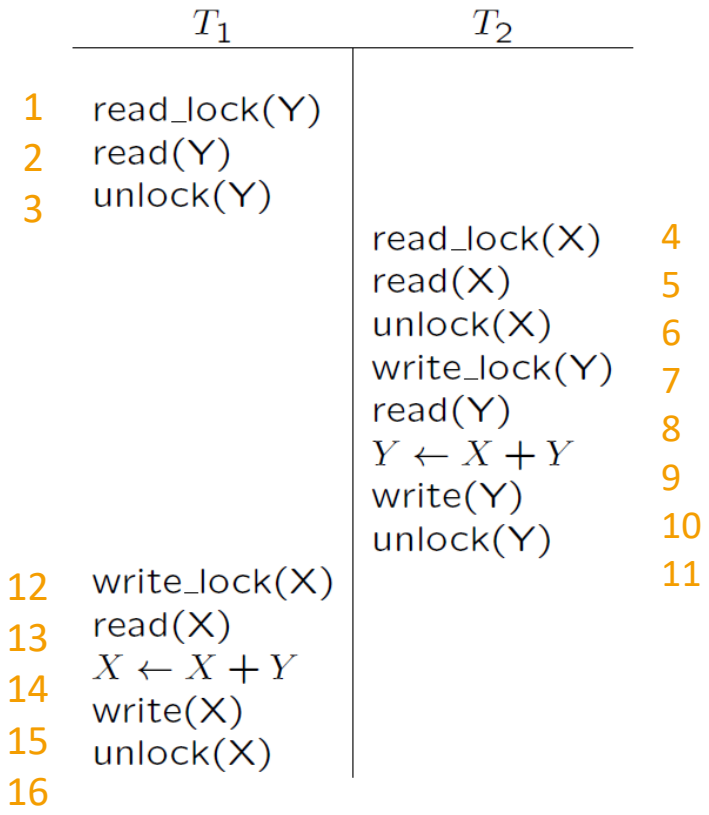
read_lock(X);
read_item(X);
unlock(X);
write_lock(Y);
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Lock-Based Protocols

Simple locking protocol in action, with minor notational differences

(Left hand side example with slight variation to lecture notation)





Lock-Based Protocols (3)

Example of transactions performing locking:

T_1: **write_lock***(B)*;
read*(B)*;
B: = B – 50;
write*(B)*;
unlock*(B)*;
write_lock*(A)*;
read*(A)*;
A: = A + 50;
write*(A)*;
unlock*(A)*;

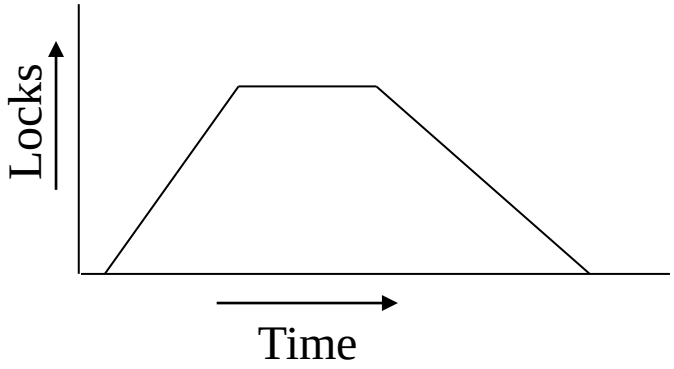
T_2: **read_lock***(A)*;
read*(A)*;
unlock*(A)*;
read_lock*(B)*;
read*(B)*;
unlock*(B)*;
display*(A+B);*

Locking as above is ***not sufficient*** to guarantee serializability

Two Phase Locking (2PL)

Locking protocol 2:

1. Phase 1: Growing Phase
   - Transaction obtains locks
   - Transaction does not release any locks
2. Phase 2: Shrinking Phase
   - Transaction releases locks
   - Transaction does not obtain new locks



This protocol ensures serializability: and produces conflict-serializable schedules. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Two Phase Locking (2PL)

Example of transactions performing locking:

T_3: **write_lock**(*B*);
read(*B*);
B: = B – 50;
write(*B*);
write_lock(*A*);
read(*A*);
A: = A + 50;
write(*B*);
unlock(*B*);
unlock(*A*);

T_4: **read_lock**(*A*);
read(*A*);
read_lock(*B*);
read(*B*);
display(*A+B*);
unlock(*A*);
unlock(*B*);

Locking as above is ***sufficient*** to guarantee serializability, if unlocking is delayed to the end of the transaction.

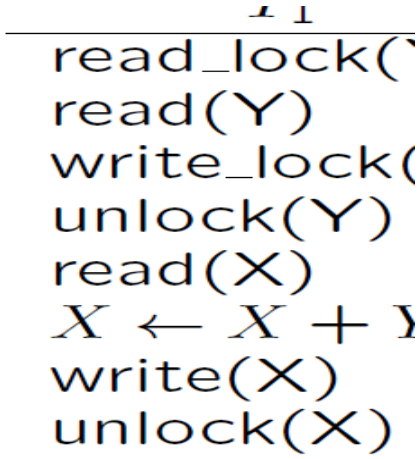
"each transaction must never acquire a lock after it has released a lock. The serializability property is guaranteed for a schedule with transactions that obey this rule"

2PL can guarantee serializability, thus allowing real-time control of concurrent executions (scheduling).

Two Phase Locking (2PL)

T_1	T_2
read_lock(Y)	
read(Y)	
unlock(Y)	
	read_lock(X)
	read(X)
	unlock(X)
	write_lock(Y)
	read(Y)
	$Y \leftarrow X + Y$
	write(Y)
	unlock(Y)
write_lock(X)	
read(X)	
$X \leftarrow X + Y$	
write(X)	
unlock(X)	

T1 that's with no the 2PL scheme



A transaction that locks with 2PL scheme

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.

In most locking protocols, a deadlock can exist.

Deadlocks

Consider the partial schedule to the right.

The instructions from T3 and T4 arrive at the system…

- executing **read_lock(B):**
  - T4 waits for T3 to release its lock on B
- executing **write_lock(A):**
  - T3 waits for T4 to release its lock on A

Such a situation is called a **deadlock**.

Issue: neither T3 nor T4 can make progress

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Deadlock Prevention Scheme

Locking protocols are used in most commercial DBMSs.

We need to address this issue of deadlocks

One way to prevent deadlock is to use a **deadlock prevention protocol**.

One More Example:

- T1: r1[x] w1[y]
- T2: r2[y] w2[x]
- Execution: r1[x] r2[y] deadlock!
- T1 waits T2 to release the read lock on y, and T2 waits T1 to release the read lock on x.
- Need a effective strategy to detect deadlocks

Deadlock Prevention Scheme

Timeout Strategy

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.

Pro: Small overhead and is simplicity.

Con: There may not be a deadlock

Timeout Strategy – More Cons

- Waiting too long for a lock.
  1. guess that there is a deadlock
  2. abort the transaction
- No harm for wrong guess, insofar as correctness is concerned.
- Performance penalty for wrong guesses.
- Tune the timeout intervals.
  - Long enough so that most transactions that are aborted are actually deadlocked.
  - Short enough that deadlocked transactions don't wait too long for their deadlocks to be noticed.

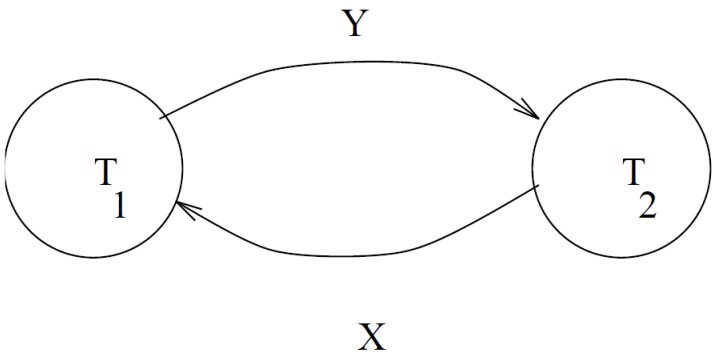
Testing for Deadlocks

Create a *wait-for graph* for the current state of on-going active transactions:

- create a vertex for each transaction; and
- an arc from T_i to T_j if T_i **is waiting** for an item locked by T_j.

If the graph has a cycle, then a *deadlock* has occurred.

Example:



Testing for Deadlocks

Case: wait-for graph with cycle(s)

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Current Observations with 2PL

1. Use of locks, combined with the 2PL protocol, **guarantees serializability of schedules**
2. Schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire
3. If transaction needs an item that is already locked, it may be forced to wait until the item is released.

Deadlock Prevention Scheme

Context:

- Lock-based concurrency control cannot prevent deadlocks.
- We need an active solution for deadlocks beyond locks.

A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Concurrency Control with Timestamps

- Not based on locks!
- Can assign each transaction **T_i** a “timestamp” T **$TS(T_i)$**
- unique identifier to identify a transaction

Timestamp Implementation Op.

Timestamps can be generated in several ways.

- Possibility 1:
  - A simple **counter** (e.g., int counter)
  - (Increment value each time its value is assigned to a transaction.)
- Possibility 2:
  - Use the current **date/time value** of the system clock.

Timestamp Ordering

Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

For two transactions T_i and T_j that may be involved in a deadlock.
Assume T_i wants a lock that T_j holds, two policies are possible:

- Policy 1 (**Wait-Die Protocol**):
  - If T_i is older, T_i waits for T_j
  - If T_i is younger, T_i aborts*
- Policy 2 (**Wound-wait Protocol**):
  - If T_i is younger, T_i waits for T_j
  - If T_i is older, T_j aborts*

Note:

1. T_i older than T_j if **$TS(T_i) < TS(T_j)$**
2. * If a transaction re-starts, it retains its original timestamp

Cyclic Restart

Notice: both the schemes end up aborting the younger of the two transaction. Why?

It could lead to **cyclic restart**

- A kind of “live lock” can occur - transactions may be constantly aborted and restarted.

Resolved if when a transaction re-starts, it retains its original timestamp

Deadlock Prevention Scheme

- Concurrency control via timestamp ordering:
  - Ensures that the result final schedule recorded is equivalent to executing the transactions serially in timestamp order
  - Therefore, ensuring serializability
- Compare this with concurrency control via 2PL:
  - In 2PL, a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols

2PL vs. TSO

In terms of the comparison among **two-phase locking** (2PL), **timestamp ordering** (TSO) concurrency control techniques.

1. 2PL can offer the greatest degree of concurrency (on average); but will result in deadlocks

To resolve the deadlocks, either

1. additional computation - detect and resolve deadlocks
2. adding other restrictions – reducing degree of concurrency

As a guideline, if most transactions are very short, it is permissible to use **2PL** + **deadlock** detection (and resolution)

2PL vs. TSO (cont.)

In terms of the comparison among **two-phase locking** (2PL), **timestamp ordering** (TSO) concurrency control techniques.

1. When compared to that of 2PL (with proper deadlock resolution)
   - TSO still achievers a lesser degree of concurrency in comparison.
2. TSO does not cause deadlocks, but exhibits other problems. E.g., cyclic restart and *cascading rollback**

* Topic for next Monday

Learning Outcomes

- Concurrency Control Techniques:
  - Lock-based - Ensures serializability, but could result in deadlocks
  - Timestamp-based - Prevents deadlocks

Plan Next Week (Week12):

- Monday
  - Database Recovery Techniques
- Wednesday
  - Summary of Course Contents (in anticipation of Exam)
  - 1-2 Helpful Mini Topics relating to SQL/ Project

Plan Final Week (Week13):

- Lecture as Consultation Sections
- Tutorials Plans (TBA)