Introduction to Database Systems CSE 444

Lectures 17-18: Concurrency Control

November 5-7, 2007

Outline

- Serial and Serializable Schedules (18.1)
- Conflict Serializability (18.2)
- Locks (18.3)
- Multiple lock modes (18.4)
- The tree protocol (18.7)
- Concurrency control by timestamps 18.8
- Concurrency control by validation 18.9

The Problem

- Multiple transactions are running concurrently T_1, T_2, …
- They read/write some common elements A_1, A_2, …
- How can we prevent unwanted interference ?

The SCHEDULER is responsible for that

Three Famous Anomalies

What can go wrong if we didn't have concurrency control:

- Dirty reads
- Lost updates
- Inconsistent reads

Many other things may go wrong, but have no names

Dirty Reads

T_1: WRITE(A)

T_1: ABORT

T_2: READ(A)

Lost Update

T_1: READ(A)

T_1: A := A+5

T_1: WRITE(A)

T_2: READ(A);

T_2: A := A*1.3

T_2: WRITE(A);

Inconsistent Read

T_1: A := 20; B := 20;	
T_1: WRITE(A)	
	T_2: READ(A);
	T_2: READ(B);
T_1: WRITE(B)	

Schedules

Given multiple transactions:

- A *schedule* is a sequence of interleaved actions from all transactions
- A *serial schedule* is one whose actions consist of all those of one transaction, followed by all those of another transaction, etc.

Example

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Serializable Schedule

- A schedule is *serializable* if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

Notice: this is NOT a serial schedule

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Ignoring Details

- Sometimes transactions' actions may commute accidentally because of specific updates
  - Serializability is undecidable !
- The scheduler shouldn't look at the transactions' details
- Assume worst case updates, only care about reads r(A) and writes w(A)

Notation

T_1: $r_1(A); w_1(A); r_1(B); w_1(B)$
T_2: $r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Conflicts:

Two actions by same transaction T_i: $\boxed{r_i(X); w_i(Y)}$

Two writes by T_i, T_j to same element $\boxed{w_i(X); w_j(X)}$

Read/write by T_i, T_j to same element $\boxed{w_i(X); r_j(X)}$ $\boxed{r_i(X); w_j(X)}$

Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

Example:

$$\boxed{r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)}$$

$$\Downarrow$$

$$\boxed{r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)}$$

Conflict Serializability

- Any conflict serializable schedule is also a serializable schedule (why ?)
- The converse is not true, even under the "worst case update" assumption

Lost write

$$\boxed{w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);}$$

$$\cancel{\Downarrow}$$

Equivalent, but can't swap

$$\boxed{w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);}$$

The Precedence Graph Test

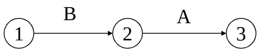
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions T_i
- Edge from T_i to T_j if T_i makes an action that conflicts with one of T_j and comes first
- The test: if the graph has no cycles, then it is conflict serializable !

Example 1

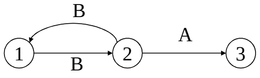
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- How? Three techniques:
  - – Locks
  - – Time stamps
  - – Validation

Locking Scheduler

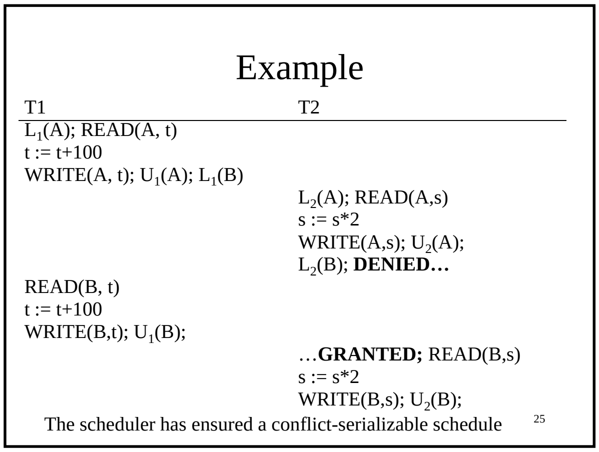
Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

Notation

$L_i(A)$ = transaction T_i acquires lock for element A

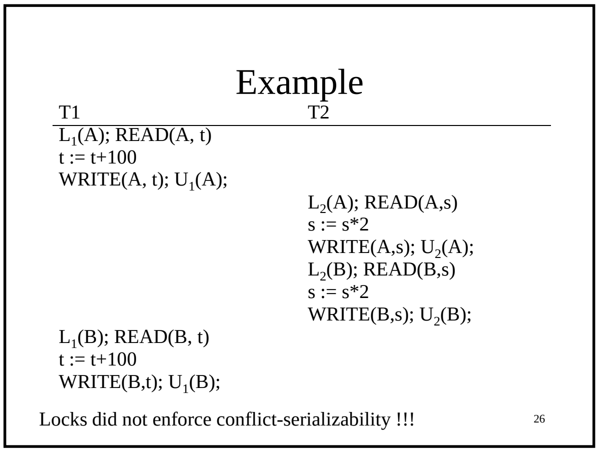
$U_i(A)$ = transaction T_i releases lock for element A



Example

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A); L_1(B)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s); U_2(A);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(B);

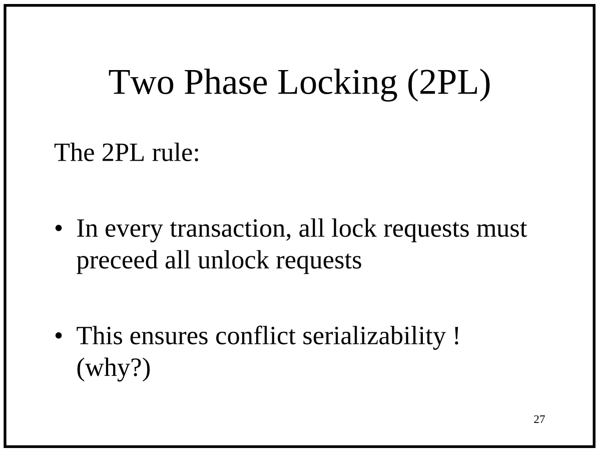
The scheduler has ensured a conflict-serializable schedule



Example

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A);	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s); U_2(A);
	L_2(B); READ(B,s)
	s := s*2
	WRITE(B,s); U_2(B);
L_1(B); READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	

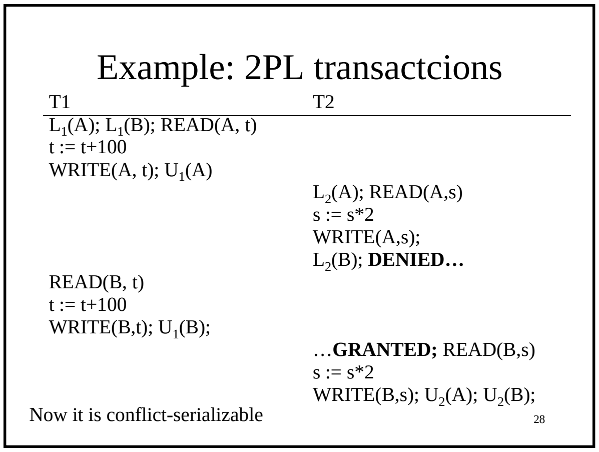
Locks did not enforce conflict-serializability !!!



Two Phase Locking (2PL)

The 2PL rule:

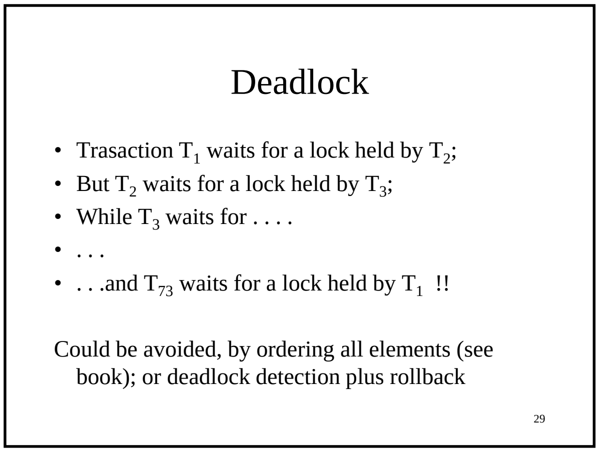
- In every transaction, all lock requests must preceed all unlock requests
- This ensures conflict serializability ! (why?)



Example: 2PL transactcions

T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);

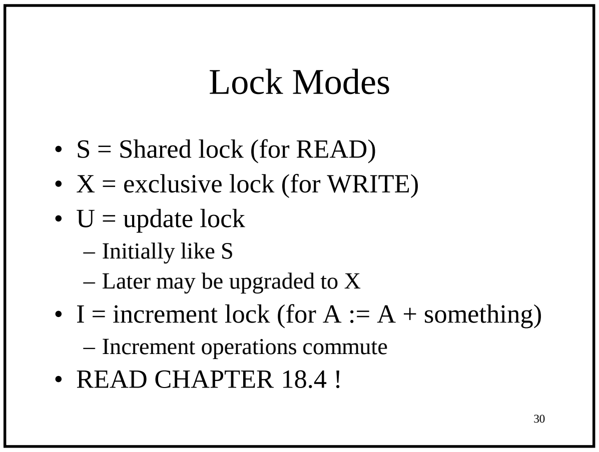
Now it is conflict-serializable



Deadlock

- Trasaction T_1 waits for a lock held by T_2;
- But T_2 waits for a lock held by T_3;
- While T_3 waits for . . . .
- . . .
- . . .and T_{73} waits for a lock held by T_1 !!

Could be avoided, by ordering all elements (see book); or deadlock detection plus rollback



Lock Modes

- S = Shared lock (for READ)
- X = exclusive lock (for WRITE)
- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute
- READ CHAPTER 18.4 !

The Locking Scheduler

Task 1:
add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure 2PL !

The Locking Scheduler

Task 2:
execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
  - Grant, or add the transaction to the element's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

The Tree Protocol

- An alternative to 2PL, for tree structures
- E.g. B-trees (the indexes of choice in databases)

The Tree Protocol

Rules:

- The first lock may be any node of the tree
- Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
- Nodes can be unlocked in any order (no 2PL necessary)

The tree protocol is NOT 2PL, yet ensures conflict-serializability !

Timestamps

Every transaction receives a unique timestamp TS(T)

Could be:

- The system's clock
- A unique counter, incremented by the scheduler

Timestaps

Main invariant:

The timestamp order defines
the searialization order of the transaction

Timestamps

Associate to each element X:

- RT(X) = the highest timestamp of any transaction that read X
- WT(X) = the highest timestamp of any transaction that wrote X
- C(X) = the commit bit: says if the transaction with highest timestamp that wrote X commited

These are associated to each page X in the buffer pool

Main Idea

For any two conflicting actions, ensure that their order is the serialized order:

In each of these cases

- $w_U(X) \ldots r_T(X)$
- $r_U(X) \ldots w_T(X)$
- $w_U(X) \ldots w_T(X)$

Check that TS(U) < TS(T)

Read too late ?

Write too late ?

No problem (WHY ??)

When T wants to read X, $r_T(X)$, how do we know U, and TS(U) ?

Details

Read too late:

- T wants to read X, and TS(T) < WT(X)

START(T) … START(U) … $w_U(X) \ldots r_T(X)$

Need to rollback T !

Details

Write too late:

- T wants to write X, and WT(X) < TS(T) < RT(X)

START(T) … START(U) … $r_U(X) \ldots w_T(X)$

Need to rollback T !

Why do we check WT(X) < TS(T) ??

Details

Write too late, but we can still handle it:

- T wants to write X, and TS(T) < RT(X) but WT(X) > TS(T)

START(T) … START(V) … $w_V(X) \ldots w_T(X)$

Don’t write X at all !
(but see later…)

More Problems

Read dirty data:

- T wants to read X, and WT(X) < TS(T)
- Seems OK, but…

START(U) … START(T) … $w_U(X) \ldots r_T(X) \ldots$ ABORT(U)

If C(X)=1, then T needs to wait for it to become 0

More Problems

Write dirty data:

- T wants to write X, and WT(X) > TS(T)
- Seems OK not to write at all, but …

START(T) … START(U)… w_U(X). . . w_T(X)… ABORT(U)

If C(X)=1, then T needs to wait for it to become 0

Timestamp-based Scheduling

When a transaction T requests r(X) or w(X), the scheduler examines RT(X), WT(X), C(X), and decides one of:

- To grant the request, or
- To rollback T (and restart with later timestamp)
- To delay T until C(X) = 0

Timestamp-based Scheduling

RULES:

- There are 4 long rules in the textbook, on page 974
- You should be able to understand them, or even derive them yourself, based on the previous slides
- Make sure you understand them !

READING ASSIGNMENT: 18.8.4

Multiversion Timestamp

- When transaction T requests r(X)
  but WT(X) > TS(T),
  then T must rollback
- Idea: keep multiple versions of X:
  X_t, X_{t-1}, X_{t-2}, . . .

$TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots$

- Let T read an older version, with appropriate timestamp

Details

- When w_T(X) occurs create a new version, denoted X_t where t = TS(T)
- When r_T(X) occurs, find a version X_t such that t < TS(T) and t is the largest such
- WT(X_t) = t and it never changes
- RD(X_t) must also be maintained, to reject certain writes (why ?)
- When can we delete X_t: if we have a later version X_{t1} and all active transactions T have TS(T) > t1

Tradeoffs

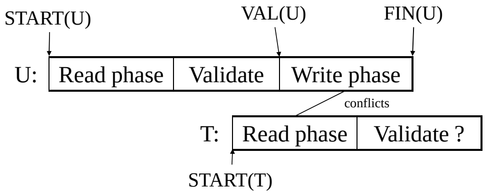
- Locks:
  - Great when there are many conflicts
  - Poor when there are few conflicts
- Timestamps
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts
- Compromise
  - READ ONLY transactions → timestamps
  - READ/WRITE transactions → locks

Concurrency Control by Validation

- Each transaction T defines a *read set* RS(T) and a *write set* WS(T)
- Each transaction proceeds in three phases:
  - Read all elements in RS(T). Time = START(T)
  - Validate (may need to rollback). Time = VAL(T)
  - Write all elements in WS(T). Time = FIN(T)

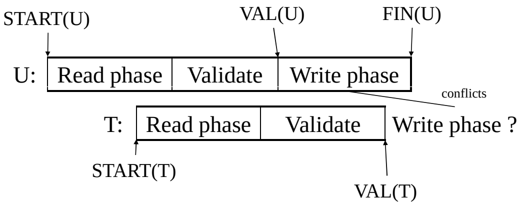
Main invariant: the serialization order is VAL(T)

Avoid $r_T(X)$ - $w_U(X)$ Conflicts



IF RS(T) $\cap$ WS(U) and FIN(U) > START(T)
(U has validated and U has not finished before T begun)
Then ROLLBACK(T)

Avoid $w_T(X)$ - $w_U(X)$ Conflicts



IF WS(T) $\cap$ WS(U) and FIN(U) > VAL(T)
(U has validated and U has not finished before T validates)
Then ROLLBACK(T)

Final comments

- Locks and timestamps: SQL Server, DB2
- Validation: Oracle

(more or less)