CSE 344

MARCH 23RD – SCHEDULING/LOCKING

ADMINISTRIVIA

- **HW7 Due Tonight**
- **OQ6 Due Tonight**
- **HW8 Due Friday, June 1**
  - Data without quotation marks
  - Extra credit
- **OQ7 Due Wednesday, May 30**
- **Course Evaluations**
  - Out over the weekend

TRANSACTIONS

We use database transactions everyday

- Bank $$$ transfers
- Online shopping
- Signing up for classes

For this class, a transaction is a series of DB queries

- Read / Write / Update / Delete / Insert
- Unit of work issued by a user that is independent from others

KNOW YOUR TRANSACTIONS: ACID

Atomic

- State shows either all the effects of txn, or none of them

Consistent

- Txn moves from a DBMS state where integrity holds, to another where integrity holds
  - remember integrity constraints?

Isolated

- Effect of txns is the same as txns running one after another (i.e., looks like batch mode)

Durable

- Once a txn has committed, its effects remain in the database

SERIAL SCHEDULE

A *serial schedule* is one in which transactions are executed one after the other, in some sequential order

Fact: nothing can go wrong if the system executes transactions serially

- (up to what we have learned so far)
- But DBMS don't do that because we want better overall system performance

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)

This is a serializable schedule.
This is NOT a serial schedule

CONFLICT SERIALIZABILITY

Conflicts: (i.e., swapping will change program behavior)

Two actions by same transaction T_i:

$r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element

$w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

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

Every conflict-serializable schedule is serializable

The converse is not true (why?)

SCHEDULER

Scheduler = the module that schedules the transaction's actions, ensuring serializability

Also called Concurrency Control Manager

We discuss next how a scheduler may be implemented

IMPLEMENTING A SCHEDULER

Major differences between database vendors

Locking Scheduler

- Aka “pessimistic concurrency control”
- SQLite, SQL Server, DB2

Multiversion Concurrency Control (MVCC)

- Aka “optimistic concurrency control”
- Postgres, Oracle: Snapshot Isolation (SI)

We discuss only locking schedulers in this class

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)

By using locks scheduler ensures conflict-serializability

WHAT DATA ELEMENTS ARE LOCKED?

Major differences between vendors:

Lock on the entire database

- SQLite

Lock on individual records

- SQL Server, DB2, etc

CASE STUDY: SQLITE

SQLite is very simple

More info: http://www.sqlite.org/atomiccommit.html

Lock types

- READ LOCK (to read)
- RESERVED LOCK (to write)
- PENDING LOCK (wants to commit)
- EXCLUSIVE LOCK (to commit)

SQLITE

Step 1: when a transaction begins

Acquire a **READ LOCK** (aka "SHARED" lock)

All these transactions may read happily

They all read data from the database file

If the transaction commits without writing anything, then it simply releases the lock

SQLITE

Step 2: when one transaction wants to write

Acquire a RESERVED LOCK

May coexists with many READ LOCKs

Writer TXN may write; these updates are only in main memory; others don't see the updates

Reader TXN continue to read from the file

New readers accepted

No other TXN is allowed a RESERVED LOCK

SQLITE

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can’t coexists with *read locks*

Acquire a PENDING LOCK

May coexists with old READ LOCKs

No new READ LOCKS are accepted

Wait for all read locks to be released

Why not write to disk right now?

SQLITE

Step 4: when all read locks have been released

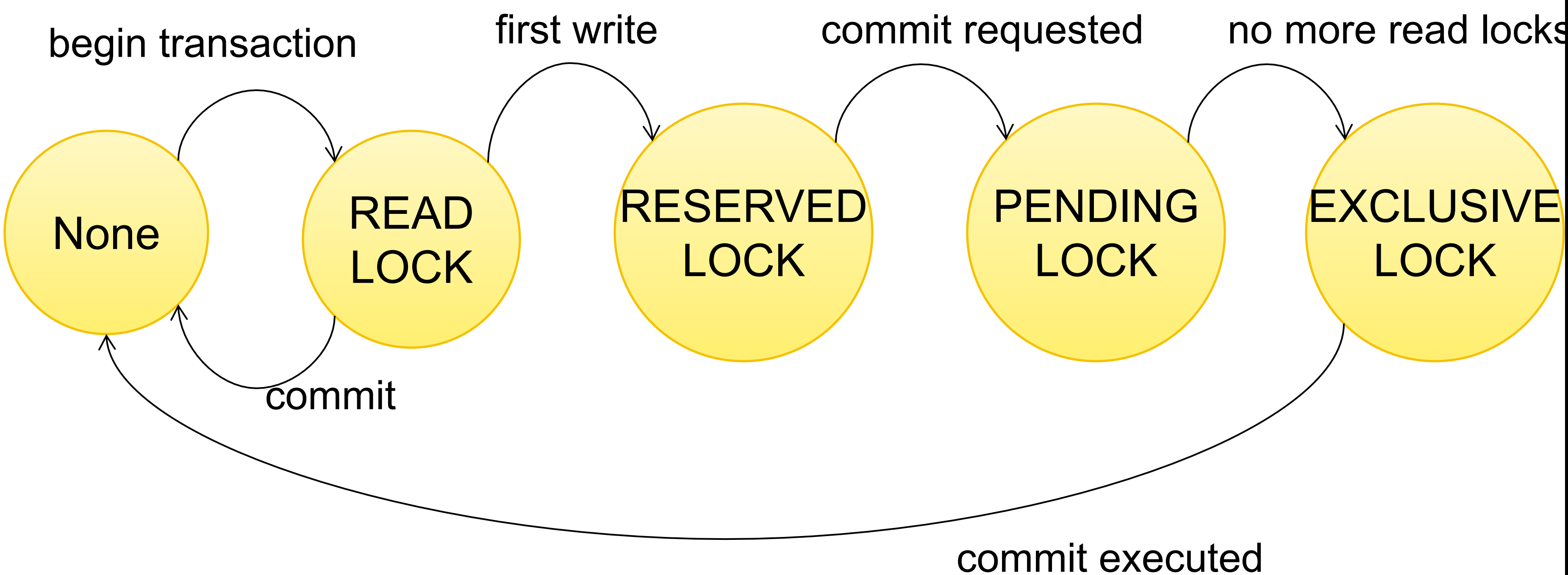
Acquire the **EXCLUSIVE LOCK**

Nobody can touch the database now

All updates are written permanently to the database file

Release the lock and **COMMIT**

SQLITE


begin transaction
first write
commit requested
no more read locks
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

SCHEDULE ANOMALIES

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

- Dirty reads (including inconsistent reads)
- Unrepeatable reads
- Lost updates

Many other things can go wrong too

DIRTY READS

Write-Read Conflict

T_1: WRITE(A)

T_2: READ(A)

T_1: ABORT

INCONSISTENT READ

Write-Read Conflict

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

UNREPEATABLE READD

Read-Write Conflict

T_1: WRITE(A)

T_2: READ(A);

T_2: READ(A);

LOST UPDATE

Write-Write Conflict

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);

MORE NOTATIONS

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

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

A NON-SERIALIZABLE SCHEDULE

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

EXAMPLE

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

BUT…

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

Locks did not enforce conflict-serializability !!! What's wrong ?

TWO PHASE LOCKING (2PL)

The 2PL rule:

In every transaction, all lock requests
must precede all unlock requests

EXAMPLE: 2PL TRANSACTIONS

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

Now it is conflict-serializable

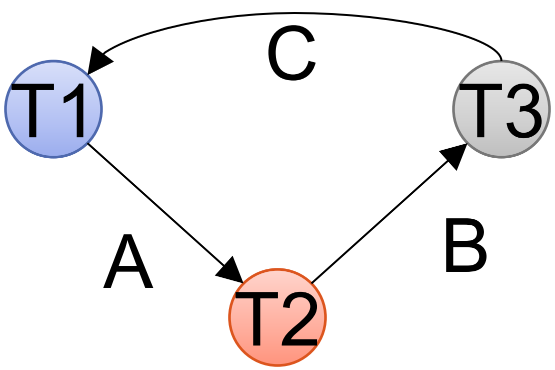
TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

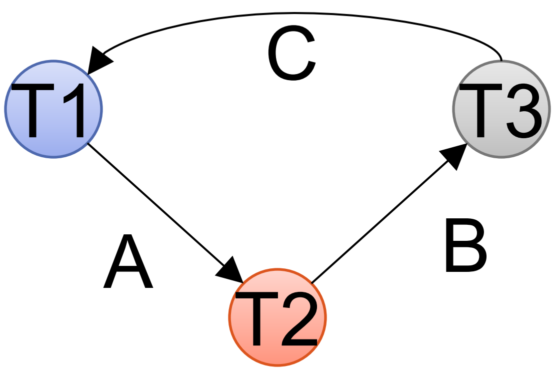
Proof. Suppose not: then there exists a cycle in the precedence graph.



TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

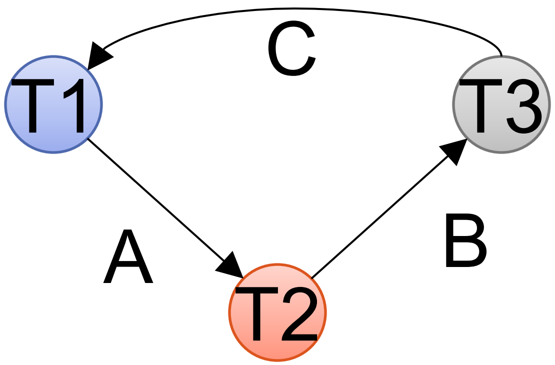


Then there is the following **temporal** cycle in the schedule:

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

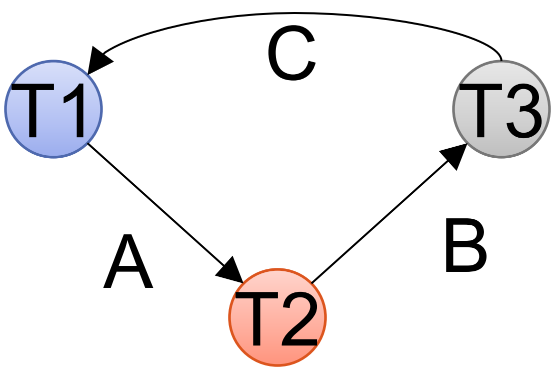
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ happened strictly *before* $L_2(A)$

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



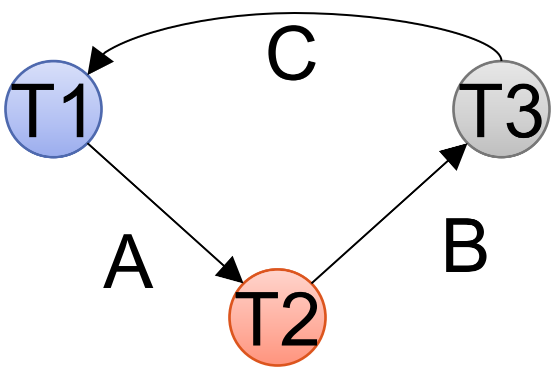
Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$ why?

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

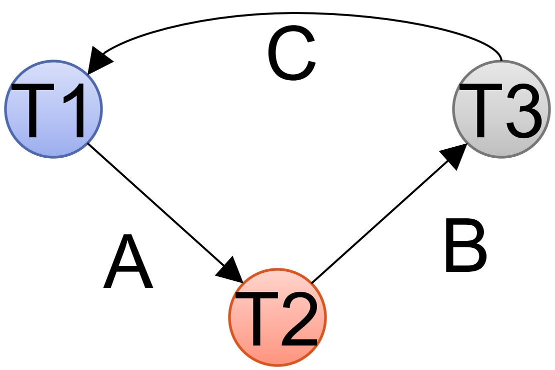
$L_2(A) \rightarrow U_2(B)$ why?

$L_2(A)$ happened strictly *before* $U_1(A)$

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

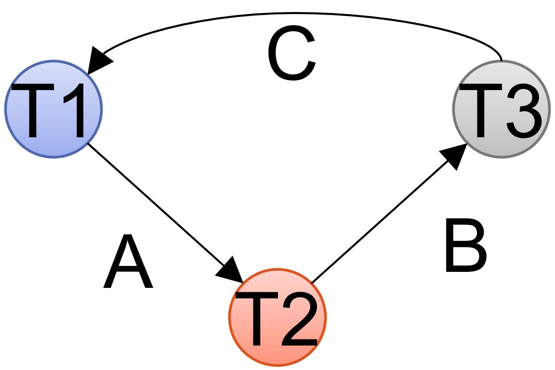
$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$ why?

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

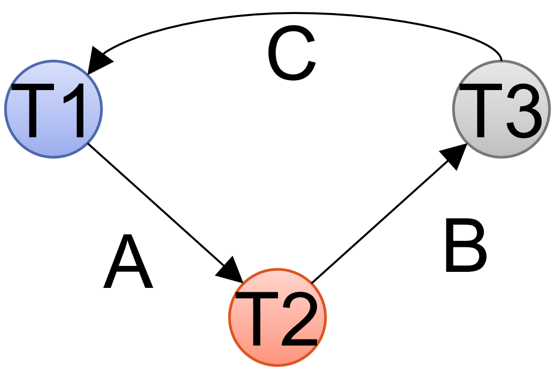
$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$ why?

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

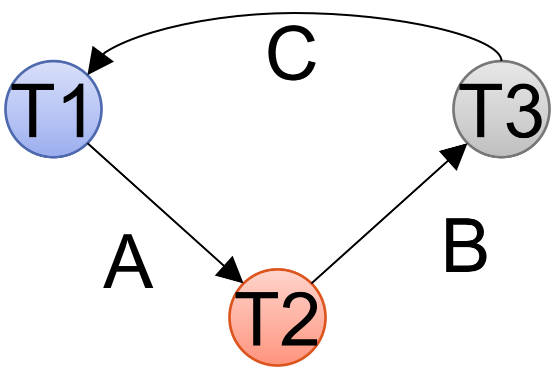
$U_2(B) \rightarrow L_3(B)$

......etc.....

TWO PHASE LOCKING (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following <u>**temporal**</u> cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$

$L_3(B) \rightarrow U_3(C)$

$U_3(C) \rightarrow L_1(C)$

$L_1(C) \rightarrow U_1(A)$

Cycle in time: Contradiction

A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

T1	T2
L_1(A); L_1(B); READ(A)	
A :=A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);
	Commit
Rollback	

Elements A, B written by T1 are restored to their original value.

A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

A NEW PROBLEM: NON-RECOVERABLE SCHEDULE

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

STRICT 2PL

The Strict 2PL rule:

All locks are held until commit/abort:
All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that
are both conflict-serializable and recoverable

STRICT 2PL

T1	T2
$L_1(A)$; READ(A)	
A :=A+100	
WRITE(A);	
	$L_2(A)$; BLOCKED…
$L_1(B)$; READ(B)	
B :=B+100	
WRITE(B);	
Rollback & $U_1(A)$;$U_1(B)$;	
	…GRANTED; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B);
	Commit & $U_2(A)$; $U_2(B)$;

STRICT 2PL

Lock-based systems always use strict 2PL

Easy to implement:

- Before a transaction reads or writes an element A, insert an L(A)
- When the transaction commits/aborts, then release all locks

Ensures both conflict serializability and recoverability