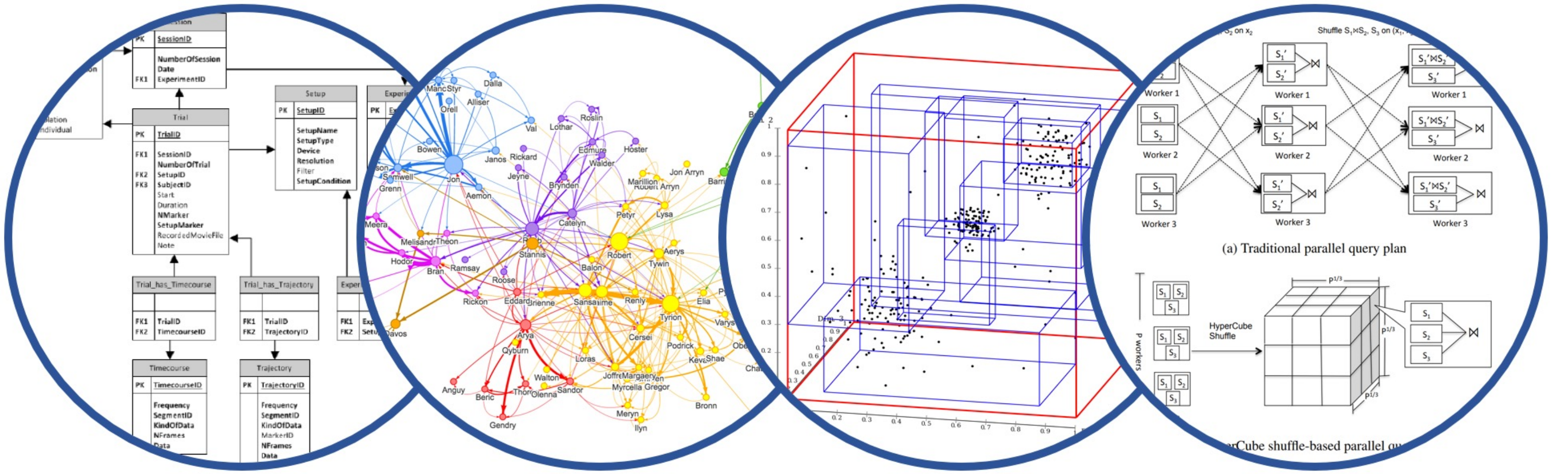


Introduction to Data Management

Transactions: Locks

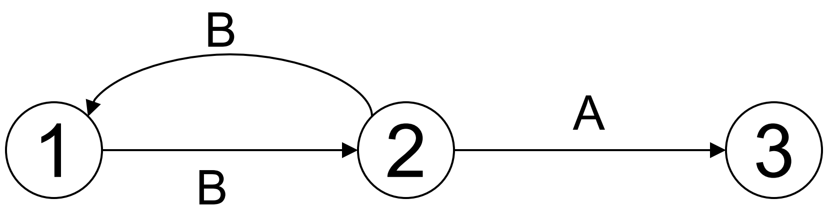
Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Recap

- TXN = sequence of Reads and Writes of elements
- Schedule = interleaving of operations of TXNs
- Serial Schedule = one TXN after the other
- Serializable Schedule = equivalent to a serial one
- Conflict Serializable Schedule = …
- Precedence Graph = to check conflict serializability

Recap: the Precedence Graph

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



This schedule is NOT conflict-serializable

Note for HWs and Exams

Always draw the full graph, unless ONLY asked if (yes or no) the schedule is conflict serializable

Today's Agenda

- Concurrency control manager
- Locks
- 2PL
- Strict 2PL
- Deadlocks

Concurrency Control Manager

- Scheduler a.k.a. Concurrency Control Manager
  - The module that schedules the transaction's actions

Will discuss how

- Main goal: ensure the schedule is serializable

- Second goal: optimize for throughput

Concurrency Control Manager

Two types:

- Pessimistic CC Manager (Locks)

We discuss only this

- Optimistic CC Manager (e.g. Snapshot Isolation)

Locks

Locking Scheduler

- Each element has a unique lock
- Each TXN must acquire lock before R/W element
- If the lock is held by another TXN, then wait
- Once lock is available, it may proceed
- The TXN must release the lock(s)

TXN Actions

- $R_i(A)$ = transaction T_i reads element A
- $W_i(A)$ = transaction T_i reads element A
- $L_i(A)$ = transaction T_i acquires lock for element A
- $U_i(A)$ = transaction T_i releases lock for element A

Recap: 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)	

Let's see how locks can prevent this

Locks in Action

T1	T2
L1(A), **READ**(A, t)	

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	

Scheduler decides that T1 should wait now

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	

Scheduler decides that T1 should wait now

Why wait?

For various performance reasons:

- It takes long time to read B from disk, or
- T2 just arrived and has higher priority, or
- T2 was waiting for too long, or
- …

We want to allow the scheduler lots of freedom to schedule another TXN when it wants.

Our focus is only to **prevent** non-serializable schedules

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…

Denied: T2 put to sleep

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…

Denied: T2 put to sleep

This is the key step:
we stopped the scheduler from allowing T2 to read B at this time

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t)	

After a while, T1 is ready to continue

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	

Releases lock on B

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	
	READ(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)

T2 may proceed now

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	
	READ(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)

But there is a BIG problem! (what???)

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
L1(B)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	
	READ(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)

But there is a BIG problem! (what???)

Let's replay…

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	

Scheduler decided
to put T1 on wait
before it acquired L1(B)

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B)

Granted

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)
L1(B)	
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)
L1(B)	
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	

This is a non-serializable schedule

Locks in Action

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s),**U2**(A)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)
L1(B)	
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	

This is a non-serializable schedule

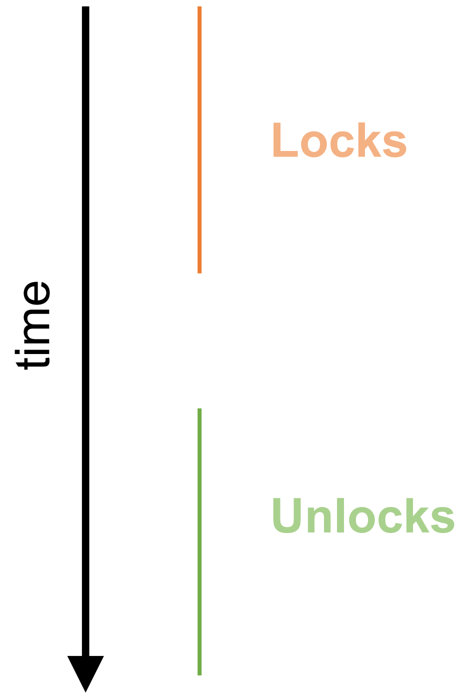
Solution: 2PL

2PL

Two-Phase Locking

The 2PL rule:

In every TXN, all locks must come before any unlock



Two-Phase Locking

Not 2PL

T1	
L1(A)	
READ(A, t)	
t := t+100	
WRITE(A, t)	
U1(A)	
L1(B)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
U1(B)	

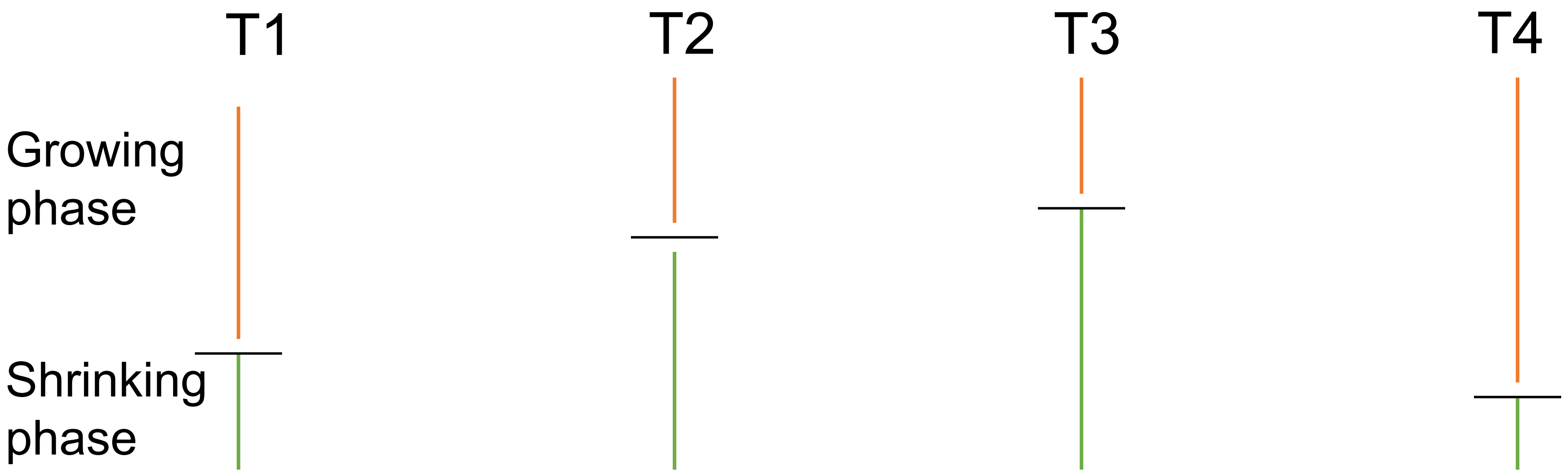
Two-Phase Locking

Not 2PL		2PL
T1		**T1**
L1(A)		**L1**(A)
READ(A, t)		**READ**(A, t)
t := t+100		t := t+100
WRITE(A, t)		**WRITE**(A, t)
U1(A)		**L1**(B)
L1(B)		**U1**(A)
READ(B, t)		**READ**(B, t)
t := t+100		t := t+100
WRITE(B,t)		**WRITE**(B,t)
U1(B)		**U1**(B)

Two-Phase Locking

Not 2PL		2PL
T1	T1	T1
L1(A)	**L1**(A)	**L1**(A)
READ(A, t)	**READ**(A, t)	**L1**(B)
t := t+100	t := t+100	**READ**(A, t)
WRITE(A, t)	**WRITE**(A, t)	t := t+100
U1(A)	**L1**(B)	**WRITE**(A, t)
L1(B)	**U1**(A)	**U1**(A)
READ(B, t)	**READ**(B, t)	**READ**(B, t)
t := t+100	t := t+100	t := t+100
WRITE(B,t)	**WRITE**(B,t)	**WRITE**(B,t)
U1(B)	**U1**(B)	**U1**(B)

Example with Multiple Transactions



Equivalent to each transaction executing entirely the **moment** it enters shrinking phase

Two-Phase Locking

T1	T2
L1(A), **L1**(B)	
READ(A, t)	
t := t+100	
WRITE(A, t),**U1**(A)	
	L2(A), **READ**(A, s)
	s := s*2
	WRITE(A,s)
	L2(B)…
READ(B, t)	
t := t+100	
WRITE(B,t),**U1**(B)	
	READ(B,s)
	s := s*2
	WRITE(B,s),**U2**(B)

Denied

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Two-Phase Locking

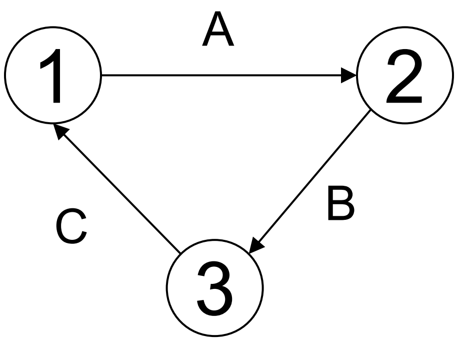
Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



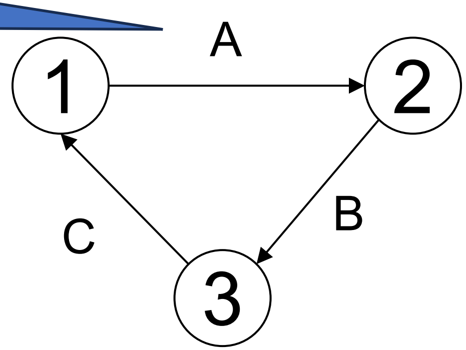
…R_1(A)… …W_2(A)…

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

Edge due to element A
E.g. …$R_1(A)$…$W_2(A)$…



…$R_1(A)$… …$W_2(A)$…

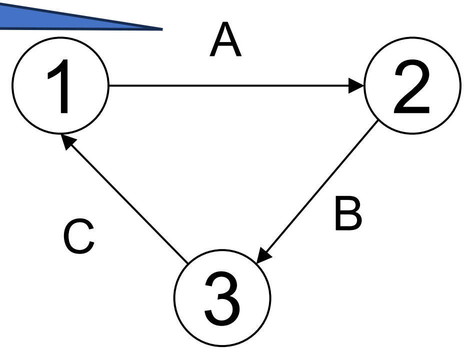
time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

Edge due to element A
E.g. …$R_1(A)$…$W_2(A)$…



T1 must release lock
before T2 can get the lock

…$R_1(A)$…$U_1(A)$…$L_2(A)$…$W_2(A)$…

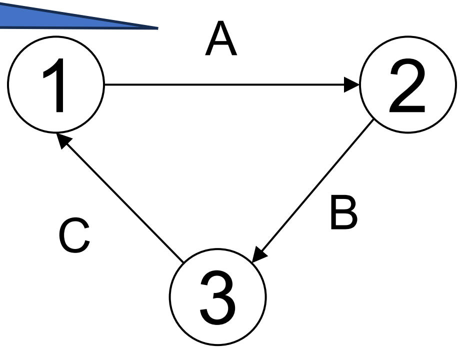
time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

Edge due to element A
E.g. …$R_1(A)$…$W_2(A)$…



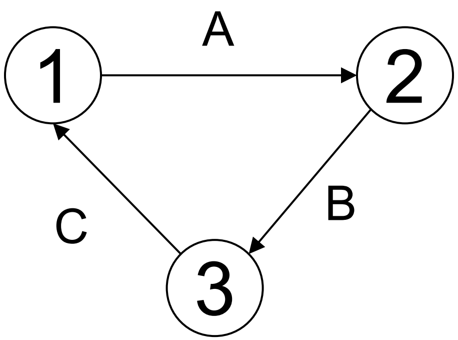
…$R_1(A)$…$U_1(A)$…$L_2(A)$…$W_2(A)$…

time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

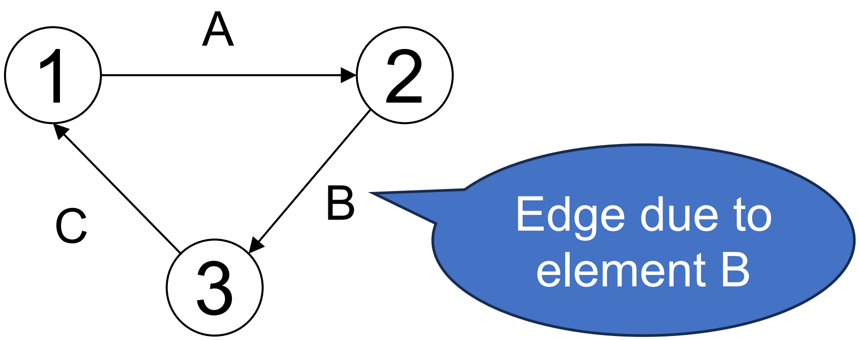


…$U_1(A)$…$L_2(A)$…

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



…U_1(A)…L_2(A)…

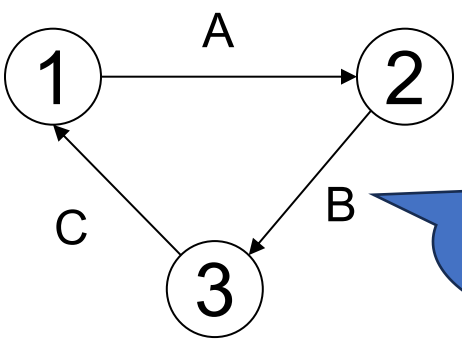
Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

WHY??

T2 releases lock on B **after** locking A



Edge due to element B

$\ldots U_1(A)\ldots L_2(A)\ldots U_2(B)\ldots$

time

Two-Phase Locking

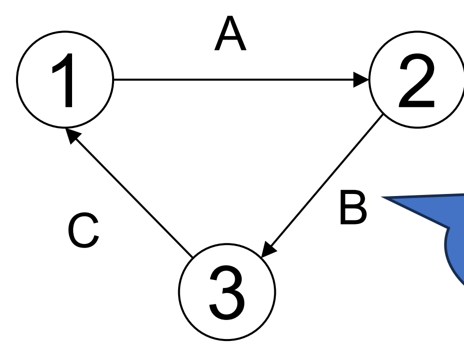
Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

WHY??

2PL!!

T2 releases lock on B
after locking A



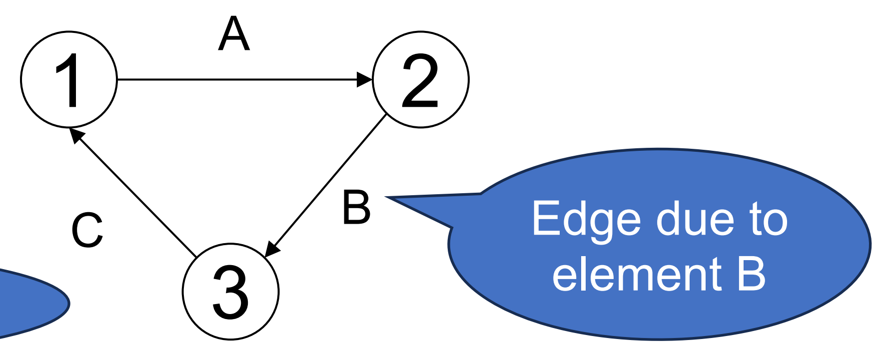
Edge due to element B

$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots$

time

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



WHY??

Comes **after** $U_2(B)$

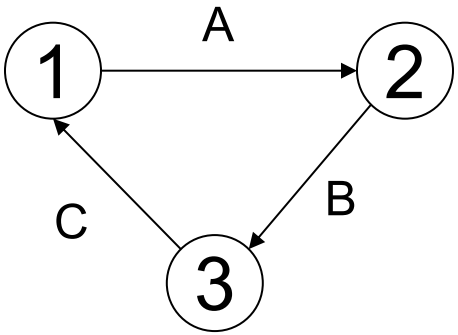
$\ldots U_1(A)\ldots L_2(A)\ldots U_2(B)\ldots L_3(B)$

time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

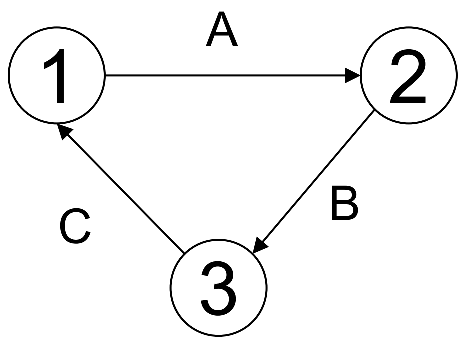


$\ldots U_1(A)\ldots L_2(A)\ldots U_2(B)\ldots L_3(B)\ldots U_3(C)\ldots$

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



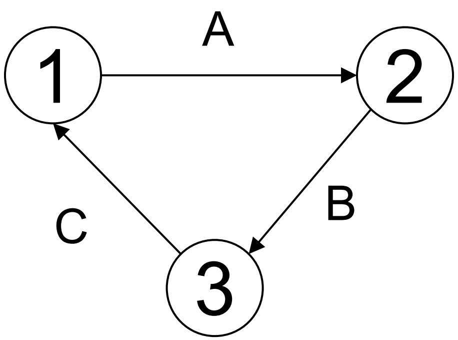
$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots L_3(B) \ldots U_3(C) \ldots L_1(C) \ldots$

time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



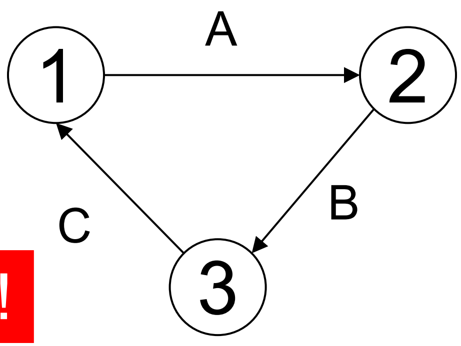
$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots L_3(B) \ldots U_3(C) \ldots L_1(C) \ldots U_1(A) \ldots$

time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle



Contradiction!

$\ldots U_1(A) \ldots L_2(A) \ldots U_2(B) \ldots L_3(B) \ldots U_3(C) \ldots L_1(C) \ldots U_1(A) \ldots$

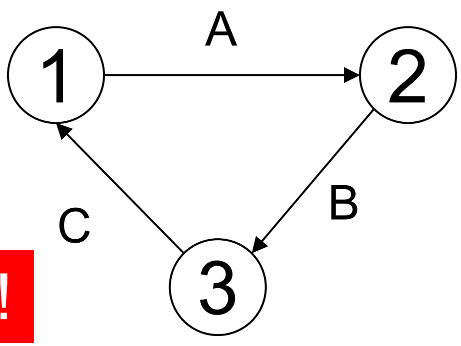
time

Two-Phase Locking

Theorem: If all TXNs follow 2PL, then schedule is conflict-serializable

Proof. Suppose precedence graph has a cycle

Precedence graph cannot have a cycle.
Schedule is conflict serializable.



Contradiction!

…$U_1(A)$…$L_2(A)$…$U_2(B)$…$L_3(B)$…$U_3(C)$…$L_1(C)$…$U_1(A)$…

time

Discussion

- Computers use locks in many places
- In databases, we need locks with the 2PL rule to guarantee conflict serializability
- However, 2PL fails to guarantee ”recoverability”

Strict 2PL

Rollback/Recovery

- If a TXN issues Rollback,
  then all its updates need to be undone

- If another TXN read those dirty values,
  then the system must abort that TXN as well

- But if the other TXN has already committed,
  then big problem!

Example

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	
.	
.	
.	
.	
.	
.	

Example

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	

Example

T1	T2
L1(A),L1(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), U1(A),U1(B)	
.	L2(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),U2(A)
.	L2(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),U2(B)
.	**COMMIT**

Takes the $$$ and leaves

Example

T1	T2
L1(A),L1(B),READ(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), U1(A),U1(B)	
.	L2(A), READ(A, s)
.	s := s*2
.	WRITE(A,s),U2(A)
.	L2(B), READ(B,s)
.	s := s*2
.	WRITE(B,s),U2(B)
.	COMMIT
ROLLBACK	

Takes the $$$ and leaves

Example

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	**COMMIT**
ROLLBACK	

Undo the writes to A and B

Takes the $$$ and leaves

Example

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	**COMMIT**
ROLLBACK	

All these reads were "dirty reads"

Undo the writes to A and B

Takes the $$$ and leaves

Example

T1	T2
L1(A),**L1**(B),**READ**(A, t)	
t := t+100	
WRITE(A, t),	
READ(B, t)	
t := t+100	
WRITE(B,t), **U1**(A),**U1**(B)	
.	**L2**(A), **READ**(A, s)
.	s := s*2
.	**WRITE**(A,s),**U2**(A)
.	**L2**(B), **READ**(B,s)
.	s := s*2
.	**WRITE**(B,s),**U2**(B)
.	**COMMIT**
ROLLBACK	

Unrecovarable schedule

All these reads were “dirty reads”

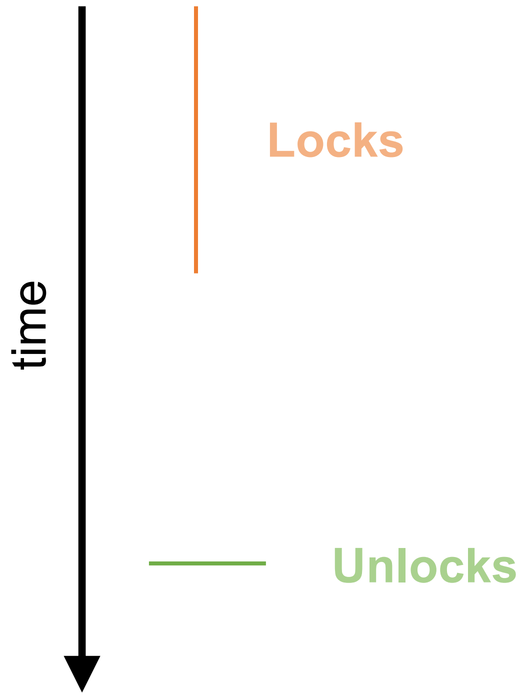
Takes the $$$ and leaves

Undo the writes to A and B

Strict Two Phase Locking

The Strict 2PL rule is:

All locks are released at Commit/Rollback time



Example Strict 2PL

Lock right before read

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t),	**L2**(A)…
L1(B),**READ**(B, t)	
t := t+100	
WRITE(B,t)	
ROLLBACK,**U1**(A),**U1**(B)	
	…. **READ**(A, s)
	s := s*2
	WRITE(A,s)
	L2(B), **READ**(B,s)
	s := s*2
	WRITE(B,s)
	COMMIT,**U2**(A),**U2**(B)

Denied

Granted

Example Strict 2PL

T1	T2
L1(A), **READ**(A, t)	
t := t+100	
WRITE(A, t)	
L1(B), **READ**(B, t)	
	L2(C), **READ**(C, s)
	s := s*2
	WRITE(C,s)
	L2(B),
t := t+100	
WRITE(B,t)	
COMMIT,**U1**(A),**U1**(B)	
	…**READ**(B,s)
	s := s*2
	WRITE(B,s)
	COMMIT,**U2**(A),**U2**(B)

Interleaving is possible; it depends on the conflicts

Strict Two Phase Locking

- If all TXN follow the Strict 2PL rule, then any schedule is conflict serializable and recoverable

- All RDBMS that use locking implement Strict 2PL:
  - When TXN wants to read or write, RDBMs inserts a Lock statement (unless TXN already has that lock)
  - When TXN commits or rolls back, RDBMs inserts all Unlock statements

- Locking (even Strict 2PL) can lead to deadlocks.

Deadlocks

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
…	…	…	…

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
…		…	…

Can’t make progress since locking phase is not complete for any TXN!

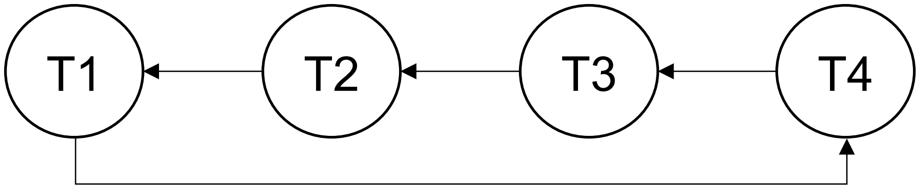
2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
…	…	…	…

Checking for deadlock:

- Construct the WAITS-FOR graph
- Check if it has a cycle

Checking for a cycle is fast (see CSE373), but it is very slow compared to the simple R/W operations



2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
…	…	…	…

If the DBMS finds a cycle:

- We rollback TXNs
- (Hopefully) make progress
- Notice: the app must always check if TXN was aborted

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…

2PL Deadlocks

T1 (A, B)	**T2 (B, C)**	**T3 (C, D)**	**T4 (D, A)**
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
			Abort, U(D)

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
			Abort, U(D)
		L(D)	

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
			Abort, U(D)
		L(D)	
		(do operations)	

2PL Deadlocks

T1 (A, B)	T2 (B, C)	T3 (C, D)	T4 (D, A)
L(A)	L(B)	L(C)	L(D)
L(B) blocked…			
	L(C) blocked…		
		L(D) blocked…	
			L(A) blocked…
			Abort, U(D)
		L(D)	
		(do operations)	
		Commit, U(C), U(D)	
	L(C)		

Discussion

- Supporting transactions usually incurs a high cost

- Performance is measured in TXN/sec (TPS)
  - 1,000-10,000 is OK
  - 10,000-100,000 is GREAT
  - 100,000-1,000,000 research papers only…

- For higher TPS: NoSQL databases
  - Distributed
  - Single operation TXN (no transfer from ACC1 to ACC2!)
  - Only for apps that can tolerate concurrency annomalies