

Introduction to Database Systems

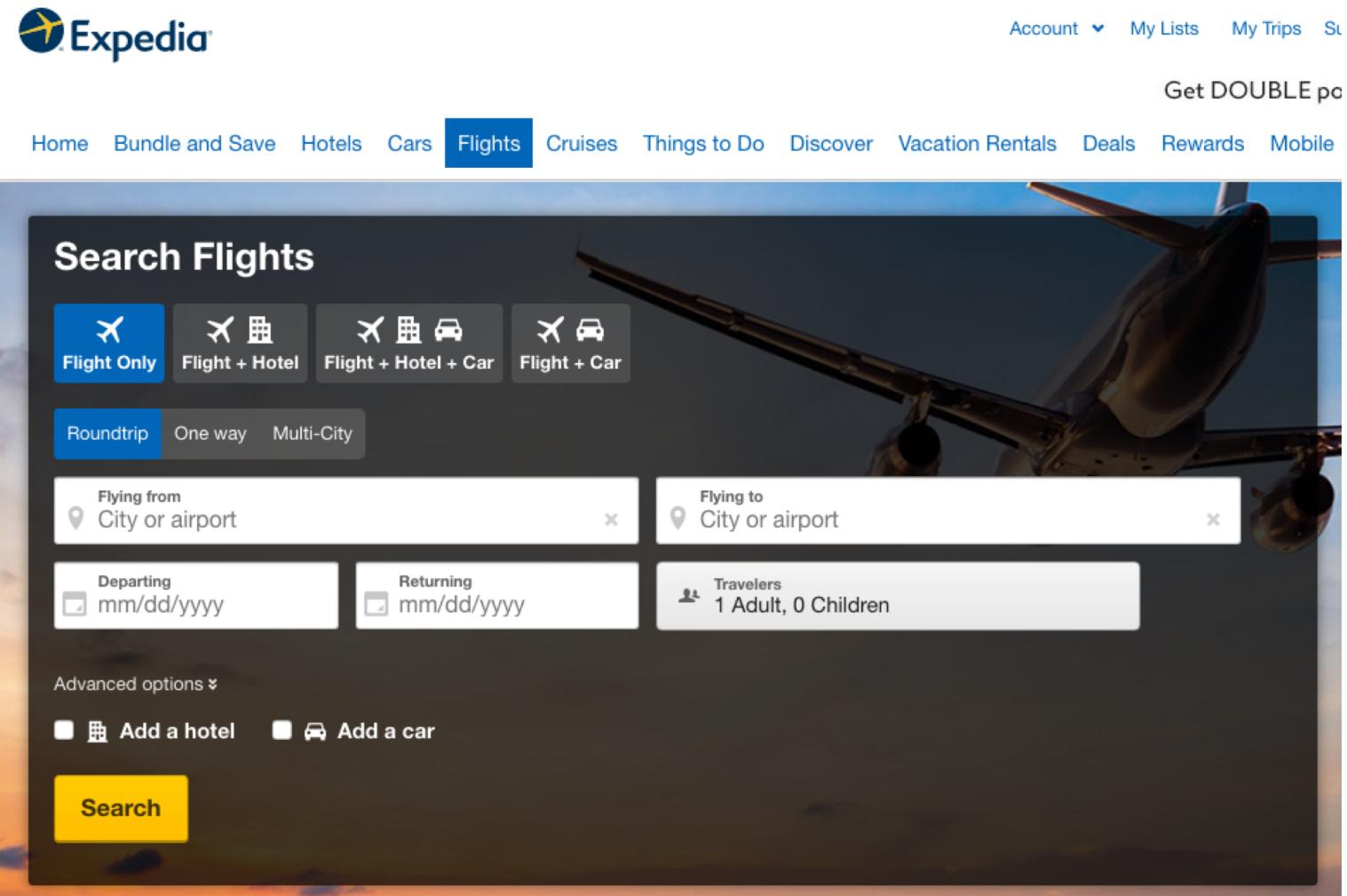
CSE 414

Lecture 26: More Transactions

Announcements

- Web quiz due tonight
- HW7 due tonight
- HW8 out, make sure to do setup early

HW8



The screenshot shows the Expedia website's flight search interface. At the top, there is a navigation bar with links for Account, My Lists, My Trips, and a search bar. Below the navigation bar, a banner offers "Get DOUBLE points". The main search form is titled "Search Flights" and features a background image of an airplane. It includes several search filters: "Flight Only" (selected), "Flight + Hotel", "Flight + Hotel + Car", and "Flight + Car". Below these are buttons for "Roundtrip", "One way", and "Multi-City". The search fields for "Flying from" and "Flying to" both have placeholder text "City or airport" and an "x" button to clear the input. There are also fields for "Departing" and "Returning" dates, each with a date input field and a calendar icon. A "Travelers" section indicates "1 Adult, 0 Children". At the bottom left, there is an "Advanced options" dropdown, and at the bottom right, there are checkboxes for "Add a hotel" and "Add a car". A large yellow "Search" button is positioned at the bottom center.

What can go wrong?

- Manager: balance budgets among projects
 - Remove \$10k from project A
 - Add \$7k to project B
 - Add \$3k to project C
- CEO: check company's total balance
 - `SELECT SUM(money) FROM budget;`
- This is called a dirty / inconsistent read
aka a **WRITE-READ** conflict

What can go wrong?

- App 1:

```
SELECT inventory FROM products WHERE pid = 1
```

- App 2:

```
UPDATE products SET inventory = 0 WHERE pid = 1
```

- App 1:

```
SELECT inventory * price FROM products  
WHERE pid = 1
```

- This is known as an unrepeatable read
aka **READ-WRITE** conflict

What can go wrong?

Account 1 = \$100

Account 2 = \$100

Total = \$200

- App 1:
 - Set Account 1 = \$200
 - Set Account 2 = \$0
- App 2:
 - Set Account 2 = \$200
 - Set Account 1 = \$0
- At the end:
 - Total = \$200
- App 1: Set Account 1 = \$200
- App 2: Set Account 2 = \$200
- App 1: Set Account 2 = \$0
- App 2: Set Account 1 = \$0
- At the end:
 - Total = \$0

This is called the lost update aka **WRITE-WRITE** conflict

What can go wrong?

- Buying tickets to the next Bieber concert:
 - Fill up form with your mailing address
 - Put in debit card number
 - Click submit
 - Screen shows money deducted from your account
 - [Your browser crashes]

Lesson:
Changes to the database
should be **ALL or NOTHING**

Transactions

- Collection of statements that are executed atomically (logically speaking)

```
BEGIN TRANSACTION  
[SQL statements]  
COMMIT      or  
ROLLBACK (=ABORT)
```

[single SQL statement]

If BEGIN... missing,
then TXN consists
of a single instruction

Know your ~~chemistry~~ 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)
user should not be able to tell that the transactions are being run at the same time
- **Durable**
 - Once a txn has committed, its effects remain in the database

Transaction Schedules

Schedules

A **schedule** is a sequence
of interleaved actions
from all transactions

Serial Schedule

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

no conflicts as we dont have separate transactions doing read/writes at same time
- **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 and B are elements
in the database
t and s are variables
in txn source code

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)

Example of 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)

Another Serial Schedule

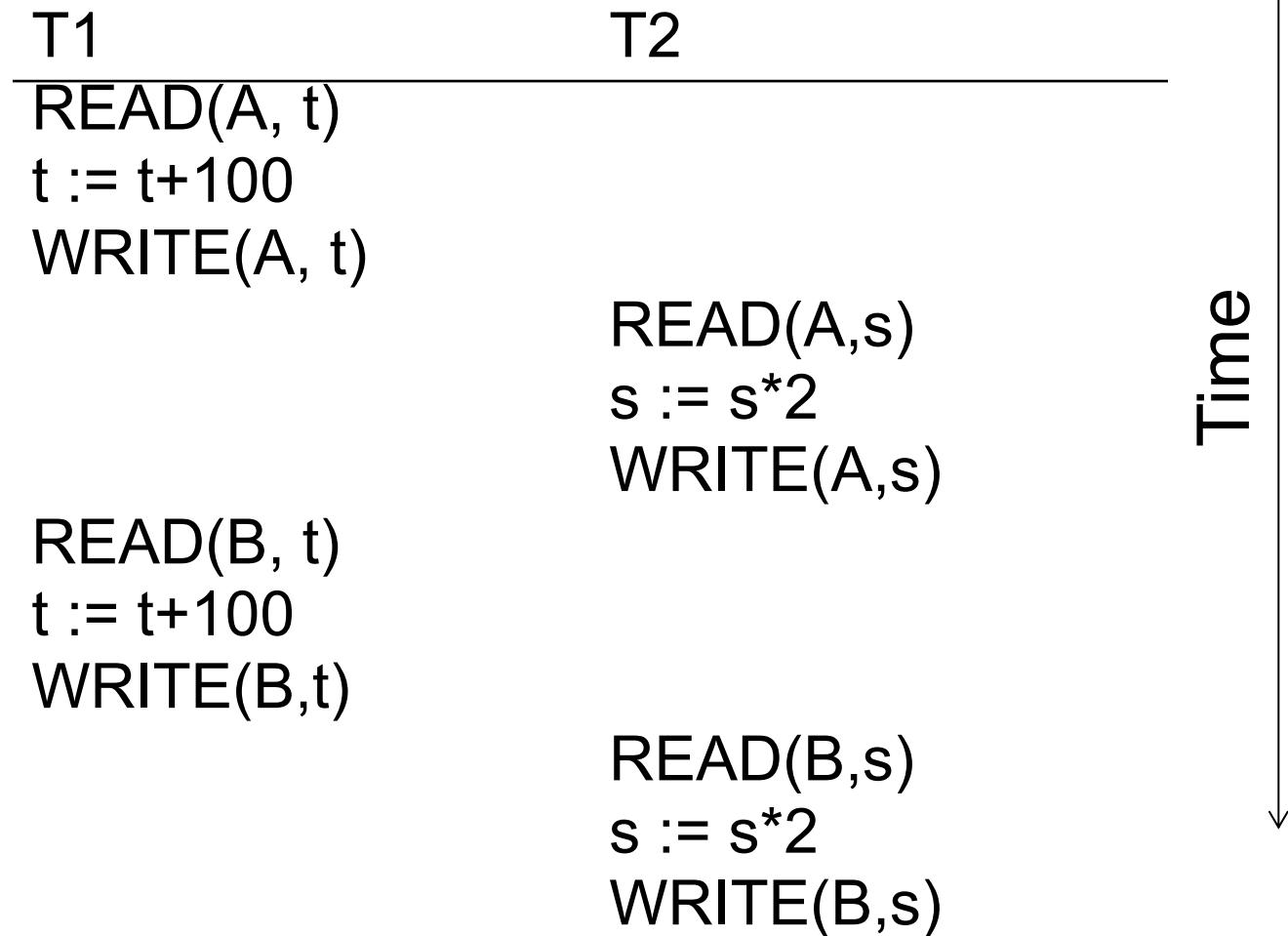
T1	T2
	READ(A,s)
	$s := s^*2$
	WRITE(A,s)
	READ(B,s)
	$s := s^*2$
	WRITE(B,s)
READ(A, t)	
$t := t+100$	
WRITE(A, t)	
READ(B, t)	
$t := t+100$	
WRITE(B,t)	

Review: Serializable Schedule

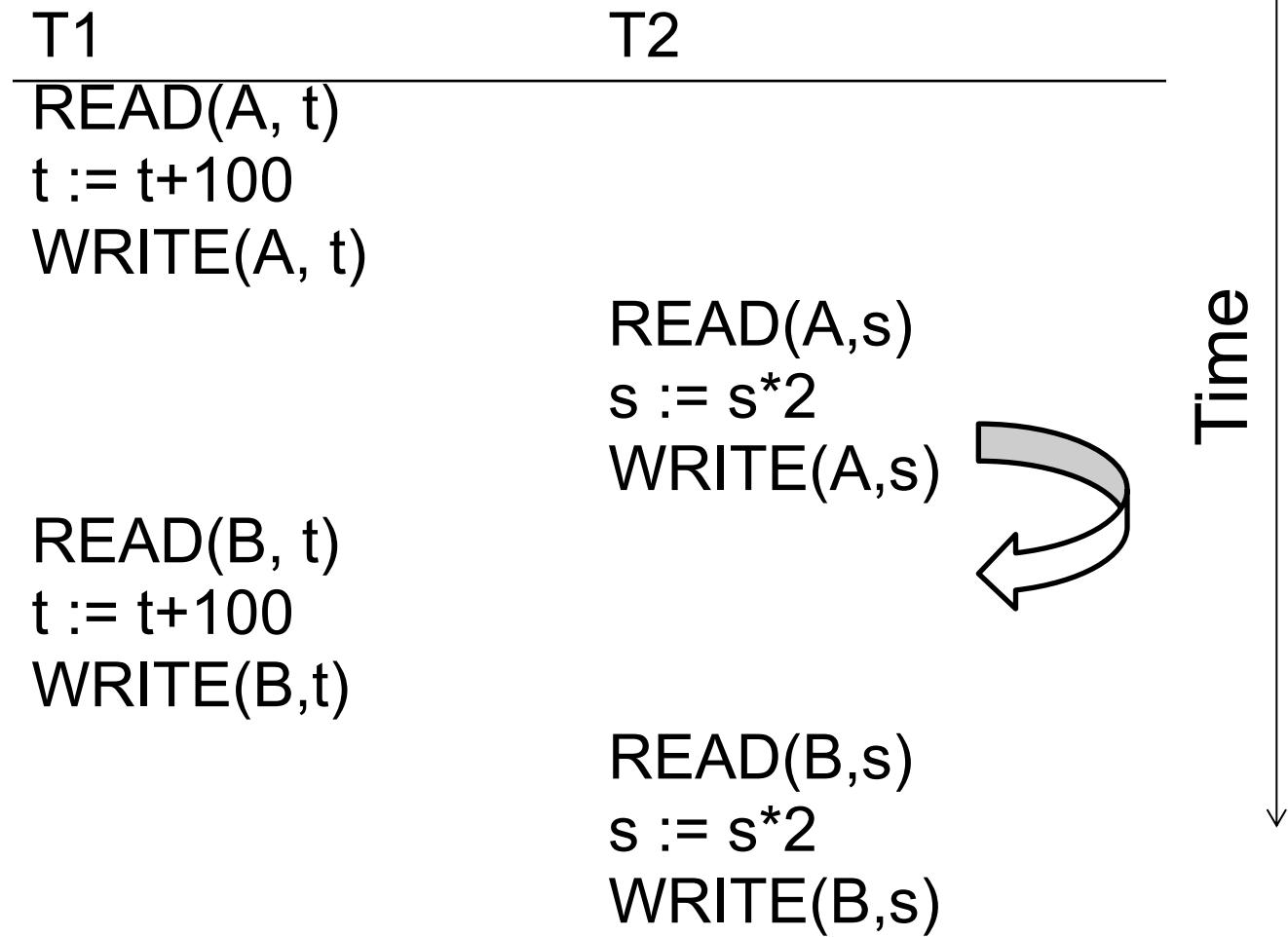
A schedule is **serializable** if it is equivalent to a serial schedule

outcome is EXACTLY the same as some serial schedule; i.e. no dirty reads/writes

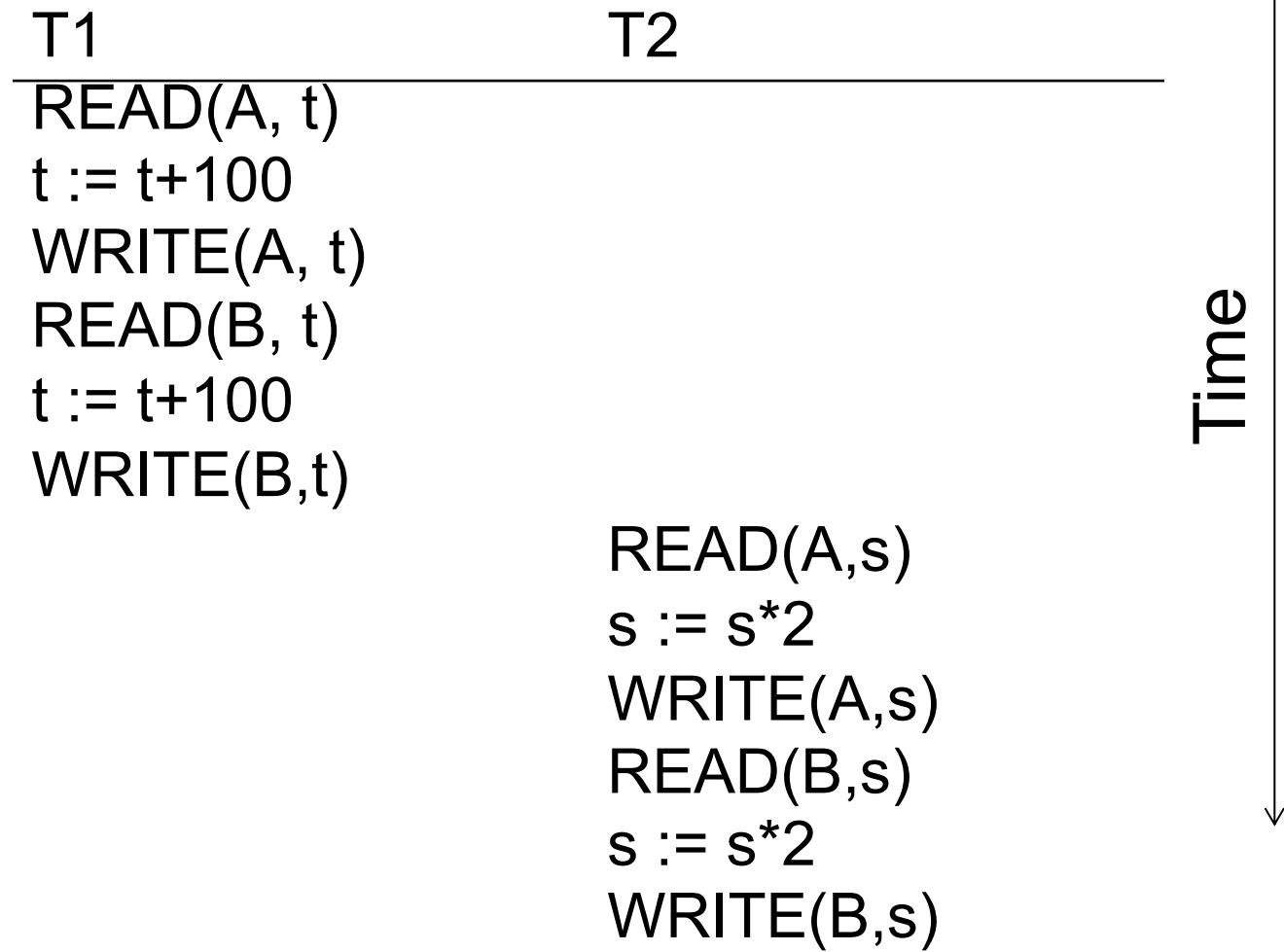
A Serializable Schedule



A Serializable Schedule



A Serializable Schedule



A Serializable Schedule

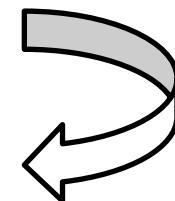
T1

READ(A, t)
t := t+100
WRITE(A, t)

T2

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)

This is a **Serializable** schedule.
This is NOT a serial schedule

Time

A Non-S 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)	

A Non-Serializable Schedule

A circular road sign with a thick red border and a diagonal red slash from top-left to bottom-right, indicating that a turn is prohibited.

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

T2

i.e. changing order of operations here changes outcome; adding by 100 first then multiplying by 2 creates different result than multiplying first before adding!

```
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)
```

How do We Know if a Schedule is Serializable?

Notation:

$$\begin{aligned} T_1 &: r_1(A); w_1(A); r_1(B); w_1(B) \\ T_2 &: r_2(A); w_2(A); r_2(B); w_2(B) \end{aligned}$$

Key Idea: Focus on *conflicting* operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW
- Read-Read?

Conflict Serializability

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

EVERYTHING ELSE IS SWAPPABLE besides these three!

Two actions by same transaction T_i :

$r_i(X); w_i(Y)$

CANT SWAP THESE; order of transaction is fixed. User wrote the query expecting it to be executed in a specific order.

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?)
 - Conflict serializable only looks at conflicts, not values
 - Schedules might have conflicts but would have the same output no matter the order depending on the values

Conflict Serializability

Example:

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

Conflict Serializability

Example:

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



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

Conflict Serializability

Example:

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



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

Conflict Serializability

Example:

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

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

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

Conflict Serializability

Example:

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

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

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

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

GOAL: all operations in one happens before 2.

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i ,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j
- The schedule is conflict-serializable iff the precedence graph is acyclic

Example 1

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

1

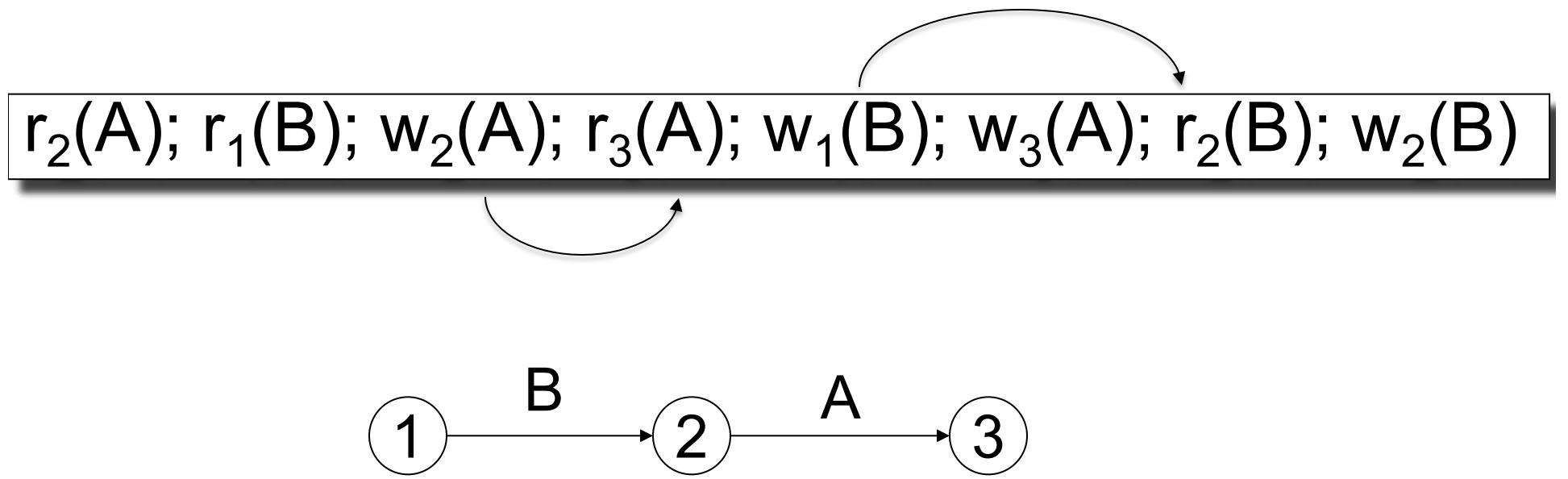
2

3

start at far left element ($r_2(A)$ in this case)
Compare it to all elements to the right of it

if they come from a different transaction AND
operate on same element (i.e. both operate on A) AND
are not both reads: then you have a conflict:
draw an arrow from transaction number of first element to second transaction number of second element and label with
the element they operate on (A)
Repeat for all transactions

Example 1



As long as all of T1 happens before T2 and T2 before T3 we will be good

This schedule is **conflict-serializable**

Then we can do the swaps to make it serial; i.e. T1 is always able to operate first on the elements that T2 wants to operate on, even though the two are run concurrently

Example 2

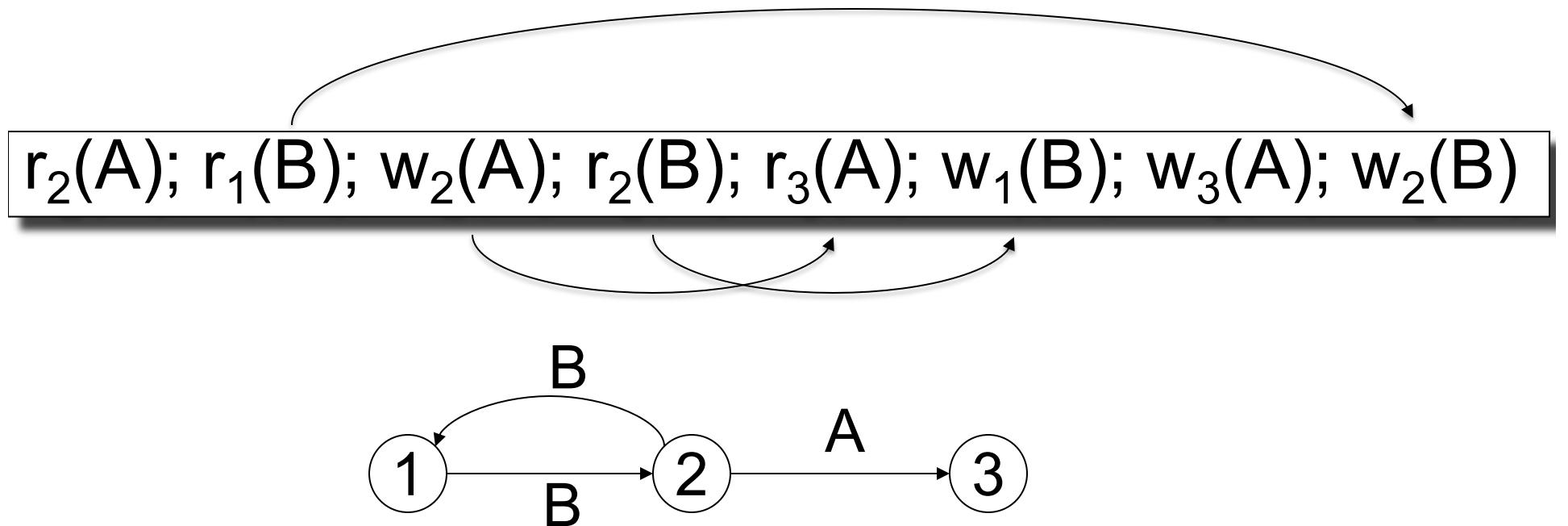
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

1

2

3

Example 2



This schedule is NOT conflict-serializable

Implementing Transactions

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

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-S 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

$L_1(A)$; READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$; $L_1(B)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$;

T2

$L_2(A)$; READ(A)

$A := A^*2$

WRITE(A); $U_2(A)$;

$L_2(B)$; BLOCKED...

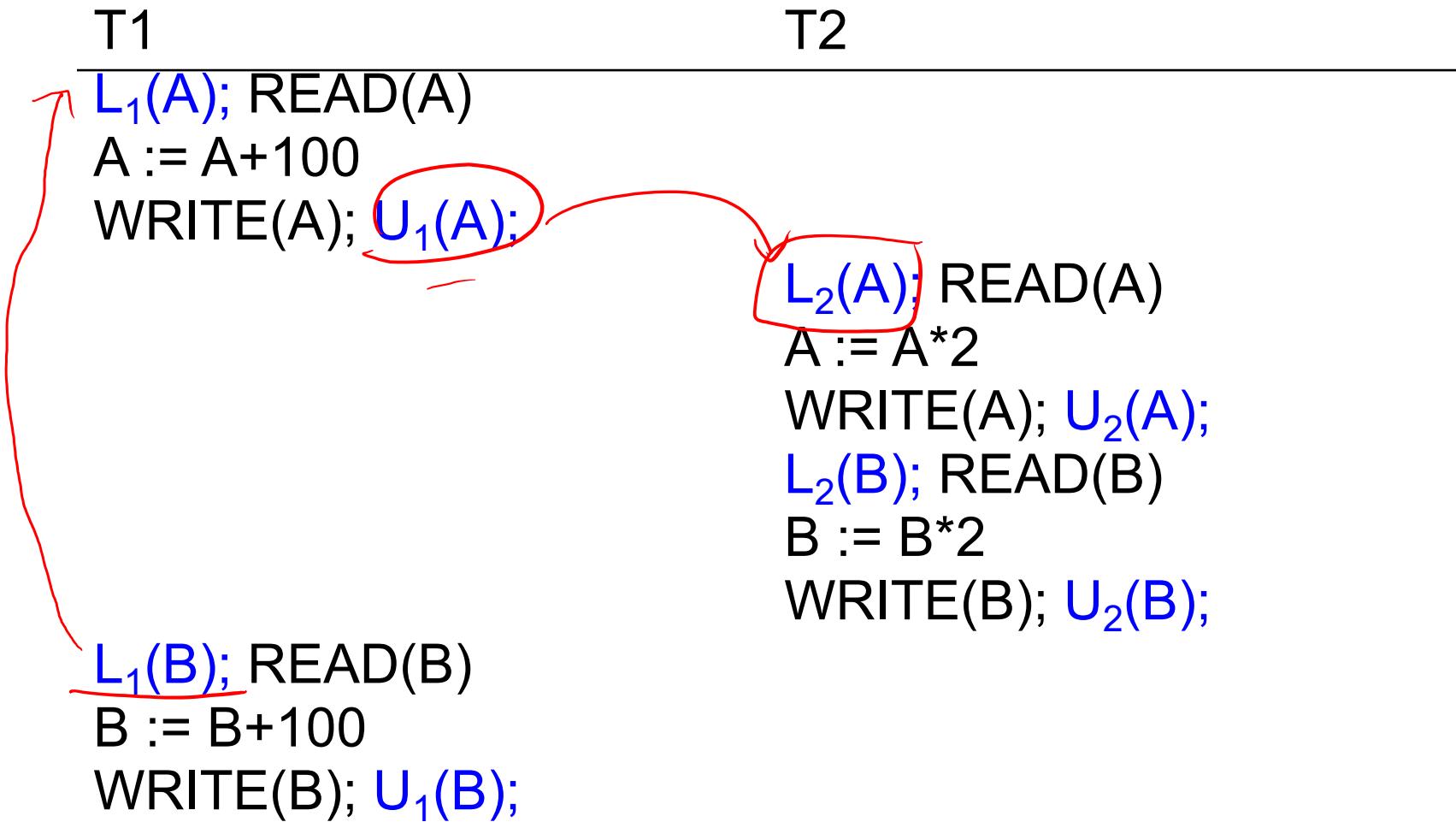
...GRANTED; READ(B)

$B := B^*2$

WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

But what if...



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

$L_1(A); L_1(B);$ READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$

T2

$L_2(A);$ READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$ BLOCKED...

...GRANTED; READ(B)

$B := B * 2$

WRITE(B); $U_2(A); U_2(B)$

Now it is conflict-serializable

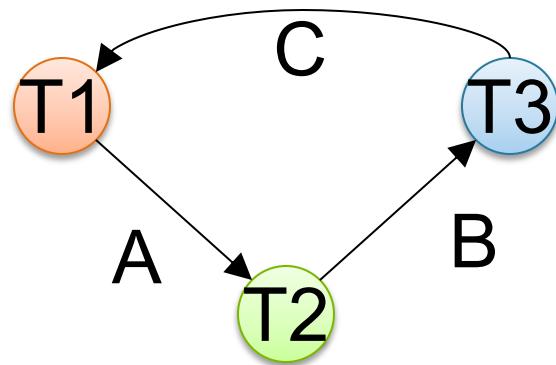
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

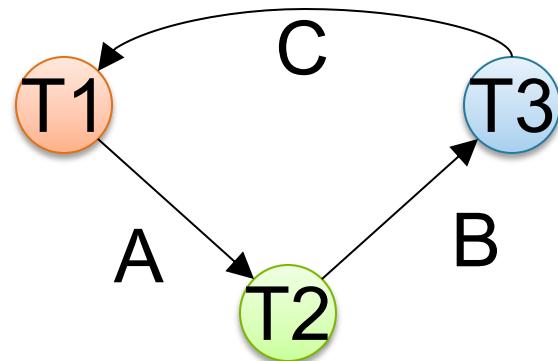
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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

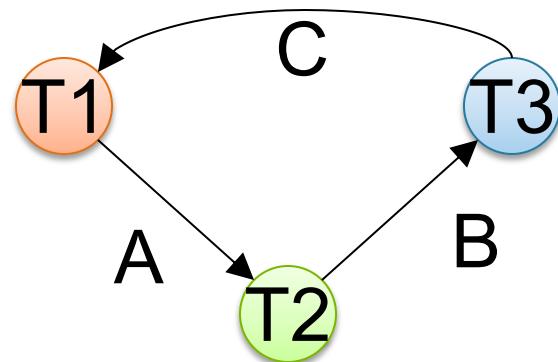


Then there is the following temporal cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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



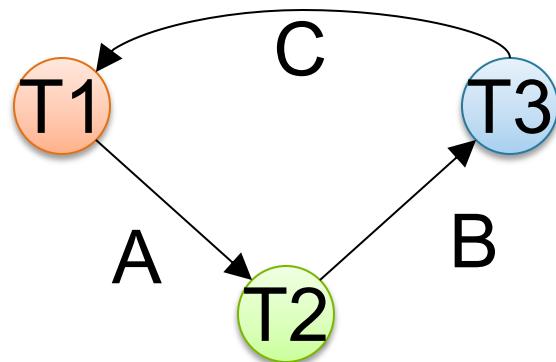
Then there is the following temporal cycle in the schedule:
 $U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ 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:

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

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

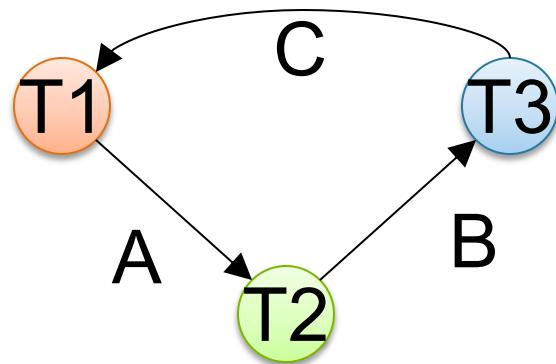
why?

L₂(A) happened strictly before U₁(A)

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

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



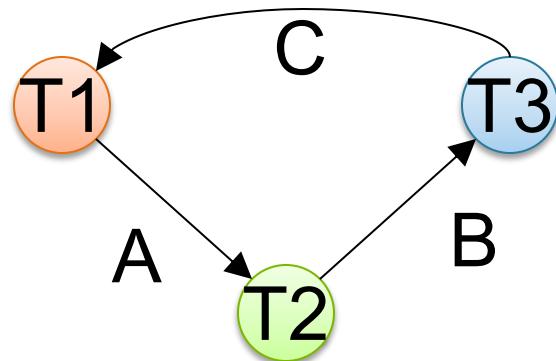
Then there is the following temporal cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$
 $L_2(A) \rightarrow U_2(B)$ 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:

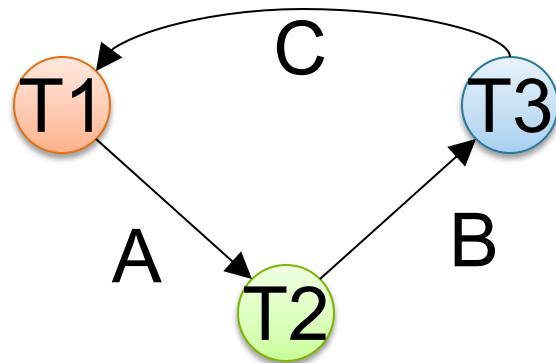
$$\begin{aligned} U_1(A) &\rightarrow L_2(A) \\ L_2(A) &\rightarrow U_2(B) \\ U_2(B) &\rightarrow L_3(B) \end{aligned}$$

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:

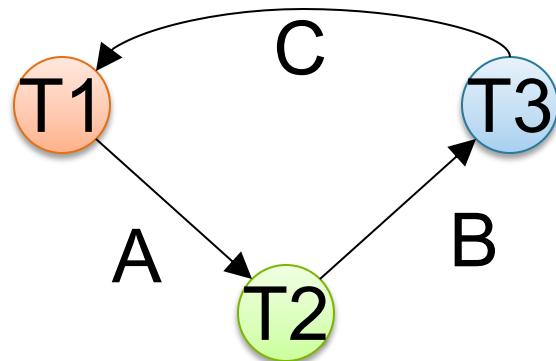
$$\begin{aligned} U_1(A) &\rightarrow L_2(A) \\ L_2(A) &\rightarrow U_2(B) \\ U_2(B) &\rightarrow L_3(B) \end{aligned}$$

.....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 temporal cycle in the schedule:

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

Cycle in time:
Contradiction

A New Problem: Non-recoverable Schedule

T1

$L_1(A); L_1(B);$ READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$

Rollback

T2

$L_2(A);$ READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$ BLOCKED...

...GRANTED; READ(B)

$B := B * 2$

WRITE(B); $U_2(A); U_2(B);$

Commit

A New Problem: Non-recoverable Schedule

T1

$L_1(A); L_1(B);$ READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$

T2

$L_2(A);$ READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$ BLOCKED...

...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

$L_1(A); L_1(B);$ READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$

Rollback

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

T2

$L_2(A);$ READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$ BLOCKED...

Dirty reads of
A, B lead to
incorrect writes.

...GRANTED; READ(B)

$B := B * 2$

WRITE(B); $U_2(A); U_2(B);$

Commit

A New Problem: Non-recoverable Schedule

T1

$L_1(A); L_1(B);$ READ(A)

$A := A + 100$

WRITE(A); $U_1(A)$

READ(B)

$B := B + 100$

WRITE(B); $U_1(B)$

Rollback

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

T2

$L_2(A);$ READ(A)

$A := A * 2$

WRITE(A);

$L_2(B);$ BLOCKED...

Dirty reads of
A, B lead to
incorrect writes.

...GRANTED; READ(B)

$B := B * 2$

WRITE(B); $U_2(A); U_2(B);$

Commit

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

L₁(A); READ(A)

A := A + 100

WRITE(A);

L₁(B); READ(B)

B := B + 100

WRITE(B);

Rollback & U₁(A); U₁(B);

T2

L₂(A); BLOCKED...

... GRANTED; READ(A)

A := A * 2

WRITE(A);

L₂(B); READ(B)

B := B * 2

WRITE(B);

Commit & U₂(A); U₂(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