Announcements

• Web quiz due tonight
• HW7 due tonight

• HW8 out, make sure to do setup early
HW8
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)
```

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)

- **Durable**
  - Once a txn has committed, its effects remain in the database
Transaction 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.

- **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
### Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

A and B are elements in the database
t and s are variables in txn source code
Example of a (Serial) Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td>WRITE(B, s)</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td></td>
</tr>
</tbody>
</table>

CSE 414 - Autumn 2018
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.
A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>
### A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ(A, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td></td>
<td>READ(B, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>

CSE 414 - Autumn 2018
A Serializable Schedule

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

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

T2
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.
### A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td>READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
</tr>
</tbody>
</table>
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
</tbody>
</table>

CSE 414 - Autumn 2018
How do We Know if a Schedule is Serializable?

Notation:

\[ T_1: r_1(A); w_1(A); r_1(B); w_1(B) \]
\[ T_2: r_2(A); w_2(A); r_2(B); w_2(B) \]

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)

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?)
  – 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:

\[
\begin{align*}
& r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \\
& r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)
\end{align*}
\]
Conflict Serializability

Example:

\[ r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B) \]

\[ r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \]

\[ \ldots \]

CSE 414 - Autumn 2018
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

\[ r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B) \]
Example 1

This schedule is conflict-serializable

r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)
Example 2

\[ r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B) \]
Example 2

This schedule is NOT conflict-serializable

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

\( L_i(A) = \text{transaction } T_i \text{ acquires lock for element } A \)

\( U_i(A) = \text{transaction } T_i \text{ releases lock for element } A \)
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B*2</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B)</td>
<td></td>
</tr>
</tbody>
</table>
Example

\[
\begin{array}{l}
T1 \\
L_1(A); \text{READ}(A) \\
A := A+100 \\
\text{WRITE}(A); \ U_1(A); \ L_1(B) \\
\end{array}
\begin{array}{l}
T2 \\
L_2(A); \text{READ}(A) \\
A := A*2 \\
\text{WRITE}(A); \ U_2(A); \\
L_2(B); \text{BLOCKED}…
\end{array}
\]

\[
\begin{array}{l}
\text{READ}(B) \\
B := B+100 \\
\text{WRITE}(B); \ U_1(B); \\
\end{array}
\begin{array}{l}
\text{…GRANTED}; \text{READ}(B) \\
B := B*2 \\
\text{WRITE}(B); \ U_2(B); \\
\end{array}
\]

Scheduler has ensured a conflict-serializable schedule
But what if…

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$</td>
<td>$L_2(A)$</td>
</tr>
<tr>
<td>READ(A)</td>
<td>READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A)</td>
<td>WRITE(A)</td>
</tr>
<tr>
<td></td>
<td>$U_1(A)$</td>
</tr>
<tr>
<td></td>
<td>$U_2(A)$</td>
</tr>
<tr>
<td>$L_1(B)$</td>
<td>$L_2(B)$</td>
</tr>
<tr>
<td>READ(B)</td>
<td>READ(B)</td>
</tr>
<tr>
<td>B := B+100</td>
<td>B := B*2</td>
</tr>
<tr>
<td>WRITE(B)</td>
<td>WRITE(B)</td>
</tr>
<tr>
<td></td>
<td>$U_1(B)$</td>
</tr>
<tr>
<td></td>
<td>$U_2(B)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_1(A); L_1(B); READ(A)</td>
<td>L_2(A); READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U_1(A)</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>L_2(B); BLOCKED…</td>
</tr>
<tr>
<td>B := B+100</td>
<td>…GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); U_1(B);</td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U_2(A); U_2(B);</td>
</tr>
</tbody>
</table>

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.

![Graph showing T1, T2, T3, A, B, and C with arrows indicating precedence]

- T1 has arrows to C and A
- T2 has arrows to B and C
- T3 has arrows to C
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?

$L_2(A)$ 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 **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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A)</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); BLOCKED…</td>
</tr>
<tr>
<td>READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); U₁(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...GRANTED; READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B * 2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U₂(A); U₂(B);</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

Rollback
A New Problem: Non-recoverable Schedule

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

READ(B)
B := B+100
WRITE(B); U₁(B);

T2
L₂(A); READ(A)
A := A*2
WRITE(A);
L₂(B); BLOCKED…

…GRANTED; READ(B)
B := B*2
WRITE(B); U₂(A); U₂(B);
Commit

Rollback
Elements A, B written by T1 are restored to their original value.
A New Problem: Non-recoverable Schedule

T1

L₁(A); L₁(B); READ(A)
A := A + 100
WRITE(A); U₁(A)

READ(B)
B := B + 100
WRITE(B); U₁(B);

T2

L₂(A); READ(A)
A := A * 2
WRITE(A);
L₂(B); BLOCKED…

…GRANTED; READ(B)
B := B * 2
WRITE(B); U₂(A); U₂(B);
Commit

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); READ(A)</td>
<td>L₂(A); READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); U₁(A)</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>READ(B)</td>
<td>L₂(B); BLOCKED…</td>
</tr>
<tr>
<td>B := B+100</td>
<td>...GRANTED; READ(B)</td>
</tr>
<tr>
<td>WRITE(B); U₁(B)</td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); U₂(A); U₂(B);</td>
</tr>
</tbody>
</table>

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

Can no longer undo!

Dirty reads of A, B lead to incorrect writes.
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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); ) READ(A)</td>
<td>( L_2(A); ) BLOCKED…</td>
</tr>
<tr>
<td>A := A+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>( L_1(B); ) READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td>( \ldots )GRANTED; READ(A)</td>
</tr>
<tr>
<td>WRITE(B);</td>
<td>A := A*2</td>
</tr>
<tr>
<td>Rollback &amp; U_1(A); U_1(B);</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td>( L_2(B); ) READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B);</td>
</tr>
<tr>
<td></td>
<td>Commit &amp; U_2(A); U_2(B);</td>
</tr>
</tbody>
</table>
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