Chapter 18  Concurrency Control
Example:

T1:  Read(A)  
     A ← A+100  
     Write(A)  
     Read(B)  
     B ← B+100  
     Write(B)  

T2:  Read(A)  
     A ← A×2  
     Write(A)  
     Read(B)  
     B ← B×2  
     Write(B)  

Constraint:  A=B
## Schedule A

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A + 100</td>
<td>Write(A);</td>
</tr>
<tr>
<td>Write(A);</td>
<td>125</td>
</tr>
<tr>
<td>Read(B); B ← B + 100;</td>
<td>Write(B);</td>
</tr>
<tr>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A × 2;</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B × 2;</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
</tr>
<tr>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Schedule B

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×2;</td>
<td></td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Read(B); B ← B×2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>

CS 277 – Spring 2002
Notes 09 5
Schedule C

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); A ← A + 100</td>
<td>Read(A); A ← A × 2;</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B + 100;</td>
<td>Read(B); B ← B × 2;</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>Write(B);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Schedule D

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td>Read(A); A ← A×2;</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>Write(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>
### Schedule E

Same as Schedule D but with new T2’

<table>
<thead>
<tr>
<th>T1</th>
<th>T2’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A + 100</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A × 1;</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B + 100;</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
</tr>
<tr>
<td>Read(B); B ← B + 100;</td>
<td>125</td>
</tr>
<tr>
<td>Write(B);</td>
<td>125</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td>25</td>
</tr>
<tr>
<td>125</td>
<td>125</td>
</tr>
</tbody>
</table>
• Want schedules that are “good”, regardless of
  – initial state and
  – transaction semantics
• Only look at order of read and writes

Example:
Sc=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)
Example:

\[ S_c = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B) \]

\[ S'_c = r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B) \]
However, for $S_d$:

$$S_d = r_1(A)w_1(A)r_2(A)w_2(A)\, r_2(B)w_2(B)r_1(B)w_1(B)$$

• as a matter of fact,
  $T_2$ must precede $T_1$
  in any equivalent schedule,
  i.e., $T_2 \rightarrow T_1$
• $T_2 \rightarrow T_1$

• Also, $T_1 \rightarrow T_2$

$T_1 \leftrightarrow T_2 \implies Sd$ cannot be rearranged into a serial schedule

$\iff Sd$ is not "equivalent" to any serial schedule

$\iff Sd$ is "bad"
Returning to Sc

Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)

T_1 \rightarrow T_2 \quad T_1 \rightarrow T_2

E no cycles \Rightarrow Sc \text{ is “equivalent” to a serial schedule}
(in this case T_1, T_2)
Concepts

Transaction: sequence of \( r_i(x), w_i(x) \) actions

Conflicting actions: \( r_1(A) \rightleftharpoons W_2(A) \rightleftharpoons W_1(A) \)

\( W_2(A) \rightleftharpoons r_1(A) \rightleftharpoons W_2(A) \)

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions
What about concurrent actions?

Ti issues read(x,t)
System issues input(x)
Input(X) completes
System issues output(B)
output(B) completes

T2 issues write(B,S)
System issues input(B)
input(B) completes
System issues output(B)
output(B) completes

t ← x
So net effect is either

- $S = \ldots r_1(x) \ldots w_2(b) \ldots$ or
- $S = \ldots w_2(B) \ldots r_1(x) \ldots$
What about conflicting, concurrent actions on same object?

- Assume equivalent to either $r_1(A) \ w_2(A)$ or $w_2(A) \ r_1(A)$
- $\Rightarrow$ low level synchronization mechanism
- Assumption called “atomic actions”
Definition

$S_1$, $S_2$ are conflict equivalent schedules if $S_1$ can be transformed into $S_2$ by a series of swaps on non-conflicting actions.
Definition

A schedule is **conflict serializable** if it is conflict equivalent to some serial schedule.
Precedence graph $P(S)$ ($S$ is schedule)

Nodes: transactions in $S$

Arcs: $Ti \rightarrow Tj$ whenever

- $p_i(A), q_j(A)$ are actions in $S$
- $p_i(A) <_S q_j(A)$
- at least one of $p_i, q_j$ is a write
Exercise:

• What is P(S) for
  \[ S = \text{w}_3(A) \text{w}_2(C) \text{r}_1(A) \text{w}_1(B) \text{r}_1(C) \text{w}_2(A) \text{r}_4(A) \text{w}_4(D) \]

• Is S serializable?
Lemma

$S_1, S_2$ conflict equivalent $\Rightarrow P(S_1) = P(S_2)$

Proof:
Assume $P(S_1) \neq P(S_2)$
$\Rightarrow \exists T_i: T_i \rightarrow T_j$ in $S_1$ and not in $S_2$
$\Rightarrow S_1 = ...p_i(A)... q_j(A)...$
$S_2 = ...q_j(A)...p_i(A)...$
\begin{align*}
\{ \text{pi, qj} \} \\
\{ \text{conflict} \}
\end{align*}$
$\Rightarrow S_1, S_2$ not conflict equivalent
Note: \( P(S_1) = P(S_2) \nRightarrow S_1, S_2 \) conflict equivalent

Counter example:

\[
S_1 = w_1(A) \ r_2(A) \quad w_2(B) \ r_1(B)
\]

\[
S_2 = r_2(A) \ w_1(A) \quad r_1(B) \ w_2(B)
\]
Theorem

\[ P(S_1) \text{ acyclic} \iff S_1 \text{ conflict serializable} \]

(\(\iff\)) Assume \(S_1\) is conflict serializable
\(\Rightarrow\) \(\exists S_s: S_s, S_1\) conflict equivalent
\(\Rightarrow\) \(P(S_s) = P(S_1)\)
\(\Rightarrow\) \(P(S_1)\) acyclic since \(P(S_s)\) is acyclic
Theorem

\[ P(S_1) \text{ acyclic } \iff S_1 \text{ conflict serializable} \]

(\Rightarrow) Assume \( P(S_1) \) is acyclic

Transform \( S_1 \) as follows:

1. Take \( T_1 \) to be transaction with no incident arcs
2. Move all \( T_1 \) actions to the front
   \[ S_1 = \ldots \ q_j(A) \ldots \ p_1(A) \ldots \]
3. we now have \( S_1 = < T_1 \text{ actions } > < \ldots \text{ rest } \ldots > \)
4. repeat above steps to serialize rest!
How to enforce serializable schedules?

Option 1: run system, recording P(S); at end of day, check for P(S) cycles and declare if execution was good
How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring

$$T_1 \quad T_2 \quad \ldots \quad T_n$$

Scheduler

DB
A locking protocol

Two new actions:

lock (exclusive): \( l_i (A) \)

unlock: \( u_i (A) \)
Rule #1: Well-formed transactions

\[ T_i: \ldots li(A) \ldots pi(A) \ldots ui(A) \ldots \]
Rule #2  Legal scheduler

\[ S = \ldots \, l_i(A) \, \ldots \ldots \ldots \, u_i(A) \, \ldots \ldots \, \text{no } l_j(A) \]
Exercise:

- What schedules are legal?
- What transactions are well-formed?

S1 = \text{l}_1(A)\text{l}_1(B)\text{r}_1(A)\text{w}_1(B)\text{l}_2(B)\text{u}_1(A)\text{u}_1(B)\text{r}_2(B)\text{w}_2(B)\text{u}_2(B)\text{l}_3(B)\text{r}_3(B)\text{u}_3(B)

S2 = \text{l}_1(A)\text{r}_1(A)\text{w}_1(B)\text{u}_1(A)\text{u}_1(B)\text{l}_2(B)\text{r}_2(B)\text{w}_2(B)\text{l}_3(B)\text{r}_3(B)\text{u}_3(B)

S3 = \text{l}_1(A)\text{r}_1(A)\text{u}_1(A)\text{l}_1(B)\text{w}_1(B)\text{u}_1(B)\text{l}_2(B)\text{r}_2(B)\text{w}_2(B)\text{u}_2(B)\text{l}_3(B)\text{r}_3(B)\text{u}_3(B)
Exercise:

- What schedules are legal?
- What transactions are well-formed?

\[ S_1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B) \]
\[ r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B) \]

\[ S_2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B) \]
\[ l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B) \]

\[ S_3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B) \]
\[ l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B) \]
## Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_1(A);\text{Read}(A))</td>
<td>(l_1(B);\text{Read}(B))</td>
</tr>
<tr>
<td>(A \leftarrow A + 100; \text{Write}(A); u_1(A))</td>
<td>(l_2(A);\text{Read}(A))</td>
</tr>
<tr>
<td></td>
<td>(A \leftarrow Ax2; \text{Write}(A); u_2(A))</td>
</tr>
<tr>
<td></td>
<td>(l_2(B);\text{Read}(B))</td>
</tr>
<tr>
<td></td>
<td>(B \leftarrow Bx2; \text{Write}(B); u_2(B))</td>
</tr>
<tr>
<td></td>
<td>(l_1(B);\text{Read}(B))</td>
</tr>
<tr>
<td></td>
<td>(B \leftarrow B + 100; \text{Write}(B); u_1(B))</td>
</tr>
</tbody>
</table>
## Schedule F

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A ← A + 100; Write(A); u₁(A)</td>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>l₂(A); Read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A ← Ax₂; Write(A); u₂(A)</td>
<td>125</td>
<td>250</td>
</tr>
<tr>
<td>l₂(B); Read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B ← Bx₂; Write(B); u₂(B)</td>
<td>250</td>
<td>50</td>
</tr>
<tr>
<td>l₁(B); Read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B ← B + 100; Write(B); u₁(B)</td>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
</tr>
</tbody>
</table>

A | B
---|---
25 | 25
125 | 250
50 | 150
150 | 150
Rule #3  Two phase locking (2PL)  for transactions

\[ T_i = \ldots \ \text{l}_i(A) \ldots \ldots \ldots \ \text{u}_i(A) \ldots \ldots \]

- no unlocks
- no locks
# locks held by Ti

Time

Growing Phase  Shrinking Phase

Notes 09
Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td>l₂(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>A ← A * 2; Write(A); l₂(B)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td></td>
</tr>
</tbody>
</table>
# Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1(A); \text{Read}(A)$</td>
<td>$l_2(A); \text{Read}(A)$</td>
</tr>
<tr>
<td>$A \leftarrow A + 100; \text{Write}(A)$</td>
<td>$A \leftarrow A x 2; \text{Write}(A); l_2(B)$</td>
</tr>
<tr>
<td>$l_1(B); u_1(A)$</td>
<td>delayed</td>
</tr>
</tbody>
</table>

Read(B); $B \leftarrow B + 100$

Write(B); $u_1(B)$
## Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1(A);\text{Read}(A)$</td>
<td>$l_2(A);\text{Read}(A)$</td>
</tr>
<tr>
<td>$A \leftarrow A + 100;\text{Write}(A)$</td>
<td>$A \leftarrow Ax2;\text{Write}(A);l_2(B)$</td>
</tr>
<tr>
<td>$l_1(B); u_1(A)$</td>
<td>$l_2(B); u_2(A);\text{Read}(B)$</td>
</tr>
<tr>
<td>Read($B$);$B \leftarrow B + 100$</td>
<td>$B \leftarrow Bx2;\text{Write}(B);u_2(B)$</td>
</tr>
<tr>
<td>Write($B$); $u_1(B)$</td>
<td></td>
</tr>
</tbody>
</table>
### Schedule H  \ ((T_2 \text{ reversed})\)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(l_1(A)); Read(A)</td>
<td>(l_2(B)); Read(B)</td>
</tr>
<tr>
<td>A (\leftarrow) A+100; Write(A)</td>
<td>B (\leftarrow) Bx2; Write(B)</td>
</tr>
<tr>
<td>delayed</td>
<td>(l_2(A)) delayed</td>
</tr>
</tbody>
</table>
• Assume deadlocked transactions are rolled back
  – They have no effect
  – They do not appear in schedule

E.g., Schedule H =

This space intentionally left blank!
Next step:

Show that rules #1,2,3 $\Rightarrow$ conflict-
serializable
schedules
Conflict rules for $l_i(A), u_i(A)$:

- $l_i(A), l_j(A)$ conflict
- $l_i(A), u_j(A)$ conflict

Note: no conflict $< u_i(A), u_j(A)>, < l_i(A), r_j(A)>, ...$
Theorem  Rules #1,2,3 ⇒ conflict
(2PL) serializable
schedule

To help in proof:
Definition  Shrink(Ti) = SH(Ti) =
first unlock action of Ti
**Lemma**

$T_i \rightarrow T_j$ in $S \Rightarrow SH(T_i) <_S SH(T_j)$

**Proof of lemma:**

$T_i \rightarrow T_j$ means that

$$S = \ldots \ p_i(A) \ldots \ q_j(A) \ldots; \quad p,q \text{ conflict}$$

By rules 1,2:

$$S = \ldots \ p_i(A) \ldots \ u_i(A) \ldots \ l_j(A) \ldots \ q_j(A) \ldots$$

By rule 3:

$$SH(T_i) \quad SH(T_j)$$

So, $SH(T_i) <_S SH(T_j)$
Theorem  Rules #1,2,3 \Rightarrow \text{conflict (2PL) serializable schedule}

Proof:
(1) Assume P(S) has cycle
\[ T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1 \]
(2) By lemma: \( SH(T_1) < SH(T_2) < ... < SH(T_1) \)
(3) Impossible, so P(S) acyclic
(4) \Rightarrow S \text{ is conflict serializable}
• Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency....
  – Shared locks
  – Multiple granularity
  – Inserts, deletes and phantoms
  – Other types of C.C. mechanisms
Shared locks

So far:
\[ S = \ldots l_1(A) \; r_1(A) \; u_1(A) \; \ldots \; l_2(A) \; r_2(A) \; u_2(A) \; \ldots \]

Do not conflict

Instead:
\[ S = \ldots \; ls_1(A) \; r_1(A) \; ls_2(A) \; r_2(A) \; \ldots \; us_1(A) \; us_2(A) \]
Lock actions
l-ti(A): lock A in t mode (t is S or X)
u-ti(A): unlock t mode (t is S or X)

Shorthand:
u_i(A): unlock whatever modes
T_i has locked A
Rule #1   Well formed transactions

\[ T_i = \ldots \text{I-S}_1(A) \ldots \text{r}_1(A) \ldots \text{u}_1(A) \ldots \]
\[ T_i = \ldots \text{I-X}_1(A) \ldots \text{w}_1(A) \ldots \text{u}_1(A) \ldots \]
• What about transactions that read and write same object?

Option 1: Request exclusive lock

\[ T_i = \ldots l-X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots \]
• What about transactions that read and write same object?

Option 2: Upgrade

(E.g., need to read, but don’t know if will write...)

$$T_i=\ldots \ l-S_1(A) \ \ldots \ r_1(A) \ \ldots \ l-X_1(A) \ \ldots w_1(A) \ \ldots u(A)\ldots$$

Think of
- Get 2nd lock on A, or
- Drop S, get X lock
Rule #2  Legal scheduler

S = ....l-S_i(A) ... ... u_i(A) ...

no l-X_j(A)

\[ \text{S = } \ldots \text{l-X}_i(A) \ldots \ldots u_i(A) \ldots \]

\[ \text{no l-X}_j(A) \]

no l-S_j(A)
A way to summarize Rule #2

Compatibility matrix

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Rule # 3  2PL transactions

No change except for upgrades:
(I) If upgrade gets more locks
    (e.g., S → {S, X}) then no change!
(II) If upgrade releases read (shared) lock (e.g., S → X)
    - can be allowed in growing phase
Theorem  
Rules 1,2,3 $\Rightarrow$ Conf. serializable
for S/X locks schedules

Proof:  similar to X locks case

Detail:
l-t_i(A), l-r_j(A) do not conflict if comp(t,r)
l-t_i(A), u-r_j(A) do not conflict if comp(t,r)
Lock types beyond S/X

Examples:

(1) increment lock
(2) update lock
Example (1): increment lock

- Atomic increment action: $\text{IN}_i(A)$
  
  $$\{\text{Read}(A); \ A \leftarrow A+k; \ \text{Write}(A)\}$$

- $\text{IN}_i(A)$, $\text{IN}_j(A)$ do not conflict!

```
A=5  \text{IN}_i(A) \quad A=15  \quad A=7  \quad \text{IN}_j(A)
+2 \quad +10 \quad +10 \quad \text{IN}_j(A)
+10 \quad \quad \quad +2 \quad \text{IN}_i(A)
```

$A=7 \quad \text{IN}_i(A) \quad \quad \quad \quad \quad A=17 \quad \text{IN}_j(A)$
<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comp

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>I</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
# Update locks

A common deadlock problem with upgrades:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l-S₁(A)</td>
<td>l-S₂(A)</td>
</tr>
<tr>
<td>l-X₁(A)</td>
<td>l-X₂(A)</td>
</tr>
</tbody>
</table>

--- Deadlock ---
Solution

If $T_i$ wants to read A and knows it may later want to write A, it requests update lock (not shared)
New request

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comp

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>TorF</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

New request

-> symmetric table?
Note: object A may be locked in different modes at the same time...

\[ S_1 = \ldots I-S_1(A)\ldots I-S_2(A)\ldots I-U_3(A)\ldots \begin{cases} I-S_4(A)\ldots ? \\ I-U_4(A)\ldots ? \end{cases} \]

- To grant a lock in mode t, mode t must be compatible with all currently held locks on object
How does locking work in practice?

• Every system is different
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)

• But here is one (simplified) way ...
Sample Locking System:

(1) Don’t trust transactions to request/release locks
(2) Hold all locks until transaction commits

![Graph showing the number of locks over time.]
Scheduler, part I

Scheduler, part II

DB

lock table

Ti

Read(A), Write(B)

l(A), Read(A), l(B), Write(B)...

Read(A), Write(B)
Lock table

Conceptually

If null, object is unlocked

Lock info for B

Lock info for C

Every possible object
But use hash table:

If object not found in hash table, it is unlocked
Lock info for A - example

Object: A
Group mode: U
Waiting: yes

List:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>S</td>
<td>no</td>
</tr>
<tr>
<td>T2</td>
<td>U</td>
<td>no</td>
</tr>
<tr>
<td>T3</td>
<td>X</td>
<td>yes</td>
</tr>
</tbody>
</table>

tran mode wait? Nxt T_link

To other T3 records
What are the objects we lock?

<table>
<thead>
<tr>
<th>Relation A</th>
<th>Tuple A</th>
<th>Disk block A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tuple B</td>
<td>Disk block A</td>
</tr>
<tr>
<td></td>
<td>Tuple C</td>
<td>Disk block B</td>
</tr>
</tbody>
</table>

?
• Locking works in any case, but should we choose small or large objects?

• If we lock large objects (e.g., Relations)
  – Need few locks
  – Low concurrency

• If we lock small objects (e.g., tuples, fields)
  – Need more locks
  – More concurrency
We **can** have it both ways!!

Ask any janitor to give you the solution...

```
<table>
<thead>
<tr>
<th>Stall 1</th>
<th>Stall 2</th>
<th>Stall 3</th>
<th>Stall 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

restroom

```
hall
```
Example

\[ R1 \]

\[ t_1 \]

\[ t_2 \]

\[ t_4 \]

\[ T_1(IS), T_2(S) \]

\[ T_1(S) \]
Example

\[
\begin{align*}
R1 & \\
\text{t}_1 & \quad \text{T}_1(\text{S}) \\
\text{t}_2 & \quad \text{t}_3 \\
\text{t}_4 & \quad \text{T}_2(\text{IX})
\end{align*}
\]
Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IX</td>
</tr>
<tr>
<td>IX</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>SIX</td>
</tr>
<tr>
<td>SIX</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
## Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IS</td>
</tr>
<tr>
<td>IS</td>
<td>T</td>
</tr>
<tr>
<td>Holder</td>
<td>IX</td>
</tr>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>SIX</td>
<td>T</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
</tr>
<tr>
<td>Parent locked in</td>
<td>Child can be locked in</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>IS</td>
<td></td>
</tr>
<tr>
<td>IX</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>SIX</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- **P**
- **C**
<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IS, S</td>
</tr>
<tr>
<td>IX</td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td>S</td>
<td>[S, IS] not necessary</td>
</tr>
<tr>
<td>SIX</td>
<td>X, IX, [SIX]</td>
</tr>
<tr>
<td>X</td>
<td>none</td>
</tr>
</tbody>
</table>
Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X, SIX, IX only if parent(Q) locked by Ti in IX, SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q’s children are locked by Ti
Exercise:

• Can T2 access object f2.2 in X mode? What locks will T2 get?
Exercise:

- Can T2 access object f2.2 in X mode? What locks will T2 get?
Exercise:

• Can T₂ access object f₃.₁ in X mode? What locks will T₂ get?
Exercise:

• Can $T_2$ access object $f_{2.2}$ in $S$ mode? What locks will $T_2$ get?
Exercise:

• Can $T_2$ access object $f_{2.2}$ in $X$ mode? What locks will $T_2$ get?
Insert + delete operations

\[
\begin{array}{|c|}
\hline
A \\
\vdots \\
Z \\
\alpha \\
\hline
\end{array}
\]

Insert
Modifications to locking rules:

(1) Get exclusive lock on A before deleting A
(2) At insert A operation by Ti, Ti is given exclusive lock on A
Still have a problem: **Phantoms**

Example: relation R (E#, name, ...)

constraint: E# is key

use tuple locking

| R   | E# | Name | ...
|-----|----|------|---
| o1  | 55 | Smith|   |
| o2  | 75 | Jones|   |
$T_1$: Insert $<99,\text{Gore},...>$ into $R$

$T_2$: Insert $<99,\text{Bush},...>$ into $R$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1(o_1)$</td>
<td>$S_2(o_1)$</td>
</tr>
<tr>
<td>$S_1(o_2)$</td>
<td>$S_2(o_2)$</td>
</tr>
<tr>
<td>Check Constraint</td>
<td>Check Constraint</td>
</tr>
</tbody>
</table>

$\vdots$

Insert $o_3[99,\text{Gore},...]$

$\vdots$

Insert $o_4[99,\text{Bush},...]$
Solution

- Use multiple granularity tree
- Before insert of node Q,
  lock parent(Q) in X mode
Back to example

$T_1$: Insert<99,Gore>

$T_2$: Insert<99,Bush>

$X_1(R)$

Check constraint

Insert<99,Gore>

U(R)

$X_2(R)$

Check constraint

Oops! e# = 99 already in R!
Instead of using R, can use index on R:

Example:

```
Index 0<E#<100
   E#=2  E#=5
```

```
Index 100<E#<200
   E#=107  E#=109
```

...
• This approach can be generalized to multiple indexes...
Next:

- Tree-based concurrency control
- Validation concurrency control
Example

• all objects accessed through root, following pointers

E can we release A lock if we no longer need A??
Idea: traverse like “Monkey Bars”
Why does this work?

- Assume all $T_i$ start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$ locks root before $T_j$

- Actually works if we don’t always start at root
Rules: tree protocol (exclusive locks)

(1) First lock by $T_i$ may be on any item
(2) After that, item $Q$ can be locked by $T_i$ only if parent($Q$) locked by $T_i$
(3) Items may be unlocked at any time
(4) After $T_i$ unlocks $Q$, it cannot relock $Q$
• Tree-like protocols are used typically for B-tree concurrency control

E.g., during insert, do not release parent lock, until you are certain child does not have to split
Validation

Transactions have 3 phases:

(1) **Read**
   - all DB values read
   - writes to temporary storage
   - no locking

(2) **Validate**
   - check if schedule so far is serializable

(3) **Write**
   - if validate ok, write to DB
Key idea

- Make validation atomic
- If $T_1, T_2, T_3, \ldots$ is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1 \ T_2 \ T_3 \ldots$
To implement validation, system keeps two sets:

- **FIN** = transactions that have finished phase 3 (and are all done)
- **VAL** = transactions that have successfully finished phase 2 (validation)
Example of what validation must prevent:

\[ RS(T_2) = \{B\} \cap RS(T_3) = \{A,B\} \neq \phi \]
\[ WS(T_2) = \{B,D\} \]
\[ WS(T_3) = \{C\} \]
Example of what validation must prevent:

\[ RS(T_2) = \{B\} \cap RS(T_3) = \{A, B\} \neq \emptyset \]
\[ WS(T_2) = \{B, D\} \]
\[ WS(T_3) = \{C\} \]
Another thing validation must prevent:

\[
\begin{align*}
RS(T_2) &= \{A\} & RS(T_3) &= \{A,B\} \\
WS(T_2) &= \{D,E\} & WS(T_3) &= \{C,D\}
\end{align*}
\]

BAD: \( w_3(D) \) \( w_2(D) \)
Another thing validation must prevent:

\[ RS(T_2) = \{A\} \quad RS(T_3) = \{A, B\} \]
\[ WS(T_2) = \{D, E\} \quad WS(T_3) = \{C, D\} \]
Validation rules for \( T_j \):

1. When \( T_j \) starts phase 1:
   
   \[
   \text{ignore}(T_j) \leftarrow \text{FIN}
   \]

2. At \( T_j \) Validation:
   
   if check \((T_j)\) then
   
   \[
   \begin{array}{l}
   \text{[ VAL } \leftarrow \text{VAL U \{T}_j\}; \\
   \text{do write phase;} \\
   \text{FIN } \leftarrow \text{FIN U \{T}_j\} \ 
   \end{array}
   \]

Check ($T_j$):

For $Ti \in \text{VAL - IGNORE (} T_j \text{)}$ DO

\[
\text{IF } [ \text{ WS}(Ti) \cap \text{ RS}(T_j) \neq \emptyset \text{ OR } Ti \notin \text{ FIN } ] \text{ THEN RETURN false;}
\]

RETURN true;

Is this check too restrictive?
Improving Check($T_j$)

For $T_i \in \text{VAL - IGNORE (} T_j \text{)}$ DO

IF [ $\text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset$ OR

$(T_i \notin \text{FIN AND WS}(T_i) \cap \text{WS}(T_j) \neq \emptyset)$]

THEN RETURN false;

RETURN true;
Exercise:

U: RS(U)={B}  WS(U)={D}
W: RS(W)={A,D}  WS(W)={A,C}
T: RS(T)={A,B}  WS(T)={A,C}
V: RS(V)={B}  WS(V)={D,E}

△ start  ⊕ validate  ★ finish
Validation (also called optimistic concurrency control) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints
Summary

Have studied C.C. mechanisms used in practice
- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation