Example:

T1: Read(A) T2: Read(A)
   A ← A + 100    A ← A × 2
   Write(A)       Write(A)
   Read(B)        Read(B)
   B ← B + 100    B ← B × 2
   Write(B)       Write(B)

Constraint: A = B

Schedule A

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

Schedule B

<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></td>
</tr>
<tr>
<td>Read(B); B ← B + 100;</td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
</tr>
</tbody>
</table>

Read(A); A ← A × 2;
Write(A);       Read(B); B ← B × 2;
Write(B);
### 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>Read(A); A ← A×2; Write(A);</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Read(B); B ← B×2; Write(B);</td>
<td></td>
<td></td>
<td>50</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; Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>150</td>
<td>150</td>
</tr>
</tbody>
</table>

### Schedule C

<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>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100; Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>125</td>
<td>125</td>
</tr>
</tbody>
</table>

### Schedule D

<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>Read(A); A ← A+100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100; Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>125</td>
<td>125</td>
</tr>
</tbody>
</table>

### Schedule E

Same as Schedule D but with new T2'
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></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A×1;</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
</tr>
</tbody>
</table>

A | B
25 | 25

- Want schedules that are “good”, regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example:

Sc=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

The Transaction Game

<table>
<thead>
<tr>
<th>A</th>
<th>r</th>
<th>w</th>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>T1</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>T2</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
</tbody>
</table>

The Transaction Game

<table>
<thead>
<tr>
<th>A</th>
<th>r</th>
<th>w</th>
<th>r</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>T1</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>T2</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
</tr>
</tbody>
</table>

- can move column
- hits something
However, for Sd:
\[ S_d = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B) \]

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

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

\[ T_1 \rightarrow T_2 \]  \[ T_1 \rightarrow T_2 \]

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

*Transaction*: sequence of \( r(x) \), \( w(x) \) actions

*Conflicting actions*: \( r_1(A) \) \( w_1(A) \) \( w_2(A) \) \( r_1(A) \) \( w_2(A) \)

*Schedule*: represents chronological order in which actions are executed

*Serial schedule*: no interleaving of actions or transactions

---

Is it OK to model reads & writes as occurring at a single point in time in a schedule?

- \( S = \ldots r(x) \ldots w_2(b) \ldots \)

---

What about conflicting, concurrent actions on same object?

\[\begin{array}{c}
\text{start } r_1(A) \\
\text{start } w_2(A)
\end{array} \quad \text{time} \quad \begin{array}{c}
\text{end } r_1(A) \\
\text{end } w_2(A)
\end{array}\]

- 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: \( T_i \rightarrow T_j \) whenever
- \( p(A), q(A) \) are actions in S
- \( p(A) <_S q(A) \)
- at least one of \( p_i, q_j \) is a write

Exercise:

- What is \( P(S) \) for 
  \( S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D) \)

- Is \( S \) serializable?

Another Exercise:

- What is \( P(S) \) for 
  \( S = w_1(A) \ r_2(A) \ r_3(A) \ w_4(A) ? \)

Lemma

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

Proof:
Assume \( P(S_1) \neq P(S_2) \)
\( \exists T_i: T_i \rightarrow T_j \) in \( S_1 \) and not in \( S_2 \)
\( S_1 = \ldots p(A) \ldots q(A) \ldots \)
\( S_2 = \ldots q(A) \ldots p(A) \ldots \) conflict
\( \Rightarrow S_1, S_2 \) not conflict equivalent

Note: \( P(S_1)=P(S_2) \neq S_1, S_2 \) conflict equivalent
Note: $P(S_1) = P(S_2) \neq S_1, S_2$ conflict equivalent

Counter example:

$S_1 = w_1(A) r_2(A) w_2(B) r_1(B)$

$S_2 = r_2(A) w_1(A) r_1(B) w_2(B)$

Theorem

$P(S_1)$ acyclic $\iff S_1$ conflict serializable

($\Rightarrow$) 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 $P(S_i)$ is acyclic

Theorem

$P(S_1)$ acyclic $\iff S_1$ conflict serializable

($\Leftarrow$) Assume $S_1$ is conflict serializable

$\Rightarrow \exists S_i: S_i, S_1$ conflict equivalent

$\Rightarrow P(S_i) = P(S_1)$

$\Rightarrow P(S_1)$ acyclic since $P(S_i)$ is acyclic

Theorem

$P(S_1)$ acyclic $\iff S_1$ 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

$S_1 = \ldots q(A) \ldots p(A) \ldots$

(3) we now have $S_1 = < T_1 \text{ actions } > < \ldots \text{ rest } >$

(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

T₁ T₂ .......... Tₙ

Scheduler

DB

A locking protocol

Two new actions:
lock (exclusive): lᵢ(A)
unlock: uᵢ(A)

Rule #1: Well-formed transactions

Ti: ... lᵢ(A) ... pᵢ(A) ... uᵢ(A) ...

Rule #2 Legal scheduler

S = ........ lᵢ(A) .......... uᵢ(A) .........

no lᵢ(A)

Exercise:

• What schedules are legal?
What transactions are well-formed?
S₁ = l₁(A)l₁(B)r₁(A)w₁(B)l₁(A)u₁(B)
r₂(B)w₂(B)u₂(B)l₂(B)r₂(B)u₂(B)
S₂ = l₁(A)r₁(A)w₁(B)u₁(A)u₁(B)
l₂(B)r₂(B)w₂(B)l₂(B)r₂(B)u₂(B)
S₃ = l₁(A)r₁(A)u₁(A)l₁(B)w₁(B)u₁(B)
l₂(B)r₂(B)w₂(B)l₂(B)r₂(B)u₂(B)

Exercise:

• What schedules are legal?
What transactions are well-formed?
S₁ = l₁(A)l₁(B)r₁(A)w₁(B)l₂(B)u₁(A)u₁(B)
r₂(B)w₂(B)u₂(B)l₂(B)r₂(B)u₂(B)
S₂ = l₁(A)r₁(A)w₁(B)u₁(A)u₁(B)
l₂(B)w₂(B)l₂(B)r₂(B)u₂(B)
S₃ = l₁(A)r₁(A)u₁(A)l₁(B)w₁(B)u₁(B)
l₂(B)r₂(B)w₂(B)l₂(B)r₂(B)u₂(B)
Schedule F

T1 | T2
---|---
\(l_1(A); \text{Read}(A)\) | \(A \leftarrow A + 100; \text{Write}(A); u_1(A)\)
\(l_2(A); \text{Read}(A)\) | \(A \leftarrow Ax2; \text{Write}(A); u_2(A)\)
\(l_1(B); \text{Read}(B)\) | \(B \leftarrow Bx2; \text{Write}(B); u_2(B)\)
\(l_2(B); \text{Read}(B)\) | \(B \leftarrow B + 100; \text{Write}(B); u_1(B)\)

Schedule G

T1 | T2
---|---
\(l_1(A); \text{Read}(A)\) | \(A \leftarrow A + 100; \text{Write}(A); u_1(A)\)
\(l_1(B); \text{Read}(A)\) | \(A \leftarrow Ax2; \text{Write}(A); u_1(A)\)
\(l_1(B); \text{Read}(B)\) | \(B \leftarrow Bx2; \text{Write}(B); u_1(B)\)
\(l_2(B); \text{Read}(B)\) | \(B \leftarrow B + 100; \text{Write}(B); u_1(B)\)

Rule #3 Two phase locking (2PL) for transactions

\(T_i = \ldots l_i(A) \ldots u_i(A) \ldots\)

\[\text{no unlocks} \quad \text{no locks}\]

Schedule F

T1 | T2 | A | B
---|---|---|---
\(l_1(A); \text{Read}(A)\) | \(A \leftarrow A + 100; \text{Write}(A); u_1(A)\) | \(25\) | \(25\)
\(l_2(A); \text{Read}(A)\) | \(A \leftarrow Ax2; \text{Write}(A); u_2(A)\) | \(125\) |
\(l_1(B); \text{Read}(B)\) | \(B \leftarrow Bx2; \text{Write}(B); u_2(B)\) | \(250\) | \(50\)
\(l_2(B); \text{Read}(B)\) | \(B \leftarrow B + 100; \text{Write}(B); u_1(B)\) | \(150\) | \(150\)

\(250\) | \(150\)

Schedule G

T1 | T2 | A | B
---|---|---|---
\(l_1(A); \text{Read}(A)\) | \(A \leftarrow A + 100; \text{Write}(A); u_1(A)\) | \(25\) | \(25\)
\(l_1(B); \text{Read}(A)\) | \(A \leftarrow Ax2; \text{Write}(A); u_1(A)\) | \(125\) |
\(l_1(B); \text{Read}(B)\) | \(B \leftarrow Bx2; \text{Write}(B); u_1(B)\) | \(250\) | \(50\)
\(l_2(B); \text{Read}(B)\) | \(B \leftarrow B + 100; \text{Write}(B); u_1(B)\) | \(150\) | \(150\)

\(250\) | \(150\)

# locks held by \(T_i\)

\[\text{Growing Phase} \quad \text{Shrinking Phase}\]

Time
**Schedule G**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1(A); Read(A)</td>
<td></td>
</tr>
<tr>
<td>A ← A+100; Write(A); u1(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100</td>
<td>delayed</td>
</tr>
<tr>
<td>Write(B); u1(B)</td>
<td></td>
</tr>
</tbody>
</table>

**Schedule H** (T2 reversed)

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

- Assume deadlocked transactions are rolled back
  - They have no effect
  - They do not appear in schedule

  E.g., Schedule H =

Next step:

Show that rules #1,2,3 ⇒ conflict-serializable schedules

**Conflict rules for l(A), u(A):**

- l(A), l(A) conflict
- l(A), u(A) conflict

Note: no conflict < u(A), u(A)>, < l(A), r(A)>, ...
Theorem: Rules #1,2,3 $\Rightarrow$ conflict serializable schedule

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

Lemma
Ti $\rightarrow$ Tj in S $\Rightarrow$ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:
Ti $\rightarrow$ Tj means that
S = ... p(A) ... q(A) ...; p,q conflict
By rules 1,2:
S = ... p(A) ... u_i(A) ... l_j(A) ... qj(A) ...

Lemma
Ti $\rightarrow$ Tj in S $\Rightarrow$ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:
Ti $\rightarrow$ Tj means that
S = ... p(A) ... q_i(A) ...; p,q conflict
By rules 1,2:
S = ... p(A) ... u_i(A) ... l_j(A) ... qj(A) ...
By rule 3: SH(Ti) $<_S$ SH(Tj)
So, SH(Ti) $<_S$ SH(Tj)

Theorem: Rules #1,2,3 $\Rightarrow$ conflict 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) $<_S$ SH(T_2) $<_S$ ... $<_S$ SH(T_n)
(3) Impossible, so P(S) acyclic
(4) $\Rightarrow$ S is conflict serializable

2PL subset of Serializable
S1: w1(x) w3(x) w2(y) w1(y)

- S1 cannot be achieved via 2PL: The lock by T1 for y must occur after w2(y), so the unlock by T1 for x must occur after this point (and before w1(x)). Thus, w3(x) cannot occur under 2PL where shown in S1 because T1 holds the x lock at that point.
- However, S1 is serializable (equivalent to T2, T1, T3).

If you need a bit more practice:
Are our schedules S_C and S_D 2PL schedules?

S_C: w1(A) w2(A) w1(B) w2(B)

S_D: w1(A) w2(A) w2(B) w1(B)

- 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 = ... l(A) r(A) u1(A) ... l(A) r2(A) u2(A) ...

Do not conflict

Instead:
S = ... ls1(A) r1(A) ls2(A) r2(A) u2(A) ...

Do not conflict
Lock actions
l-t(A): lock A in t mode (t is S or X)
u-t(A): unlock t mode (t is S or X)

Shorthand:
u(A): unlock whatever modes
Ti has locked A

Rule #1  Well formed transactions
Ti =... l-Si(A) ... r1(A) ... u1(A) ...
Ti =... l-Xi(A) ... w1(A) ... u1(A) ...

• What about transactions that read and write same object?

Option 1: Request exclusive lock
Ti = ... l-Xi(A) ... r1(A) ... w1(A) ... u(A) ...

Option 2: Upgrade
(E.g., need to read, but don’t know if will write...)
Ti =... l-Si(A) ... r1(A) ... l-X1(A) ... w1(A) ... u(A)...

 Think of
• Get 2nd lock on A, or
• Drop S, get X lock

• What about transactions that read and write same object?

Rule #2  Legal scheduler
S = ... l-Si(A) ... u1(A) ...
no l-Xj(A)
S = ... l-Xi(A) ... u1(A) ...
no l-Xj(A)
no l-Sj(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., $S \rightarrow \{S, X\}$) then no change!
(II) If upgrade releases read (shared)
   lock (e.g., $S \rightarrow X$)
   - 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:
$l-b(A), l-r_j(A)$ do not conflict if comp(t,r)
l-t(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: $IN_i(A)
  \{Read(A); A \leftarrow A+k; Write(A)\}$
- $IN_i(A), IN_j(A)$ do not conflict!

Comp S X I

Comp

Example (2): update lock

$Update(A, k)$: $\{Read(A); A \leftarrow A+k; Write(A)\}$

$Update(A, k)$: $\{Read(A); A \leftarrow A\times k; Write(A)\}$

Proof: similar to X locks case

Detail:
$l-b(A), l-r_j(A)$ do not conflict if comp(t,r)
l-t(A), $u-r_j(A)$ do not conflict if comp(t,r)
**Update locks**

A common deadlock problem with upgrades:

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

--- Deadlock ---

**Solution**

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

Lock already held in

- Symmetric table?

**Note:** object A may be locked in different modes at the same time...

S1=...l-S1(A)...l-S2(A)...l-U3(A)... l-S4(A)...? l-U4(A)...?
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

Lock table Conceptually

Every possible object

- If null, object is unlocked
- Lock info for B
- Lock info for C

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: 

<table>
<thead>
<tr>
<th>Tran mode wait?</th>
<th>Next T_link</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T1</td>
</tr>
<tr>
<td>U</td>
<td>T2</td>
</tr>
<tr>
<td>X</td>
<td>T3</td>
</tr>
</tbody>
</table>

To other T3 records

What are the objects we lock?

- Relation A
- Relation B
- Tuple A
- Tuple B
- Tuple C
- Disk block A
- Disk block B

- 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...

Example

- R1
- t1
- t2
- t3
- t4
Example

```
R1
  /\       /\   \
 t1 t2 t3 t4
```

Example

```
R1
  /\       /\   \
 t1 t2 t3 t4
```

Example (b)

```
R1
  /\       /\   \
 t1 t2 t3 t4
```

Example

```
R1
  /\       /\   \
 t1 t2 t3 t4
```

Multiple granularity

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

Multiple granularity

<table>
<thead>
<tr>
<th>Comp</th>
<th>Requestor</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IX</td>
</tr>
<tr>
<td>S</td>
<td>SIX</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IS</td>
<td></td>
</tr>
<tr>
<td>Holder</td>
<td>IX</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>
Parent
locked in | Child can be
locked in
---|---
IS | IS
IX | IS, S
S | IS, S, IX, X, SIX
SIX | none
X | X, IX, [SIX]

Parent
locked in | Child can be locked
by same transaction in
---|---
IS | IS, S
IX | IS, S, IX, X, SIX
S | none
SIX | none
X | not necessary

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 f3.1 in X mode? What locks will T2 get?
Exercise:
- Can T2 access object f2.2 in S mode? What locks will T2 get?

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

Insert + delete operations

<table>
<thead>
<tr>
<th>A</th>
<th>Z</th>
<th>α</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>α</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>R</th>
<th>E#</th>
<th>Name</th>
<th>....</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>55</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>o2</td>
<td>75</td>
<td>Jones</td>
<td></td>
</tr>
</tbody>
</table>
Solution
• Use multiple granularity tree
• Before insert of node Q, lock parent(Q) in X mode

Back to example
T1: Insert<12,Obama>
T2: Insert<12,Romney>

T1
R1
\( t_1 \)
\( t_2 \)
\( t_3 \)

X:(R)
Check constraint
Insert<12,Obama>
U1(R)

Oops! e# = 12 already in R!

Instead of using R, can use index on R:

Example:

\( R \)

Index
0<E#<100

Index
100<E#<200

E#=2
E#=5
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
Example

- all objects accessed through root, following pointers

• all objects accessed through root, following pointers

- can we release A lock if we no longer need A??

Idea: traverse like “Monkey Bars”

Why does this work?

- Assume all Ti start at root; exclusive lock
- Ti \rightarrow Tj \Rightarrow Ti locks root before Tj

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

1. First lock by Ti may be on any item
2. After that, item Q can be locked by Ti only if parent(Q) locked by Ti
3. Items may be unlocked at any time
4. After Ti 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

Tree Protocol with Shared Locks

• Rules for shared & exclusive locks?

T1 X lock (released)
T1 S lock (held)
T1 S lock (will get)

T2 reads:
• B modified by T1
• F not yet modified by T1

Tree Protocol with Shared Locks

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

Tree Protocol with Shared Locks

• Need more restrictive protocol
• Will this work??
  - Once T1 locks one object in X mode, all further locks down the tree must be in X mode
Key idea

• Make validation atomic
• If T₁, T₂, T₃,... is validation order, then resulting schedule will be conflict equivalent to Sₛ = T₁ T₂ T₃...

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(T₂)={B}  RS(T₃)={A,B} ≠ ∅
WS(T₂)={B,D}  WS(T₃)={C}  
```

Example of what validation must prevent:

```
RS(T₂)={B}  RS(T₃)={A,B} ≠ ∅
WS(T₂)={B,D}  WS(T₃)={C}  
```

Another thing validation must prevent:

```
RS(T₂)={A}  RS(T₃)={A,B}  
WS(T₂)={D,E}  WS(T₃)={C,D}  
```

Another thing validation must prevent:

```
RS(T₂)={A}  RS(T₃)={A,B}  
WS(T₂)={D,E}  WS(T₃)={C,D}  
```

BAD: w₃(D) w₂(D)
Another thing validation must prevent:

\[ \text{RS}(T_2) = \{A\} \quad \text{RS}(T_3) = \{A, B\} \]
\[ \text{WS}(T_2) = \{D, E\} \quad \text{WS}(T_3) = \{C, D\} \]

Validation rules for \( T_j \):

1. When \( T_j \) starts phase 1:
   \[ \text{ignore}(T_j) \leftarrow \text{FIN} \]
2. at \( T_j \) Validation:
   \[ \text{if check}(T_j) \text{ then} \]
   \[ \text{VAL} \leftarrow \text{VAL} \cup \{T_j\}; \]
   \[ \text{do write phase}; \]
   \[ \text{FIN} \leftarrow \text{FIN} \cup \{T_j\} \]

Check (\( T_j \)):

\[ \text{For } T_i \in \text{VAL} - \text{IGNORE}(T_j) \text{ do } \]
\[ \text{if } (\text{WS}(T_i) \cap \text{RS}(T_j) \neq \varnothing \text{ OR } T_i \notin \text{FIN}) \text{ then RETURN false; } \]
\[ \text{RETURN true; } \]

Improving Check(\( T_j \))

\[ \text{For } T_i \in \text{VAL} - \text{IGNORE}(T_j) \text{ do } \]
\[ \text{if } (\text{WS}(T_i) \cap \text{RS}(T_j) \neq \varnothing \text{ OR } \]
\[ (T_i \notin \text{FIN AND WS}(T_i) \cap \text{WS}(T_j) \neq \varnothing)) \]
\[ \text{then RETURN false; } \]
\[ \text{RETURN true; } \]

Exercise:

\[ \text{U: RS}(U) = \{B\} \quad \text{WS}(U) = \{D\} \]
\[ \text{W: RS}(W) = \{A, D\} \quad \text{WS}(W) = \{A, C\} \]
\[ \text{T: RS}(T) = \{A, B\} \quad \text{WS}(T) = \{A, C\} \]
\[ \text{V: RS}(V) = \{B\} \quad \text{WS}(V) = \{D, E\} \]
Is Validation = 2PL?

Validation subset of 2PL?

Conclusion: Validation subset 2PL
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