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

A locking protocol

• Two new actions:
  – lock (exclusive): \( l_i (A) \)
  – unlock: \( u_i (A) \)
Rules for Locking

• Rule #1: Well-formed transactions
  – Ti: … li(A) … pi(A) … ui(A) …
    (pi is a read or write)

• Rule #2: Legal scheduler
  – S = ........ li(A) ........... ui(A) ........

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)l3(B)r3(B)u3(B)
  – S2 = l1(A)r1(A)w1(B)u1(A)u1(B)l2(B)r2(B)w2(B)l3(B)r3(B)u3(B)
  – S3 = l1(A)r1(A)u1(A)l1(B)w1(B)u1(B)l2(B)r2(B)w2(B)u2(B)l3(B)r3(B)u3(B)
Schedule F

<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); u₁(A)</td>
<td>A ↦ A+x₂; Write(A); u₂(A)</td>
</tr>
<tr>
<td>l₁(B); Read(B)</td>
<td>l₂(B); Read(B)</td>
</tr>
<tr>
<td>B ↦ B+100; Write(B); u₁(B)</td>
<td>B ↦ B+x₂; Write(B); u₂(B)</td>
</tr>
</tbody>
</table>

Rule #3: Two phase locking (2PL)

Ti = ....... li(A) ............ ui(A) ...........

no unlocks                                     no locks

11/9/2021
The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).
### 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>$A \leftarrow Ax_2; \text{Write}(A)$</td>
</tr>
<tr>
<td>$l_1(B); u_1(A)$</td>
<td>Delayed</td>
</tr>
<tr>
<td>$l_2(B)$</td>
<td>$l_2(B)$</td>
</tr>
<tr>
<td>$\text{Read}(B); B \leftarrow B + 100$</td>
<td>$\text{Write}(B); u_1(B)$</td>
</tr>
</tbody>
</table>

11/9/2021
### 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 x 2; Write(A); <strong>delayed</strong></td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>l₂(B); u₂(A); Read(B)</td>
</tr>
<tr>
<td>Read(B); B ← B + 100</td>
<td>B ← B x 2; Write(B); <strong>delayed</strong></td>
</tr>
<tr>
<td>Write(B); u₁(B)</td>
<td></td>
</tr>
</tbody>
</table>

11/9/2021

### Schedule H  (T2 reversed)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td>l₂(B); Read(B)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>B ← B x 2; Write(B)</td>
</tr>
<tr>
<td>l₁(B); <strong>delayed</strong></td>
<td>l₂(A); <strong>delayed</strong></td>
</tr>
</tbody>
</table>

11/9/2021
The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.

- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary for conflict serializability in the following sense:
  - Given a transaction \( T_i \) that does not follow two-phase locking, we can find a transaction \( T_j \) that uses two-phase locking, and a schedule for \( T_i \) and \( T_j \) that is not conflict serializable.

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(( B ))</td>
<td>lock-S(( A ))</td>
</tr>
<tr>
<td>read(( B ))</td>
<td>read(( A ))</td>
</tr>
<tr>
<td>( B := B - 50 )</td>
<td>unlock(( A ))</td>
</tr>
<tr>
<td>write(( B ))</td>
<td>lock-S(( B ))</td>
</tr>
<tr>
<td>unlock(( B ))</td>
<td>read(( B ))</td>
</tr>
<tr>
<td>display(( A + B ))</td>
<td>unlock(( B ))</td>
</tr>
</tbody>
</table>

Schedule H: Deadlock

- 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!
**Deadlock: Another Example**

- Consider the partial schedule

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_3$</td>
<td>$T_4$</td>
</tr>
<tr>
<td>lock-$X(B)$</td>
<td>lock-$S(A)$</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$B := B - 50$</td>
<td>lock-$S(B)$</td>
</tr>
<tr>
<td>write($B$)</td>
<td>lock-$X(A)$</td>
</tr>
</tbody>
</table>

- Neither $T_3$ nor $T_4$ can make progress — executing lock-$S(B)$ causes $T_4$ to wait for $T_3$ to release its lock on $B$, while executing lock-$X(A)$ causes $T_3$ to wait for $T_4$ to release its lock on $A$.

- Such a situation is called a **deadlock**.
  - To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released.

---

**Deadlock (Cont.)**

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.

- Concurrency control manager can be designed to prevent starvation.
The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking**: a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

Next step:

- Show that rules #1,2,3 ⇒ 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
Ti → Tj in S ⇒ SH(Ti) <s SH(Tj)

Proof of lemma:
Ti → Tj means that
S = ... p_i(A) ... q_j(A) ...; p,q conflict
By rules 1,2:
S = ... p_i(A) ... u_i(A) ... l_i(A) ... q_j(A) ...

By rule 3: SH(Ti) SH(Tj)
So, SH(Ti) <s SH(Tj)
Theorem  Rules #1,2,3 $\Rightarrow$ conflict

(2PL) serializable schedule

Proof:
(1) Assume P(S) has cycle
   \[ T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \]
(2) By lemma: \( SH(T_1) < SH(T_2) < \ldots < SH(T_1) \)
(3) Impossible, so P(S) acyclic
(4) $\Rightarrow$ S 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