Schedules

• A Schedule (Χρονοδιάγραμμα ή Χρονοπρόγραμμα) is a sequence of the actions taken by one or more transactions.

• Important actions: read and write (they take place in the main-memory buffers, not the disk)

• When a DB element A that is brought to a buffer by some transaction T, it may be read or written in that buffer not only by T but by other transactions as well.
Serial schedules

- A schedule is **serial** if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on.
- No mixing of the actions is allowed.
- A serial schedule **preserves the DB consistency**.
  Notation: \((T_1, T_2)\) or \((T_2, T_1)\)

Example: **Constraint A=B (consistency constraint)**

\[
\begin{align*}
T1: & \quad \text{Read}(A) \quad \text{Read}(B) \\
& \quad A \leftarrow A+100 \quad B \leftarrow B+100 \\
& \quad \text{Write}(A) \quad \text{Write}(B) \\
T2: & \quad \text{Read}(A) \quad \text{Read}(B) \\
& \quad A \leftarrow A\times2 \quad B \leftarrow B\times2 \\
& \quad \text{Write}(A) \quad \text{Write}(B)
\end{align*}
\]
## Serial schedule

<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>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A*2;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</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>
<tr>
<td>Schedule A</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>
## Serial schedule

<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>Read(A); A ← A*2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B*2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A+100;</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schedule B
Serializable schedules

• Are there any other schedules that also are guaranteed to preserve consistency?
• A schedule $S$ is **serializable** ($σειριόμορφο$) if there is a serial schedule $S'$ such that for every initial DB state, the effects of $S$ and $S'$ are the same.
Serializable schedule

<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></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schedule C</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

### Schedule C

- **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);
### Non-Serializable schedule

<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>Read(A); A ← A*2;</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B*2;</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td>250</td>
</tr>
<tr>
<td>Write(B);</td>
<td>Schedule D</td>
<td></td>
<td>150</td>
</tr>
</tbody>
</table>

Schedule D
## Serializable schedule

<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>125</td>
<td></td>
</tr>
<tr>
<td>Read(A); A ← A+200;</td>
<td>Write(A);</td>
<td>325</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+200;</td>
<td>Write(B);</td>
<td></td>
<td>225</td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td>325</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>325</td>
<td>325</td>
</tr>
</tbody>
</table>

**Schedule E**

The details of the transactions do matter!
But it is not realistic for the scheduler to concern itself with the such details.
Transaction Scheduling

• Only want to allow the execution of “good” schedules regardless of
  – initial states
  – transaction semantics
• Only the order of reads and writes matters
• **Example**: Given a schedule in the following form, determine whether it should be allowed to proceed without causing DB inconsistency problems.

\[ S_C = r_1(A);w_1(A);r_2(A);w_2(A);r_1(B);w_1(B);r_2(B);w_2(B) \]
Conflicts (Συγκρούσεις)

• **Conflict**: a pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change. Most pairs of actions do not conflict.

• We may not swap the order of actions when:
  – Two actions belong to the *same* transaction, because the order of actions of a single transaction are fixed and may not be reordered
  – Two *writes* of the same DB element by *different* transactions conflict
  – A *read* and a *write* of the same DB element by *different* transactions also conflict.

• Generally, any two actions of different transactions may be swapped unless:
  – they involve the same database element
  – at least one is a *write*
Conflicts (Συγκρούσεις)

• A schedule conflict-serializable (συγκρουσιακώς σειριόμορφο) if it is conflict-equivalent (συγκρουσιακώς ισοδύναμο) to a serial schedule (i.e. if it can be turned into the other by a sequence of non-conflicting swaps).

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

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

Is \( S_C \) conflict-serializable?
### The transaction “game”

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

We can move a column until it hits something.
<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td>w</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**move**

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

**move**
Basic Concepts

• *Transaction (Δοσοληψία)*: a sequence of actions of the form $ri(X)$ and $wi(X)$

• *Conflicting actions (Συγκρουόμενες ενέργειες)*:
  - $r1(A); w2(A)$, $w2(A); r1(A)$, $w1(A); w2(A)$

• *Schedule (Χρονοπρόγραμμα)*: represents the chronological order in which actions are executed

• *Serial schedule (Σειριακό χρονοπρόγραμμα)*: no interleaving (διαπλοκή) of transactions is allowed
Transaction Scheduling (Χρονοπρογραμματισμός Δοσοληψιών)

- Example: Is there an equivalent serial schedule for the following

\[ S_D = r1(A)w1(A)r2(A)w2(A) r2(B)w2(B)r1(B)w1(B) \]

- T1 must precede T2 or T2 must precede T1 in any equivalent schedule. However

\[ S_D = r1(A)w1(A)r2(A)w2(A) r2(B)w2(B)r1(B)w1(B) \]

- \( S_D \) cannot be rearranged into a serial schedule, hence \( S_D \) is not equivalent to any serial schedule, hence \( S_D \) is "bad"
Precedence graph (Γράφημα προήγησης)

• For a schedule S, its precedence graph \( P(S) \) is defined as follows
  – Nodes (Κόμβοι): transactions in S
  – Arcs (Ακμές): \( T_i \) precedes \( T_j \) (\( T_i \rightarrow T_j \)) whenever
    • \( p_i(A) \), \( q_j(A) \) are actions in S
    • \( p_i(A) <_S q_j(A) \) and involve the same element \( A \)
    • at least one of \( p_i, q_j \) is a write action

• This is a method to test for conflict-serializability
Precedence graph (Γράφημα προήγησης)

• Example 1

\[ S : r2(A); r1(B); w2(A); r3(A); w1(B); w3(A); r2(B); w2(B); \]

- For actions involving A, \( T2(A) <_S T3(A) \)

- For actions involving B, \( T1(B) <_S T2(B) \)

- Overall (and therefore \( S \) is conflict-serializable):
**Precedence graph (Γράφημα προήγησης)**

- Example 2

\[ S' : r2(A); r1(B); w2(A); r2(B); r3(A); w1(B); w3(A); w2(B); \]

  - For actions involving A, \( T2(A) <_{S'} T3(A) \)

- For actions involving B, we observe that \( T1(B) <_{S'} T2(B) \) and \( T2(B) <_{S'} T1(B) \)

- Overall:
Precedence graph (Γράφημα προήγησης)

• Exercise: What is the precedence graph P(S) for the following schedule
  \[ S = w_3(A) \ w_2(C) \ r_1(A) \ w_1(B) \ r_1(C) \ w_2(A) \ r_4(A) \ w_4(D) \]

• To determine if a schedule is conflict-serializable, we construct the precedence graph and check if there are cycles. If yes, then we can conclude that it is not conflict-serializable.
**Precedence graph (Γράφημα προήγησης)**

- **Lemma** If $S_1$, $S_2$ conflict equivalent $\Rightarrow P(S_1)=P(S_2)$
- **Proof**: Assume $P(S_1) \neq P(S_2)$
  
  Then, $\exists T_i: T_i \rightarrow T_j$ in $S_1$ and not in $S_2$
  
  $S_1 = ...pi(A)... qj(A)...$ $\left\{ \begin{array}{l} \pi, qj \\ \text{conflict} \end{array} \right.$
  
  $S_2 = ...qj(A)...pi(A)...$

  Hence, $S_1$ and $S_2$ cannot be conflict equivalent

- **Note**: $P(S_1)=P(S_2)$ does not imply $S_1$, $S_2$ conflict equivalent

- **Counterexample**:
  
  $S_1=w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)$
  
  $S_2=r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)$
Theorem: P(S1) acyclic iff S1 conflict serializable

Proof:

- (if) Assume S1 is conflict serializable
  \[ \exists Ss: Ss, S1 \text{ conflict equivalent} \]
  \[ \Rightarrow P(Ss) = P(S1) \]
  \[ \Rightarrow P(S1) \text{ acyclic since } P(Ss) \text{ is acyclic} \]

- (only if) Assume P(S1) is acyclic. Transform S1 as follows:
  1. Take T1 to be transaction with no incoming arcs
  2. Move all T1 actions to the front
  3. We now have S1 = \(< T1 \text{ actions } >\text{... rest } ...>\); remove T1 and incident arcs
  4. Repeat above steps to serialize rest!
Enforcing Serializability (Επιβολή σειριομορφίας)

• Option 1: allow any schedule; check for cycles in precedence graph (reactive)
• Option 2: prevent cycles in precedence graphs of schedules (proactive)
• Locking protocols (Πρωτόκολλα φραγμών/κλειδωμάτων) are used to implement option 2
  – New actions: lock (exclusive), unlock
  – Scheduler maintains information about locks in a lock table
• Rule 1: well-formed transactions
  – A transaction is well-formed when every operation on a DB item X (read or write) is preceded by a lock request on X and followed by an unlock request on X
• Rule 2: legal schedule
  – A schedule is legal if no lock request is granted to a transaction Tj for a DB item X when a transaction Ti has already been granted the lock to X
Simple lock protocol

- Two new actions:
  - lock (exclusive): $li(X)$
  - unlock: $ui(X)$
- We have a **single** type of lock

- **Rule #1**: Well-formed transactions
  - $Ti$: ...
    - $li(X)$ ...
    - $pi(X)$ ...
    - $ui(X)$ ...

- **Rule #2**: Legal schedule
  - $S = \ldots li(X) \ldots ui(X) \ldots$
  - **no** $lj(X)$
Simple lock protocol

- Exercise: Which of the following schedules are legal? Which transactions are well-formed?

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

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

\[ S_3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B) \]
2-Phase Lock (Διφασική φραγή)

- This method guarantees that a schedule is conflict-serializable.
- Rule #3: 2-Phase Locking (2PL) In every transaction, all lock requests precede all unlock requests.

- Idea: $T_i = \ldots l_i(A) \ldots \ldots \ldots u_i(A) \ldots \ldots$

In the previous example, does S3 comply with the 2PL protocol?
2-Phase Lock (Διφασική φραγή)

• Intuitively, each 2PL transaction may be thought to execute in its entirety at the moment it issues its first unlock request (release of the first item).

• For every schedule S of 2PL transactions, there is always at least one conflict-equivalent serial schedule (συγκρουσιακώς ισοδύναμο σειριακό χρονοπρόγραμμα): the one in which the transactions appear in the same order as their first unlocks.

• **Proof:** The conversion is best described as an induction on n, the number of transactions in legal schedule S.
  
  – **Note:** we swap the order of reads and writes and we ignore the lock and unlock actions. Once we have the read and write actions ordered serially, we can place the lock and unlock actions around them.
2-Phase Lock (Διφασική φραγή)

- Induction
  - If \( n=1 \) then \( S \) is already a serial schedule
  - Let \( S \) involve transactions \( T_1, T_2, ..., T_n \). Let \( T_i \) be the first transaction to unlock an item by op. \( U_i(X) \). Then, it is possible to move all read/write actions of \( T_i \) to the beginning of \( S \) without passing any conflicting action.

Let \( T_i \) include an action \( W_i(Y) \). If there existed an action \( W_j(Y) \) in \( S \) that precedes \( W_i(Y) \), then \( U_j(Y) \) and \( L_i(Y) \) must also appear between \( W_j(Y) \) and \( W_i(Y) \).

We assumed that \( U_i(X) \) is the first unlock, hence it precedes \( U_j(Y) \). This means that \( U_i(X) \) must also appear before \( L_i(Y) \). But then, \( T_i \) is not a 2PL transaction.
Example:

S1: w1(A); w3(A); w2(B); w1(B)

- S1 is serializable (equivalent to T2 → T1 → T3)
- S1 cannot be achieved via 2PL: The lock by T1 for B must occur after w2(B), so the unlock by T1 for A must occur after this point. Thus, the lock for w3(A) cannot occur under 2PL because T1 still holds the A lock at that point.
Exercise:

• Are the schedules $S_C$ and $S_D$ serializable and/or 2PL schedules?

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

$S_D$: w1(A) w2(A) w2(B) w1(B)
2PL and Deadlocks

• One problem that is not solved by 2PL is the potential for **deadlocks**

• Example: Consider the following schedule $S$

$$L1(A), R1(A), L2(B), R2(B), W1(A), W2(B), L1(B), L2(A)$$

Neither transaction can proceed, and they wait forever for a lock held by another transaction.
Locking with several lock modes

• Problem: With 2PL we must take a lock on a DB element $X$ even if we want to read $X$ and not write it.

• Improvements to 2PL’s performance for allowing more concurrency with two different lock types:
  – Shared lock or read lock (Κοινόχρηστος φραγμός)
  – Exclusive lock or write lock (Αποκλειστικός φραγμός)

• Shared lock
  – read operations do not conflict
  – no need to lock exclusively for read
  – Notation:
    • $\text{SLi}(X)$: “$Ti$ requests a shared lock on $X$”
    • $\text{XLi}(X)$: “$Ti$ requests an exclusive lock on $X$”
    • $\text{Ui}(X)$: “$Ti$ releases lock on $X$”
Shared lock

• Until now with 2PL:
  \[ S = \ldots l_1(A) r_1(A) u_1(A) \ldots l_2(A) r_2(A) u_2(A) \ldots \]

  Do not conflict

• Now:
  \[ S = \ldots sl_1(A) r_1(A) sl_2(A) r_2(A) \ldots u_1(A) u_2(A) \]

  \( u_i(A) \): releases any type of lock that \( T_i \) has for \( A \)
Requirements for shared and exclusive locks

- **Consistency** (Συνέπειας)
  - An action Ri(X) must be preceded by SLi(X) or X Li(X) with no intervening Ui(X)
  - An action Wi(X) must be preceded by X Li(X) with no intervening Ui(X)
  - All locks must be followed by an unlock of the element

- **2PL**
  - For any 2PL transaction Ti, no SLi(X) or X Li(X) can be preceded by an Ui(Y) action, for any Y
Requirements for shared and exclusive locks

• **Legality** (Επιτρεπτότητα)

An element may either be locked exclusively by one transaction or by several in shared mode, but not both:

– If $X_{Li}(X)$ appears in a schedule, then there cannot be a following $X_{Lj}(X)$ or $S_{Lj}(X)$ for $j \neq i$ without an intervening $U_{i}(X)$ action

– If $S_{Li}(X)$ appears in a schedule, then there cannot be a following $X_{Lj}(X)$ for $j \neq i$ without an intervening $U_{i}(X)$ action
Example

• We consider the following transactions
  
  T1: SL1(A); R1(A); XL1(B); R1(B); W1(B); U1(A); U1(B)
  
  T2: SL2(A); R2(A); SL2(B); R2(B); U2(A); U2(B)

• A legal interleaved execution of T1, T2 is as follows:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1(A); R1(A);</td>
<td>SL2(A); R2(A);</td>
</tr>
<tr>
<td>XL1(B); wait</td>
<td>SL2(B), R2(B);</td>
</tr>
<tr>
<td>XL1(B); R1(B); W1(B);</td>
<td>U2(A); U2(B);</td>
</tr>
<tr>
<td>U1(A); U1(B)</td>
<td></td>
</tr>
</tbody>
</table>

Note: This schedule is conflict-serializable because there is an equivalent serial order of execution (T2 → T1)
• If we use several lock modes, then the scheduler needs a policy about when it can grant a lock request.

• A compatibility matrix is a convenient way to describe lock-management policies (for every element).

<table>
<thead>
<tr>
<th></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>

The column for S says that we can grant a s-lock if there is no x-lock.

The column for X says that we can grant a x-lock if there are no other locks.