# Databases 2

# Lecture VII

Alessandro Artale

Faculty of Computer Science – Free University of Bolzano

Room: 221

artale@inf.unibz.it

http://www.inf.unibz.it/∼artale/

2003/2004 – First Semester

# Summary of Lecture VII

- **Concurrency Control**
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - Locking Techniques
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

# Concurrent Execution

There are good reasons for allowing *concurrency* instead of *serial* processing:

1. **Improved throughput and resource utilization.**
   (THROUGHPUT = Number of Transactions executed per unit of time.)
   The CPU and the Disk can operate in parallel. When a Transaction Read/Write the Disk another Transaction can be running in the CPU.
   The CPU and Disk **utilization** also increases.

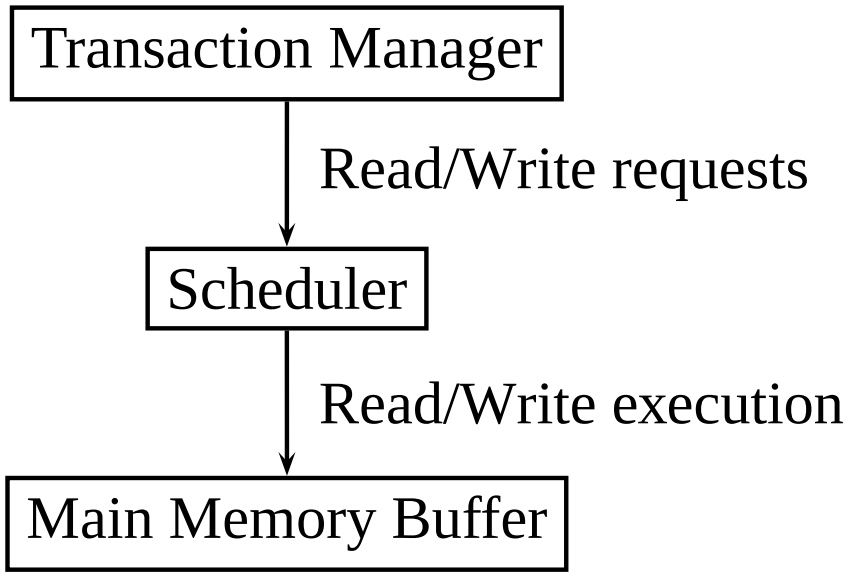
2. **Reduced waiting time.**
   In a serial processing a short Transaction may have to wait for a long Transaction to complete. Concurrent execution reduces the *average response time*: The average time for a Transaction to be completed.

## Concurrency Control

- Even when there is no “system failure” several transactions can interact to turn a consistent DB state into an **inconsistent state**.
- The *order* in which the actions of the different transactions occur has to be controlled.
- The **Scheduler** is the DBMS component for the *concurrency control*.

# Scheduler

- The **Scheduler** manages read/write requests from transactions – either executed directly or delayed – to maintain a consistent DB state.
- Read/Write requests are sent to main memory buffers in an order that is **Serializable** i.e., the execution has the same effect of a serial execution where Transactions executed one-at-a-time without interleaving.



## Serial Execution: An Example

In a banking system two Transactions $\mathtt{T}_1$ and $\mathtt{T}_2$ update two accounts – $A, B$.

- Current money in the accounts: $A = 1{,}000£$ and $B = 2{,}000£$;
- $\mathtt{T}_1$ transfers $50£$ from $A$ to $B$, while $\mathtt{T}_2$ transfers $10\%$ from $A$ to $B$;
- The DB constraint is that the sum $A + B$ must be preserved.

# Serial Execution: An Example (cont.)

The following table shows the sequential execution of $\mathtt{T}_1$ followed by $\mathtt{T}_2$ together with the main memory values for $A$ and $B$.

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 2,000 |
| t := t-50 | | | |
| WRITE(A,t) | | 950 | |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,050 |
| | READ(A,t) | | |
| | s:= t*0.1 | | |
| | t:= t-s | | |
| | WRITE(A,t) | 855 | |
| | READ(B,t) | | |
| | t:= t+s | | |
| | WRITE(B,t) | | 2,145 |

# Summary

- Concurrency Control
  - **Schedule**
    1. **Serial and Serializable Schedules;**
    2. Conflict-Serializable Schedules;
  - Locking Techniques
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

# Schedule

- A *Schedule* is a *time-ordered* sequence of actions belonging to one or more transactions: They represent the chronological order in which instructions are executed.
- A schedule is **Serial** if there is no interleaving – i.e., it consists of all the actions of one transaction, then all the actions of another transaction, and so on.
- Serial schedules are represented by the ordered list of the transactions composing the schedule – e.g., in case of two transactions either $(T_1; T_2)$ or $(T_2; T_1)$ are serial schedules.

# Serial Schedules: An Example

The banking example is a case of the serial schedule $(\mathtt{T}_1;\mathtt{T}_2)$.

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 2,000 |
| t := t-50 | | | |
| WRITE(A,t) | | 950 | |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,050 |
| | READ(A,t) | | |
| | s:= t*0.1 | | |
| | t:= t-s | | |
| | WRITE(A,t) | 855 | |
| | READ(B,t) | | |
| | t:= t+s | | |
| | WRITE(B,t) | | 2,145 |

# Another Serial Schedule: An Example

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| | READ(A,t) | 1,000 | 2,000 |
| | s:= t*0.1 | | |
| | t:= t-s | | |
| | WRITE(A,t) | 900 | |
| | READ(B,t) | | |
| | t:= t+s | | |
| | WRITE(B,t) | | 2,100 |
| READ(A,t) | | | |
| t := t-50 | | | |
| WRITE(A,t) | | 850 | |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,150 |

- **Note. The final values for $A, B$ are different in the two schedules:**
  1. $(T_1; T_2)$: $A = 855, \quad B = 2{,}145$;
  2. $(T_2; T_1)$: $A = 850, \quad B = 2{,}150$.
- Concurrency control deals only with preserving consistent DB states, while the DB state depends from the order of transactions.

# Serializable Schedules

- The **Consistency** property of transactions guarantees that every **serial schedule** will preserve consistency:
  *A transaction, if completed, will take the DB from a consistent state to another consistent state.*
- There is a possibility to interleave different transactions and still maintaining a consistent DB state.
- **Serializable Schedule.**
  **A schedule whose effect on the DB state is equivalent to that one of a serial schedule.**
- In the following we will try to characterize such Serializable Schedules.

# Serializable Schedules: An Example

- We refer here to our running example. This is a case of a **serializable –** but not **serial –** schedule.

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 2,000 |
| t := t-50 | | | |
| WRITE(A,t) | | 950 | |
| | READ(A,s) | | |
| | v:= s*0.1 | | |
| | s:= s-v | | |
| | WRITE(A,s) | 855 | |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,050 |
| | READ(B,s) | | |
| | s:= s+v | | |
| | WRITE(B,s) | | 2,145 |

- This schedule is equivalent to the serial schedule $(\mathtt{T_1}; \mathtt{T_2})$.

# Interleaving: A Not Working Example

- Interleaving has to be controlled. The following schedule is not serializable:

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 2,000 |
| t := t-50 | | | |
| | READ(A,s) | | |
| | v:= s*0.1 | | |
| | s:= s-v | | |
| | WRITE(A,s) | 900 | |
| | READ(B,s) | | |
| WRITE(A,t) | | 950 | |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,050 |
| | s:= s+v | | |
| | WRITE(B,s) | | 2,100 |

- The new DB state is *not consistent* with $A = 950$ and $B = 2{,}100$ – i.e., $A + B = 3{,}050 \neq 3{,}000$.

  The scheduler must avoid such situations.

# Summing Up

- **Transactions.** Only the Read and Write matter. Thus a transaction is a sequence of Read – $r_T(X)$ – and Write – $w_T(X)$ – actions on database elements.
  - If transactions are $T_1, \ldots, T_n$ , then $r_i$ and $w_i$ are used instead of $r_{T_i}, w_{T_i}$.
- **Schedule.** Sequence of Read/Write events performed by a collection of transactions.
- **Serial Schedule.** All actions for each transaction are consecutive.
  - Example: $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$.
- **Serializable Schedule.** One whose effect is guaranteed to be equivalent to that of some serial schedule.
  - Example: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$.

# Summary

- Concurrency Control
  - **Schedule**
    1. Serial and Serializable Schedules;
    2. **Conflict-Serializable Schedules;**
  - Locking Techniques
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

## Conflict

- **Objective**: introduce sufficient conditions to assure that a schedule is serializable.
- **Conflict**: Pair of consecutive actions in a schedule such that if their order is interchanged then the effects of one of the involved transactions may change.

# Conflict (cont.)

- **Pair of actions that do *not* conflict:**
  1. $\mathtt{r_i(X); r_j(Y)}$ is never a conflict even if the read element is the same since these actions do not change any value.
  2. $\mathtt{r_i(X); w_j(Y)}$ – or $\mathtt{w_i(X); r_j(Y)}$ – is not a conflict as far as $X \neq Y$.
  3. $\mathtt{w_i(X); w_j(Y)}$ is not a conflict as far as $X \neq Y$.
- **Pair of actions that *do* conflict:**
  1. Two actions of the *same* transaction. The order of actions of a single transaction is fixed and cannot be changed by the DBMS.
  2. $\mathtt{w_i(X); w_j(X)}$. Since the values written by $\mathtt{T_i}$ and $\mathtt{T_j}$ might be different, the next reading of $X$ will read different values depending on the order of these two actions.
  3. $\mathtt{r_i(X); w_j(X)}$ – and $\mathtt{w_i(X); r_j(X)}$. Swapping the order of $\mathtt{r_i(X)}$ and $\mathtt{w_j(X)}$ affects the value reads for $X$ by $\mathtt{T_i}$ and could affect what $\mathtt{T_i}$ does.

# Conflict-Serializable Schedules

- **Rule for swapping actions.**
  Any two actions of *different* transactions may be swapped unless:
  - They involve the *same* DB element, and at least one is a *Write*.
- Two schedules are said **Conflict-Equivalent** if they can be turned one into the other by a sequence of non-conflicting swaps of adjacent actions.
- A schedule is **Conflict-Serializable** if it is conflict-equivalent to a serial schedule – i.e., we can turn the schedule into a serial schedule by a sequence of non-conflicting swapping.
- Conflict-Serializability is a **sufficient** condition for serializability: *A Conflict-Serializable schedule is a serializable schedule – i.e., a schedule whose effect on the DB state is equivalent to that one of a serial schedule. The vice-versa is not always true*.

# Conflict-Serializable Schedules: An Example

- The following is an example of a *conflict-serializable* schedule:

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$

- We can turn the schedule into a serial schedule by the following swaps of non-conflicting adjacent actions:

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$$
$$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$$
$$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$$
$$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B)$$
$$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$$

- We obtain at the end the serial schedule $(T_1; T_2)$.

# Serializable Vs. Conflict-Serializable: An Example

**Example.** We show a Serializable schedule which is not Conflict-Serializable.

In our banking system we have now a transaction $\mathtt{T_3}$ that transfers $10\pounds$ from account $B$ to $A$.

| $T_1$ | $T_3$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 2,000 |
| t := t-50 | | | |
| WRITE(A,t) | | 950 | |
| | READ(B,s) | | |
| | s:= s-10 | | |
| | WRITE(B,s) | | 1,990 |
| READ(B,t) | | | |
| t := t+50 | | | |
| WRITE(B,t) | | | 2,040 |
| | READ(A,s) | | |
| | s:= s+10 | | |
| | WRITE(A,s) | 960 | |

Since $\mathtt{WRITE_3(B,s)}$ *conflicts* with $\mathtt{READ_1(B,t)}$ the schedule is not *conflict-equivalent* to the serial schedule $(\mathtt{T_1};\mathtt{T_3})$.

To realize that the schedule is equivalent to $(\mathtt{T_1};\mathtt{T_3})$ the computation performed by $\mathtt{T_1},\mathtt{T_3}$ on $B$ must be analyzed (rather that just READ and WRITE):

$B - 10 + 50 \equiv B + 50 - 10$ (since "+" & "−" are commutative).

# Precedence Graph

**Objective**: Procedure to decide whether a schedule is or not conflict-serializable.

- Conflicting pairs of actions put constraints on the order of transactions:
  - Let $S$ be a schedule and $A_i, A_j$ be two conflicting actions with $A_i <_S A_j$ (i.e., $A_i$ is before $A_j$ in $S$), then
  - Transactions $T_i, T_j$ performing those actions are such that $T_i$ **takes precedence over** $T_j$ – written, $T_i <_S T_j$ – meaning that
  - They must appear in the *same order* in any conflict-equivalent serial schedule.

# Precedence Graph (cont.)

- A **Precedence Graph** depicts the constraints on the order of transactions:
  1. Nodes are transactions;
  2. If $T_i <_S T_j$ there is an arch $T_i \rightarrow T_j$.
- **Rule for checking conflict-serializability.**
  If $T_i \rightarrow T_j$ then, in any serial schedule $S'$ equivalent to $S$, $T_i$ must appear before $T_j$.
  A schedule $S$ is conflict-serializable if and only if there is an **acyclic** precedence graph.

# Precedence Graph: Examples

**Example 1.**

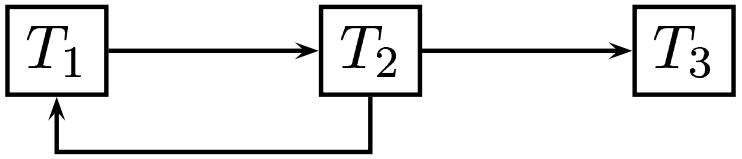
- $S$: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$.
- **Precedence Constraints:**
  - $r_2(A) <_S w_3(A)$, $w_2(A) <_S r_3(A)$, $w_2(A) <_S w_3(A)$, then $T_2 <_S T_3$;
  - $r_1(B) <_S w_2(B)$, $w_1(B) <_S r_2(B)$, $w_1(B) <_S w_2(B)$, then $T_1 <_S T_2$;
- **Precedence Graph:**


  - The precedence graph is **acyclic**, then the schedule is conflict-serializable.
  - The schedule $S$ is equivalent to the serial schedule $(\mathtt{T_1}; \mathtt{T_2}; \mathtt{T_3})$.

# Precedence Graph: Examples (cont.)

**Example 2.**

- $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$.
- **Precedence Constraints:**
  - $r_2(A) <_S w_3(A)$, then $T_2 <_S T_3$;
  - $r_1(B) <_S w_2(B)$, then $T_1 <_S T_2$;
  - $r_2(B) <_S w_1(B)$, then $T_2 <_S T_1$;
- **Precedence Graph:**

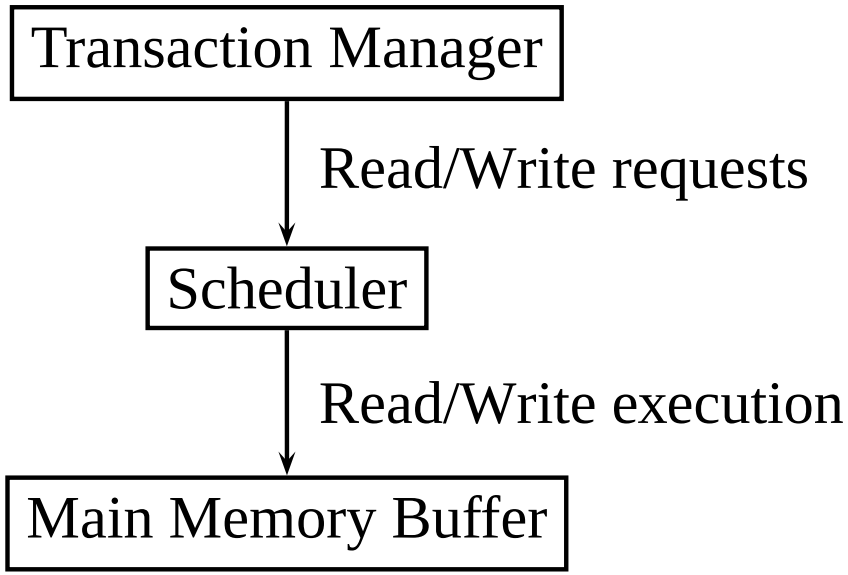



  - The precedence graph is **cyclic**, then the schedule is not conflict-serializable.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - **Locking Techniques**
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

# Scheduler and Serializability




- **Scheduler Job.** Manage the Read/Write actions of various concurrent transactions to prevent orders of actions that lead to not serializable schedules.
- The most common technique used by a scheduler is based on a **Locking** mechanism.
  - Transactions must posses a lock on a DB element to perform an operation on it.

# Locking Scheduler

- A **Locking Scheduler** enforces conflict-serializability – which is more stringent than serializability.
- The locking scheduler requires that data is accessed in a mutually exclusive manner.
- **Lock Actions**:
  1. A transaction $T_i$ requests a lock on a DB element X from the scheduler;
  2. The scheduler, using a *Lock Table* containing a list of the current locks, can either grant the lock to $T_i$ or make $T_i$ wait for the lock;
  3. If granted, $T_i$ should eventually unlock (release) the lock on the DB element $X$.
- Notation for locking and unlocking actions:
  1. $l_i(X)$: Transaction $T_i$ requests a lock on $X$;
  2. $u_i(X)$: $T_i$ releases the lock ("unlock") on $X$.

# Proper Use of Locks

The use of locks must be proper both w.r.t. the structure of transactions, and w.r.t. the structure of schedules.

- **Legal Conditions for Transactions**
  1. A transaction can Read or Write a DB element $X$ only when it holds a lock on $X$;
     - $r_i(X)$ or $w_i(X)$ must be preceded by a $l_i(X)$ with no occurrences of $u_i(X)$ in between.
  2. If a transaction locks an element, it must eventually unlock that element.
     - Every $l_i(X)$ must be followed by $u_i(X)$.
- **Legal Schedule**

  Two transactions cannot hold a lock simultaneously on the same DB element $X$.
  - A schedule with $l_i(X)$ cannot have another $l_j(X)$ until $u_i(X)$ appears.

# Serial Schedule: An Example

In a banking system two transactions $\mathtt{T}_1$ and $\mathtt{T}_2$ update two accounts – $A, B$.

- Current money in the accounts: $A = B = 1{,}000£$;
- $\mathtt{T}_1$ transfers $100£$ to both $A$ and $B$. $\mathtt{T}_2$ increments by $10\%$ both $A$ and $B$;
- The DB constraint is that $A = B$.

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| READ(A,t) | | 1,000 | 1,000 |
| t := t+100 | | | |
| WRITE(A,t) | | 1,100 | |
| READ(B,t) | | | |
| t := t+100 | | | |
| WRITE(B,t) | | | 1,100 |
| | READ(A,s) | | |
| | s:= s+s*0.1 | | |
| | WRITE(A,s) | 1,210 | |
| | READ(B,s) | | |
| | s:= s+s*0.1 | | |
| | WRITE(B,s) | | 1,210 |

# A Not Serializable Example

We consider now the case where $\mathtt{T}_1$ and $\mathtt{T}_2$ are executed concurrently and we show that having both **Legal Transactions** and a **Legal Schedule** is not sufficient.

- The following are **Legal Transactions** of the previous example:

$$\mathtt{T}_1: \quad \mathtt{l}_1(\mathtt{A});\mathtt{r}_1(\mathtt{A});\mathtt{A} := \mathtt{A} + 100;\mathtt{w}_1(\mathtt{A});\mathtt{u}_1(\mathtt{A});$$
$$\mathtt{l}_1(\mathtt{B});\mathtt{r}_1(\mathtt{B});\mathtt{B} := \mathtt{B} + 100;\mathtt{w}_1(\mathtt{B});\mathtt{u}_1(\mathtt{B})$$
$$\mathtt{T}_2: \quad \mathtt{l}_2(\mathtt{A});\mathtt{r}_2(\mathtt{A});\mathtt{A} := \mathtt{A} + \mathtt{A} * 0.1;\mathtt{w}_2(\mathtt{A});\mathtt{u}_2(\mathtt{A});$$
$$\mathtt{l}_2(\mathtt{B});\mathtt{r}_2(\mathtt{B});\mathtt{B} := \mathtt{B} + \mathtt{B} * 0.1;\mathtt{w}_2(\mathtt{B});\mathtt{u}_2(\mathtt{B})$$

# A Not Serializable Example (cont.)

- A possible **Legal Schedule** is:

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| $l_1(A)$;$r_1(A)$; | | 1,000 | 1,000 |
| A := A + 100; | | | |
| $w_1(A)$; $u_1(A)$; | | 1,100 | |
| | $l_2(A)$;$r_2(A)$; | | |
| | A := A + A * 0.1; | | |
| | $w_2(A)$; $u_2(A)$; | 1,210 | |
| | $l_2(B)$;$r_2(B)$; | | |
| | B := B + B * 0.1; | | |
| | $w_2(B)$; $u_2(B)$ | | 1,100 |
| $l_1(B)$;$r_1(B)$; | | | |
| B := B + 100; | | | |
| $w_1(B)$; $u_1(B)$ | | | 1,200 |

- The schedule is legal (the two transactions never hold a lock on the same element), **BUT** the schedule is **NOT Serializable** (in blue there are the confliciting actions).

  Then we need additional conditions.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - **Locking Techniques**
    1. **Two-Phase Locking;**
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

# 2PL: Two-Phase Locking

- **Two-Phase Locking.** Condition on the structure of transactions that guarantees conflict-serializability.
  - A transaction is *Two-Phase Locked*, or 2PL if *ALL* lock requests precede *ALL* unlock requests.
- The "two phases" are thus the first where locks are requested, and the second where locks are released.
- **Sufficient conditions for Conflict-Serializability.**
  - Given a set of **2PL** and **legal** transactions, and a **legal** schedule, then the schedule is also conflict-serializable.

# Two-Phase Locking: An Example

The two transactions in our banking example are legal but not 2PL:

$$\begin{aligned}
\mathtt{T}_1: \quad & \mathtt{l}_1(\mathtt{A});\mathtt{r}_1(\mathtt{A});\mathtt{A} := \mathtt{A} + 100;\mathtt{w}_1(\mathtt{A});\mathtt{u}_1(\mathtt{A}); \\
& \mathtt{l}_1(\mathtt{B});\mathtt{r}_1(\mathtt{B});\mathtt{B} := \mathtt{B} + 100;\mathtt{w}_1(\mathtt{B});\mathtt{u}_1(\mathtt{B}) \\
\mathtt{T}_2: \quad & \mathtt{l}_2(\mathtt{A});\mathtt{r}_2(\mathtt{A});\mathtt{A} := \mathtt{A} + \mathtt{A} * 0.1;\mathtt{w}_2(\mathtt{A});\mathtt{u}_2(\mathtt{A}); \\
& \mathtt{l}_2(\mathtt{B});\mathtt{r}_2(\mathtt{B});\mathtt{B} := \mathtt{B} + \mathtt{B} * 0.1;\mathtt{w}_2(\mathtt{B});\mathtt{u}_2(\mathtt{B})
\end{aligned}$$

We modify the banking example in such a way that $\mathtt{T}_1$ and $\mathtt{T}_2$ are now 2PL:

$$\begin{aligned}
\mathtt{T}_1: \quad & \mathtt{l}_1(\mathtt{A});\mathtt{r}_1(\mathtt{A});\mathtt{A} := \mathtt{A} + 100;\mathtt{w}_1(\mathtt{A});\mathtt{l}_1(\mathtt{B}); \\
& \mathtt{u}_1(\mathtt{A});\mathtt{r}_1(\mathtt{B});\mathtt{B} := \mathtt{B} + 100;\mathtt{w}_1(\mathtt{B});\mathtt{u}_1(\mathtt{B}) \\
\mathtt{T}_2: \quad & \mathtt{l}_2(\mathtt{A});\mathtt{r}_2(\mathtt{A});\mathtt{A} := \mathtt{A} + \mathtt{A} * 0.1;\mathtt{w}_2(\mathtt{A});\mathtt{l}_2(\mathtt{B}); \\
& \mathtt{u}_2(\mathtt{A});\mathtt{r}_2(\mathtt{B});\mathtt{B} := \mathtt{B} + \mathtt{B} * 0.1;\mathtt{w}_2(\mathtt{B});\mathtt{u}_2(\mathtt{B})
\end{aligned}$$

# Two-Phase Locking: An Example (cont.)

- A possible **Legal Schedule** is:

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| $l_1(A); r_1(A);$ | | 1,000 | 1,000 |
| $\texttt{A} := \texttt{A} + 100;$ | | | |
| $w_1(A);$ | | 1,100 | |
| $l_1(B); u_1(A);$ | | | |
| | $l_2(A); r_2(A);$ | | |
| | $\texttt{A} := \texttt{A} + \texttt{A} * 0.1;$ | | |
| | $w_2(A);$ | 1,210 | |
| | $l_2(B);$Denied | | |
| $r_1(B);$ | | | |
| $\texttt{B} := \texttt{B} + 100;$ | | | |
| $w_1(B); u_1(B)$ | | | 1,100 |
| | $l_2(B); u_2(A); r_2(B);$ | | |
| | $\texttt{B} := \texttt{B} + \texttt{B} * 0.1;$ | | |
| | $w_2(B); u_2(B)$ | | 1,210 |

- Now the schedule is also **Serializable** and indeed equivalent to the serial schedule $(T_1; T_2)$ (as you can see from the blue conflicting actions).

# Deadlock

**Problem.** Transactions can be forced by the scheduler to wait *forever* for a lock held by another transaction.

- **Example.** $T_2$ is changed to work on $B$ first:

  $\mathtt{T_1}: \mathtt{l_1(A); r_1(A); A := A + 100; w_1(A); l_1(B); u_1(A); r_1(B); B := B + 100; w_1(B); u_1(B)}$

  $\mathtt{T_2}: \mathtt{l_2(B); r_2(B); B := B + B * 0.1; w_2(B); l_2(A); u_2(B); r_2(A); A := A + A * 0.1; w_2(A); u_2}$

- A possible schedule is:

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| $l_1(A); r_1(A);$ | | 1,000 | 1,000 |
| | $l_2(B); r_2(B);$ | | |
| $\mathtt{A := A + 100;}$ | | | |
| | $\mathtt{B := B + B * 0.1;}$ | | |
| $w_1(A);$ | | 1,100 | |
| | $w_2(B);$ | | 1,100 |
| $l_1(B);\mathtt{Denied}$ | | | |
| | $l_2(A);\mathtt{Denied}$ | | |

- DBMS should deal with deadlocks: the system must abort and restart one or more transactions.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - **Locking Techniques**
    1. Two-Phase Locking;
    2. **Shared and Exclusive Locks;**
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.

# Locking with Several Lock Modes

- **Problem.** A transaction must take a lock on a DB element $X$ even if it reads $X$ but does not write $X$. There is no reason why several transactions could not read $X$ at the same time, as long as they don't write $X$.
- **Solution.** Introduce two kinds of locks: **Read Lock** (or **Shared Lock**), and **Write Lock** (or **Exclusive Lock**).
- We also examine the improved versions:
  1. *Upgrade Locks*: A transaction can "upgrade" a shared lock to an exclusive lock.
  2. *Update Locks*: Only "update" locks can be upgraded to exclusive locks.

# Shared and Exclusive Locks

- Multiple readers is not a problem for conflict-serializability since read actions—of different transactions—can commute.
- A scheduler operating under the **Shared and Exclusive Locks** policy is such that: for any DB element $X$ there can be either an *exclusive* lock, or no exclusive locks but any number of *shared* locks.
  1. **Shared lock** (**read lock**) $\texttt{sl}_\texttt{i}(\texttt{X})$: allows $T_i$ to read, but not to write $X$. It also prevents other transactions from writing $X$, but without preventing them from reading $X$.
  2. **Exclusive lock** (**write lock**) $\texttt{xl}_\texttt{i}(\texttt{X})$: allows $T_i$ to read and/or write $X$; no other transactions may either read or write $X$.

# Proper Use of Shared and Exclusive Locks

The use of locks must be proper both w.r.t. the structure of transactions, and w.r.t. the structure of schedules.

- **Legal Conditions for Transactions**

  1. A transaction can write a DB element $X$ only when it holds an *exclusive lock* on $X$, while it can read $X$ when holding some lock:
     (a) $r_i(X)$ must be preceded by a $sl_i(X)$ or $xl_i(X)$ with no occurrences of $u_i(X)$ in between;
     (b) $w_i(X)$ must be preceded by an $xl_i(X)$ with no occurrences of $u_i(X)$ in between;
  2. If a transaction locks an element, it must eventually unlock that element.

- **2PL for Transactions**

  Locking must precede unlocking: A transaction must not have a $sl_i(X)$ or $xl_i(X)$ after any $u_i(Y)$.

## Proper Use of Shared and Exclusive Locks (cont.)

- **Legal Schedule**

  A DB element may be either locked exclusively by one transaction, or by several in shared mode.

  1. If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$ until after an $u_i(X)$ appears;
  2. If $sl_i(X)$ appears, then there cannot be a following $xl_j(X)$ until after $u_i(X)$.

# Shared and Exclusive Locks: An Example

- Let $T_1, T_2$ be two *legal* transactions such that both $T_1$ and $T_2$ read $A$ and $B$, but only $T_1$ writes $B$:

  $T_1 : \ sl_1(A); r_1(A); xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B)$

  $T_2 : \ sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B)$

- A possible **Legal Schedule** is:

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A); r_1(A);$ | |
| | $sl_2(A); r_2(A);$ |
| | $sl_2(B); r_2(B);$ |
| $xl_1(B);$ `Denied` | |
| | $u_2(A); u_2(B)$ |
| $xl_1(B); r_1(B);$ | |
| $w_1(B);$ | |
| $u_1(A); u_1(B)$ | |

- **Note.** The schedule is conflict-serializable: the conflict-equivalent serial schedule is $(T_2; T_1)$. This is true in general for every **legal schedule** of **legal** and **2PL** transactions with shared and exclusive locks.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - **Locking Techniques**
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. **Upgrading Locks;**
    4. Update Locks
- The “Dirty” Data Problem.

# Upgrading Locks

- **Main Idea.** Instead of taking an exclusive lock immediately, a transaction can take a shared lock, read, and when ready to write **upgrade** to exclusive:
  - Request an exclusive lock on $X$ in addition to an already held shared lock on $X$.
- Upgrading locks allows more concurrent operation – see the next example.
- **Note.** Upgrading conforms to the legal conditions for both transactions and schedules as formulated for shared and exclusive locks apart from the second rule for legal schedules:

  **Legal Schedule: New Second Rule**

  2. If $sl_i(X)$ appears, then there cannot be a following $xl_j(X)$, for $j \neq i$, until after $u_i(X)$.

# Upgrading Locks: An Example

Let $\mathtt{T}_1, \mathtt{T}_2$ be two *legal* transactions – as given in the example on shared and exclusive locks – such that $\mathtt{T}_1$ and $\mathtt{T}_2$ read both $A$ and $B$, but only $\mathtt{T}_1$ writes $B$.

- We change $\mathtt{T}_1$ in a way that it first takes a shared lock on $B$, and later, after its computation on $A$ and $B$ is finished, $T_1$ requests an exclusive lock on $B$.

  $\mathtt{T}_1 : sl_1(A); r_1(A); sl_1(B); r_1(B); xl_1(B); w_1(B); u_1(A); u_1(B)$

  $\mathtt{T}_2 : sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B)$

- A possible **Legal Schedule** is:

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A); r_1(A);$ | |
| | $sl_2(A); r_2(A);$ |
| | $sl_2(B); r_2(B);$ |
| $sl_1(B); r_1(B);$ | |
| $xl_1(B);$`Denied` | |
| | $u_2(A); u_2(B)$ |
| $xl_1(B); w_1(B);$ | |
| $u_1(A); u_1(B)$ | |

- **Note.** The part in red shows the concurrent computation that was not possible if $T_1$ asked for an exclusive lock on $B$ initially – see the previous example.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - **Locking Techniques**
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. **Update Locks**
- The “Dirty” Data Problem.

# Update Locks

**Problem.** When we allow upgrades, it is easy to get into a deadlock situation.

- **Example.** $T_1$ and $T_2$ each read $A$ and later write $A$.

| $T_1$ | $T_2$ |
|---|---|
| $sl_1(A); r_1(A);$ | |
| | $sl_2(A); r_2(A);$ |
| $xl_1(A);$ `Denied` | |
| | $xl_2(A);$ `Denied` |

- **Solution.** **Update Locks** $ul_i(X)$:
  - An update lock gives only the permission to read, but not to write, $X$;
  - Only an update lock – no more a shared lock – can be upgraded to an exclusive lock;
  - An update lock on $X$ can be granted while there is a shared lock on $X$, but the scheduler will not grant any kind of lock on $X$ when there is an update lock on $X$ (apart from upgrading it to an exclusive lock).
  - The update lock looks like a shared lock when it is requested, and looks like an exclusive lock when it is already held.

# Update Locks: An Example

- Update locks would have no effect on the upgrading lock example – $T_1$ would take an update, rather than a shared, lock on $B$.
- Update locks solves the deadlock example: Now both $T_1$ and $T_2$ request an update lock on $A$:

  $T_1 : \ ul_1(A); r_1(A); xl_1(A); w_1(A); u_1(A)$

  $T_2 : \ ul_2(A); r_2(A); xl_2(A); w_2(A); u_2(A)$

| $T_1$ | $T_2$ |
|---|---|
| $ul_1(A); r_1(A);$ | |
| | $ul_2(A);$ `Denied` |
| $xl_1(A); w_1(A); u_1(A)$ | |
| | $ul_2(A); r_2(A);$ |
| | $xl_2(A); w_2(A); u_2(A)$ |

- **Note.** The update locks technique has eliminated the concurrent execution, but in this example, any significant amount of concurrency would result in either a deadlock or an inconsistent state.

# Summary

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - Locking Techniques
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- **The “Dirty” Data Problem.**

## The "Dirty" Data Problem

A data is **Dirty** if it has been written by an uncommitted transaction.

- A transaction can write a DB element $X$ and then abort, while a successive transaction can use the "dirty" value for $X$, resulting in an inconsistent state.
- **Note:** The problem we are considering here is dirty data stored in main memory buffers.

# The "Dirty" Data Problem: An Example

**Example.** We consider our running example and the following schedule:

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| $l_1(A); r_1(A);$ | | 1,000 | 1,000 |
| $\texttt{A} := \texttt{A} + \texttt{100};$ | | | |
| $w_1(A);$ | | 1,100 | |
| $l_1(B); u_1(A);$ | | | |
| | $l_2(A); r_2(A);$ | | |
| | $\texttt{A} := \texttt{A} + \texttt{A} * \texttt{0.1};$ | | |
| | $w_2(A);$ | 1,210 | |
| | $l_2(B); \texttt{Denied}$ | | |
| $r_1(B); \texttt{Abort}(\texttt{T}_1); u_1(B)$ | | | |
| | $l_2(B); u_2(A); r_2(B);$ | | |
| | $\texttt{B} := \texttt{B} + \texttt{B} * \texttt{0.1};$ | | |
| | $w_2(B); u_2(B)$ | | 1,100 |

- The value $A = 1,100$ written by the aborted transaction $T_1$ is a **dirty data**.
- We introduce two methods to prevent reading dirty data: *Cascading Rollback* and *Recoverable Schedule*.

# Cascading Rollbacks

- **Rules for applying the Cascading Rollback method.**
  1. When a transaction $T$ aborts, determine which transactions read data written by $T$, abort them, and iteratively abort all transactions that read data written by an aborted transaction.
  2. To cancel the effect of aborted transactions and fix main memory buffers containing dirty values an UNDO or UNDO/REDO logging technique is used. Indeed, old values are always needed.

# Recoverable Schedule

**Recoverable Schedule** is a simple method that eliminates the need for Cascading Rollbacks:

1. A transaction must not release any write lock until the transaction has either committed or aborted.
2. If a transaction aborts and writes dirty data in buffers then the buffers must be fixed before releasing the locks – generally using the log.

**Notes**.

1. The first rule is such that a transaction cannot read a dirty data, since data written by a transaction remains locked until either the transaction commits or the dirty buffers are recovered.
2. There is a disadvantage: Exclusive locks cannot be released as soon as possible but we need to unlock written data only at the end of a transaction. This can inhibit concurrency severely.

# Summary of Lecture VII

- Concurrency Control
  - Schedule
    1. Serial and Serializable Schedules;
    2. Conflict-Serializable Schedules;
  - Locking Techniques
    1. Two-Phase Locking;
    2. Shared and Exclusive Locks;
    3. Upgrading Locks;
    4. Update Locks
- The “Dirty” Data Problem.