CS525: Advanced Database Organization

Notes 8: Concurrency Control

Gerald Balekaki

Department of Computer Science

Illinois Institute of Technology

gbalekaki@iit.edu

July 31st, 2023

Slides: adapted from a course taught by Shun Yan Cheung, Emory University

Concurrency and Recovery

- DBMS should enable reestablish correctness of data in the presence of failures
  - System should restore a correct state after failure (recovery)
- DBMS should enable multiple clients to access the database concurrently
  - This can lead to problems with correctness of data because of interleaving of operations from different clients
    - System should ensure correctness (concurrency control)

Techniques to implement transactions

- `Logging`
  - Implements the `atomicity` property
  - Implements the `durability` property
- `Synchronization` (e.g.: locking)
  - Implements the `isolation` property
- The `consistency` property is assumed — otherwise, there is a bug in the `transaction`

Concurrent execution of transactions

- Concurrent execution of transactions can cause the database state to become inconsistent
- Example: Consider the following 2 transactions
  - T_1: transfer $1 from B to A
  - T_2: transfer $2 from B to A

```
T1:
   READ(A)
   A = A + 1
   WRITE(A)
   READ(B)
   B = B - 1
   WRITE(B)

 T2:
   READ(A)
   A = A + 2
   WRITE(A)
   READ(B)
   B = B - 2
   WRITE(B)
```

Serial (non-concurrent) execution

```
Initial data value:
   A = 10
   B = 10                  A + B = 20
--------------------------------------
T1:
   READ(A)
   A = A + 1
   WRITE(A)
   READ(B)
   B = B - 1
   WRITE(B)
--------------------------------------
   A = 11
   B = 9                   A + B = 20
--------------------------------------

 T2:
   READ(A)
   A = A + 2
   WRITE(A)
   READ(B)
   B = B - 2
   WRITE(B)
--------------------------------------
   A = 13
   B = 7                   A + B = 20
```

Concurrent execution

```
Initial data value:
   A = 10
   B = 10                        A + B = 20
-----------------------------------------------
T1:                     T2:
   READ(A)   (10)
                           READ(A)
                           A = A + 2
                           WRITE(A)     ==> A = 12
                           READ(B)
                           B = B - 2
                           WRITE(B)     ==> B = 8
   A = A + 1    (11)
   WRITE(A)     ==> A = 11
   READ(B)      (8)
   B = B - 1
   WRITE(B)     ==> B = 7
-----------------------------------------------
Final data value:
   A = 11
   B = 7                         A + B ≠ 20
-----------------------------------------------
```

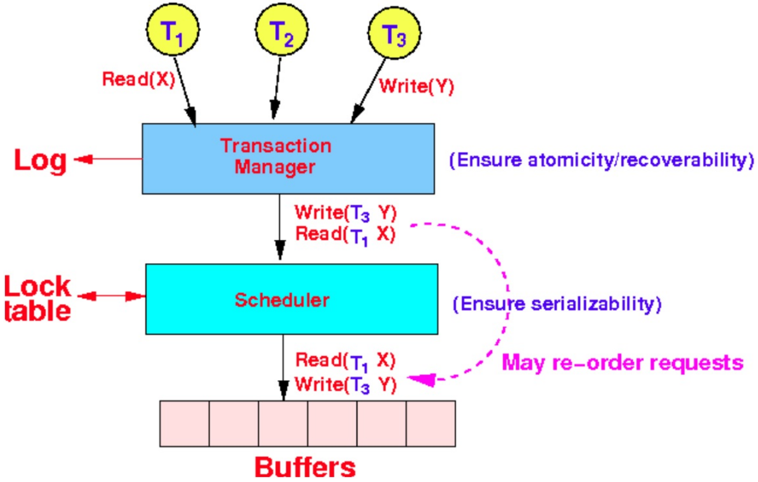
Observed fact from the above example

- The ordering of operations (actions) of different `transactions` can affect the `consistency` of the database
- To ensure `consistency` of the data:
    - We must impose a certain ordering on the operations from different `transaction`
    - Note: we still need to define what this “ordering” is, but we do know that we need some kind of ordering just by looking at the two examples

Scheduler and concurrency control

- `Scheduler`: the software component in a DBMS that is responsible for
  - regulating (scheduling) the executions of the actions (**`read/write`** operations) of the transactions
- `Concurrency control`: the procedure/algorithm to ensure/guarantee that:
  - a **`concurrent execution`** of the actions (operations) in **`(multiple) transactions`** will preserve the **`consistency`** of the database

Software organization of a DBMS



Software organization of a DBMS

- `Transactions` submit (read/write) requests for `database elements` to the `Transaction Manager`
- The `transaction manager` will write `log records` and then forward the (read/write) requests to the `Scheduler`
- The `Scheduler` can grant or delay the (read/write) requests based on the currently granted requests
  - If the read/write request is safe, the `scheduler` will grant the read/write request and the `transaction` can continue
  - If the read/write request is unsafe, the `scheduler` will delay the request and the `transaction` will wait
- A commonly used technique to decide on grant/delay is `Locking`
- Note:
  - In some `concurrency control techniques` (e.g., locking), the `scheduler` may need to `abort transactions`
  - This happens when `transactions` are `dead locked`

Topics that will be discussed

1. Define the correctness of a concurrent execution of transactions
    - This definition is called `serializability`
    - `Serializable schedule` (order of execution) will guarantee data consistency
2. `Serializability` turns out to be impractical to implement
    - We will look at a subset of correct schedules called `Conflict Serializability`
    - This notion of correctness can be implemented using `locks`
3. We will study how to implement `concurrent execution` of `transactions` are `conflict serializable` by using various kinds of locks
    - Exclusive locks
    - Shared/exclusive locks
    - Shared/update/exclusive locks
    - Shared/increment/exclusive locks

Serial schedules

- `Transaction` will transform a database from a `consistent state` to another `consistent state` if:
  - There are no `system failures`
  - The `transaction` is executed in isolation (no other `transaction` is active)
- Correctness of an execution of multiple transactions
  - `Multiple transactions` will transform a database from a `consistent state` to another `consistent state` if:
    - There are no `system failures`
    - Each `transaction` is executed in isolation (no other `transaction` is active)

Schedule

- `Schedule`: a sequence of (important) operations performed by one or more `transactions`
- In `concurrency control`, the important operations performed by `transactions` are:
  - read operations (`READ(X)`)
  - write operations (`WRITE(X)`)

Schedule: Example

```
We have 2 DB elements:
   A = 25
   B = 25                          (A = B)
----------------------------------------------------
We have 2 transactions:
   T1 : add 100 to database elements A and B
   T2 : double the value of database elements A and B
----------------------------------------------------
In code:
       T1               T2
   =================================
      READ(A,t)        READ(A,s)
      t = t + 100      s = 2×s
      WRITE(A,t)       WRITE(A,s)
      READ(B,t)        READ(B,s)
      t = t + 100      s = 2×s
      WRITE(B,t)       WRITE(B,s)
----------------------------------------------------
Example of a schedule:
      READ1(A,t)
      READ2(A,s)
      t = t + 100
      s = 2×s
      WRITE1(A,t)
      WRITE2(A,s)
      READ1(B,t)
      t = t + 100
      WRITE1(B,t)
      READ2(B,s)
      s = 2×s
      WRITE2(B,t)
```

Correctness consideration when executing T_1 and T_2

```
When T1 or T2 are executed in isolation:
   T1
   READ(A,t)
   t = t + 100
   WRITE(A,t)
   READ(B,t)
   t = t + 100
   WRITE(B,t)


  T2
   READ(A,s)
   s = 2×s
   WRITE(A,s)
   READ(B,s)
   s = 2×s
   WRITE(B,t)

T1 and T2 will preserve the following property:

       A = B
```

Serial schedule

- Serial schedule: a schedule where the all operations of each transactions are executed in sequence
- i.e., operations from different transactions are not interspersed with one another
- Example:

```
Execute T1 then T2:  // Subscript indicate the transaction
   READ1(A,t)
   t = t + 100
   WRITE1(A,t)
   READ1(B,t)
   t = t + 100
   WRITE1(B,t)

   READ2(A,s)
   s = 2×s
   WRITE2(A,s)
   READ2(B,s)
   s = 2×s
   WRITE2(B,t)
```

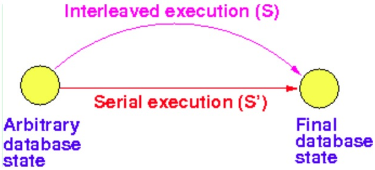
- Representation of a serial schedule
  - We can represent a serial schedule by a list of the transaction IDs.
    - T_1 T_2

Database consistency and serial schedules

- A `serial schedule` executes each `transaction` in isolation
- Therefore, every `serial schedule` will preserve the `consistency (correctness)` of the database
- Conclusion: We can use a `serial schedule` as the basis for the definition of correctness for `Concurrent execution` of `transactions`

Serializable schedule

- A `schedule S` is `serializable schedule` iff:
  - There is a **`serial schedule S'`** such that the effect of the execution of `S` and `S'` are identical for every `DB state`
- Starting in any `DB state`, executing the `schedules S/S'` will result in the same `DB state`



Example: serializable schedule

```
        T1                 T2                   A          B
    =====================================================
   |    READ(A,t)                               c          c
   |    t = t + 100
   |    WRITE(A,t) --------------------->  c+100
   |                     READ(A,s)
   |                     s = 2×s
   |                     WRITE(A,s) ---->  2c+200
   |    READ(B,t)
   |    t = t + 100
   |    WRITE(B,t) ------------------------------> c+100
   |                     READ(B,s)
   |                     s = 2×s
   v                     WRITE(B,s) -------------> 2c+200
  time
```

The interleaved execution (schedule) is equivalent to the following serial schedule:

```
        T1                 T2                   A          B
    =====================================================
        READ(A, t)                              c          c
        t = t + 100
        WRITE(A, t) -------------------->  c+100
        READ(B, t)
        t = t + 100
        WRITE(B, t) -----------------------------> c+100
                         READ(A,s)
                         s = 2×s
                         WRITE(A,s) ---->  2c+200
                         READ(B,s)
                         s = 2×s
                         WRITE(B,s) -------------> 2c+200
```

The schedule is therefore serializable

Non-serializable schedules

- Consider the following schedule:

```
      T1              T2
=================================
   READ(A,t)
   t = t + 100
   WRITE(A,t)

                   READ(A,s)
                   s = 2×s
                   WRITE(A,s)
                   READ(B,s)
                   s = 2×s
                   WRITE(B,s)

   READ(B,t)
   t = t + 100
   WRITE(B,t)
```

Non-serializable schedules

- The execution will result in an **inconsistent database state**:

```
          T1                T2                  A         B
     =======================================================
    |     READ(A,t)                             c         c
    |     t = t + 100
    |     WRITE(A,t) --------------------->     c+100
    |
    |                       READ(A,s)
    |                       s = 2×s
    |                       WRITE(A,s) ---->    2c+200
    |
    |                       READ(B,s)
    |                       s = 2×s
    |                       WRITE(B,s) ------------->     2c
    |
    |     READ(B,t)
    |     t = t + 100
    |     WRITE(B,t) ----------------------------->       2c+100
    V
  time
 =======================================================
  At the end of the execution, we have:

   A = 2c + 100
   B =  c + 100
```

- This result can never be obtained by a **serial execution** of T_1 and T_2
- Therefore, the **schedule** is not **serializable**

Determinant of serializable behavior

- To determine whether an `interleaved execution` of `transaction` is `serializable`, we must discard the semantics (operations performed) of the `transaction`
- The `serializable` of an `interleaved execution` of `transaction` is only determined by the interleaving of their `READ(...)` and `WRITE(...)` operations

Example 1

The schedule of T_1 and T_3:

```
        T1                T3                 A         B
   =======================================================
        READ(A,t)                            c         c
        t = t + 100
        WRITE(A,t) ---------------------->   c+100
                          READ(A,s)
                          s = s + 200
                          WRITE(A,s) ---->   c+300
                          READ(B,s)
                          s = s + 200
                          WRITE(B,s) -------------->  c+200
        READ(B,t)
        t = t + 100
        WRITE(B,t) -------------------------------->  c+300
```

The schedule without semantics:

```
          T1                T3
        ======================================
   |      READ(A,t)
   |      WRITE(A,t)
   |                        READ(A,s)
   |                        WRITE(A,s)
   |                        READ(B,s)
   |                        WRITE(B,s)
   |      READ(B,t)
   |      WRITE(B,t)
   V
  time
```

Example 2

The schedule of T_1 and T_2:

```
        T1                 T3                 A         B
=======================================================
        READ(A,t)                             c         c
        t = t + 100
        WRITE(A,t) --------------------->  c+100
                          READ(A,s)
                          s 2×s
                          WRITE(A,s) ---->  c+300
                          READ(B,s)
                           2×s
                          WRITE(B,s) --------------> c+200
        READ(B,t)
        t = t + 100
        WRITE(B,t) -----------------------------> c+300
```

The schedule without semantics:

```
        T1                 T2
    ======================================
  |     READ(A,t)
  |     WRITE(A,t)
  |                        READ(A,s)
  |                        WRITE(A,s)
  |                        READ(B,s)
  |                        WRITE(B,s)
  |     READ(B,t)
  |     WRITE(B,t)
  V
 time
```

The `schedules` without semantics (meaningful operations) in `examples 1` and `2` are identical.

Determinant of serializable behavior

- The criteria to decide if a `schedule` is
  - `Serializable` or
  - `Not serializable`
- must be based solely on the ordering of the `READ(.)` and `WRITE(.)` operations in the `schedule`

Notation for a schedule

- $r_i(X)$: transaction T_i reads the DB element X
- $w_i(X)$: transaction T_i writes the DB element X
- Example: notation for a schedule

```
Schedule:
          T1                  T2
       ================================
         READ(A,t)
         WRITE(A,t)
                              READ(A,s)
                              WRITE(A,s)
                              READ(B,s)
                              WRITE(B,s)
         READ(B,t)
         WRITE(B,t)
```

- Notation: $r_1(A)$ $w_1(A)$ $r_2(A)$ $w_2(A)$ $r_2(B)$ $w_2(B)$ $r_1(B)$ $w_1(B)$

Conflict-serializability: a more practical type of serializability

- General definition of serializability:
  - `S` is `serializable` iff:
    - For every `DB state`, the effect of the execution of `S` is `identical/equivalent` to some `serial schedule`
- The general definition of `serializability` is difficult to apply (detect, verify).
- We have therefore define different kinds of `serializability` that are more practical (easier to detect):
  - `Conflict-serializability`: based on the concept of conflicting operations (This type of `serializability` can be easily enforced by `locks`)
  - `View-serializability`: based of the concept of view (what did the `transaction` "see" (read)) (This type of `serializability` is more general but harder to enforce)
- The most common form of `serializability` implemented in `DMBS` is `Conflict-serializability`

Conflicting operations

- `2` consecutive operations in a `schedule S` (`S = ... op1 op2 ...`) is conflicting iff the behavior/result of at least one operation is changed when their order is interchanged

Interactions between the basic operations

- Recall: The basic operations used by transactions to operate on the database are `READ(.)` and `WRITE(.)`
- A `READ(.)` and `WRITE(.)` operation can operate:
  - On the same database elements
  - On different database elements

Non-conflicting and conflicting operations

- The interaction of `2` basic operations is `non-conflicting` if:
  - The ordering of the execution of the basic operation does not affect the outcome
  - Otherwise, the operations are `conflicting`
- For a `READ(.)` operation, the outcome is the return value of the read
- For a `WRITE(.)` operation, the outcome is the final value of the `database element` that is written

Read/write operations within the same transaction

- Read/write operations within a transaction will be executed in a certain order
- We assume that the ordering of read/write within one transaction is fixed and cannot be altered
- I.e., Operations from the same transaction can not be executed in a different order

Summary on conflicting operations

- 2 `actions` from the `same transaction` are `conflicting`
- 2 `actions` from `different transactions` are `conflicting` if
  - They operate on the same `DB element`
  - At least one of the operation is `WRITE`

1. $\texttt{r}_i\texttt{(X); w}_j\texttt{(X)}$
2. $\texttt{w}_i\texttt{(X); r}_j\texttt{(X)}$
2. $\texttt{w}_i\texttt{(X); w}_j\texttt{(X)}$

Re-ordering non-conflicting operations

- If `2 actions` `op1` and `op2` are `non-conflicting`, we can:
  - `Re-order` their execution in a `schedule` (and the result of the execution will remain unchanged)

Conflict-serializable schedules: conflict-serializability

- Conflict-equivalent schedules
  - `2 schedules` S_1 and S_2 are `conflict equivalent` iff
    - `Schedule` S_1 can be transformed into `schedule` S_2 by a sequence of `non-conflicting swaps` of adjacent actions
- Example
  - Consider the following `2 schedules`

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

    S_2: $r_1(A)$ $w_1(A)$ $r_2(A)$ $r_1(B)$ $w_2(A)$ $w_1(B)$ $r_2(B)$ $w_2(B)$
  - The `schedules` are `conflict equivalent` because:

    S_1: $r_1(A)$ $w_1(A)$ $r_2(A)$ $\boxed{w_2(A)\ r_1(B)}$ $w_1(B)$ $r_2(B)$ $w_2(B)$

    S_2: $r_1(A)$ $w_1(A)$ $r_2(A)$ $\boxed{r_1(B)\ w_2(A)}$ $w_1(B)$ $r_2(B)$ $w_2(B)$

Conflict-serializable schedules: conflict-serializability

- Conflict-serializable schedules
  - A **schedule** S_1 is **conflict serializable** iff
    - **Schedule** S_1 can be transformed into a **serial schedule** by a sequence of **non-conflicting swaps** of adjacent actions
- Example
  - Consider the following **schedule**

    S_1: $r_1(A)$ $w_1(A)$ $r_2(A)$ $w_2(A)$ $r_1(B)$ $w_1(B)$ $r_2(B)$ $w_2(B)$
  - We perform the following sequence of **non-conflicting swaps** and obtain a **serial schedule**

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

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

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

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

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

    S_1: T_1 T_2
  - $\therefore$ The **schedule** S_1 is a **conflict-serializable schedule**

Conflict-serializable schedule are always serializable

- Swapping two non-conflicting operations in a schedule will not change the effect of the execution of the operations in a schedule
- Therefore, the effect of the execution of a conflict-serializable schedule is equivalent to the execution of a serial schedule of the transactions
- So by definition of serializability, a conflict-serializable schedule is a serializable schedule

Schedules can be serializable but not conflict-serializable

- Example of a schedule that is `serializable` but `non-conflict serializable`
  - Consider the following `3 transactions`

    T_1: $w_1(Y)$ $w_1(X)$

    T_2: $w_2(Y)$ $w_2(X)$

    T_3: $w_3(X)$
  - Consider the following `2 schedules` with the `3 transactions`: (Schedule S_2 is a `serial schedule`)

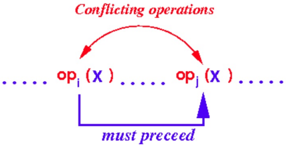
    S_1: $w_1(Y)$ $w_2(Y)$ $w_2(X)$ $w_1(X)$ $w_3(X)$

    S_2: $w_1(Y)$ $w_1(X)$ $w_2(Y)$ $w_2(X)$ $w_3(X)$
  - The effect (state of the database) of schedules S_1 and S_2 are identical

    - The database state: the updated made by the last write operation
  - By definition: `Schedule` S_1 is equivalent to a serial schedule
  - However, we cannot transform the `schedule` S_1 into a serial schedule by swapping adjacent `non-conflicting actions`
  - ∴ The `schedule` S_1 is not a `conflict-serializable schedule`

Precedence graph test for conflict-serializability

- The reason why do we use `conflict-serializability`, there is a simple procedure the test/check whether a `schedule S` is `conflict-serializable` or `not conflict-serializable`
- The precedence relationship
- Observation
  - If we find `2 conflicting actions` (anywhere in a schedule)



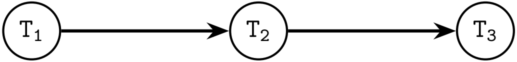
  - then T_i must precede T_j in a `serial schedule` (because op_j`(X)` can never be swap with op_i`X)`)
  - Therefore, the ordering in a `serial schedule` must be as follows

Precedence graph test for conflict-serializability

- This observation has determined a `precedence relationship` between `transactions` T_i `and` T_j.
  - Specifically, T_i must precede T_j (in a `serial schedule`)
- Representing (all) the precedence relationships in a `schedule S`
  - We use a graph to represent all `precedence relationships` found in a `schedule S`
    - For every pair of conflicting actions in a `schedule S`
      we add the edge $T_i \rightarrow T_j$ to the graph

Example

- S_1: $r_2(A)$ $r_1(B)$ $w_2(A)$ $r_3(A)$ $w_1(B)$ $w_3(A)$ $r_2(B)$ $w_2(B)$
- Graph with all `precedence relationships`
  - Specifically, T_i must precede T_j (in a `serial schedule`)
- Representing (all) the precedence relationships in a `schedule S`

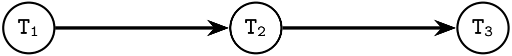


A simple test for conflict-serializable schedules

- A `schedule S` is `conflict-serializable` iff the corresponding `precedence graph` of `schedule S` does not contain any `cycles`
- Example:

  S_1: $r_2(A)$ $r_1(B)$ $w_2(A)$ $r_3(A)$ $w_1(B)$ $w_3(A)$ $r_2(B)$ $w_2(B)$

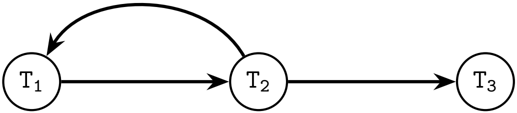
  - Graph with all `precedence relationships`



  - The `serial schedule` is: T_1 T_2 T_3
  - Proof: swap the `non-conflicting operations`

A simple test for conflict-serializable schedules: Example 2

- S_1: $r_2(A)$ $r_1(B)$ $w_2(A)$ $r_2(B)$ $r_3(A)$ $w_1(B)$ $w_3(A)$ $w_2(B)$
- Graph with all precedence relationships



- This schedule is not conflict-serializable because there is a cycle
- Proof
  - To make a serial schedule with the order T_1 ... T_2 we must swap this pair of conflicting action:

    S_1: $r_2(A)$ $r_1(B)$ $w_2(A)$ $r_2(B)$ $r_3(A)$ $w_1(B)$ $w_3(A)$ $w_2(B)$
  - To make a serial schedule with the order T_2 ... T_1 we must swap this pair of conflicting action:

    S_1: $r_2(A)$ $r_1(B)$ $w_2(A)$ $r_2(B)$ $r_3(A)$ $w_1(B)$ $w_3(A)$ $w_2(B)$
  - Therefore: there is no way to swap non-conflicting actions to convert the schedule into a serial schedule

Intro to locks

- Overview: techniques to enforce `conflict-serializable` behavior

Techniques to enforce `conflict-serializable` behavior

- `Locks`
  - Exclusive locks
  - Shared/Exclusive locks
  - Shared/Update/Exclusive locks
  - Shared/Increment/Exclusive locks
- Locking will also require `2-phase locking` to be sufficient (later)
- Timestamps
  - Single-value timestamp
  - Multi-value timestamp
- Validation
  - Verification

Lock

- Lock: an object that have:
  - A state (locked, unlocked, etc)
  - A number of lock holders
  - A queue of pending (locking/unlocking) operations
- Some `lock operation` will be `granted` (succeed) when the lock is in a certain state
  - If a `lock operation` is `granted`, the execution will continue
- Some `lock operation` be `denied (fail)` when the lock is in a certain state
  - If a `lock operation` is `denied`, the execution will wait until the state of the lock changes to a compatible state
  - When the state of the lock becomes compatible, the `lock operation` will be `granted` and the execution will be continued

Transactions, conflict-serializability and locks

- Uncontrolled access to `DB elements` will result in `non-serializable` behavior
- Locks can alter the order of the execution of operations
  - By denying a request for a lock, we can alter the order of the execution of certain operations
- Goal
  - Define **`grant/deny rules`** so that the modified order of operations is `conflict-serializable`

Exclusive locks

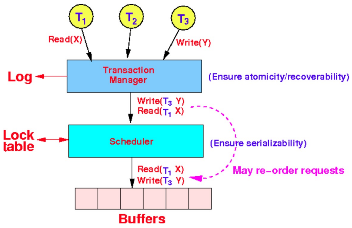
- Lock states
  - unlocked
  - (exclusive) locked
- Lock operations
  - `lock`
    - `lock requests` is granted if the `lock state` is `unlocked`
      - The state of the lock becomes locked
    - `lock requests` is denied if the `lock state` is `locked`
  - `unlock`
    - The state of the lock will become `unlocked`
      - Only applicable if the transaction currently hold the lock
- Notations
  - l_i(X): `transaction i` requests a `lock` on `DB element X`
  - u_i(X): `transaction i` requests to `unlock` the `DB element X`

Locking rules for exclusive locks

- Rules
  1. When a `transaction T` want to perform an action (read/write) on a `DB element X`, `transaction T` must first obtain the `lock` on `DB element X`
  2. The `transaction T` must release all locks before it terminates

  - T_i: ... $l_i(X)$ $r_i(X)$ ... $u_i(X)$ ...(read)
  - T_i: ... $l_i(X)$ $w_i(X)$ ... $u_i(X)$ ...(write)
- Note: `transactions` must hold the lock to perform an action. So this is not allowed
  - ... $l_i(X)$ ... $u_i(X)$ $w_i(X)$ ...(write)

The lock scheduler



- `Lock scheduler`: a scheduler that uses locks to make re-ordering decisions in the execution of the actions of the `transactions`
  - The `scheduler` will grant the `lock request` for an operation if the request cannot result in an `inconsistent state`
  - The `scheduler` will deny the `lock request` for an operation if the request can result in an `inconsistent state`

Lock table: an intro

- ~~The exact implementation of a lock table will be discussed later~~
- For now, imagine that a `lock table` is a set of tuples of the following format:
  - `(DB Element, Transaction holding lock)`
- Example:
  - (X, T_1)
  - (Y, T_2)
- Note: Due to the `locking rule` of `exclusive locks`, only one `transaction` can hold a lock

Example of locking

- Transactions

```
T₁:                          T₂:
    r₁(A)                        r₂(A)
    A = A + 1                    A = 2 × A
    w₁(A)                        w₂(A)

    r₁(B)                        r₂(B)
    B = B - 1                    B = 2 × B
    w₁(B)                        w₂(B)
```

- Transactions augmented with lock operations

```
T₁:                          T₂:
    l₁(A)                        l₂(A)
    r₁(A)                        r₂(A)
    A = A + 1                    A = 2 × A
    w₁(A)                        w₂(A)
    u₁(A)                        u₂(A)

    l₁(B)                        l₂(B)
    r₁(B)                        r₂(B)
    B = B - 1                    B = 2 × B
    w₁(B)                        w₂(B)
    u₁(B)                        u₂(B)
```

Example of a schedule with locks

```
        T1                T2                 A        B
=======================================================
        l1(A) r1(A)<--------------------  25       25
        A = A + 100
        w1(A) u1(A) --------------------> 125
                                          :
                          l2(A) r2(A)<-----
                          A = 2 × A
                          w2(A) u2(A) --> 250
                          l2(B) r2(B)<------------ 25
                          B = 2 × B
                          w2(B) u2(B) -----------> 50
                                                   :
         l1(B) r1(B)<-------------------------------
         B = B + 100
         w1(B) u1(B) -----------------------------> 150
```

- Notice that
  - The **transactions** obeys the **locking rules**

    $\ldots\ l_i(X)\, r_i(X) \ldots w_i(X)\, u_i(X) \ldots$

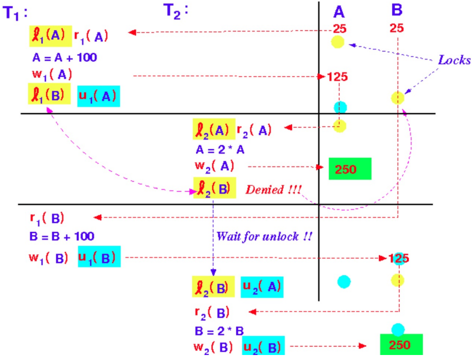
  - However, the resulting schedule is **non-serializable**.
    - (Because **serial schedules** of **transactions** T_1 and T_2 will preserve the property A = B)
- Conclusion: The **locking rules** is insufficient to ensure **(conflict)-serializability**

Example 2 of locking (with a correct outcome)

Consider a slightly changed order in the transactions: We re-order these 2 unlock/lock operations in the previous example:

```
T1:                 T2:
   l1(A)               l2(A)
   r1(A)               r2(A)
   A = A + 1           A = 2 × A
   w1(A)               w2(A)
   l1(B)               l2(B)

   u1(A)               u2(A)
   r1(B)               r2(B)
   B = B - 1           B = 2 × B
   w1(B)               w2(B)
   u1(B)               u2(B)
```



Example 2 of locking (with a correct outcome)

- Conclusion
  - The execution results in a `consistent state A = B`
  - So `locks` can force a `schedule` to change the order of execution of the actions so that the resulting execution of the `schedule` is `conflict-serializable`
- Question: What is the secret ingredient that we need to add to make `locking` a sufficient technique to ensure `conflict-serializability`

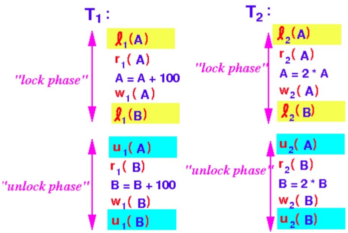
Two Phase Locking (2PL)

- `Two Phase Locking`: ordering of locking operations in a `transaction` where all the `lock operations` **precedes** all the `unlock operations`

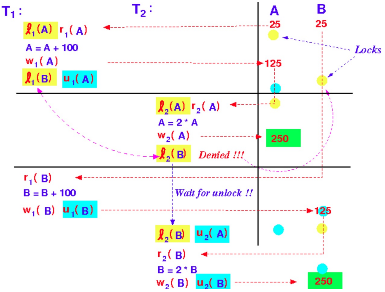
```
Transaction:
        lock(A)
        ...
        lock(B)
        ...
        ...
        lock(X)
        ...
        ...
        unlock(B)
        ...
        unlock(X)
        ...
        unlock(A)
        ...
```

- The `2 phases`
  - `Phase 1`: Acquire locks
  - `Phase 2`: Release locks

Two Phase Locking (2PL): Example



- Notice: All `lock` operations **precedes** all `unlock` operations
- We saw in the previous example (slides 53) that execution using `2-phase locking` results in a `consistent database state`



Two Phase Locking (2PL)

- Recall: `Transactions` using non `2-phase locking` can result in a `inconsistent DB state`
- As in previous example (slides 51, 52), the `transactions` are `not two-phase` since the `unlock operation` $u_i(A)$ happens before an `lock operation` $l_i(B)$
- Results in execution results in `inconsistent database state`

```
      T1                T2                A         B
=======================================================
      l1(A) r1(A)<--------------------  25        25
      A = A + 100
      w1(A) u1(A) --------------------> 125
                                          :
                        l2(A) r2(A)<-----
                        A = 2 × A
                        w2(A) u2(A) --> 250
                        l2(B) r2(B)<------------ 25
                        B = 2 × B
                        w2(B) u2(B) -----------> 50
                                                    :
       l1(B) r1(B)<-------------------------------
       B = B + 100
       w1(B) u1(B) ----------------------------> 150
```

- Proof: `2PL` is sufficient condition for conflict-serializability (omitted)

Deadlock

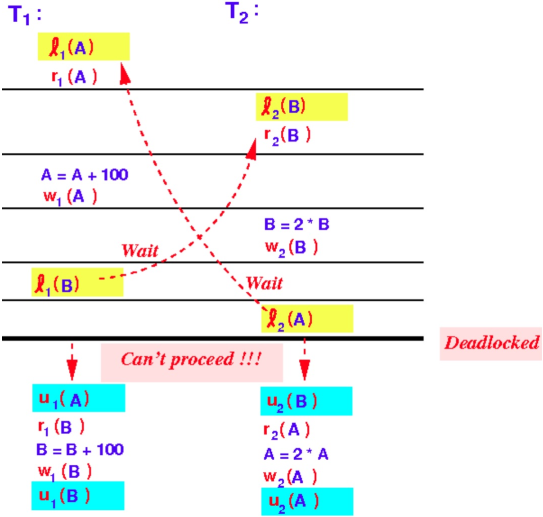
- `Deadlock`: a state of execution where `two or more transactions` are unable to proceed because the `transactions` involved are waiting of each other's locks
- Example

```
Consider the following 2 transactions:
T1:                 T2:
   l1(A)               l2(B)
   r1(A)               r2(B)
   A = A + 1           B = 2 × B
   w1(A)               w2(B)
   l1(B)               l2(A)

   u1(A)               u2(B)
   r1(B)               r2(A)
   B = B - 1           A = 2 × A
   w1(B)               w2(A)
   u1(B)               u2(A)
```

Deadlock: Example

- A possible interleaved execution (that obeys the locking rules) is

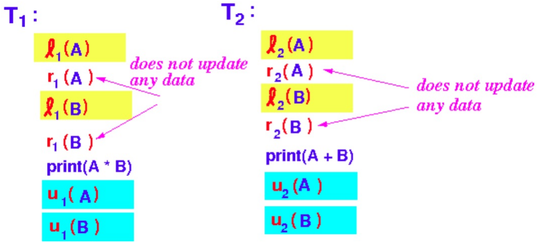


2PL and deadlocks

- `2PL` can not prevent `deadlocks`
  - In the previous example (slides 58,59), both `transactions` are `2PL`.
  - However, their execution resulted in `deadlock`
- Deadlock detection
  - using time out
  - using wait-for graph
- Deadlock prevention
  - using ordered DB elements
  - using wait-die timestamp scheme
  - using wound-wait timestamp scheme

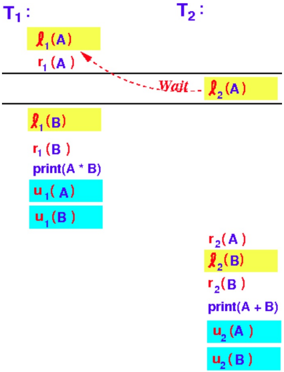
Shared/exclusive locks

- Consider the following `2 transactions`



- Notice that: $\mathtt{T}_1$ and $\mathtt{T}_2$ only reads the `DB elements A` and `B`

- A possible schedule of T_1 and T_2



- T_2 could proceed when T_1 holds the `lock` for `DB element A` because the value of the `DB element A` will not be changed by T_1. So there was no technical reason to block the T_2
- The `exclusive lock` is too restrictive in some situations

Shared/exclusive locks: locking method

- Uses 2 kinds of locks:
  - Shared lock
  - Exclusive lock
- `DB element X` can have one of the following locks
  - 1 exclusive lock
  - any number of shared locks
- Transactions using shared/exclusive locking
  - In order to perform a **`read operation`** on a `DB element X`, the **`transaction`** must be holding a **`shared lock`** on the `DB element X`
  - In order to perform a **`write operation`** on a `DB element X`, the **`transaction`** must be holding an **`exclusive lock`** on the `DB element X`
  - A **`shared/exclusive lock operation`** must be followed by an **`unlock operation`**

Shared/exclusive locks: Notation

- $\mathtt{sl}_i(\mathtt{X})$: transaction T_i requests a **shared lock** on the DB element X
- $\mathtt{xl}_i(\mathtt{X})$: transaction T_i requests an **exclusive lock** on the DB element X
- $\mathtt{u}_i(\mathtt{X})$: transaction T_i **unlocks** on the DB element X

  - $\mathtt{u}_i(\mathtt{X})$ can only be applied if transaction T_i currently holds a (shared or exclusive) lock on the DB element X

Requirements to ensure conflict-serializability

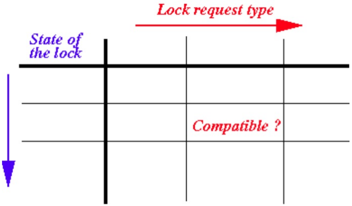
- The following 3 types of requirements can guarantee that current execution transactions will be `conflict-serializable`
  - `Transaction consistency (semantics)`
    - In order to perform a `read operation` on a `DB element X`, the `transaction` must be holding a `shared lock` on the `DB element X`
    - In order to perform a `write operation` on a `DB element X`, the `transaction` must be holding an `exclusive lock` on `DB element X`
    - A `shared/exclusive lock operation` must be followed by an `unlock operation`
  - `2 Phase Locking`: All the `unlock operations` must follow all the `lock operations` within one `transaction`
  - `Legality` of the lock schedule
    - A `DB element X` can be in one of the following states
      - unlocked
      - exclusively locked by one transaction (with no shared locks)
      - shared locked by one or more transactions

Note

- A transaction cannot hold on the same `DB element`
  - a shared lock and
  - an exclusive lock
- Therefore, a `transaction` that `read` the `DB element X` and then `write` the `DB element X` must request an `exclusive lock` for the `DB element X`

Lock compatibility matrix

- If we use several lock modes, then the scheduler needs a policy about when it can grant a lock request, given the other locks that may already be held on the same database element
- There is a clearer way to represent the `compatibility` of `locks` by using a `compatibility matrix`
- Structure of a compatibility matrix



- `State of a lock`: the type of lock that has been granted
- `Lock request type`: the type of lock being requested
- The compatibility decision is
    - `Yes`, if the `requested lock type` can be `granted`
    - `No`, if the `requested lock type` can `not be granted`

Lock compatibility matrix for shared/exclusive locks

- Full compatibility matrix for shared/exclusive locks:

Lock request type

State of the lock	Shared	Exclusive
Unlock	Yes	Yes
Shared	Yes	No
Exclusive	No	No

Trivial

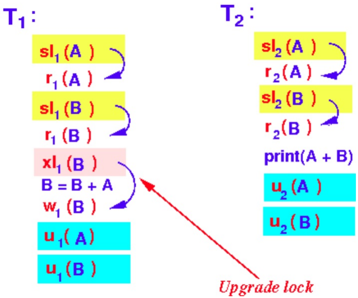
- Abbreviated compatibility matrix for shared/exclusive locks:

Lock request type

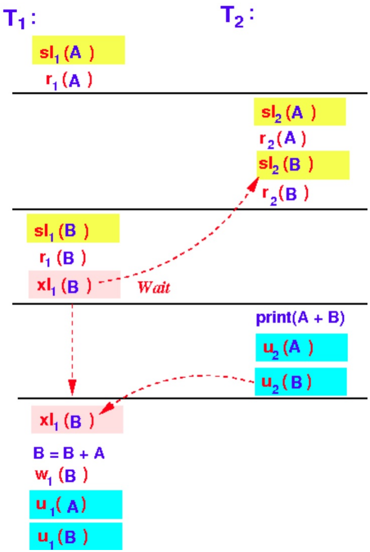
State of the lock	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Upgrading locks

- Technically it's very easy to implement upgradable locks
  - A **transaction T** that wants to **read** a **DB element** first acquires a **shared lock**
    - We set the state of the lock to SHARED
  - When **transaction T** wants to **write** a **DB element** later, it can update the **shared lock** to an **exclusive lock**
    - We simply set the state of the lock to EXCLUSIVE
- Example



Example of interleaved execution (possible schedule)



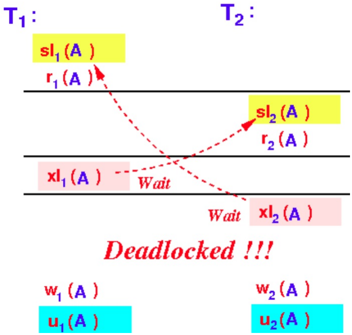
The danger of upgrading locks

- `lock upgrade` can result in `deadlock` with just `one DB element`

Consider the following 2 transactions:

T_1:	T_2:
$sl_1(A)$	$sl_2(A)$
$r_1(A)$	$r_2(A)$
$xl_1(A)$	$xl_2(A)$
$w_1(A)$	$w_2(A)$
$u_1(A)$	$u_2(A)$

- Possible interleaved execution:



Increment lock

- The increment operation and the increment lock
  - The increment operation: **INC(A, c)**: performs the following steps atomically

```
READ(A, t) //t is a local variable in the tranx
t = t + c
WRITE(A, t)
```

Note: `c` can be negative

- Many applications make use of the `increment operation`
  - `banking: transfer sum`

    `A = A + c`

    `B = B + (-c)`
  - `booking`

    `#seats = #seats + (-1)`

Increment lock

- Types of operations on database elements
  - `READ(.)`
  - `WRITE(.)`
  - `INC(., c)`
- For each type of operation, we use a specific type of lock
  - **`shared lock`**
    - grants a transaction the privilege to perform a **`read operation`** on the `DB element`
  - **`exclusive lock`**
    - grants a transaction the privilege to perform a **`write operation`** (also **`read`**) on the `DB element`
  - **`increment lock`**
    - grants a transaction the privilege to perform an **`increment operation`** on the `DB element`
- $\mathtt{INC}_i(\mathtt{A}, \mathtt{c}_1)$ and $\mathtt{INC}_j(\mathtt{A}, \mathtt{c}_2)$ do not conflict

Compatibility matrix for shared/increment/exclusive locking

	Lock request type		
State of the lock	*Shared*	*Exclusive*	*Increment*
Shared	Yes	No	No
Exclusive	No	No	No
Increment	No	No	Yes