# Transaction Processing

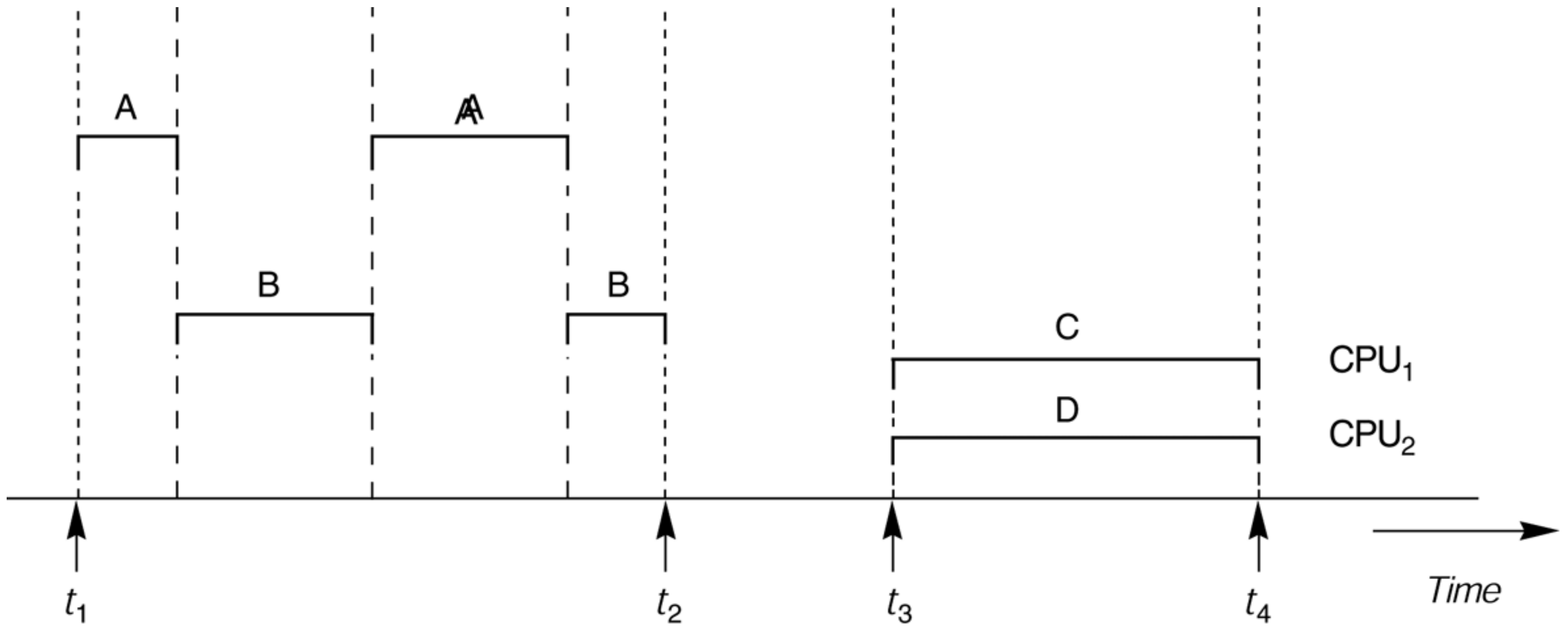
## Overview

- Transaction Processing – An Introduction
- Why concurrency control?
- Why Recovery is Needed?
- Transaction States
- System Log
- Commit Point of a Transaction
- ACID Properties
- Schedules
- Serializability of Schedules
  - Result equivalent
  - Conflict equivalent
  - View equivalent

## Introduction

- **Single-User System:** At most one user at a time can use the system.
- **Multi-user System**: Many users can access the system concurrently.
- **Concurrency**
    - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
    - **Parallel processing**: processes are concurrently executed in multiple CPUs.

# Introduction



*interleaved processing versus parallel processing of concurrent transactions*

## Introduction

- **A Transaction:** logical unit of database processing that includes one or more access operations (read - retrieval, write - insert or update, delete).
- **A transaction (set of operations)** may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries**: Begin and End transaction.
- Basic operations are read and write:
  - **read_item(X)**: Reads a database item named X into a program variable X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

## Introduction

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database
- **read_item(X) command includes the following steps:**
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

## Introduction

- **write_item(X) command includes the following steps:**
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

## Introduction

| (a) $T_1$ | (b) $T_2$ |
|---|---|
| read_item ($X$); | read_item ($X$); |
| $X$:=$X$-$N$; | $X$:=$X$+$M$; |
| write_item ($X$); | write_item ($X$); |
| read_item ($Y$); | |
| $Y$:=$Y$+$N$; | |
| write_item ($Y$); | |

*two sample transactions $T_1$ and $T_2$*

## Why Concurrency Control ?

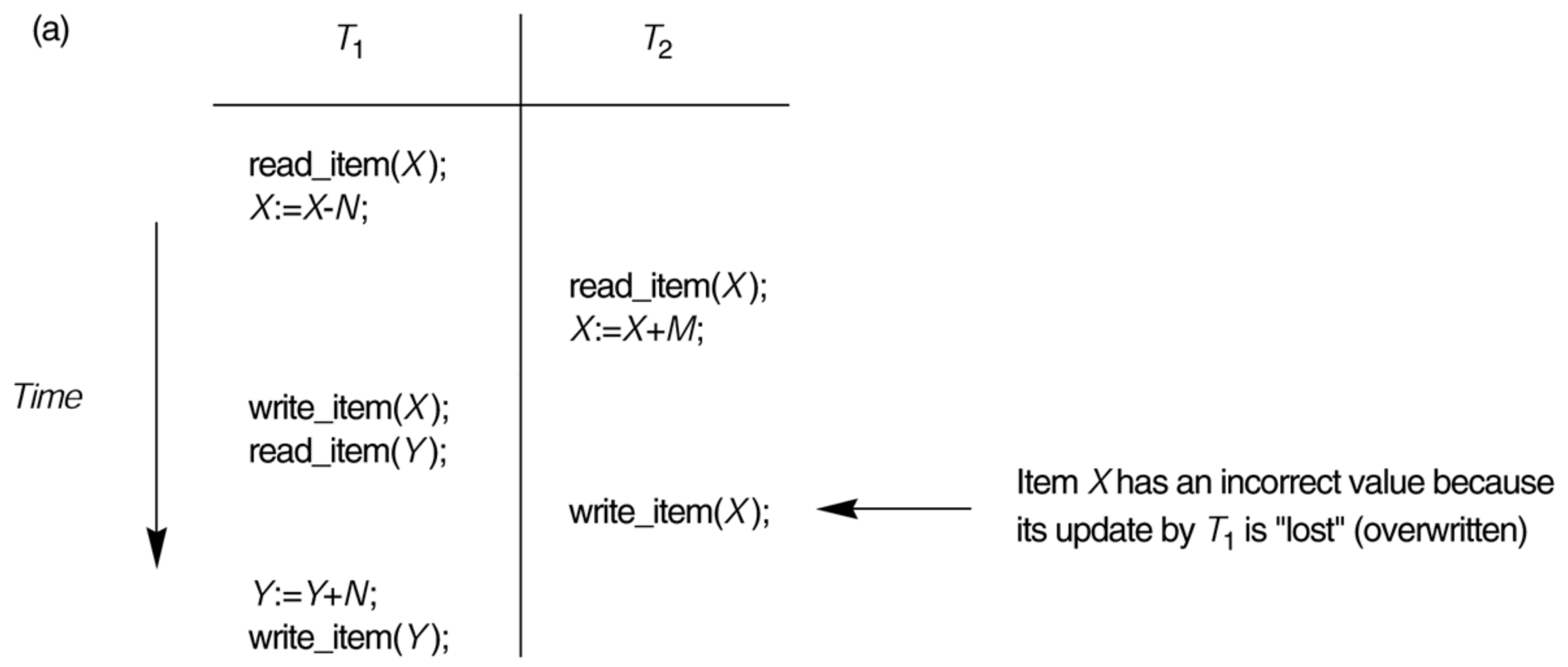
- **The Lost Update Problem.**

  This occurs when two transactions that *access the same* database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **The Temporary Update (or Dirty Read) Problem.**

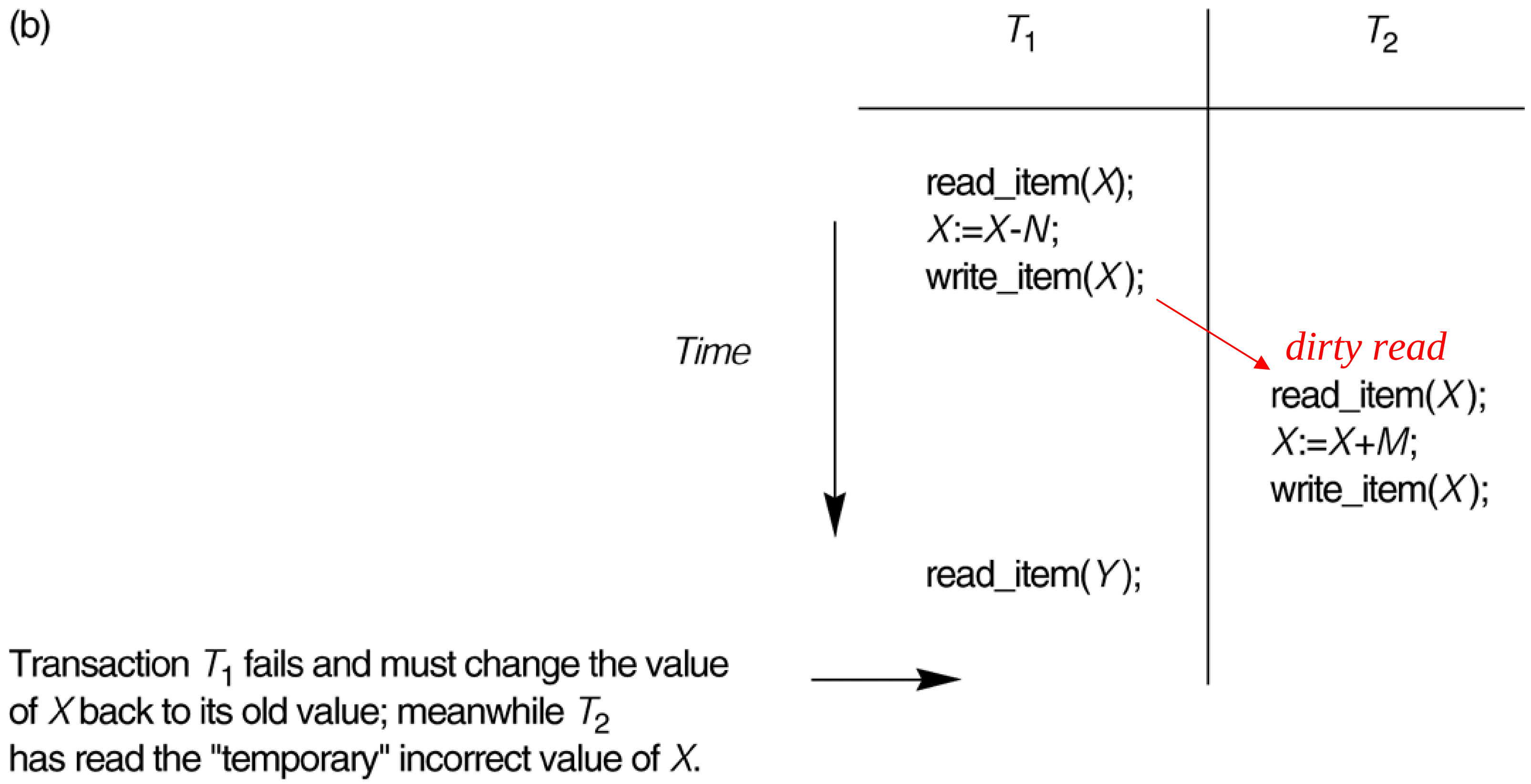
  This occurs when one transaction updates a database item and then the *transaction fails* for some reason. The updated item is accessed by another transaction before it is changed back to its original value.

## Why Concurrency Control ?



Lost Update: $T_2$ reads the value of X *before* $T_1$ changes it in the database

# Why Concurrency Control ?


(b)
$T_1$
$T_2$
read_item($X$);
$X$:=$X$-$N$;
write_item($X$);
Time
dirty read
read_item($X$);
$X$:=$X$+$M$;
write_item($X$);
read_item($Y$);
Transaction $T_1$ fails and must change the value of $X$ back to its old value; meanwhile $T_2$ has read the "temporary" incorrect value of $X$.

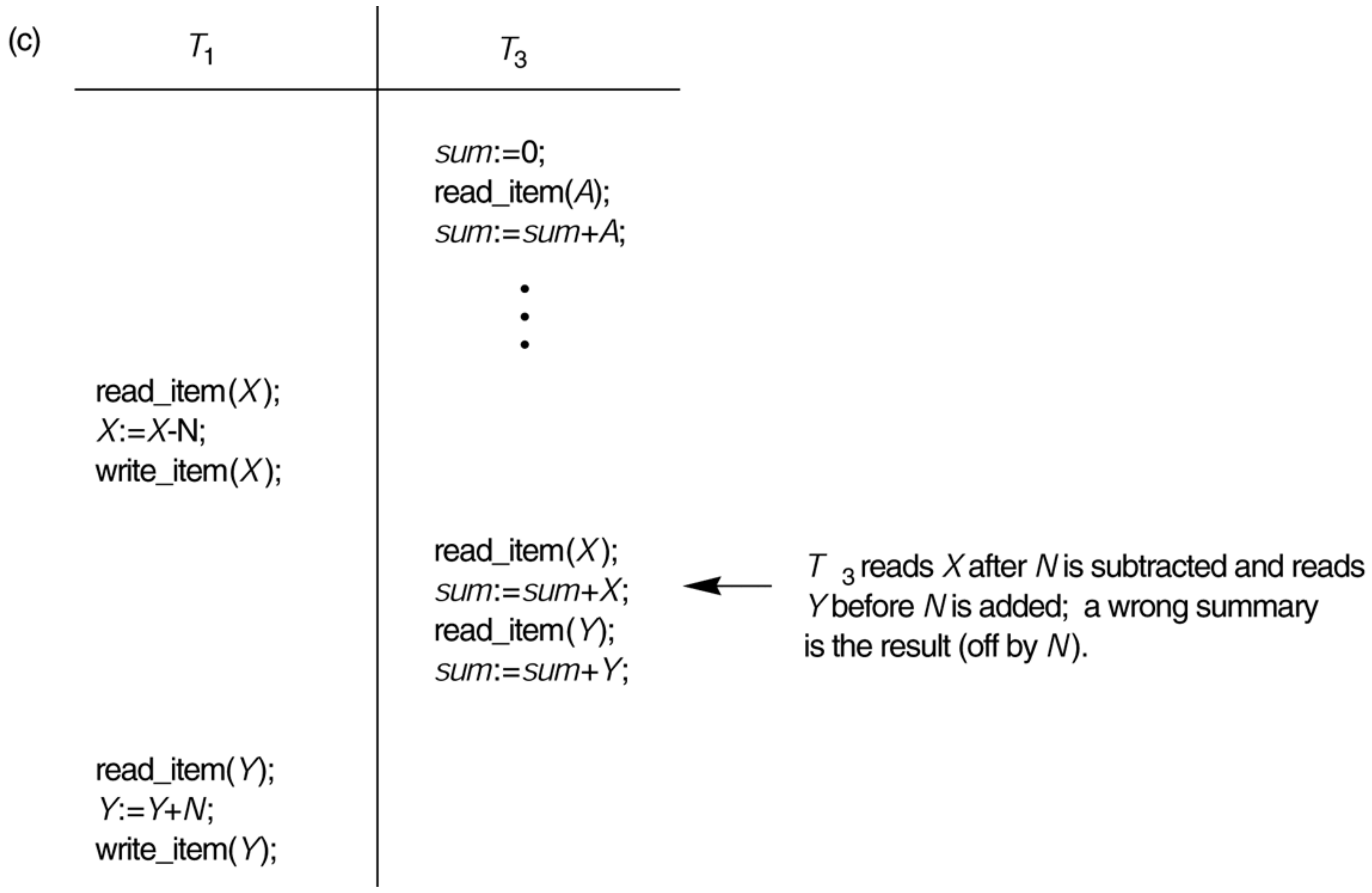
Temporary Update: $T_2$ reads the *temporary* value of X *before* $T_1$ commits

## Why Concurrency Control ?

- **The Incorrect Summary Problem .**

  If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

# Why Concurrency Control ?



(c)

| $T_1$ | $T_3$ |
|---|---|
| | *sum*:=0;<br>read_item(*A*);<br>*sum*:=*sum*+*A*;<br>⋮ |
| read_item(*X*);<br>*X*:=*X*-N;<br>write_item(*X*); | |
| | read_item(*X*);<br>*sum*:=*sum*+*X*;<br>read_item(*Y*);<br>*sum*:=*sum*+*Y*; |
| read_item(*Y*);<br>*Y*:=*Y*+*N*;<br>write_item(*Y*); | |

$\longleftarrow$ $T_3$ reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

## Why Recovery is Needed ?

- What causes a Transaction to fail ?
- **System failures may occur**
- Types of failures:
    - System crash
    - Transaction or system error
    - Local errors or exception conditions
    - Concurrency control enforcement
    - Disk failure
    - Physical failures
- DBMS has a *Recovery* Subsystem to protect database against system failures

## Transaction States

- A **transaction** is an atomic unit of work that is *either completed in its entirety or not done at all*.
- For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

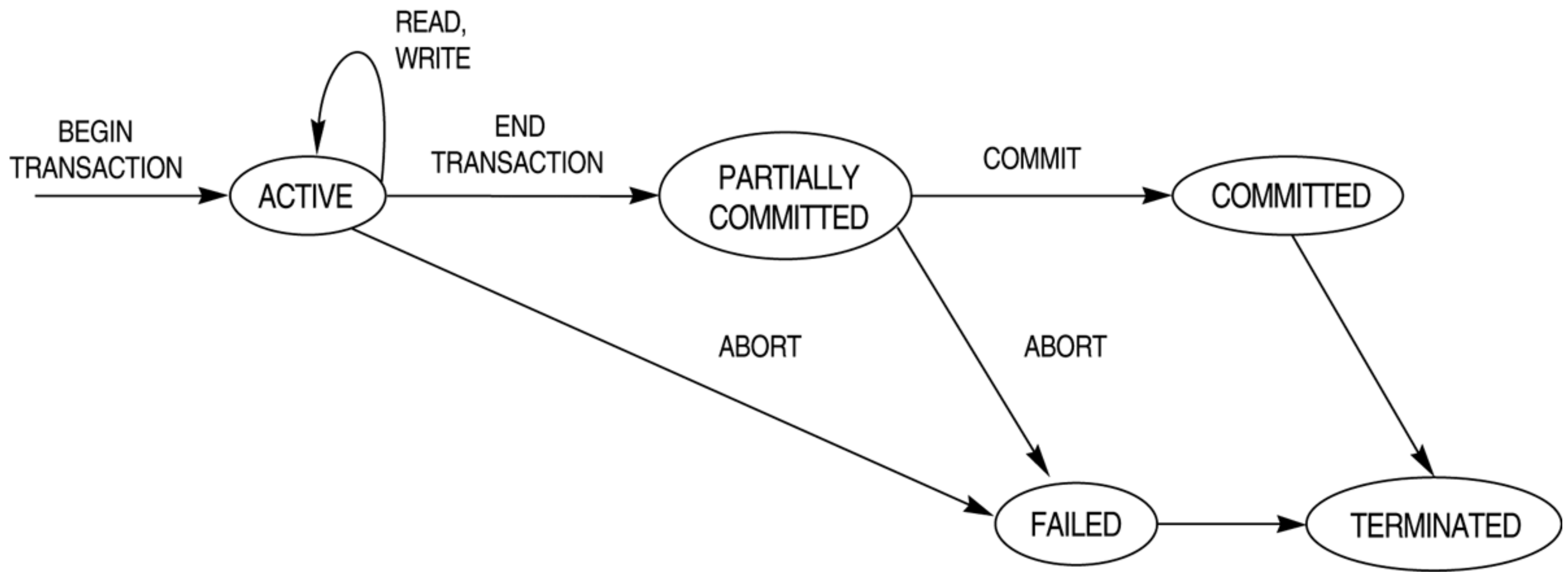
## Transaction States

- Recovery manager keeps track of the following operations:
- `BEGIN_TRANSACTION`: This marks the beginning of transaction execution.
- `READ` or `WRITE`: These specify read or write operations on the database items that are executed as part of a transaction.
- `END_TRANSACTION`: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
  - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (or) whether the transaction has to be aborted because it violates concurrency control or for some other reason.

## Transaction States

- Recovery manager keeps track of the following operations (cont):
- `COMMIT_TRANSACTION`**:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- `ROLLBACK` **(or** `ABORT`**):** This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone.*

# Transaction States



State transition diagram: states for transaction execution

## System Log

- To recover from transaction failures, the system maintains a **log.**
- The log keeps track of all transaction operations that affect the values of database items.
- The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure

## System Log

- **Types of log record:**
  - [start_transaction,T]:

    Records that transaction T has started execution.
  - [write_item,T,X,old_value,new_value]:

    Records that transaction T has changed the value of database item X from old_value to new_value.
  - [read_item,T,X]:

    Records that transaction T  has read the value of database item X.
  - [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed to the database
  - [abort,T]: Records that transaction T has been aborted.

## System Log

- **Recovery using log records:**

  If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques:

  - Need to either *redo* or *undo* everything that happened since *last commit point*
  - UNDO by *resetting all items* changed by a write operation of T to their old_values
  - REDO by *setting all items* changed by a write operation of T to their new_values

# Commit Point of a Transaction

- **Definition:** A transaction T reaches its **commit point** when all its operations have been executed successfully *and* the effect of all the transaction operations has been recorded in the log.
    - Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database.
    - The transaction then writes an entry [`commit,T`] into the log.
- **Roll Back of transactions:** Needed for transactions that have a [`start_transaction,T`] entry into the log but *no commit entry* [`commit,T`] into the log.

## Commit Point of a Transaction

- **Redoing transactions:** Transactions that have written their commit entry in the log must also have recorded all their write operations in the log, so their effect on the database can be *redone* from the log entries.
- Notice that the log file must be kept on disk. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost.
- **Force writing a log:** *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

## ACID Properties

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and eliminates cascading rollbacks.
  - Level 0: no dirty reads
  - Level 1: no lost updates
  - Level 2: no lost updates and no dirty reads
  - Level 3: Level 2 + repeatable reads

## ACID Properties

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
- Atomicity – responsibility of the transaction *recovery subsystem*
- Consistency – responsibility of the programmer or the DBMS module that enforces integrity constraints
- Isolation – enforced by *concurrency control subsystem* of the DBMS
- Durability – responsibility of the *recovery subsystem* of the DBMS
- Recovery protocols enforces *atomicity* and *durability*

## Schedules

- Schedules
  - Conflict Operations
- Characterizing schedules
  - Recoverable and non-recoverable schedules
  - Cascading rollback and cascadeless
  - Strict schedule
- Serializability of Schedules
  - Serial and non-serial schedules
  - Conflict serializability
  - View serializability

## Schedules

- A **schedule ( or history)** S of n transactions T1, T2, ...., Tn is an ordering of the operations of the transactions subject to the constraint that:
  - for each transaction Ti that participates in S, the operation of Ti in S must appear in the same order in which they occur in Ti.
  - Note, however, that operations from other transactions Tj <u>can be interleaved</u> with the operations of Ti in S.

## Schedules

- Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;
- Two operations in a schedule are said to **conflict:**
  - if they belong to different transactions
  - if they access the same item X
  - if one of the two operations is a write_item(X)

Hence,

r1(X) and w2(X) – conflict

r2(X) and w1(X) – conflict

w1(X) and w2(X) – conflict

r1(X) and r2(X) – DO NOT conflict

# Schedules

- Schedules
  - Conflict Operations
- Characterizing schedules
  - Recoverable and non-recoverable schedules
  - Cascadeless schedules and Cascading rollback schedules
  - Strict schedule
- Serializability of Schedules
  - Serial and non-serial schedules
  - Conflict serializability
  - View serializability

## Characterizing Schedules

- **Recoverable Schedule**:
- Once a transaction T is committed, it should *never be necessary* to roll back T. The schedules that meet this criteria are called **recoverable schedules.**
- A schedule S is said to be **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

# Characterizing Schedules

Consider two schedules:

Sc: r1 (X); w1 (X); r2 (X); r1(Y); w2 (X); c2; a1;

Sd: r1 (X); w1 (X); r2(X); r1(Y); w2 (X); w1(Y); c1; c2;

a) Sd is recoverable

b) Sc is not recoverable, because T2 reads item X from T1, and then T2 commits before T1 commits. If T1 aborts after the C2; then the value of X that T2 read is no longer valid and T2 must be aborted *after it had already committed ---> schedule not recoverable.*

# Characterizing Schedules

- **Cascadeless Schedule**:
- An uncommitted transaction has to be rolled back because it read an item from a transaction that failed.
- This phenomenon is known as cascading rollback

Consider the schedule:

Se: r1(X); w1(X); r2(X) ; r1(Y); w2(X); w1(Y); a1; a2;

Transaction T2 has to be *rolled back* because it read an item X from T1, and T1 then aborted.

## Characterizing Schedules

- **Cascadeless Schedule**:
- A schedule is said to be **cascadeless**, if every transaction in the schedule only reads an items that were written by the **committed** transactions

Consider the schedule:

Se: r1(X); w1(X); r2(X) ; r1(Y); w2(X); w1(Y); a1; a2;

To avoid rollback:

then r2(X) must be postponed until after T1 has committed (aborted), thus delaying T2.

## Characterizing Schedules

- **Strict Schedule**:
- Transactions can neither read *nor write* an item X until the last transaction that wrote X has committed (or aborted)
- Strict schedules simplify the recovery process

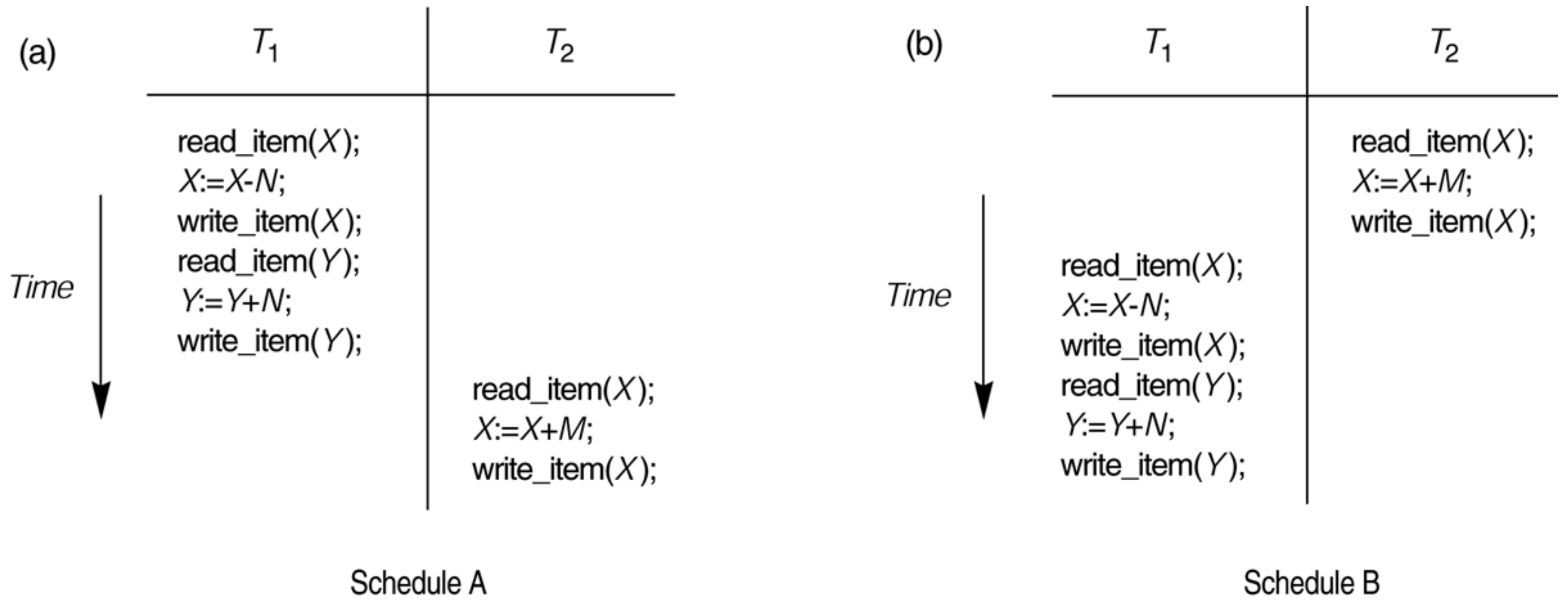
# Serializability of Schedules

- Schedules
  - Conflict Operations
- Characterizing schedules
  - Recoverable and non-recoverable schedules
  - Cascadeless schedules and Cascading rollback schedules
  - Strict schedule
- Serializability of Schedules
  - Serial and non-serial schedules
  - Conflict serializability
  - View serializability

## Serializability of Schedules

- Consider two transactions T1 and T2 which is submitted at the same time. If no interleaving is permitted then there are two possible ways:
  - Execute all the operations of T1 and then T2
  - Execute all the operations of T2 and then T1
- Serializability theory, attempts to determine which schedules are "*correct*" and which are not and to develop techniques that allow only correct schedules.

# Serial Schedules



(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$); | |
| $X$:=$X$-$N$; | |
| write_item($X$); | |
| read_item($Y$); | |
| $Y$:=$Y$+$N$; | |
| write_item($Y$); | |
| | read_item($X$); |
| | $X$:=$X$+$M$; |
| | write_item($X$); |

Schedule A

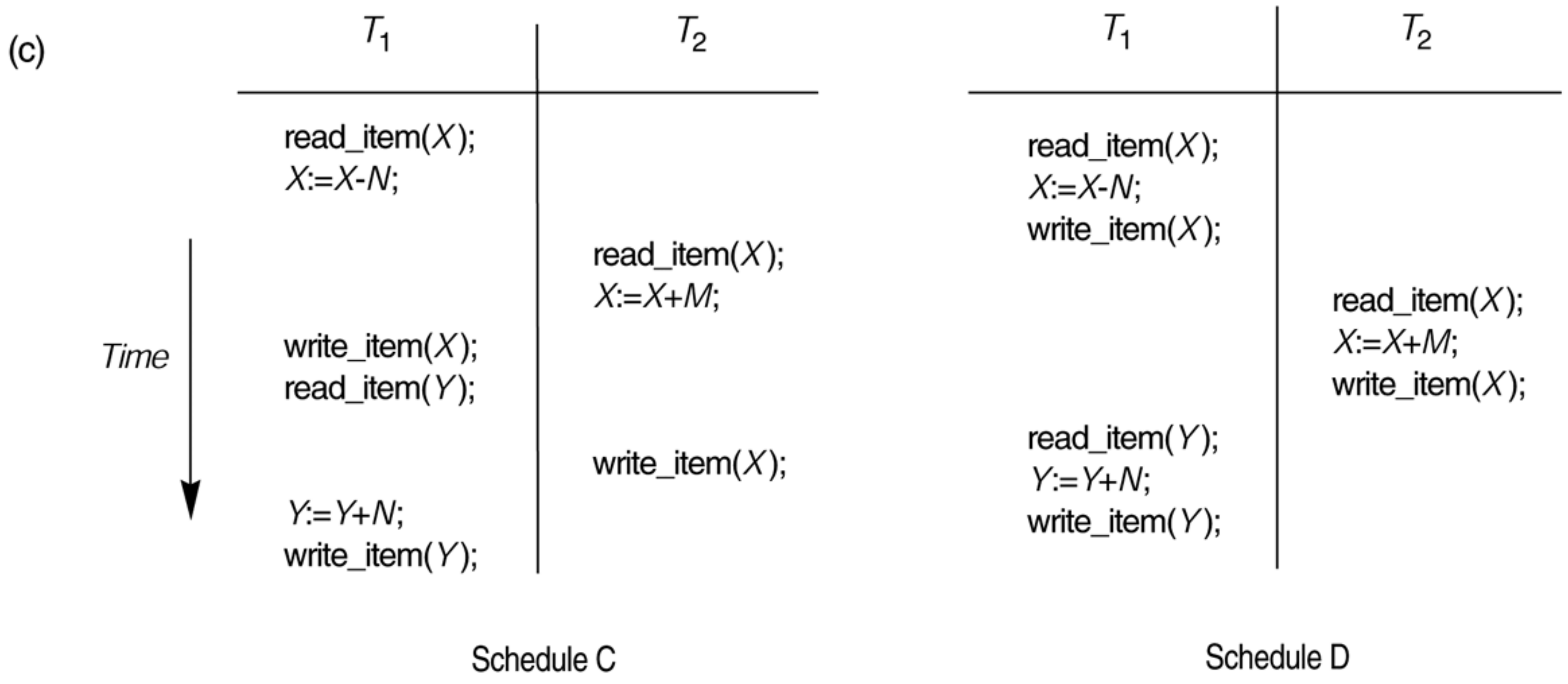
(b)

| $T_1$ | $T_2$ |
|---|---|
| | read_item($X$); |
| | $X$:=$X$+$M$; |
| | write_item($X$); |
| read_item($X$); | |
| $X$:=$X$-$N$; | |
| write_item($X$); | |
| read_item($Y$); | |
| $Y$:=$Y$+$N$; | |
| write_item($Y$); | |

Schedule B

*(a) Serial schedule A: $T_1$ followed by $T_2$. (b) Serial schedule B: $T_2$ followed by $T_1$.*

# Non-serial Schedules



(c)

Time

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$); | |
| $X$:=$X$-$N$; | |
| | read_item($X$); |
| | $X$:=$X$+$M$; |
| write_item($X$); | |
| read_item($Y$); | |
| | write_item($X$); |
| $Y$:=$Y$+$N$; | |
| write_item($Y$); | |

Schedule C

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$); | |
| $X$:=$X$-$N$; | |
| write_item($X$); | |
| | read_item($X$); |
| | $X$:=$X$+$M$; |
| | write_item($X$); |
| read_item($Y$); | |
| $Y$:=$Y$+$N$; | |
| write_item($Y$); | |

Schedule D

*Two nonserial schedules C and D with interleaving of operations*

## Serial / Non-serial Schedules

- Schedules A and B are called *serial schedules* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- In a serial schedule, transactions are performed in serial order.
- Schedules C and D are *nonserial* because of interleaving operations from two transactions.
- Every serial schedule is considered correct, because the transaction do not depend on one another.
- Every transaction will execute without any interference from the operation of other transactions.

## Serial / Non-serial Schedules

- Disadvantages:
  - Limit concurrency or interleaving of operations
  - If a transaction waits for an IO operation to complete, we cannot switch the CPU processor to another transaction --> wastage of CPU time

## Serializability of Schedules

- Some non-serial schedules give correct results.
- We have to determine which of the non-serial schedule *always* give a correct result.
- A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions.
- Question is : When are two schedules considered “equivalent” ?
  - Result equivalent
  - Conflict equivalent – Conflict Serializable
  - View equivalent – View Serializable

## Result Equivalent

- Two schedules are called **result equivalent** if they produce the same final state of the database.
- Disadvantages:
  - Not all the result equivalent schedules produces same final state.

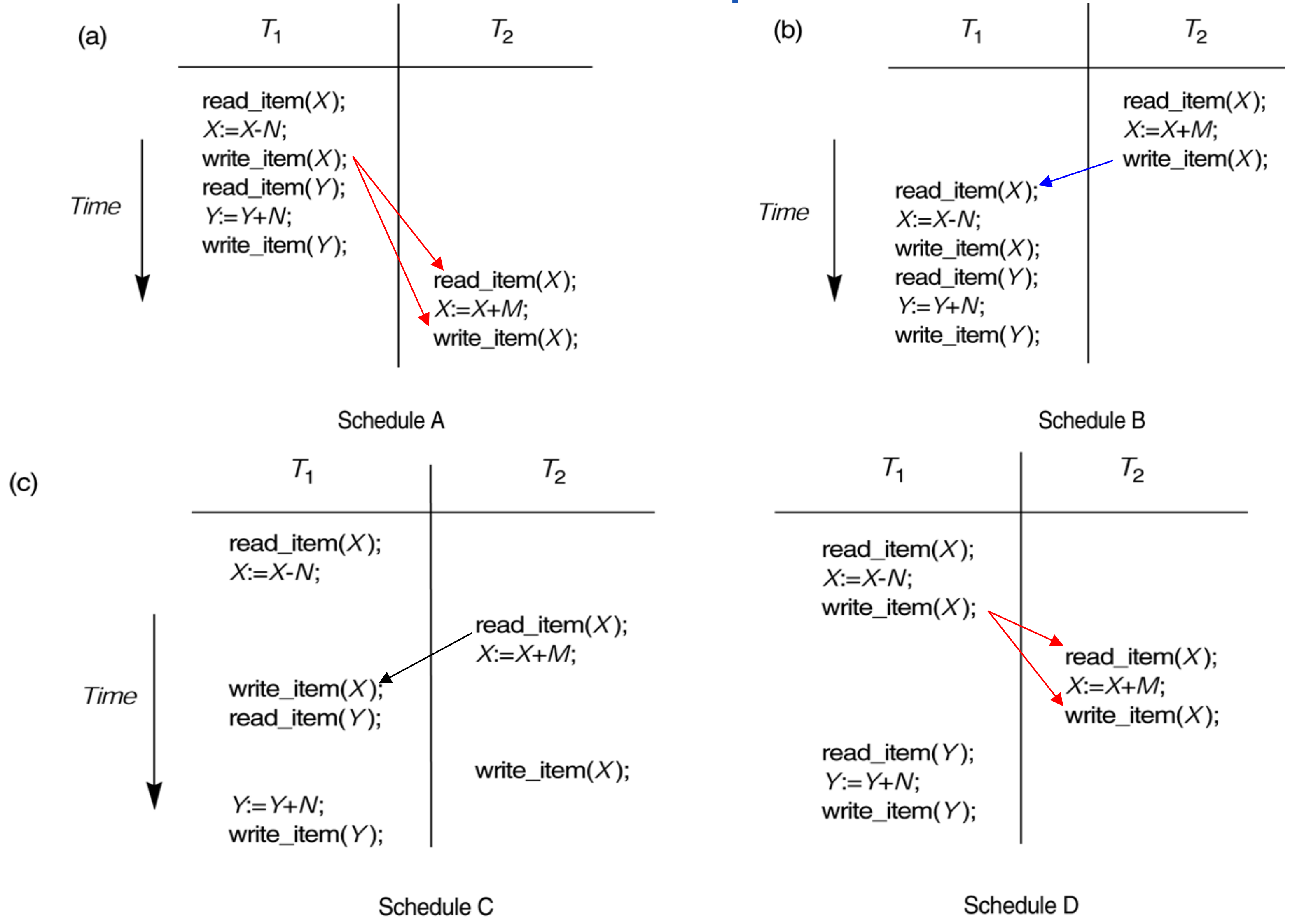
| $S_1$ | $S_2$ |
|---|---|
| read_item($X$); | read_item($X$); |
| $X$:=$X$+10; | $X$:=$X$*1.1; |
| write_item($X$); | write_item($X$); |

*Two schedules that are result equivalent for the initial value of X = 100 but are not result equivalent in general.*

## Conflict Equivalent

- Two schedules are said to be **conflict equivalent** if the order of any two *conflicting* operations is the same in both schedules.
- According to this, schedule D is *equivalent* to the serial schedule A. Since A is serial schedule and D is equivalent to A, D is **serializable schedule.**
- We say that a schedule *S* is *conflict serializable* if it is conflict equivalent to a serial schedule, hence D is *conflict serializable.*

## Conflict Equivalent

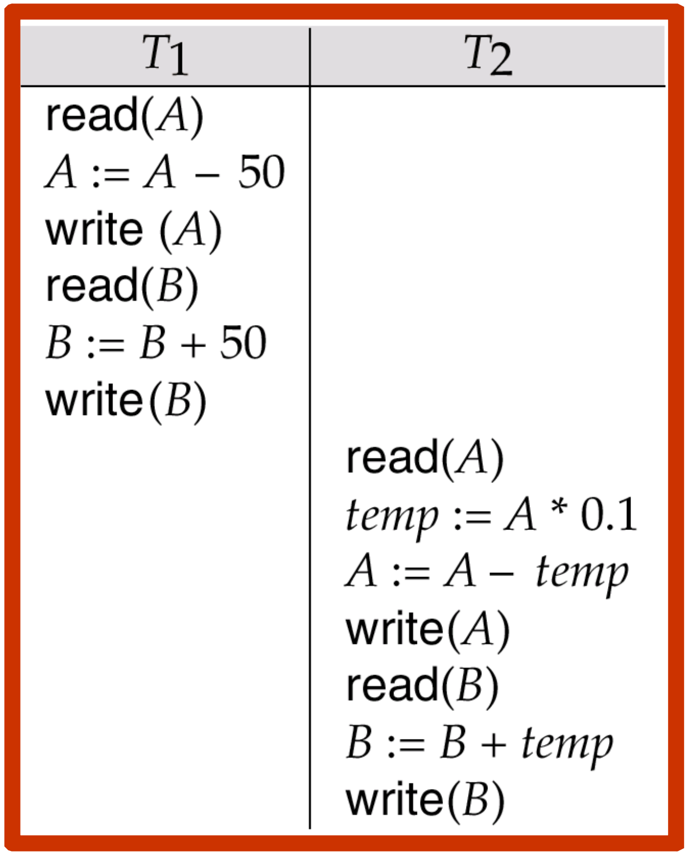


# Conflict Equivalent

- If $I_i$ and $I_j$ are consecutive in a schedule and they *do not conflict*, their results would remain the same if they are interchanged in the schedule.
- If a schedule *S* can be transformed into an equivalent schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are ***conflict equivalent.***
  - schedule that is not conflict serializable:

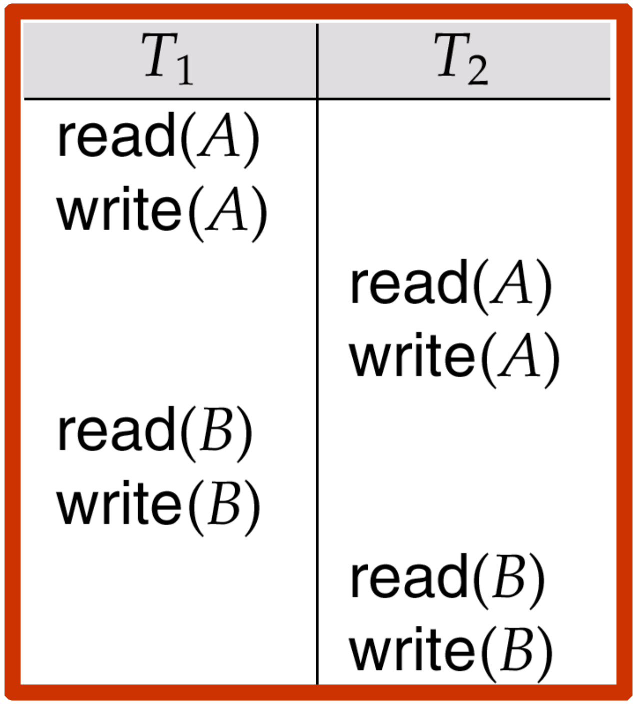
| $T_3$ | $T_4$ |
|---|---|
| **read**($Q$) | |
| | **write**($Q$) |
| **write**($Q$) | |

# Conflict Equivalent



| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write ($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

$S_1$



| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

$S_1$

$S_2$

## Conflict Equivalent

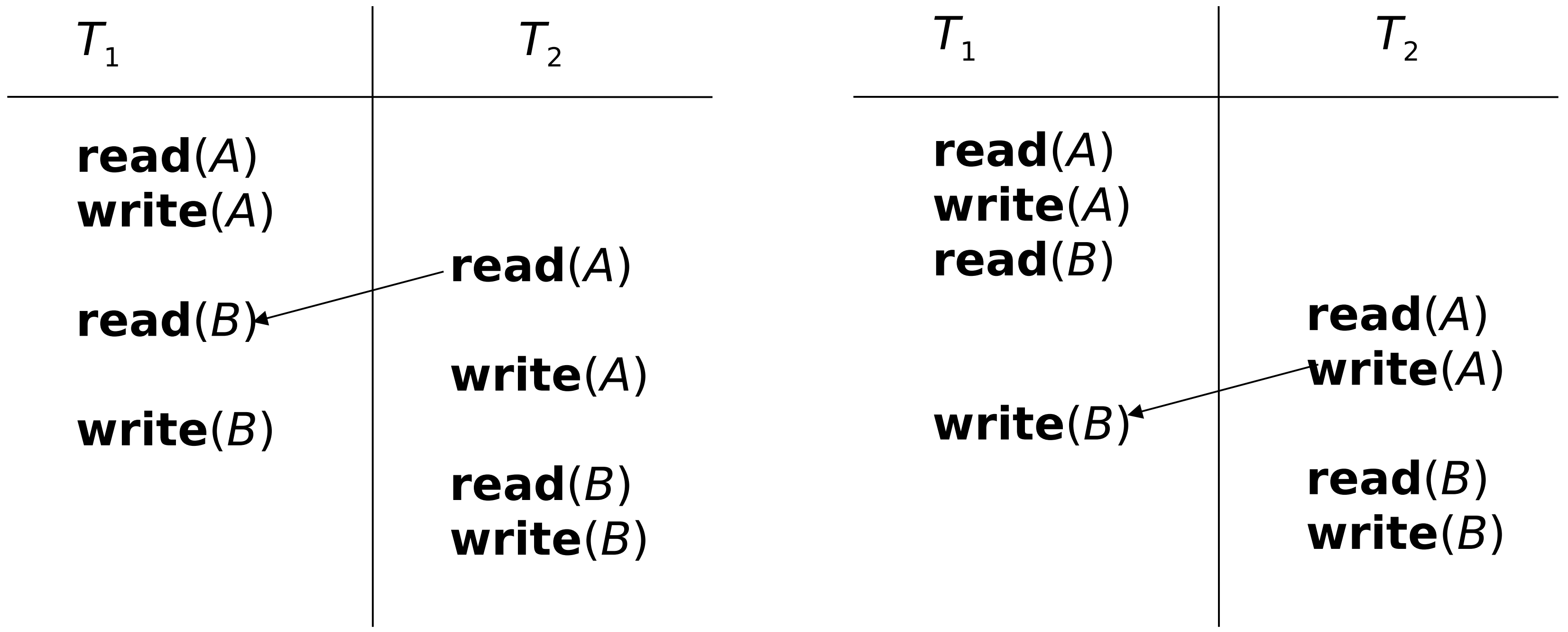
| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| | **read**($A$) |
| | **write**($A$) |
| **read**($B$) | |
| **write**($B$) | |
| | **read**($B$) |
| | **write**($B$) |

Swap non-conflict operations

| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| | **read**($A$) |
| **read**($B$) | |
| | **write**($A$) |
| **write**($B$) | |
| | **read**($B$) |
| | **write**($B$) |

After swapping

# Conflict Equivalent



| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| | **read**($A$) |
| **read**($B$) | |
| | **write**($A$) |
| **write**($B$) | |
| | **read**($B$) |
| | **write**($B$) |

| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| **read**($B$) | |
| | **read**($A$) |
| | **write**($A$) |
| **write**($B$) | |
| | **read**($B$) |
| | **write**($B$) |

Swap non-conflict operations

## Conflict Equivalent

| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| **read**($B$) | |
| | **read**($A$) |
| **write**($B$) | |
| | **write**($A$) |
| | **read**($B$) |
| | **write**($B$) |

| $T_1$ | $T_2$ |
|---|---|
| **read**($A$) | |
| **write**($A$) | |
| **read**($B$) | |
| **write**($B$) | |
| | **read**($A$) |
| | **write**($A$) |
| | **read**($B$) |
| | **write**($B$) |

*equivalent to S1*

# Testing for Conflict Serializability

- **Precedence Graph - Algorithm 17.1:**

  Looks at only read_Item (X) and write_Item (X) operations

  - Constructs a precedence graph (serialization graph) - a graph with directed edges
  - An edge is created from $T_i$ to $T_j$ if one of the operations in $T_i$ appears before a conflicting operation in $T_j$
  - The schedule is serializable if and only if the precedence graph has no cycles.

## Testing for Conflict Serializability

*Serial schedule A*

(a)

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \end{tikzcd}$$

*Serial schedule B*

(b)

$$\begin{tikzcd} T_1 & T_2 \arrow[l, bend right, "X"'] \end{tikzcd}$$

(c)

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \arrow[l, bend right, "X"'] \end{tikzcd}$$

*Schedule C (not serializable)*

(d)

$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \end{tikzcd}$$

*Schedule D (serializable, equivalent to A)*

# View Equivalent

- View Equivalence leads to another definition of serializability called ***view serializability***.
- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.
- **“The view”**: the read operations are said to see the *the same view* in both schedules.

## View Equivalent

- Two schedules are said to be **view equivalent** if the following three conditions hold:
- The same set of transactions participates in S and S', and S and S' include the same operations of these transactions.
- For any operation ri(X) of Ti in S, if the value of X read by the operation has been written by an operation wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation ri(X) of Ti in S'.
- If the operation wk(Y) of Tk is the last operation to write item Y in S, then wk(Y) of Tk must also be the last operation to write item Y in S'.

## View Equivalent

- A schedule S is said to be **view serializable** if it is view equivalent to a serial schedule.
- Constrained writes: the value written by w(X) in Ti depends only on the value of X read by r1(X) in Ti
- Blind write: the value written by w(X) in Ti is *independent* of its old value.

## View Equivalent – Example

- Consider the following serial schedule:

| T1 | T2 | T3 |
|---|---|---|
| r1(X) | | |
| w1(X) | | |
| | w2(X) -> Blind write | |
| | | w3(X) -> Blind write |

- Consider the following non-serial:

| T1 | T2 | T3 |
|---|---|---|
| r1(X) | | |
| | w2(X) | |
| w1(X) | | |
| | | w3(X) |

## References

- Fundamentals of Database Systems, Elmasri and Navathe, Pearson, 3rd Edition

THANK YOU!