

Introduction to Transaction Processing Concepts and Theory



Introduction to Transaction Processing

- **Transaction:** An executing program (process) that includes one or more database access operations
 - Read operations (database retrieval, such as SQL SELECT)
 - Write operations (modify database, such as SQL INSERT, UPDATE, DELETE)
 - Transaction: A logical unit of database processing
 - Example: Bank balance transfer of \$100 dollars from a checking account to a saving account in a BANK database
- **Note:** Each execution of a program is a *distinct transaction* with different parameters
 - Bank transfer program parameters: savings account number, checking account number, transfer amount



Introduction to Transaction Processing (cont.)

- A transaction (set of operations) may be:
 - stand-alone, specified in a high level language like SQL submitted interactively, or
 - consist of database operations embedded within a program (most transactions)
- **Transaction boundaries:** Begin and End transaction.
 - Note: An **application program** may contain several transactions separated by Begin and End transaction boundaries



Introduction to Transaction Processing (cont.)

- **Transaction Processing Systems:** Large multi-user database systems supporting thousands of *concurrent transactions* (user processes) per minute
- **Two Modes of Concurrency**
 - **Interleaved processing:** concurrent execution of processes is interleaved in a single CPU
 - **Parallel processing:** processes are concurrently executed in multiple CPUs (Figure 21.1)
 - Basic transaction processing theory assumes interleaved concurrency



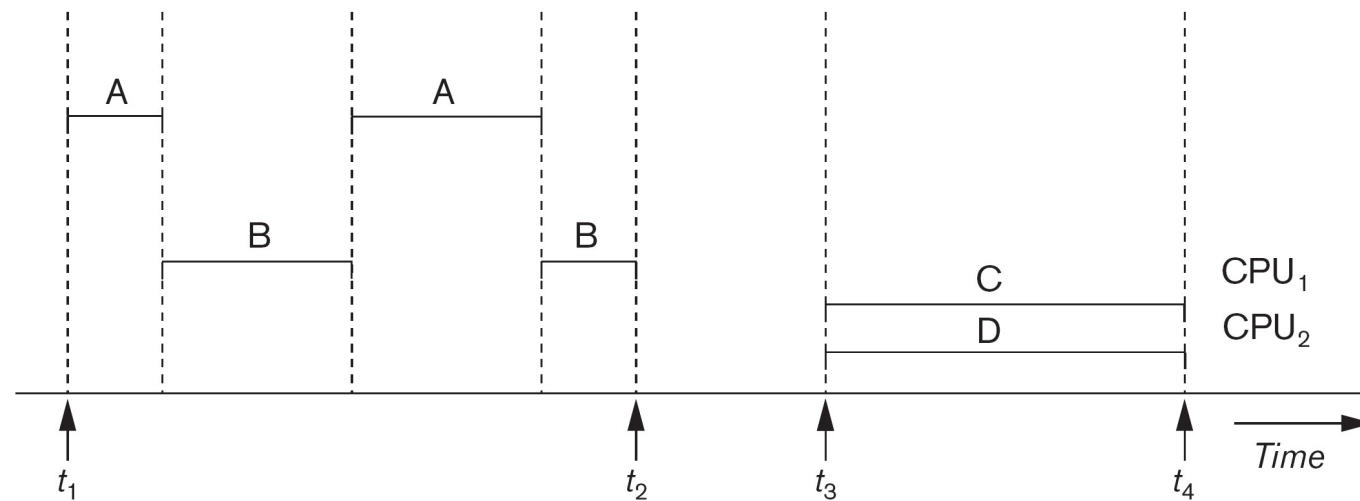


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

Introduction to Transaction Processing (cont.)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one disk block (or page). A data item X (what is read or written) will usually be the field of some record in the database, although it may be a larger unit such as a whole record or even a whole block.
- **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 to Transaction Processing (cont.)

READ AND WRITE OPERATIONS (cont.):

- **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 it 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).



Transaction Notation

- Figure 21.2 (next slide) shows two examples of transactions
- Notation focuses on the read and write operations
- Can also write in shorthand notation:
 - T1: b1; r1(X); w1(X); r1(Y); w1(Y); e1;
 - T2: b2; r2(Y); w2(Y); e2;
- bi and ei specify transaction boundaries (begin and end)
- i specifies a unique transaction identifier (Tid)



(a)

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

(b)

T_2
read_item(X); $X := X + M$; write_item(X);

Figure 21.2

Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2 .

Why we need concurrency control

Without Concurrency Control, problems may occur with concurrent transactions:

- **Lost Update Problem.**

Occurs when two transactions update the same data item, but both read the same original value before update (Figure 21.3(a), next slide)

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

This occurs when one transaction T1 updates a database item X, which is accessed (read) by another transaction T2; then T1 fails for some reason (Figure 21.3(b)); X was (read) by T2 before its value is changed back (rolled back or UNDONE) after T1 fails



(a)

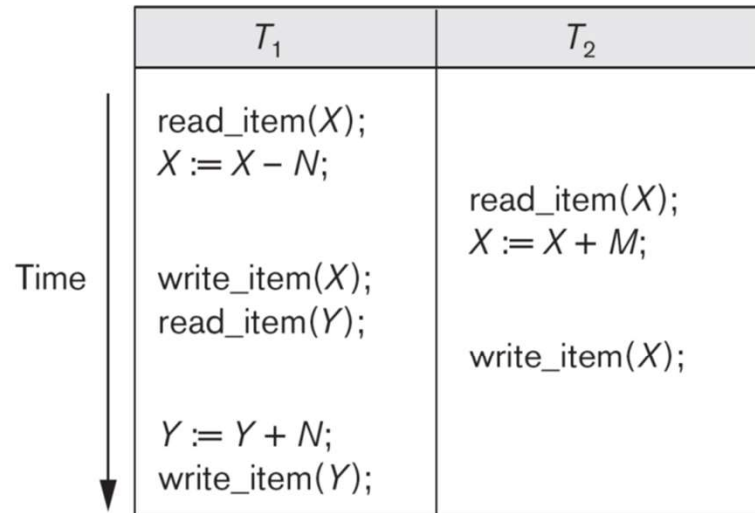
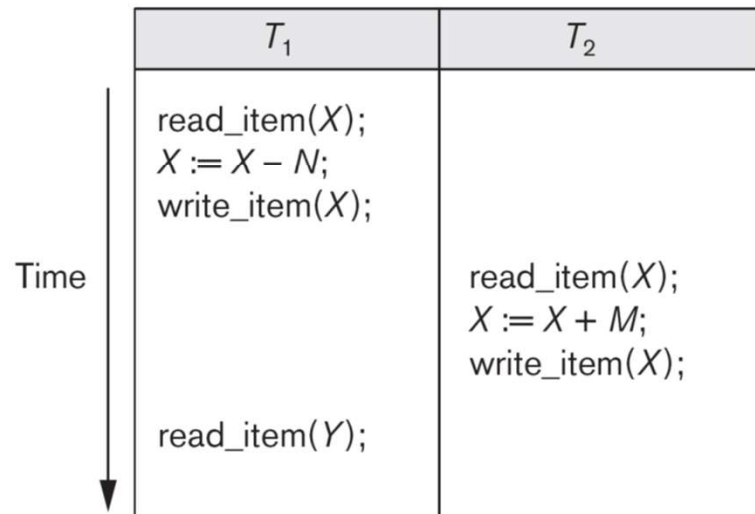


Figure 21.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

(b)



← 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 .

Why we need concurrency control (cont.)

- **The Incorrect Summary Problem .**

One transaction is calculating an aggregate summary function on a number of records (for example, sum (total) of all bank account balances) while other transactions are updating some of these records (for example, transferring a large amount between two accounts, see Figure 21.3(c)); the aggregate function may read some values before they are updated and others after they are updated.



(c)

T_1	T_3
<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>sum := 0; read_item(A); sum := sum + A; ⋮ read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre>

← 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 we need concurrency control (cont.)

- **The Unrepeatable Read Problem .**

A transaction T1 may read an item (say, available seats on a flight); later, T1 may read the same item again and get a different value because another transaction T2 has updated the item (reserved seats on the flight) between the two reads by T1

Example:

T1	T2
Read(X)	
	Read(X)
Write(X)	
	Read(X)

Why recovery is needed

Causes of transaction failure:

1. **A computer failure (system crash):** A hardware or software error occurs during transaction execution. If the hardware crashes, the contents of the computer's internal main memory may be lost.
2. **A transaction or system error :** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Why recovery is needed (cont.)

3. **Local errors or exception conditions** detected by the transaction:

- certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled
- a programmed abort causes the transaction to fail.

4. **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock (see Chapter 22).



Why recovery is needed (cont.)

5. **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This kind of failure and item 6 are more severe than items 1 through 4.
6. **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.



Transaction and System Concepts

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all. A transaction passes through several states (Figure 21.4, similar to process states in operating systems).

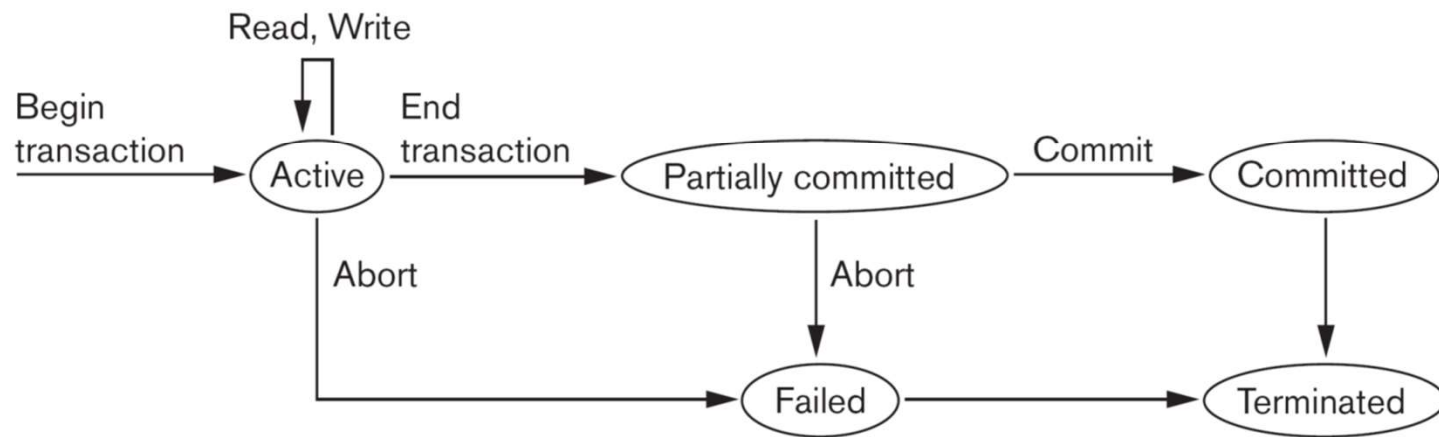
Transaction states:

- Active state (executing read, write operations)
- Partially committed state (ended but waiting for system checks to determine success or failure)
- Committed state (transaction succeeded)
- Failed state (transaction failed, must be rolled back)
- Terminated State (transaction leaves system)



Figure 21.4

State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts (cont.)

- **undo(X)**: Similar to rollback except that it applies to a single write operation rather than to a whole transaction.
- **redo(X)**: This specifies that a *write operation* of a committed transaction must be *redone* to ensure that it has been applied permanently to the database on disk.



Desirable Properties of Transactions

Called ACID properties – Atomicity, Consistency, Isolation, Durability:

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all. Enforced by the recovery protocol.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation:** Specifies that each transaction does a correct action on the database *on its own*. Application programmers and DBMS constraint enforcement are responsible for this.

Desirable Properties of Transactions (cont.)

ACID properties (cont.):

- **Isolation:** Even though transactions are executing concurrently, they should appear to be executed in isolation – that is, their final effect should be as if each transaction was executed in isolation from start to finish.
- Responsibility of the concurrency control protocol.
- **Durability or permanency:** Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure. Enforced by the recovery protocol.



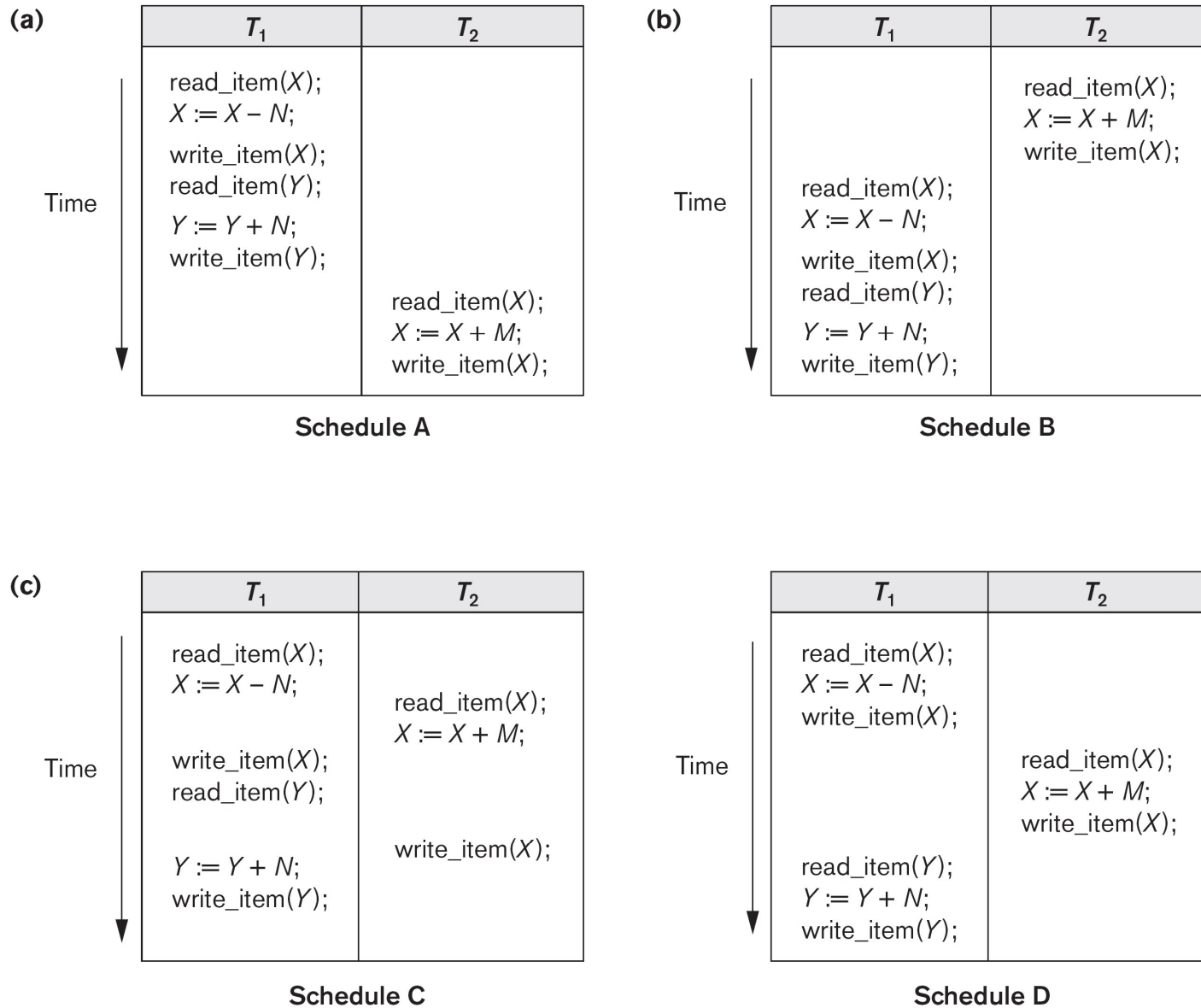
Schedules of Transactions

- **Transaction schedule (or history):** When transactions are executing concurrently in an interleaved fashion, the *order of execution* of operations from the various transactions forms what is known as a **transaction schedule** (or history).
- Figure 21.5 (next slide) shows 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
 - Order of operations from top to bottom
 - Each schedule includes *same operations*
 - Different *order of operations* in each schedule



Figure 21.5

Examples of serial and nonserial schedules involving transactions T_1 and T_2 . (a) Serial schedule A: T_1 followed by T_2 . (b) Serial schedule B: T_2 followed by T_1 . (c) Two nonserial schedules C and D with interleaving of operations.



Schedules of Transactions (cont.)

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, ...) and item name (X, Y, ...)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)
- Schedules in Figure 21.5 would be displayed as follows:
 - Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);
 - Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
 - Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);
 - Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);



Schedules of Transactions (cont.)

- Formal definition of a **schedule** (or **history**) S of n transactions T_1, T_2, \dots, T_n :
An ordering of all the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S , the operations of T_i in S must appear *in the same order* in which they occur in T_i .

Note: Operations from other transactions T_j can be interleaved with the operations of T_i in S .



Schedules of Transactions (cont.)

- For n transactions T_1, T_2, \dots, T_n , where each T_i has m_i read and write operations, the number of possible schedules is (! is *factorial* function):
$$(m_1 + m_2 + \dots + m_n)! / ((m_1)! * (m_2)! * \dots * (m_n)!)$$
- Generally very large number of possible schedules
- Some schedules are easy to recover from after a failure, while others are not
- Some schedules produce correct results, while others produce incorrect results
- Rest of chapter characterizes schedules by classifying them based on ease of recovery (**recoverability**) and correctness (**serializability**)

Characterizing Schedules based on Recoverability

Schedules classified into two main classes:

- **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.



Characterizing Schedules Based on Recoverability (cont.)

- **Example:** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)
- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
 - known as *cascading rollback*
- Schedule A: r1(X); w1(X); r2(X); w2(X); c2; r1(Y); w1(Y); c1 (or a1)
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1); ...



Characterizing Schedules based on Recoverability (cont.)

Recoverable schedules can be further refined:

- **Cascadeless schedule:** A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed.
- The set of cascadeless schedules is a *subset of* the set of recoverable schedules.

Schedules requiring cascaded rollback: A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back.



Characterizing Schedules Based on Recoverability (cont.)

- **Example:** Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits
- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the $r2(X)$ of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C
- Schedule B: $r1(X); w1(X); \underline{r2(X)}; w2(X); r1(Y); w1(Y); c1$ (or $a1$);
- Schedule C: $r1(X); w1(X); r1(Y); w1(Y); c1; \underline{r2(X)}; w2(X); \dots$



Characterizing Schedules based on Recoverability (cont.)

Cascadeless schedules can be further refined:

- **Strict schedule:** A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- The set of strict schedules is a *subset of* the set of cascadeless schedules.
- If *blind writes* are not allowed, all cascadeless schedules are also strict

Blind write: A write operation $w_2(X)$ that is not preceded by a read $r_2(X)$.



Characterizing Schedules Based on Recoverability (cont.)

- **Example:** Schedule C below is **cascadeless** and also **strict** (because it has no blind writes)
- Schedule D is cascadeless, but not strict (because of the blind write $w_3(X)$, which writes the value of X before T_1 commits)
- To make it strict, $w_3(X)$ must be delayed until after T_1 commits – see Schedule E
- Schedule C: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); \dots$
- Schedule D: $r_1(X); w_1(X); \underline{w_3(X)}; r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); \dots$
- Schedule E: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; \underline{w_3(X)}; r_2(X); w_2(X); \dots$



Characterizing Schedules based on Serializability

- Among the large set of possible schedules, we want to characterize which schedules are *guaranteed to give a correct result*
- The **consistency preservation** property of the ACID properties states that: each transaction if executed on its own (from start to finish) will transform a consistent state of the database into another consistent state
- Hence, each transaction is *correct* on its own



Characterizing Schedules based on Serializability (cont.)

- **Serial schedule:** A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively (without interleaving of operations from other transactions) in the schedule. Otherwise, the schedule is called **nonserial**.
- Based on the consistency preservation property, *any serial schedule will produce a correct result* (assuming no inter-dependencies among different transactions)



Characterizing Schedules based on Serializability (cont.)

- Serial schedules are *not feasible* for performance reasons:
 - No interleaving of operations
 - Long transactions force other transactions to wait
 - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
 - Need to allow concurrency with interleaving without sacrificing correctness



Characterizing Schedules based on Serializability (cont.)

- **Serializable schedule:** A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.
- There are $(n)!$ serial schedules for n transactions – a serializable schedule can be equivalent to *any of the serial schedules*
- **Question:** How do we define equivalence of schedules?



Equivalence of Schedules

- **Result equivalent:** Two schedules are called result equivalent if they produce the same final state of the database.
- Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general.
- May also get result equivalence *by chance* for a particular input parameter even though schedules *are not equivalent in general* (see Figure 21.6, next slide)



Figure 21.6

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

S_1
read_item(X); $X := X + 10$; write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Equivalence of Schedules (cont.)

- **Conflict equivalent:** Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.
- Commonly used definition of schedule equivalence
- Two operations are **conflicting** if:
 - They access the same data item X
 - They are from two different transactions
 - At least one is a write operation
- Read-Write conflict example: $r_1(X)$ and $w_2(X)$
- Write-write conflict example: $w_1(Y)$ and $w_2(Y)$



Equivalence of Schedules (cont.)

- Changing the order of conflicting operations generally *causes a different outcome*
- **Example:** changing $r1(X); w2(X)$ to $w2(X); r1(X)$ means that T1 will read *a different value for X*
- **Example:** changing $w1(Y); w2(Y)$ to $w2(Y); w1(Y)$ means that the final value for Y in the database can be different
- Note that read operations are **not conflicting**; changing $r1(Z); r2(Z)$ to $r2(Z); r1(Z)$ does not change the outcome

Characterizing Schedules Based on Serializability (cont.)

- **Conflict equivalence** of schedules is used to determine which schedules are correct in general (serializable)

A schedule S is said to be **serializable** if it is conflict equivalent to some serial schedule S' .



Characterizing Schedules based on Serializability (cont.)

- A serializable schedule is considered to be correct because it is equivalent to a serial schedule, and any serial schedule is considered to be correct
 - It will leave the database in a consistent state.
 - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution and interleaving of operations from different transactions.



Thank
You

