Lecture 9: Transaction Processing Concepts and Theory

CS3402 Database Systems

Single-User versus Multiuser Systems (1/2)

- A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.
- Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of users and travel agents concurrently.
- Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

Single-User versus Multiuser Systems (2/2)

- Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time.
- A single central processing unit (CPU) can only execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**.
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

Transactions

- A **transaction** is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.
- One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

Read and Write Operations (1/2)

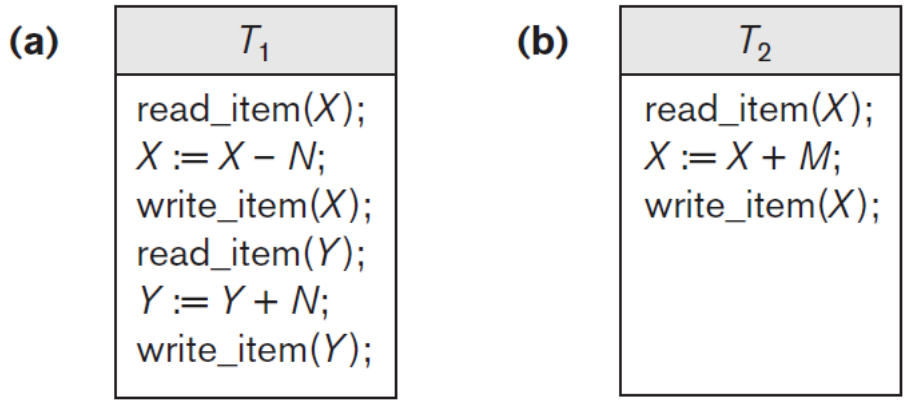
- The basic database access operations that a transaction can include are as follows:
  - **read_item(X).** Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
  - **write_item(X).** Writes the value of program variable X into the database item named X.
- Executing a read_item(X) command includes the following steps:
  1. Find the address of the disk block that contains item X.
  2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
  3. Copy item X from the buffer to the program variable named X.

Read and Write Operations (2/2)

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

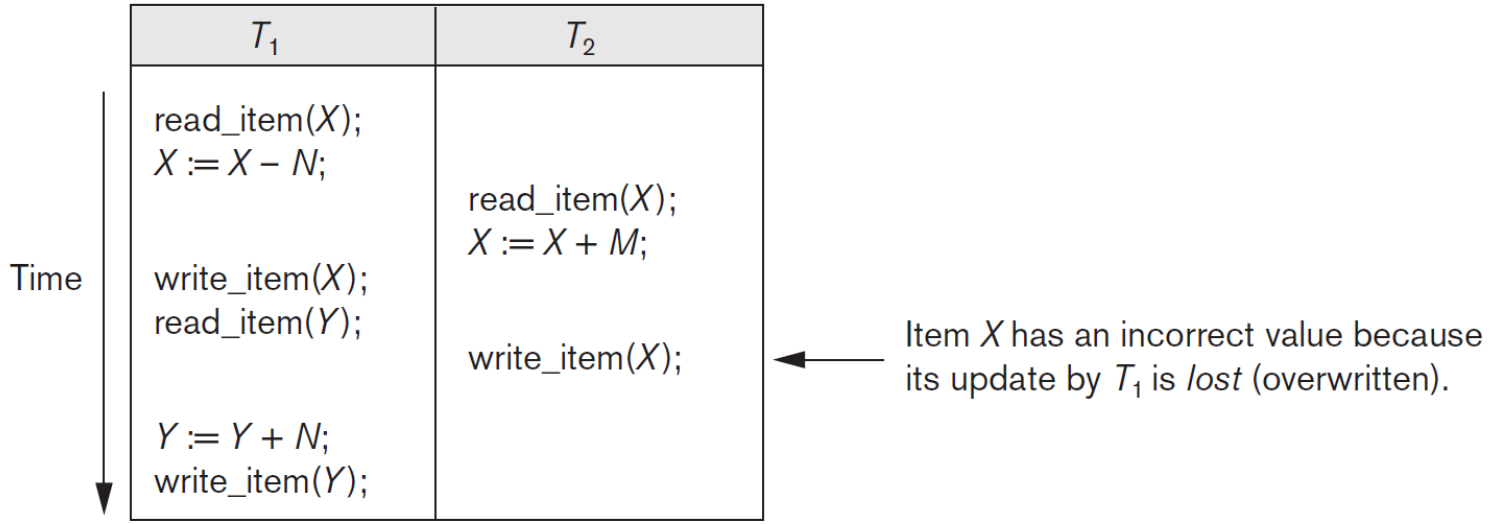
Why Concurrency Control Is Needed

- Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight.
- Each record includes the number of reserved seats on that flight as a named (uniquely identifiable) data item, among other information. Figure (a) shows a transaction T_1 that transfers N reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y.
- Figure (b) shows a simpler transaction T_2 that just reserves M seats on the first flight (X) referenced in transaction T_1.



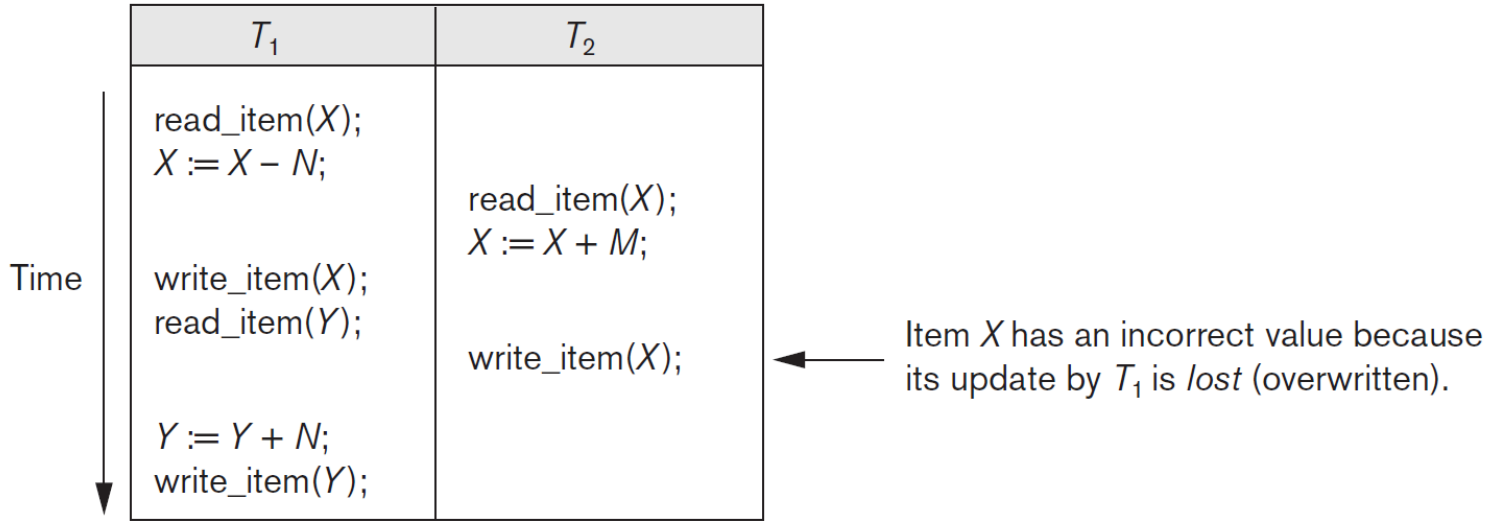
Why Concurrency Control Is Needed: The Lost Update Problem (1/2)

- This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions
- T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved; then the final value of item X is incorrect because T_2 reads the value of X before T_1 changes it in the database, and hence the updated value resulting from T_1 is lost.



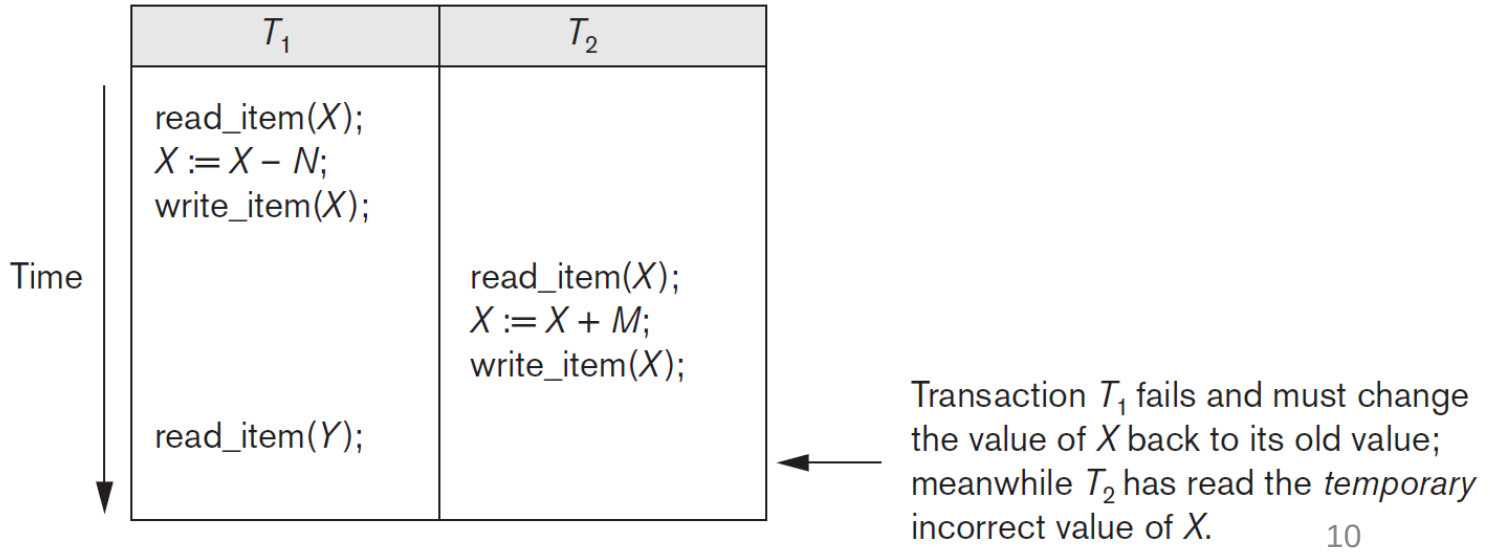
Why Concurrency Control Is Needed: The Lost Update Problem (2/2)

- For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T_2 reserves 4 seats on X), the final result should be X = 79. However, in the interleaving of operations, it is X = 84 because the update in T_1 that removed the five seats from X was lost.



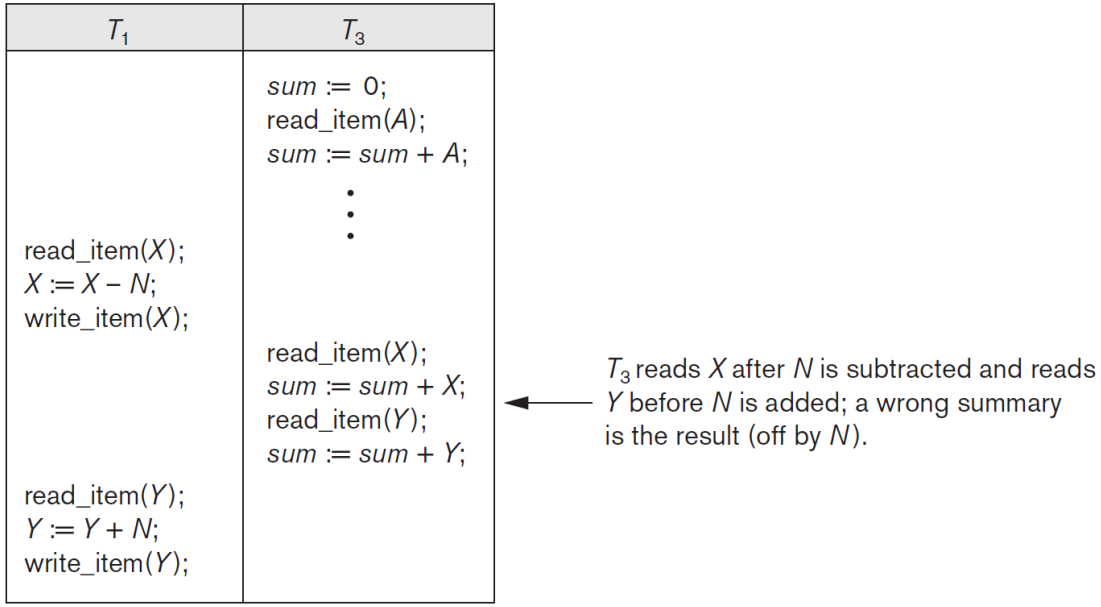
Why Concurrency Control Is Needed: The Dirty Read Problem

- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.
- T_1 updates item X and then fails before completion, so the system must roll back X to its original value. Before it can do so, however, transaction T_2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T_1.
- The value of item X that is read by T_2 is called dirty data because it has been created by a transaction that has not completed and committed yet.



Why Concurrency Control Is Needed: The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated.
- For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations occurs, the result of T_3 will be off by an amount N because T_3 reads the value of X after N seats have been subtracted from it but reads the value of Y before those N seats have been added to it.



Why Concurrency Control Is Needed: The Unrepeatable Read Problem

- Another problem that may occur is called unrepeatable read, where a transaction T reads the same item twice and the item is changed by another transaction T′ between the two reads. Hence, T receives different values for its two reads of the same item.
- This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions.
- In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**.
- The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because the whole transaction is a logical unit of database processing.
- If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Types of Failures

- Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:
  - **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution.
  - **A transaction or system error.** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero.
  - **Local errors or exception conditions detected by the transaction. For example for a**n exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled.
  - **Concurrency control enforcement.** The concurrency control method may abort a transaction because it violates serializability.
  - **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.

Desirable Properties of Transactions (1/2)

- Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:
  - **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
  - **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

Desirable Properties of Transactions (2/2)

- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
- **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Transaction States (1/2)

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, the recovery manager of the DBMS needs to keep 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 of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

Transaction States (2/2)

- 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**.

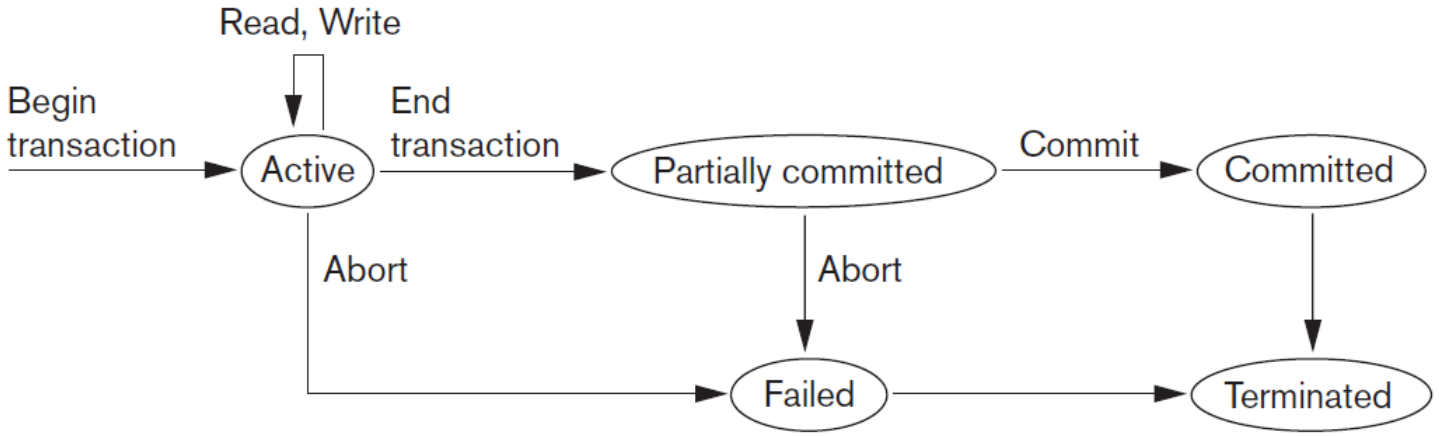


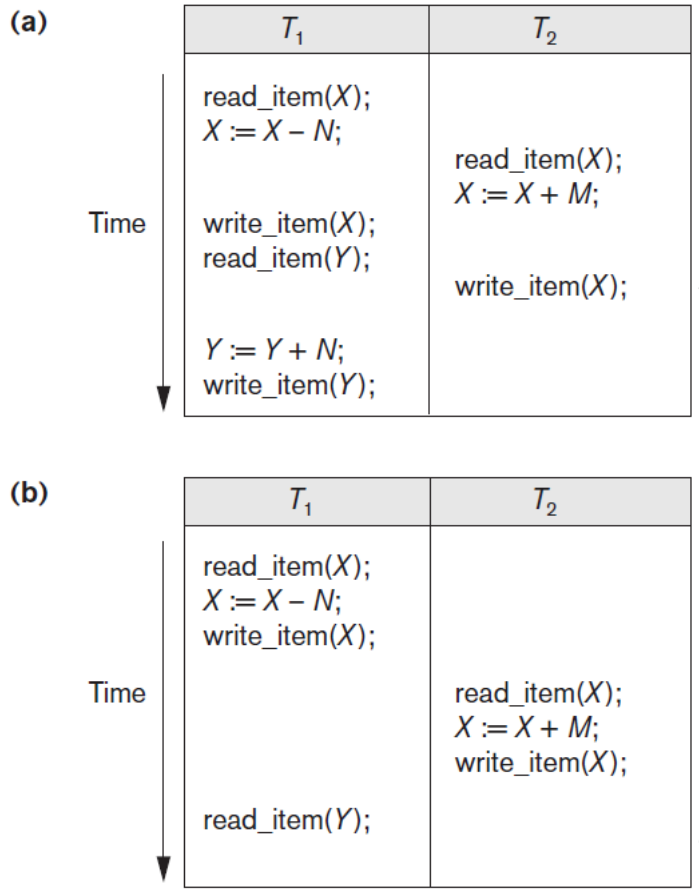
Figure 20.4
State transition diagram illustrating the states for transaction execution.

Schedules of Transactions (1/2)

- A **schedule** (or **history**) S of n transactions T_1, T_2, … , T_n is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule S.
- However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i.
- The order of operations in S is considered to be a total ordering, meaning that for any two operations in the schedule, one must occur before the other.
- A shorthand notation for describing a schedule uses the symbols for the operations begin_transaction (b), read_item (r), write_item (w), end_transaction (e), commit (c), and abort (a), and appends as a subscript the transaction id (transaction number) to each operation in the schedule.

Schedules of Transactions (2/2)

- In some schedules, we will only show the read and write operations, whereas in other schedules we will show additional operations, such as commit or abort.
- The schedule in Figure (a), which we shall call S_a, can be written as follows in this notation:
  S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- The schedule for Figure (b), which we call S_b, can be written as follows, if we assume that transaction T_1 aborted after its read_item(Y) operation:
  S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;



Conflicting Operations in a Schedule (1/2)

- Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
  - They belong to different transactions
  - They access the same item X
  - At least one of the operations is a write_item(X)
- For example, in schedule S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$
  - $r_1(X)$ and $w_2(X)$ conflict, as do $r_2(X)$ and $w_1(X)$, and $w_1(X)$ and $w_2(X)$
  - $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations
  - $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y
  - $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

Conflicting Operations in a Schedule (2/2)

- Intuitively, two operations are conflicting if changing their order can result in a different outcome.
- For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second ordering the value of X is read by $r_1(X)$ after it is changed by $w_2(X)$, whereas in the first ordering the value is read before it is changed. This is called a **read-write conflict**.
- The other type is called a **write-write conflict** and is illustrated by the case where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the last value of X will differ because in one case it is written by T_2 and in the other case by T_1.

Characterizing Schedules Based on Recoverability (1/4)

- It is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple.
- These characterizations do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.
- Once a transaction T is committed, it should never be necessary to roll back T. This ensures that the durability property of transactions is not violated. The schedules that theoretically meet this criterion are called **recoverable schedules**.
- A schedule where a committed transaction may have to be rolled back during recovery is called **unrecoverable** and hence should not be permitted by the DBMS. The schedule suffering from the dirty read problem is unrecoverable.

Characterizing Schedules Based on Recoverability (2/4)

- *The condition for a **recoverable schedule** is as follows: A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X that T reads have committed.*
- For example, S_a': $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1; is recoverable, even though it suffers from the lost update problem.
- For another example, S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1; is not recoverable because T_2 reads item X from T_1, but T_2 commits before T_1 commits. The problem occurs if T_1 aborts after the c_2 operation in S_c; then the value of X that T_2 read is no longer valid and T_2 must be aborted after it is committed, leading to a schedule that is not recoverable.

Characterizing Schedules Based on Recoverability (3/4)

- For the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T_1 commits, i.e., S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;
- If T_1 aborts instead of committing, then T_2 should also abort, i.e.,
  S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;
  because the value of X it read is no longer valid. In S_e, aborting T_2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S_c.
- It is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule Se, where transaction T_2 has to be rolled back because it read item X from T_1, and T_1 then aborted.

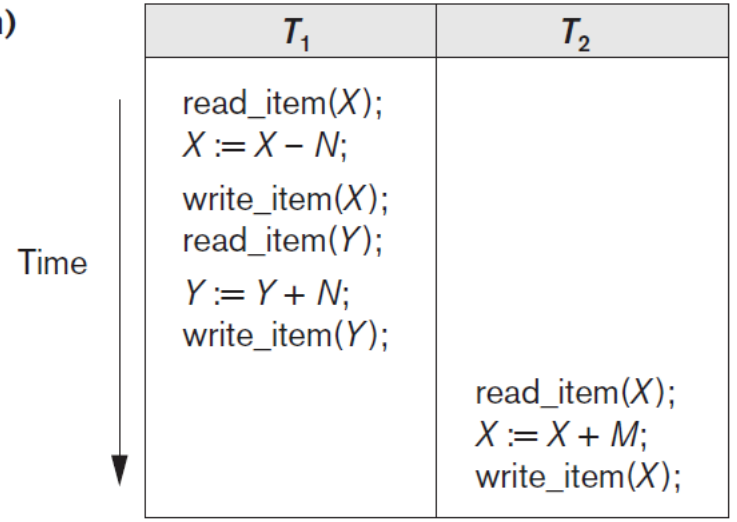
Characterizing Schedules Based on Recoverability (4/4)

- *A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.* In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur.
- There is a more restrictive type of schedule, called a **strict schedule**, in which *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.

Serial and Nonserial Schedules (1/2)

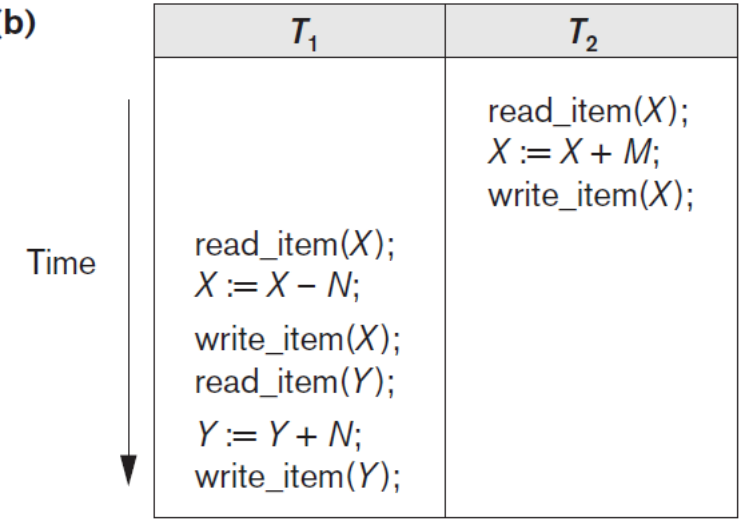
- If no interleaving of operations is permitted, there are only two possible outcomes:
  - Execute all the operations of transaction T1 (in sequence) followed by all th operations of transaction T2 (in sequence).
  - Execute all the operations of transaction T2 (in sequence) followed by all th operations of transaction T1 (in sequence).
- These two schedules—called serial schedules.
- Schedules A and B in Figures (a) and (b) are called serial because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- In a serial schedule, entire transactions are performed in serial order: T_1 and then T_2 in Figure (a), and T_2 and then T_1 in Figure (b).

(a)



Schedule A

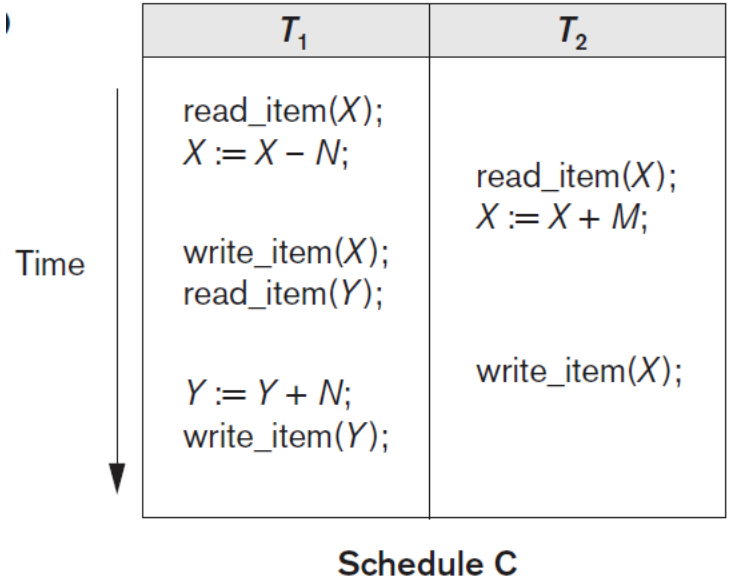
(b)



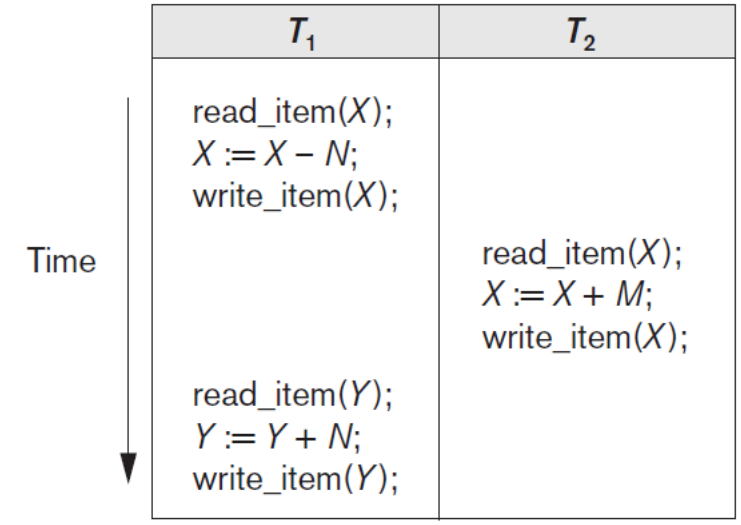
Schedule B

Serial and Nonserial Schedules (2/2)

- Schedules C and D in Figure 20.5(c) are called nonserial because each sequence interleaves operations from the two transactions.
- Formally, a schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule; otherwise, the schedule is called **nonserial**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction.
- No interleaving occurs in a serial schedule.



Schedule C



Schedule D

Serializable Schedule

- A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n
- Saying that a nonserial schedule S is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. transactions.
- For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in the same order.
- Conflict equivalence is generally used as the definition of equivalence of schedules.

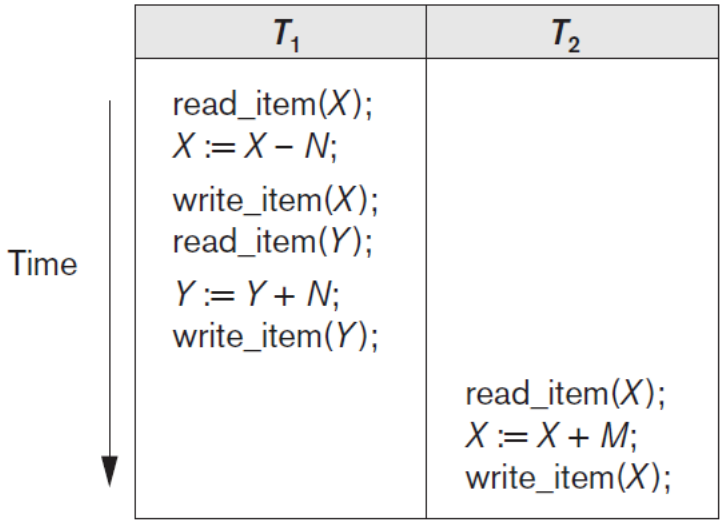
Conflict Equivalence

- Two schedules are said to be **conflict equivalent** if the relative order of any two conflicting operations is the same in both schedules.
- If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent.

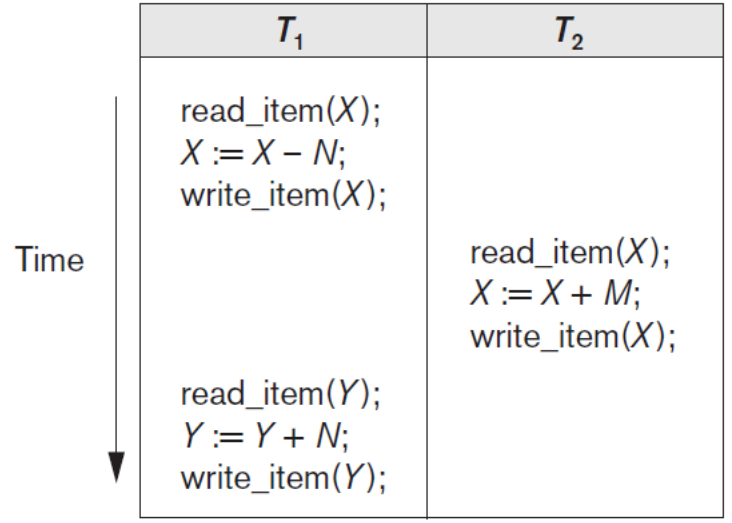
Operations of different transactions on the same data item		Conflict	Reason
Read	Read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed
Read	Write	Yes	Because the effect of a read and a write operation depends on the order of their execution
Write	Write	Yes	Because the effect of a pair of write operations depends on the order of their execution

Serializable Schedules (1/3)

- Using the notion of conflict equivalence, we define a schedule S to be **serializable** if it is (conflict) equivalent to some serial schedule S′. In such a case, we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S′.
- According to this definition, schedule D is equivalent to the serial schedule A. In both schedules, the read_item(X) of T_2 reads the value of X written by T_1 (i.e., $w_1(X)$, $r_2(X)$), whereas the other read_item operations read the database values from the initial database state (i.e., $r_1(X)$, $w_2(X)$).
- Additionally, T_2 is the last transaction to write X in both schedules (i.e., $w_1(X)$, $w_2(X)$).
- Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. We can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule T1, T2.



Schedule A



Schedule D

Serializable Schedules (2/3)

- Schedule C in is not equivalent to either of the two possible serial schedules A and B, and hence is not serializable.
- Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because $r_2(X)$ and $w_1(X)$ conflict, which means that we cannot move $r_2(X)$ down to get the equivalent serial schedule T_1, T_2.
- $w_1(X)$, $r_2(X)$ in schedule A, but $r_2(X)$, $w_1(X)$ in schedule C.

Time

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

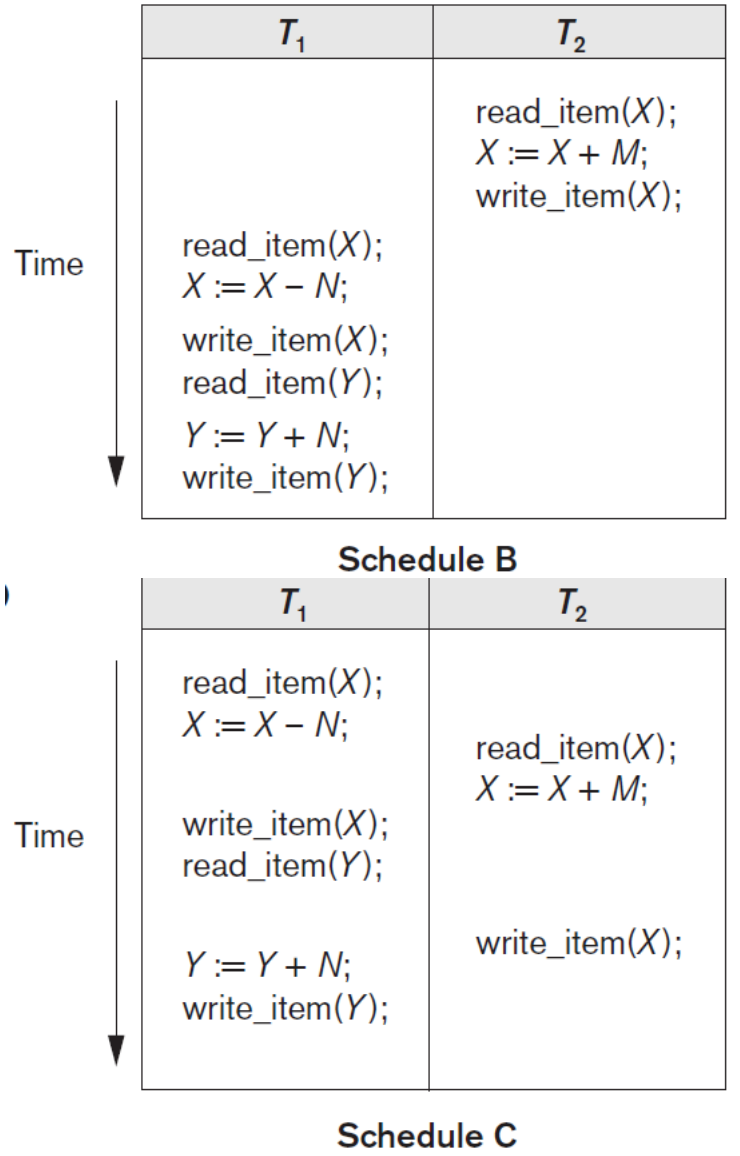
Time

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

Serializable Schedules (3/3)

- Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule T_2, T_1.
- $w_2(X)$, $w_1(X)$ in schedule A, but $w_1(X)$, $w_2(X)$ in schedule C.



Testing for Serializability of a Schedule (1/7)

- There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not.
- The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** G = (N, E) that consists of a set of nodes N = {T_1, T_2, … , T_n } and a set of directed edges E = {e_1, e_2, … , e_m }.
- There is one node in the graph for each transaction T_i in the schedule.
- Each edge e_i in the graph is of the form ($T_j \rightarrow T_k$), 1 ≤ j ≤ n, 1 ≤ k ≤ n, where T_j is the **starting node** of e_i and T_k is the **ending node** of e_i. Such an edge from node T_j to node T_k is created by the algorithm if a pair of conflicting operations exist in T_j and T_k and the conflicting operation in T_j appears in the schedule before the conflicting operation in T_k.

Testing for Serializability of a Schedule (2/7)

- If there is a cycle in the precedence graph, schedule S is **not (conflict) serializable**; if there is no cycle, S is **serializable**.
- A **cycle** in a directed graph is a **sequence of edges** C = (($T_j \rightarrow T_k$), ($T_k \rightarrow T_p$), … , ($T_i \rightarrow T_j$)) with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node)

Testing for Serializability of a Schedule (3/7)

- Algorithm tests conflict serializability of a schedule S
  1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
  2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  5. The schedule S is serializable if and only if the precedence graph has no cycles.

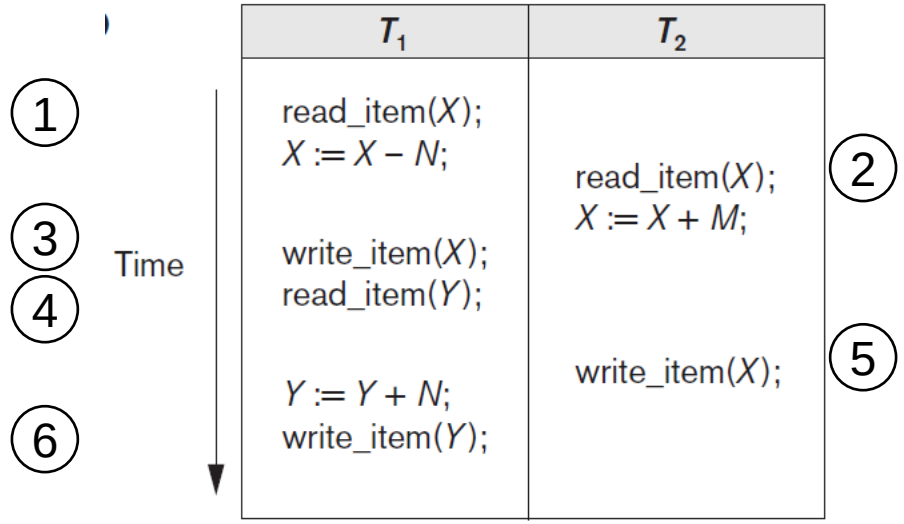
Testing for Serializability of a Schedule (4/7)

- In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order.
- If there is no cycle in the precedence graph, we can create an **equivalent serial schedule** S′ that is equivalent to S, by ordering the transactions that participate in S as follows: Whenever an edge exists in the precedence graph from T_i to T_j, T_i must appear before T_j in the equivalent serial schedule S′.
- If the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable.

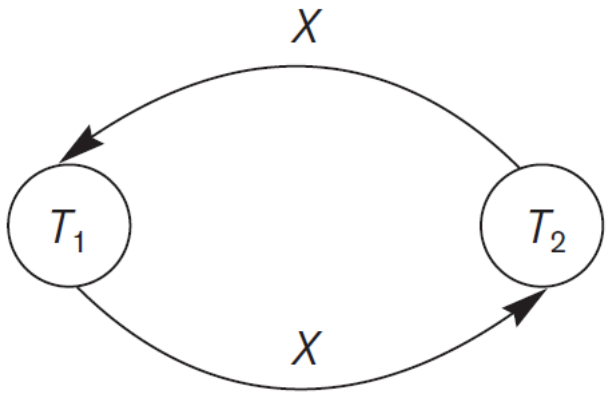
Testing for Serializability of a Schedule (5/7)

① ② ③ ④ ⑤ ⑥

- C: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;
- Step 1: Add two nodes T_1 and T_2
- Step 2: Add edge for read-write conflict
  - Add an edge from T_1 to T_2 for $r_1(X)$; $w_2(X)$;
  - Add an edge from T_2 to T_1 for $r_2(X)$; $w_1(X)$;
- Step 3: Add edge for write-read conflict
  - No write-read conflict
- Step 4: Add edge for write-write conflict
  - Add an edge from T_1 to T_2 for $w_1(X)$; $w_2(X)$;
- Step 5: C is NOT serializable as the precedence graph has a cycle.

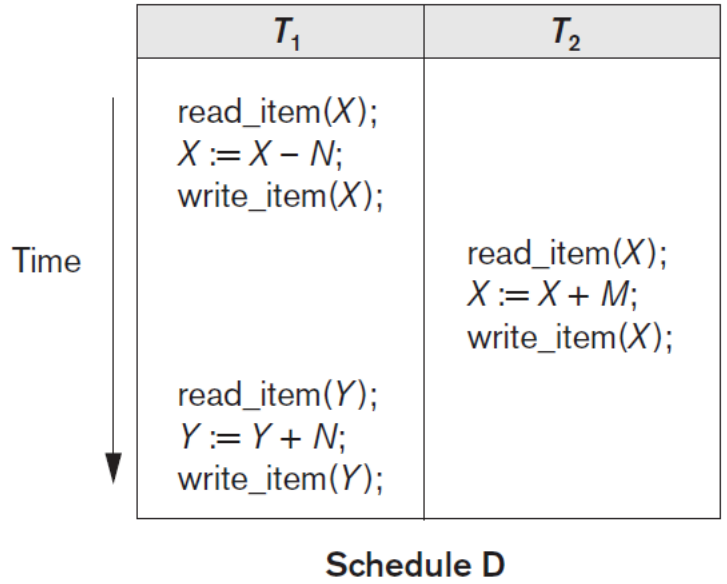


Schedule C

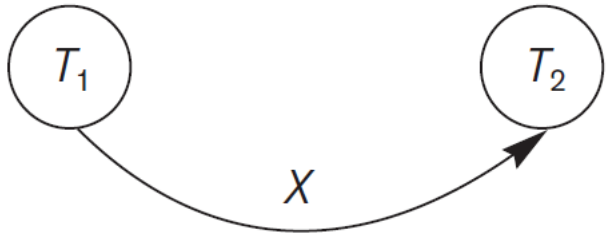


Testing for Serializability of a Schedule (6/7)

- D: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(x)$; $r_1(Y)$; $w_1(Y)$;
- Step 1: Add two nodes T_1 and T_2
- Step 2: Add edge for read-write conflict
  - Add an edge from T_1 to T_2 for $r_1(X)$; $w_2(X)$;
- Step 3: Add edge for write-read conflict
  - Add an edge from T_1 to T_2 for $w_1(X)$; $r_2(X)$;
- Step 4: Add edge for write-write conflict
  - Add an edge from T_1 to T_2 for $w_1(X)$; $w_2(X)$;
- Step 5: D is serializable as the precedence graph has no cycles.



Schedule D

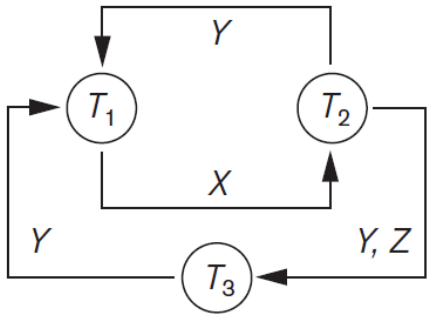


Testing for Serializability of a Schedule (7/7)

- E: $r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X);$
- Step 1: Add three nodes T_1 T_2 and T_3
- Step 2: Add edge for read-write conflict
  - Add edge from T_2 to T_3 for $r_2(Z); w_3(Z);$
  - Add edge from T_2 to T_3 for $r_2(Y); w_3(Y);$
  - Add edge from T_2 to T_1 for $r_2(Y); w_1(Y);$
  - Add edge from T_3 to T_1 for $r_3(Y); w_1(Y);$
  - Add edge from T_1 to T_2 for $r_1(X); w_2(X);$
- Step 3: Add edge for write-read conflict
  - Add edge from T_2 to T_3 for $w_2(Y); r_3(Y);$
  - Add edge from T_2 to T_1 for $w_2(Y); r_1(Y);$
  - Add edge from T_1 to T_2 for $w_1(X); r_2(X);$
  - Add edge from T_3 to T_1 for $w_3(Y); r_1(Y);$
- Step 4: Add edge for write-write conflict
  - Add edge from T_2 to T_3 for $w_2(Y); w_3(Y);$
  - Add edge from T_1 to T_2 for $w_1(X); w_2(X);$
  - Add edge from T_2 to T_1 for $w_2(Y); w_1(Y);$
  - Add edge from T_3 to T_1 for $w_3(Y); w_1(Y);$
- Step 5: E is NOT serializable as the precedence graph has two cycles.

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
Time ↓		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(X);	
read_item(Y);		
write_item(Y);		
	write_item(X);	

Schedule E



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$