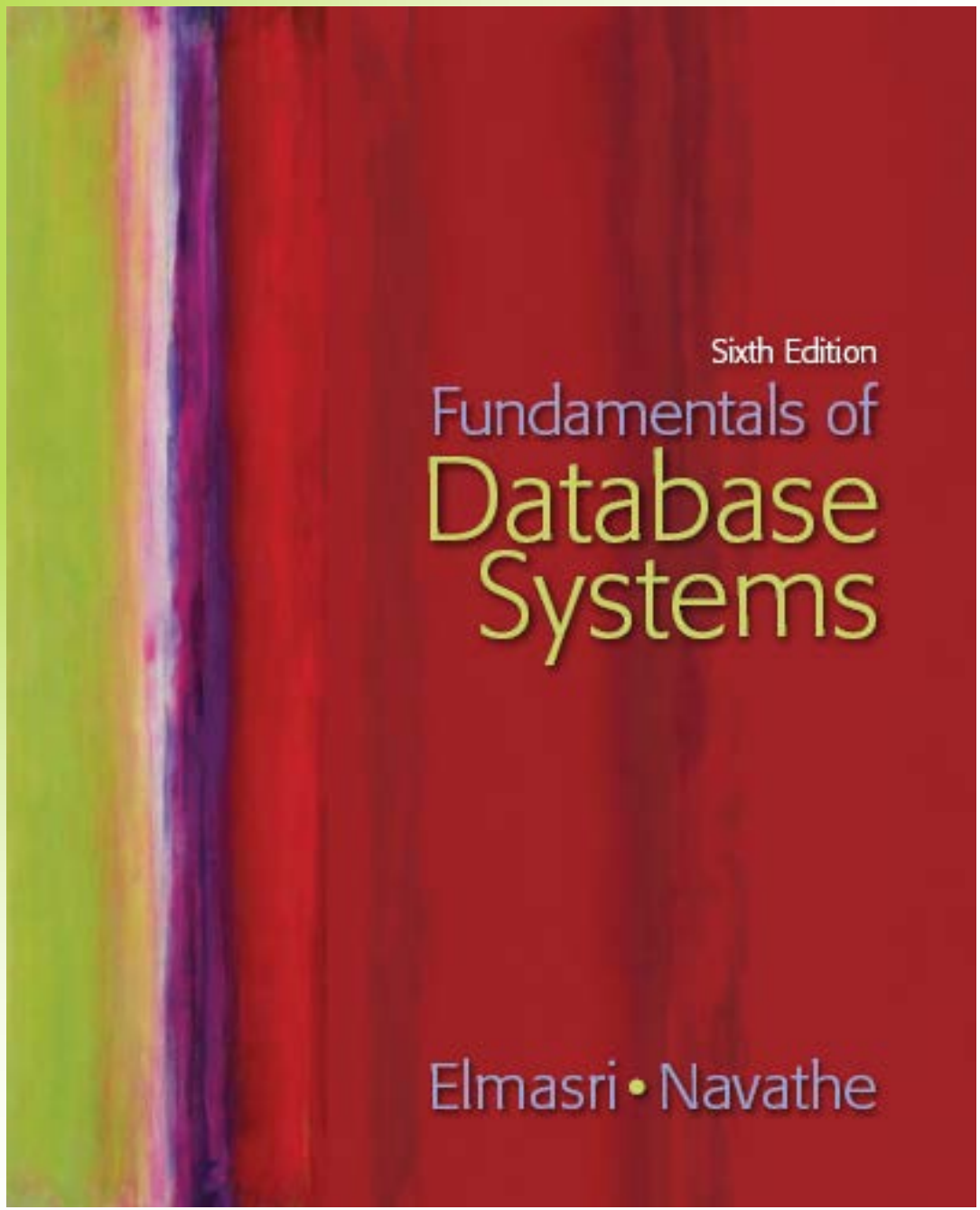
Chapter 21

Introduction to Transaction Processing Concepts and Theory



Addison-Wesley
is an imprint of
PEARSON

1 Introduction to Transaction Processing (1)

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser 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 to Transaction Processing (2)

- **Transaction**:
  - Logical unit of database processing
  - 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
  - Embedded within a program
- **Transaction boundaries**:
  - Begin and End transaction.

Introduction to Transaction Processing (3)

SIMPLE MODEL OF A DATABASE:

- **A database** is a collection of named data items
- **Granularity** of data items - a field, a record, or a whole disk block
- Basic operations are **read** and **write**
  - **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.

Introduction to Transaction Processing (4)

READ AND WRITE OPERATIONS:

- 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, although it may be a larger unit such as a 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 (5)

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 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).

Two Sample Transactions

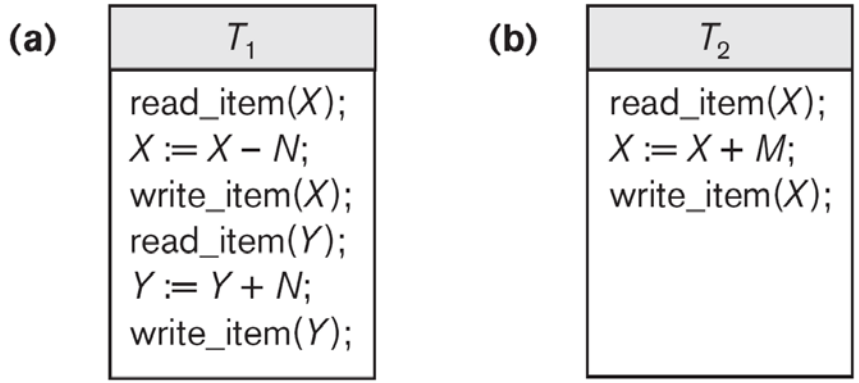


Figure 21.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

Introduction to Transaction Processing (6)

Why Concurrency Control is needed: Transactions should be performed as if they are performed ***serially***.

- **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.
- **Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 21.1.4).
  - The updated item is accessed by another transaction before it is changed back to its original value.
- **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.

Concurrent execution is uncontrolled: (a) lost update problem.

(a)

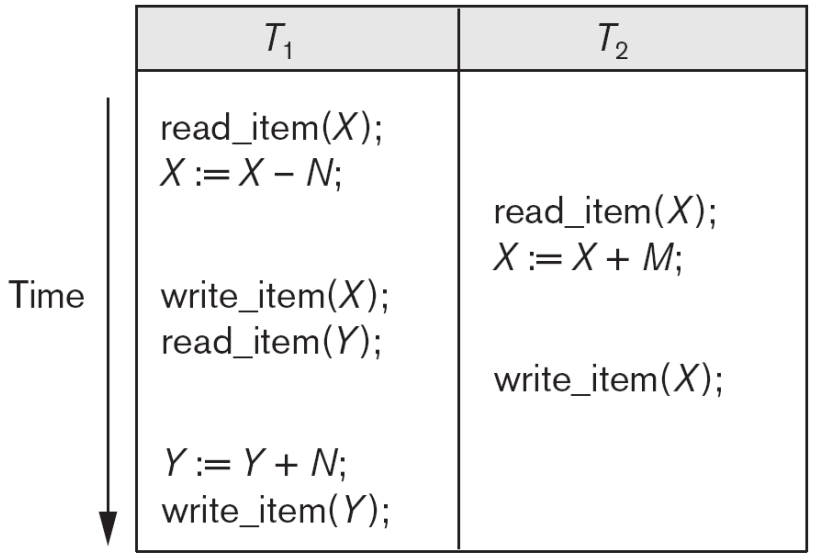
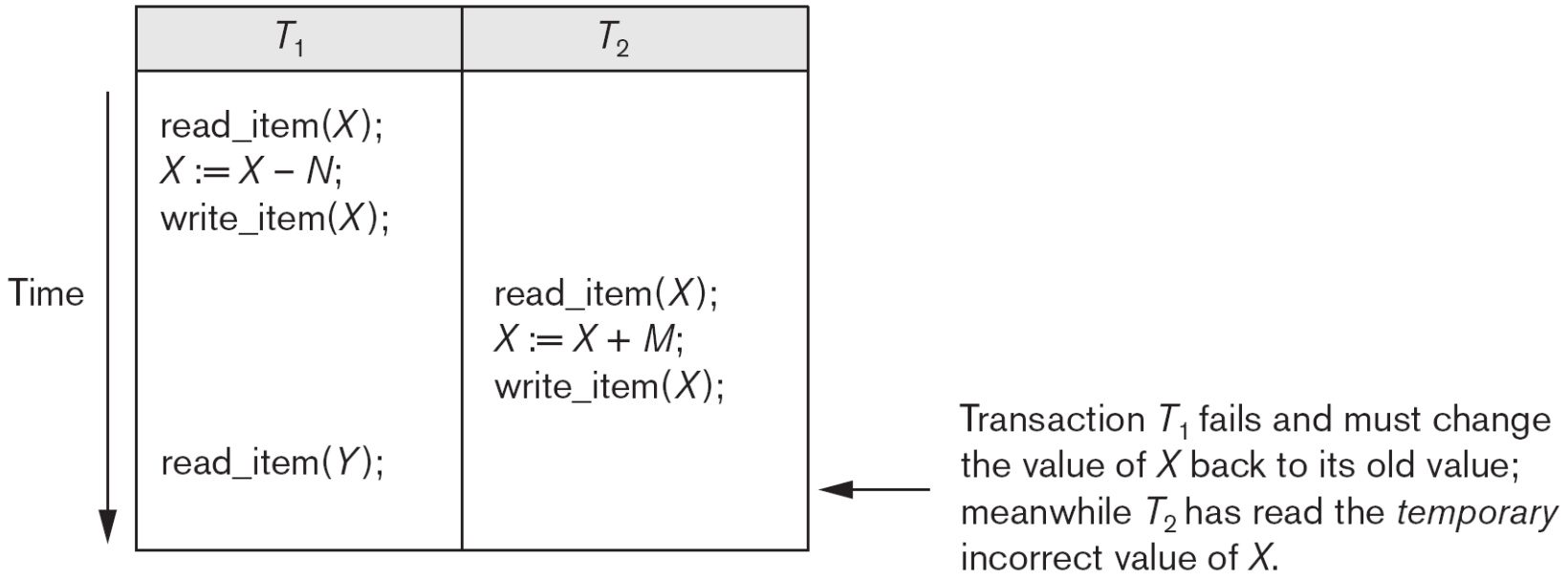


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

Concurrent execution is uncontrolled: (b) temporary update problem.

(b)



Concurrent execution is uncontrolled: (c) incorrect summary problem.

(c)

T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; • • •

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

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

Introduction to Transaction Processing (12)

Why **recovery** is needed: (What causes a Transaction to fail)

1. A computer failure (system crash):
   - A hardware or software error occurs in the computer system during transaction execution.
   - If the hardware crashes, the contents of the computer's internal 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.
   - In addition, the user may interrupt the transaction during its execution.

Introduction to Transaction Processing (13)

Why **recovery** is needed (cont.): (What causes a Transaction to fail)

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

Certain conditions necessitate cancellation of the transaction. For example, a condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, to be restarted later, because several transactions are in a state of deadlock (see Chapter 22).

Introduction to Transaction Processing (14)

Why **recovery** is needed (cont.): (What causes a Transaction to fail)

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 may happen during a read or a write operation of the transaction.

6. Physical problem:

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.

2 Transaction and System Concepts (1)

- 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 and System Concepts (2)

- 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 and System Concepts (3)

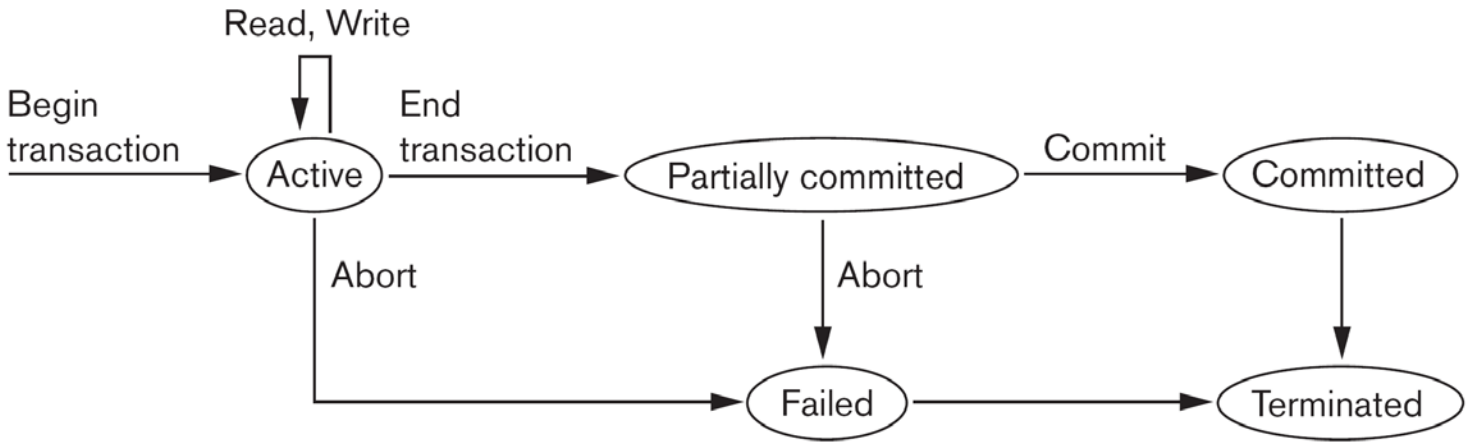
- Recovery manager keeps track of the following operations (cont):
  - **commit_transaction**: This signals a successful end of the transaction
    - So that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback** (or **abort**): 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 and System Concepts (4)

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone*
    - To ensure that all the operations of a committed transaction have been applied successfully to the database.

State Transition Diagram Illustrating the States for Transaction Execution

Figure 21.4
State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts (6)

- System Log
  - **Log** or **Journal**: The log keeps track of all transaction operations that ***affect the values*** of database items.
    - Needed to permit recovery from transaction failures.
    - Kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
    - In addition, periodically backed up to archival storage (tape) to guard against such catastrophic failures.

Transaction and System Concepts (7)

- System Log (cont):
  - T in the following discussion refers to a unique **transaction-id**
    - Generated automatically by the system to identify each transaction:
  - 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 (recorded permanently) to the database.
    - [abort,T]: Records that transaction T has been aborted.

Transaction and System Concepts (9)

Recovery using log records:

- If the system crashes, we can recover to a consistent database state by examining the log.
  1. It is possible to **undo** the effect of the write operations of a transaction T
     1. by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
  2. We can also **redo** the effect of the write operations of a transaction T
     1. by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Transaction and System Concepts (10)

Commit Point of a Transaction:

- **Commit Point:**
  - A transaction T reaches its **commit point**
    - when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database 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.

Transaction and System Concepts (11)

Commit Point of a Transaction (cont):

- **Redoing transactions:**
  - Transactions that have written their commit entry in the log must also have recorded all their write operations in the log;
    - otherwise they would not be committed, so their effect on the database can be redone from the log entries.
  - 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.

3 Desirable Properties of Transactions (1)

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 makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

4 Characterizing Schedules Based on Recoverability (1)

- **Transaction schedule or history**:
  - The order of execution of operations from the various transactions forms.
    - When transactions are executing concurrently in an interleaved fashion
- A **schedule** (or **history**) S of n transactions T1, T2, …, Tn:
  - Ordering of the operations of the transactions subject to the constraint that,
    - For each transaction Ti that participates in S, the operations of Ti in S must appear in the same order in which they occur in Ti.
  - Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

Characterizing Schedules Based on Recoverability (2)

Schedules classified on recoverability:

- **Recoverable schedule**:
  - One where no transaction needs to be rolled back.
  - 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.

5 Characterizing Schedules Based on Serializability (1)

- Serial schedule:
  - 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 schedule.
- Serializable schedule:
  - A schedule S is ***serializable*** if it is equivalent to some serial schedule of the same *n* transactions.

Characterizing Schedules Based on Serializability (3)

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - 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.

Characterizing Schedules Based on Serializability (5)

Practical approach:

- Come up with methods (protocols) to ensure serializability.
- It's not possible to determine when a schedule begins and when it ends.
  - Hence, we reduce the problem of checking the whole schedule to checking only a **committed project** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
  - Use of locks with two phase locking

Characterizing Schedules Based on Serializability (7)

- Two schedules are said to be view equivalent if the following three conditions hold:
  1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
  2. 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, the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
  3. 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'.

Characterizing Schedules Based on Serializability (8)

- The premise behind view equivalence:
  - 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 same view* in both schedules.

Characterizing Schedules Based on Serializability (11)

Testing for conflict serializability: Algorithm 21.1:

- 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 Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

Constructing the Precedence Graphs

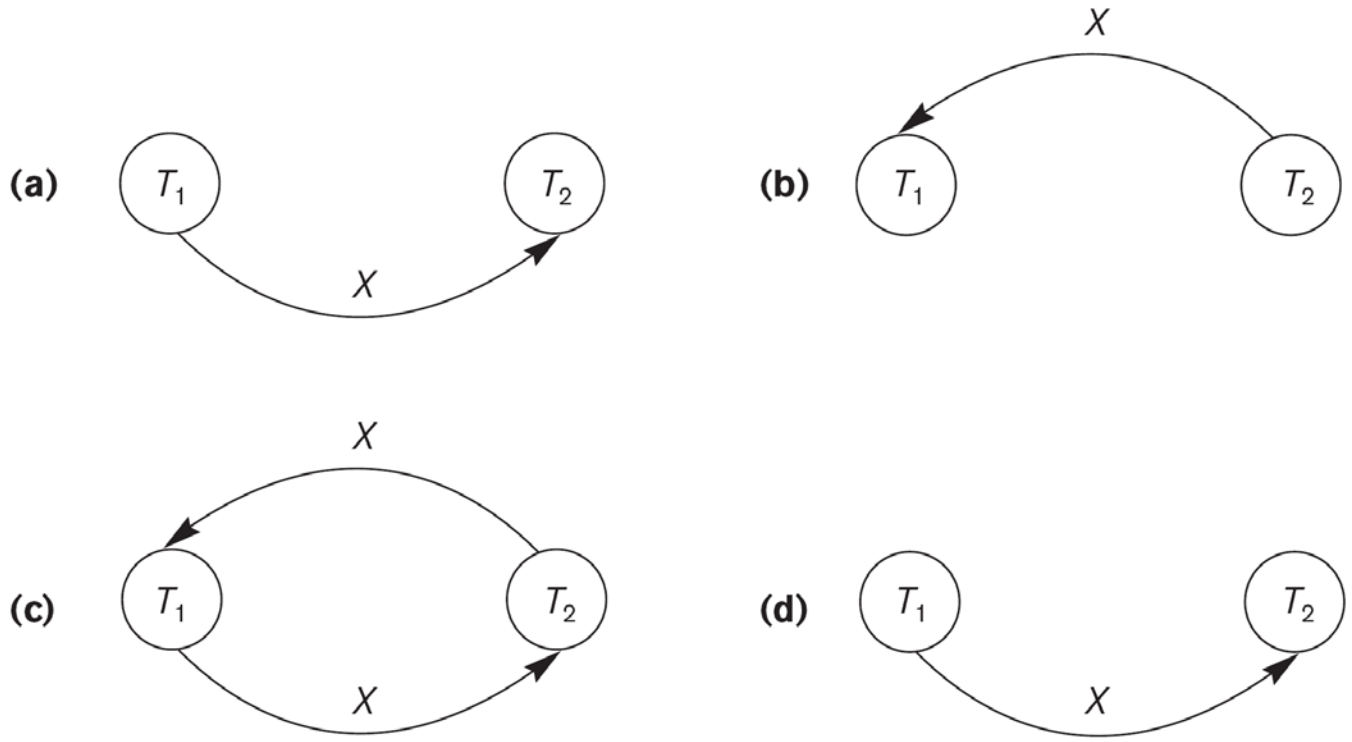


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Another Example of Serializability Testing

(a)

Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

Transaction T_2
read_item(Z);
read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

Another Example of Serializability Testing

(b)

Transaction T_1	Transaction T_2	Transaction T_3
	read_item(Z);	
	read_item(Y);	
	write_item(Y);	
		read_item(Y);
		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);	

Time

Schedule E

Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

Another Example of Serializability Testing

(c)

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

Time

Schedule F

Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.