Chapter 20: Transaction Processing

CS-6360 Database Design

Dr. Chris Irwin Davis

Email: cid021000@utdallas.edu
Phone: (972) 883-3574
Office: ECSS 4.705

Introduction to Transaction Processing

- A **Transaction**:
  - Logical unit of database processing that includes one or more access operations: read (retrieval) or 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.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

Transaction Processing Hierarchy

- Schedule
  - Transaction
    - Operation
      - Sub-operation

- **Simple Model of a Database**
  - for purposes of discussing transactions:
- A database is a collection of named data items
- **Granularity of data** - a field, a record , or a whole disk block
  - Concepts are independent of granularity
- 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*.

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.

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

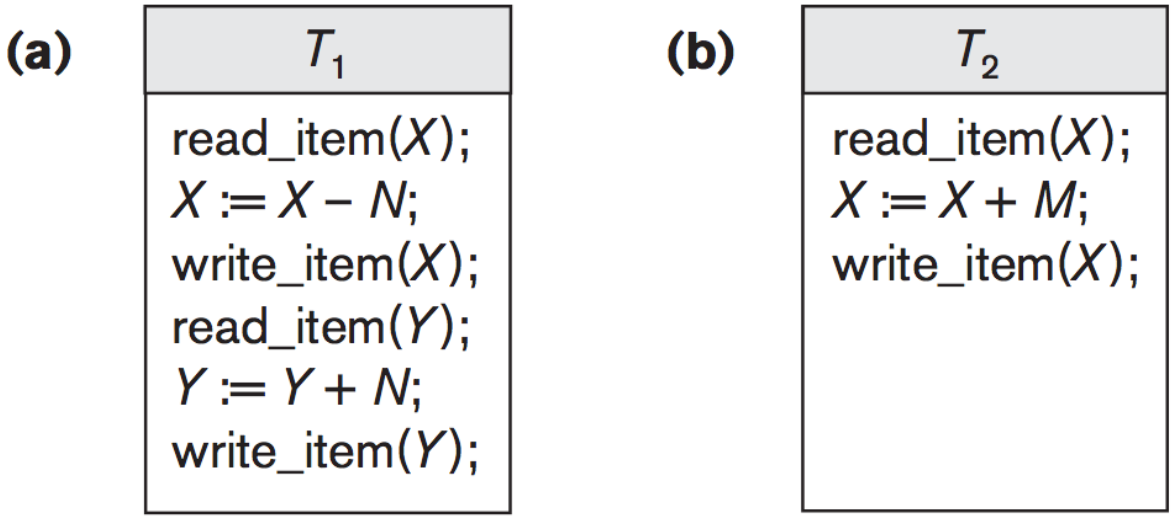


Figure 20.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

Two Sample Transactions

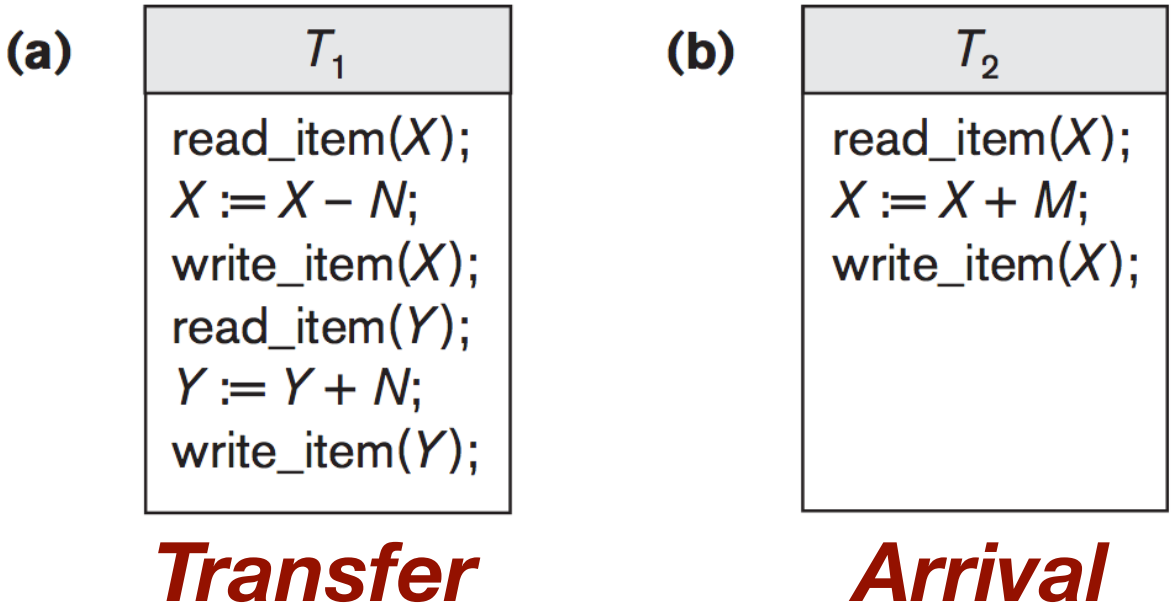


Figure 20.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

Why Concurrency Control is needed:

- **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 (see Section 21.1.4). The updated item is accessed by another transaction before it is changed back (reverted) to its original value.

- **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.

- **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.

Concurrent execution is uncontrolled:

(a) The lost update problem.

(a)

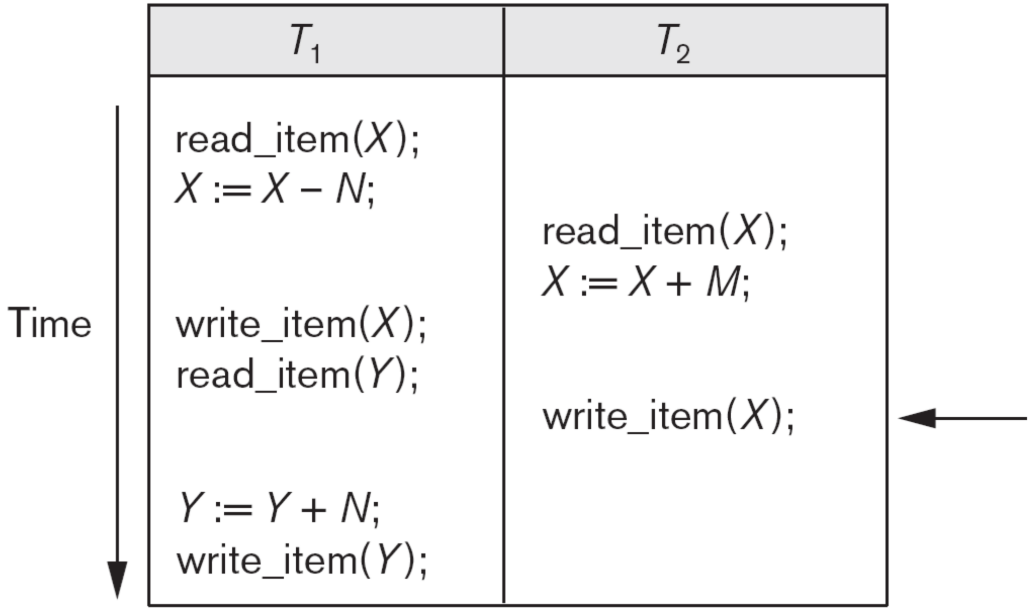
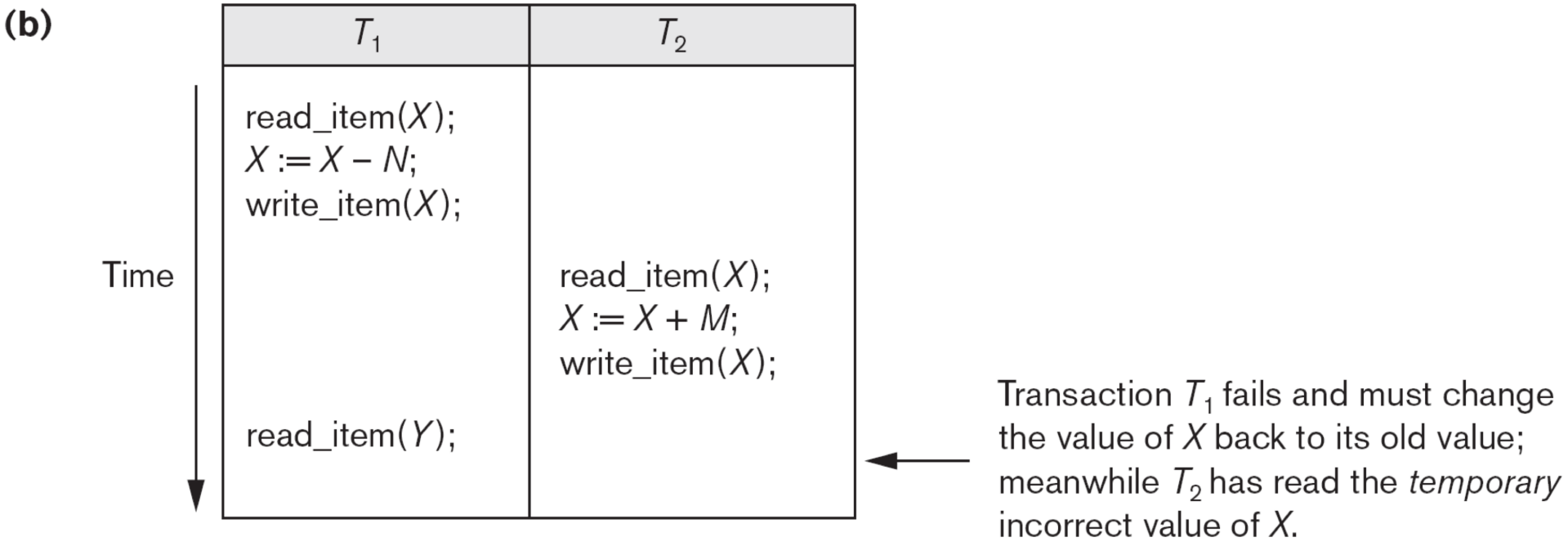


Figure 20.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

Concurrent execution is uncontrolled:

(b) The temporary update problem.



Concurrent execution is uncontrolled:

(c) The 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*).

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
  - The transaction does not have *any* effect on the database or any other transactions (i.e. *full* revert)
- 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 only 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:

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. 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.

Types of Failures (cont'd)

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 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 it violates serializability or because several transactions are in a state of deadlock (see Chapter 22).

Types of Failures (cont'd)

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 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.

§20.2 Transaction and System Concepts

Transaction and System Concepts

- 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 and System Concepts

UTD

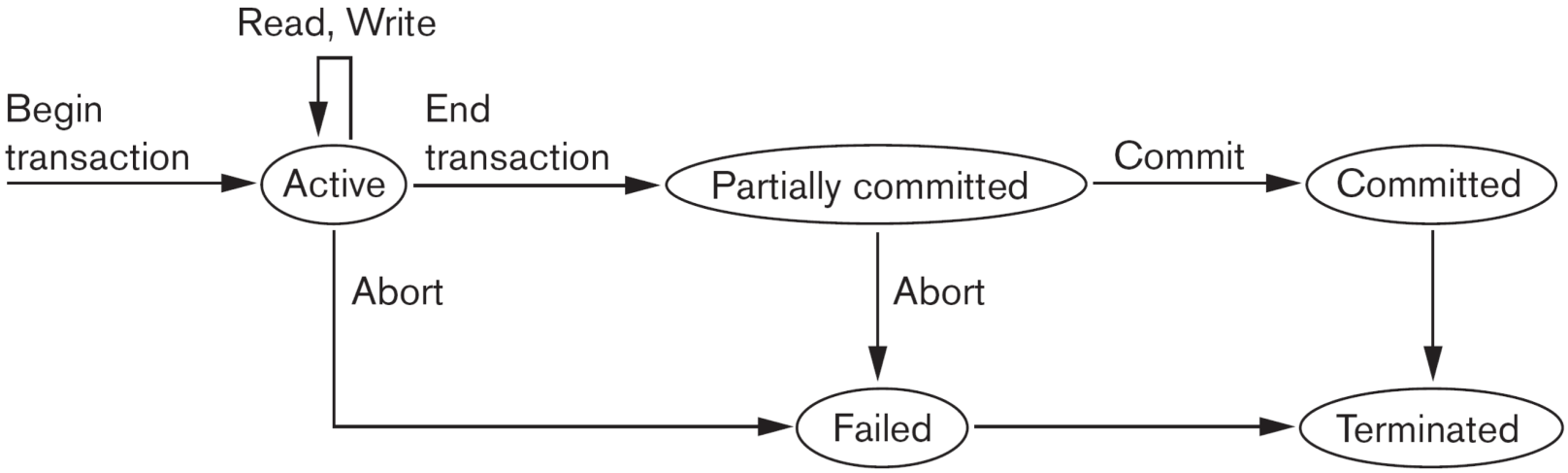
- Recovery manager keeps track of the following **operations**:
  - **BEGIN_TRANSACTION**: This marks the beginning of transaction execution.
  - **READ** or **WRITE**: These specify read or write operations on the database items that are executed as part of a transaction.
  - **END_TRANSACTION**: This 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.

- Recovery manager keeps track of the following **operations**:
  - **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.

- Recovery manager keeps track of the following **operations**:
  - **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 20.4
State transition diagram illustrating the states for transaction execution.



System Log

The System Log

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

The System Log (cont'd)

- *T* in the following discussion refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:
  - Each of the following System Log entry types are associated with a **Transaction ID**.

Types of System Log Entries

- `[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.

The System Log (cont'd)

- Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log,* whereas other protocols require these entries for recovery.
- Strict protocols require simpler write entries that do not include new_value (see Section 21.4).

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 described in Chapter 19.

1. Because the log contains a record of every **`write`** operation that changes the value of some database item, it is possible to **undo** the effect of these **`write`** operations of a transaction *T* 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* 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`**.

Commit Point of a Transaction

- **Definition a 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.

Commit Point of a Transaction (cont'd)

- **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
  - 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.

§21.3 – Desirable Properties of Transactions

Desirable Properties of Transactions

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.

21.4 – Characterizing Schedules Based on Recoverability

Characterizing Schedules Based on Recoverability

UTD

- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or history).
- A **schedule** 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. Otherwise, T_i has been redefined.

Characterizing Schedules Based on Recoverability

- **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").
- A **schedule** (or "history") S of n transactions T_1, T_2, …, T_n:
  - It is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_1 in S must appear in the same order in which they occur in T_1.
  - Note, however, that operations from other transactions T_j *can be interleaved* with the operations of T_i in S.

Characterizing Schedules Based on Recoverability

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.

- **Cascadeless schedule**:
  - One where every transaction can `read` only the items that are written by committed transactions.

Characterizing Schedules Based on Recoverability

Schedules classified on recoverability (cont'd):

- **Schedules requiring cascaded rollback**:
  - A schedule in which uncommitted transactions that **`read`** an item from a failed transaction must be rolled back (see previous bullet).
- **Strict Schedules**:
  - A schedule in which a transaction can *neither* **`read`** *nor* **`write`** an item *X* until the last transaction that wrote *X* has committed.

§21.5 – Characterizing Schedules Based on Serializability

Characterizing Schedules Based on Serializability

- **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 non-serial 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

- **Result equivalent:**
  - Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent:**
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable:**
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Result Equivalence

- **Result Equivalence** example that is neither extendible nor global

Figure 20.6
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);

Characterizing Schedules Based on Serializability

- Being **serializable** is not the same as being **serial**
  - **serializable ≠ serial**
- A **serial** schedule is trivially correct.
- 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

UTD

- Serializability is hard to check. Because…
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

- If no interleaving of operations is permitted, there are only two possible outcomes:
  - Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence)
  - Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

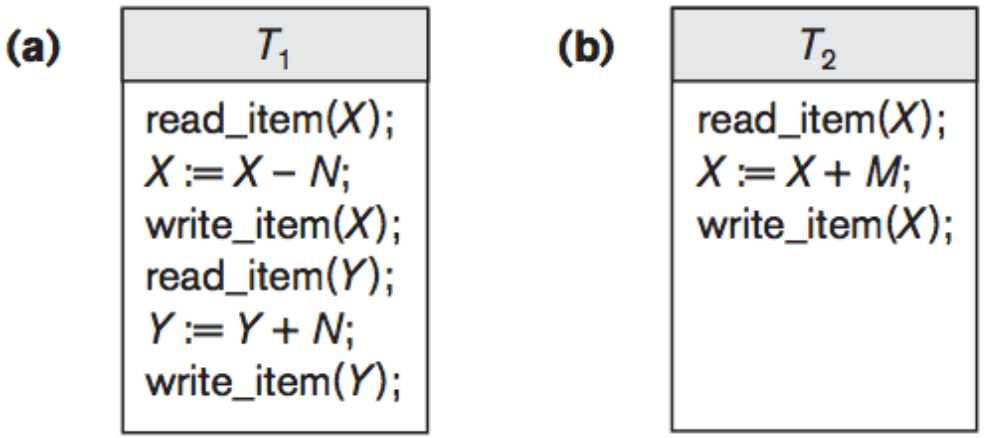
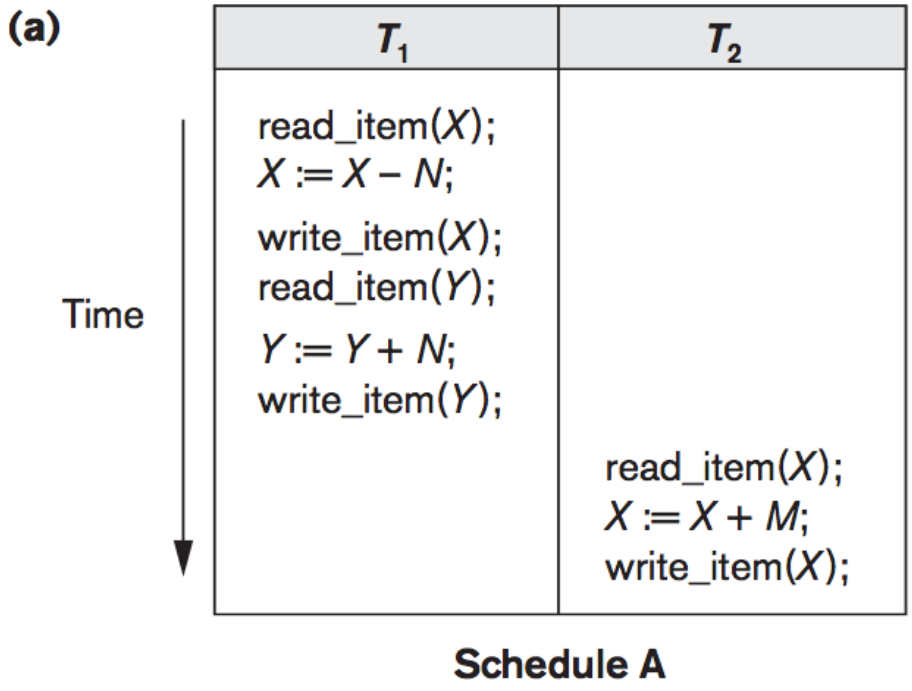


Figure 20.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

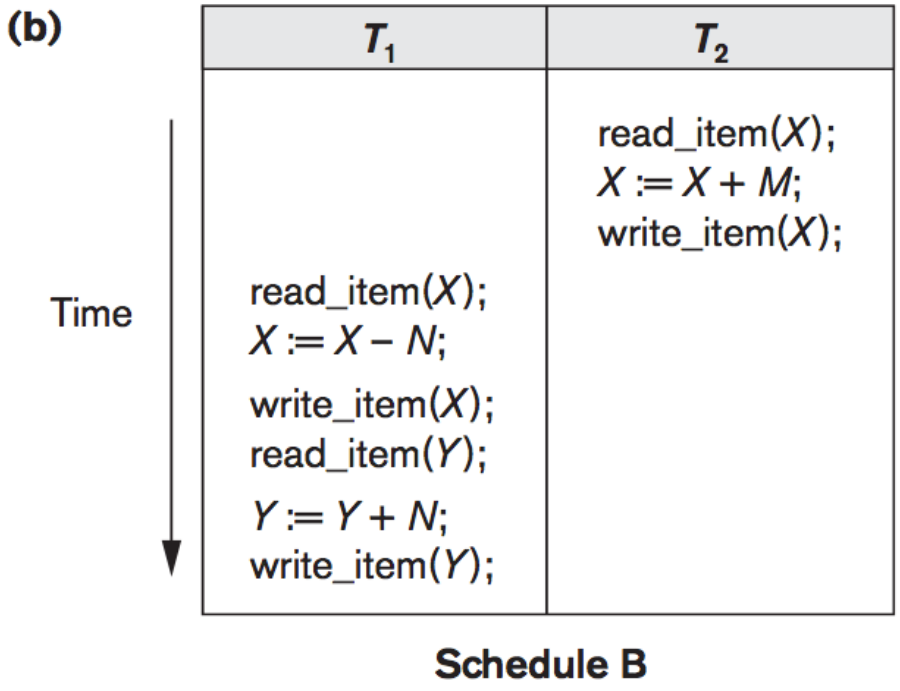
Interleaving Schedules: Scenarios

(a) **Serial** schedule A: $T1$ followed by $T2$.

(b) **Serial** schedule B: $T2$ followed by $T1$.



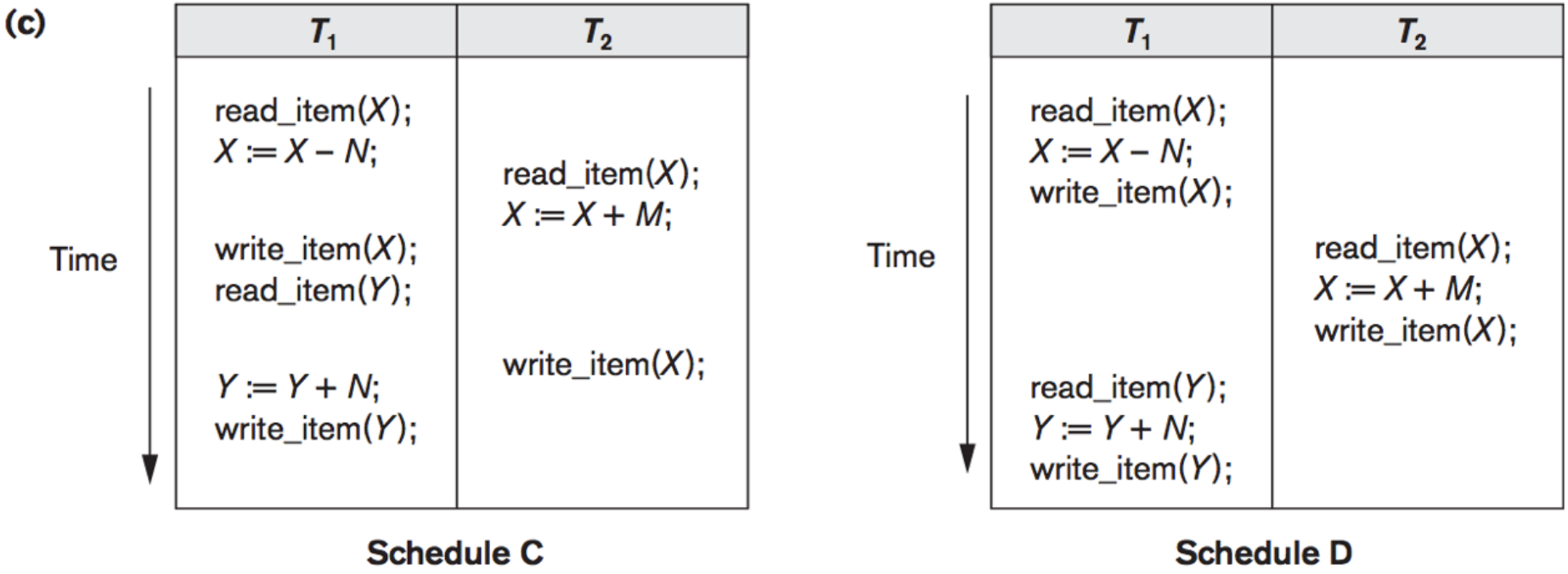
Schedule A



Schedule B

Interleaving Schedules: Scenarios

(c) Two **non-serial** schedules C and D with interleaving of operations.



- 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

UTD

- View equivalence:
  - A less restrictive definition of equivalence of schedules
- View serializability:
  - Definition of serializability based on view equivalence.
  - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Characterizing Schedules Based on Serializability

UTD

- 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

- 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 $R_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $W_j(X)$ of T_j (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 $R_i(X)$ of T_i in S'.
3. If the operation $W_k(Y)$ of T_k is the last operation to write item Y in S, then $W_k(Y)$ of T_k must also be the last operation to write item Y in S'.

Characterizing Schedules Based on Serializability

UTD

- Relationship between **view equivalence** and **conflict equivalence**:
  - The two are the **same** under "constrained write assumption" which assumes that if T writes X, it is constrained by the value of X it read; i.e., new X = f(old X)
  - However, conflict serializability is **stricter** than view serializability.
    - **With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.**
  - Thus, any conflict serializable schedule is also view serializable, but not vice versa.

Characterizing Schedules Based on Serializability

- Relationship between view and conflict equivalence (cont'd):
  - Consider the following schedule of three transactions
    - T_1: $r_1(X)$, $w_1(X)$;
    - T_2: $w_2(X)$;
    - T_3: $w_3(X)$:
  - Schedule S_a: $r_1(X); w_2(X); w_1(X); w_3(X); c_1; c_2; c_3;$

- In S_a, the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X.
  - S_a is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3.
  - However, S_a is not conflict serializable, since it is *not* conflict equivalent to any serial schedule.

Testing Conflict Serializability of a Schedule S

Algorithm 20.1 Testing Conflict Serializability of a Schedule S

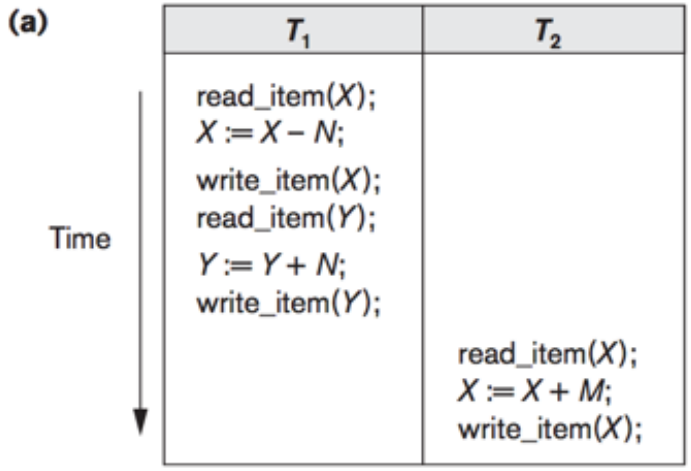
1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes 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 conflict serializability: Algorithm 20.1:

- 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.

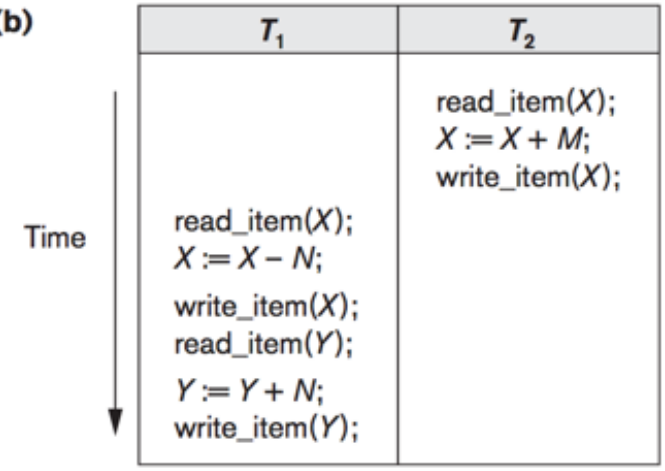
Construct Precedence Graphs for a, b, c, & d

(a)



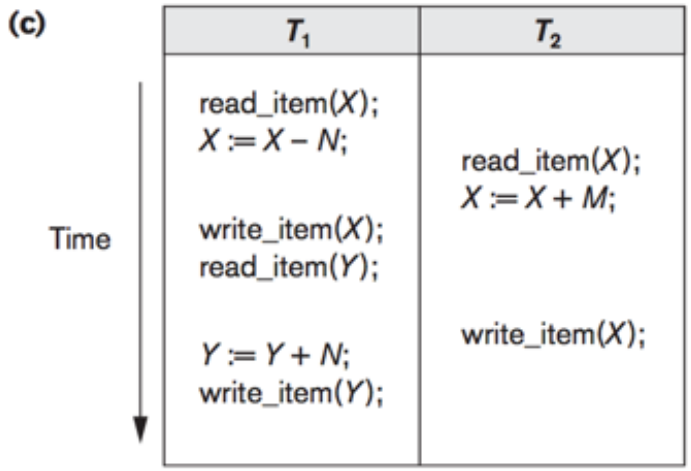
Schedule A

(b)

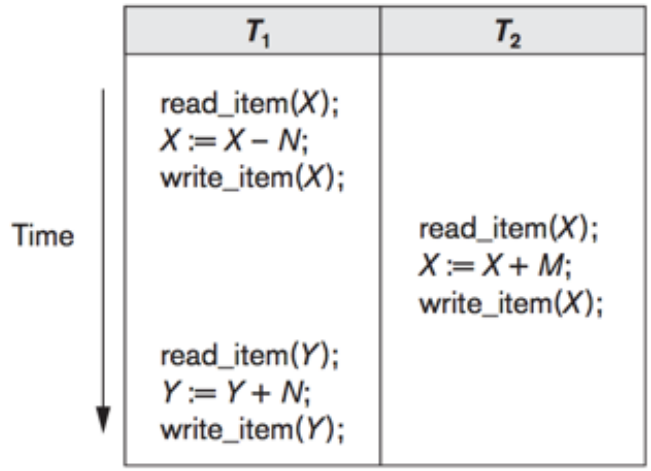


Schedule B

(c)



Schedule C



Schedule D

Constructing the Precedence Graphs

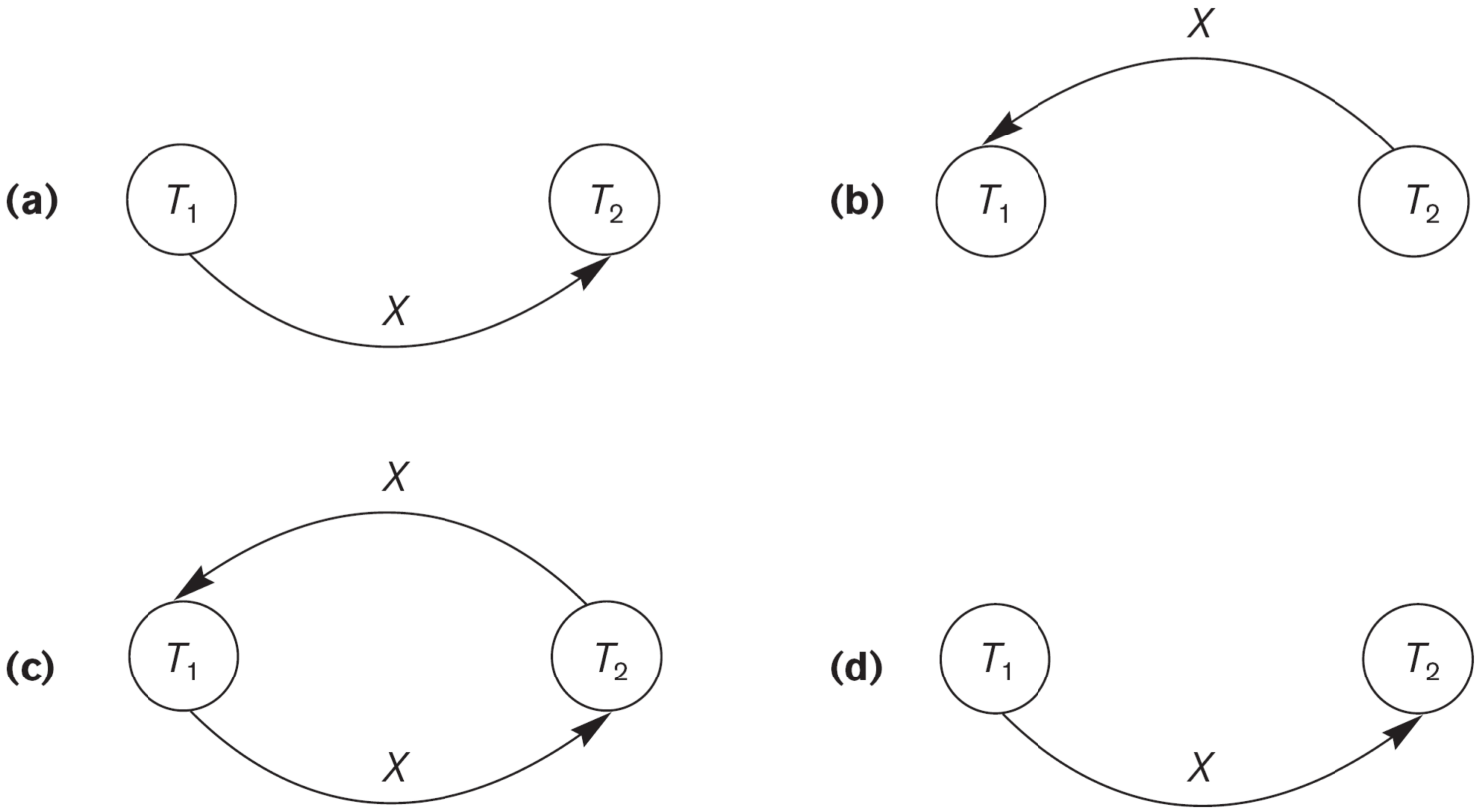


Figure 20.7

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).

Constructing the Precedence Graphs

- Required components of a Precedence Graph:
  - A ***node*** to represent each Transaction, even those which have no dependencies.
  - An ***single directed edge*** to represent *each set items* for a given ordered pairs of transactions,
    - e.g. $X(T_1 \rightarrow T_2)$ and $Y(T_1 \rightarrow T_2)$ only require a single edge with the combined label **X,Y**.
    - However, $X(T_1 \rightarrow T_2)$ and $Y(T_2 \rightarrow T_1)$ require two directed edges since these two dependencies are in different directions, i.e. Transactions are *ordered pairs*.
  - A ***label*** for each directed edge of potentially multiple items

Constructing the Precedence Graphs

- Required components of a Precedence Graph:
  - No cycles? $\Longrightarrow$ Serializable (**good**)
    - List ALL equivalent Serial Schedules
  - Yes cycles? $\Longrightarrow$ Non-serializable (**bad**)
    - List ALL cycles

Another Example of Serializability Testing

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)

Time

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);	

Schedule E

Another Example of Serializability Testing

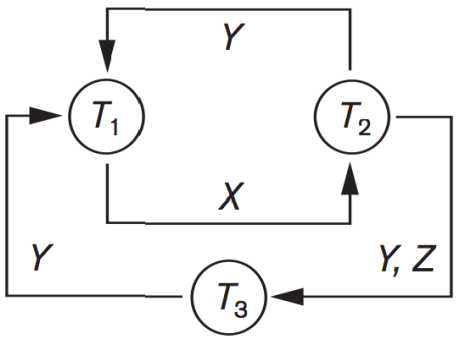
(b)

Time

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);	

Schedule E

(d)



Equivalent serial schedules

None

Reason

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

Another Example of Serializability Testing

(c)

Time

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);	

Schedule F

Another Example of Serializability Testing

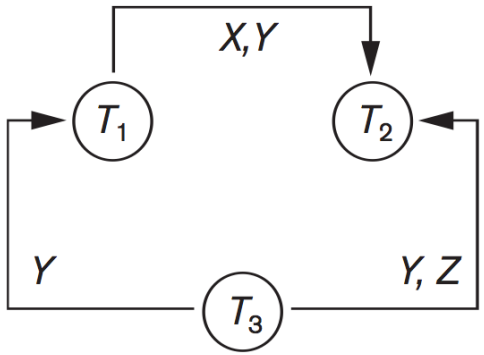
(c)

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

(e)

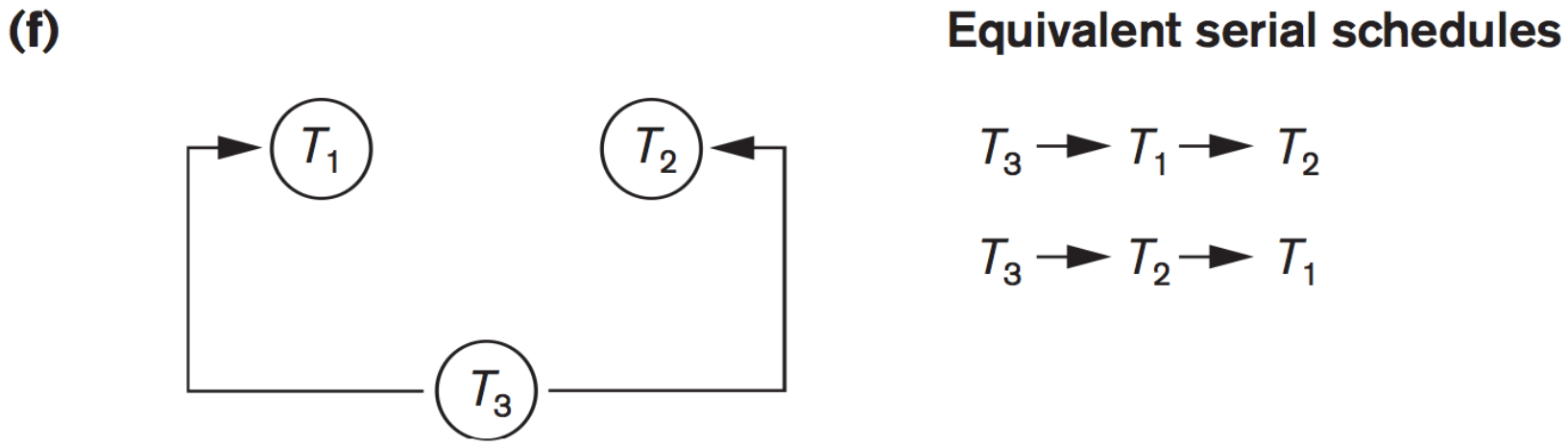


Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Equivalent Serial Schedules

- Equivalent Serial Schedules may not be unique
- Precedence graph with two equivalent serial schedules.



Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly.
  - Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly