Database Management System (MDS 505)

Jagdish Bhatta

Unit – 5.1

Transaction Processing

Single User Vs. Multi-User System

- One criterion for classifying a database system is according to the number of users who can use the system **concurrently**. 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.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- 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.
- The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- 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. A single application program may contain more than one transaction if it contains several transaction boundaries.
- 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**.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- The *database model* that is used to present transaction processing concepts is simple when compared to the data models that we discussed earlier, such as the relational model or the object model.
- A **database** is basically represented as a collection of *named data items*. The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. If the item granularity is a single record, then the record id can be the item name.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- 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*.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- 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*.
- 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 is 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).

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- In write_item(*X*) command, It is step 4 that actually updates the database on disk. Sometimes the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system.
- The DBMS will maintain in the **database cache** a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some **buffer replacement policy** is used to choose which of the current occupied buffers is to be replaced.

Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- A transaction includes read_item and write_item operations to access and update the database. Following shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of $T1$ in the figure is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

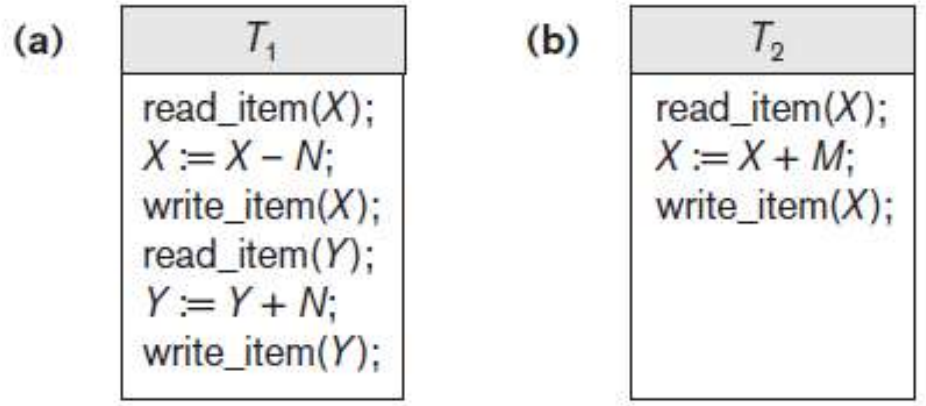


Figure 20.2
Two sample transactions.
(a) Transaction T_1.
(b) Transaction T_2.

Why Concurrency Control Is Needed?

- **Concurrency control and recovery mechanisms** are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- If this concurrent execution is *uncontrolled*, it may lead to problems, such as an inconsistent database. In the next section, we informally introduce some of the problems that may occur.

Why Concurrency Control Is Needed?

- Generally, database system allows multiple transactions to run concurrently. Concurrent execution of transaction in database system improves database system performance, reducing transaction waiting time to proceed. It improves resource utilization. But it may lead the database in inconsistent state due to interference among actions of concurrent transactions. Concurrent execution of transaction in database system leads several concurrency control problems.

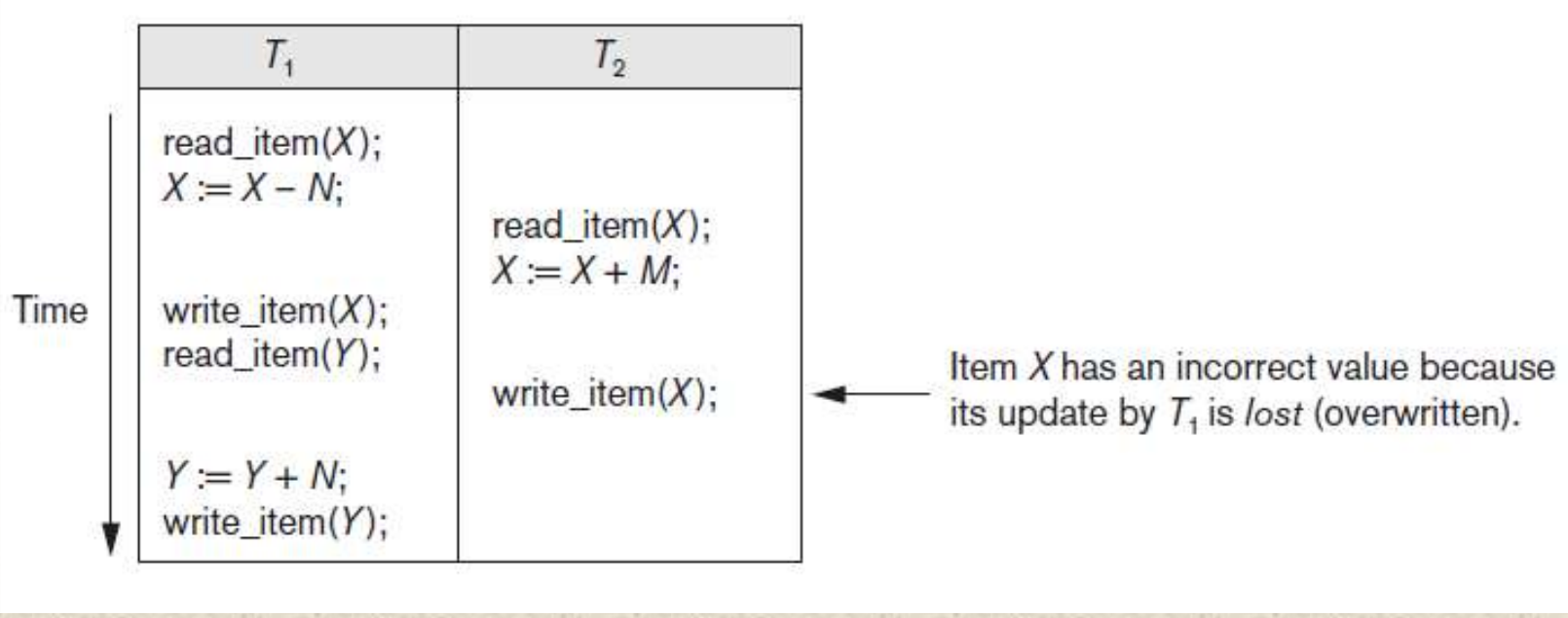
Why Concurrency Control is needed?

- With concurrent execution of transactions, if they are allowed to execute in uncontrolled manner several problems may occur such as;

- **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.

Why Concurrency Control is needed?

- **The Lost Update Problem:** Suppose that transactions *T*1 and *T*2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in following figure; 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. For example, if *X* = 80 at the start, *N* = 5, and *M* = 4, the final result should be *X* = 79. However, in the interleaving of operations shown in the figure, it is *X* = 84 because the update in *T*1 that removed the five seats from *X* was *lost*.

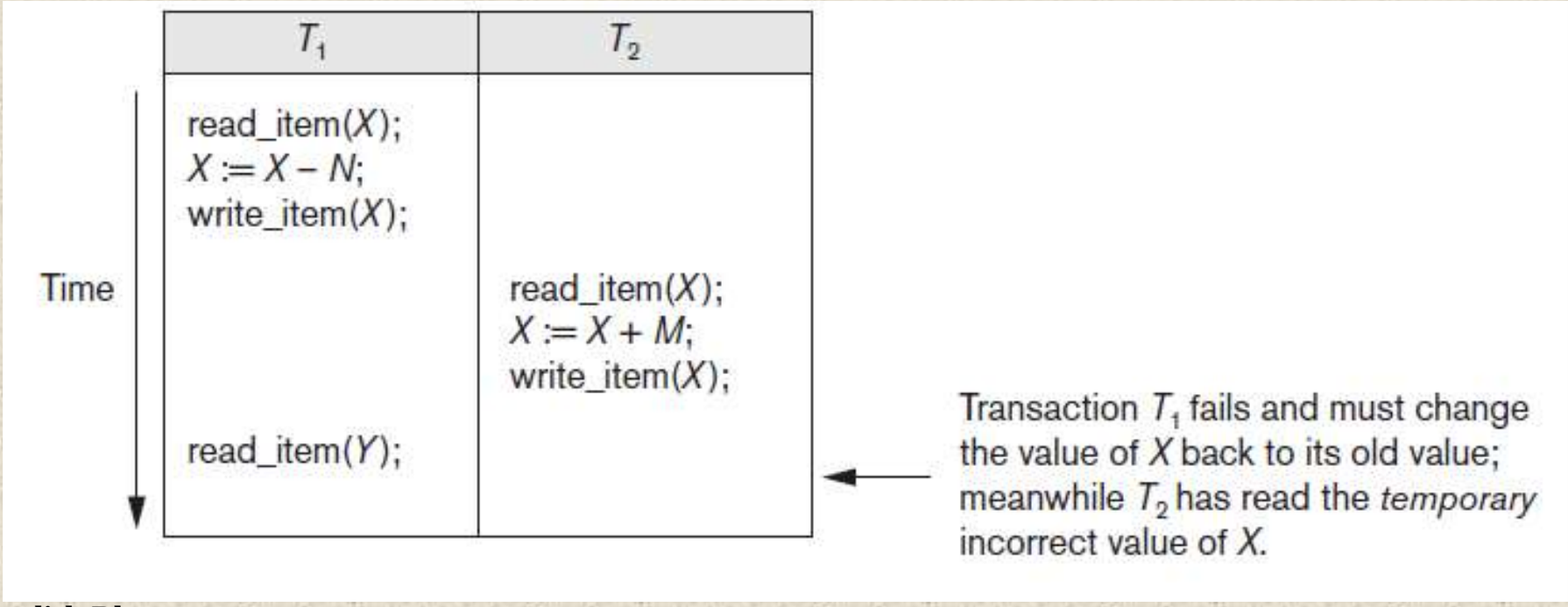


Why Concurrency Control is needed?

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

Why Concurrency Control is needed?

- **The Temporary Update (or Dirty Read) Problem:** Following Figure shows an example where *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; hence, this problem is also known as the *dirty read problem*.

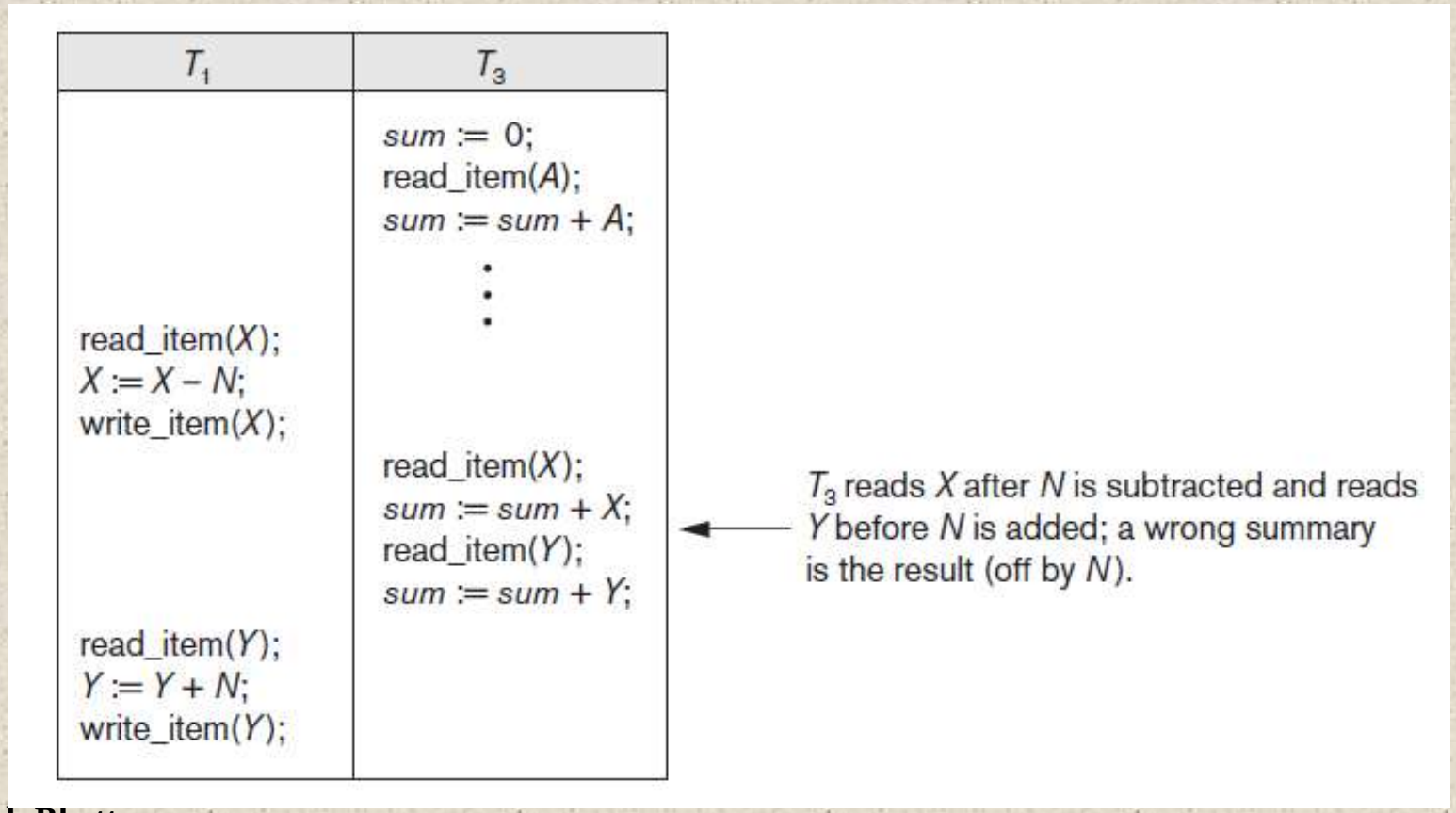


Why Concurrency Control is needed?

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

Why Concurrency Control is needed?

- **The Incorrect Summary Problem:** For example, suppose that a transaction *T*3 is calculating the sum; meanwhile, transaction *T*1 is executing. If the interleaving of operations shown in following figure occurs, the result of *T*3 will be off by an amount *N* because *T*3 reads the value of *X after N* have been subtracted from it but reads the value of *Y before* those *N* 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.

Why Recovery is needed?

- **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, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
  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. Additionally, the user may interrupt the transaction during its execution.
  3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

Why Recovery is needed?

- **Types of Failures:**
  4. **Concurrency control enforcement.** The concurrency control method may abort a transaction because it violates serializability, or it may abort one or more transactions to resolve a state of deadlock among several transactions. Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
  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.
- Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task.

Transaction and System Concepts

- **Transaction States and Operations**
- 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.
  - 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

- **Transaction States and Operations**

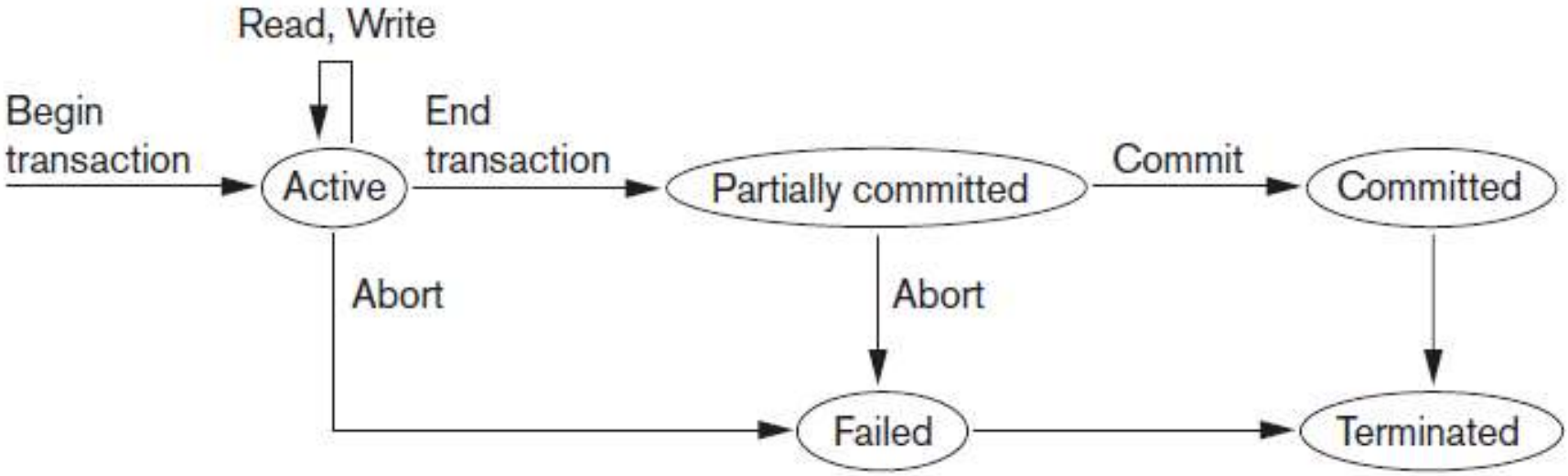


Figure 20.4
State transition diagram illustrating the states for transaction execution.

Transaction and System Concepts

- **Transaction States and Operations**
- A state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.
- When the transaction ends, it moves to the **partially committed state**. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log).
- If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**.. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

Transaction and System Concepts

- **Transaction States and Operations**
- However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.
- The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions

Transaction and System Concepts

- **The System Log**
- To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures.

Transaction and System Concepts

- **The System Log**
- The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries, *T* refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:
  1. **[start_transaction, *T*].** Indicates that transaction *T* has started execution.
  2. **[write_item, *T*, *X*, *old_value*, *new_value*].** Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value*.
  3. **[read_item, *T*, *X*].** Indicates that transaction *T* has read the value of database item *X*.
  4. **[commit, *T*].** Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
  5. **[abort, *T*].** Indicates that transaction *T* has been aborted.

Transaction and System Concepts

- **Commit Point of a Transaction**
- 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 have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database. The transaction then writes a commit record [commit, *T*] into the log.
- If a system failure occurs, we can search back in the log for all transactions *T* that have written a [start_transaction, *T*] record into the log but have not written their [commit, *T*] record yet; these transactions may have to be *rolled back* to *undo their effect* on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

Desirable Properties of Transactions: ACID properties

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

Desirable Properties of Transactions: ACID properties

- **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.
- The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

Desirable Properties of Transactions: ACID properties

- **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.
- The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.
- **Levels of Isolation.** There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads.

Desirable Properties of Transactions: ACID properties

- **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. The *durability property* is the responsibility of the *recovery subsystem* of the DBMS.

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:** When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from all the various transactions is known as a transaction schedule (or history).

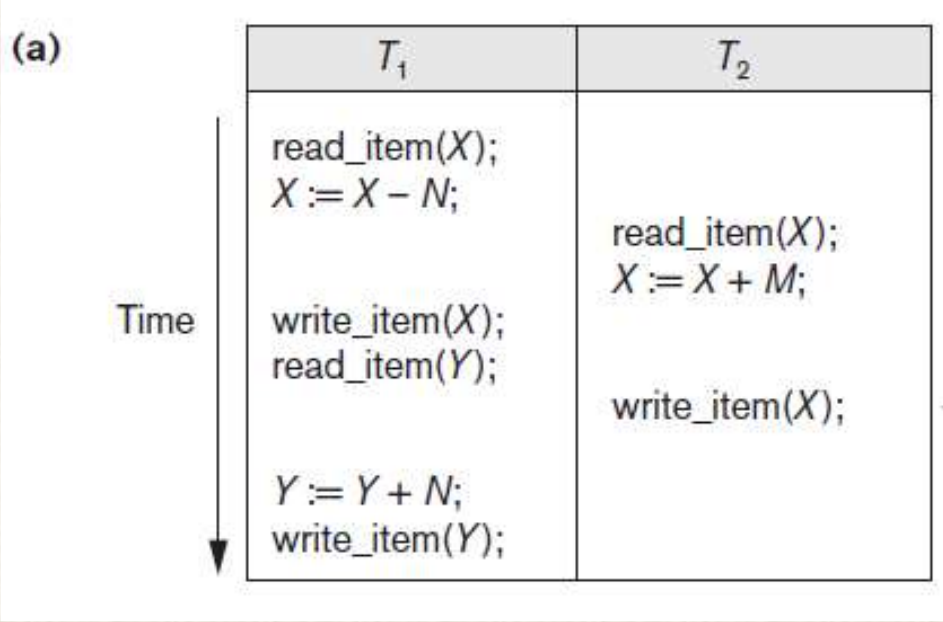
- A **schedule** (or **history**) *S* of *n* transactions *T*1, *T*2, … , *Tn* is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule *S*. However, for each transaction *Ti* that participates in the schedule *S*, the operations of *Ti* in *S* must appear in the same order in which they occur in *Ti*. 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. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.

Characterizing Schedules Based on Recoverability

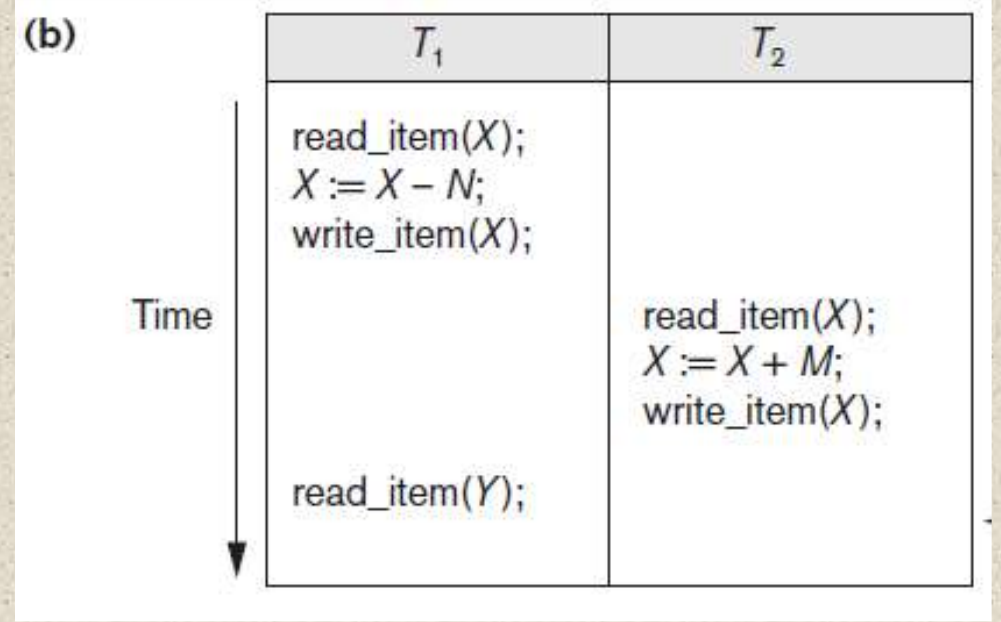
- **Schedules (Histories) of Transactions:**
- A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, *e*, *c*, and *a* for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule. In this notation, the database item *X* that is read or written follows the *r* and *w* operations in parentheses.

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:**



***Sa*: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*);**



***Sb*: *r*1(*X*); *w*1(*X*); *r*2(*X*); *w*2(*X*); *r*1(*Y*); *a*1; [Here it is assumed T1 is aborted after read_item(Y)]**

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:**
- **Conflicting Operations in a Schedule.** 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*; and
  - *at least one* of the operations is a write_item(*X*).
- For example, in schedule *Sa*, the operations *r*1(*X*) and *w*2(*X*) conflict, as do the operations *r*2(*X*) and *w*1(*X*), and the operations *w*1(*X*) and *w*2(*X*). However, the operations *r*1(*X*) and *r*2(*X*) do not conflict, since they are both read operations; the operations *w*2(*X*) and *w*1(*Y*) do not conflict because they operate on distinct data items *X* and *Y*; and the operations *r*1(*X*) and *w*1(*X*) do not conflict because they belong to the same transaction.

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:**
- 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 $r1(X)$; $w2(X)$ to $w2(X)$; $r1(X)$, then the value of X that is read by transaction $T1$ changes, because in the second ordering the value of X is read by $r1(X)$ *after* it is changed by $w2(X)$, 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 $w1(X)$; $w2(X)$ to $w2(X)$; $w1(X)$. For a write-write conflict, the *last value* of X will differ because in one case it is written by $T2$ and in the other case by $T1$. Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:**
- A schedule *S* of *n* transactions *T*1, *T*2, … , *Tn* is said to be a **complete schedule** if the following conditions hold:
  1. The operations in *S* are exactly those operations in *T*1, *T*2, … , *Tn*, including a commit or abort operation as the last operation for each transaction in the schedule.
  2. For any pair of operations from the same transaction *Ti*, their relative order of appearance in *S* is the same as their order of appearance in *Ti*.
  3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules Based on Recoverability

- **Schedules (Histories) of Transactions:**
- In general, it is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection** *C*(*S*) of a schedule *S*, which includes only the operations in *S* that belong to committed transactions—that is, transactions *Ti* whose commit operation *ci* is in *S*.

Characterizing Schedules Based on Recoverability

- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved.

- In some cases, it is even not possible to recover correctly after a failure. Hence, 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.

Characterizing Schedules Based on Recoverability

- First, we would like to ensure that, 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 **nonrecoverable** and hence should not be permitted by the DBMS. The condition for a **recoverable schedule** is that 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.
- A transaction *T* **reads** from transaction *T′* in a schedule *S* if some item *X* is first written by *T′* and later read by *T*. In addition, *T′* should not have been aborted before *T* reads item *X*, and there should be no transactions that write *X* after *T′* writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *X*).

Characterizing Schedules Based on Recoverability

- Consider the schedule *Sa′* given below:

  Sa′: *r*1(*X*); *r*2(*X*); *w*1(*X*); *r*1(*Y*); *w*2(*X*); *c*2; *w*1(*Y*); *c*1;

- *Sa′* is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

- However, consider the two (partial) schedules *Sc* and *Sd* that follow:

  Sc: *r*1(*X*); *w*1(*X*); *r*2(*X*); *r*1(*Y*); *w*2(*X*); *c*2; *a*1;

  Sd: *r*1(*X*); *w*1(*X*); *r*2(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*); *c*1; *c*2;

  Se: *r*1(*X*); *w*1(*X*); *r*2(*X*); *r*1(*Y*); *w*2(*X*); *w*1(*Y*); *a*1; *a*2;

- *Sc* 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 *Sc*; 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*. For the schedule to be recoverable, the *c*2 operation in *Sc* must be postponed until after *T*1 commits, as shown in *Sd*. If *T*1 aborts instead of committing, then *T*2 should also abort as shown in *Se*, because the value of *X* it read is no longer valid. In *Se*, aborting *T*2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule *Sc*.

Characterizing Schedules Based on Recoverability

- In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated. However, 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.
- Because cascading rollback can be time-consuming—since numerous transactions can be rolled back—it is important to characterize the schedules where this phenomenon is guaranteed not to occur. 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. To satisfy this criterion, the *r*2(*X*) command in schedules *Sd* and *Se* must be postponed until after *T*1 has committed (or aborted), thus delaying *T*2 but ensuring no cascading rollback if *T*1 aborts.

Characterizing Schedules Based on Recoverability

- 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. In a strict schedule, the process of undoing a write_item(*X*) operation of an aborted transaction is simply to restore the **before image** (old_value or BFIM) of data item *X*. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule *Sf* :

  Sf : *w*1(*X*, 5); *w*2(*X*, 8); *a*1;

- Suppose that the value of *X* was originally 9, which is the before image stored in the system log along with the *w*1(*X*, 5) operation. If *T*1 aborts, as in *Sf*, the recovery procedure that restores the before image of an aborted write operation will restore the value of *X* to 9, even though it has already been changed to 8 by transaction *T*2, thus leading to potentially incorrect results. Although schedule *Sf* is cascadeless, it is not a strict schedule, since it permits *T*2 to write item *X* even though the transaction *T*1 that last wrote *X* had not yet committed (or aborted). A strict schedule does not have this problem.

Characterizing Schedules Based on Recoverability

- It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose we have *i* transactions *T*1, *T*2, … , *Ti*, and their number of operations are *n*1, *n*2, … , *ni*, respectively. If we make a set of *all possible schedules* of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

Characterizing Schedules Based on Serializability

- The schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as ***serializable schedules***.
- Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions *T*1 and *T*2, in figure shown below, at approximately the same time. 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).

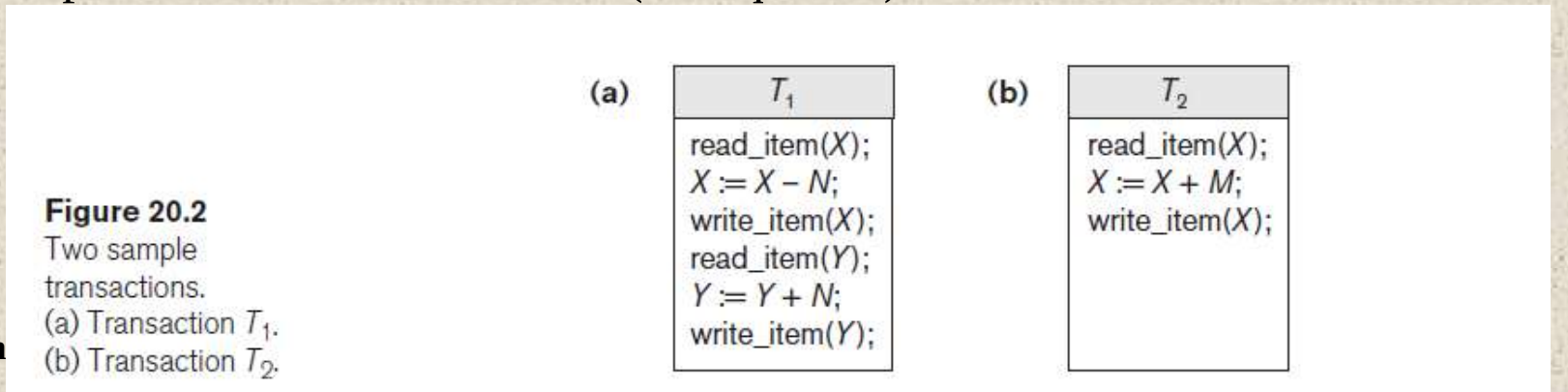


Figure 20.2
Two sample transactions.
(a) Transaction T_1.
(b) Transaction T_2.

Characterizing Schedules Based on Serializability

- Considering the transactions from figure 20.2, in previous slide.
- The two schedules—called *serial schedules*—are shown in figures in next slide Figure 20.5(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure 20.5(c). The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules.

Characterizing Schedules Based on Serializability

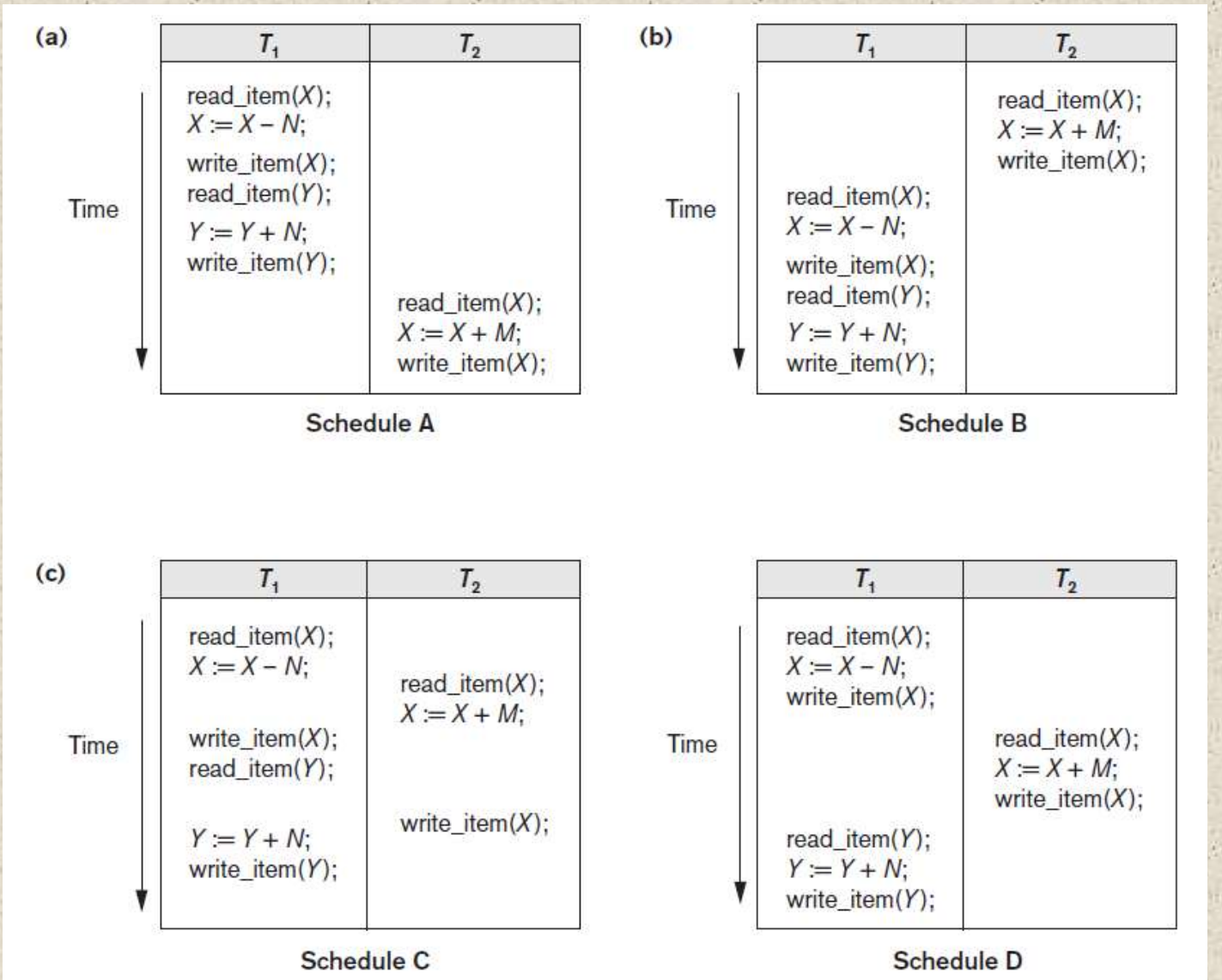


Figure 20.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- 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. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*. We can assume this because every transaction is assumed to be correct if executed on its own (according to the *consistency preservation* property). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result.
- Schedules A and B in the figure in previous slide, Figures 20.5(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 the figure 20.5(a), and *T*2 and then *T*1 in Figure 20.5(b). Schedules C and D in figure 20.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O\ operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction *T* is long, the other transactions must wait for *T* to complete all its operations before starting. Hence, serial schedules are *unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- A schedule *S* of *n* transactions is **serializable** if it is *equivalent to some serial schedule* of the same *n* transactions. We will define the concept of *equivalence of schedules* shortly. Notice that there are *n*! possible serial schedules of *n* transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules— those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to *any* serial schedule and hence are not serializable.
- 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.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- The simplest but least satisfactory definition involves comparing the effects of the schedules on the database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state.
- Hence, result equivalence alone cannot be used to define equivalence of schedules. The safest and most general approach to defining schedule equivalence is to focus only on the read_item and write_item operations of the transactions, and not make any assumptions about the other internal operations included in the 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*.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- **Conflict Equivalence of Two Schedules.** Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both schedules.
- Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item. 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.
- For example, if a read and write operation occur in the order $r1(X)$, $w2(X)$ in schedule $S1$, and in the reverse order $w2(X)$, $r1(X)$ in schedule $S2$, the value read by $r1(X)$ can be different in the two schedules. Similarly, if two write operations occur in the order $w1(X)$, $w2(X)$ in $S1$, and in the reverse order $w2(X)$, $w1(X)$ in $S2$, the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- **Serializable Schedules.** 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′*.

Characterizing Schedules Based on Serializability

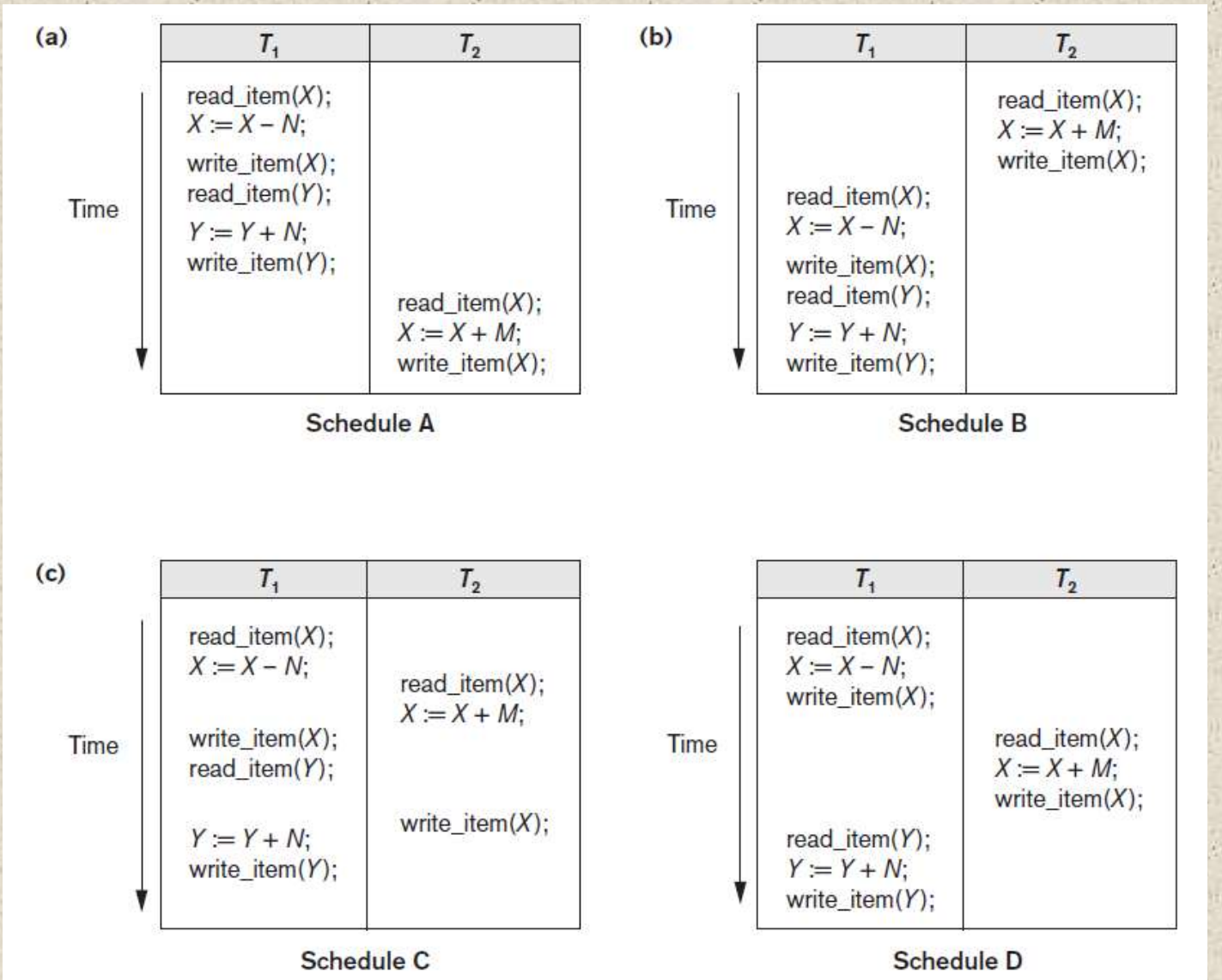


Figure 20.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

Characterizing Schedules Based on Serializability

- **Serial, Nonserial, and Conflict-Serializable Schedules**
- Schedule D in Figure 20.5(c), in previous slide, is equivalent to the serial schedule A in Figure 20.5(a). In both schedules, the read_item(X) of $T2$ reads the value of X written by $T1$, whereas the other read_item operations read the database values from the initial database state. Additionally, $T1$ is the last transaction to write Y, and $T2$ is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations $r1(Y)$ and $w1(Y)$ of schedule D do not conflict with the operations $r2(X)$ and $w2(X)$, since they access different data items. Therefore, we can move $r1(Y)$, $w1(Y)$ before $r2(X)$, $w2(X)$, leading to the equivalent serial schedule $T1$, $T2$.
- Schedule C in Figure 20.5(c) 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 $r2(X)$ and $w1(X)$ conflict, which means that we cannot move $r2(X)$ down to get the equivalent serial schedule $T1$, $T2$. Similarly, because $w1(X)$ and $w2(X)$ conflict, we cannot move $w1(X)$ down to get the equivalent serial schedule $T2$, $T1$.

Characterizing Schedules Based on Serializability

- **Testing for Serializability of a Schedule**
- The algorithm for testing seriablizability 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 $N = \{T1, T2, \ldots, Tn\}$ and a set of directed edges $E = \{e1, e2, \ldots, em\}$.
- There is one node in the graph for each transaction *Ti* in the schedule. Each edge *ei* in the graph is of the form $(Tj \rightarrow Tk)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where *Tj* is the **starting node** of *ei* and *Tk* is the **ending node** of e*i*. Such an edge from node *Tj* to node *Tk* is created by the algorithm if a pair of conflicting operations exist in *Tj* and *Tk* and the conflicting operation in *Tj* appears in the schedule *before* the *conflicting operation* in *Tk*. *(Refer slide no 42 for conflicting operations)*

Characterizing Schedules Based on Serializability

- **Testing for Serializability of a Schedule**

Algorithm: Testing Conflict Serializability of a Schedule *S*

1. For each transaction *Ti* participating in schedule *S*, create a node labeled *Ti* in the precedence graph.
2. For each case in *S* where *Tj* executes a read_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
3. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a read_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
4. For each case in *S* where *Tj* executes a write_item(*X*) after *Ti* executes a write_item(*X*), create an edge (*Ti* → *Tj*) in the precedence graph.
5. The schedule *S* is serializable if and only if the precedence graph has no cycles

- If there is a cycle in the precedence graph, schedule *S* is not (conflict) serializable; if there is no cycle, *S* is (conflict) serializable. A **cycle** in a directed graph is a **sequence of edges** *C* = ((*Tj* → *Tk*), (*Tk* → *Tp*), … , (*Ti* → *Tj*)) 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).

Characterizing Schedules Based on Serializability

- In the precedence graph, an edge from *Ti* to *Tj* means that transaction *Ti* must come before transaction *Tj* 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 *Ti* to *Tj*, *Ti* must appear before *Tj* in the equivalent serial schedule *S′*.
- In general, several serial schedules can be equivalent to *S* if the precedence graph for *S* has no cycle. However, 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

Characterizing Schedules Based on Serializability

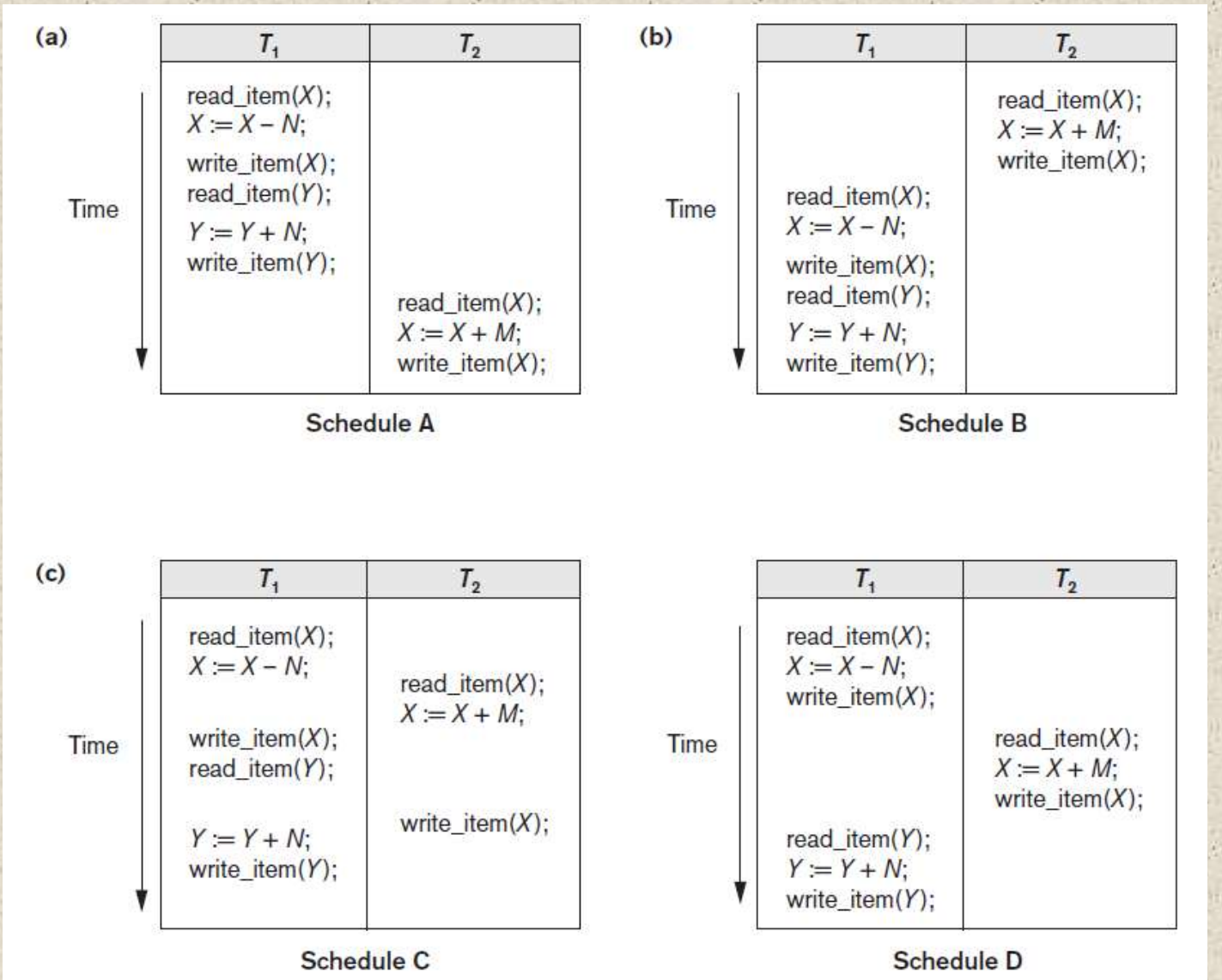


Figure 20.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

Characterizing Schedules Based on Serializability

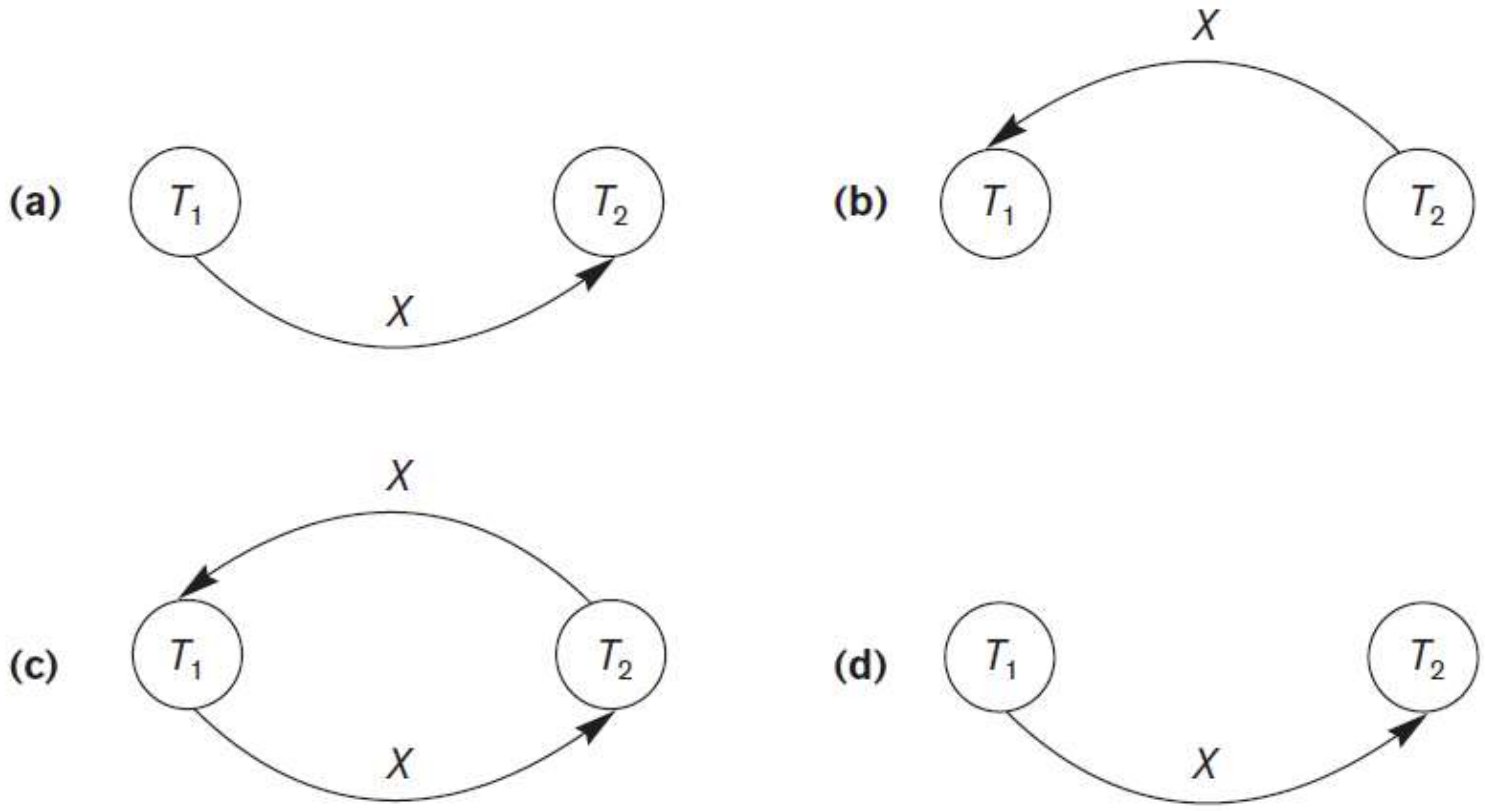


Figure 20.7
Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Characterizing Schedules Based on Serializability

- The graph for schedule C has a cycle, so it is not serializable.
- The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is *T*1 followed by *T*2.
- The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

How Serializability Is Used for Concurrency Control

- Saying that a schedule *S* is (conflict) serializable—that is, *S* is (conflict) equivalent to a serial schedule—is tantamount to saying that *S* is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for a long transaction to delay other transactions, thus slowing down transaction processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness.
- In practice, it is difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to all processes.

How Serializability Is Used for Concurrency Control

- Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule.
- Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability. If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of *all schedules in which the transactions participate*.

How Serializability Is Used for Concurrency Control

- Another problem is that transactions are submitted continuously to the system, so it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule *S*. The *committed projection* *C*(*S*) of a schedule *S* includes only the operations in *S* that belong to committed transactions. We can theoretically define a schedule *S* to be erializable if its committed projection *C*(*S*) is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.