

Introduction to Transaction Processing Concepts and Theory

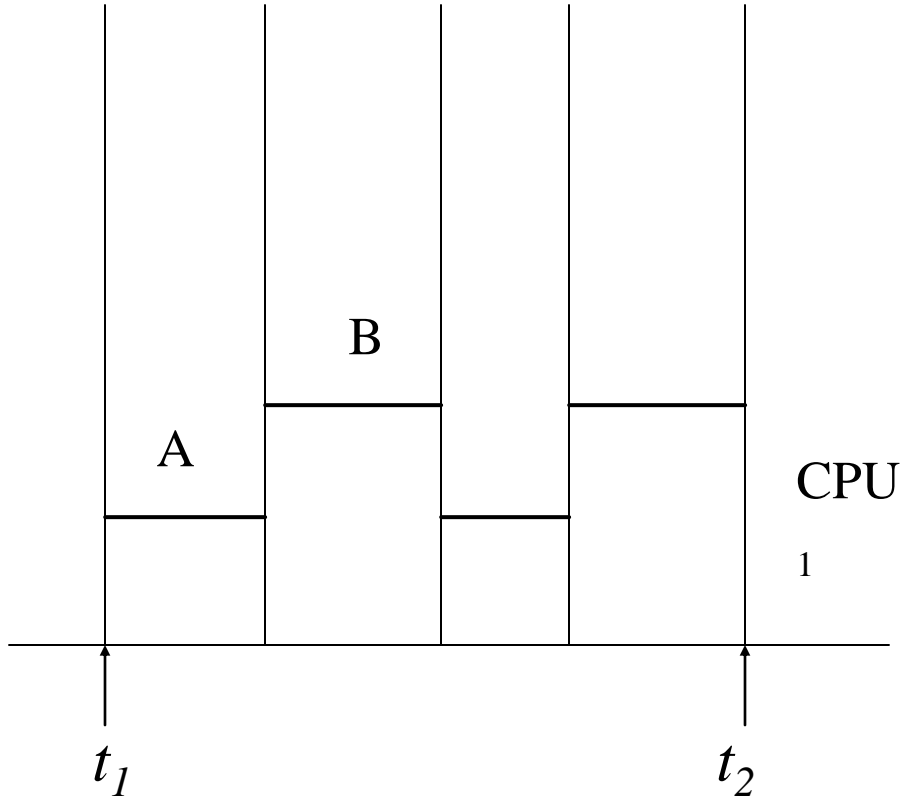
Summary

- 1 Introduction to Transaction Processing
- 2 Transaction and System Concepts
- 3 Transaction Schedule
- 4 Serializability
- 5 Transaction Support in SQL

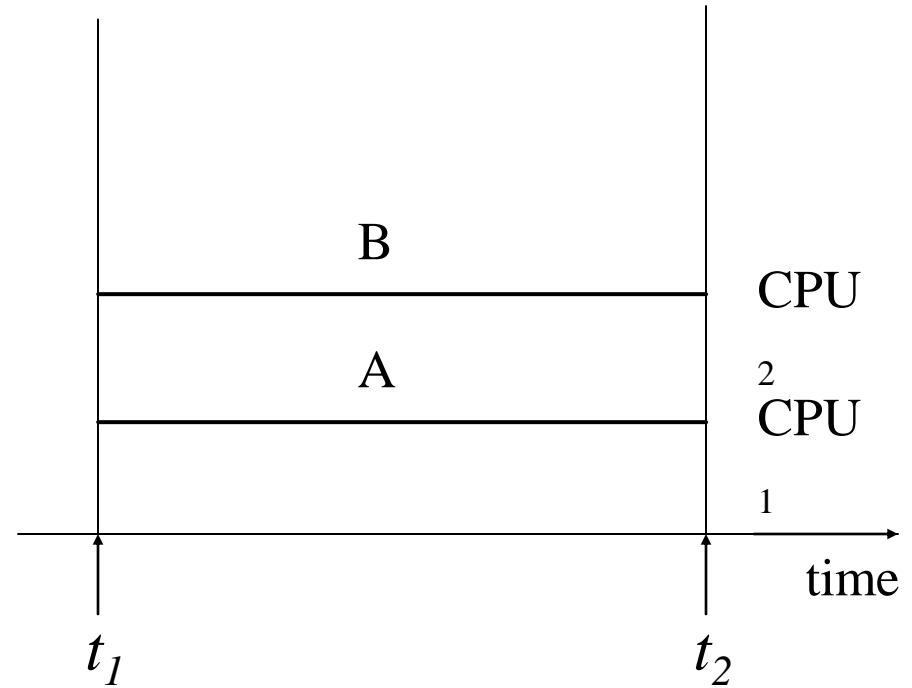
Introduction to Transaction Processing

- **Single-User System:**
 - At most one user at a time can use the system.
- **Multuser 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.

Concurrent Transactions



interleaved processing



parallel processing

What is Transaction?

- **A Transaction:**
 - Logical unit of database processing that 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, 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.

Why Do We Need Transactions?

- It's all about fast query response time and correctness
- DBMS is a multi-user systems
 - Many different requests
 - Some against same data items
- Figure out how to interleave requests to shorten response time while guaranteeing correct result
 - How does DBMS know which actions belong together?
- Solution: Group database operations that must be performed together into transactions
 - Either execute all operations or none

Simple Model of a Database

- 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 & 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.
- **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).

Sample Transaction (informal)

- Example: Move \$40 from checking to savings account
- To user, appears as one activity
- To database:
 - Read balance of checking account: read(X)
 - Read balance of savings account: read (Y)
 - Subtract \$40 from X
 - Add \$40 to Y
 - Write new value of X back to disk
 - Write new value of Y back to disk

Sample Transaction (Formal)

T1

t_0

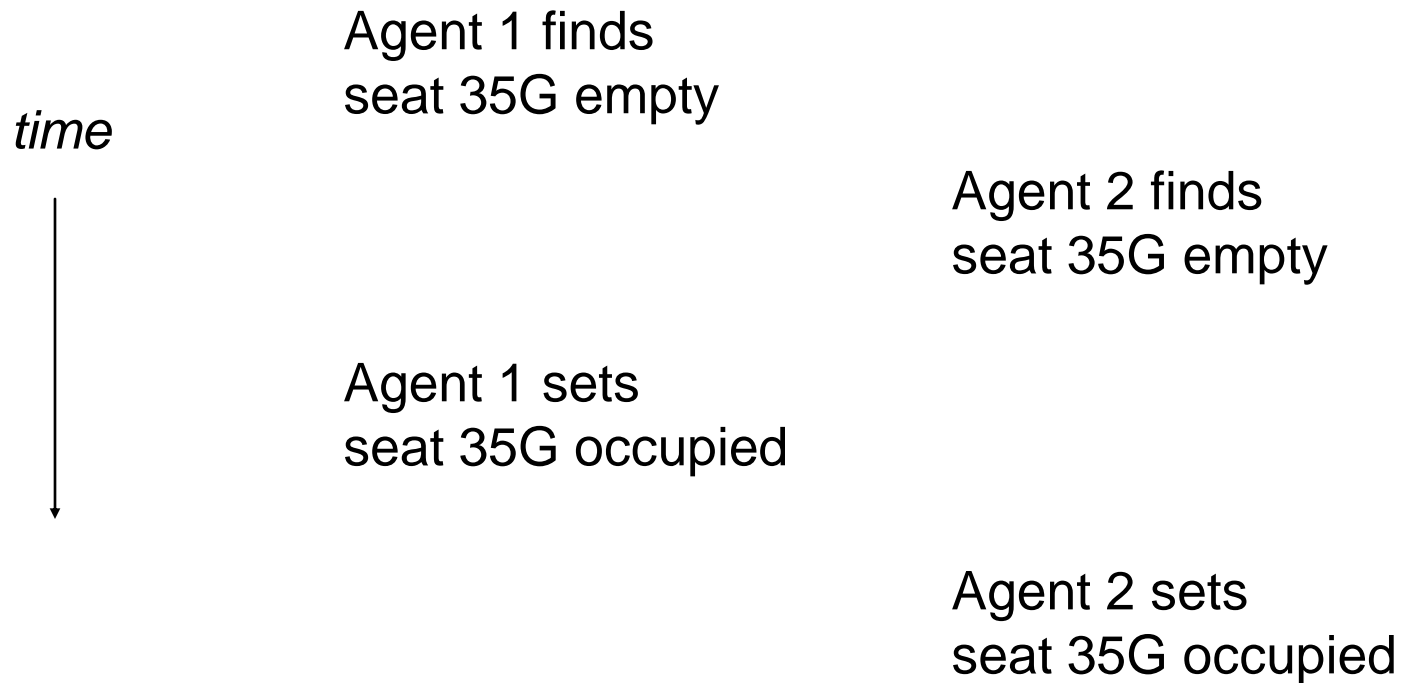
↓

t_k

```
read_item(X);  
read_item(Y);  
X:=X-40;  
Y:=Y+40;  
write _item(X);  
write_item(Y);
```

Another Sample Transaction

- Reserving a seat for a flight
- If concurrent access to data in DBMS, two users may try to book the same seat simultaneously



SQL Transaction Example

- Register credit sale of 100 units of product X to customer Y for \$500

```
UPDATE PRODUCT
SET PROD_QOH = PROD_QOH - 100
WHERE PROD_CODE = 'X';
UPDATE ACCT_RECEIVABLE
SET ACCT_BALANCE = ACCT_BALANCE + 500
WHERE ACCT_NUM = 'Y';
COMMIT;
```

- Consistent state only if both transactions are fully completed
- DBMS doesn't guarantee transaction represents real-world event
- Transaction begins when first SQL statement is encountered, and ends at COMMIT or end

Two sample transactions

(a) T_1

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

(b) T_2

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

Why Concurrency Control is needed

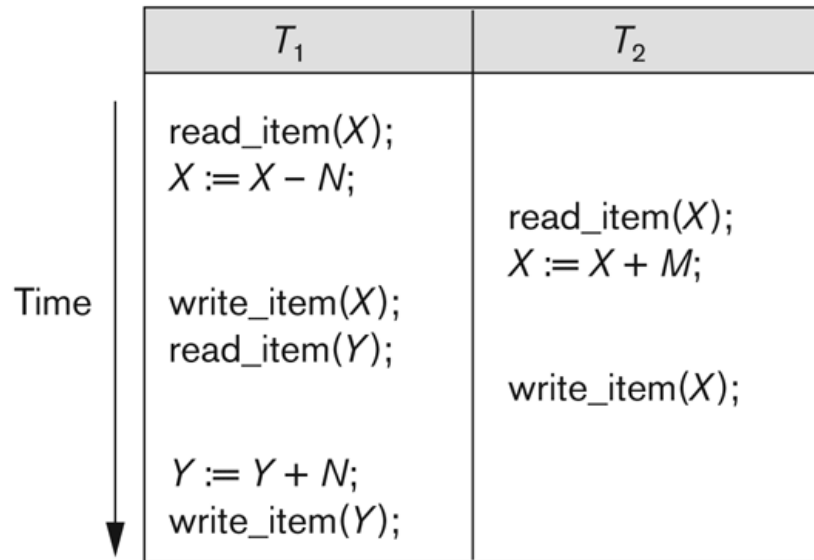
- **The Lost Update Problem**
 - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
 - This occurs when one transaction updates a database item and then the transaction fails for some reason.
 - The updated item is accessed by another transaction before it is changed back 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.

(a) The lost update problem.

Figure 17.3

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

(a)

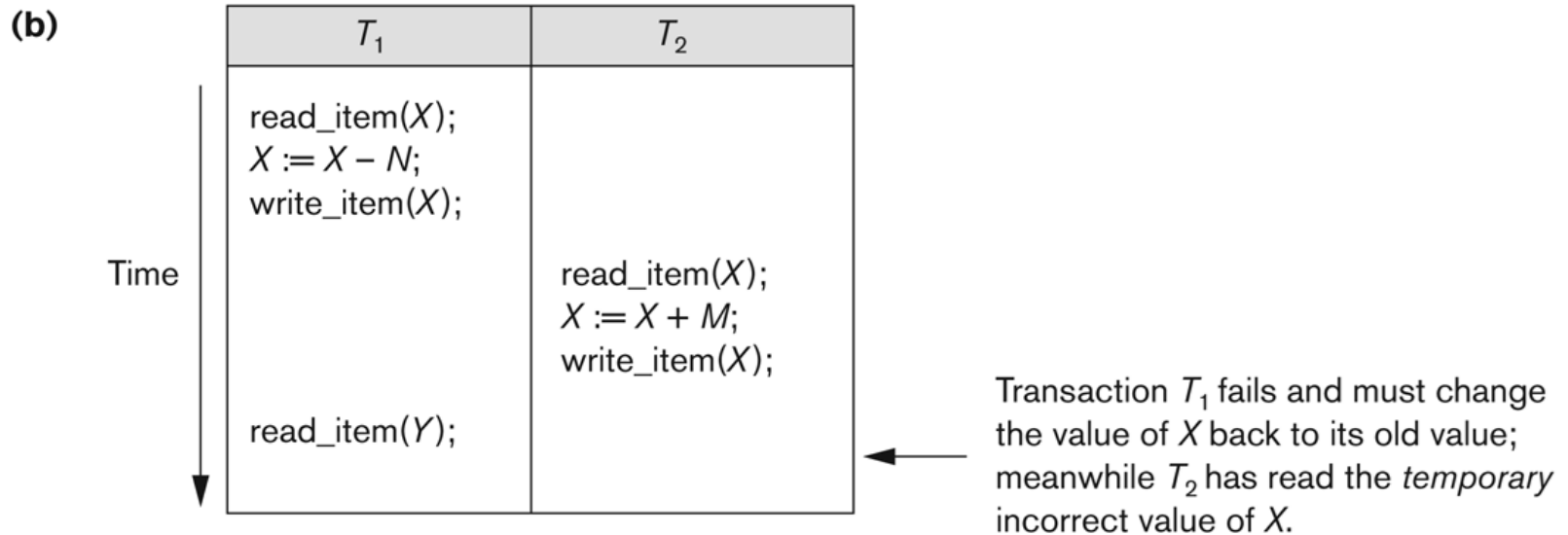


← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

(b) The temporary update problem.

Figure 17.3

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



(c) The incorrect summary problem.

Figure 17.3

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

(c)

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

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

What Can Go Wrong?

- System may crash before data is written back to disk
- Some other transaction is modifying shared data while our transaction is ongoing (or vice versa)
- System may not be able to obtain one or more of the data items
- System may not be able to write one or more of the data items
- DBMS has a Concurrency Control subsystem to assure database remains in consistent state despite concurrent execution of transactions

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. 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.

3. 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.

What causes a Transaction to fail?

4. 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.

5. 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.

6. 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.

2 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

State transition diagram illustrating the states for transaction execution

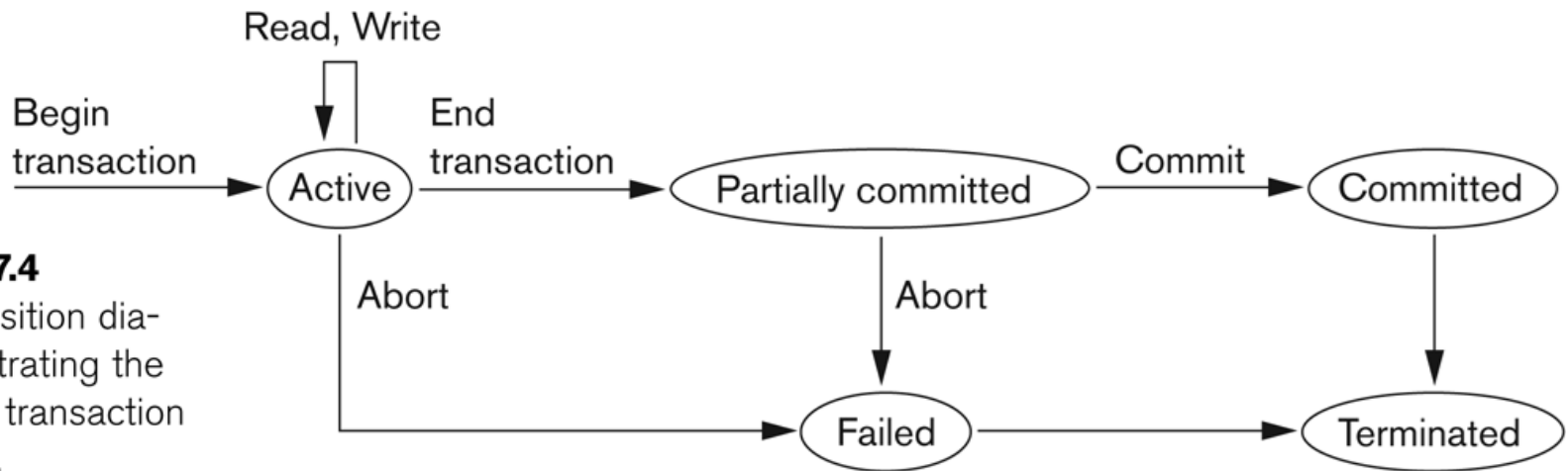


Figure 17.4

State transition diagram illustrating the states for transaction execution.

Transaction Events

- **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.
- **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.

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.)

- 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:
- **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 Log Example

For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.

Back P and Next P point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

How is the Log File Used?

- All permanent changes to data are recorded
 - Possible to undo changes to data
- After crash, search log backwards until find last commit point
 - Know that beyond this point, effects of transaction are permanently recorded
- Need to either *redo* or *undo* everything that happened since last commit point
 - Undo: When transaction only partially completed (before crash)
 - Redo: Transaction completed but we are unsure whether data was written to disk

ACID Properties of Transactions

- *Atomicity*: Transaction is either performed in its entirety or not performed at all
- *Consistency preservation*: Transaction must take the database from one consistent state to another
- *Isolation*: Transaction should appear as though it is being executed in isolation from other transaction
- *Durability*: Changes applied to the database by a committed transaction must persist

Ensuring Atomicity

- Transactions can be incomplete due to several reasons
 - Aborted (terminated) by the DBMS because of some anomalies during execution
 - in that case automatically restarted and executed anew
 - The system may crash (say no power supply)
 - A transaction may decide to abort itself encountering an unexpected situation
 - e.g. read an unexpected data value or unable to access disks

Ensuring Atomicity

- A transaction interrupted in the middle can leave the database in an inconsistent state
- DBMS has to remove the effects of partial transactions from the database
- DBMS ensures atomicity by “undoing” the actions of incomplete transactions
- DBMS maintains a “log” of all changes to do so

Ensuring Consistency

- e.g. Money debit and credit between accounts
- User's responsibility to maintain the integrity constraints
- DBMS may not be able to catch such errors in user program's logic
- However, the DBMS may be in inconsistent state "during a transaction" between actions
 - which is ok, but it should leave the database at a consistent state when it commits or aborts
- **Database consistency** follows from transaction consistency, isolation, and atomicity

Ensuring Isolation

- DBMS guarantees isolation (later, how)
- If T1 and T2 are executed concurrently, either the effect would be T1->T2 or T2->T1 (and from a consistent state to a consistent state)
- But DBMS provides no guarantee on which of these order is chosen
- Often ensured by “locks” but there are other methods too

Ensuring Durability

- The **log** also ensures durability
- If the system crashes before the changes made by a completed transactions are written to the disk, the log is used to remember and restore these changes when the system restarts
- “**recovery manager**” will be discussed later
 - takes care of atomicity and durability

3 Transaction Schedule

- Recall: Multiple transactions can be executed concurrently by interleaving their operations
- Ordering of execution of operations from various transactions T_1, T_2, \dots, T_n is called a schedule S
- Constraint: For each transaction T_i , the order in which operations occur in S must be the same as in T_i
- Only interested in read (r), write (w), commit (c), abort (a)

Sample Schedule

- | T1: $r(X)$; $w(X)$; $r(Y)$; $w(Y)$; c

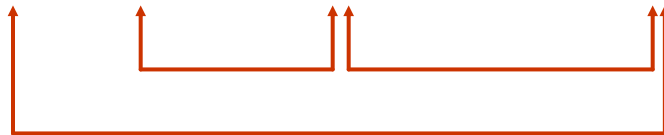
- | T2: $r(X)$; $w(X)$; c

- | Sample schedule:

 - S: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1 ; c_2

Conflicts

- Two operations *conflict* if they satisfy ALL three conditions:
 - they belong to different transactions **AND**
 - they access the same item **AND**
 - at least one is a `write_item()` operation
- Ex.:
 - S: $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$



conflicts

Why Do We Interleave Transactions?

Schedule S

T1

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

T2

```
read_item(X):  
X:=X+M;  
write_item(X);
```

Could be a long wait

S is a serial schedule – no interleaving!

Serial Schedule

- If we consider transactions to be independent, serial schedule is correct
 - Based on C property in ACID above assumption is valid
- Furthermore, it does not matter which transaction is executed first, as long as every transaction is executed in its entirety, from beginning to end
- Assume $X=90$, $Y=90$, $N=3$, $M=2$, then result of schedule S is $X=89$ and $Y=93$
 - Same result if we start with T2

Better?

Schedule S'

T1

read_item(X);

X:=X-N;

write_item(X);

read_item(Y);

Y:=Y+N;

write_item(Y);

T2

read_item(X);

X:=X+M;

write_item(X);

S' is a non-serial schedule

T2 will be done faster but is the result correct?

Concurrent Executions

- Serial execution is by far simplest method to execute transactions
 - No extra work ensuring consistency
- Inefficient!
- Reasons for concurrency:
 - Increased throughput
 - Reduces average response time
- Need concept of *correct* concurrent execution
 - Using same X, Y, N, M values as before, result of S is X=92 and Y=93 (not correct)

Better?

Schedule S''

T1

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

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

T2

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

S'' is a non-serial schedule

Produces same result as serial schedule S

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

Serializability Definitions

- 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 Equivalent Schedules

- Two schedules are result equivalent if they produce the same final state of the database
- Problem: May produce same result by accident!

S1	S2
<hr/> read_item(X); X:=X+10; write_item(X);	<hr/> read_item(X); X:=X*1.1; write_item(X);

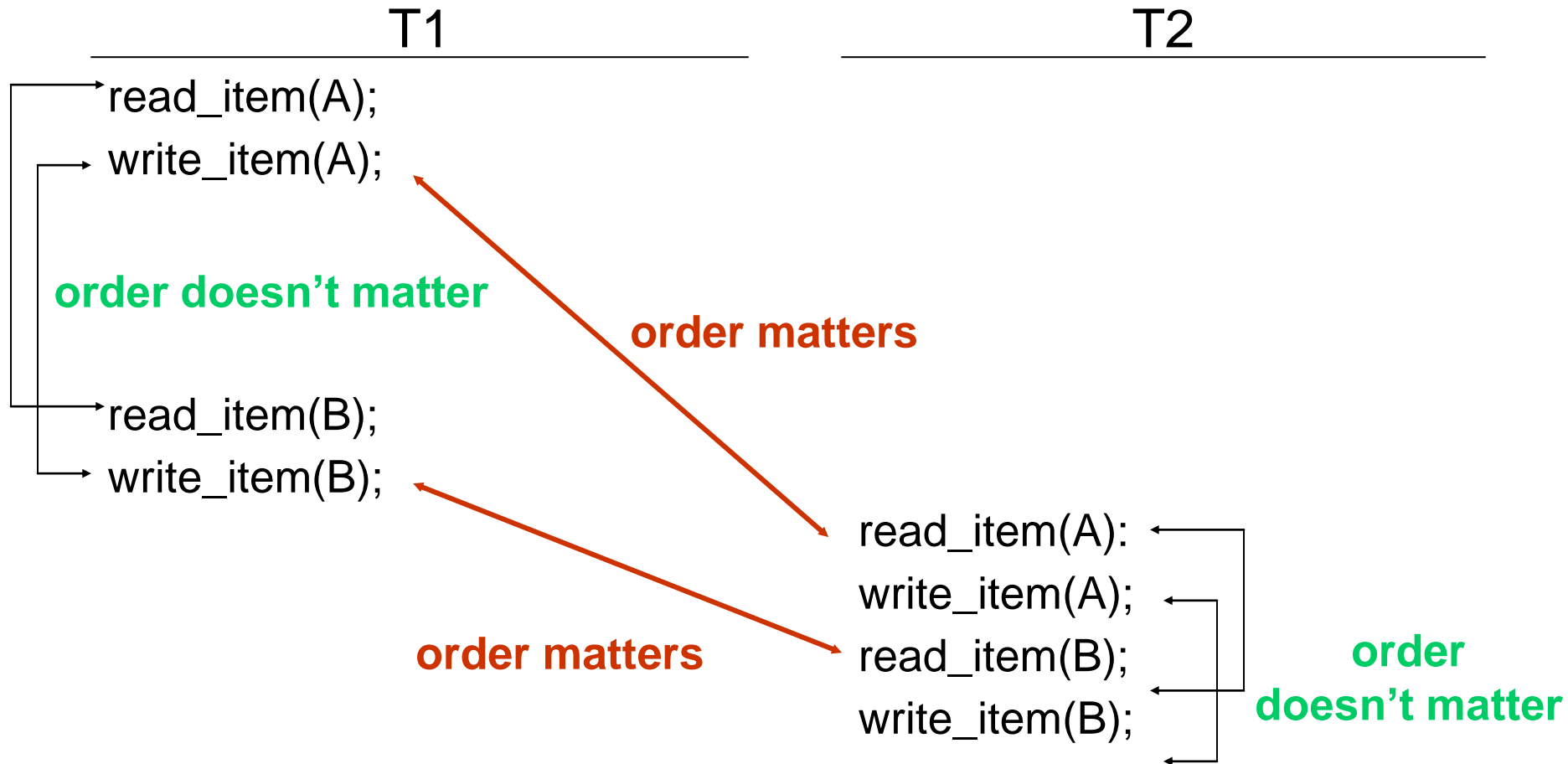
Schedules S1 and S2 are result equivalent for $X=100$
but not in general

Conflict Equivalent Schedules

- Two schedules are *conflict equivalent*, if the order of any two *conflicting operations* is the same in both schedules

Conflict Equivalence

Serial Schedule S1



Conflict Equivalence

Schedule S1'

T1

T2

read_item(A);
read_item(B);
write_item(A);

write_item(B);

same order as in S1

same order as in S1

read_item(A):

write_item(A);

read_item(B);

write_item(B);

S1 and S1' are conflict equivalent
(S1' produces the same result as S1)

Conflict Equivalence

Schedule S1''

T1

T2

read_item(A);
write_item(A);

read_item(A);
write_item(A);

different order as in S1

read_item(B);
write_item(B);

read_item(B);
write_item(B);

different order as in S1

Schedule S1'' is not conflict equivalent to S1
(produces a different result than S1)

Conflict Serializable

- Schedule S is conflict serializable if it is conflict equivalent to some *serial* schedule S'
 - Can reorder the *non-conflicting* operations to improve efficiency
- Non-conflicting operations:
 - Reads and writes from same transaction
 - Reads from different transactions
 - Reads and writes from different transactions on different data items
- Conflicting operations:
 - Reads and writes from different transactions on same data item

Example

Schedule A

T1	T2
read_item(X); X:=X-N; write_item(X); read_item(Y); Y:=Y+N; write_item(Y);	read_item(X); X:=X+M; write_item(X);

Schedule B

T1	T2
read_item(X); X:=X-N; write_item(X); read_item(Y); Y:=Y+N; write_item(Y);	read_item(X); X:=X+M; write_item(X);

**B is conflict equivalent to A
⇒ B is serializable**

Serial Vs Serializable

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

Ensure Serializability

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

Test for Serializability

- Construct a directed graph, *precedence graph*, $G = (V, E)$
 - V : set of all transactions participating in schedule
 - E : set of edges $T_i \rightarrow T_j$ for which one of the following holds:
 - T_i executes a `write_item(X)` before T_j executes `read_item(X)`
 - T_i executes a `read_item(X)` before T_j executes `write_item(X)`
 - T_i executes a `write_item(X)` before T_j executes `write_item(X)`
- An edge $T_i \rightarrow T_j$ means that in any serial schedule equivalent to S , T_i must come before T_j
- If G has a cycle, then S is not conflict serializable
- If not, use topological sort to obtain serializable schedule (linear order consistent with precedence order of graph)

Sample Schedule S

T1

T2

T3

read_item(X);
write_item(X);

read_item(Y);
write_item(Y);

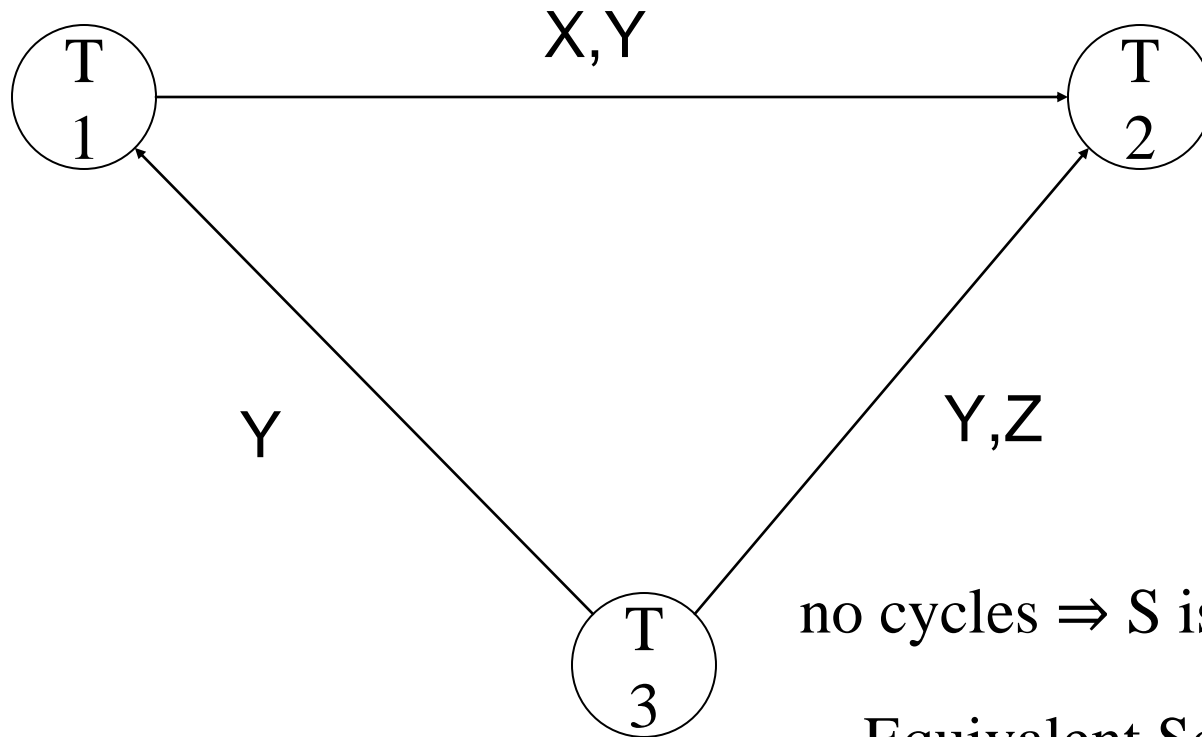
read_item(Z);

read_item(Y);
write_item(Y);
read_item(X);
write_item(X);

read_item(Y);
read_item(Z);

write_item(Y);
write_item(Z);

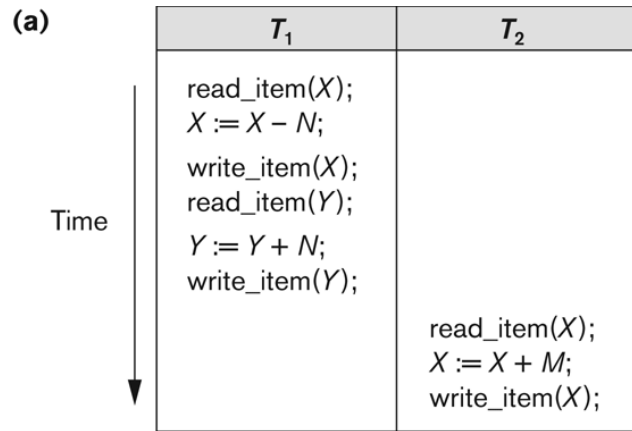
Precedence Graph for S



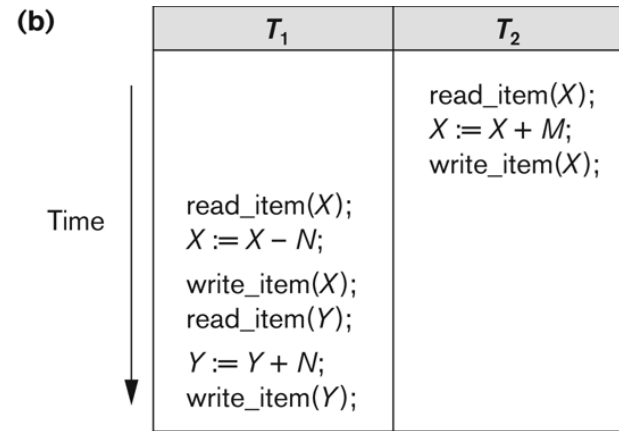
no cycles \Rightarrow S is serializable

Equivalent Serial Schedule:
 $T3 \rightarrow T1 \rightarrow T2$
(precedence order)

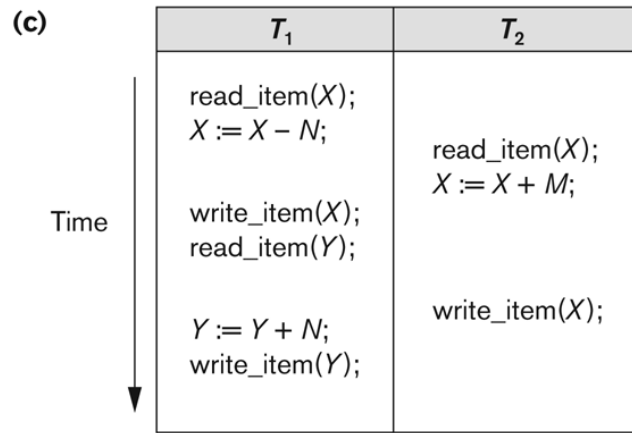
Examples of serial and nonserial schedules



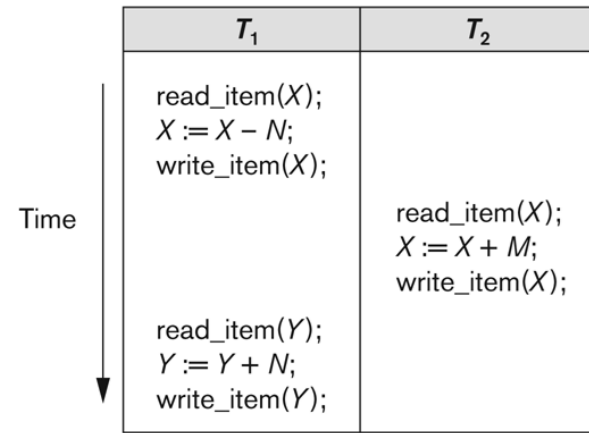
Schedule A



Schedule B



Schedule C

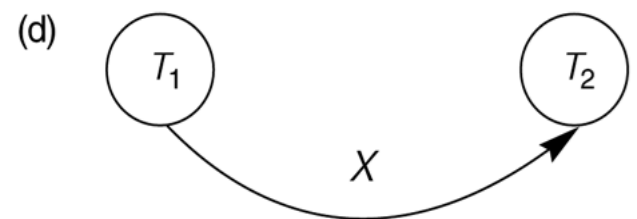
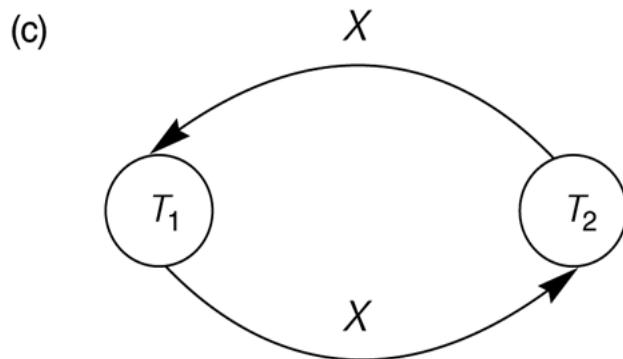
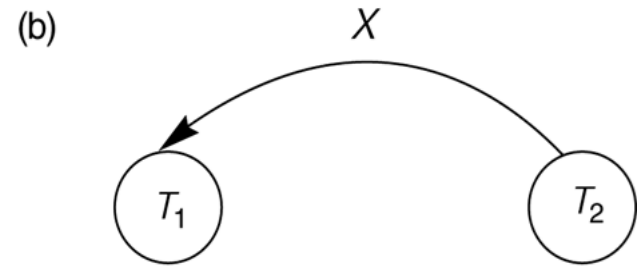
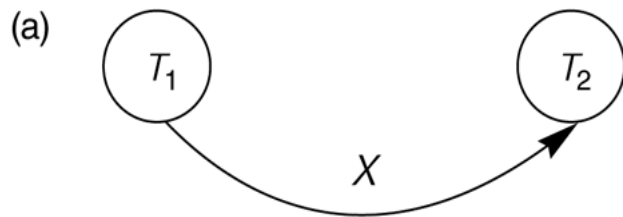


Schedule D

Constructing the Precedence Graphs

FIGURE 17.7 Constructing the precedence graphs for schedules A and D from Figure 17.5 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).



Serializability Testing Example

Figure 17.8

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.

(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);

Serializability Testing Example – Schedule E

Figure 17.8

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.

(b)

Time ↓	Transaction T_1	Transaction T_2	Transaction T_3
	<div>read_item(X); write_item(X);</div> <div>read_item(Y); write_item(Y);</div>	<div>read_item(Z); read_item(Y); write_item(Y);</div> <div>read_item(X);</div> <div>write_item(X);</div>	<div>read_item(Y); read_item(Z);</div> <div>write_item(Y); write_item(Z);</div>

Schedule E


Serializability Testing Example – Schedule F

Figure 17.8

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.

(c)

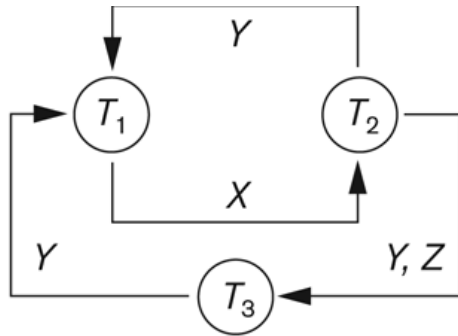
Time



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

Schedule F

(d)



Equivalent serial schedules

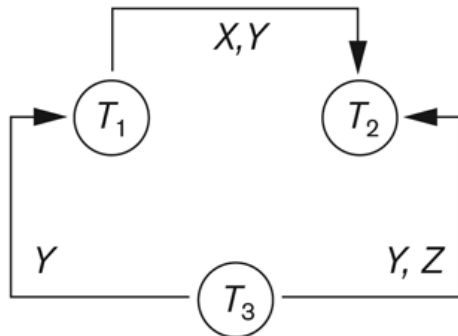
None

Reason

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

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

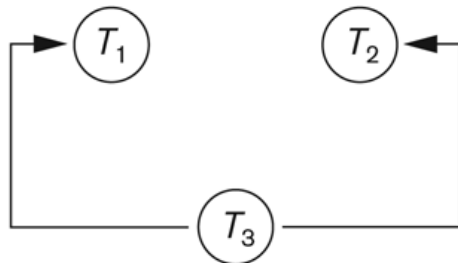
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Figure 17.8 (continued)

Another example of serializability testing. (d) Precedence graph for schedule E . (e) Precedence graph for schedule F . (f) Precedence graph with two equivalent serial schedules.

5 Transaction Support in SQL

- A **single** SQL statement is always considered to be **atomic**.
 - Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement.
 - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

Sample SQL Transaction

```
EXEC SQL whenever sqlerror go to UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTICS SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
    INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
    VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
    SET SALARY = SALARY * 1.1
    WHERE DNO = 2;
EXEC SQL COMMIT;
    GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...
```

Summary

- 1 Introduction to Transaction Processing
- 2 Transaction and System Concepts
- 3 Transaction Schedule
- 4 Serializability
- 5 Transaction Support in SQL