Chapter Outline

- Introduction to Transaction Processing
- Transaction and System Concepts
- Desirable Properties of Transactions
- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability
- Transaction Support in SQL

Transaction Processing System

- TPS are the systems with large databases and hundreds of concurrent users that are executing database transactions.
- Reservations, Banking,credit card processing, stock markets etc..
- Require high availability and fast response.

Introduction to Transaction Processing

Classification of DB system

According to the no. of users who can use the system concurrently…

- **Single-User System:** At most one user at a time can use the system.
- **Multiuser System**: Many users can access the system concurrently.
- Possible because of the concept of **multiprogramming**.
- **Concurrency**
  - **Interleaved processing**: concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**: processes are concurrently executed in multiple CPUs.

Introduction to Transaction Processing

- **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.

- If the db operations in a transaction do not update the db but only retrieve data , the transaction is called the **Read-only transaction**

Introduction to Transaction Processing

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database -** collection of **named data items**
- **Granularity of data(size 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.

Introduction to Transaction Processing

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.

- **Executing 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).
  3. Copy item X from the buffer to the program variable named X.

Introduction to Transaction Processing

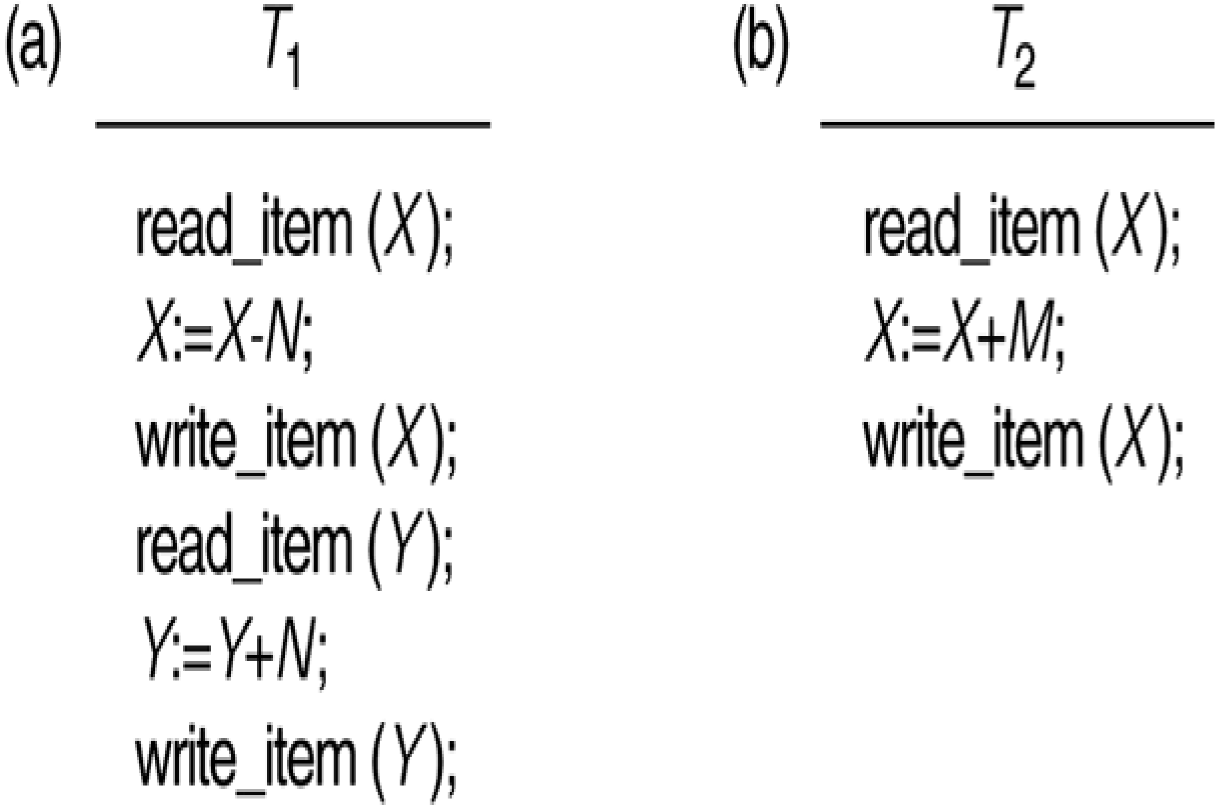
READ AND WRITE OPERATIONS (cont.):

- **Executing 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 block from the buffer back to disk (either immediately or at some later point in time).

FIGURE

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



(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);
```

Introduction to Transaction Processing

Why Concurrency Control is needed:

If the concurrent execution of the transaction is uncontrolled leads to different problems..

- **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.

Introduction to Transaction Processing

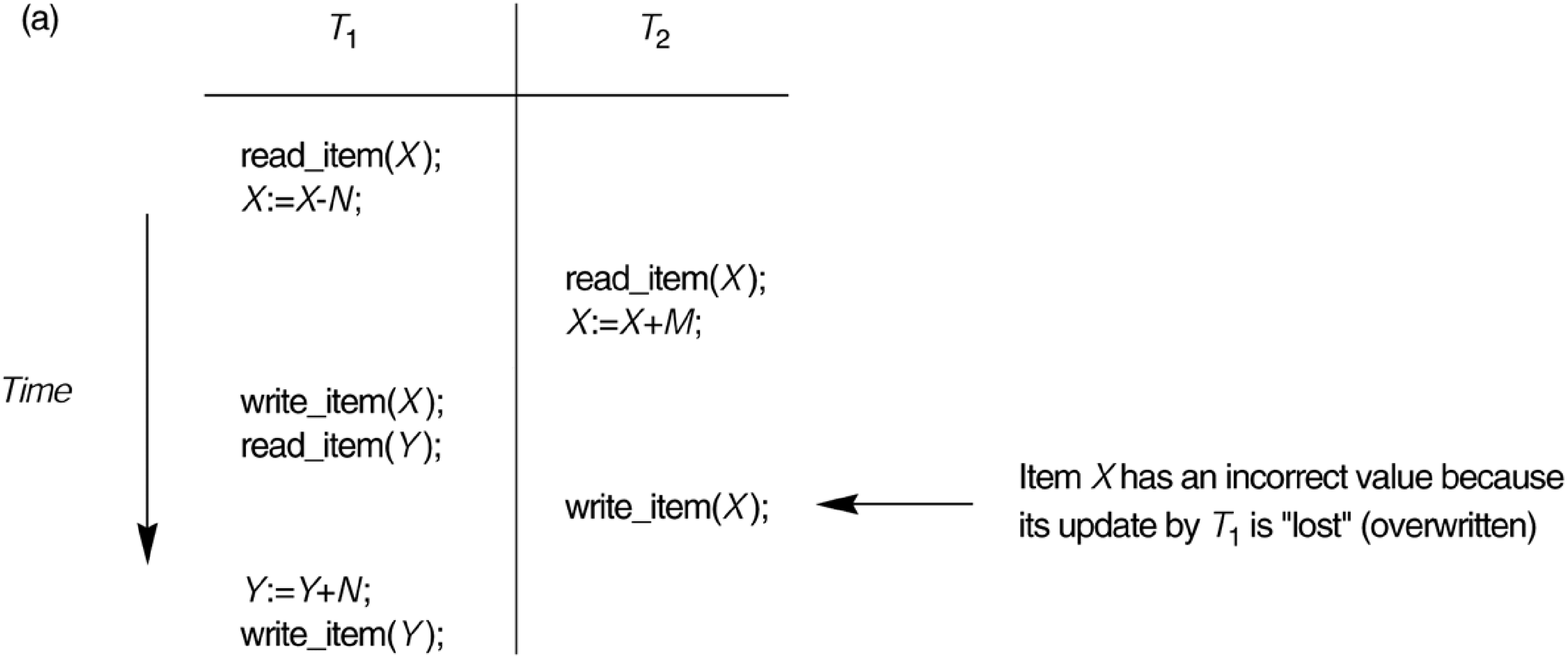
Why Concurrency Control is needed (cont.):

- **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.

FIGURE

Some problems that occur when concurrent execution is uncontrolled. (a) **<u>The lost update problem.</u>**



(a)

T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;

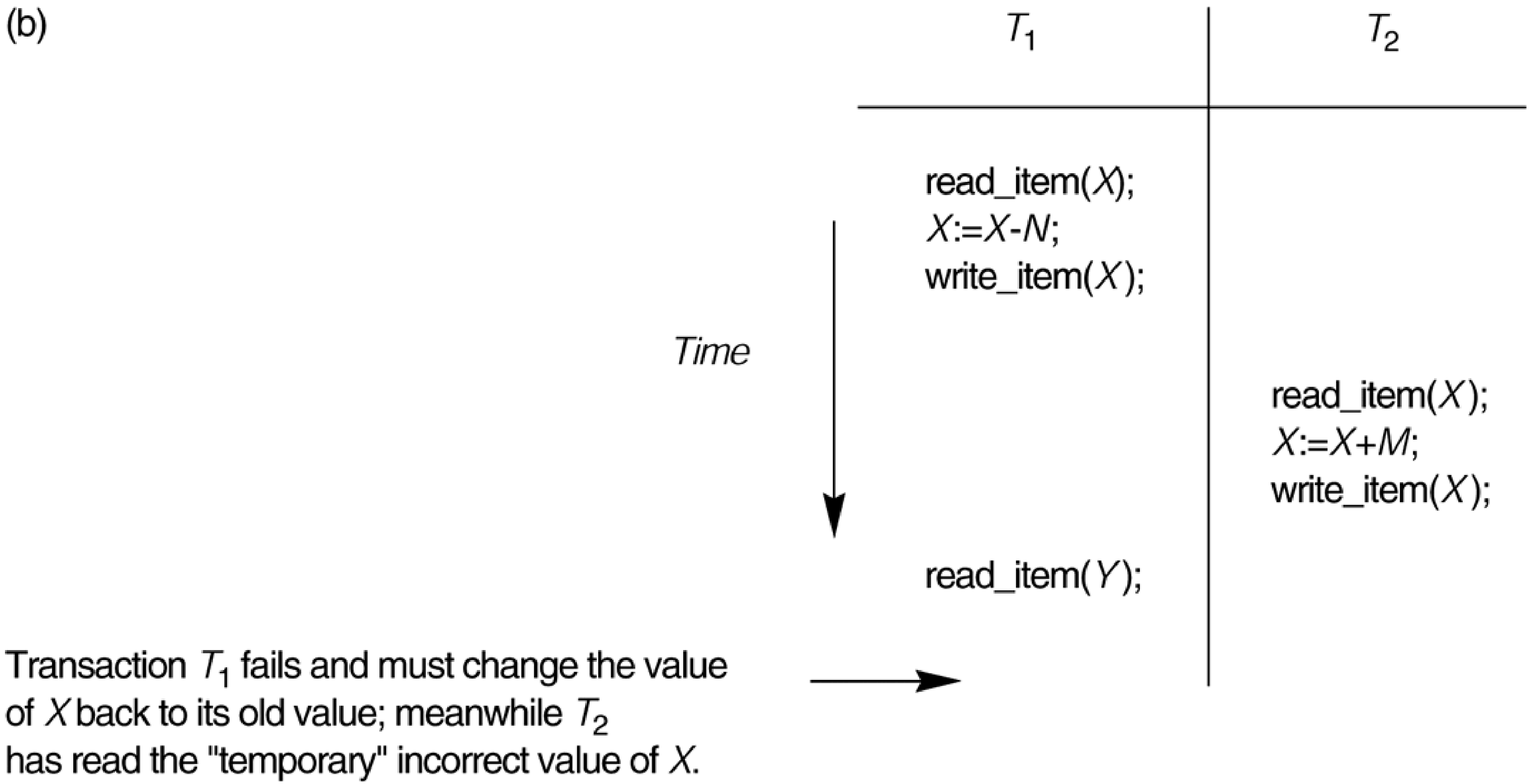
write_item(X);
read_item(Y);	

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

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

FIGURE

Some problems that occur when concurrent execution is uncontrolled. (b) The temporary update problem.



(b)

T_1	T_2

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

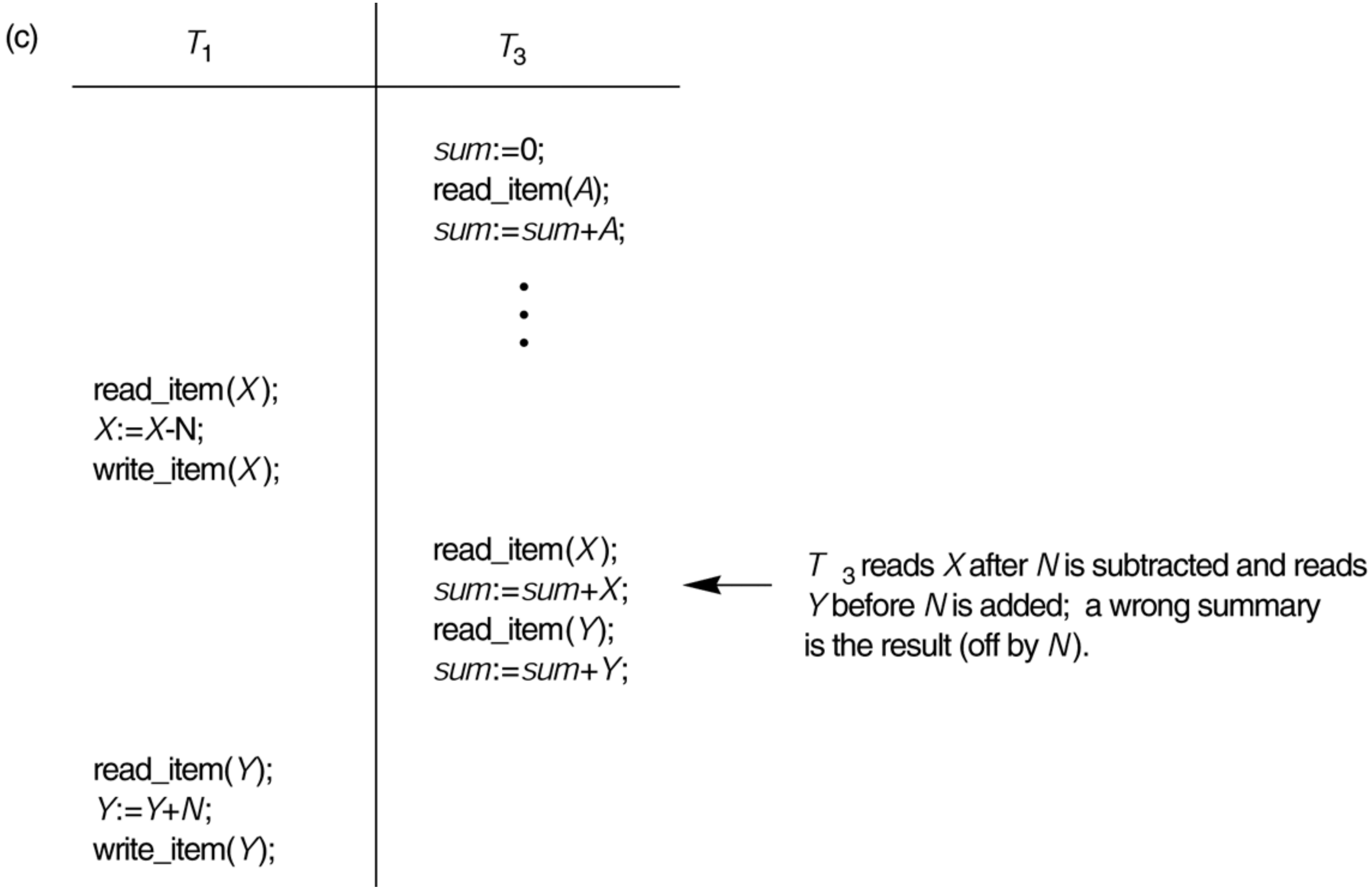
	read_item(X);
	X:=X+M;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

FIGURE

Some problems that occur when 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*).

Introduction to Transaction Processing

Why recovery is needed:

(What causes a Transaction to fail-**Transaction, system and media failures.)**

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.

Introduction to Transaction Processing

Why recovery is needed (cont.):

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 .

Introduction to Transaction Processing

Why recovery is needed (cont.):

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.

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

Transaction and System Concepts

Recovery manager keeps track of the following operations:

- **begin_transaction:** This marks the beginning of transaction execution.

- **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.

- **end_transaction:** This specifies that read and write transaction operations have ended and marks the end limit of transaction execution. At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

Transaction and System Concepts

Recovery manager keeps track of the following operations (cont):

- **commit_transaction:** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

- **rollback (or abort):** This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

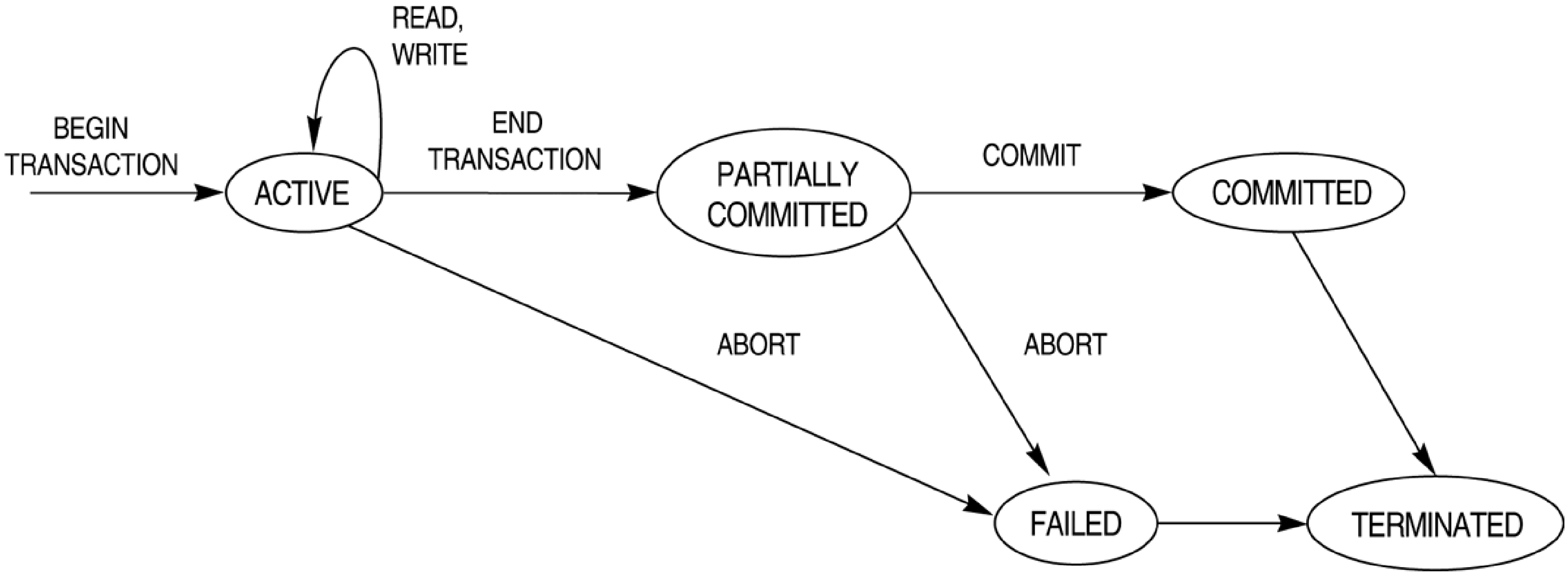
Transaction and System Concepts

Recovery techniques use the following operators:

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

FIGURE

State transition diagram illustrating the states for transaction execution.



Transaction and System Concepts

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.

- 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:

Transaction and System Concepts

The System Log (cont):

Types of log record:

1. **[start_transaction,T]**: Records that transaction T has started execution.

2. [**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.

3. [**read_item,T,X]:** Records that transaction T has read the value of database item X.

4. [**commit,T**]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

5. [**abort,T**]: Records that transaction T has been aborted.

Transaction and System Concepts

The System Log (cont):

- 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.

Transaction and System Concepts

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 later.

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.

Transaction and System Concepts

Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
- Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.

- **Roll Back of transactions:** Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Transaction and System Concepts

Commit Point of a Transaction (cont):

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

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.

- **Durability or permanency**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

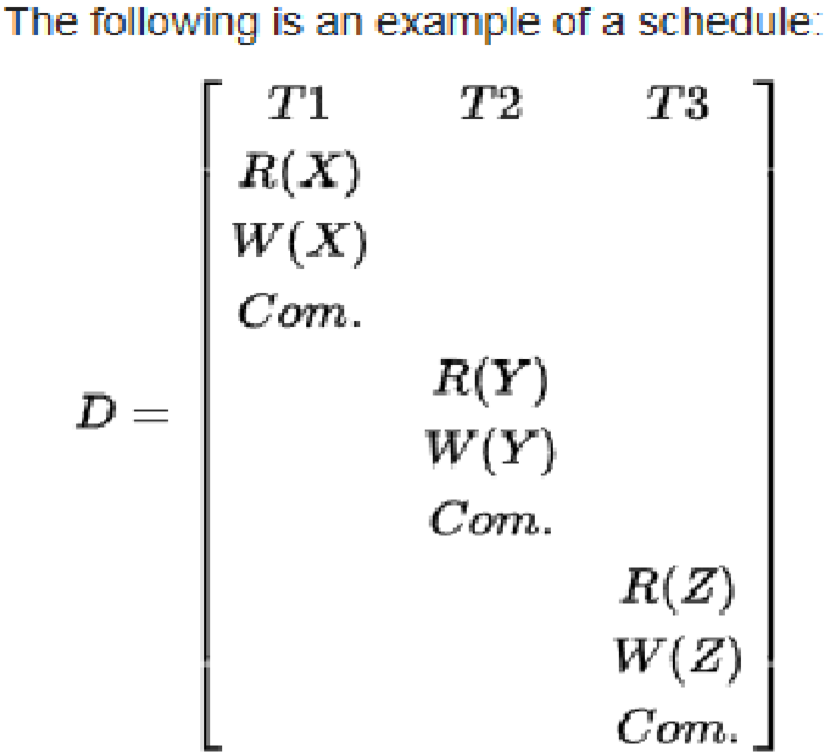
Schedules(Histories) of Transactions

- **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 T1, T2, ..., Tn :

  It is an ordering of the operations of the transactions subject to the constraint that, *for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.*

Examples

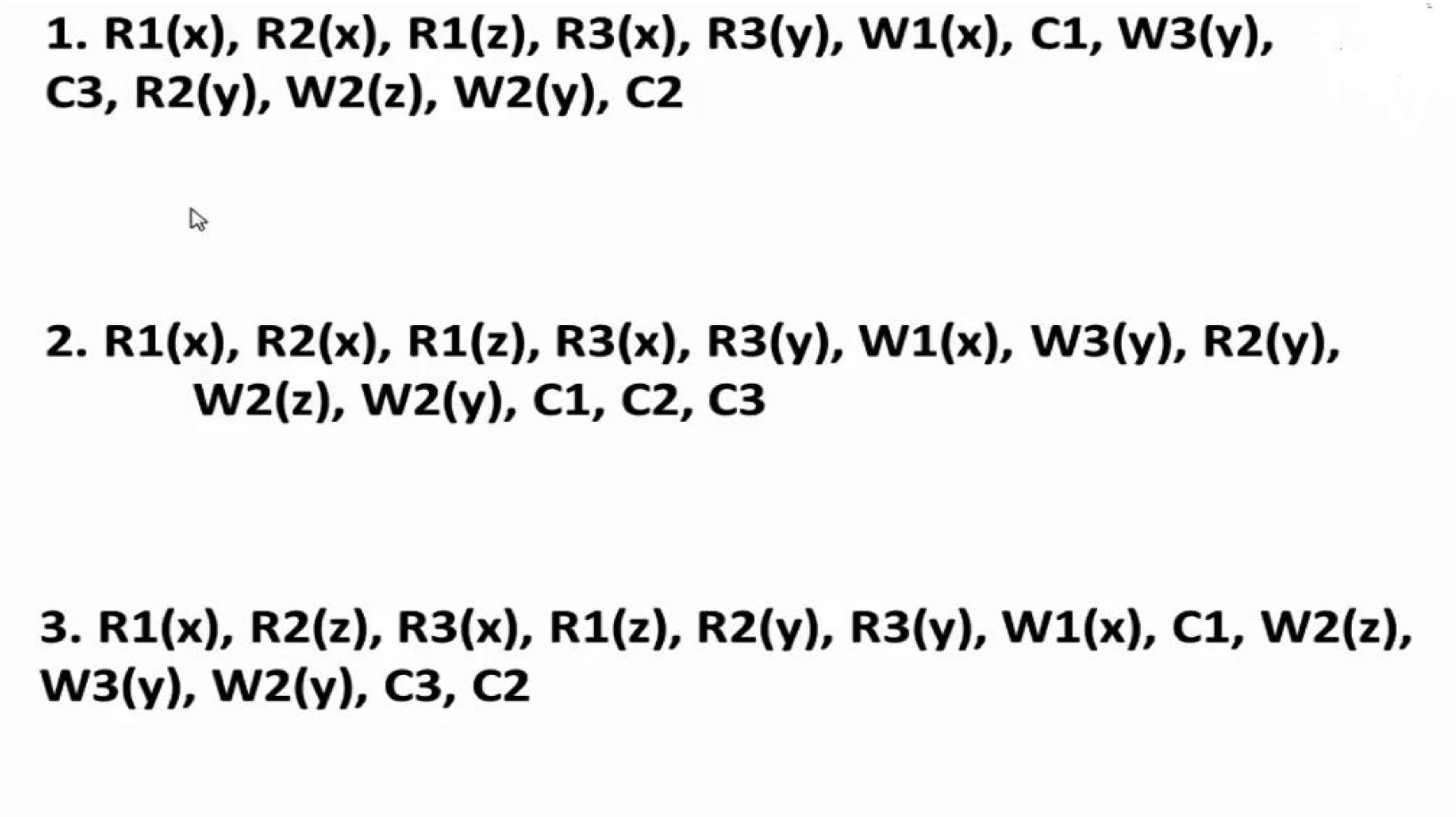


The following is an example of a schedule:

$$D = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ W(X) & & \\ Com. & & \\ & R(Y) & \\ & W(Y) & \\ & Com. & \\ & & R(Z) \\ & & W(Z) \\ & & Com. \end{bmatrix}$$

S= R1(X) ;W1(X) ;C1 ;R2(Y); W2(Y); C2; R3(Z) ;W3(Z) ;C3

Examples:



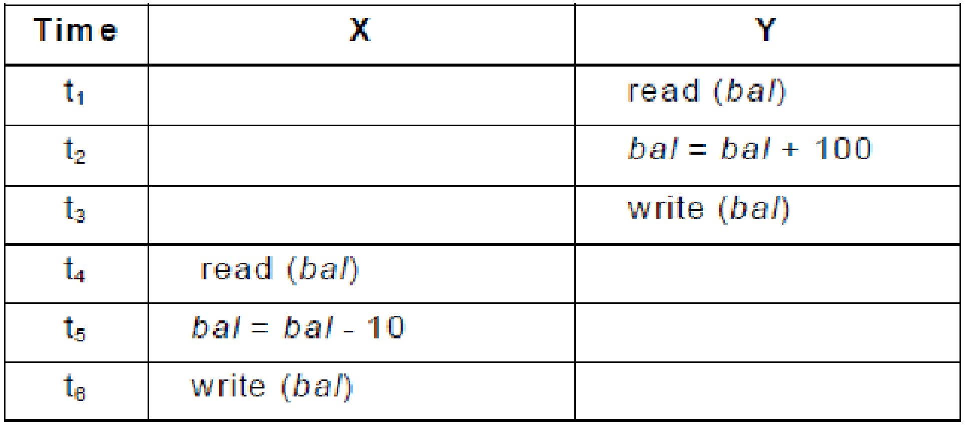
1. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2

2. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), W3(y), R2(y), W2(z), W2(y), C1, C2, C3

3. R1(x), R2(z), R3(x), R1(z), R2(y), R3(y), W1(x), C1, W2(z), W3(y), W2(y), C3, C2

Serial Schedule

serial schedule is one in which no transaction starts until a running transaction has ended

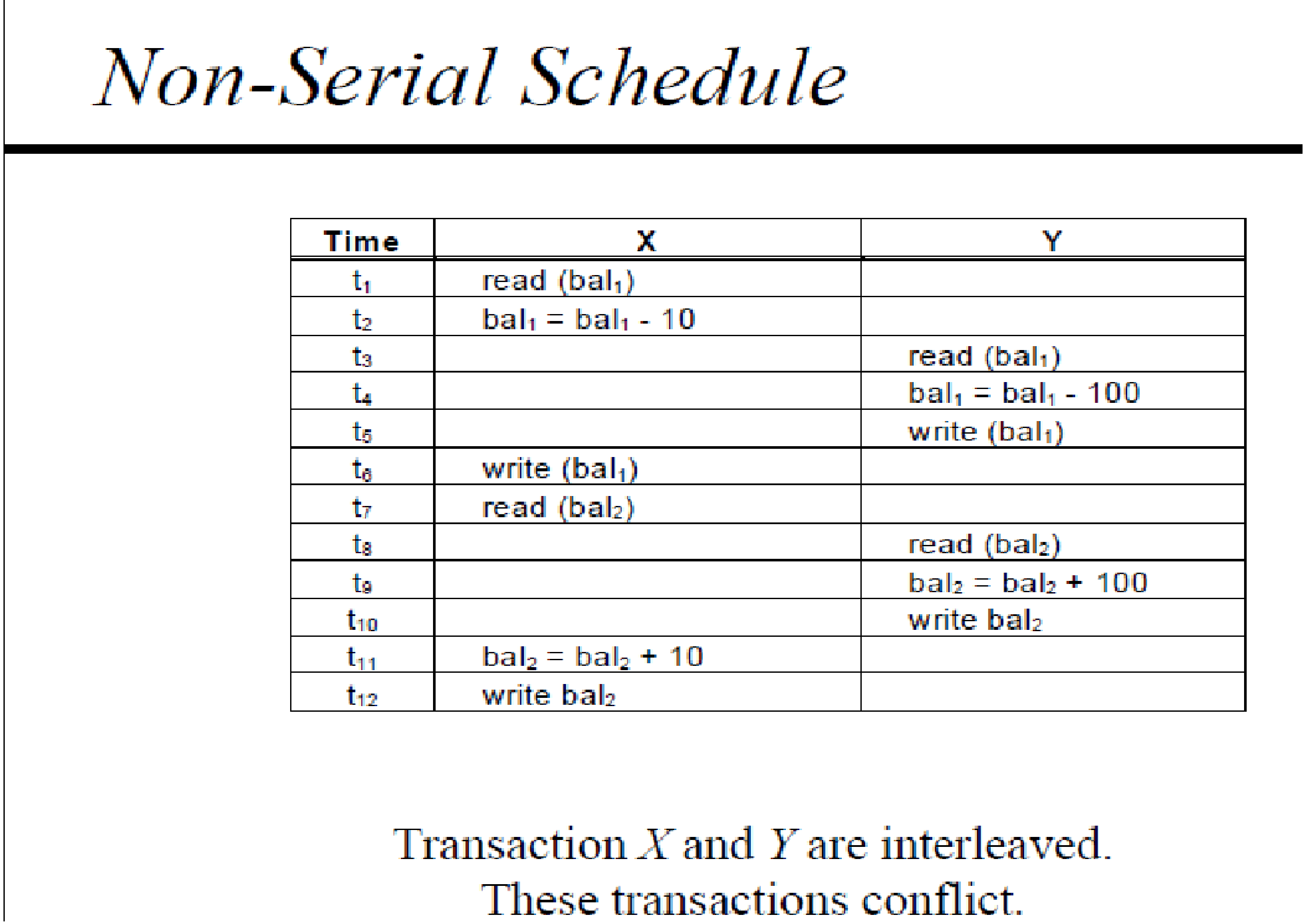


Time	X	Y
t_1		read (*bal*)
t_2		*bal* = *bal* + 100
t_3		write (*bal*)
t_4	read (*bal*)	
t_5	*bal* = *bal* - 10	
t_6	write (*bal*)	

Transaction *Y* executes before transaction *X*

Transactions *Y* and *X* are executed one after another. Therefore, they cannot interfere with each other.

This is a *serial schedule*.



Non-Serial Schedule

Time	X	Y
t_1	read (bal_1)	
t_2	$bal_1 = bal_1 - 10$	
t_3		read (bal_1)
t_4		$bal_1 = bal_1 - 100$
t_5		write (bal_1)
t_6	write (bal_1)	
t_7	read (bal_2)	
t_8		read (bal_2)
t_9		$bal_2 = bal_2 + 100$
t_{10}		write bal_2
t_{11}	$bal_2 = bal_2 + 10$	
t_{12}	write bal_2	

Transaction *X* and *Y* are interleaved.
These transactions conflict.

Transactions *X* and *Y* are interleaved. That is, the operations of transaction *X* overlap with the operations of transaction *Y*.

This is a *non-serial schedule*.

This schedule produces a conflict because at the same time transaction *X* is transferring £10 from bal_1 to bal_2, transaction *Y* is also transferring money between bal_1 and bal_2.

Conflicting Operations

- Two operations are said to be in conflict (conflicting pair) if:
  - ➢The operations belong to different transactions.
  - ➢At least one of the operations is a write operation.
  - ➢The operations access the same object (read or write).
- The following set of actions is conflicting:
  - – R1(X), W2(X), W3(X) (3 conflicting pairs)
- While the following sets of actions are not:
  - – R1(X), R2(X), R3(X)
  - – R1(X), W2(Y), R3(X)

- A schedule *S of n transactions , , ..., , is said to be a* ***complete schedule if the following conditions*** hold:

1. The operations in *S are exactly those operations in T1 , , ...,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 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

- 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 whose commit operation is in S.

Characterizing Schedules based on Recoverability

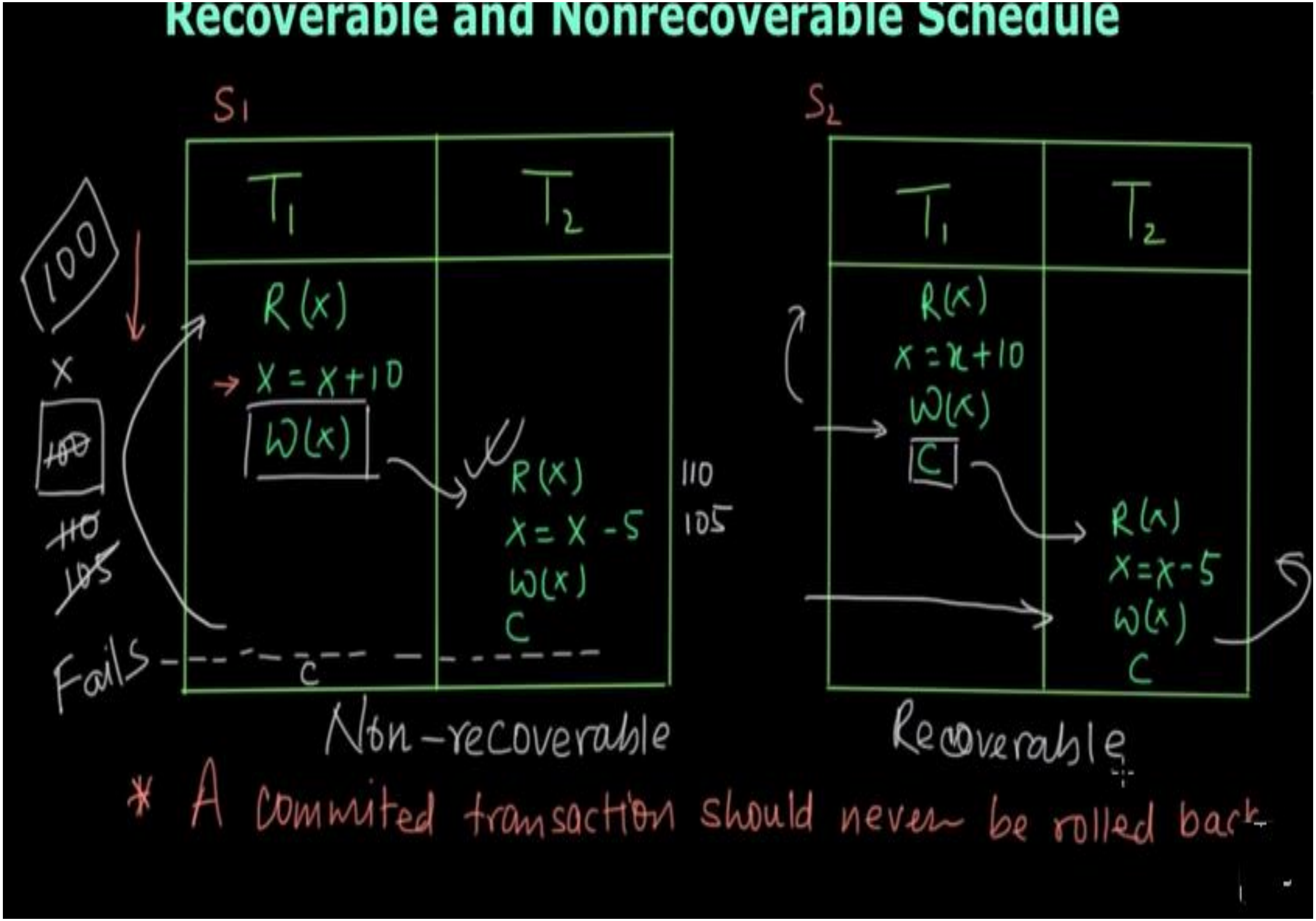
Schedules classified on recoverability:

- **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 reads only the items that are written by committed transactions.

  Schedules requiring cascaded rollback: A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.



Recoverable and Nonrecoverable Schedule

S1

T1	T2
R(x)	
X=X+10	
W(x)	
	R(x)
	X=X-5
	W(x)
	C

Non-recoverable

S2

T1	T2
R(x)	
x=x+10	
W(x)	
C	
	R(x)
	x=x-5
	W(x)
	C

Recoverable

* A commited transaction should never be rolled back

Characterizing Schedules based on Recoverability

Schedules classified on recoverability:

- **Strict Schedules:** A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

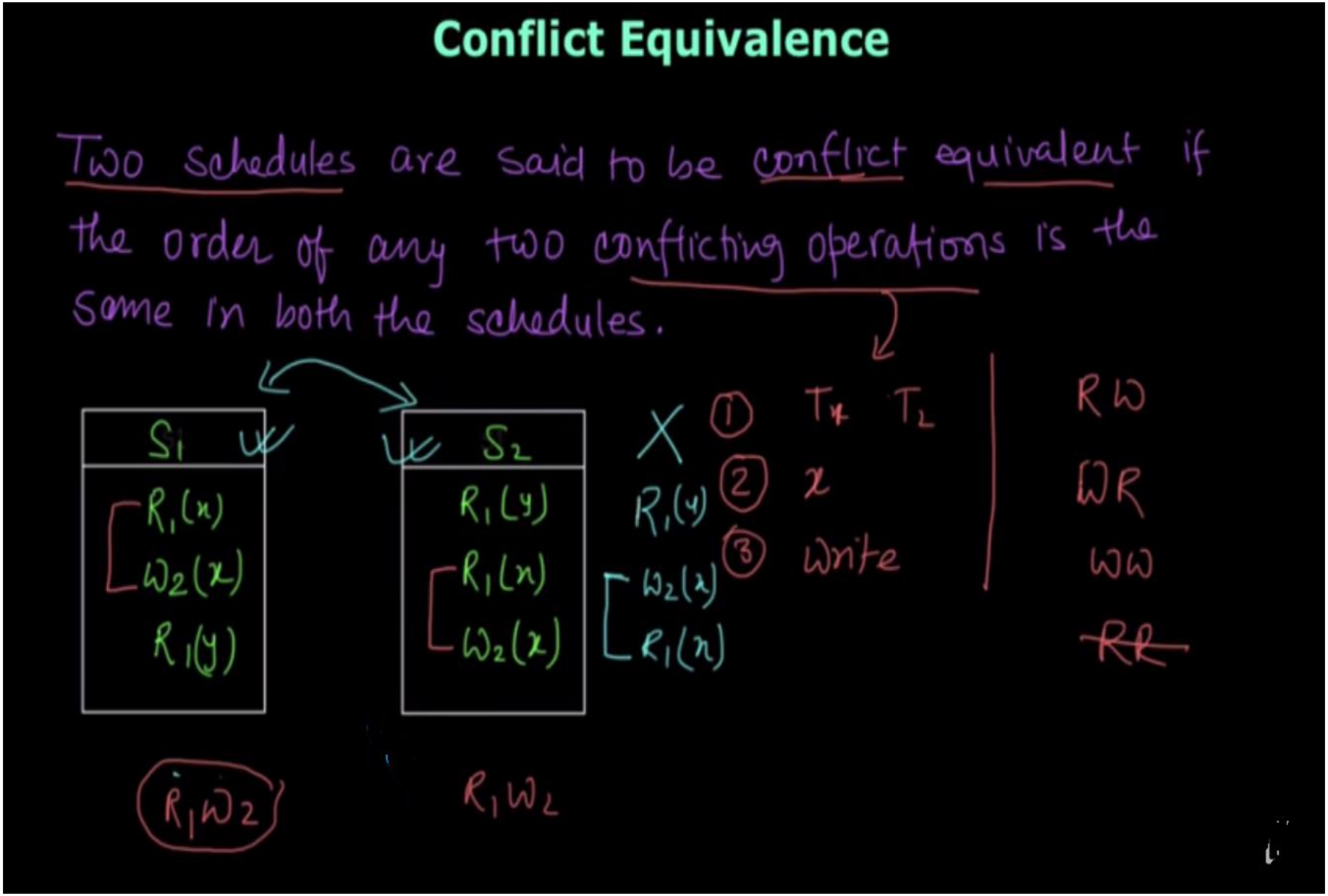
Characterizing Schedules based on Serializability

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

Characterizing Schedules based on Serializability

- **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'.

Conflict equivalent:



Conflict Equivalence

Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both the schedules.

S_1	S_2
$R_1(x)$	$R_1(y)$
$W_2(x)$	$R_1(x)$
$R_1(y)$	$W_2(x)$

$R_1(y)$

$W_2(x)$

$R_1(x)$

① T_1 T_2

② x

③ write

RW

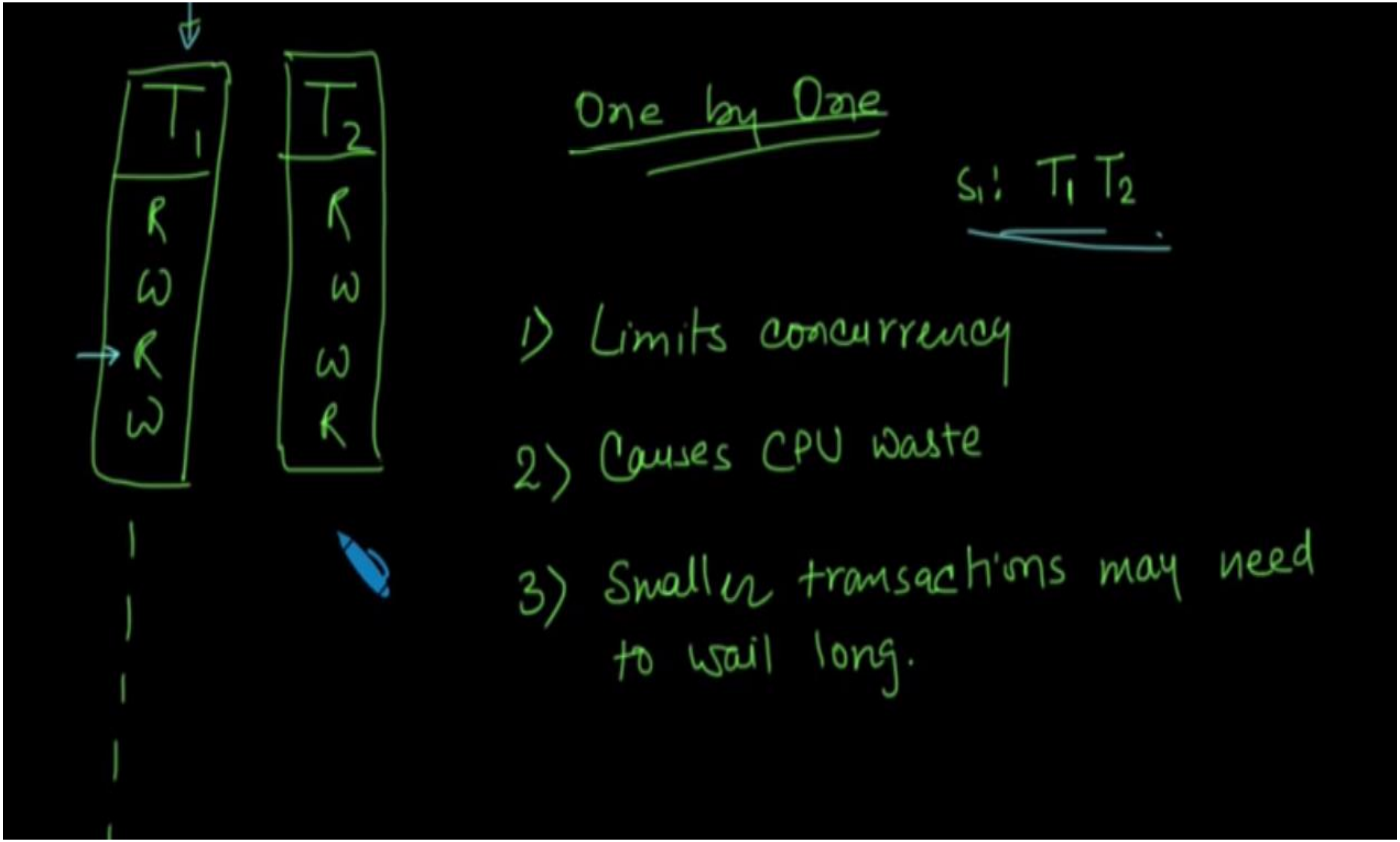
WR

WW

~~RR~~

R_1W_2 R_1W_2

Understanding Serializability



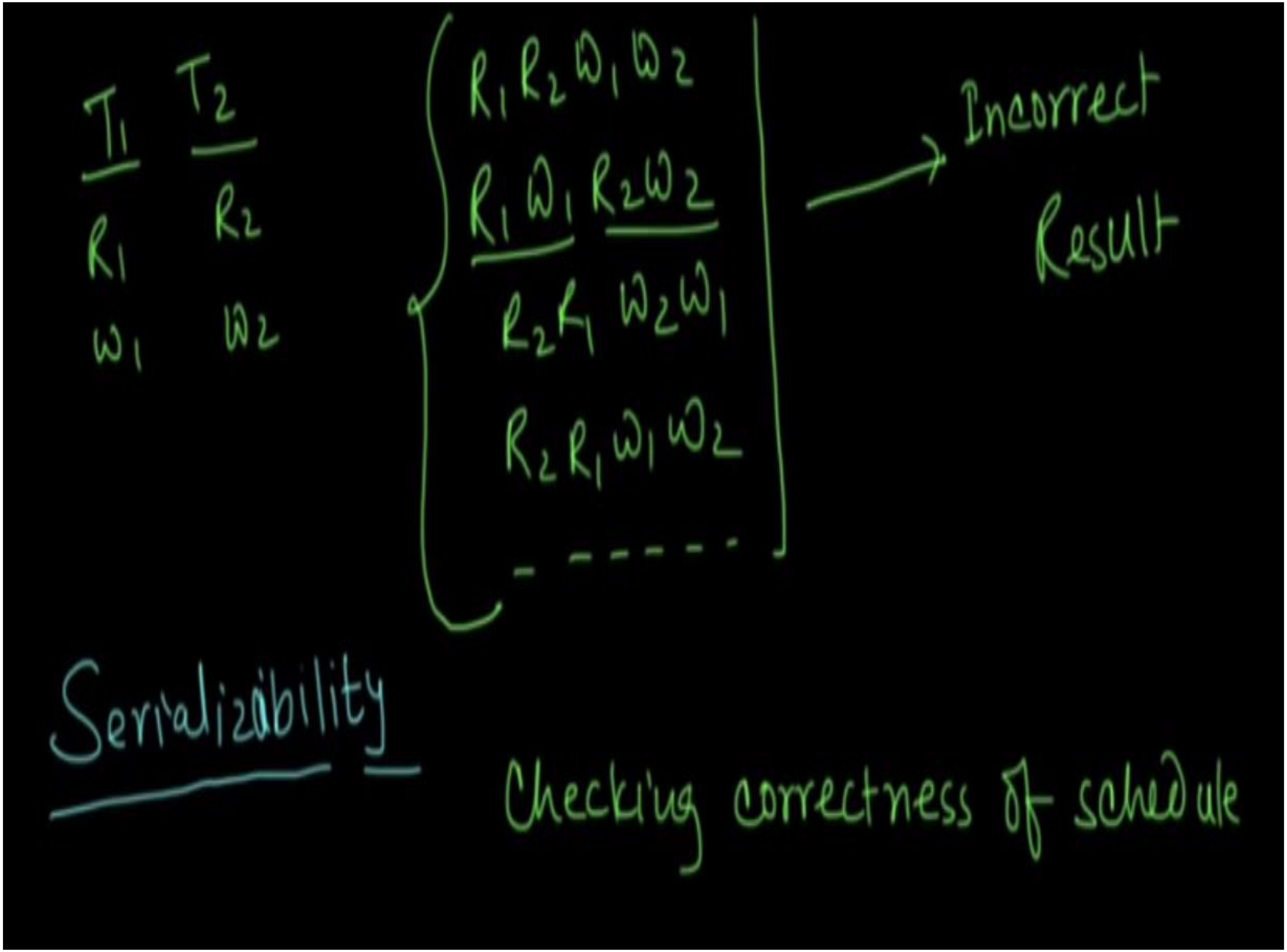
One by One

S_1: T_1 T_2

1) Limits concurrency

2) Causes CPU waste

3) Smaller transactions may need to wail long.



T_1	T_2
R_1	R_2
W_1	W_2

$R_1 R_2 W_1 W_2$

$\underline{R_1 W_1}\ \underline{R_2 W_2}$

$R_2 R_1 W_2 W_1$

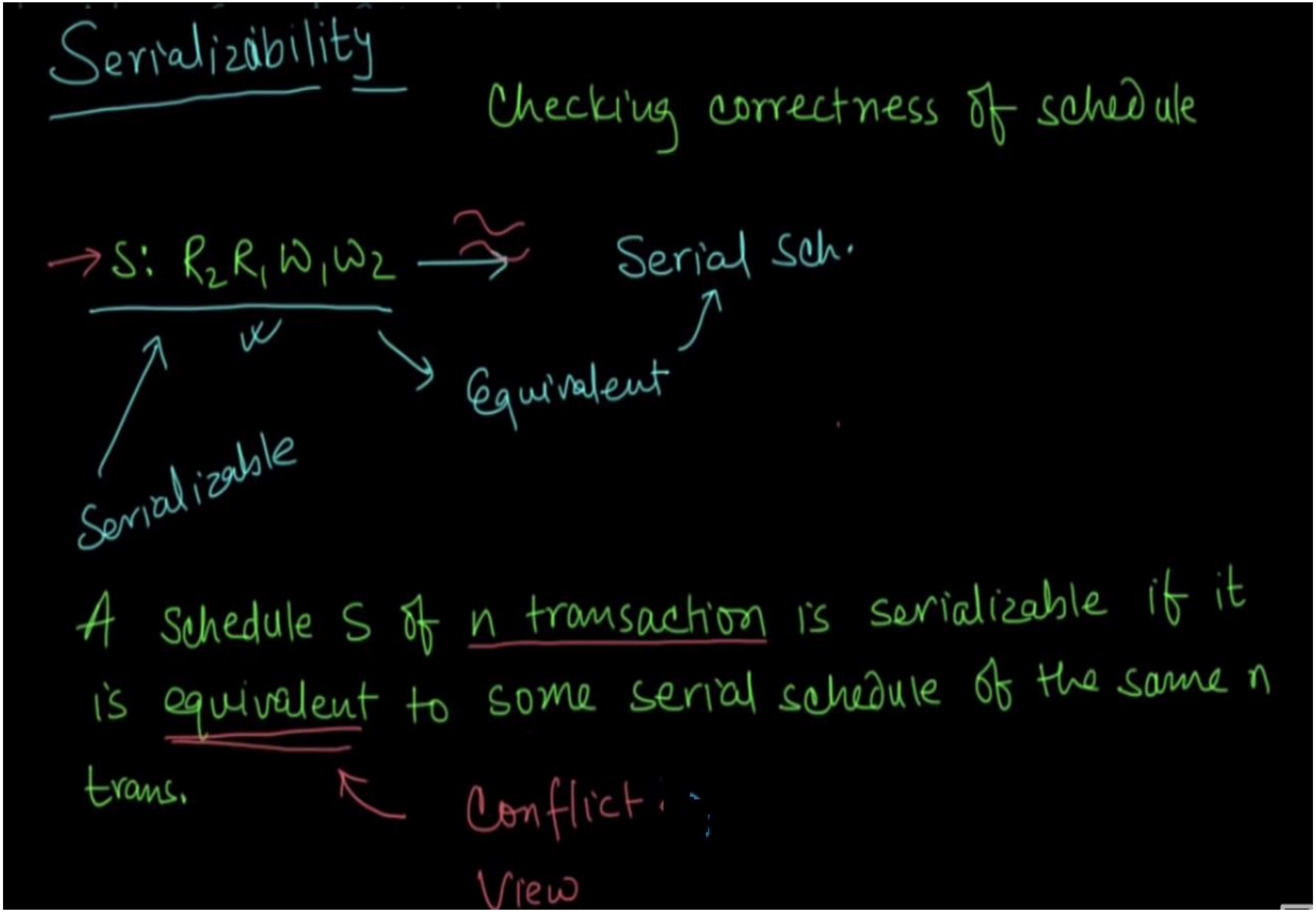
$R_2 R_1 W_1 W_2$

- - - - - - -

→ Incorrect Result

Serializability

Checking correctness of schedule



Serializability

Checking correctness of schedule

$S: R_2 R_1 W_1 W_2 \xrightarrow{\simeq}$ Serial Sch.

Serializable

Equivalent

A schedule S of n transaction is serializable if it is equivalent to some serial schedule of the same n trans.

Conflict

View

Characterizing Schedules based on Serializability

- Being serializable is not the same as being serial

- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Characterizing Schedules based on Serializability

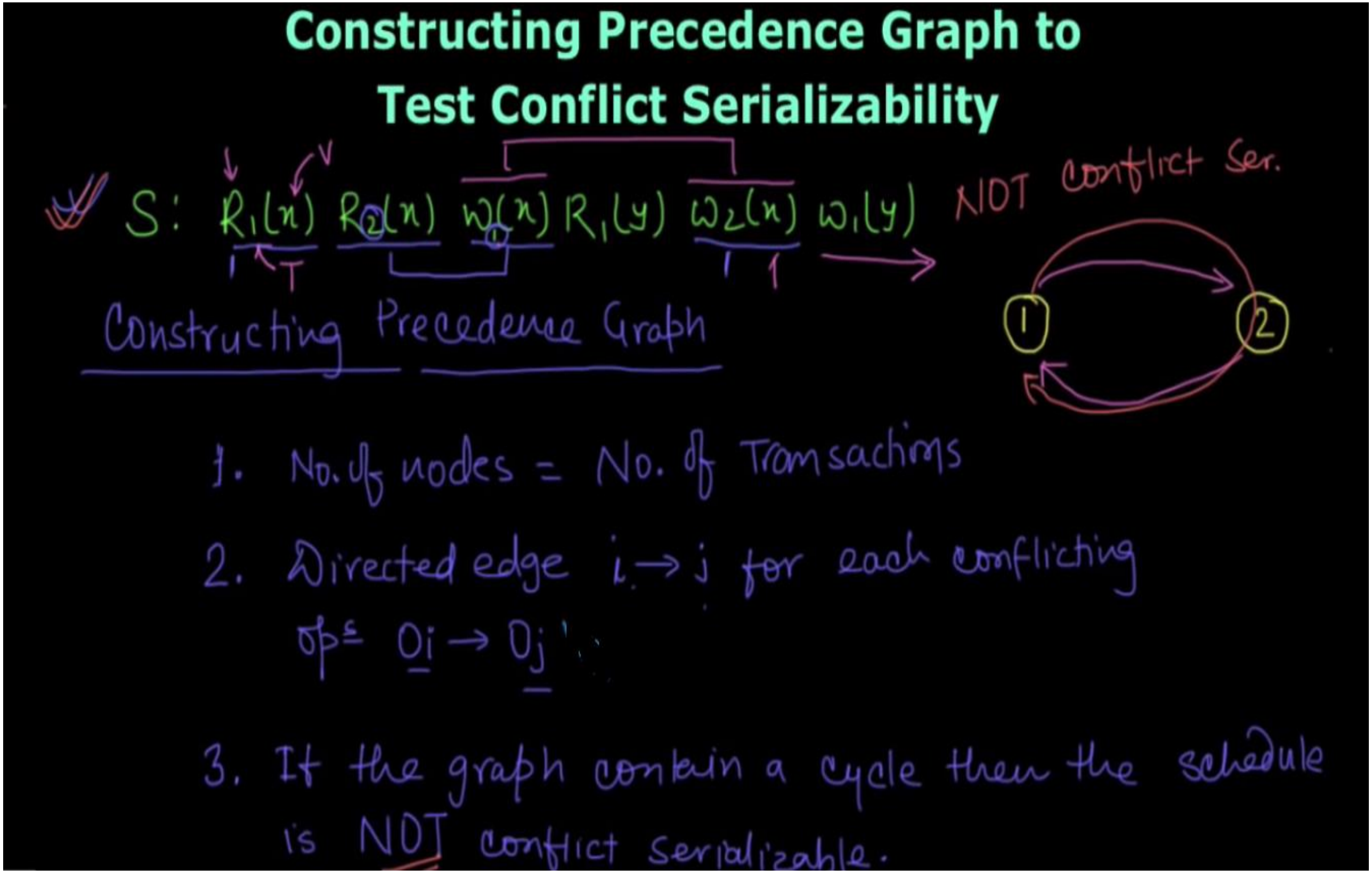
- Serializability is hard to check.
  – Interleaving of operations occurs in an operating system through some scheduler
  – Difficult to determine beforehand how the operations in a schedule will be interleaved.

Characterizing Schedules based on 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

Testing for Conflict Serializability



Constructing Precedence Graph to Test Conflict Serializability

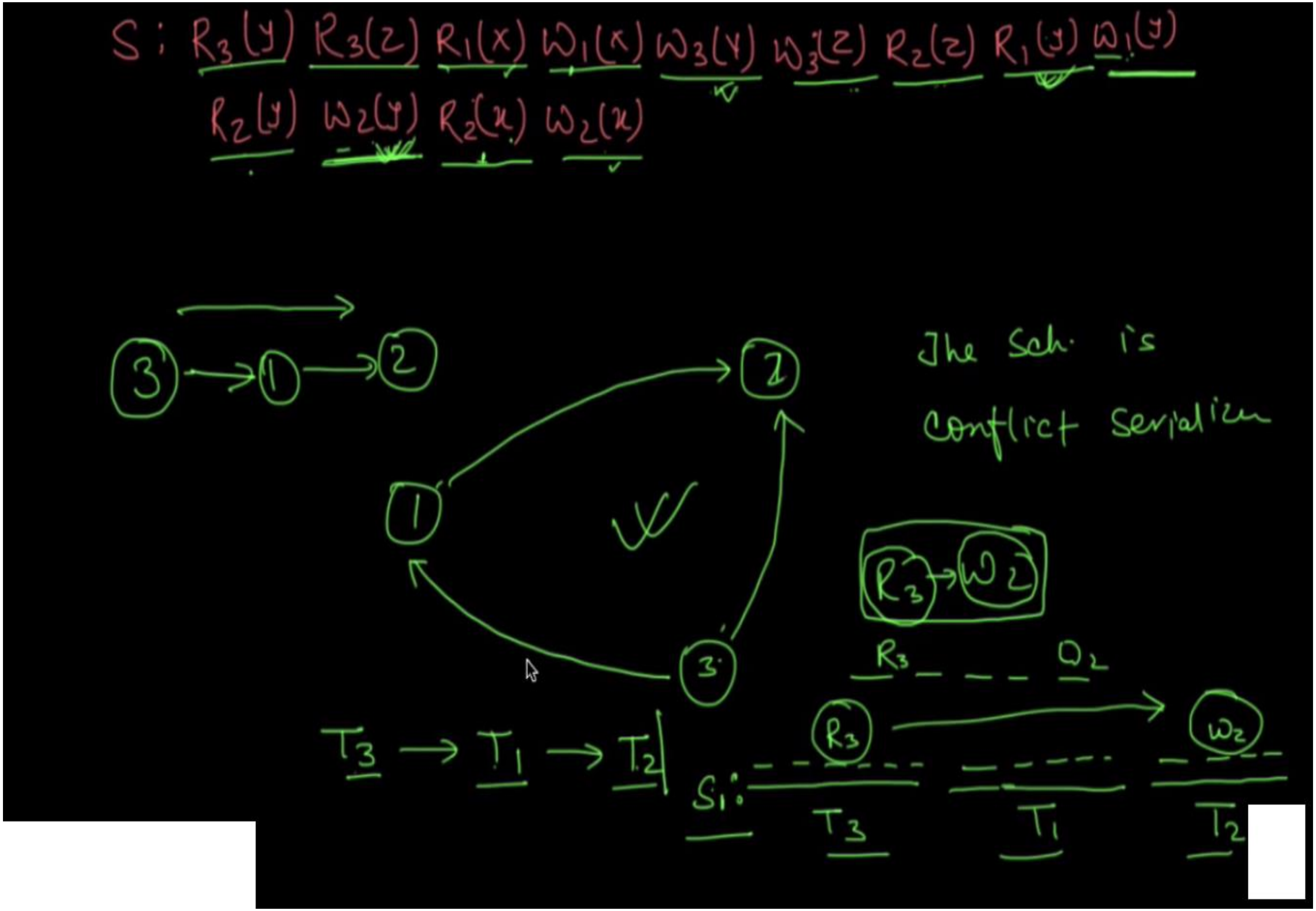
S: $R_1(x)$ $R_2(x)$ $W_1(x)$ $R_1(y)$ $W_2(x)$ $W_1(y)$ → NOT conflict Ser.

Constructing Precedence Graph

1. No. of nodes = No. of Transactions

2. Directed edge i → j for each conflicting ops $O_i \rightarrow O_j$

3. If the graph contain a cycle then the schedule is NOT conflict serializable.



$S: R_3(y)\ R_3(z)\ R_1(x)\ W_1(x)\ W_3(y)\ W_3(z)\ R_2(z)\ R_1(y)\ W_1(y)$
$R_2(y)\ W_2(y)\ R_2(x)\ W_2(x)$

The Sch. is conflict serializ

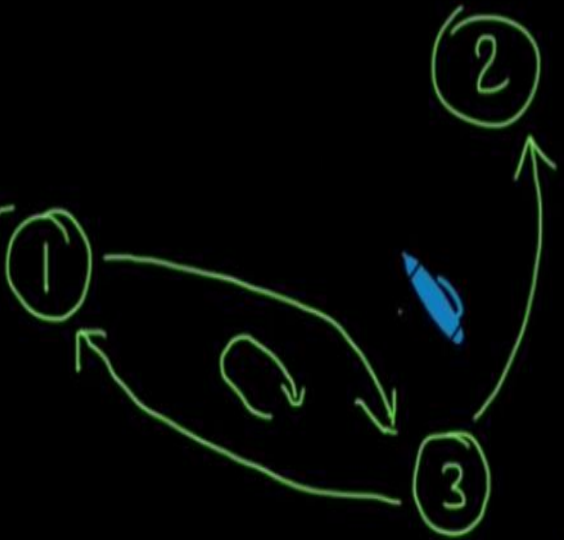
$T_3 \rightarrow T_1 \rightarrow T_2$

S_1: T_3 T_1 T_2

Find the conflict equivalent serial schedule for the following Schedule :-

$$\underline{S} : \underline{R_3(y)}\ \underline{R_3(z)}\ \underline{R_1(\overset{\downarrow}{y})}\ W_1(x)\ W_3(y)\ W_3(z)\ R_2(z)\ R_1(y)\ W_1(y)\ R_2(y)$$
$$W_2(y)\ R_2(x)\ W_2(x)$$

Not Conflict Serializ



We can not have any confict equivalent Serial sch. for the Schedule as the schedule is not conflict serizable.

Characterizing Schedules based on Serializability

- **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 View 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 Ri(X) of Ti in S, if the value of X read by the operation has been written by an operation Wj(X) of Tj (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 Ri(X) of Ti in S’.
3. If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S’.

Characterizing Schedules based on Serializability (8)

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction musr produce the same results.
- **“The view”**: the read operations are said to see the *the same view* in both schedules.

Characterizing Schedules based on Serializability

Relationship between view and conflict equivalence:

- The two are 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)
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Characterizing Schedules based on Serializability (10)

Relationship between view and conflict equivalence (cont):

Consider the following schedule of three transactions

T1: r1(X), w1(X);   T2: w2(X);   and   T3: w3(X):

Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sa, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.

Sa is **view serializable**, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is **not conflict serializable**, since it is not conflict equivalent to any serial schedule.

Characterizing Schedules based on Serializability (11)

Testing for conflict serializability

Algorithm 17.1:

1. Looks at only read_Item (X) and write_Item (X) operations
2. Constructs a precedence graph (serialization graph) - a graph with directed edges
3. 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
4. The schedule is serializable if and only if the precedence graph has no cycles.

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*).

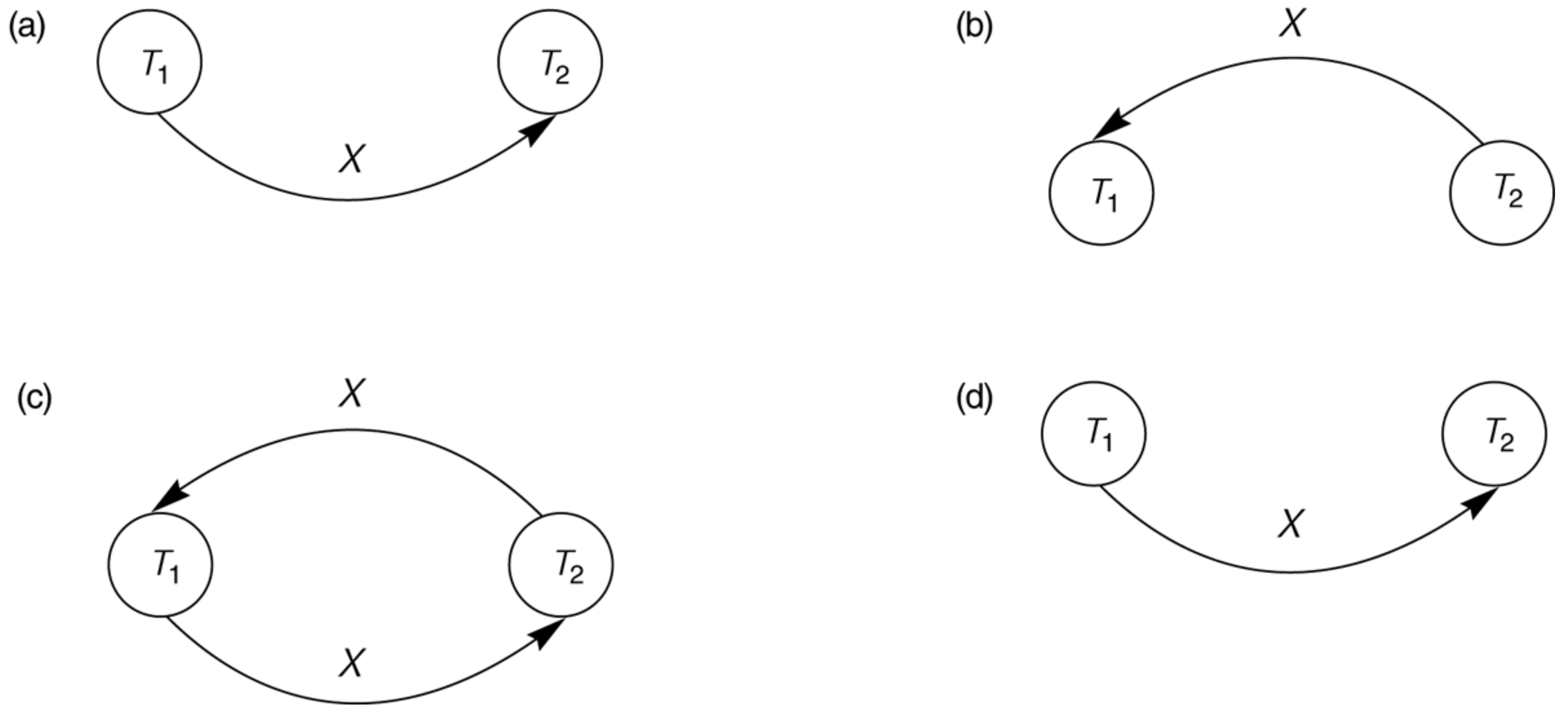
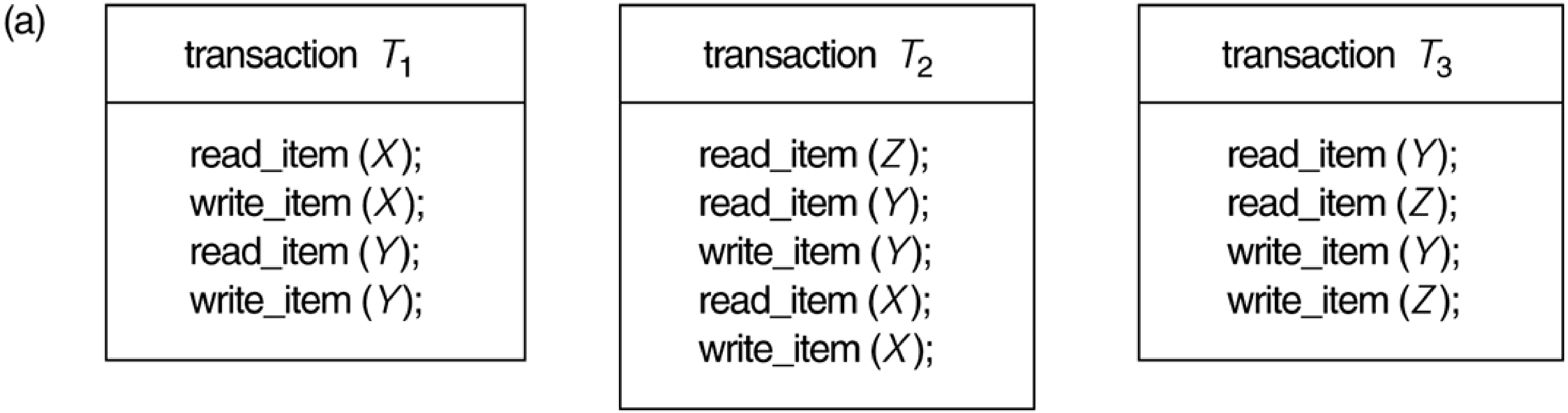


FIGURE 17.8

Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1, T_2, and T_3.

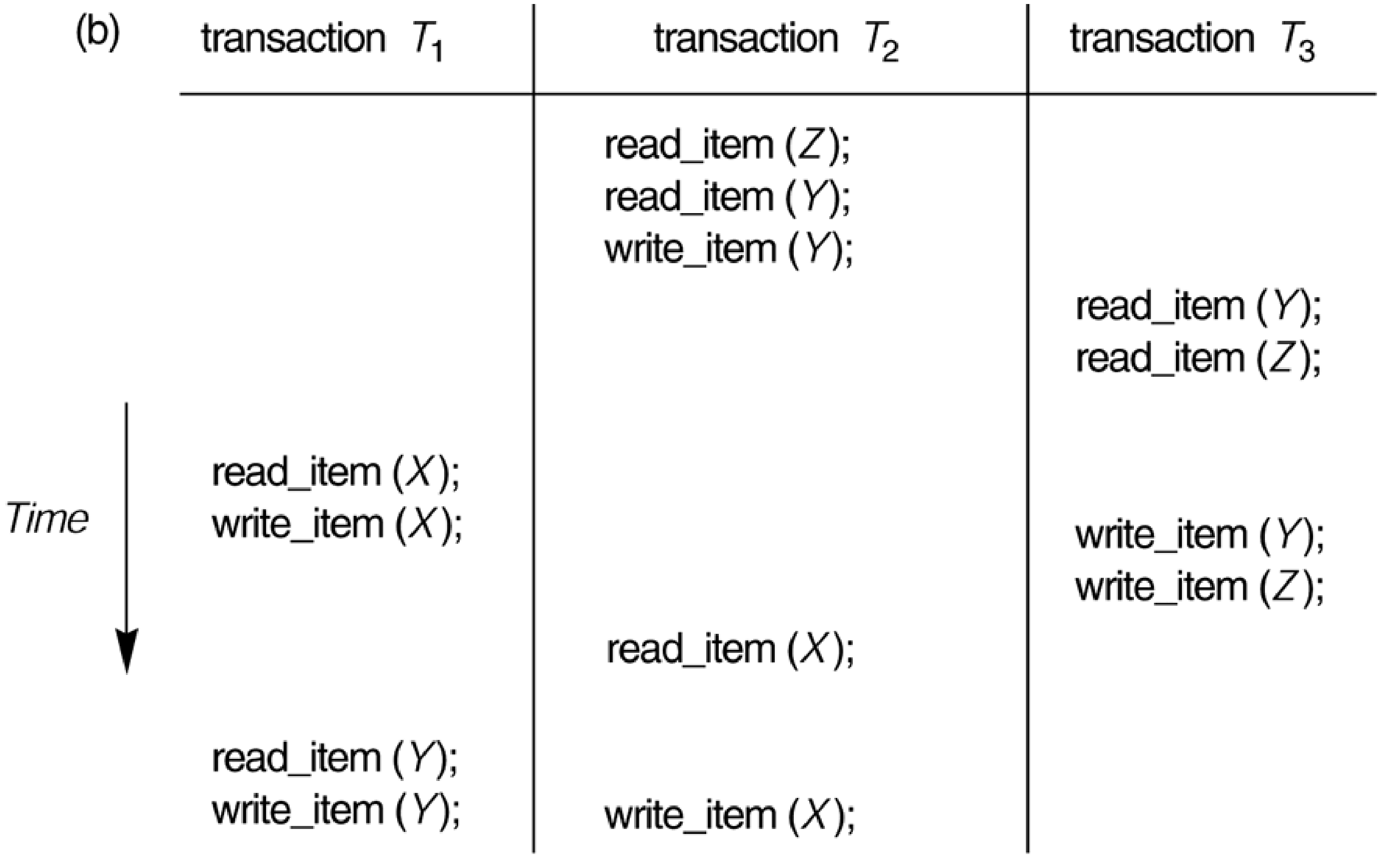


(a)

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

FIGURE 17.8 (continued)

Another example of serializability testing. (b) Schedule *E*.

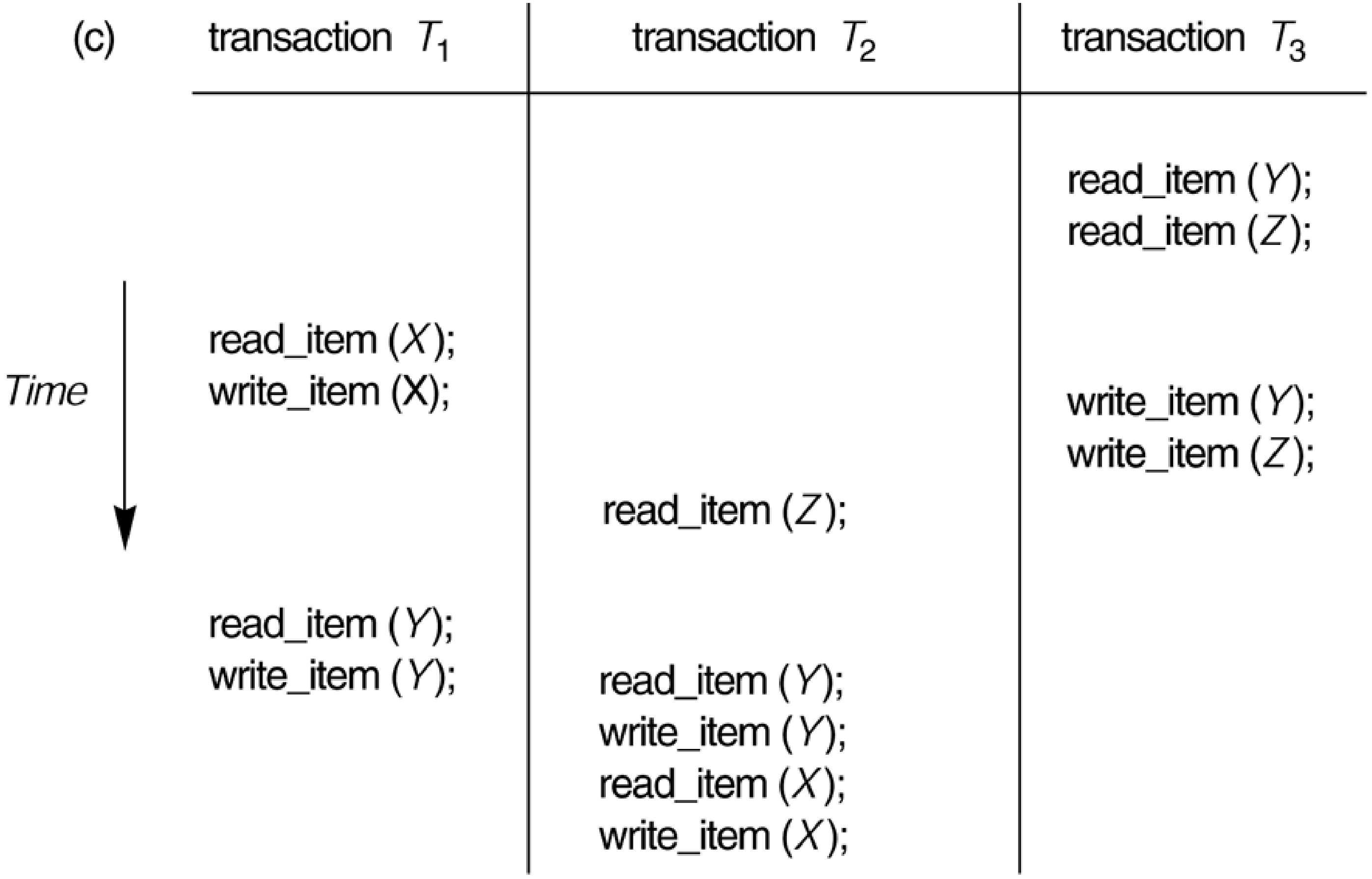


transaction T_1	transaction T_2	transaction T_3
	read_item (*Z*);	
	read_item (*Y*);	
	write_item (*Y*);	
		read_item (*Y*);
		read_item (*Z*);
read_item (*X*);		
write_item (*X*);		
		write_item (*Y*);
		write_item (*Z*);
	read_item (*X*);	
read_item (*Y*);		
write_item (*Y*);		
	write_item (*X*);	

Schedule E

FIGURE 17.8 (continued)

Another example of serializability testing. (c) Schedule *F*.



(c)	transaction T_1	transaction T_2	transaction T_3
			read_item (*Y*);
			read_item (*Z*);
	read_item (*X*);		
Time	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

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.