Transaction Processing Concepts and Theory

Outline

1 Introduction to Transaction Processing

2 Transaction and System Concepts

3 Desirable Properties of Transactions

4 Characterizing Schedules based on Recoverability

5 Characterizing Schedules based on Serializability

1.1 Introduction to Transaction Processing

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

1.2 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.

1.3 Introduction to Transaction Processing

- 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 in the following discussion)*.

  write_item(X): Writes the value of program variable X into the database item named X.

1.4 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.**

1.5 Introduction to Transaction Processing

read_item(X) :

- **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 X.**

1.6 Introduction to Transaction Processing

write_item(X) :

- **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).**

1.7 Introduction to Transaction Processing - Two sample transactions T_1(a) & T_2(b)

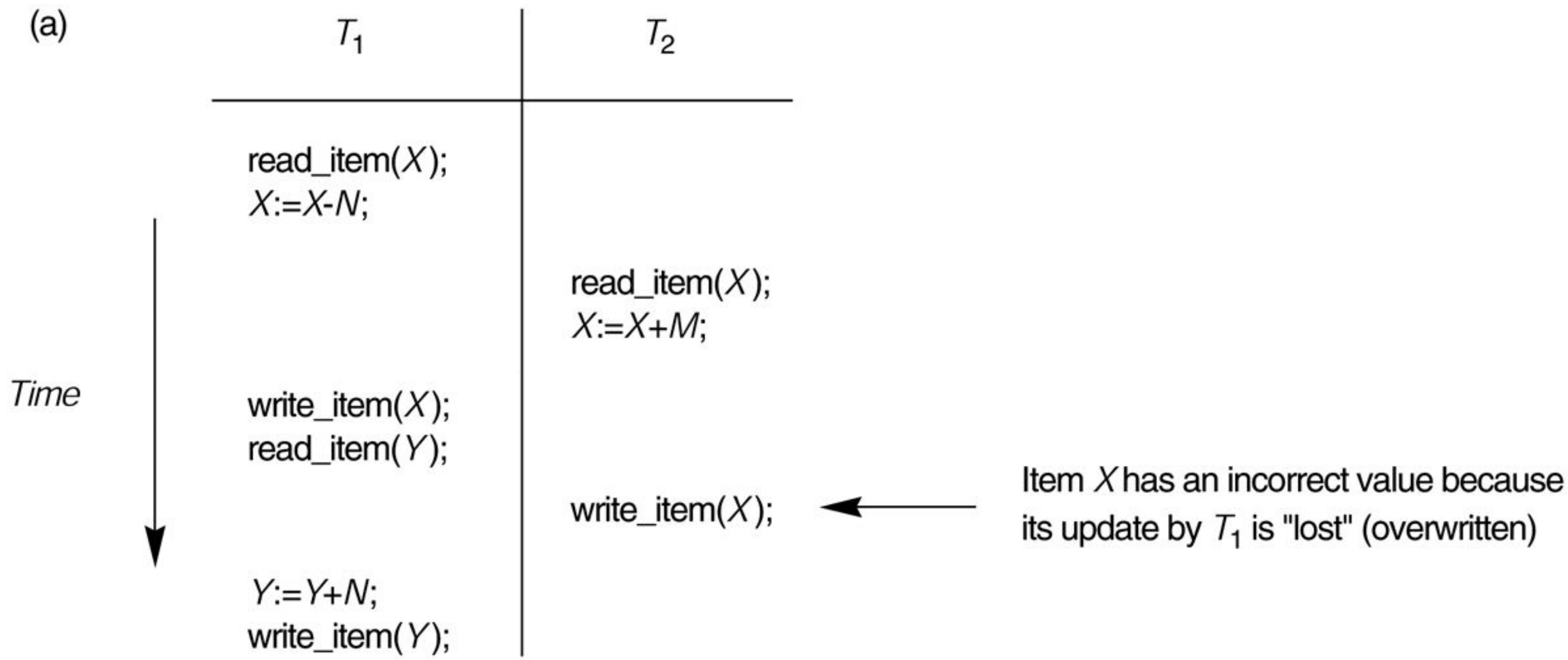
(a) T_1	(b) T_2
read_item (X);	read_item (X);
X:=X-N;	X:=X+M;
write_item (X);	write_item (X);
read_item (Y);	
Y:=Y+N;	
write_item (Y);	

1.8 Introduction to Transaction Processing

The need for Concurrency Control:

- **The Lost Update Problem - two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.**

1.9 Introduction to Transaction Processing - The lost update problem.

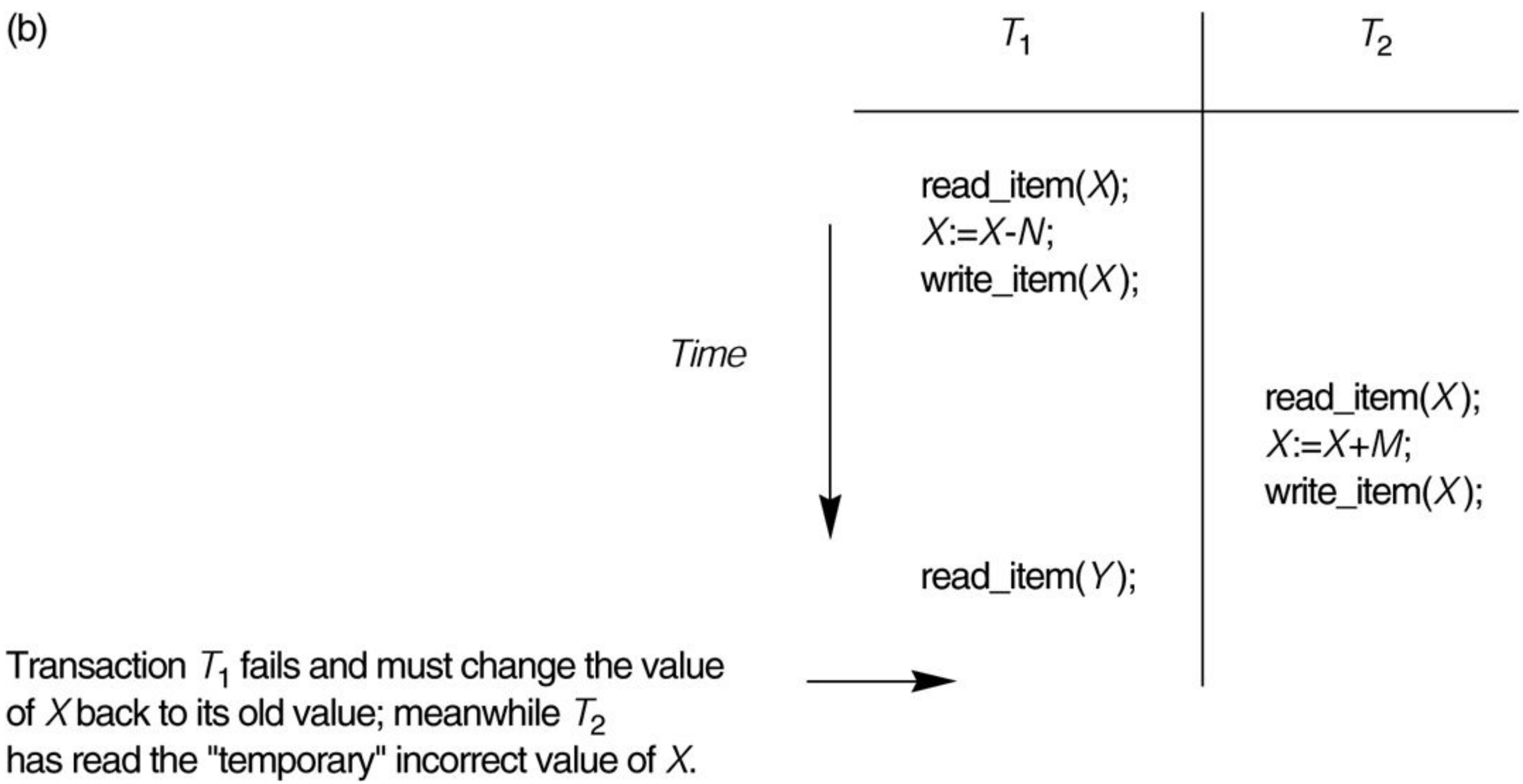


1.10 Introduction to Transaction Processing

The need for Concurrency Control:

- **The Temporary Update (or Dirty Read) Problem - 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.**

1.11 Introduction to Transaction Processing - The temporary update problem.



1.12 Introduction to Transaction Processing

The need for Concurrency Control:

- **The Incorrect Summary Problem - one transaction is calculating an aggregate summary function records while other transactions are updating some of these records, the aggregate function may calculate some values**

1.13 Introduction to Transaction Processing - 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*).

1.14 Introduction to Transaction Processing

Causes of Transaction Failure:

- **System Crash - hardware or software error during transaction execution.**
- **A transaction or system error – Programming errors like integer overflow or division by zero, erroneous parameter values or logical programming error.**
- **User Interruption, Program abortion**
- **Local errors or exception conditions – Ex. insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.**
- **Physical problems and catastrophes- Ex. Power failures**

2.1 Transaction and System Concepts

- **A transaction - 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**

2.2 Transaction and System Concepts

Recovery manager keeps track of following operations:

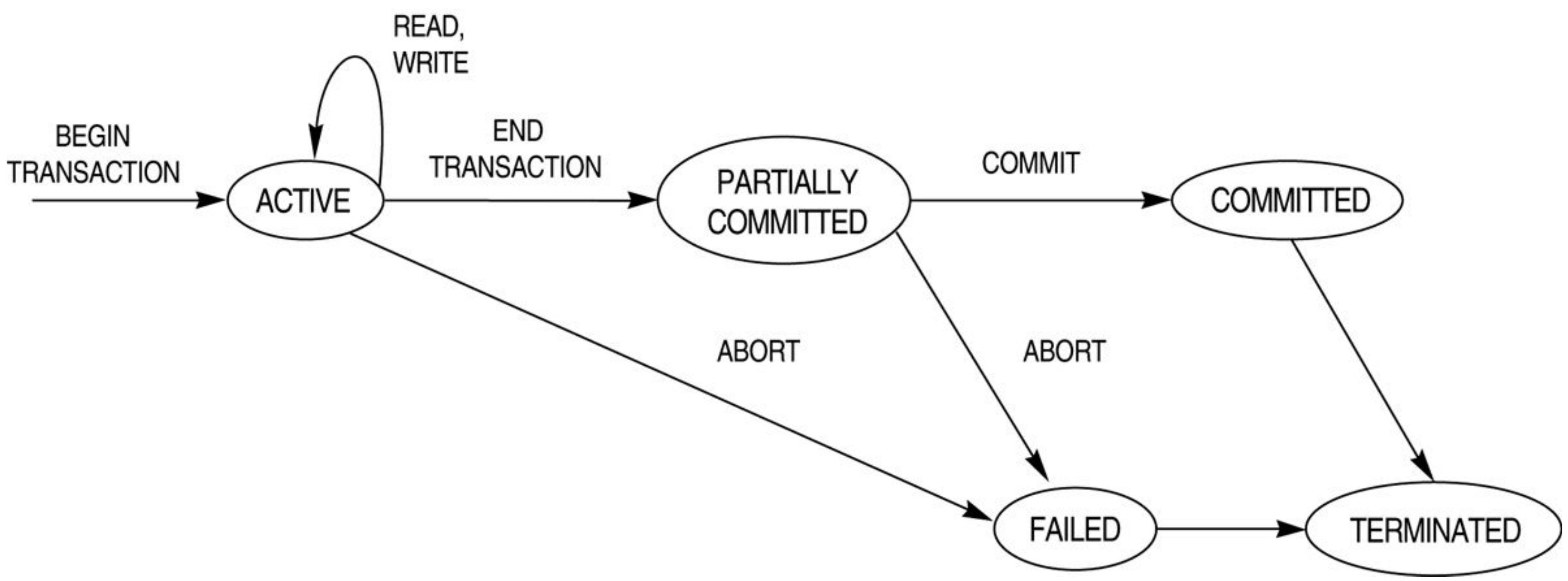
- **begin_transaction - beginning of transaction execution.**
- **read or write - read or write operations on the database items that are executed as part of a transaction.**
- **end_transaction - specifies that read and write transaction operations have ended and marks the end limit of transaction execution.**
- **commit_transaction - successful end of the transaction so that any changes (updates) can be safely committed to the database and will not be undone.**
- **rollback (or abort) - transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.**

2.3 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.**

2.4 Transaction and System Concepts



2.5 Transaction and System Concepts

- **Log or Journal**

  – records transaction history; stored on the disk to avoid catastrophic failures. Using the history any transaction can be undone if it has not reached the commit point.

2.6 Transaction and System Concepts

Types of log entries:

- **[start_transaction,T]: Records that transaction T has started execution.**
- **[write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.**
- **[read_item,T,X]: Records that transaction T has read the value of database item X.**
- **[commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.**
- **[abort,T]: Records that transaction T has been aborted.**

3.1 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.**

3.2 Desirable Properties of Transactions

ACID properties (cont.):

- **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.

4.1 Characterizing Schedules based on Recoverability

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

- Formal Definition – Recoverability
  - A schedule is recoverable if no transaction T commits until all the transactions that provide input to T have committed
  - No outputting transaction should have aborted before inputting to T
  - There should be no intermediate transactions between the outputting and inputting transactions

4.2 Characterizing Schedules based on Recoverability

Examples:

S_1: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1
(Recoverable; no input from any transaction to the other)

S_2: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1
(Not recoverable; T1 input to T2 and then aborts)

S_3: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2
(Recoverable; T1 input to T2 but commit before T2)

4.3 Characterizing Schedules based on Recoverability

- **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.

Ex: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2

4.4 Characterizing Schedules based 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. (See S1 schedule)
  - Undoing a Write is simply to restore the BFIM (Before Image)

5.1 Characterizing Schedules based on Serializability

- **Serial schedule** – all the operations of a transaction T are executed consecutively in the schedule; no interleaving, CPU time is wasted

- Non-serial – if the schedule is not serial; interleaving is done

5.2 Characterizing Schedules based on Serializability

Ex: 1. *Serial* 2. *Non-serial*

T1 T2 T1 T2

Read(x); read(x);

X=x-n; x=x-n;

Write(x); read(x);

Read(y); x=x+m;

Y=y+n; write(x);

Write(y); read(y);

Read(x); write(x);

X=x+m; y=y+n;

Write(x); write(y);

5.3 Characterizing Schedules based on Serializability

Ex: 3. *Serial* 4. *Non-serial*

T1 T2 T1 T2

read(x); read(x); x=x+m;
x=x-n; write(x); write(x);
read(x) read(x); x=x-n;
x=x+m; write(x); write(x); read(y);
read(y); y=y+n; y=y+n;
write(y); write(y);

5.4 Characterizing Schedules based on Serializability

- **Serializable schedule** - if the operations are equivalent to some serial schedule (can be made serial)
  - Check examples 1 – 4 with, X=90, Y=90, N=3 and M=2. 1 & 3 produce the same result X=89, Y= 93. 4 gives the same result (serializable) whereas 3 does not (non-serializable)

5.5 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.

5.6 Characterizing Schedules based on Serializability

Test for serializability -Algorithm (Precedence Graph)

1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedent graph.

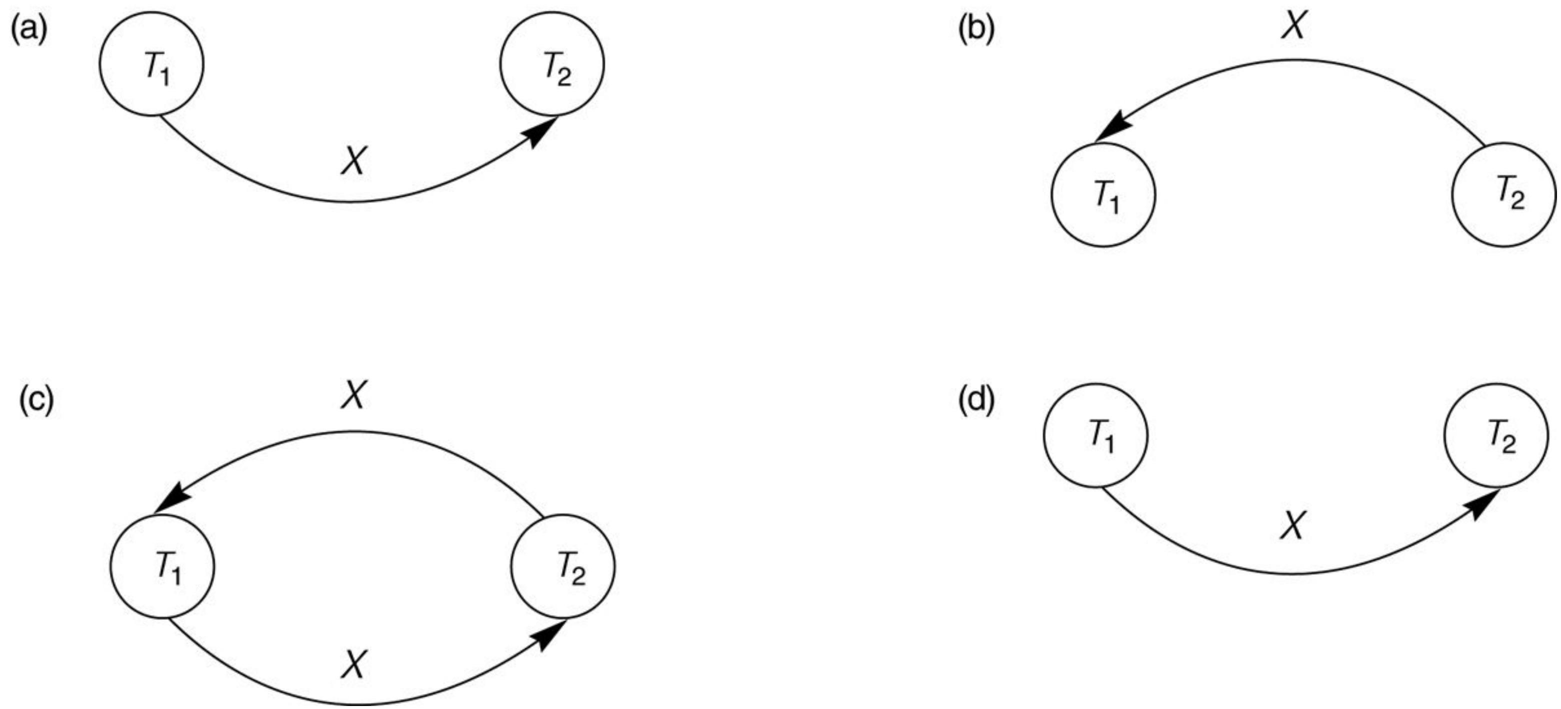
2. For each case in S where Tj executes read_item(x) after Ti executes a write_item(x), create an edge Ti -> Tj in the graph.

3. For each case in S where Tj executes write_item(x) after Ti executes a read_item(x), create an edge Ti -> Tj in the graph.

4. For each case in S where Tj executes write_item(x) after Ti executes a write_item(x), create an edge Ti -> Tj in the graph

5. The schedule S is s serializable if and only if the precedence graph has no cycles.

5.7 Characterizing Schedules based on Serializability



5.9 Characterizing Schedules based on Serializability

(b)

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

Schedule E

5.10 Characterizing Schedules based on Serializability

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

Schedule F