CS3402 Database Systems:
Transactions and Concurrency Control

Introduction to Transaction Processing

- **Single-User System**:
  - At most one user can use the system at one time.
- **Multiuser System**:
  - Many users can access the system concurrently (at the same time).
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

Interleaved vs Parallel

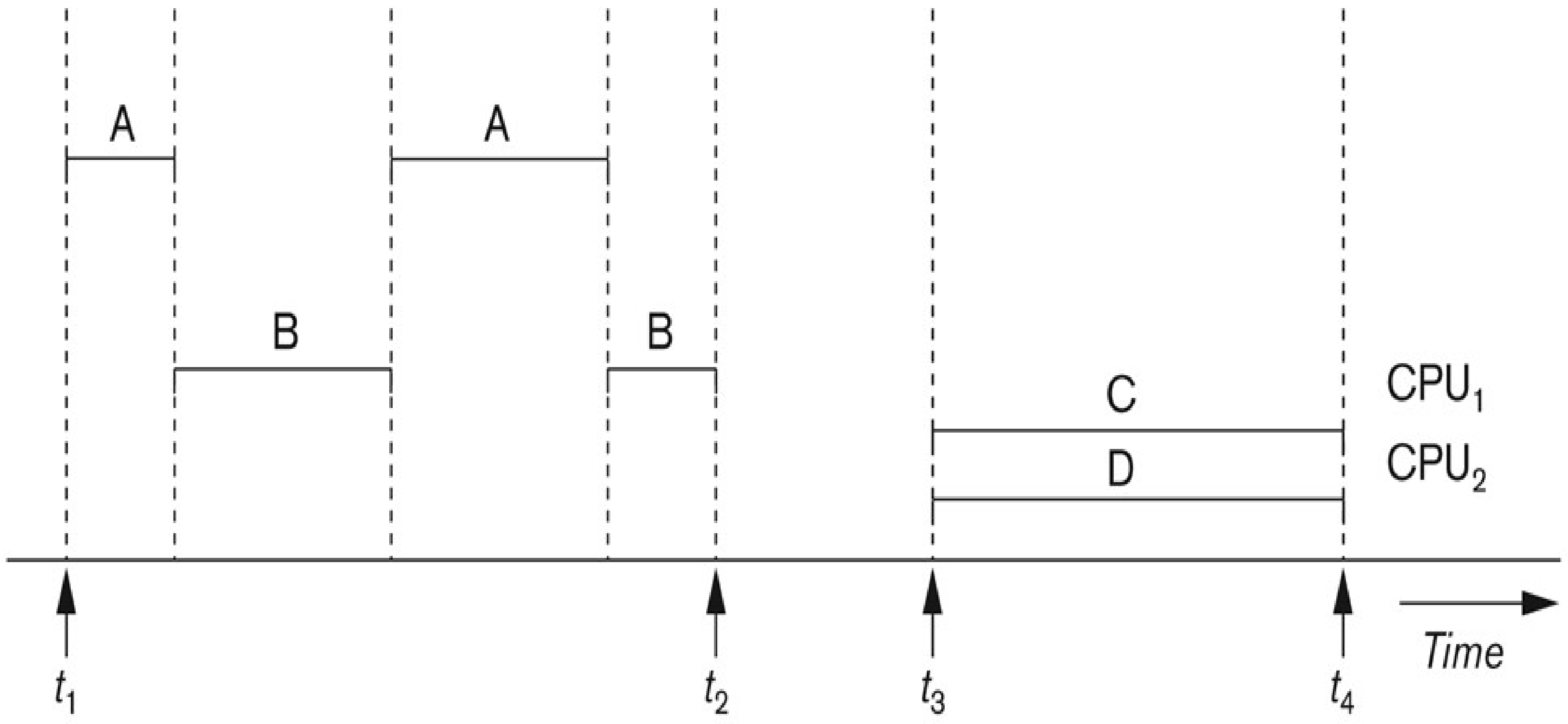


Figure 17.1
Interleaved processing versus parallel processing of concurrent transactions.

What is Transaction

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

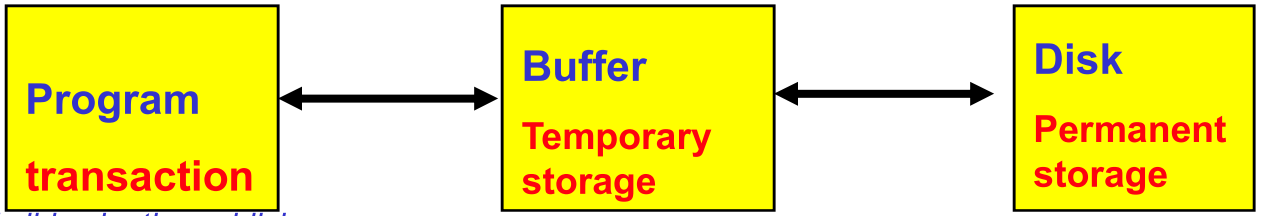
Simple Model of a Database

(for purpose of discussing transactions)

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block
- 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. (X=**read_item(X)**)
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

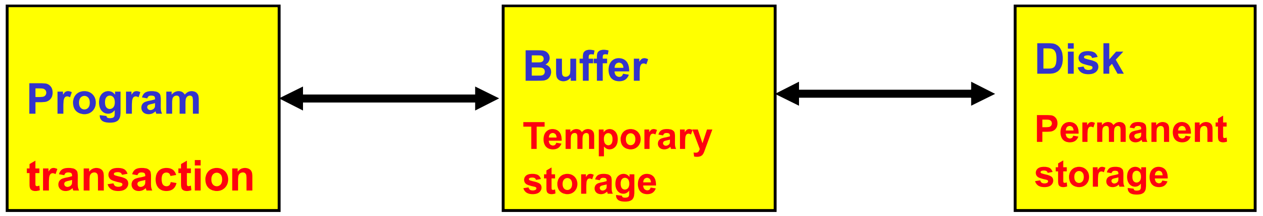
Read and Write Operations

- Basic unit of data transfer from the disk to the computer main memory is one block.
- In general, a data item (what is read or written) will be the field of some record in the database.
- **read_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.



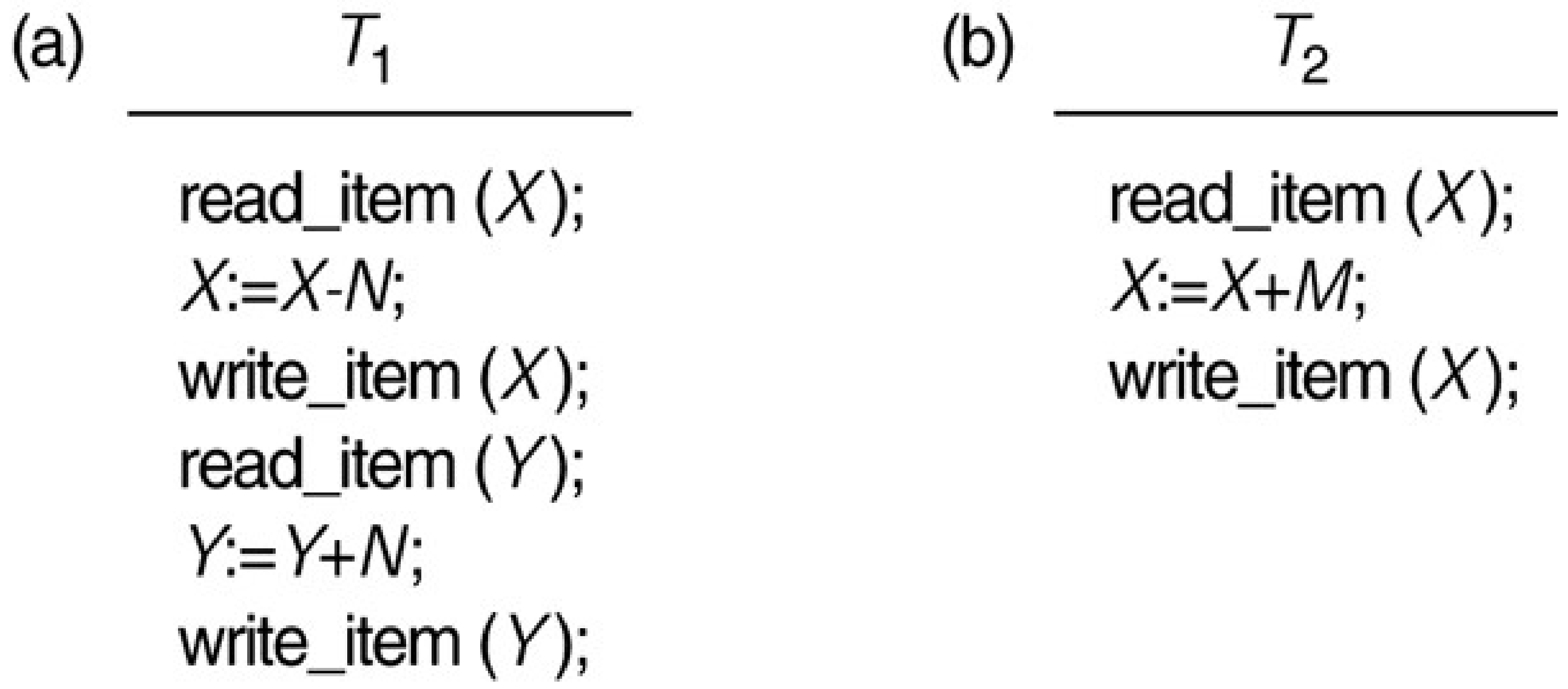
Read and Write Operations

- **write_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).



Two Sample Transactions

- Two sample transactions (flight seat reservation):

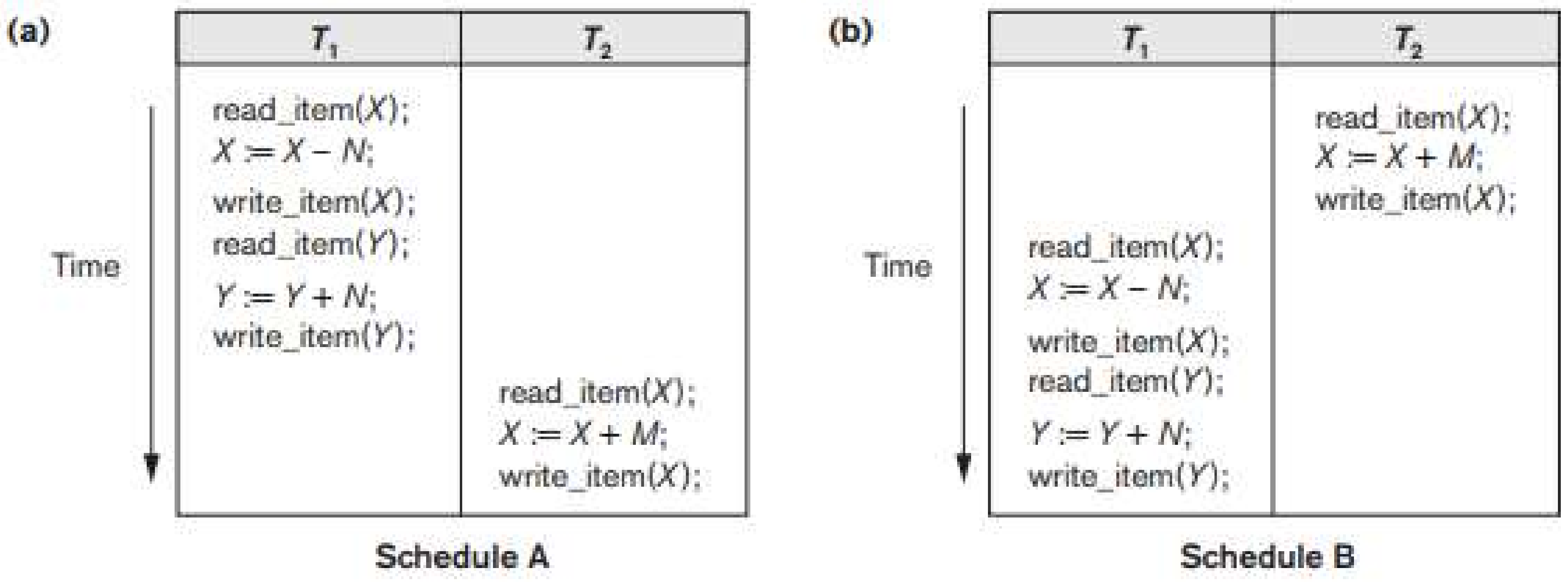


(a) T_1	(b) T_2
read_item (X);	read_item (X);
X:=X-N;	X:=X+M;
write_item (X);	write_item (X);
read_item (Y);	
Y:=Y+N;	
write_item (Y);	

Transaction Schedule

- Transaction schedule
  - The order of execution of operations from the transactions forms a transaction schedule
- A schedule S of *n* transactions *T1, T2, …, Tn* is an ordering of the operations of the transactions subject to the constraint:
  - For each transaction *Ti* that participates in S, the operations of *Ti* in S must appear in the same order as in *Ti* (operations from other transactions *Tj* can be interleaved with the operations of *Ti* in S)
- A serial schedule: a new transaction only starts AFTER the current transaction is completed
- A concurrent schedule: a new transaction starts BEFORE the completion of the current transaction

Serial Schedule



(a)

Time

T_1	T_2
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 A

(b)

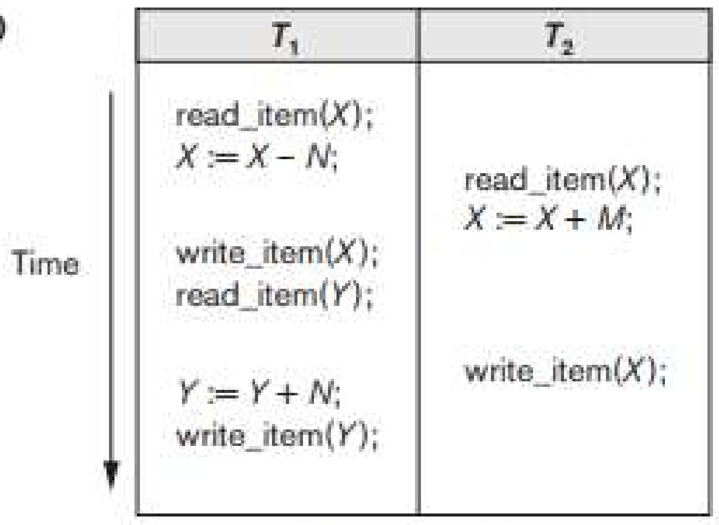
Time

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

Schedule B

Non-Serial (concurrent)Schedule

(c)

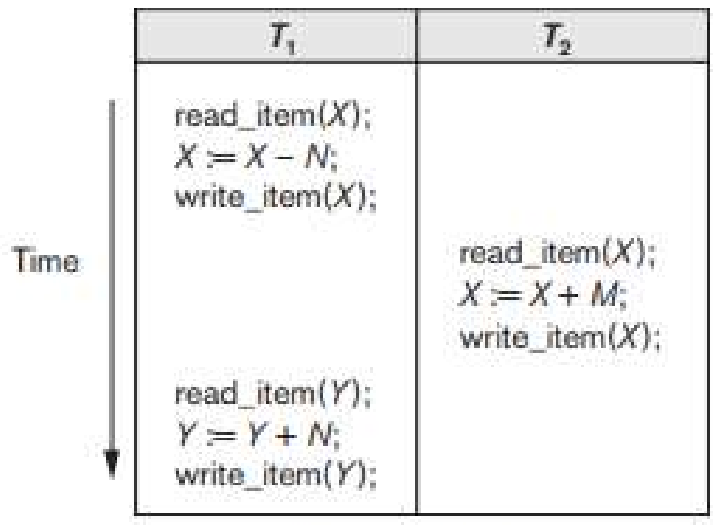


	T_1	T_2
Time ↓	read_item(X); $X := X - N$;	
		read_item(X); $X := X + M$;

	write_item(X);
	read_item(Y);	

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

Schedule C



	T_1	T_2

Time ↓	read_item(X);
	$X := X - N$;
	write_item(X);	

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

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

Schedule D

Problem with Nonserial Schedule

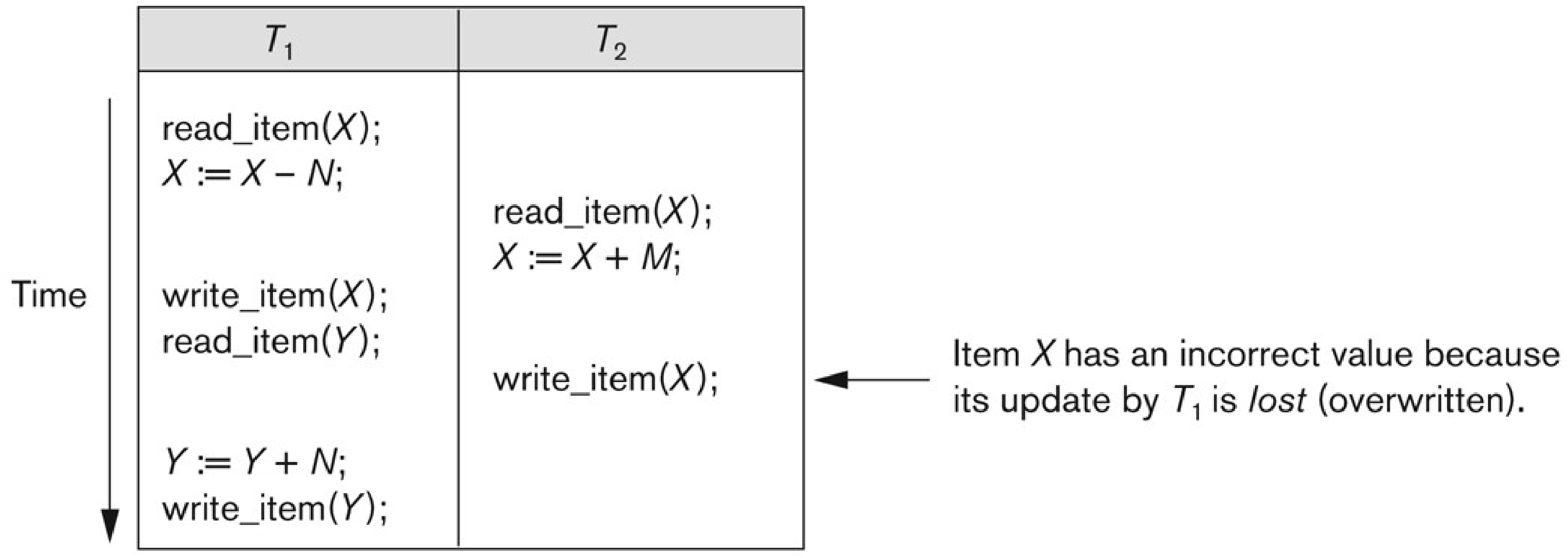
Why Concurrency Control is needed?

- **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.

Concurrent execution is uncontrolled: (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)



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

write_item(X);
read_item(Y);	

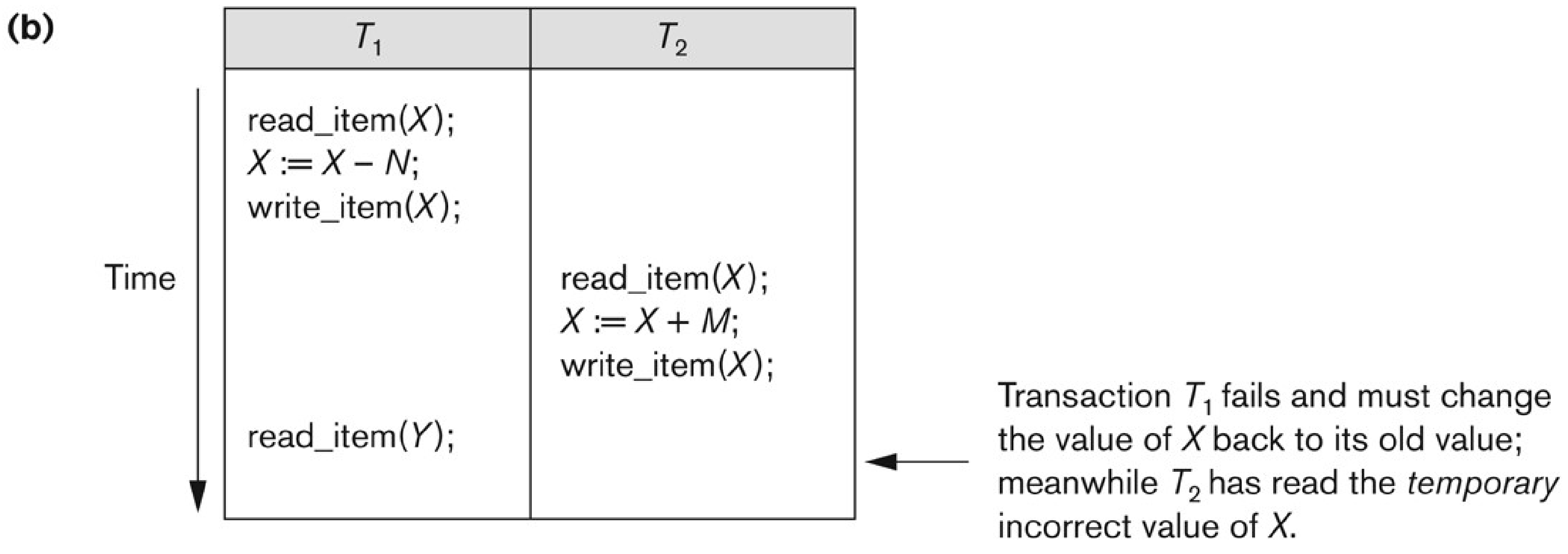
	write_item(X);
$Y := Y + N$; write_item(Y);	

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

Concurrent execution is uncontrolled: (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.

(b)



T_1
T_2
Time
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.

Concurrent execution is uncontrolled: (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
	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).

Schedules Classified 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
  - Serial schedules can maintain the database consistency
    - BUT, poor performance
- Serializable schedule:
  - A concurrent schedule S which is equivalent to a serial schedule
  - Can guarantee the database consistency and can have better performance

Schedules Classified on Serializability

- Being serializable is <u>not</u> 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.

Schedules Classified 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 (RW,WW) 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'.

Conflicting Operations

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a*write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Example 4 (Conflict Equivalent)

S1

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	

S2

T1	T2
Read(A)	
Write(A)	
Read(B)	
	Read(A)

Conflict Equivalent schedules S1 and S2

Can you find an example which is not conflict equivalent?

Schedules Classified on Serializability

Testing for conflict serializability:

- Constructs a *precedence graph* (*serialization graph*) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj:
- W/R conflict: T_i executes write(x) before T_j executes read(x)
- R/W conflict: T_i executes read(x) before T_j executes write(x)
- W/W conflict: T_i executes write(x) before T_j executes write(x)

- The schedule is serializable **if and only if** the precedence graph has no cycles.

(a)

Time ↓

T_1	T_2

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 A

(b)

Time ↓

T_1	T_2

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

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

Schedule B

(c)

Time ↓

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);	

Schedule C

Time ↓

T_1	T_2

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

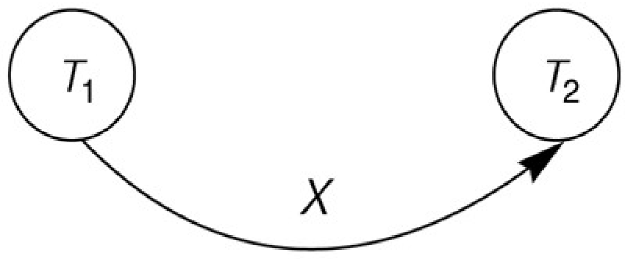
	read_item(X);
	$X := X + M$;
	write_item(X);

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

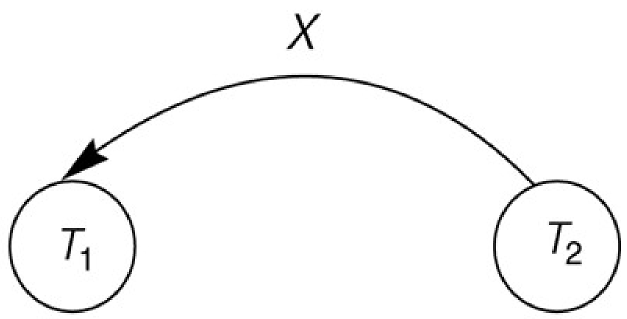
Schedule D

Figure 17.5
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.

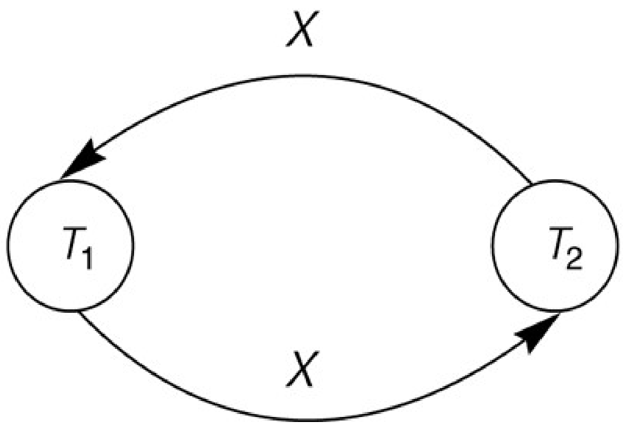
(a)



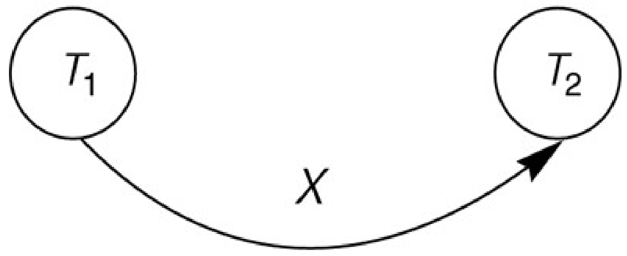
(b)



(c)



(d)



- 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).

How to ensure Recoverability in a schedule?

A Transaction may Fail

Why recovery is needed?

(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. 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.

A Transaction may Fail

Why recovery is needed ?

(What causes a Transaction to fail)

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.

A Transaction may Fail

Why recovery is needed ?

(What causes a Transaction to fail)

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, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

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

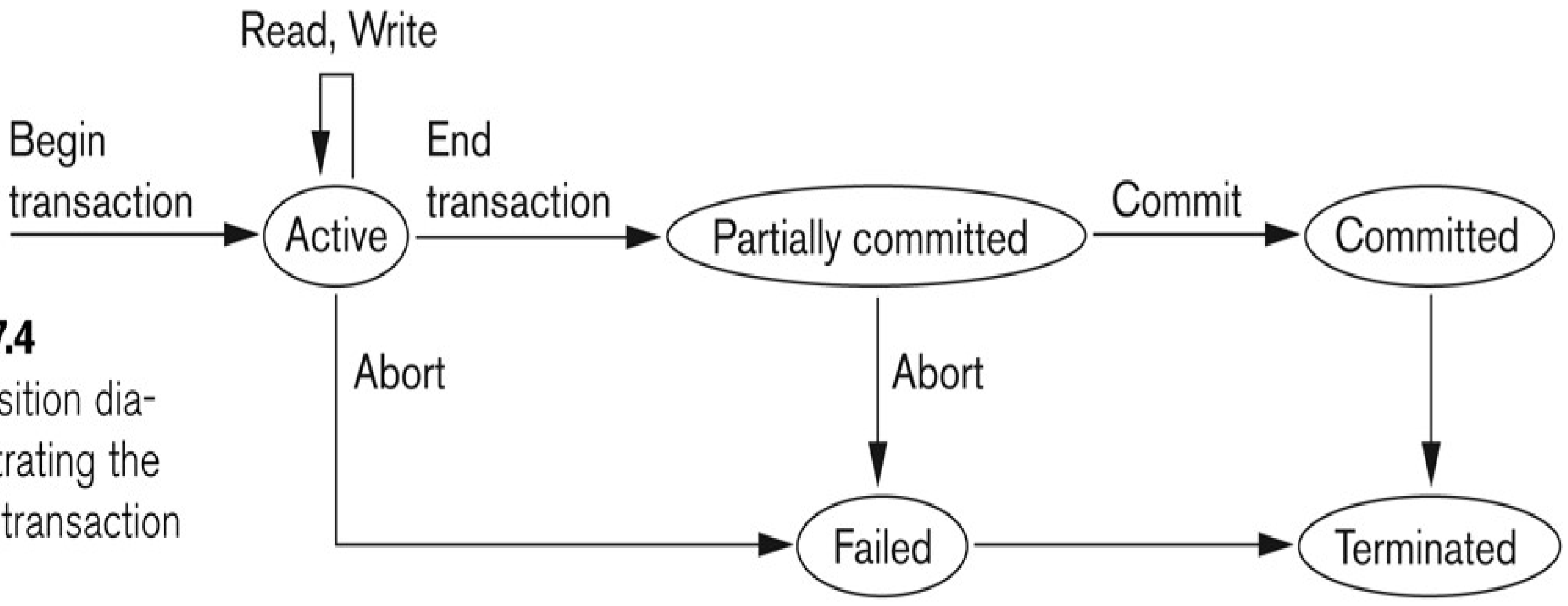


Figure 17.4
State transition diagram illustrating the states for transaction execution.

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:
  - **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

- The System 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 a sequential, append-only file
  - The Log 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.
  - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

Schedules Classified on Recoverability

- **Recoverable schedule**:
    - No committed transaction needs to be rolled back.
- A schedule S is recoverable if
    - 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.
- Cascaded rollback:
    - A single failure leads to a series of rollback
    - All uncommitted transactions that read an item from a failed transaction must be rolled back.

Example 1 (Non-Recoverable)

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Commit/Abort
Read(B)	
Commit/Abort	

Non-recoverable

If T2 commits and then

T1 Aborts…

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
Commit/Abort	
	Commit/Abort

Recoverable

Example 2 (Cascade Rollback)

T1	T2	T3
Read(A)		
Read(B)		
Write(A)		
	Read(A)	
	Write(A)	
		Read(A)
Commit/Abort		
	Commit/Abort	
		Commit/Abort

Recoverable but:

When T1 fails, T2 and T3 should rollback

Schedules Classified on Recoverability

- **Cascadeless schedule:**
  - Every transaction reads only the items that are written by committed transactions.
  - In other words
    - Before Ti reads an item written by Tj
    - Tj is already committed

- **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.

Example 3 (Strict vs Cascadeless)

- S: $w_1(X,5)$; $w_2(X,8)$; a_1 (T1 aborts);
  - Initially, X=9
  - T1 writes a value 5 for X (keeping 9 as “before image”)
  - T2 writes a value 8 for X (keeping 5 as “before image”)
  - Then T1 aborts
  - **Cascadeless**
    - If the system restore according to the “before image” kept in T1, then 9 will be the value for X now, which means, the effect of T2 is lost
  - **Strict**
    - w_2 can not happen until T1 commits

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.

Database Concurrency Control using Two Phase Locking (2PL)

Database Concurrency Control

- Purposes of Concurrency Control
  - To preserve database consistency
    - To ensure all schedules are serializable
  - To maximize the system performance (higher concurrency)

- Example:
  - In concurrent execution environment, if T1 conflicts with T2 over a data item A, then the existing concurrency control decides whether T1 or T2 should get the A and whether the other transaction is rolled-back or waits

Implementation Issue

- Lock Manager:
  - Managing locks on data items
- Lock table:
  - An array to show the lock entry for each data item
  - Each entry of the array stores the identify of transaction that has set a lock on the data item including the mode
- One entry for one database? One record? One field? Lock granularity (Coarse granularity vs. fine granularity)
- Larger granularity -> higher conflict probability but lower locking overhead
- Note: Lock and Unlock are atomic operations

Lock

- Binary Locks
  - two states or values: locked=1 and unlocked=0
  - lock(X): set the value to be 1, item X cannot be accessed by other transaction
  - unlock(X): set the value to be 0, item X can be accessed by other transaction

Implementation Issue

- The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)
     then LOCK (X) ← 1 (*lock the item*)
     else begin
       wait (until LOCK (X) = 0) and
       the lock manager wakes up the transaction);
     goto B
     end;
```

- The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
  wake up one of the waiting transactions;
```

Lock

- Shared/Exclusive (or Read/Write) Locks
  - ◆read_lock(X): set to be the share state, more than one transaction can apply shared lock on X for reading its value but no write lock can be applied on X by any other transaction.
  - ◆write_lock(X): set to be the exclusive state, only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
  - ◆unlock(x): set to be unlock state, item X can be accessed by other transaction

	Read	Write
Read	Y	N
Write	N	N

Implementation Issue

- The following code performs the read lock operation:

```
B: if LOCK (X) = "unlocked" then
   begin LOCK (X) ← "read-locked";
      no_of_reads (X) ← 1;
   end
   else if LOCK (X) = "read-locked" then
            no_of_reads (X) ← no_of_reads (X) +1
        else begin wait (until LOCK (X) = "unlocked" and
          the lock manager wakes up the transaction);
          go to B
        end;
```

Implementation Issue

- The following code performs the write lock operation:

```
B: if LOCK (X) = “unlocked” then
        LOCK (X) ← “write-locked”;
   else begin
        wait (until LOCK(X)=“unlocked”
            and the lock manager wakes up the transaction);
        go to B
        end;
```

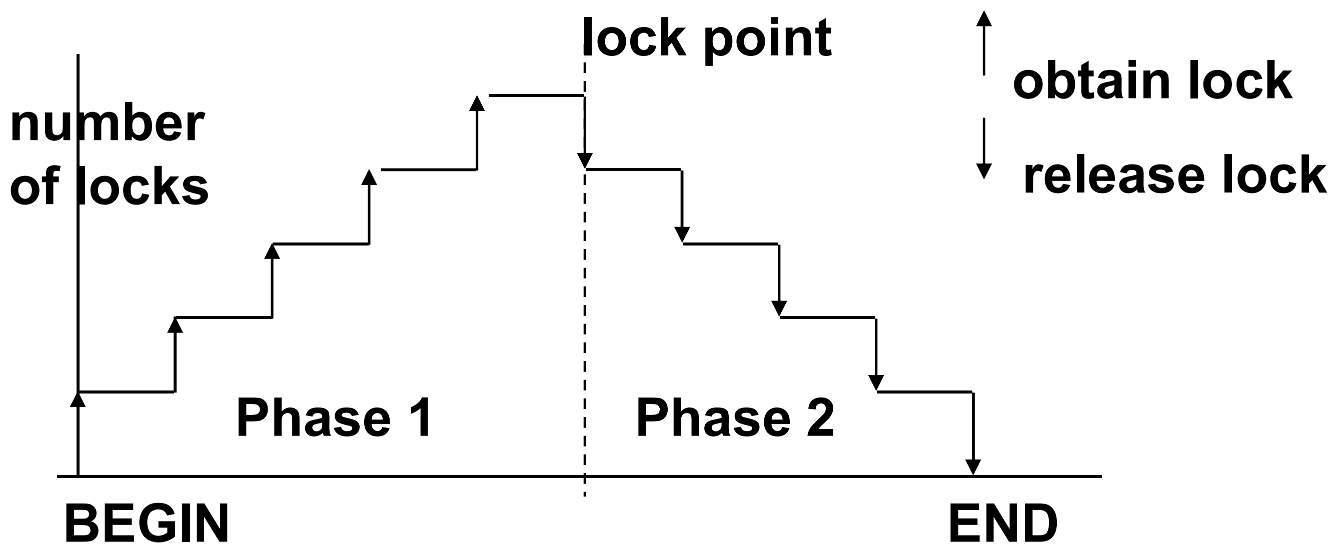
Implementation Issue

- The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
 begin LOCK (X) ← “unlocked”;
     wakes up one of the transactions, if any
 end
 else if LOCK (X) = “read-locked” then
    begin
          no_of_reads (X) ← no_of_reads (X) -1
          if  no_of_reads (X) = 0 then
          begin
       LOCK (X) = “unlocked”;
      wake up one of the transactions, if any
          end
```

Basic Two Phase Locking (B2PL)

- All locking operations precede the first unlock operation in the transaction.
- A transaction can be divided into two phases: growing phase and shrinking phase
- It can guarantee that all pairs of conflicting operations of two transactions are scheduled in the same order.-> guarantee serializability
  - E.g., T1 -> T2 or T2 -> T1 and NO T1<->T2



Basic Two Phase Locking (B2PL)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Non two-phase

T_1'
read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'
read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

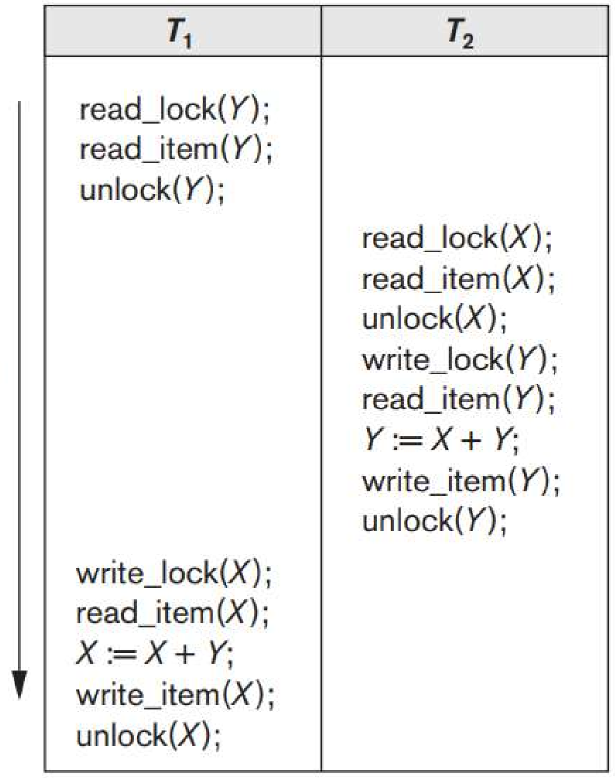
two-phase

Basic Two Phase Locking (B2PL)

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50



T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Figure 21.3
Transactions that do n
(a) Two transactions T
possible serial schedul
nonserializable schedu

Non two-phase

Conservative Two Phase Locking (C2PL)

- requires a transaction to lock all the items it accesses before the transaction begins execution.
- If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking.
- Every time the scheduler releases the locks of a complete transactions, it examines the waiting queue to see if it can grant all the lock requests of any waiting transactions
- In Conservative 2PL, if a transaction T_i is waiting for a lock held by T_j, T_i is holding no locks (no hold and wait situation=> no deadlock)

Conservative Two Phase Locking (C2PL)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a,b)
	Write(a)
WriteLock(a,b) => blocked	
	Write (b)
	Commit
	ReleaseLock(a,b)
WriteLock(a,b)	
Write(a)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Conservative 2PL is a deadlock-free protocol

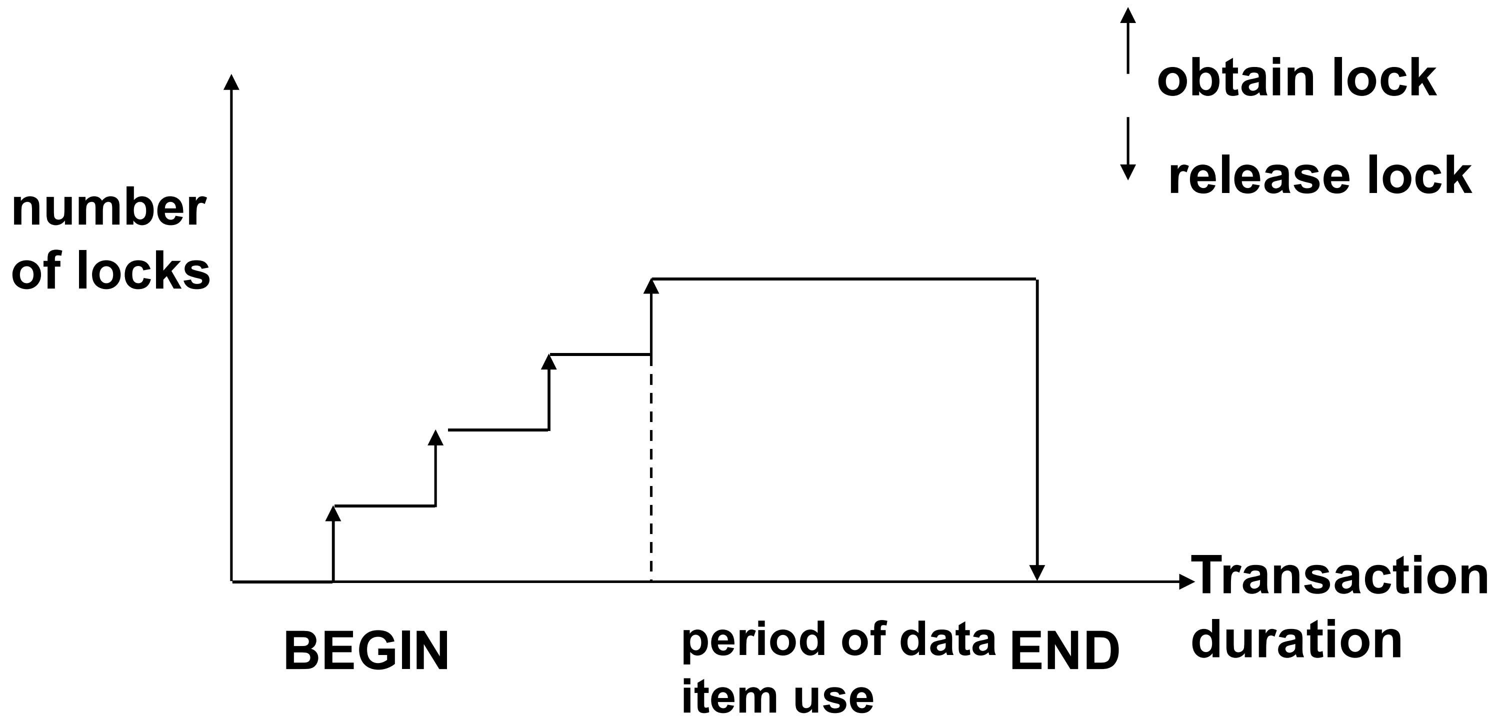
Strict Two Phase Locking (S2PL)

- Requires a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed.

- Compared with B2PL, the lock holding time may be longer and the concurrency may be lower
- Compared with C2PL, the lock holding time may be shorter and the concurrency may be higher

- It may have the problem of deadlock but all schedule is recoverable

Strict Two Phase Locking (S2PL)



Strict Two Phase Locking (S2PL)

T_1	T_2
	Write(a)
Write(a)	
	Write (b)
	Commit
Write(b)	
Commit	

T_1	T_2
	WriteLock(a)
	Write(a)
WriteLock(a) => blocked	
	WriteLock(b)
	Write (b)
	Commit
	ReleaseLock(a,b)
WriteLock(a)	
Write(a)	
WriteLock(b)	
Write(b)	
Commit	
ReleaseLock(a,b)	

Performance Issues

- S2PL is better than C2PL when the transaction workload is not heavy since the lock holding time is shorter in S2PL

- When the transaction is heavy, C2PL is better than S2PL since deadlock may occur in S2PL.

Deadlock

T1	T2
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (Y);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 do follow two-phase policy but they are deadlock

- Deadlock (T1 and T2)

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T′ in the set.

Deadlock Prevention

- Deadlock condition
  - Hold and wait
  - Cyclic wait

- Deadlock Prevention

  1 A transaction locks all data items it refers to before it begins execution (C2PL). This way of locking prevents deadlock since a transaction never waits for a data item

  2 Ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order

Deadlock Prevention

- 3 Timestamp: Each transaction is assigned a unique time-stamp, e.g., its creation time (distributed dbs: creation time + site ID)
- The timestamps are typically based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2).
- Two Schemes preventing deadlock: (Suppose transaction Ti tries to lock an item X, but X is locked by Tj)
- Wait-die Rule (non-preemptive): If TS(Ti) < TS(Tj) (Ti older than Tj), Ti waits else Ti dies and restart it later with the same timestamp
- Wound-Wait Rule (preemptive): If TS(Ti) < TS(Tj), abort Tj (Ti wounds Tj) and restart it later with the same timestamp, else Ti waits

Deadlock Avoidance using TS

- If TS(Ti) < TS(Tj), Ti waits else Ti dies (Wait-die)
- If TS(Ti) < TS(Tj), Tj wounds else Ti waits (Wound-wait)
- Note a smaller TS means the transaction is older
- Note both methods restart the younger transaction
- Both methods prevent cyclic wait:
  - Consider this deadlock cycle: T1->T2->T3->…->Tn->T1
  - It is impossible since if T1 …->Tn, then Tn is not allowed to wait for T1
  - Wait-die: Older transaction is allowed to wait
  - Wound-wait: Older transaction is allowed to get the lock

Deadlock Example

Transaction U: TS of U < TS of T	Transaction T:
Read (A)	
Write (B)	
	Read (C)
	Write (A) (blocked)
Write (C) (blocked) deadlock formed	

Deadlock Example (wait-die)

Transaction U: TS of U < TS of T	Transaction T:
ReadLock (A); Read (A)	
WriteLock(B); Write (B)	
	ReadLock(C): Read (C)
	Write (A) (restarts)
	*** T is restarted since it is younger than U
	*** T releases its read lock on C before restart
WriteLock(C); Write (C)	
ReleaseLock(U)	
	ReadLock(C); Read(C)
	….

Deadlock Example (wound-wait)

Transaction U: TS of U < TS of T	Transaction T:
Read (A)	
Write (B)	
	Read (C)
	Write (A) (blocked)
	*** since T is younger than U
Write (C)	
*** T is restarted by U since T is younger than U	
*** The write lock on C is granted to U after T has released its read lock on C	WriteLock(A); Write(A)
	….

Deadlock detection and resolution

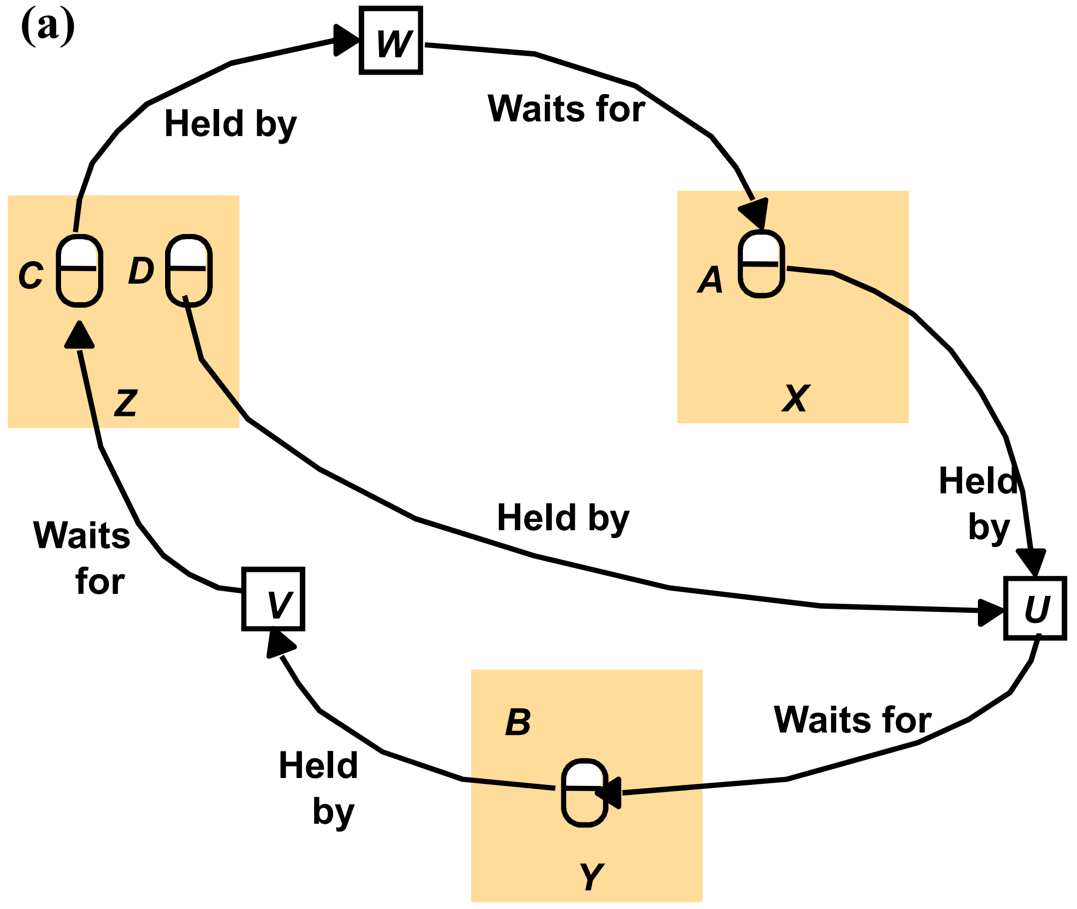
- Deadlock Detection
  - In this approach, deadlocks are allowed to happen e.g., in Strict 2PL. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj, Tj waits for Tk and Tk waits for Ti occurs, this creates a cycle. One of the transactions will be chosen to abort.

Interleaving of Transactions U, V and W

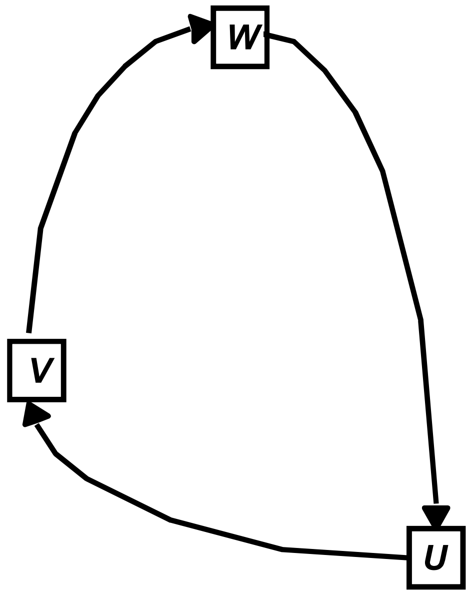
U		*V*		*W*	
Write(D)	lock *D*				
		Write(B)	lock *B* at *Y*		
Write(A)	lock *A* at *X*				
				Write(C)	lock *C* at *Z*
Write(B)	wait at *Y*				
		Write(C)	wait at *Z*		
				Write(A)	wait at *X*

Distributed Deadlock

(a)



(b
)



Starvation

- Starvation
  - Starvation occurs when a particular transaction consistently waits or restarts and never gets a chance to proceed further
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled back

Learning Objectives

1. Know the desirable properties of transactions
2. Understand the difference among schedules based on **Recoverability** (non-recoverable, recoverable, cascadeless, and strict)
3. Understand the difference among schedules based on **Serializability** (conflict serializable and non-serializable)
4. Know the purpose of concurrency control and able to judge whether a schedule satisfies Two-Phase locking (2PL)!