Transaction Processing Concepts

政治大學
資訊科學系
沈錳坤

中國信託
Chinatrust

自動櫃員機交易明細單
ATM Transaction(TX) Receipt

交易日 Date	時間 Time	機號 Atm No.	交易類別 TX Type
09 /11/18	11:22	04953095	CW7

交易序號 TX No.	發卡行編號 Isu Bank No.	帳號／卡號 Account No.
455288		498022******0028

交易金額 TX Amount	轉入帳號 Trans for Act No.
10,000	

帳戶餘額 Account Balance	可用餘額 Available Balance
179,818,931	

訊息代號 MSG Code	手續費 Fee	發鈔張數 Cash detail 千元	佰元
OK	100	10	00

（幣別：新台幣）

授權碼	
002316	交易代號：CWD取款．TWD轉帳 INQ餘額查詢

Transaction Processing System

- System with large DB & hundreds of concurrent users that are executing DB transactions
- Require high availability & fast response time for hundreds of concurrent users

Transaction

- Transaction
  - A logical unit of DB processing
  - Each transaction includes one or more DB access operations (insertion, deletion, modification, retrieval operations)
  - e.g. 轉帳: 從帳號X轉帳N元到帳號Y

```
read(X);
X=X-N;
write(X);
read(Y);
Y=Y+N;
write(Y);
```

Interleaved Processing

- Modern computer
  - time-sharing system
  - Interleaved processing

read(X);
X=X-N;
write(X);
read(Y);
Y=Y+N;
write(Y);
 	
read(X);
X=X+M;
write(X);
	Serial processing	read(X);
		X=X-N;
		read(X);
		X=X+M;
		write(X);
		read(Y);
		write(X);
		Y=Y+N;
		write(Y);
			Interleaved processing

Transaction Processing

- Concurrency control
- Recovery

Concurrency Problem: Why Concurrency Control Is needed

Concurrency Problems

- Types of concurrency problems
  - Lost update
  - Temporary update (dirty read): uncommitted dependency
  - Incorrect summary
  - Unrepeatable read

Lost Update Problem

X=1000, Y=200
X transfers N=500 to Y,
M=300 is transferred to X

X=1000-500+300=800
Y=200+500=700

	T1	T2	
	read(X);		
X'=500	X=X-N;		
		read(X);	X'=1000
		X=X+M;	X'=1300
X=500	write(X);		
Y'=200	read(Y);		
		write(X);	X=1300
Y'=700	Y=Y+N;		
Y=700	write(Y);		

Temporary Update Problem

X=1000, Y=200
X transfers N=500 to Y
M=300 is transferred to X

X=1000-500+300=800
Y=200+500=700

	T1	T2	
	read(X);		
X'=500	X=X-N;		
X=500	write(X);		
		read(X);	X'=500
		X=X+M;	
		write(X);	X=800
Y=200	read(Y);		
	fails		

Incorrect Summary Problem

X=1000, Y=200
X transfers N=500 to Y
X+Y=1200

	T1	T2	
		sum=0	sum=0
X'=1000	read(X);		
	X=X-N;		
X=500	write(X);		
		read(X);	
		sum=sum+X	sum=500
		read(Y)	
		sum=sum+Y	sum=700
	read(Y);		
	Y=Y+N;		
	write(Y);		

Unrepeated Read Problem

- A transaction *T* reads an item twice & the item is changed by another transaction *T'* between the two reads
- *T* receives different values for its two reads of the same item

Solution for Concurrency Control Problems

Critical Section

- Locking techniques

交易1

X'=1000	writelock(X)
X'= 500	read(X);
X = 500	X=X-N;
	write(X)
	unlock(X)

X=1000, Y=200
X transfers N=500 to Y,
M=300 is transferred to Y

X=1000-500+300=800
Y=200+500=700

交易二

	writelock(X)
X'=500	read(X);
X'=800	X=X+M;
X=800	write(X);
	unlock(X)

交易1 (continued)

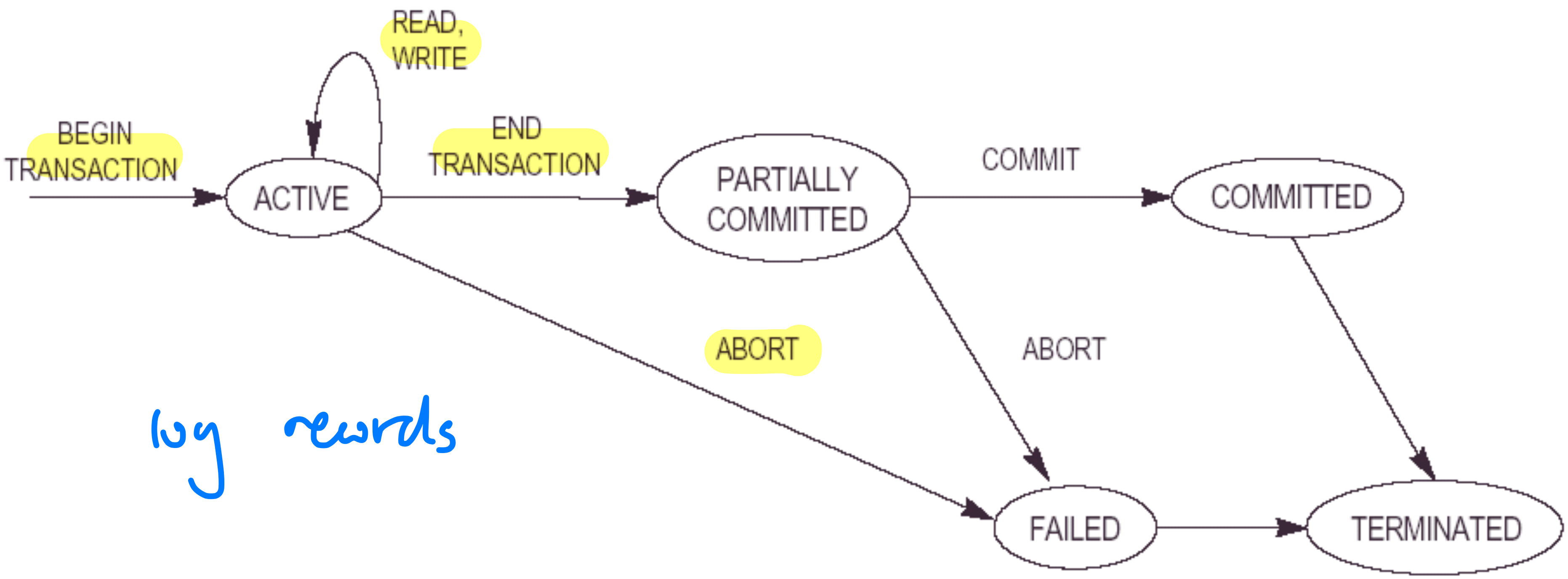
	writelock(X)
Y'=200	read(Y);
Y'=700	Y=Y+N;
Y=500	write(Y);
	unlock(X)

Recovery Problem: Why Recovery Is needed

Types of Failures

- Computer failure (system crash)
- Disk failure
- Transaction or system error (overflow, interrupt)
- Local errors or exception conditions detected by the transaction (e.g. data not found)
- Concurrency control enforcement (deadlock)
- Physical problems

Transaction States



log records

System Log

- To be recovered from failures that affect transactions, system maintains a log
- Log keep track of all transaction operations that affect the values of DB items
- Log is kept on disk
- Log is not affected by any type of failure except for disk failure or catastrophe
- Log is periodically backed up to archival storage

(e.g. magnetic tape)

System Log (cont.)

- Log records
  - start transaction
  - transaction has changed value of DB item X
  - transaction has read the value of DB item X
  - transaction has completed successfully & committed to the DB
  - transaction has been aborted
- Undo, Redo
- Force-writing
  - 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

ACID Properties of Transactions

- Atomicity
  - A transaction is an atomic unit of processing
  - It's either performed in its entirety or not performed at all "all or nothing"
  - Responsibility of transaction recovery subsystem
- Consistency preservation
  - The complete execution takes the DB from one consistent state to another
  - Responsibility of DB programmer or integrity constraint of DBMS

ACID Properties of Transactions (cont.)

- Isolation
  - A transaction should appear as though it's being executed in isolation from other transactions
  - Responsibility of concurrency control subsystem
- Durability (permanency)
  - Changes applied to DB must not be lost because of any failure
  - Responsibility of transaction recovery subsystem

Schedules & Recoverability

Interleaved Processing

- Modern computer
  - time-sharing system
  - Interleaved processing

read(X);
X=X-N;
write(X);
read(Y);
Y=Y+N;
write(Y);

read(X);
X=X+M;
write(X);

Serial processing

read(X);
X=X-N;
read(X);
X=X+M;
write(X);
read(Y);
write(X);
Y=Y+N;
write(Y);

Interleaved processing

Schedule of Transactions

- A schedule S of n transactions T_1, T_2, …, T_n is
  – An ordering of the operations of the transactions
  – Subject to the constraint that: for each transaction T_i, the operations of T_i in S must appear in the same order in which they occur in T_i
  – However, operations from other transactions T_j can be interleaved with the operations of T_i in S
  – Partial ordering
    - a partial order on a set: an arrangement such that, for some pairs of elements, one precedes the other.

discrete math

Schedules of Transactions (cont.)

- For the purpose of recovery & concurrency control, only
  - read_item (r)
  - write_item (w)
  - commit (c)
  - abort (a)

  operations are considered

Conflict Operations

- Two operations in a schedule are conflict if
  - They belong to different transactions
  - They access the same item x
  - At least one of the operations is a write_item(x)
- E.g.
  - S_a: r1(x); r2(x);w1(x);r1(y);w2(x);w1(y)
  - r1(x) & w2(x) conflict
  - r2(x) & w1(x) conflict
  - w2(x) & w1(y) do not conflict (different items)
  - r1(x) & w1(x) do not conflict (same transactions)

Complete schedule

- A schedule S of n transactions T_1, T_2, …, T_n is said to be a complete schedule if
  - All operations in the transactions must appear in the complete schedule including a commit or abort operations as the last operations for each transaction in the schedule
  - For any pair of operations from the same transaction T_i , their order of appearance in S is the same as their order of appearance in T_i
  - For any two conflicting operations, one of the two must occur before the other in the schedule

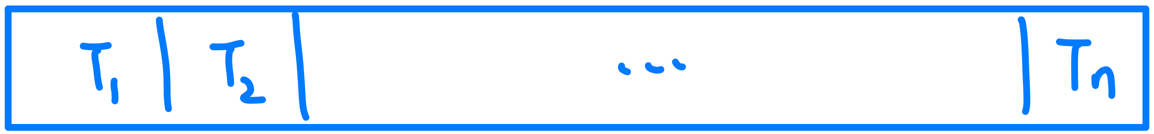
Recoverable Schedule

- Recoverable schedule: once a transaction is committed, it should never be necessary to roll back.
- A schedule *S* is recoverable if   not recoverable
  - No transaction *T* in *S* commits until   One   before
    all transactions *T'* (that have written an item that *T* reads) have committed
- E.g.
  - r1(x);w1(x);r2(x);r1(y);w2(x);c2;a1; not recoverable   T = 2, T' = {1}
  - r1(x);r2(x);w1(x);r1(y);w2(x);c2;w1(y);c1; recoverable
  - r1(x);w1(x);r2(x);r1(y);w2(x);w1(y);c1;c2; recoverable
  - r1(x);w1(x);r2(x);r1(y);w2(x);w1(y);a1;a2;

Serializability of Schedules

Serial Schedules

- A schedule *S* is serial
  - if every transaction *T* in *S*, all the operations of *T* are executed consecutively in the schedule
  - Otherwise, non-serial
- In a serial schedule
  - only one transaction at a time is active
  - The commit (or abort) of the active transaction initiates execution of the next transaction



(a)

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);

Time

serial

Schedule A

(b)

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);	

Time

Schedule B

serial

(c)

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);	

Time

not serial

Schedule C

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

not serial

Serializable

- A schedule *S* of *n* transactions is serializable if it is equivalent to some serial schedule of the same *n* transactions
- If a non-serial schedule *S* is serializable -> it is correct

Conflict Equivalent

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules
- If two conflicting operations are applied in different orders in two schedules, the effect can be different

≡ one can be transformed to another by swapping consecutive non-conflicting operations

Conflict Operations

- Two operations in a schedule are conflict if
  - They belong to different transactions
  - They access the same item X
  - At least one of the operations is a write_item(x)
- E.g.
  - S_a: r1(x); r2(x);w1(x);r1(y);w2(x);w1(y)
  - r1(x) & w2(x) conflict
  - r2(x) & w1(x) conflict
  - w2(x) & w1(y) do not conflict (different items)
  - r1(x) & w1(x) do not conflict (same transactions)

Conflict Serializable

- A schedule S is conflict serializable
  if it is conflict equivalent to
  some serial schedule S'

☆ Conflict serializability is a conservative test for serializability. (may give some false negatives)

(a)

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);

Time

(serial) schedule A

(b)

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);	

Time

(serial) schedule B

(c)

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);	

Time

(non-serializable) schedule C

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);	

(serializable) schedule D

Testing for Conflict Serializability

- There exists a simple algorithm to test for conflict serializability
- However
  – most concurrency control methods do not actually test for serializability
  – Protocols or rules are developed to guarantee that a schedule is serializability (e.g. two phase-locking)

Testing for Conflict Serializability (cont.)

- Algorithm to test for conflict serializability
  - precedence graph
    - *A* directed graph $G = (N, E)$
    - $N = \{T_1, T_2, \ldots, T_n\}$, one node for each transaction T_i in the schedule
    - $E = \{T_j \rightarrow T_k \mid$ one of operations in T_j appears in the schedule before some conflicting operation in $T_k\}$

- Algorithm

conflicting operations

1. construct each node in G for each transaction T_i
2. create an edge $(T_i \rightarrow T_j)$ in G if $w_i(x)$ appears before $r_j(x)$ in S
3. create an edge $(T_i \rightarrow T_j)$ in G if $r_i(x)$ appears before $w_j(x)$ in S
4. create an edge $(T_i \rightarrow T_j)$ in G if $w_i(x)$ appears before $w_j(x)$ in S
5. The schedule S is (conflict) serializable if and only if G has no cycles

Testing for Conflict Serializability (cont.)

(acyclic)

- If there is no cycle in the precedence graph, we can create an equivalent serial schedule S' that is equivalent to S

(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);

(serial) 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);	

(serial) schedule B

(c)

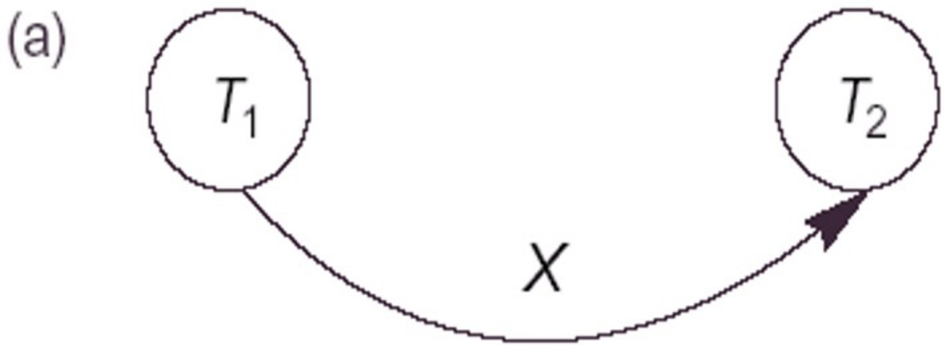
Time

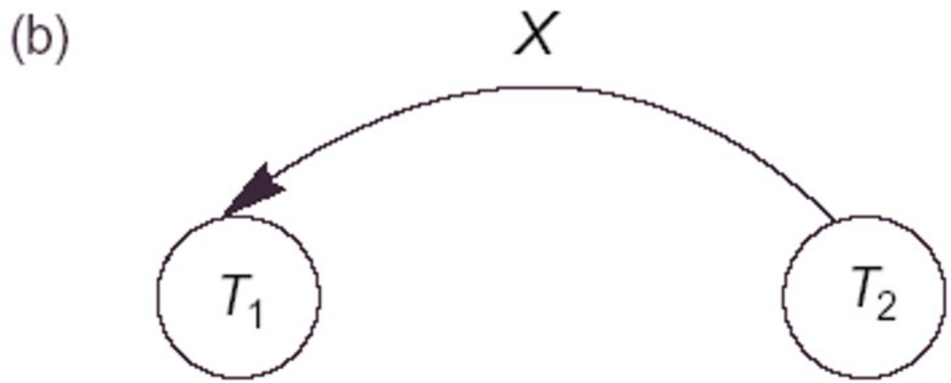
T_1	T_2
read_item(X);	
X:=X-N;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	

(non-serializable) schedule C

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);	

(serializable) schedule D





(serial) **schedule A**

(serial) **schedule B**

(a)

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);

Time

Schedule A

(b)

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);	

Time

Schedule B

(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);

(serial) 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);	

(serial) schedule B

(c)

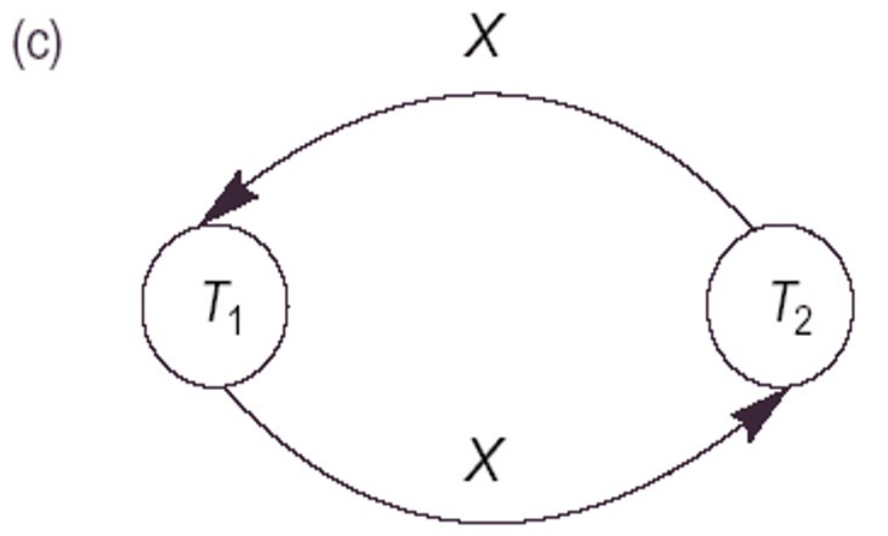
Time ↓

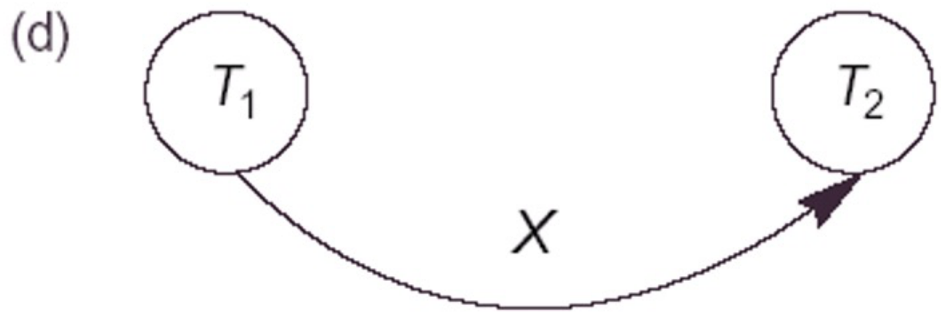
T_1	T_2
read_item(X);	
X:=X-N;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	

(non-serializable) schedule C

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);	

(serializable) schedule D





(non-serializable) **schedule C**

(serializable) **schedule D**

(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

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

(a)

transaction T_1
read_item (X); write_item (X); read_item (Y); write_item (Y);

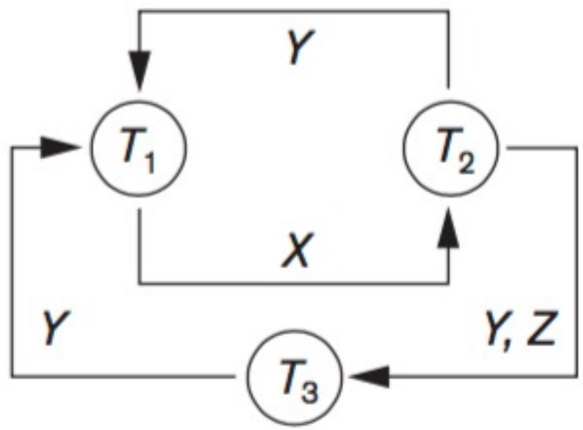
transaction T_2
read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);

transaction T_3
read_item (Y); read_item (Z); write_item (Y); write_item (Z);

(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



Schedule E

(a)

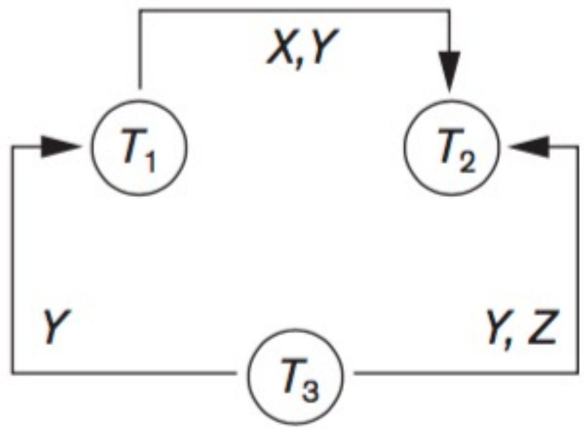
transaction T_1	transaction T_2	transaction T_3
read_item (X); write_item (X); read_item (Y); write_item (Y);	read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);	read_item (Y); read_item (Z); write_item (Y); write_item (Z);

(c)

Time

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



Schedule F

Uses of Serializability

- Serial schedule
  - inefficient processing (no interleaving)
- Serializable schedule
  - In practice, interleaving is determined by O.S.scheduler
  - It is difficult to determine how operations of a schedule will be interleaved beforehand to ensure serializability
  - Most practical DBMS use methods that ensure serializability, without having to test the schedules

Uses of Serializability (cont.) *

- Protocols used in practical DBMS
  - Two-phase locking (2PL)
    - Locking data items to prevent concurrent transactions from interleaving with one another
    - Enforcing additional condition that guarantees serializability
    - Used in most DBMS
  - Timestamp ordering
    - Each transaction is assigned a unique timestamp
    - Protocol ensures that any conflicting operations are executed in the order of the transaction timestamps
  - Multiversion protocol
    - Maintain multiple versions of data items
  - Optimistic protocol
    - Check for possible serializability violations after transactions terminate but before they are permitted to commit

Concurrency Control Technique:
Two Phase Locking

Locks

- Lock
  - a variable associated with a data item that
    describes the status of the item
    with respect to possible operations that can be applied to it.
  - there is one lock for each data item in the DB.
  - Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Binary Locks

- A binary lock
  - have two states or values: locked and unlocked (1 and 0).
  - If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
  - If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1.
  - a binary lock enforces mutual exclusion on the data item
  - lock(X): current state of the lock associated with item X
- Two operations
  - lock_item
  - unlock_item.

```
lock_item(X):
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);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                      (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

* lock_item and unlock_item operations must be implemented as indivisible units (known as critical sections in operating systems)

Read/Write Locks

- 3 locking operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X).
- Lock(X), has three possible states
  - read-locked
  - write-locked
  - unlocked.
- read-locked = share-locked, because other transactions are allowed to read the item,
- write-locked = exclusive-locked, because a single transaction exclusively holds the lock on the item.

Read/Write Locks (cont.)

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*.
2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T*.
3. A transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.
4. A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. (This rule may be relaxed.)
5. A transaction *T* will not issue a write_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*. (This rule may also be relaxed.)
6. A transaction *T* will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

```
read_lock(X):
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;
```

```
write_lock(X):
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;
```

```
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it 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";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Lock Conversion

- A transaction that already holds a lock on item *X* is allowed under certain conditions to convert the lock from one locked state to another.
- For example, it is possible for a transaction *T* to issue a read_lock(*X*) and then later to upgrade the lock by issuing a write_lock(*X*) operation.
- If *T* is the only transaction holding a read lock on *X* at the time it issues the write_lock(*X*) operation, the lock can be upgraded; otherwise, the transaction must wait.
- It is also possible for a transaction *T* to issue a write_lock(*X*) and then later to downgrade the lock by issuing a read_lock(*X*) operation.

Two Phase Locking

- A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- Two phases
  1. Expanding or growing phase: new locks on items can be acquired but none can be released
  2. Shrinking phase: existing locks can be released but no new locks can be acquired.
- It can be proved that, if every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedules.

X=20, Y=30

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);

T1, T2: X=50, Y=80

T2, T1: Y=50, X=70

(c)

Time

	T_1	T_2	
	read_lock(Y);		
Y'=30	read_item(Y);		
	unlock(Y);		
		read_lock(X);	
		read_item(X);	X'=20
		unlock(X);	
		write_lock(Y);	
		read_item(Y);	Y'=30
		$Y := X + Y$;	Y= 50
		write_item(Y);	
		unlock(Y);	
	write_lock(X);		
X'=20	read_item(X);		
X'=50	$X := X + Y$;		
X =50	write_item(X);		
	unlock(X);		

T_1	T_2

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

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);

Transaction Support in SQL

```
EXEC SQL SET Transaction
          Read Write
          Diagnostics size 5
          Isolation Level Serializable;
EXEC SQL Insert into Employee …;
EXEC SQL Update Employee…;
EXEC SQL Commit;
Goto The_End;
Undo; EXEC SQL Rollback
The_End
```

Conclusions

- Concurrent transactions
- Concurrency Problems
- Recovery Problems
- Transaction State
- ACID Properties
- Recoverable schedules
- Serializable schedules
- 2 Phase Locking