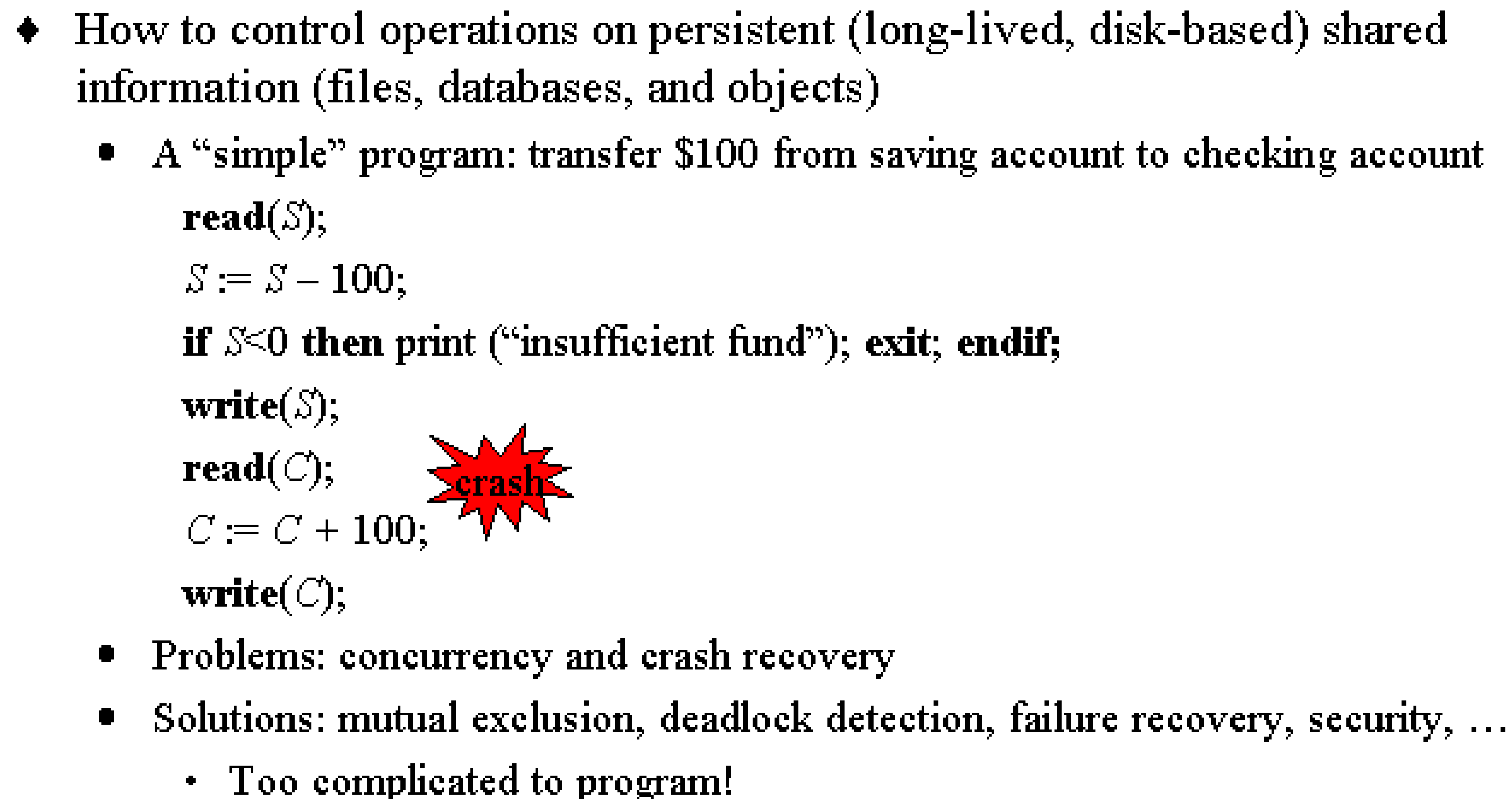
Why Do We Need Transaction Processing?



- How to control operations on persistent (long-lived, disk-based) shared information (files, databases, and objects)
  - A “simple” program: transfer $100 from saving account to checking account

    read(S);

    $S := S - 100$;

    if S<0 **then** print (“insufficient fund”); **exit**; **endif;**

    write(S);

    read(C);

    $C := C + 100$;

    write(C);
  - Problems: concurrency and crash recovery
  - Solutions: mutual exclusion, deadlock detection, failure recovery, security, …
    - Too complicated to program!

Why Do We Need Transaction Processing? (Cont.)

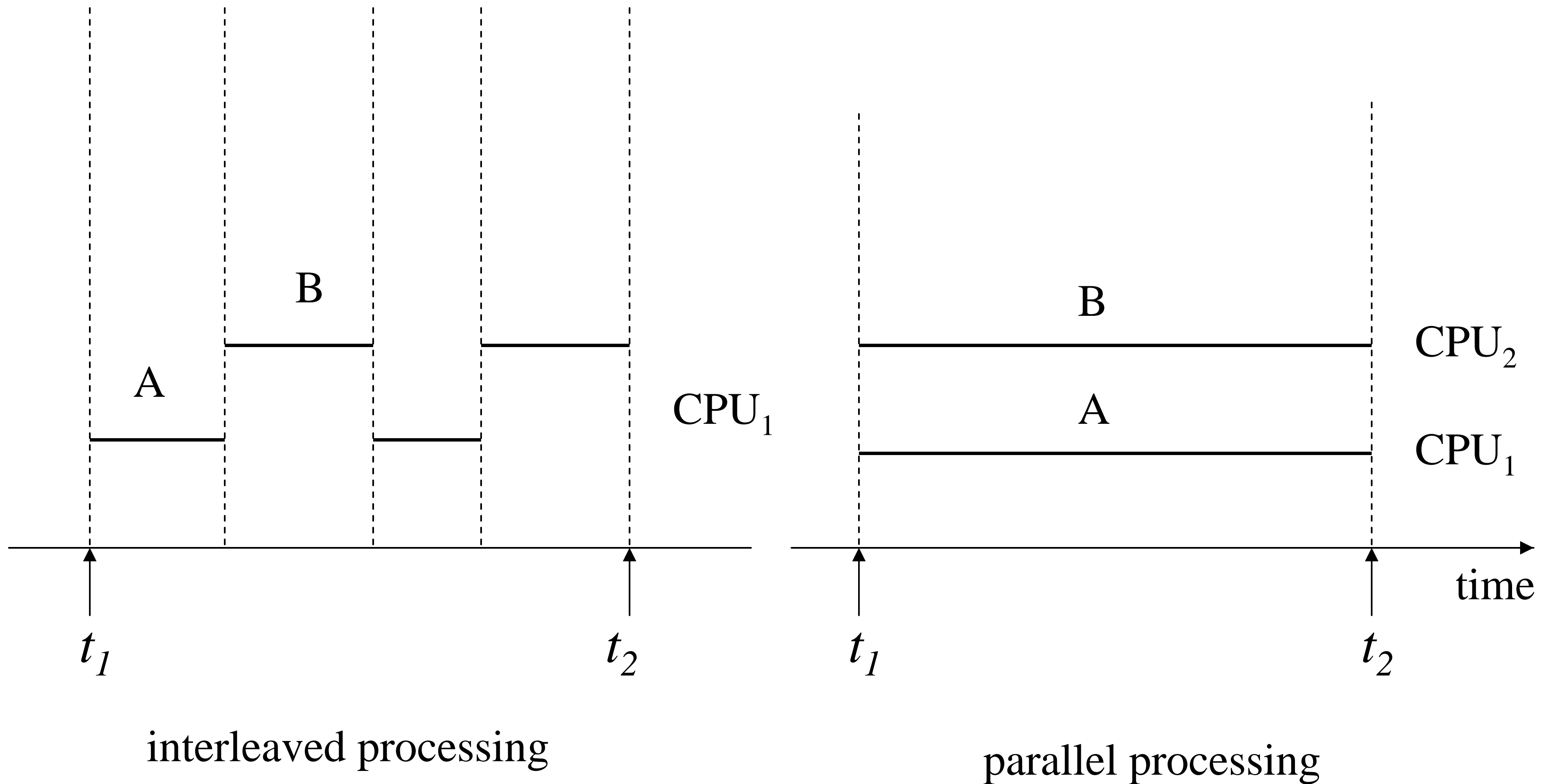
- Transaction processing
  - Transaction: High level atomic unit
    - Only two outcomes: all or nothing
    - Easy to program
  - Mature technique provides high performance and reliable service
    - concurrency transparency
    - failure transparency
  - Many commercial products are widely used
    - Database: RDB, OODB, ORDB ….
    - TP monitor: IBM CICS, BEA Tuxedo, Microsoft MTS, Java JTA …

```
Begin_transaction Transfer
Begin
    read (S);
    S = S – 100;
    if (S < 0) then
        print (“insufficient fund”);
        abort;
    endif;
    write(S);
    read(C);
    C := C + 100;
    write(C);
    commit;
end;
```

Why Do We Need Transactions?

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

Concurrent Transactions



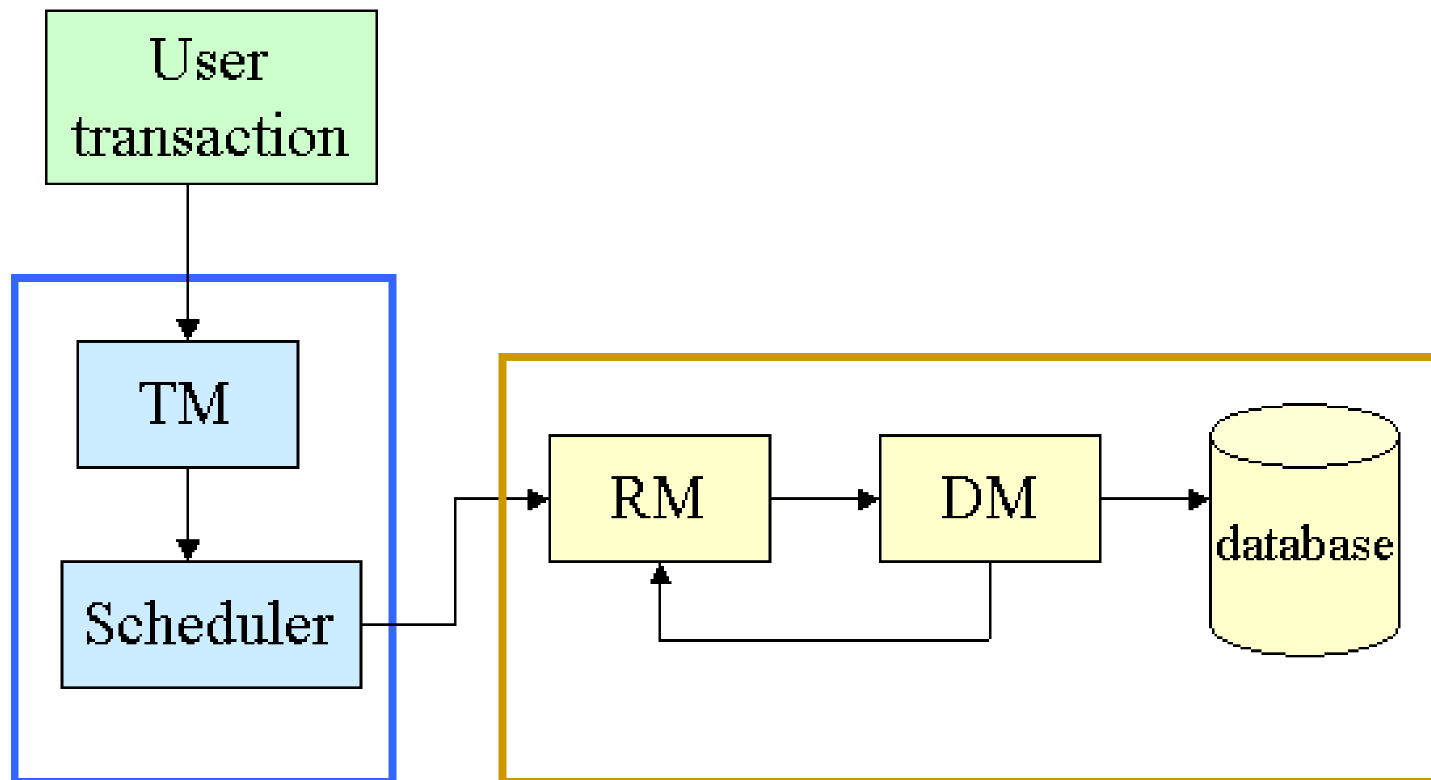
Terminology

- A transaction T is a logical unit of database processing that includes one or more database access operations
  - Embedded within application program
  - Specified interactively (e.g., via SQL)
- Transaction boundaries: Begin/end transaction
- Read vs. write transaction

Components in Database Systems

- A database system consists of 4 modules
  - transaction manager
    - performs the required preprocessing for transactions received from clients
  - scheduler
    - controls the order of execution of transaction operations
    - the main part for concurrency control
  - recovery manager
    - performs actions for transaction commitment and abortion
  - database manager
    - operates directly on the database
    - executes read, write, update operations

Components in Database Systems

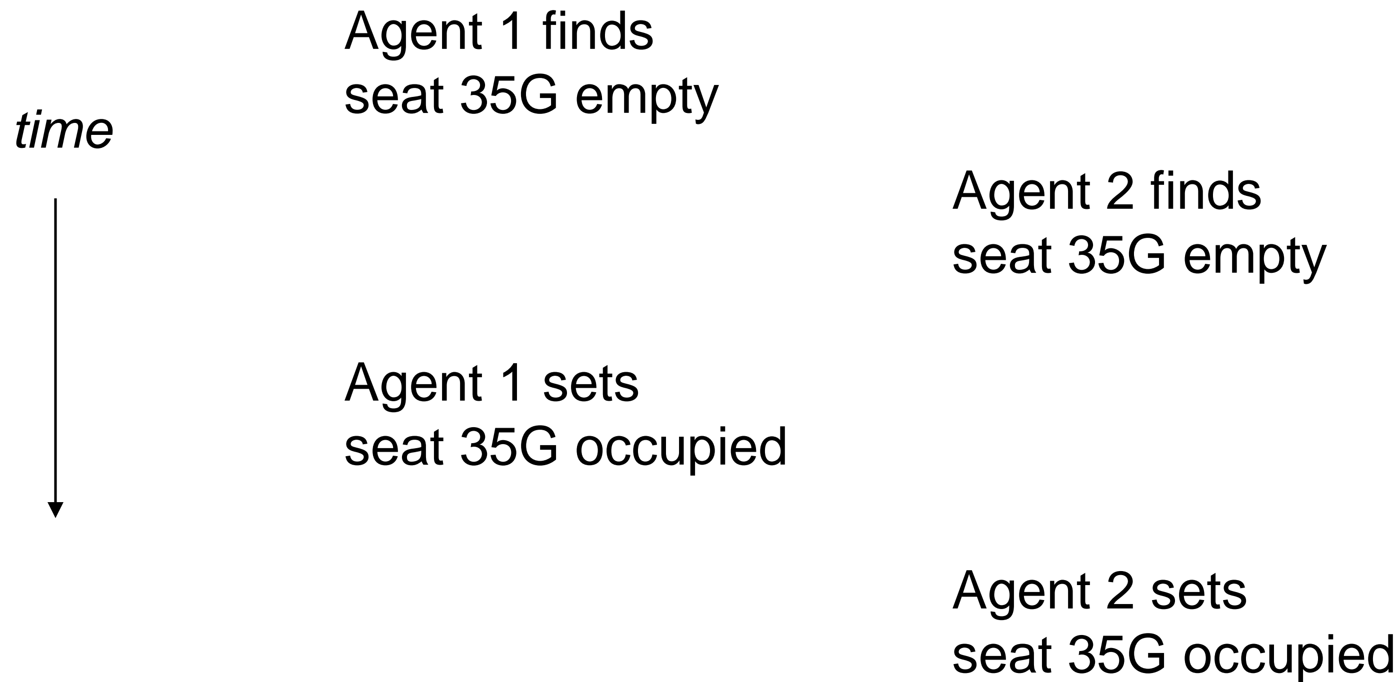


Sample Transaction (informal)

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

Another Sample Transaction

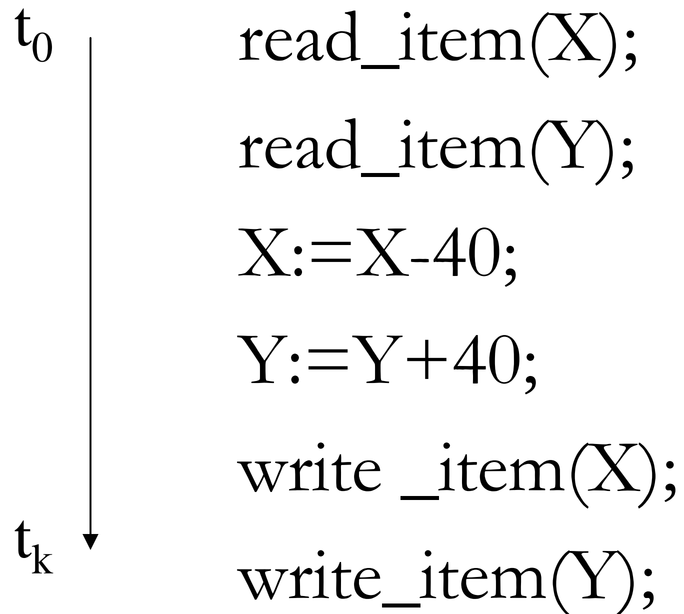
- Reserving a seat for a flight
- If concurrent access to data in DBMS, two users may try to book the same seat simultaneously



Terminology

- Basic access operations:
  - `read_item(X)`
  - `write_item(X)`
- How are read_item or write_item implemented?
- Read-set of T: all items that transaction reads
- Write-set of T: all items that transaction writes

Sample Transaction (Formal)



read_item(X);

read_item(Y);

X:=X-40;

Y:=Y+40;

write _item(X);

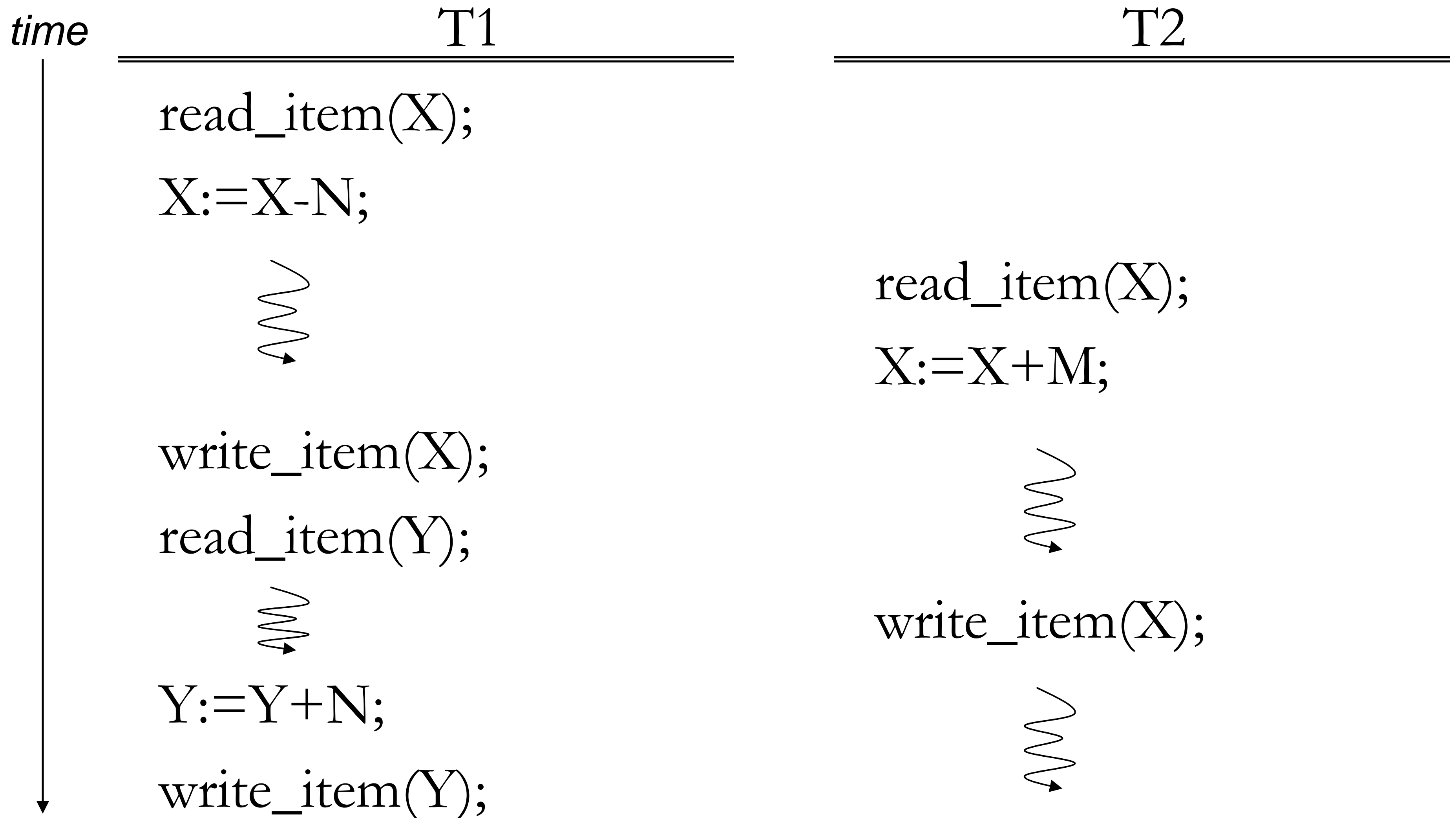
write_item(Y);

What Can Go Wrong?

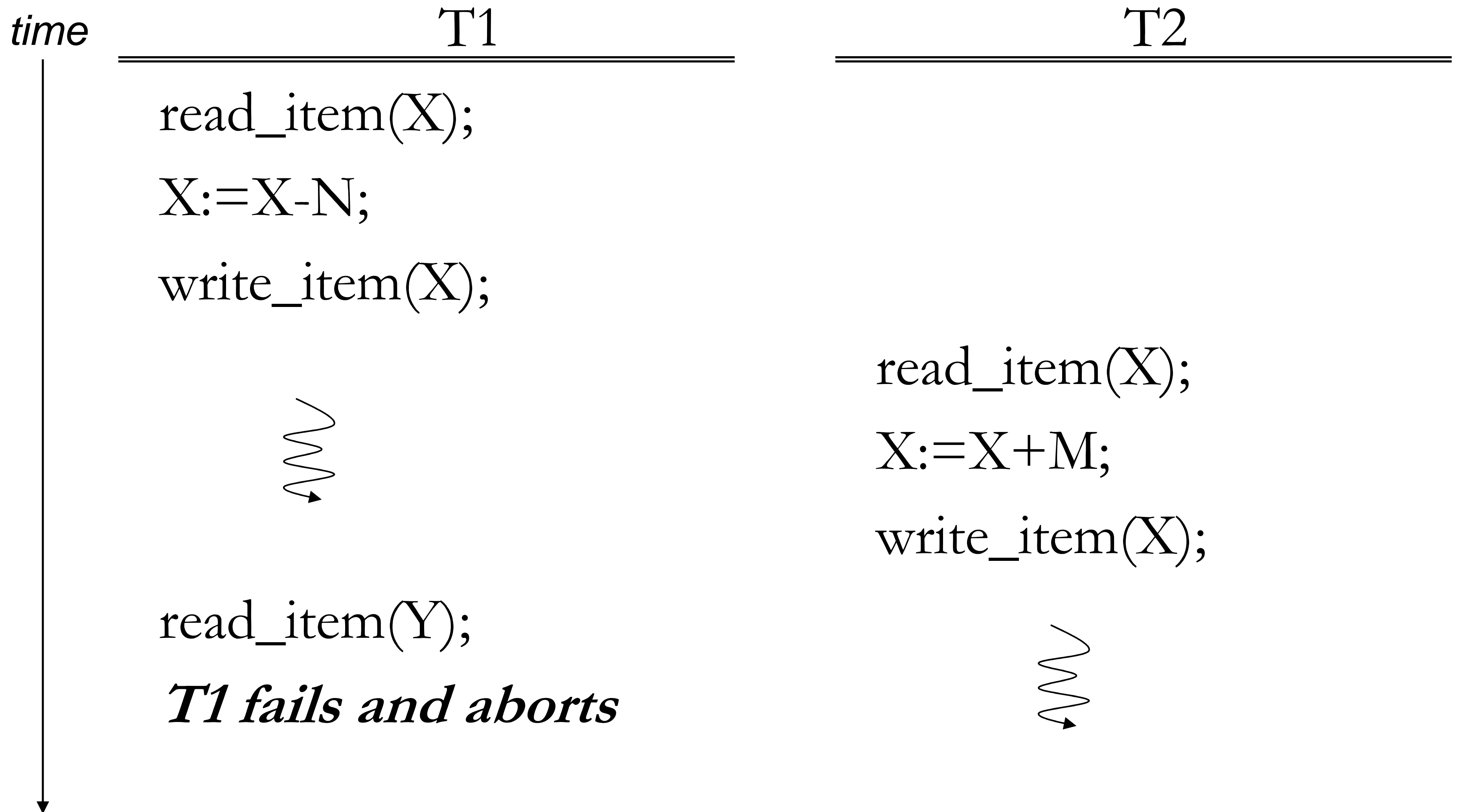
- Several problems can occur when concurrent transactions execute without control

- Illustrate several scenarios on the next slides

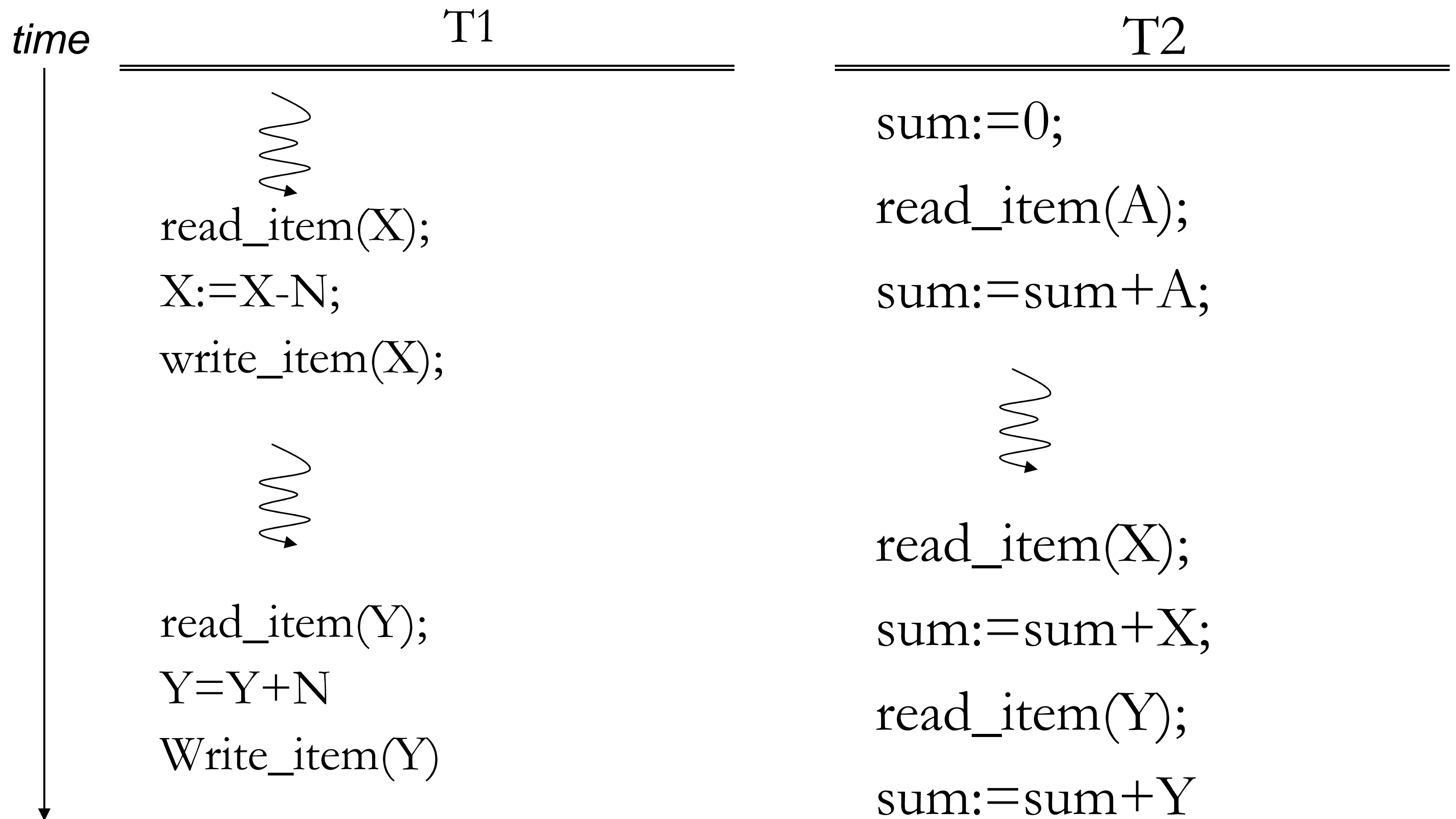
Lost Update Problem



Temporary Update (Dirty Read)



Incorrect Summary Problem



What Can Go Wrong?

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

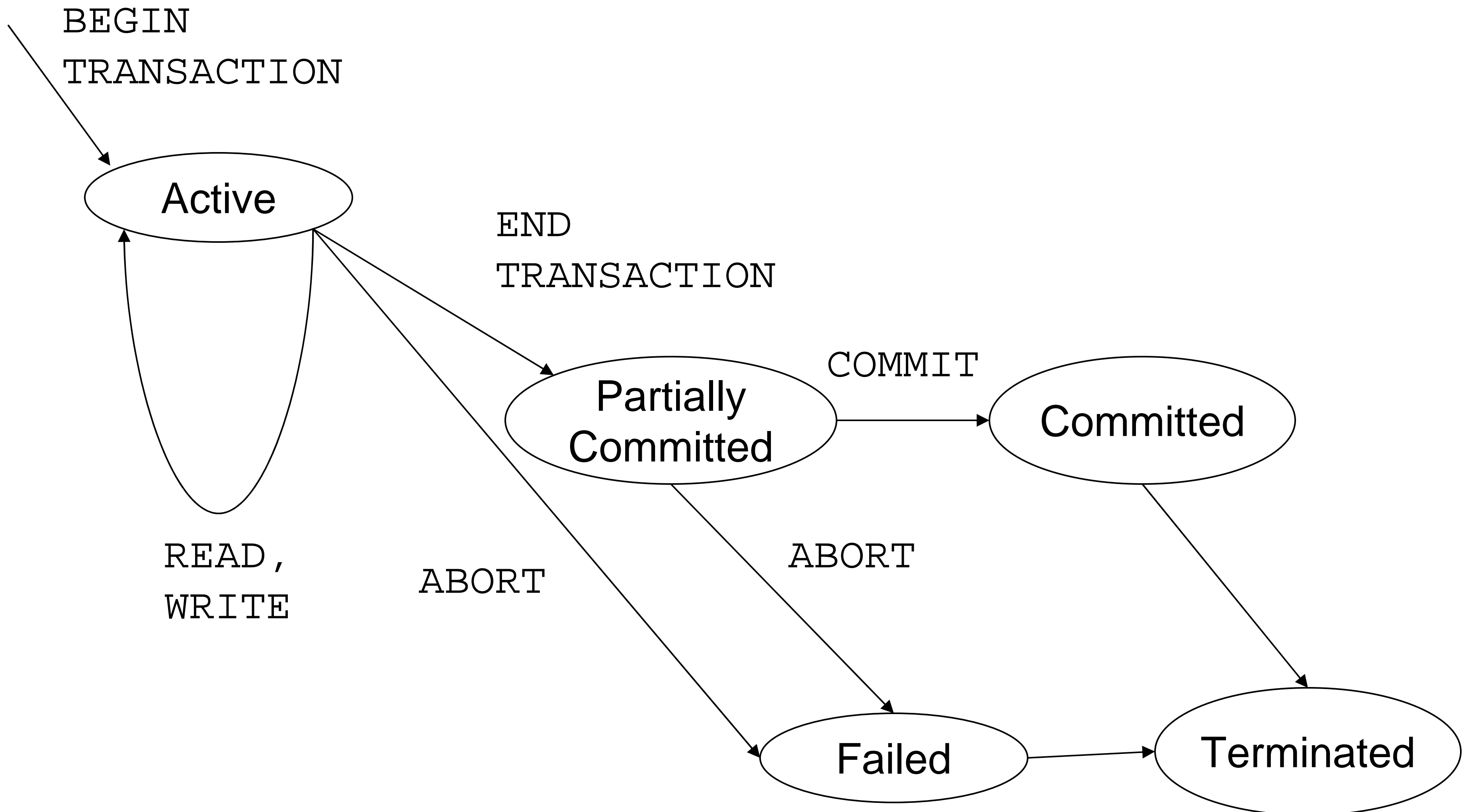
Other Problems

- System failures may occur
- Types of failures:
  - System crash
  - Transaction or system error
  - Local errors
  - Concurrency control enforcement
  - Disk failure
  - Physical failures
- DBMS has a *Recovery* Subsystem to protect database against system failures

Transaction States

- `BEGIN_TRANSACTION`: marks start of transaction
- `READ` or `WRITE`: two possible operations on the data
- `END_TRANSACTION`: marks the end of the read or write operations; start checking whether everything went according to plan
- `COMMIT_TRANSACTION`: signals successful end of transaction; changes can be "committed" to DB
- `ROLLBACK` (or `ABORT`): signals unsuccessful end of transaction, changes applied to DB must be undone

State Diagram



System Log

- Remember, DBMS must assure that we don't loose information due to system crashes
  - i.e., how do we recover from failure?
- Keep system log
  - Kept on disk, backed up periodically
  - Record every action

[`start_transaction`, *T*]

[`write_item`, *T*, *X*, *old*, *new*]

[`read_item`, *T*, *X*]

[`commit`, *T*]

[`abort`, T]

How is the Log File Used?

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

ACID Properties of Transactions

- *Atomicity:* Transaction is either performed in its entirety or not performed at all
  - Task of the recovery subsystem to enforce atomicity
- *Consistency preservation:* Transaction must take the database from one consistent state to another
  - Users/DBMS: enforce implicit and explicit constraints
- *Isolation:* Transaction should appear as though it is being executed in isolation from other transaction
  - Enforced by concurrency control subsystem
- *Durability:* Changes applied to the database by a committed transaction must persist
  - Enforced by recovery subsystem

Transaction Schedule

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

Sample Schedule

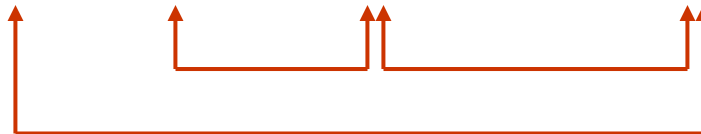
- T1: r(X); w(X); r(Y); w(Y); c
- T2: r(X); w(X); c

- Sample schedule:

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

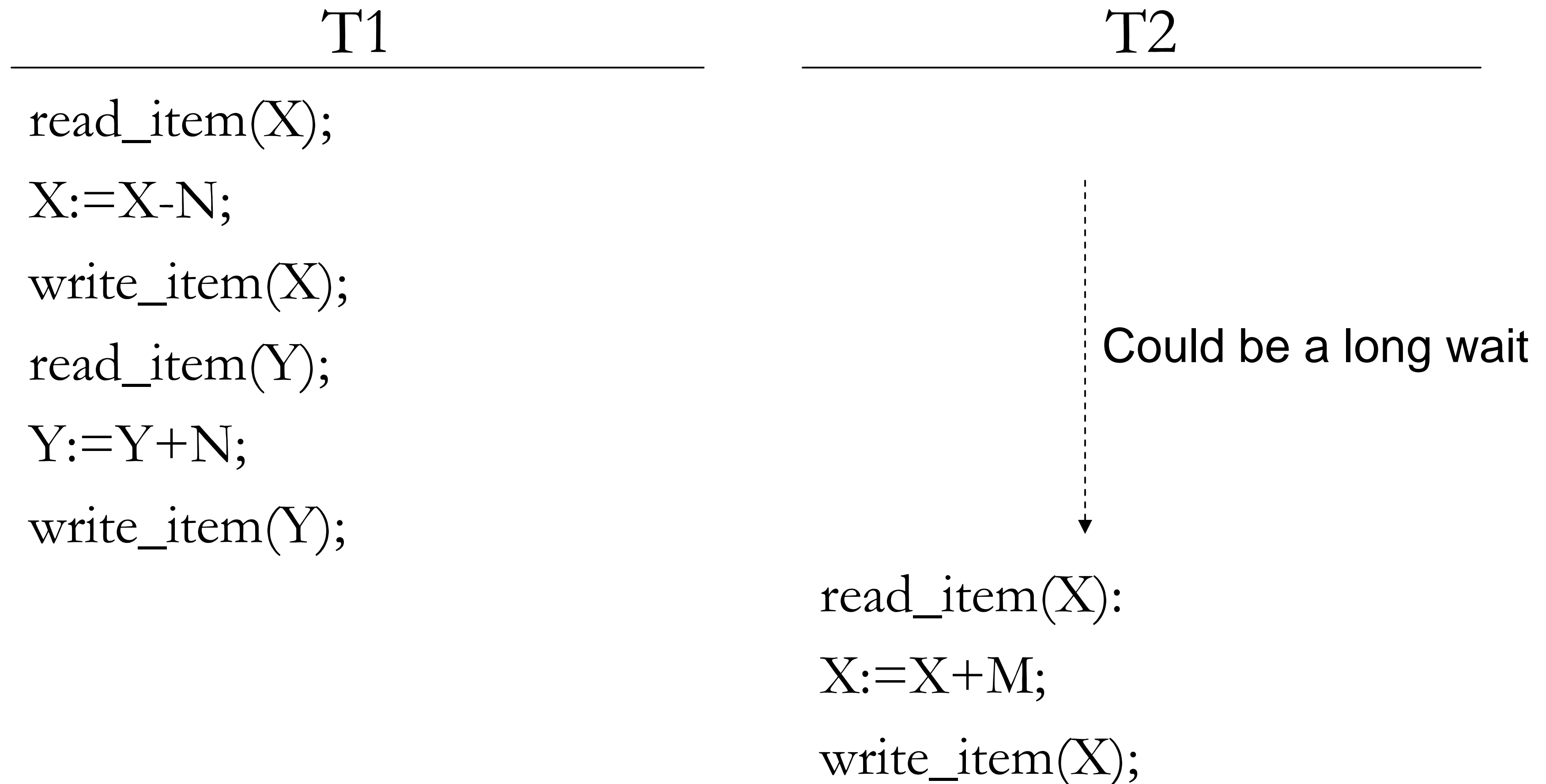
Conflicts

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



conflicts

Why Do We Interleave Transactions?



S is a serial schedule – no interleaving!

Serial Schedule

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

Better?

Schedule S'

T1	T2
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);	

S' is a non-serial schedule

T2 will be done faster but is the result correct?

Concurrent Executions

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

Better?

Schedule S"

T1	T2
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);	

S" is a non-serial schedule

Produces same result as serial schedule S

Serializability

- Assumption: Every serial schedule is correct
- Goal: Like to find *non-serial* schedules which are also correct
- A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions
- When are two schedules equivalent?
- Option 1: They lead to same result (*result equivalent*)
- Option 2: The order of any two conflicting operations is the same (*conflict equivalent*)

Result Equivalent Schedules

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

S1	S2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

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

Conflict Equivalent Schedules

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

Conflict Serializable

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

Example

Schedule A

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

Schedule B

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

B is conflict equivalent to A
$\Rightarrow$ B is serializable

Test for Serializability

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

Remember ACID Properties?

- **Atomicity** → enforced by *transaction recovery subsystem*
- **Consistency** → enforced by *application programmers* and part of DBMS that enforces *integrity constraints*
- **Isolation** → enforced by *concurrency control system*
- **Durability** → enforced by *transaction recovery subsystem*

- Now, focus on concurrency control
- Ensure serializability of schedules through protocols that result in serializable schedules

Quick Recap

- Difference between serial and serializable
- Real DBMS does not test for serializability
  - Very inefficient since transactions are continuously arriving
  - Would require a lot of undoing
- Solution: concurrency protocols
- If followed by every transaction, and if enforced by transaction processing system, guarantee serializability of schedules in which transaction appears

Concurrency Control Through Locks

- **Lock:** variable associated with each data item
  - Describes status of item wrt operations that can be performed on it
- Binary locks: Locked/unlocked
  - Enforces mutual exclusion
- Multiple-mode locks: Read/write
  - a.k.a. Shared/Exclusive
- Three operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X)
- Each data item can be in one of three lock states

Implementation

- Maintain lock table
- Keep track of locked items and their locks
  <data item, LOCK, no_of_reads, locking_transaction>
- For read locks, keep track of the number of transactions that hold a read lock on an item

Locking Rules

1. T must issue read_lock(X) or write_lock(X) before any read_item(X) op is performed in T
2. T must issue write_lock(X) before any write_item(X) op is performed in T
3. T must issue unlock(X) after all read_item(X) and write_item(X) ops are completed in T
4. T will not issue a read_lock(X) if it already holds a read lock or write lock on X (may be relaxed)
5. T will not issue a write_lock(X) if it already holds a read lock or write lock on X (may be relaxed)

Lock Conversions

- Sometimes beneficial to relax locking rules 4 and 5
- Upgrade read lock on X to a write lock (by issuing a write_lock(X))
  - Only possible if T is the only transaction holding a read lock on X
- Downgrade a write lock by issuing a read_lock(X)
- Must be noted in lock table

Granting of Locks

- Suppose T2 has read-lock on item X
- T1 is requesting write-lock on item X; needs to wait for T2 to release
- T3 requests read-lock on X; request is granted
  - Assume shortly thereafter T2 relinquishes read-lock
  - Continue scenario through a sequence of transactions all requesting read-lock on X
  - T1 will never make progress
- T1 is said to be *starved*

Granting of Locks

- How do you avoid starvation in the presence of locks?

  When a new read lock request arrives, and some "old" transaction is waiting for a write lock on the same object, wait for the "old" transaction to get its lock first.

Two Transactions

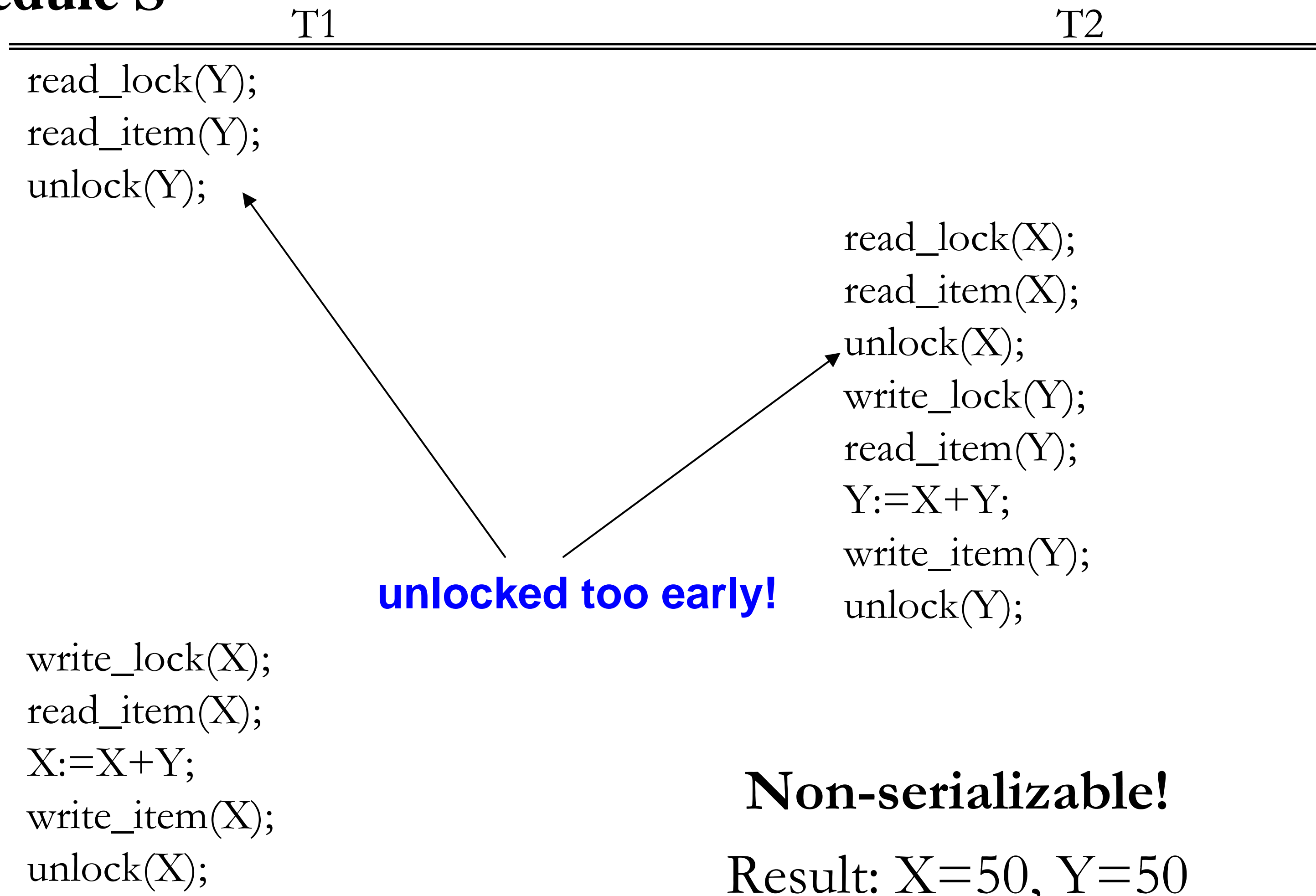
T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Let's assume serial schedule S1: T1;T2

Initial values: X=20, Y=30 $\rightarrow$ Result: X=50, Y=80

Locks Alone Don't Do the Trick!

Let's run T1 and T2 in interleaved fashion



Non-serializable!

Result: X=50, Y=50

Two-Phase Locking (2PL)

- Def.: Transaction is said to follow the *two-phase-locking protocol* if all locking operations precede the *first* unlock operation
  - Expanding (growing) = first phase
  - Shrinking = second phase
- During the shrinking phase no new locks can be acquired!
  - Downgrading ok
  - Upgrading is not

Example

T1'	T2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

- Both T1' and T2' follow the 2PL protocol
- Any schedule including T1' and T2' is guaranteed to be serializable
- Limits the amount of concurrency

Variations to the Basic Protocol

- Previous technique knows as *basic 2PL*
- *Conservative* 2PL (static) 2PL: Lock all items needed BEFORE execution begins by predeclaring its read and write set
  - If any of the items in read or write set is already locked (by other transactions), transaction waits (does not acquire any locks)
  - Deadlock-free, but not flexible (sometimes transactions are interactively controlled)

Variations to the Basic Protocol

- *Strict* 2PL: Transaction does not release its write locks until AFTER it aborts/commits
  - Not deadlock free but guarantees recoverable schedules (strict schedule: transaction can neither read/write X until last transaction that wrote X has committed/aborted)
  - Most popular variation of 2PL

Variations to the Basic Protocol

- *Rigorous* 2PL: No lock is released until after abort/commit
  - Transaction is in its expanding phase until it ends

Remarks

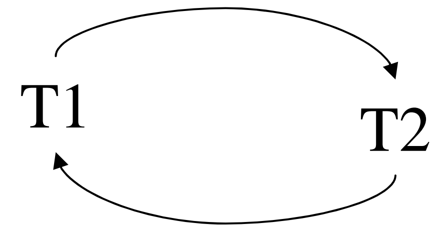
- Concurrency control subsystem is responsible for inserting locks at right places into your transaction
  - Strict 2PL is widely used
  - Requires use of waiting queue
- All 2PL locking protocols guarantee serializability
- Does not permit all possible serial schedules
  - Conservative and rigorous 2PL charge a high price for serializability
- However, deadlock-based algorithms may suffer from starvation and *deadlock*

What is Deadlock?

- Occurs when each transaction T_i in a set of two or more is waiting on an item locked by some other transaction T_j in the set

S

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



Deadlock Prevention

- *Locking* as deadlock prevention leads to very inefficient schedules (e.g., conservative 2PL)
- Better, use *transaction timestamp* TS(T)
  - TS is unique identifier assigned to each transaction
  - if T1 starts before T2, then TS(T1) < TS(T2) (older has smaller timestamp value)
  - Not actually "real time", just a counter

Wait-Die Scheme

- Assume T_i tries to lock X which is locked by T_j
- If $TS(T_i) < TS(T_j)$ (T_i older than T_j), then T_i is allowed to wait
- Otherwise, T_i younger than T_j, abort T_i (T_i dies) and restart later with SAME timestamp
- Older transaction is allowed to wait on younger transaction
- Younger transaction requesting an item held by older transaction is aborted and restarted

More Deadlock Prevention

- *Waiting schemes* (require no timestamps)
- No waiting: if transaction cannot obtain lock, aborted immediately and restarted after time *t*
  - $\rightarrow$ needless restarts
- Cautious waiting:
  - Suppose T_i tries to lock item X which is locked by T_j
  - If T_j is not blocked, T_i is blocked and allowed to wait
  - O.w. abort T_i
  - Cautious waiting is deadlock-free
- Ordering of resources (a fixed ordering on how objects must be locked, eg, must lock X before locking Y – if all transactions use the same ordering, deadlock-free!)

Deadlock Detection

- DBMS checks if deadlock has occurred
  - Works well if few short transactions with little interference
  - O.w., use deadlock prevention
- Two approaches to deadlock detection:

1. Wait-for graph
   - If cycle, abort one of the transactions (victim selection)
2. Timeouts

Starvation

- Transaction cannot continue for indefinite amount of time while others proceed normally
- When? Unfair waiting scheme with priorities for certain transactions
  - E.g., in deadlock detection, if we choose victim always based on cost factors, same transaction may always be picked as victim
  - Include rollbacks in cost factor

Nested Transactions

- Some systems allow a Transaction T to be composed of subtransactions T = T1, T2, T3, ….
- Each subtransaction can in turn have "subsub" transactions, and so forth.
- Parents inherit the locks of their children (remember, we want 2PL overall). Aborted subtransactions can be retried.
- Subtransactions of a parent can execute concurrently.