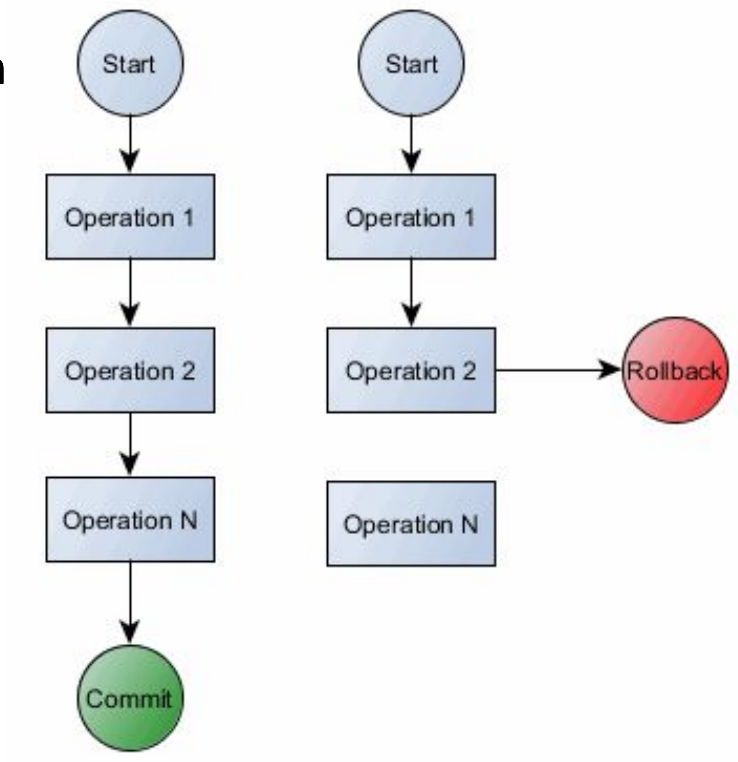
Chapter 16: Transaction Management

- Looking at the execution of user programs on the database

- Interleaving transactions are the foundation
  - Concurrent execution

- Interleaving must be done with care
- Should be equivalent to some serial order of transaction execution (Concurrency control)

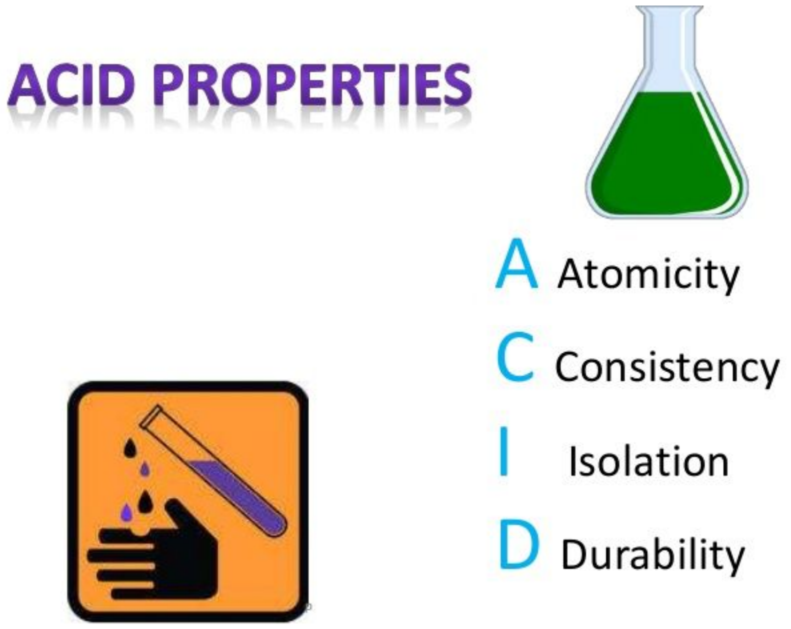


Transactions

- A single execution of a user program on the database.
- A single transaction might require many queries, each reading and/or writing information to the database.
- A transaction is an isolated sequence of operations that can either all be saved to the database or all cancelled and ignored .

Main Properties of Transactions

- To resolve issues relating to concurrency, crash recovery, reliability, and consistency,
- A transaction in a database management system must maintain 4 properties:



[1] Atomic

- A transaction is an atomic unit of processing
- Either all operations of a transaction are executed or none of them are. (All or nothing)
- Either all actions are carried out or none.
  - Incomplete transactions are aborted (rolled back)
- The user should not worry about the effect of incomplete transactions.

[1] DBMS abort the transaction

[2] System may crash

[3] Transaction may encounter a problem by itself

- Example ATM Machine:

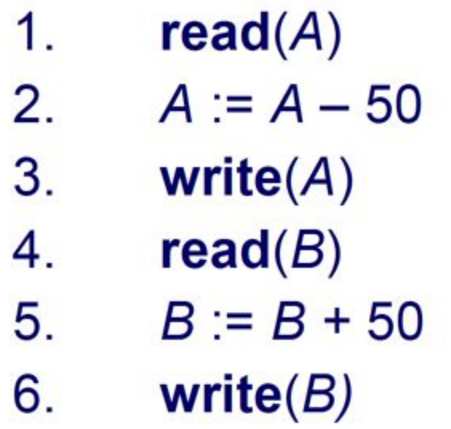
[2] Consistency

- Every transaction running by itself must preserve the consistency of the database.
- A requested action must be reflected correctly and accurately on the database.
- In other words, a transaction must leave the database in consistent state.

- **This is the responsibility of database developers.**

Example (Consistency)

- Transaction to transfer $50 from account A to account B



```
1.    read(A)
2.    A := A – 50
3.    write(A)
4.    read(B)
5.    B := B + 50
6.    write(B)
```

- Consistency requirement:
  - The sum of A and B is unchanged by the execution of the transaction

[3] Isolation

- Even though transactions maybe interleaved, the net effect is identical to executing all transactions one after another in some serial order
  - Transactions are isolated or protected from the effects of concurrently scheduling other transactions.
  - Every transaction is an independent entity.
  - One transaction should not affect any other transaction running at the same time.
  - Example: **T1** and **T2** may be interleaved, but the net effect should be same as running **T1** and then running **T2**.

[4] Durability

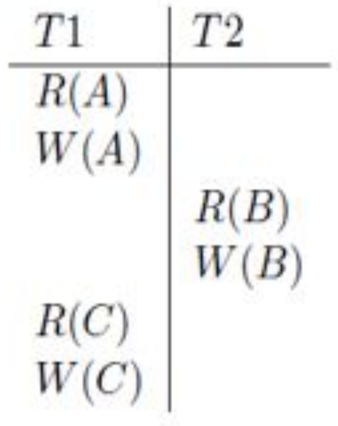
- Once the DBMS informs the user that the transaction has been successfully completed:
  - Its effects should persist.
  - Even if the system crashes before changes are reflected to disk
  - How?!
    - A separate log is maintained for every action in the database.
    - The log entry is written to stable storage before any action is taken.
    - This is the work of the Recovery Manager module.

Transactions and Schedules

- A transaction is seen by DBMS as a series of **actions**…..**read** and **write**

- $R_T(O)$: transaction reading an object from DB
- $W_T(O)$: transaction writing an object to DB
- Abort T : action of a transaction aborting
- Commit T : action of transaction committing

Transactions and Schedules (2)



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

- **<u>A schedule:</u>** is a chronological ordering of actions (read/write/abort/commit) from a set of transactions.
  - The order in which two actions of a transaction T appears in a schedule must be the same as the order as they would appear in T.
  - Realistically: A schedule is an actual execution sequence!

- A **<u>complete schedule</u>** must include all actions of all transactions appearing in it
- **<u>Serial schedule</u>** is the one that transactions in it are not interleaved

Concurrent Execution Of Transactions

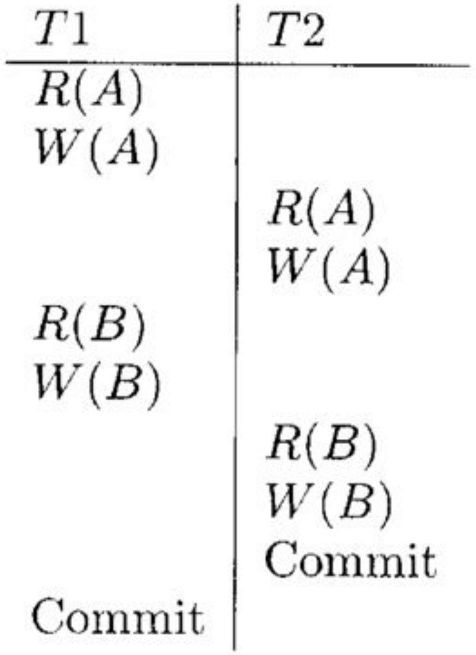
- The DBMS interleaves the actions of different transactions to improve performance
- Must ensure Transaction Isolation
  - **not all interleaves should be allowed**

- Concurrency necessary for:
  - Overlapping I/O and CPU operations reduces amount of time disks and processors are idle. This increases system throughput
    - **Throughput**: Average number of transactions completed in a unit time.

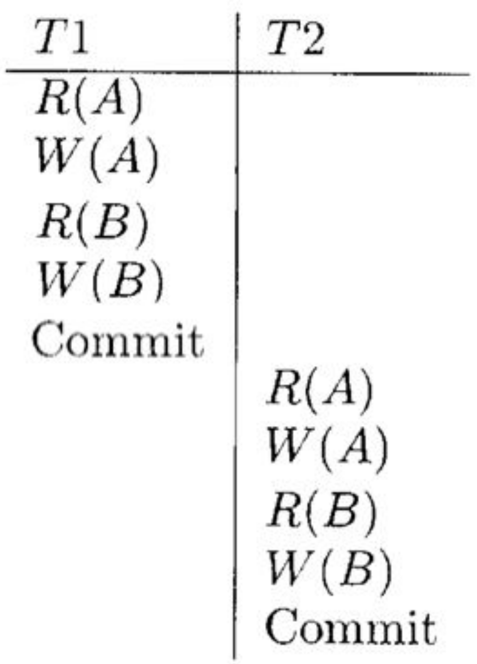
  - Interleaved execution of a short transaction with a long transaction allows the short transaction to complete quickly.
    - In serial execution of transactions, a short transaction might get stuck behind a long transaction leading to unacceptable delays in response time.
      - **Response time**: Average time take to complete a transaction.

Serializability

- **<u>A serializable schedule</u>:** over a set of a committed transactions is a schedule that is guaranteed to be identical to that of some complete serial schedule.

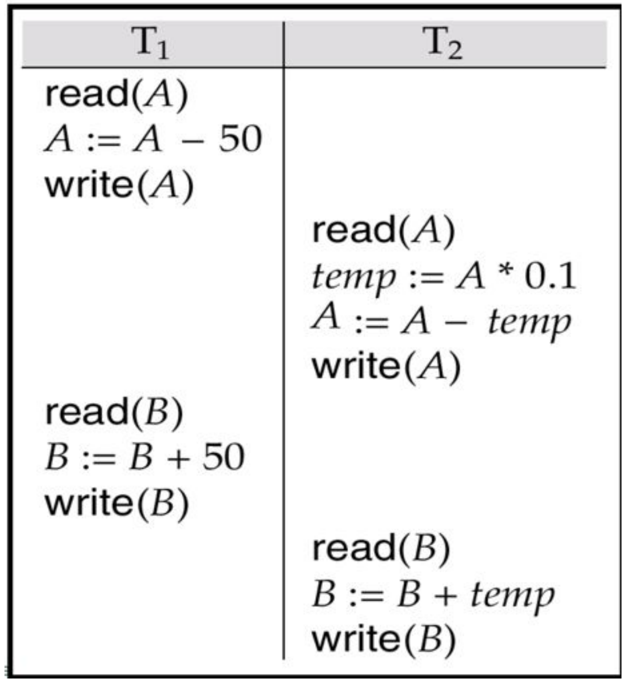


$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
$R(B)$	
$W(B)$	
	$R(B)$
	$W(B)$
	Commit
Commit	

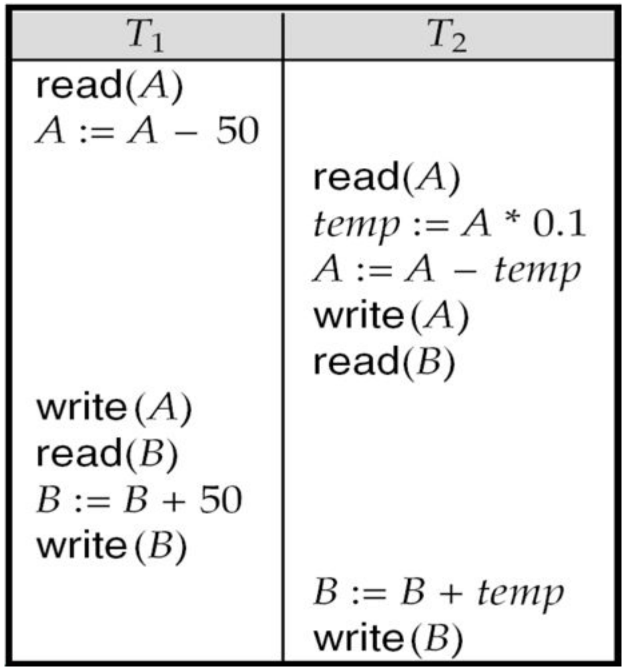


$T1$	$T2$
$R(A)$	
$W(A)$	
$R(B)$	
$W(B)$	
Commit	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit

Serializability (2)



T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)



T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability (3)

- DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution;
  - i.e., using a schedule that is not serializable.
- How could this happen?!
  - First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule.
  - Second, SQL gives application programmers the ability to instruct the DBMS to choose non-serializable schedulers.

The root of all problems

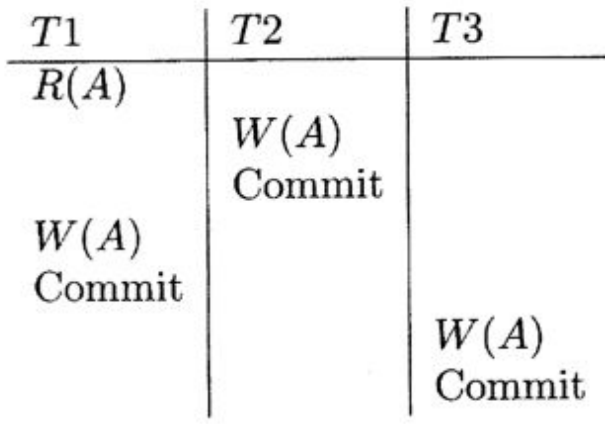
- Writing is the main problem

- A conflict occurs
  - Two actions from different transactions
  - On the same data object
  - At least one of them is a **write**

- **WR**
- **RW**
- **WW**

Serializability Graph

- A node is drawn for each Transaction
- An arc from Ti to Tj if an action of Ti precedes and CONFLICTS with one of Tj's actions

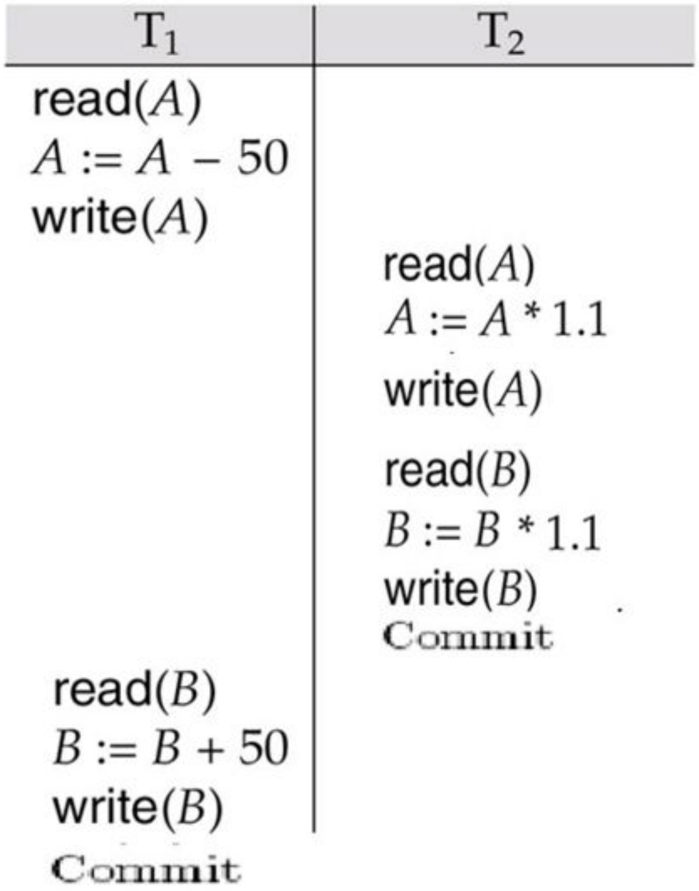


$T1$	$T2$	$T3$
$R(A)$		
	$W(A)$	
	Commit	
$W(A)$		
Commit		
		$W(A)$
		Commit

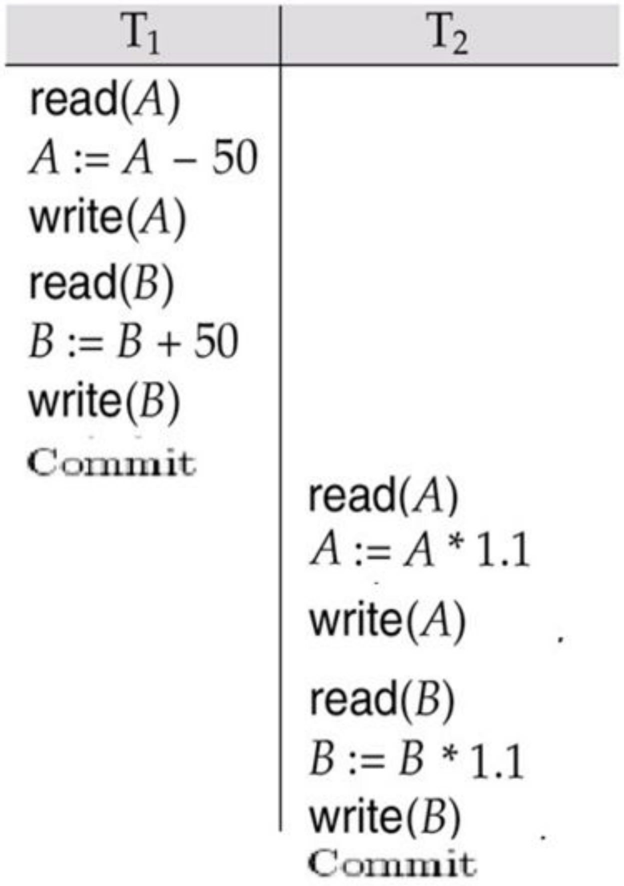
Concurrent Execution Of Transactions

- The DBMS interleaves the actions of different transactions to improve performance
- Must ensure Transaction Isolation
  - **not all interleaves should be allowed**
- Must ensure schedule remains serializable
  - Or the serializability graph contains no cycles!

Reading Uncommitted Data (WR)



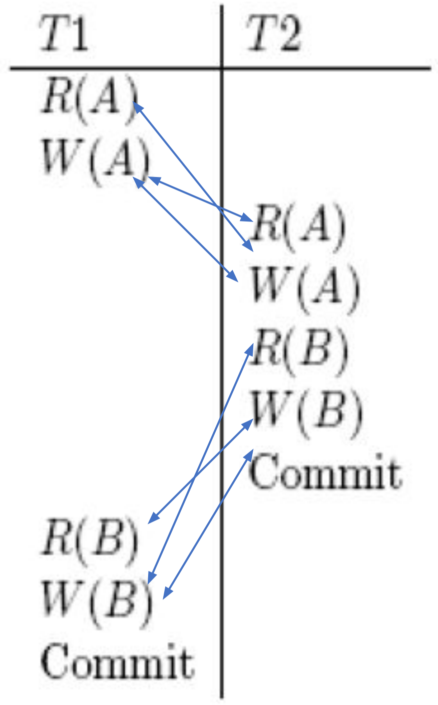
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$A := A * 1.1$
	write(A)
	read(B)
	$B := B * 1.1$
	write(B)
	Commit
read(B)	
$B := B + 50$	
write(B)	
Commit	

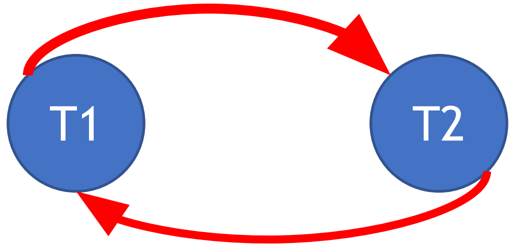


T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
Commit	
	read(A)
	$A := A * 1.1$
	write(A)
	read(B)
	$B := B * 1.1$
	write(B)
	Commit

Reading Uncommitted Data (2)

- The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other.
- The problem can be traced to the fact that the value of A written by T1 is read by T2 before T1 has completed all its changes.



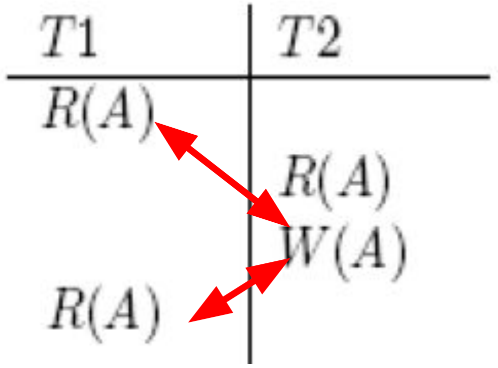


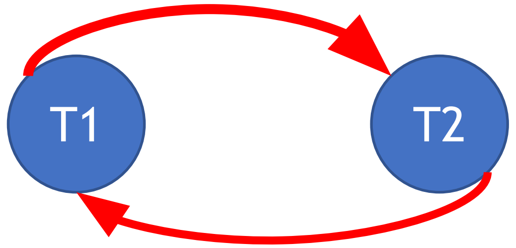
Unrepeatable Reads (RW)

- where a transaction *T* reads the same item twice and the item is changed by another transaction *T*′ between the two reads.
- A transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress.
- If T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime.

Unrepeatable Reads (RW) (2)

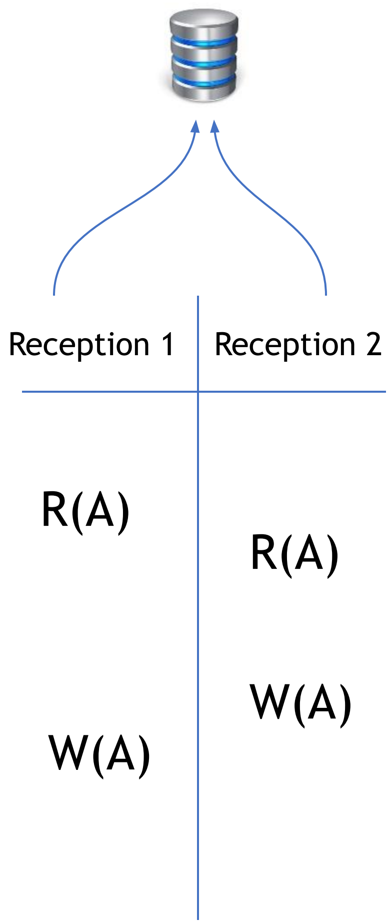
- Suppose A is available number of copies of a book in a library.
- A transaction that places an order first reads the A, checks that it is greater than 0 and decrements it.
- Transaction T1 reads A and finds 1,
- Transaction T2 also reads A, finds 1 and decrements A to 0
- Transaction T1 reads A and gets a different value!





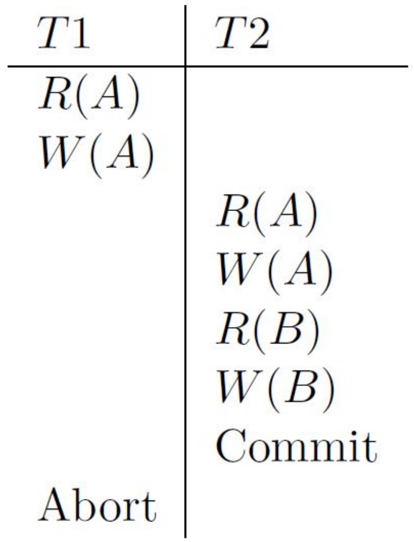
Overwriting Uncommitted Data

- A transaction T2 could overwrite the value of an object A, which has already been modified by a transaction T1, while T1 is still in progress.
- This is also known as Lost Update Problem
  - Flight example
  - Two users
    - One Ticket left!



Optional: Unrecoverable schedule

- Committing uncommitted data
- How to make it recoverable?
- Can we avoid cascading aborts?



$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
Abort	

Figure 18.3 An Unrecoverable Schedule

Lock-Based Concurrency Control

- A DBMS must be able to ensure that:
  - **only serializable, recoverable schedules are allowed,**
  - **and that no actions of committed transactions are lost while undoing aborted transactions.**
- A DBMS typically uses a *locking protocol* to achieve this.
- A **locking protocol** is a set of rules to be followed by each transaction in order to ensure that even though actions of several transactions might be interleaved, the net efect is identical to executing all transactions in some serial order

Strict Two Phase Locking (S2PL)

- It is the most widely used locking protocol.
- It has two rules:
- A)
  - If a transaction *T* wants to ***read*** it first requests a *shared* lock on the object
  - If a transaction *T* wants to ***modify*** an object, it first requests a ***exclusive*** lock on the object
- B) All locks held by a transaction are **released** when the transaction is **completed**.

Example S2PL:

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(A)$
	$W(A)$
	$R(B)$
	$W(B)$
	Commit
$R(B)$	
$W(B)$	
Commit	

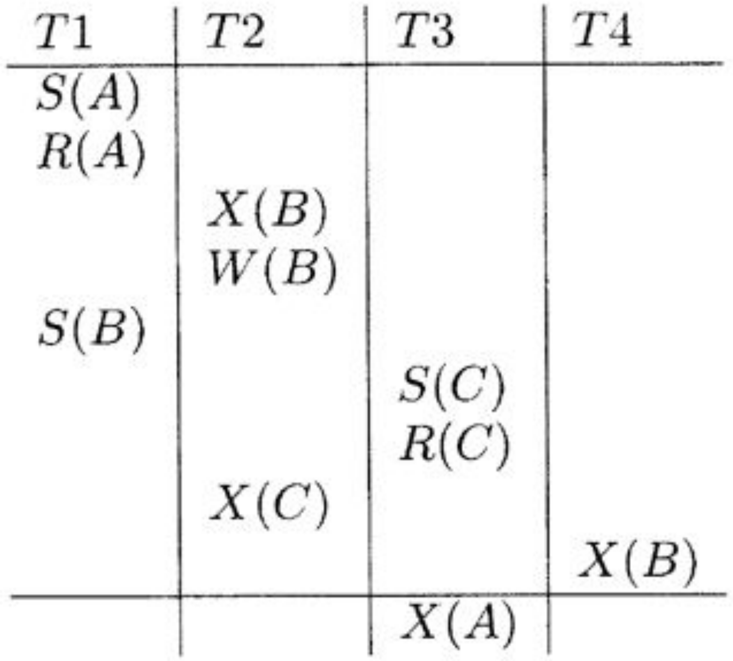
$T1$	$T2$
$X(A)$	
$R(A)$	
$W(A)$	
$X(B)$	
$R(B)$	
$W(B)$	
Commit	
	$X(A)$
	$R(A)$
	$W(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit

Example S2PL (2):

$T1$	$T2$
$S(A)$	
$R(A)$	
	$S(A)$
	$R(A)$
	$X(B)$
	$R(B)$
	$W(B)$
	Commit
$X(C)$	
$R(C)$	
$W(C)$	
Commit	

Deadlocks, Thrashing

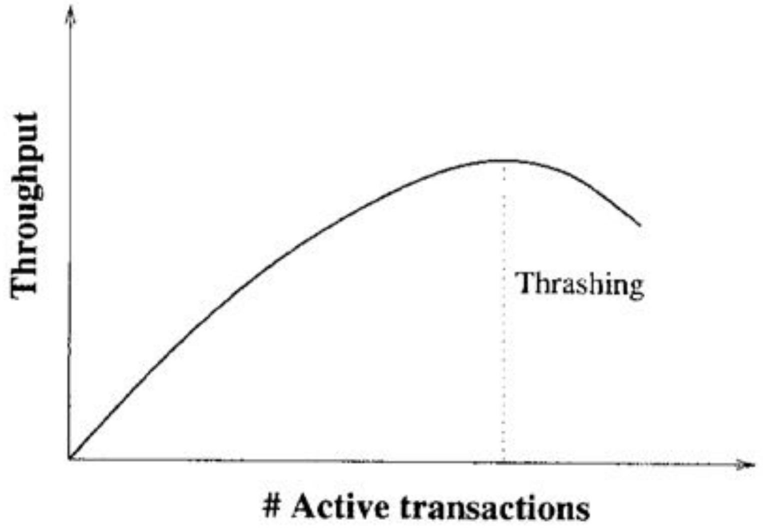
- What is a deadlock?
- How do we detect a deadlock?
  - Construct a wait-for graph
    - For each Transaction a node is drawn
    - An arch from Ti to Tj is drawn if and only if Ti is waiting for a lock to be released by Tj



$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	

Thrashing

- Seen when around 30% of active transactions are blocked
- At this point adding any new transaction actually reduces throughput.


Throughput
Thrashing
# Active transactions