CS354: Database

Transaction Management, Concurrency Control and Recovery Control

Transaction

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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*)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

- To preserve the integrity of data the database system must ensure:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Atomicity

- **Atomicity requirement**
  - Either all operations of the transaction are properly reflected in the database or none are.
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

Consistency

- Execution of a transaction in isolation preserves the consistency of the database.
- 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** in above example:
  - the sum of A and B is unchanged by the execution of the transaction

Consistency (contd.)

- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Isolation

- Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Isolation (contd.)

- **Isolation requirement** - if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

Isolation (contd.)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits

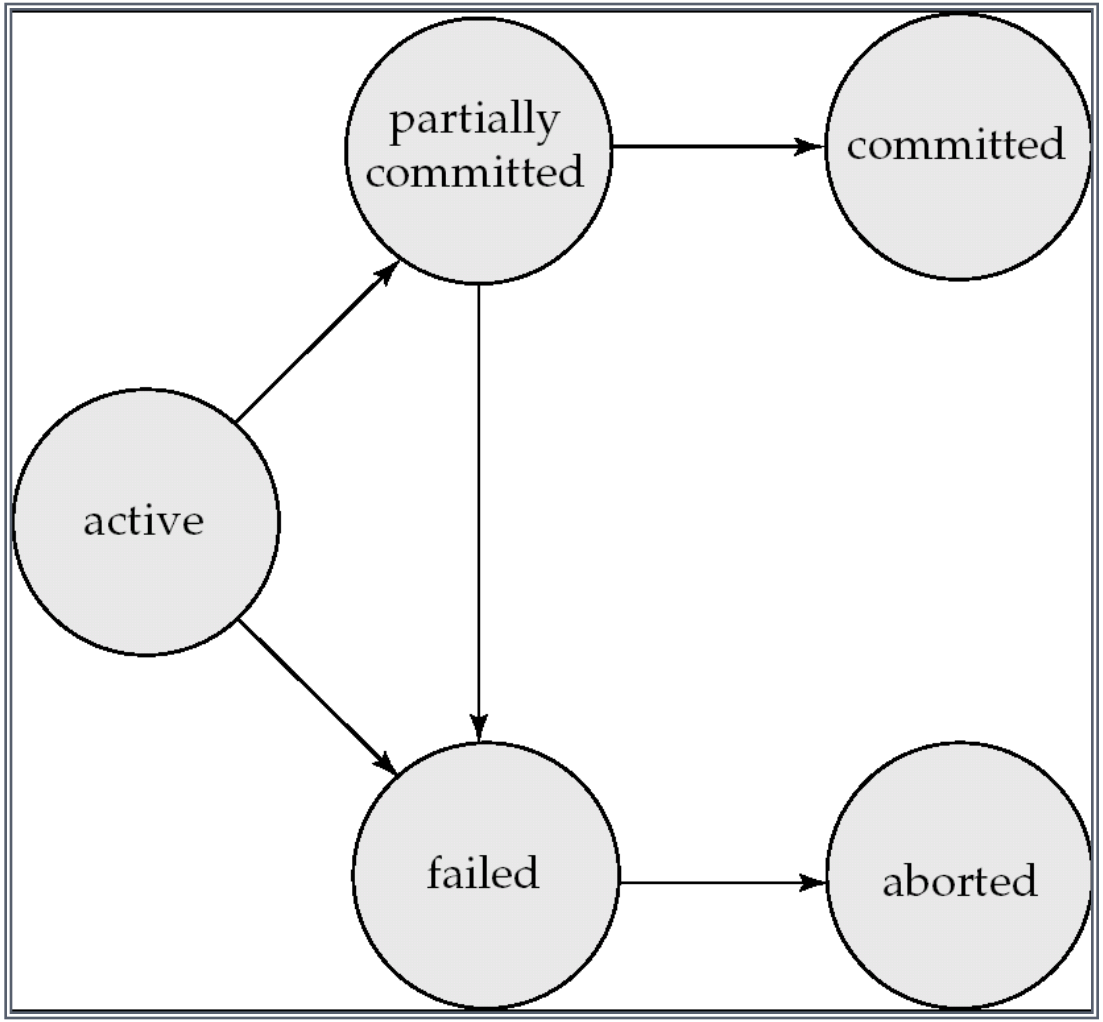
Durability

- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Transaction State

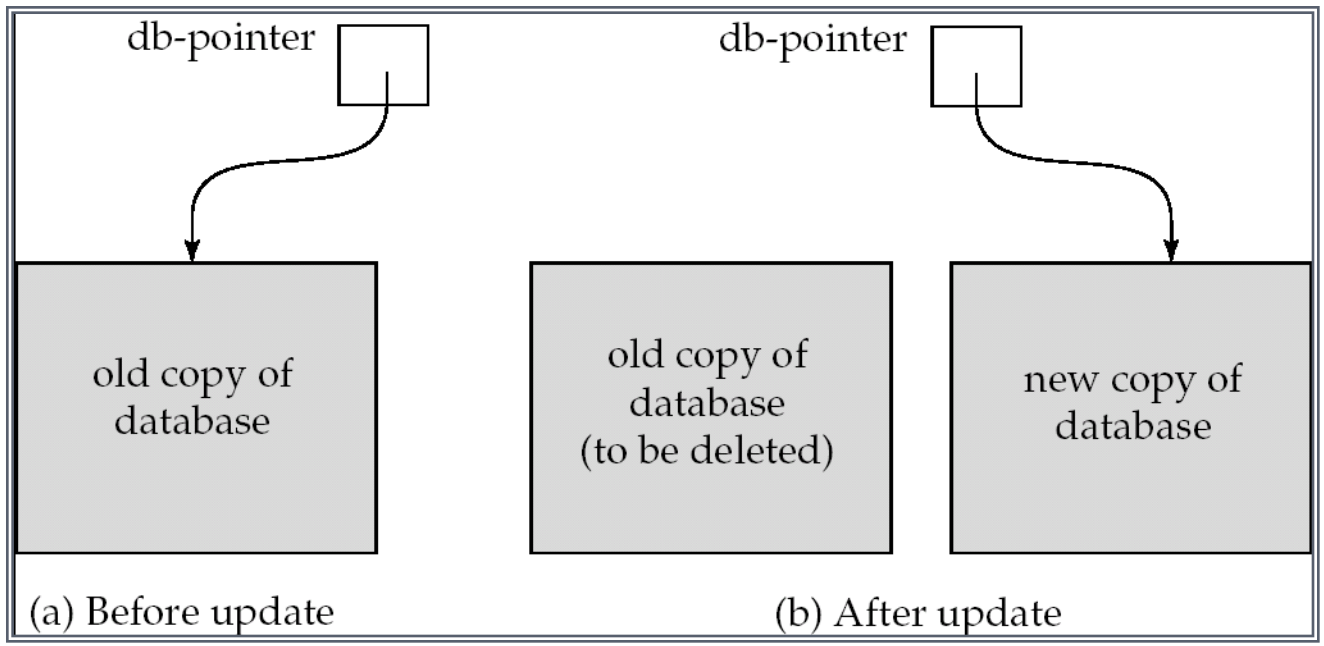
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
  - Kill the transaction
- **Committed** – after successful completion.

Transaction state



Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a *shadow copy* of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - all updated pages have been flushed to disk.



Implementation of Atomicity and Durability (Cont.)

- db_pointer always points to the current consistent copy of the database.
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
  - Assumes that only one transaction is active at a time.
  - Assumes disks do not fail
  - Useful for text editors, but
    - extremely inefficient for large databases
  - Does not handle concurrent transactions

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedule

- A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a *commit* instruction as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an *abort* instruction as the last statement

Schedule 1

- Let T_1 transfers \$50 from A to B, and T_2 transfers 10% of the balance from A to B.
- A *serial* schedule in which T_1 is followed by T_2 :

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

SCHEDULE 2

- A serial schedule where T_2 is followed by T_1

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

SCHEDULE 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

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)

In Schedules 1, 2 and 3, the sum A + B is preserved.

SCHEDULE 4

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)

- The above concurrent schedule does not preserve the value of $(A + B)$.

SERIALIZABILITY

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- *Simplified view of transactions*
  - We ignore operations other than **read** and **write** instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

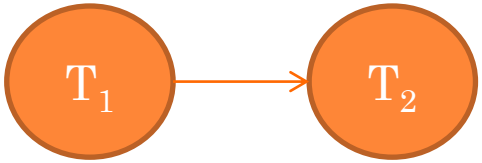
T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a directed graph where the vertices are the transactions participating in the schedule
- We draw an edge from T_i to T_j if the two transaction conflict, i.e.,
  - T_i executes write(Q) before T_j executes read(Q)
  - T_i executes read(Q) before T_j executes write(Q)
  - T_i executes write(Q) before T_j executes write(Q)

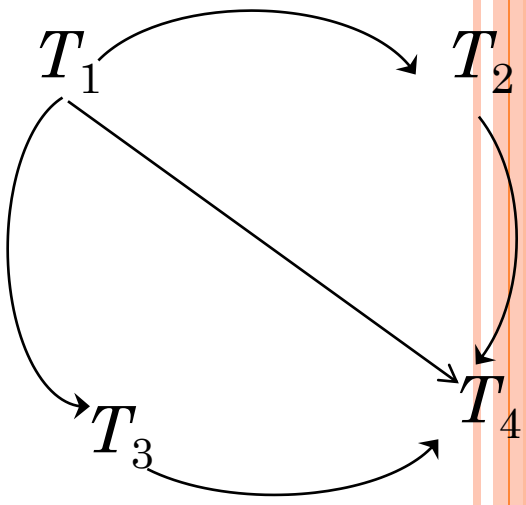
- We may label the arc by the item that was accessed.
- **Example 1**



- Now if an edge $T_1 \rightarrow T_2$ exists in the precedence graph, then, in any serial schedule S′ equivalent to S, T_1 must appear before T_2

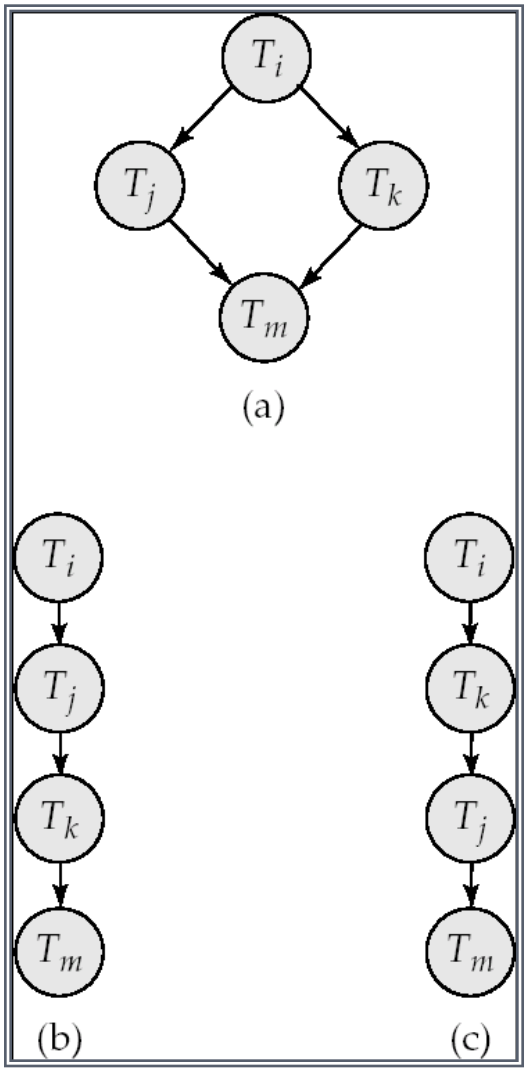
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i **reads** the initial value of Q, then in schedule S' also transaction T_i must **read** the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also a view serializable but there are view-serializable schedules that are not conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- The above schedule is view equivalent to serial schedule $<T_3, T_4, T_6>$
- Every view serializable schedule that is not conflict serializable has **blind writes**

View Serializability

- View serializability provides weaker conditions then conflict serializability
  - Still this will ensure serializability
- The key difference between view and conflict serializability appears when a transaction writes a value without reading it
- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.

Other Notions of Serializability

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- The schedule shown produces same outcome as the serial schedule $< T_1, T_5 >$
  - yet is not conflict equivalent or view equivalent to it.

Recoverability

- So far we have seen whether a schedule is acceptable from the viewpoint of consistency of the database
  - Implicit assumption- no transaction failures
- However if a transaction fails then we need to undo the effect of this transaction to ensure the atomicity property
- If concurrent execution is allowed then a transaction may depend on some other transaction
  - So in the case of a failure, if a transaction is aborted then the dependent transaction may also be aborted

Recoverable Schedules

- Let's consider the following schedule (Schedule 11)

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- It is not recoverable if T_9 commits immediately after the read
- If T_8 aborts, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.
- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions have yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work
- How to avoid cascading rollbacks?

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - *either conflict or view serializable, and*
  - *are recoverable and preferably cascadeless*
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.

  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager
- Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is ***compatible*** with locks already held on the item by other transactions
- Any number of transactions can hold *shared locks* on an item,
  - but if any transaction holds an *exclusive lock* on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made *to wait till all incompatible locks held by other transactions have been released*
  - the lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

T_1: **lock-X**(*B*);
read (*B*);
B:=B-50;
write (*B*)
unlock(*B*);
lock-X(*A*);
read (*A*);
A:=A+50;
write (*A*);
unlock(*A*);

Lock-Based Protocols (Cont.)

- Example of another transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols.
- Deadlocks are a necessary evil
  - Preferable to inconsistent states

Starvation

- **Starvation** is also possible if concurrency control manager is badly designed.
- For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.

Granting of locks

- Concurrency control manager can be designed to prevent *starvation*
- When a transaction T_i requests a lock on data item Q in a particular mode M, the concurrency control manager grants the lock provided that –
  - There is no other transaction holding a lock on Q in a mode that conflicts with M
  - There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i
- Thus a lock request will never get blocked by a lock request that is made later

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedule.
- This protocol issues lock and unlock requests in two phases:
- **Phase 1: Growing Phase**
  - A transaction may obtain locks, but may not release locks
- **Phase 2: Shrinking Phase**
  - A transaction may release locks but may not obtain any new locks
- The transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Two phase locking protocol (contd.)

- Cascading roll-back is also possible under two-phase locking.

T5	T6	T7
lock-X(A) read (A) lock-S(B) read (B) write (A) unlock (A)		
	lock-X(A) read (A) write (A) unlock (A)	
		lock-S(A) read (A)

- Cascading rollbacks can be avoided by
  - Following a modified protocol called **strict two-phase locking**. Here a transaction must hold all its *exclusive locks* till it commits/aborts.
  - **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort.

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

- Consider the following two transactions-

T_8

Read(A_1)
Read(A_2)
…
Read(A_n)
Write(A_1)

T_9

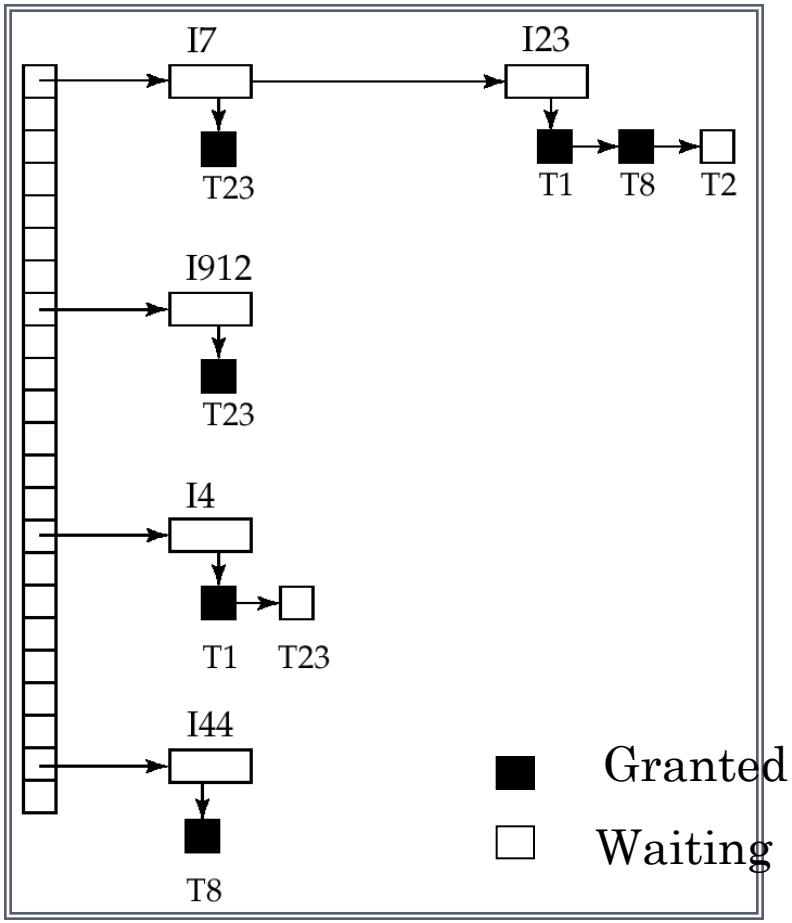
Read(A_1)
Read(A_2)
display(A_1+ A_2)

If two phase locking protocol is employed, then T_8 must lock A_1 in exclusive mode
However, T_8 needs the exclusive mode lock only at the end
So initially it can lock A_1 in shared mode and later upgrade to exclusive mode
This will allow more concurrency

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



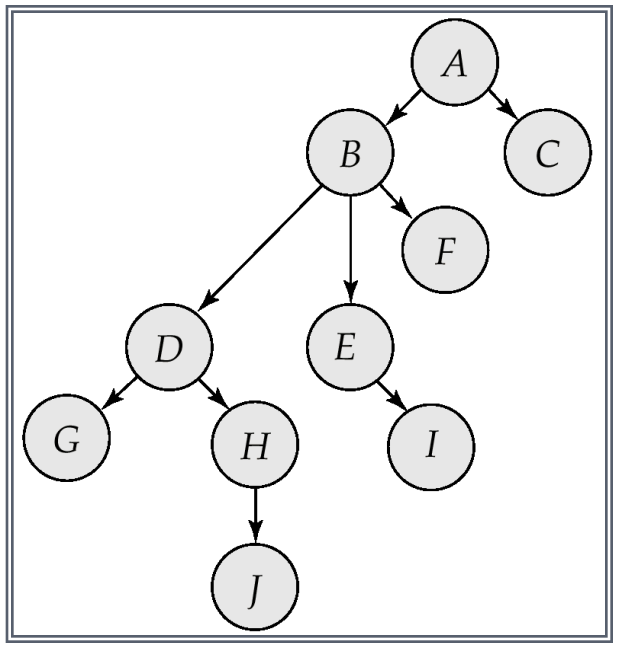
- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



SCHEDULE USING THE TREE PROTOCOL

T_{10}	T_{11}	T_{12}	T_{13}
Lock-X(B)			
	Lock-X(D)		
	Lock-X(H)		
	Unlock (D)		
Lock-X(E)			
Lock-X(D)			
Unlock(B)			
Unlock (E)			
		Lock-X(B)	
		Lock-X(E)	
	Unlock (H)		
Lock-X(G)			
Unlock (D)			
			Lock-X(D)
			Lock-X(H)
			Unlock (D)
			Unlock (H)
		Unlock (E)	
		Unlock (B)	
Unlock (G)			

Graph-Based Protocols (Cont.)

- **Advantages:**
  - The tree protocol ensures conflict serializability
  - No rollback is required as it is free from deadlock
  - Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    - shorter waiting times, and increase in concurrency
- **Disadvantages:**
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency

Deadlock Handling

- Consider the following two transactions:

  T_1: write (A) T_2: write(B)

  write(B) write(A)

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

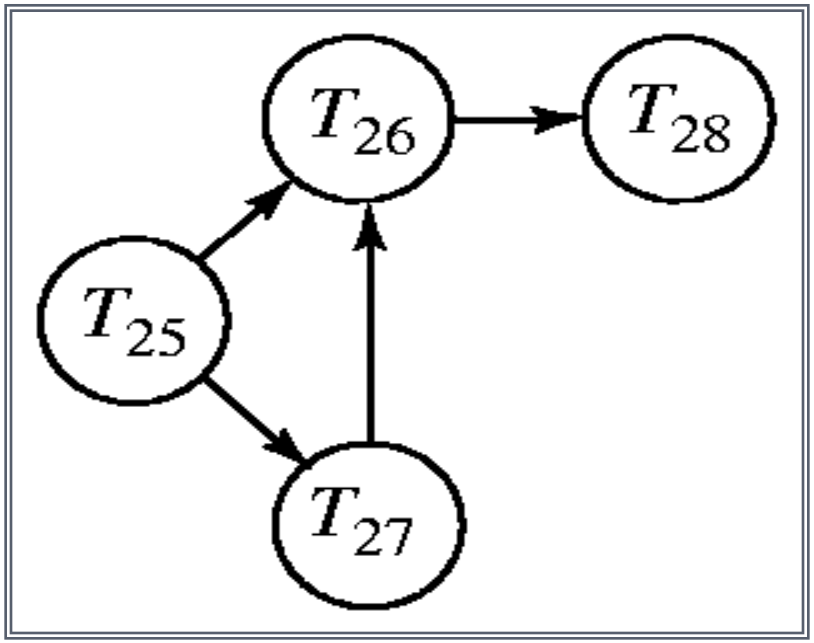
Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes
  - A rolled back transaction is restarted with its original timestamp.
  - Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes :
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus deadlocks are not possible
  - Simple to implement; but starvation is possible.
  - Also difficult to determine good value of the timeout interval.

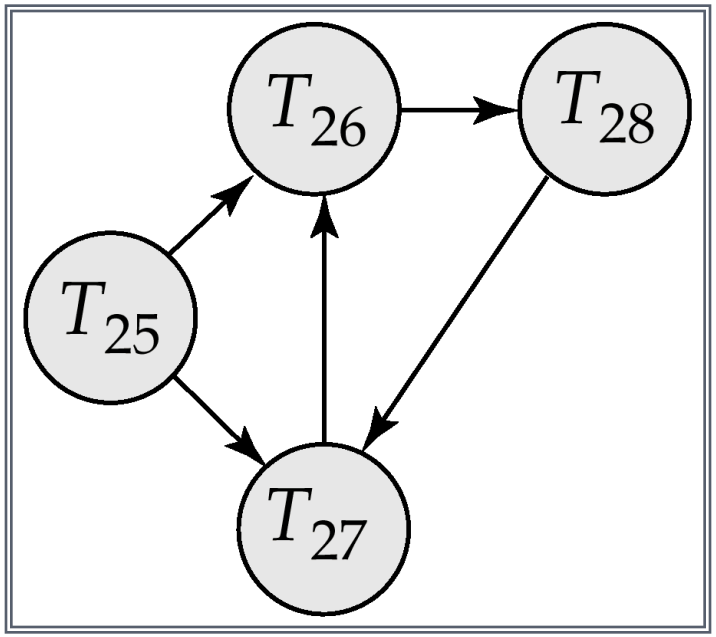
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected
  - The system must recover from the deadlock state
  - The most common solution is to roll back one or more transactions
- Actions to be taken
  - Selection of a victim
  - Rollback
  - Starvation

Selection of a victim

- Some transactions will have to be rolled back (made a victim) to break deadlock
- Select that transaction as victim that will incur minimum cost while rolling back
- Depends on
  - How long the transaction has computed, and how much longer the transaction will compute before it completes it designated task
  - How many data items the transaction has used
  - How many more data items the transactions needs for it to complete
  - How many transactions will be involved in the rollback

Rollback

- How far a particular transaction should be rolled back
  - Total rollback: abort the transaction and restarts it
  - Partial rollback: rollback the transaction only as far as necessary to break deadlock
    - The system must maintain some additional information – sequence of lock requests/grants, updates performed by the transaction, etc.

Starvation

- The selection of a victim is based on cost factor
- Starvation happens if same transaction is always chosen as victim
- As a result this transaction never completes its designated task
- A transaction can be picked as a victim only a finite no. of times
- A possible solution:
  - Include the number of rollbacks in the cost factor to avoid starvation

Recovery System

- A computer system is subject to failure from a variety of causes
- In any failure data may be lost
- So a good database system must take actions in advance to ensure that the atomicity and durability properties are preserved
- Recovery system is an integral part of a database system
- The recovery system does the followings-
  - restore the database to a consistent state
  - provide high availability

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed not to be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing
     - to ensure enough information exists to recover from failures
  2. Actions taken after a failure
     - to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- To ensure atomicity despite failures
  - we first output information describing the modifications to stable storage without modifying the database itself.
- We study the following approach:
  - **log-based recovery**
- For simplicity, we assume that transactions run serially

Log-Based Recovery

- A **log** is kept on stable storage
  - The log is a sequence of records to keep track of all the update activities in the database
- The fields of a log record
  - Transaction identifier
  - Data item identifier
  - Old value
  - New value

Log-Based Recovery (contd.)

- Update log record:
  - $<T_i, X_j, V_1, V_2>$
    - Transaction T_i has performed a write on data item X_j. X_j had old value V_1 before the write, and will have value V_2 after the write.
- Special log records to record significant events in transaction processing:
  - $<T_i$ **start**$>$
    - Transaction T_i has started
  - $<T_i$ **commit**$>$
    - Transaction T_i has committed
  - $<T_i$ **abort**$>$
    - Transaction T_i has aborted

- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A write(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
  - The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both $<T_i$ **start**> and $<T_i$ **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- **redo** operation must be idempotent
  - Executing it several times must be equivalent to executing it once
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken

- Example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)
$A := A - 50$
write (A)
read (B)
$B := B + 50$
write (B)

T_1 : **read** (C)
$C := C - 100$
write (C)

Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) redo(T_0) must be performed since $<T_0$ **commit**> is present
  - (c) **redo**(T_0) must be performed followed by redo(T_1) since $<T_0$ **commit**> and $<T_1$ commit> are present

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **undo(T_i)** restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo(T_i)** sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery

Immediate Database Modification (Cont.)

- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record $<T_i$ **start**$>$, but does not contain the record $<T_i$ **commit**$>$.
  - Transaction T_i needs to be redone if the log contains both the record $<T_i$ **start**$>$ and the record $<T_i$ **commit**$>$.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Common Problems in Recovery Procedure

- Searching the entire log is time-consuming
- Unnecessarily redo transactions which have already output their updates to the database

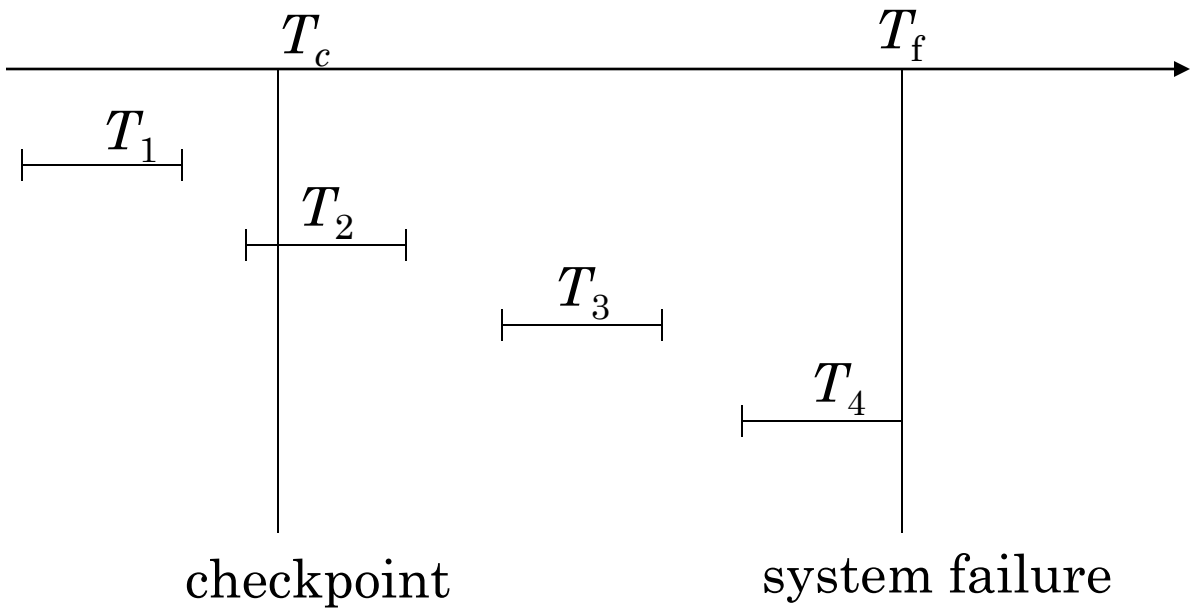
CHECKPOINTING

- Streamline recovery procedure by periodically performing **checkpointing**
- It performs the following sequence of operations-
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage
- Transactions are not allowed to perform any update actions such as writing to buffer block or writing a log record, while a checkpoint is in progress

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent **<checkpoint>** record
  2. Continue scanning backwards till a record $<T_i$ **start>** is found.
  3. Need only to consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no $<T_i$ **commit>**, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit>**, execute **redo(***T_i***)**

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone