Transaction Management (Part 2)

COMP9311 24T2; Week 8.2

By Zhengyi Yang, UNSW

Recap - Conflict Serializability

Operations *O1* and *O2* of different transactions *T1* and *T2* respectively, **conflict** if and only if some item Q is accessed by both *O1* and *O2*, and at least one of these operations write *Q.*

- ➢ For two conflicting operations, we have to keep the order, otherwise, the result can be different.
- ➢ For two non-conflicting operations, we can reorder them without changing the result.

A schedule *S* is conflict serializable if it is (conflict) equivalent to some serial schedule *S.*

In another words:

- ➢ We can reorder the *nonconflicting* operations in *S* until we form the equivalent serial schedule *S'*.

Testing Conflict Serializability

There is a simple algorithm for determining whether a particular schedule is conflict serializable or not. The algorithm looks at only the read and write operations in a schedule. (A Simplified View of Transactions)

Algorithm

Step 1: Construct a *precedence* graph.

Step 2: Check if the graph is *cyclic*:

- Cyclic: non-serializable.
- Acyclic: serializable.

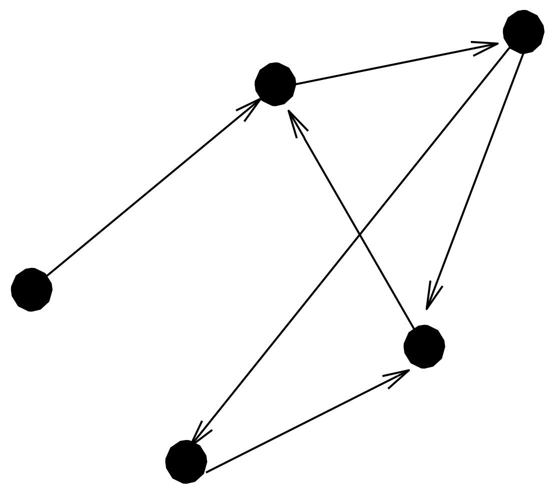
Preliminary

A *directed graph G* = (*V*, *A*) consists of

a vertex set *V*

an arc set *A* such that each arc connects two vertices.

Cyclic: G is *cyclic* if *G* contains a directed cycle.



Cyclic Graph

Serializability Testing: Precedence Graph

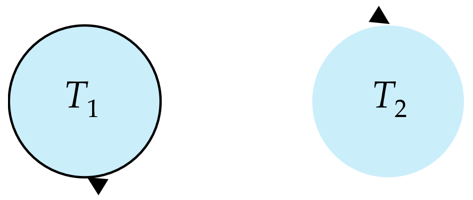
Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$ Precedence graph — a directed graph $G = (V, E)$ where the vertices (V) are the transactions.

We draw an arc from T_i to T_j, $T_i \rightarrow T_j$, if the two transactions are conflict, and T_i accessed the data item earlier.

- 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 (of a precedence graph):

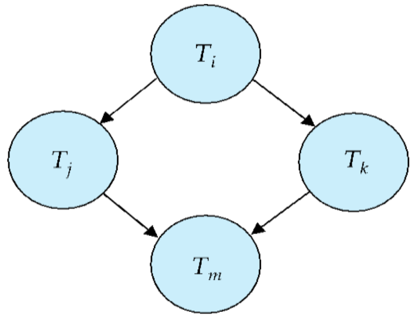


Conflict Serializability Testing

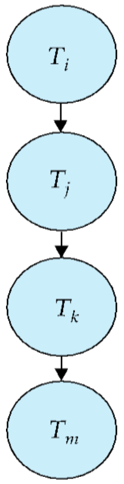
A schedule is conflict serializable if and only if its precedence graph is acyclic (cycle free).

If the precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.

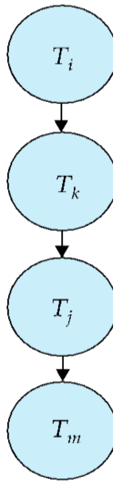
- ➢ This is a linear order consistent with the partial order of the graph.
- ➢ For (a), there are two linear orders (b) and (c).



(a)



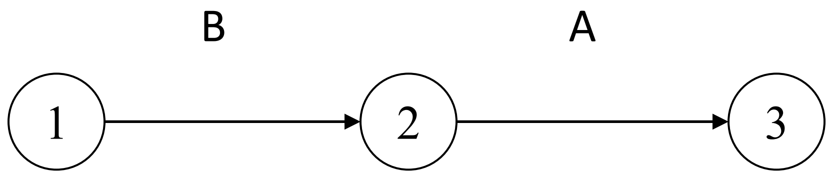
(b)



(c)

Example

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



$$\begin{tikzcd} 1 \arrow[r, "B"] & 2 \arrow[r, "A"] & 3 \end{tikzcd}$$

If there is **no** cycle in the precedence graph, this schedule is conflict-serializable

Note: Here we label the arc by the item that was accessed.

Example (2)

$$\boxed{r_2(A);\ r_1(B);\ w_2(A);\ r_2(B);\ r_3(A);\ w_1(B);\ w_3(A);\ w_2(B)}$$

$$\begin{tikzcd}
1 \arrow[r, "B"'] & 2 \arrow[l, bend right, "B"'] \arrow[r, "A"] & 3
\end{tikzcd}$$

If there is a cycle in the precedence graph, this schedule is NOT conflict-serializable

Note: Here we label the arc by the item that was accessed.

Example (3)

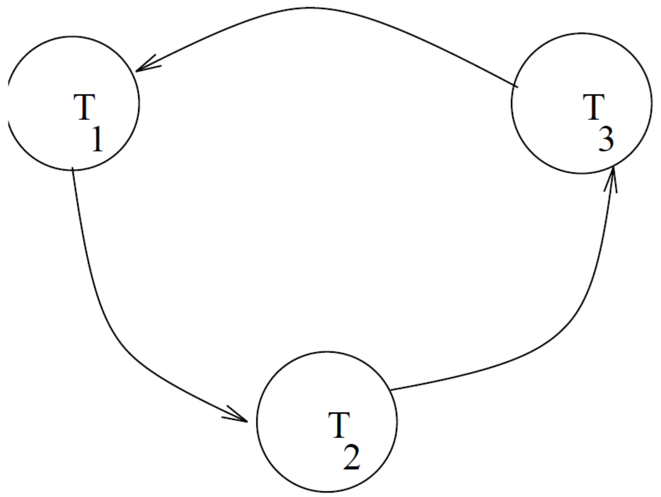
Example 1:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
write(B)		write(B)	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)	write(C)		
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

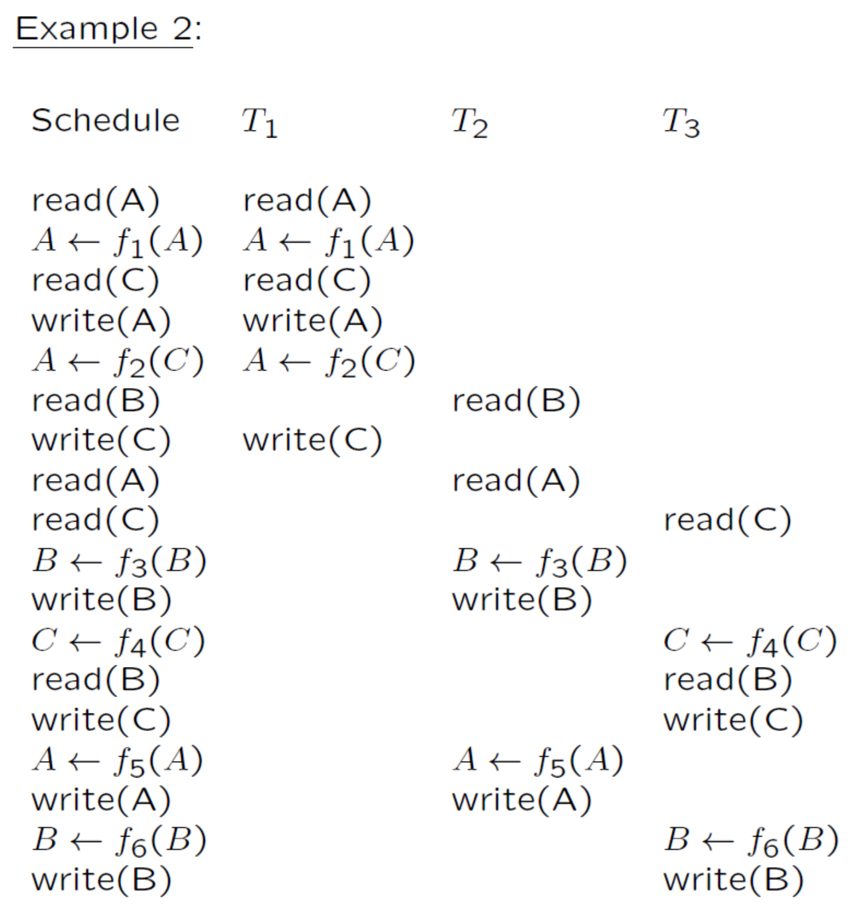
Example (3) - Answer

Example 1:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
read(B)		read(B)	
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)			read(C)
$B \leftarrow f_2(B)$		$B \leftarrow f_2(B)$	
write(B)		write(B)	
$C \leftarrow f_3(C)$			$C \leftarrow f_3(C)$
write(C)			write(C)
write(A)	write(A)		
read(B)			read(B)
read(A)		read(A)	
$A \leftarrow f_4(A)$		$A \leftarrow f_4(A)$	
read(C)	read(C)		
write(A)		write(A)	
$C \leftarrow f_5(C)$	$C \leftarrow f_5(C)$		
write(C)	write(C)		
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)


$$\begin{tikzcd} T_1 \arrow[dr] & & T_3 \arrow[ll, bend right] \\ & T_2 \arrow[ur] & \end{tikzcd}$$

Example (4)



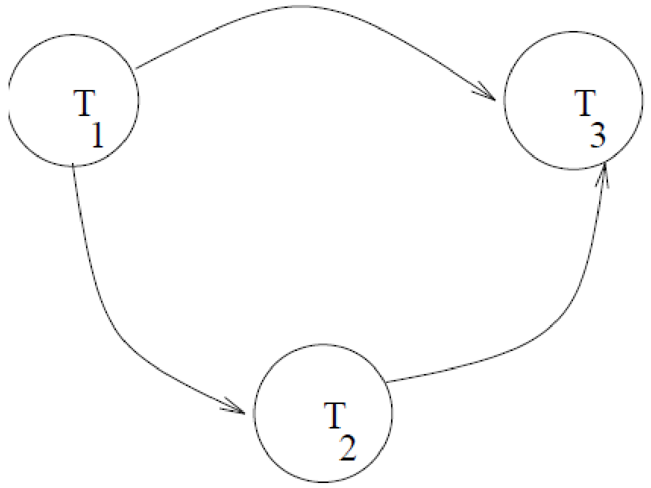
Example 2:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$		$B \leftarrow f_3(B)$	
write(B)		write(B)	
$C \leftarrow f_4(C)$			$C \leftarrow f_4(C)$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow f_5(A)$		$A \leftarrow f_5(A)$	
write(A)		write(A)	
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)

Example (4) - Answer

Example 2:

Schedule	T_1	T_2	T_3
read(A)	read(A)		
$A \leftarrow f_1(A)$	$A \leftarrow f_1(A)$		
read(C)	read(C)		
write(A)	write(A)		
$A \leftarrow f_2(C)$	$A \leftarrow f_2(C)$		
read(B)		read(B)	
write(C)	write(C)		
read(A)		read(A)	
read(C)			read(C)
$B \leftarrow f_3(B)$		$B \leftarrow f_3(B)$	
write(B)		write(B)	
$C \leftarrow f_4(C)$			$C \leftarrow f_4(C)$
read(B)			read(B)
write(C)			write(C)
$A \leftarrow f_5(A)$		$A \leftarrow f_5(A)$	
write(A)		write(A)	
$B \leftarrow f_6(B)$			$B \leftarrow f_6(B)$
write(B)			write(B)


$$\begin{tikzcd}
T_1 \arrow[rr, bend left] \arrow[dr, bend right] & & T_3 \\
& T_2 \arrow[ur, bend right] &
\end{tikzcd}$$

Concurrency Control

A database must provide a mechanism that will ensure that all possible schedules executed are serializable.

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.

Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Concurrency Control vs. Serializability Tests

Concurrency-control protocols allow concurrent schedules, ensure that the schedules are serializable.

Concurrency control protocols (generally) do not examine the precedence graph

- Instead, a **protocol** imposes a discipline that avoids non-serializable schedules. (We study such a protocol later)

There are many concurrency control protocols

- provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur

Tests for serializability help us understand if and why a concurrency control protocol is correct.

Concurrency Control

Transactions are submitted to the system

How can concurrency control help us produce correct results?

We have categorized schedules that produce correct results

- ❑ Serial schedules (not efficient)
- ❑ Non-serial schedules
  - ➢ (Conflict) Serializable (efficient, correct)
  - ➢ Non-(Conflict) serializable (not correct)

Why do we need concurrency control?

Because it is impractical to test serializability on the fly.

Locks

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*.
   - ➢ The data item can be both read as well as written.
   - ➢ Also known as a **Write Lock**.
2. ***Shared*** *(S) mode*.
   - ➢ The data item can only be read.
   - ➢ Also known as a **Read Lock**.

To use a data item, you must request the relevant locks, which each transaction can read/write the data only after the lock is granted.

Lock-Based Protocols

Lock requests are made to **concurrency-control manager**.

An exclusive lock is requested using **write_lock()** function.
A shared lock is requested using **read_lock()** function.

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

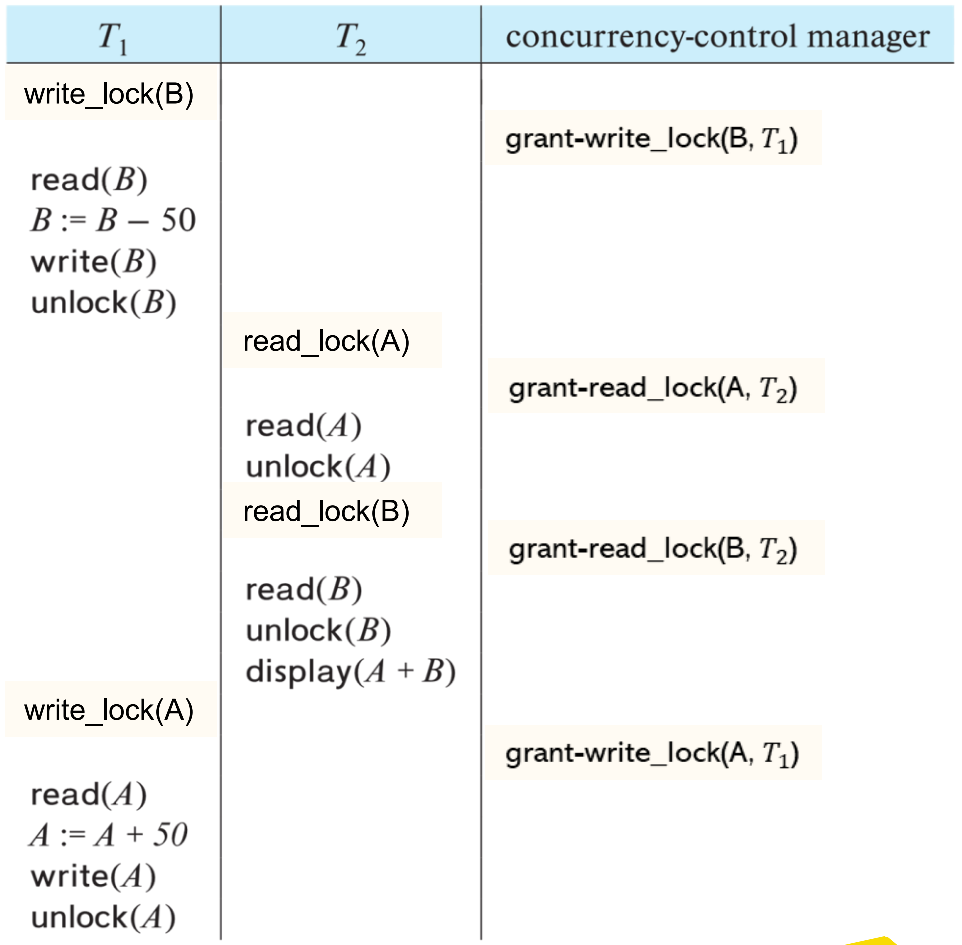
If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Goal: Locking protocols enforce serializability by restricting the set of possible schedules

Lock-Based Protocols

<u>Note:</u>

- Grants omitted in rest of slides
- Assume grant happens just before the next instruction following lock request



T_1	T_2	concurrency-control manager
write_lock(B)		
		grant-write_lock(B, T_1)

read(B)
$B := B - 50$
write(B)
unlock(B)		

	read_lock(A)	
		grant-read_lock(A, T_2)

	read(A)
	unlock(A)	

	read_lock(B)	
		grant-read_lock(B, T_2)

	read(B)
	unlock(B)
	display($A + B$)	

write_lock(A)		
		grant-write_lock(A, T_1)

read(A)
$A := A + 50$
write(A)
unlock(A)		

Lock-Based Protocols

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

This means:

- Any number of transactions can hold shared locks on an item,
- But if a transaction holds an exclusive on an item, then no other transaction may hold a lock on the item.

Lock-Based Protocols

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocol 1: A simple locking protocol

If *T* has only one operation manipulating an item *X*:

- *if read:* obtain a read lock on *X* before reading
- *if write:* obtain a write lock on *X* before writing
- unlock *X after this operation on X*

Even if *T* has several operations manipulating *X,* we obtain **one** lock still:

- if all operations on *X* are reads, obtain read lock
- if at least **one** operation on *X* is a write, obtain write lock
- unlock *X* after the **last** operation on *X*

Lock-Based Protocols

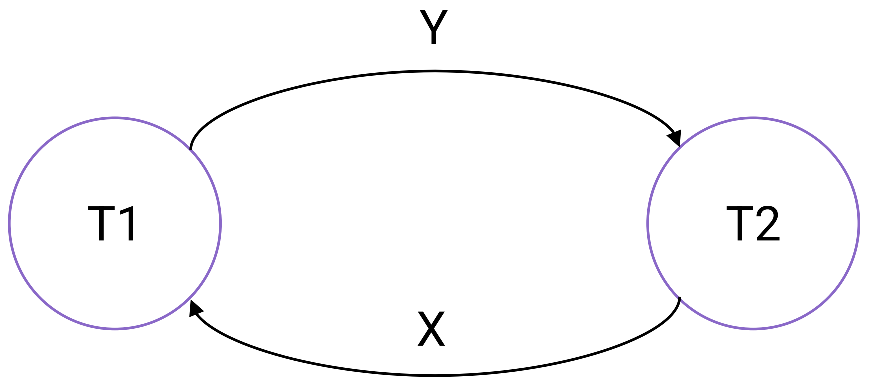
Simple locking protocol in action:

Example: Based on E/N Fig 18.3.

	T_1	T_2	
1	read_lock(Y)		
2	read(Y)		
3	unlock(Y)		
		read_lock(X)	4
		read(X)	5
		unlock(X)	6
		write_lock(Y)	7
		read(Y)	8
		$Y \leftarrow X + Y$	9
		write(Y)	10
		unlock(Y)	11
12	write_lock(X)		
13	read(X)		
14	$X \leftarrow X + Y$		
15	write(X)		
16	unlock(X)		

Lock-Based Protocols

Simple locking protocol in action:



Locking like this is ***not sufficient*** to guarantee serializability

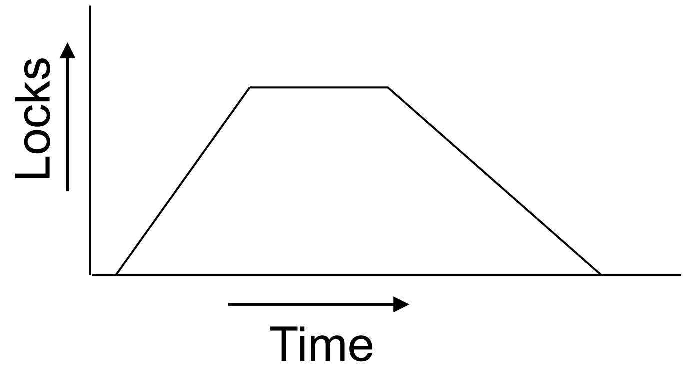
Example: Based on E/N Fig 18.3.

	T_1	T_2	
1	read_lock(Y)		
2	read(Y)		
3	unlock(Y)		
		read_lock(X)	4
		read(X)	5
		unlock(X)	6
		write_lock(Y)	7
		read(Y)	8
		$Y \leftarrow X + Y$	9
		write(Y)	10
		unlock(Y)	11
12	write_lock(X)		
13	read(X)		
14	$X \leftarrow X + Y$		
15	write(X)		
16	unlock(X)		

Two Phase Locking (2PL)

Locking protocol 2:

1. Phase 1: Growing Phase
   - Transaction obtains locks
   - Transaction does not release any locks
2. Phase 2: Shrinking Phase
   - Transaction releases locks
   - Transaction does not obtain new locks



This protocol ensures serializability: and produces conflict-serializable schedules. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Two Phase Locking (2PL)

Example of transactions performing locking:

```
T3: write_lock(B);
    read(B);
    B: = B – 50;
    write(B);
    write_lock(A);
    read(A);
    A: = A + 50;
    write(A);
    unlock(B);
    unlock(A);
```

```
T4: read_lock(A);
    read(A);
    read_lock(B);
    read(B);
    display(A+B);
    unlock(A);
    unlock(B);
```

Locking as above is *sufficient* to guarantee serializability, if unlocking is delayed to the end of the transaction.

But this introduces **Deadlocks**.

Deadlocks

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

In most locking protocols, a deadlock can exist.

Deadlocks

Consider the partial schedule to the right.

The instructions from T3 and T4 arrive at the system…

- ➢ executing **read_lock(B):**
  - ➢ T4 waits for T3 to release its lock on B
- ➢ executing **write_lock(A):**
  - ➢ T3 waits for T4 to release its lock on A

Such a situation is called a **deadlock**.

Issue: neither T3 nor T4 can make progress

	T_3	T_4
1	write_lock(B)	
2	read(B)	
3	$B := B - 50$	
4	write(B)	
5		read_lock(A)
6		read(A)
7		read_lock(B)
8	write_lock(A)	

Deadlock Prevention Scheme

Locking protocols are used in most commercial DBMSs.

We need to address this issue of deadlocks

One way to prevent deadlock is to use a **deadlock prevention protocol**.

Deadlock Prevention Scheme

Timeouts

If a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it.

Pro: Small overhead and is simple.

Con: There may not be a deadlock. The transaction may be slow just because of the high system load.

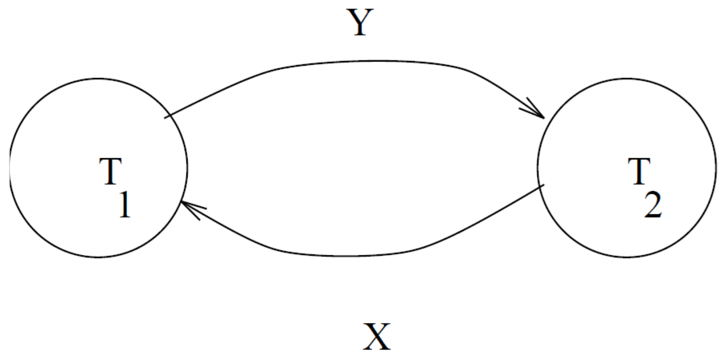
Testing for Deadlocks

Create a *wait-for graph* for currently active transactions:

- create a vertex for each transaction; and
- an arc from T_i to T_j if T_i **is waiting** for an item locked by T_j.

If the graph has a cycle, then a *deadlock* has occurred.

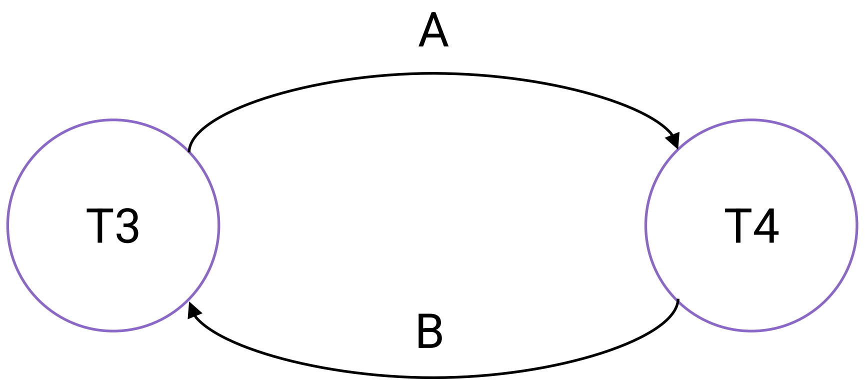
Example:



Testing for Deadlocks

Case: wait-for graph with cycle(s)

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)	


$$\begin{tikzcd}T3 \arrow[r, bend left, "A"] & T4 \arrow[l, bend left, "B"]\end{tikzcd}$$

Prevent the deadlock by abort/rollback T3 or T4

Summary

- The use of locks, combined with the 2PL protocol, guarantees serializability of schedules.
- The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which the transaction obtains the last lock.
- If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem.

Transaction Failures

Up to now, we discuss schedules assuming implicitly that there are no transaction failures.

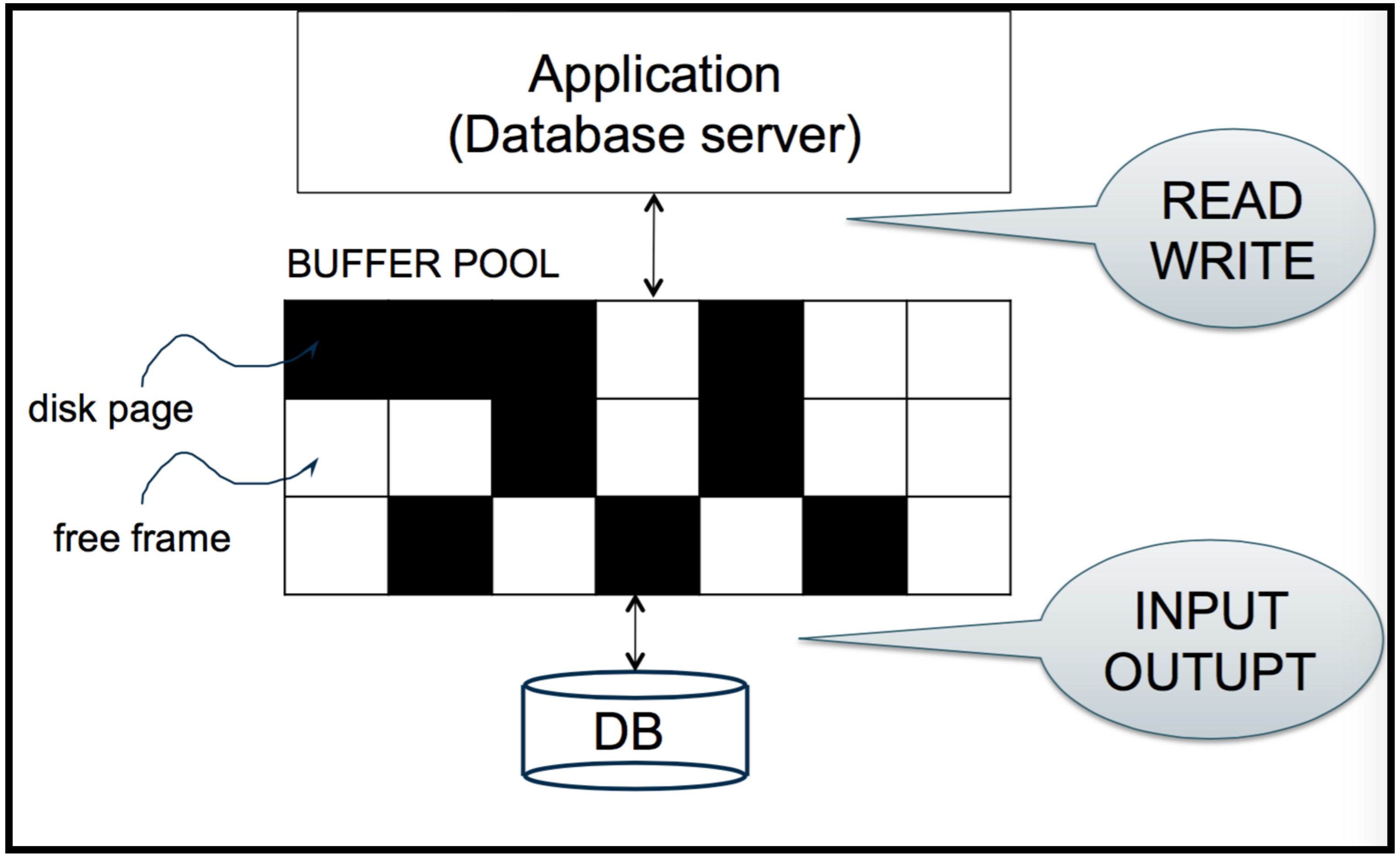
1. We need to consider the effect of transaction failures during concurrent execution.
2. Why? If transaction T fails, we need to undo the effect of this transaction to ensure the atomicity property.

Atomicity: Either all operations of the transaction are properly reflected in the database, or none are

Recall Failures

1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution.
2. **A transaction or system error.** Some operation in the transaction may cause it to fail.
3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction
4. **Concurrency control enforcement.** The concurrency control method may decide to abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions.

Buffer Management in a DBMS



Database Recovery

- The recovery strategy is to identify any changes that may cause an inconsistency in the database.
- We need to have information to *rollback* an unsuccessful transaction (undo any partial updates).

System Log

To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items.

1. The system needs to record the states information to recover failures correctly.
2. The information is maintained in a log (also called journal).
3. The system log is kept in hard disk but maintains its current contents in main memory.

System Log

Log records

1. [start transaction, *T*]: Start transaction marker, records that transaction *T* has started execution.
2. [read item, *T*, *X*]: Records that transaction *T* has read the value of database item *X*.
3. [write item, *T*, *X*, old value, new value]: Records that *T* has changed the value of database item *X* from old value to new value.

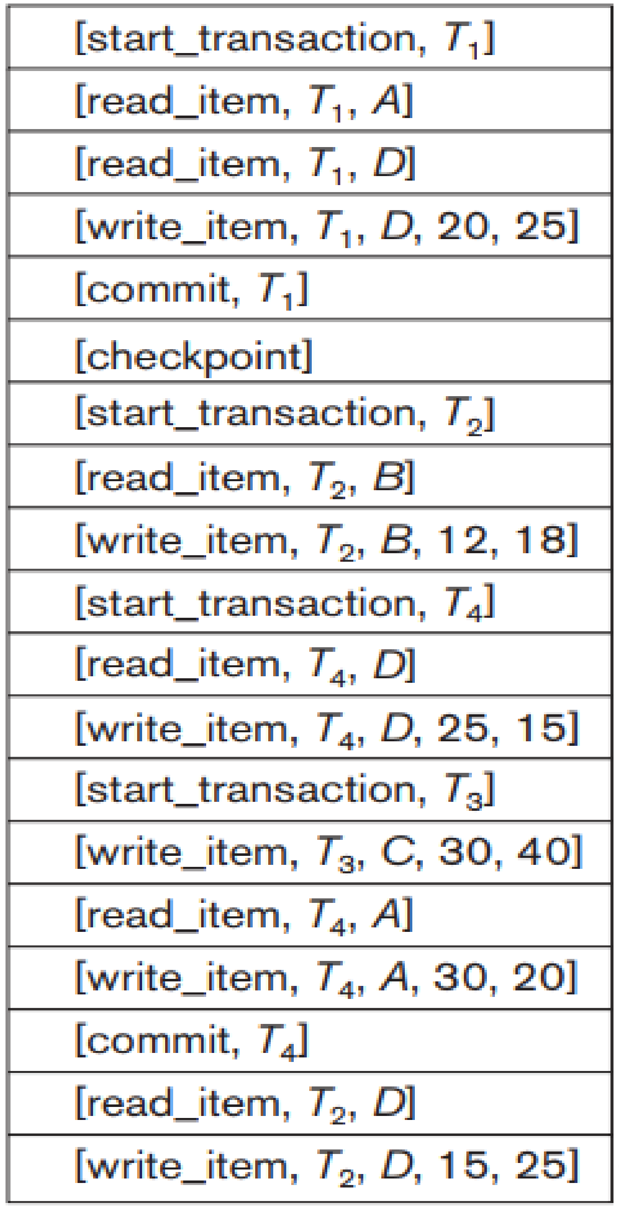
System Log

Log records (Cont.)

1. [commit, *T*]: Commit transaction marker, records that transaction *T* has completed successfully, and confirms that its effect can be committed (recorded permanently) to the database.
2. [abort, *T*]: Records that transaction *T* has been aborted.

System Log

Sample log



[start_transaction, T_1]
[read_item, T_1, A]
[read_item, T_1, D]
[write_item, T_1, D, 20, 25]
[commit, T_1]
[checkpoint]
[start_transaction, T_2]
[read_item, T_2, B]
[write_item, T_2, B, 12, 18]
[start_transaction, T_4]
[read_item, T_4, D]
[write_item, T_4, D, 25, 15]
[start_transaction, T_3]
[write_item, T_3, C, 30, 40]
[read_item, T_4, A]
[write_item, T_4, A, 30, 20]
[commit, T_4]
[read_item, T_2, D]
[write_item, T_2, D, 15, 25]

Transaction Roll Back

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction.

If any data item has been changed by the transaction and written to the database, they must be restored to their previous values.

Undo And Redo

Procedure UNDO (WRITE_OP):

- Undoing a write_item operation WRITE_OP
- Consists of examining its log entry [write_item, *T*, *X*, old_value, new_value] and setting the value of item *X* in the database to old_value.
- Undoing a number of write operations from one or more transactions from the log must proceed in the *reverse order* in which the operations were written in the log.

Procedure REDO (WRITE_OP)**:**

- Redoing a write_item operation WRITE_OP
- Consists of examining its log entry [write_item, *T*, *X*, old_value, new_value] and setting the value of item *X* in the database to new_value.

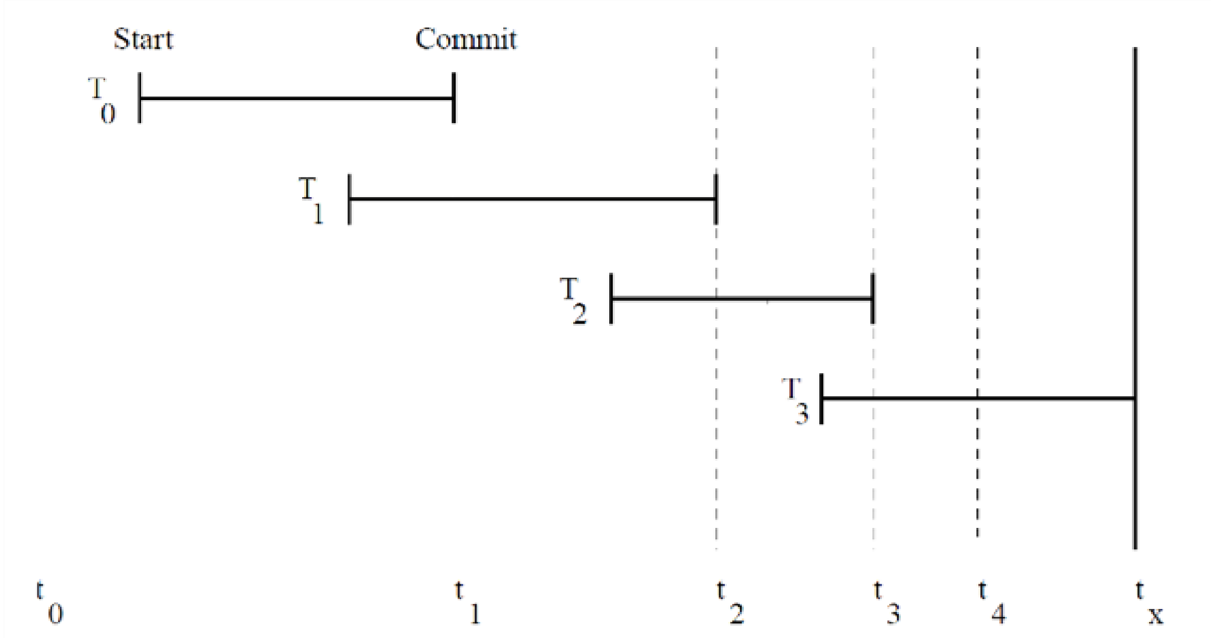
Write-Ahead Logging

Write-ahead log strategy

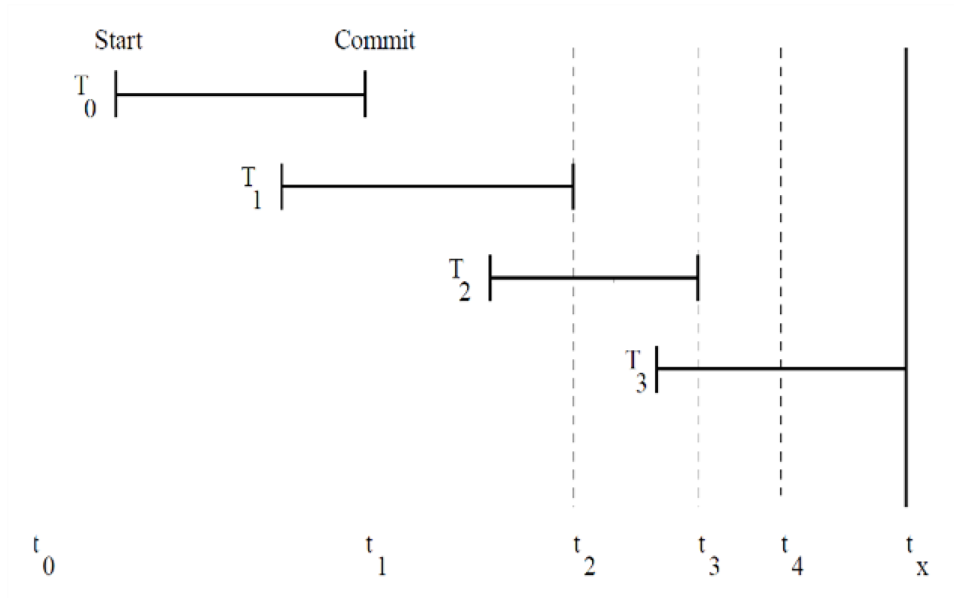
1. Must **force** the **log record** for an update **before** corresponding data page gets to the disk.
2. Must **force all log records** for a transaction **before** commit.

Log-based Recovery

- Assume that the database was recently created, the diagram shows transactions up until a crash.
- The database state will be somewhere between that at t_0 and the state at t_x (also true for log entries)



Log-based Recovery



Suppose the log was last written to disk at t_4 (shortly after t_3)

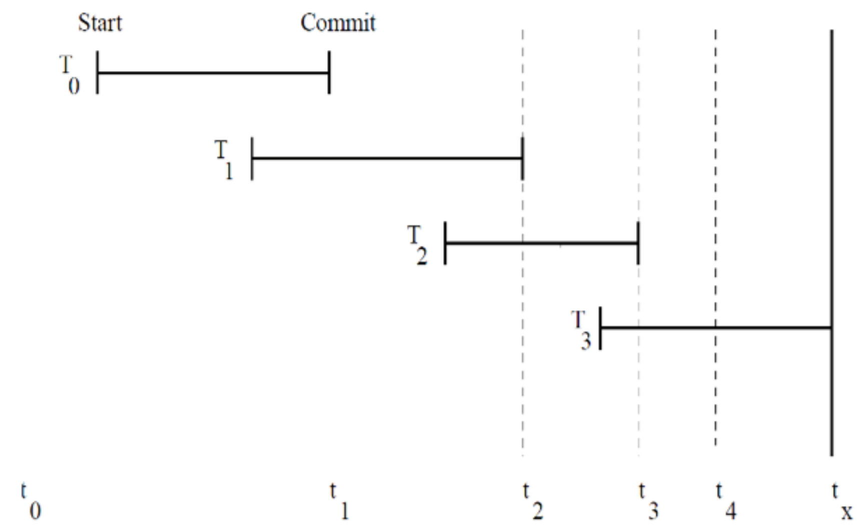
We would know that T_0, T_1 and T_2 have committed, and their effects **should be** reflected in the database after recovery.

We also know that T_3 has started, may have modified some data, but is not committed. Thus, operations of T_3 should be undone.

Rolling back (Undo) T_3

With a write-ahead strategy, we would be able to make some recovery by rolling back T_3

Step 1:
Undo the values written by T_3 to the old data values from the log

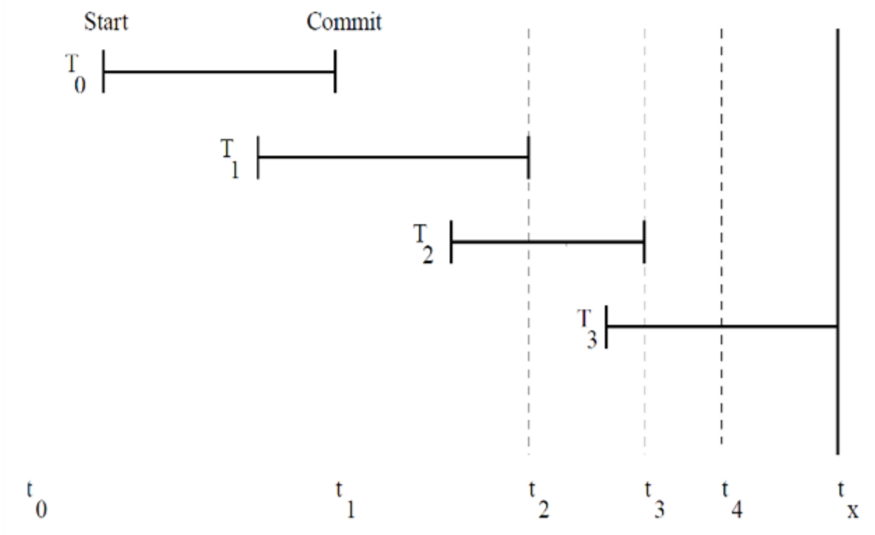


Undo helps guarantee **atomicity** in **ACID**

Redoing $T_0 \ldots T_2$

With a write-ahead strategy, we would be able to make some recovery by rolling back T_3

Step 2:
Redoing the changes made by $T_0 \ldots T_2$ using the new data values (for these committed transactions) from the log.



Redo guarantees **durability** in **ACID**

What if there are many (e.g. 100) transactions committed in the log?
We must redo all of them, and the recovery can take a long time.

Checkpoints

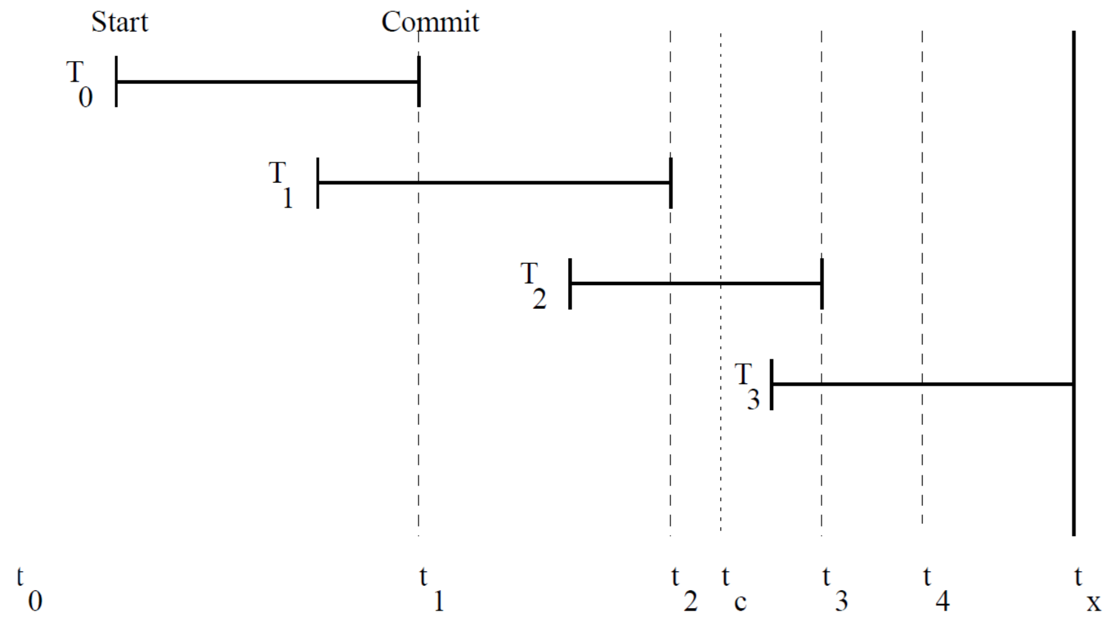
To reduce this problem, the system could take *checkpoints* at regular intervals.

Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.
3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

Log Checkpoints

In our example, suppose a checkpoint is taken at time t_c. Then on recovery we only need redo T_2.



Catastrophic Failures

1. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash.
2. **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or fire, theft, overwriting disks by mistake…
3. **Natural disaster.** Earthquake, flood, storms, tsunami…

Database Backup

The main technique used to handle such crashes is a **database backup**

- (1) whole database and (2) the log
- The latest backup copy can be reloaded and the system can be restarted.

To not lose all transactions they have performed since the last database backup. We also backup the system log at more frequent intervals than full database backup.

Disaster Recovery

A plan for disaster recovery can include one or more of the following:

- A site to be used in the event of an emergency
- A different machine on which to recover the database
- Offsite storage of either database backups, table space backups, or both, as well as archived logs.

Learning Outcomes

- Test the serializability of a schedule
- Concurrency Control Techniques:
  - Lock-based
    - Ensures serializability, but could result in deadlocks
- Database Recovery:
  - Logs, Checkpoints, Backups