

Transactions, Recovery and Concurrency (I)

Air-line Reservation

- 10 available seats vs 15 travel agents.
- How do you design a robust and fair reservation system?
 - Insufficient resources
 - Fair policy to every body
 - Robustness

Failures

Number of factors might cause failures in user requirements processing.

1. System failure:
 - Disk failure - e.g. head crash, media fault.
 - System crash - unexpected failure requiring a reboot.
2. Program error - e.g. a divide by zero.
3. Exception conditions - e.g. no seats for your reservation.
4. Concurrency control - e.g. deadlock, expired locks.

To handle failures correctly and efficiently

Each database user must express his requirements as a set of program units.

Each program unit is a *transaction* that either

- accesses the contents of the database, or
- changes the state of the database, from one consistent state to another.

- Sydney → Tokyo → LA
→ N.Y
- It does not make sense
only partial trip has tickets



Example transaction: buy a ticket from Sydney to N.Y. by JAL.

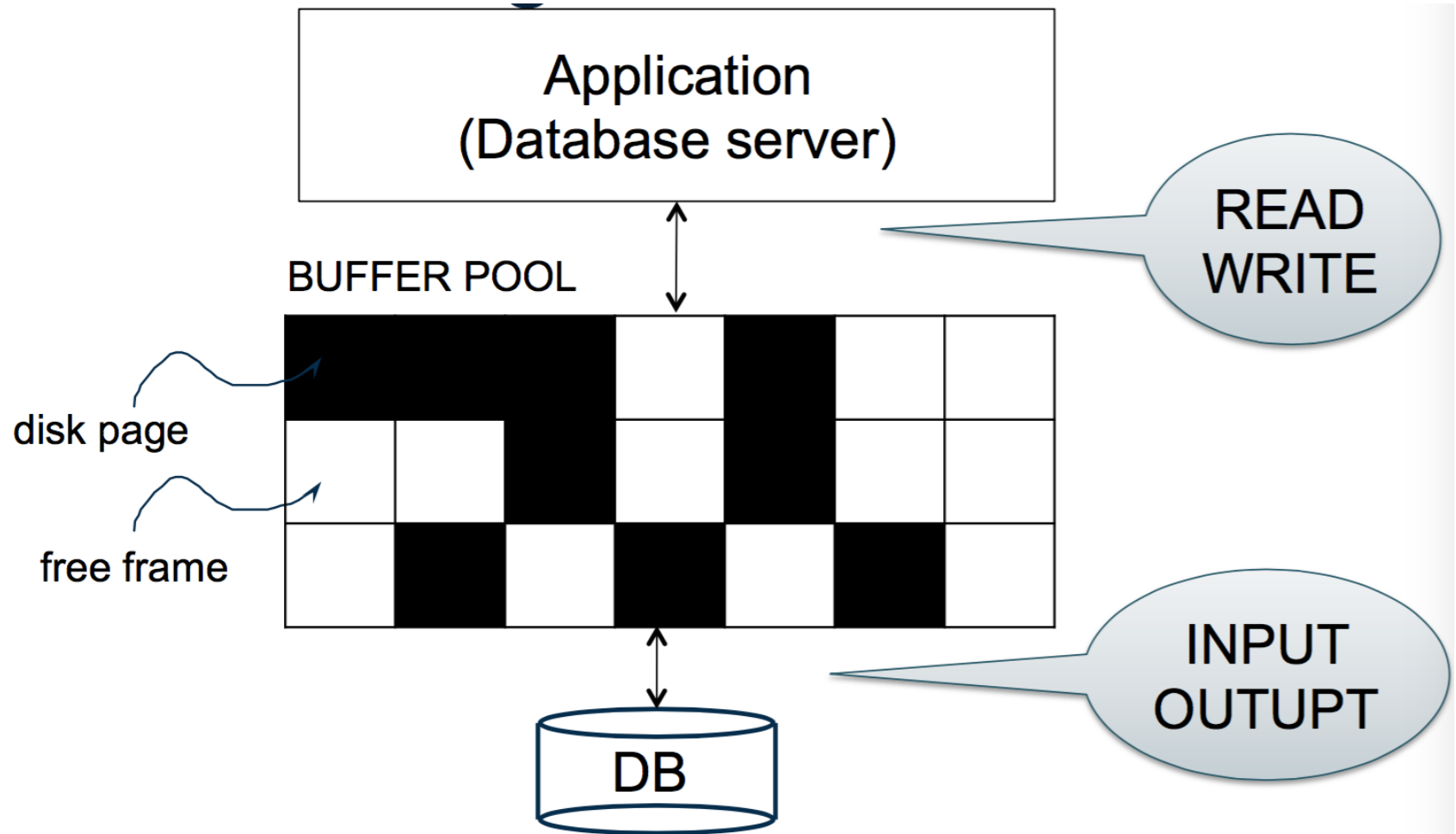
A transaction must be treated as an *atomic* unit.

Transaction Processing

Three kinds of operations may be used in a transaction:

- *Read.*
- *Write.*
- Computation.

Buffer Management in a DBMS



Read

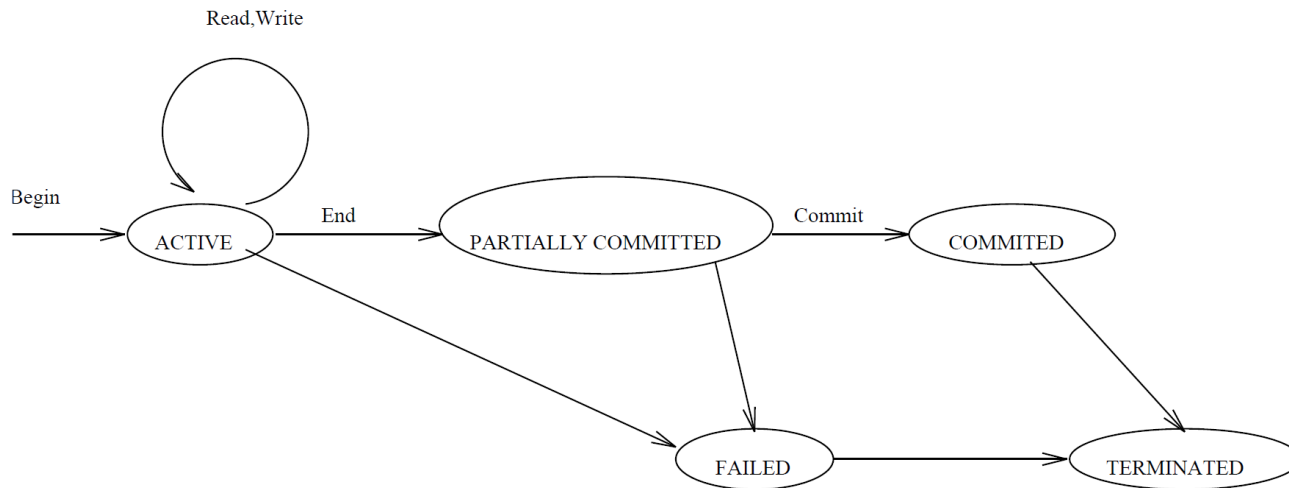
1. Compute the data block that contains the item to be read
2. Either
 - find a buffer containing the block, or
 - read from disk into a buffer
3. Copy the value from the buffer.

Write

1. Compute the disk block containing the item to be written,
2. Either
 - find a buffer containing the block, or
 - read from disk into a buffer,
3. Copy the new value into the buffer,
4. At some point (maybe later), write the buffer back to disk.

Processing States of a Transaction

- The typical processing states are illustrated in the figure below (E/N Fig 17.4):



- Partially committed point:*** At this point, check and enforce the correctness of the concurrent execution.
- Committed state:*** Once a transaction enters the committed state, it has concluded its execution successfully.

Desirable Properties of Transaction Processing **ACID**

- Atomicity: A transaction is either performed in its entirety or not performed at all.
- Consistency preservation: A correct execution of the transaction must take the database from one consistent state to another.
- Isolation: A transaction should not make its updates visible to other transactions until it is committed.
- Durability or permanency: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Problems without Enforcing ACID

- For a banking system,
 - If durability is not enforced, then a customer may lose a deposit.
 - If consistency preservation is not enforced, then the bank runs a high risk of bankrupt. E.g., run-over upper-limit.
- Below are the problems if atomicity and isolation are not enforced in a concurrent execution of transactions.

Lost Update Problem (Isolation is not enforced)

- Suppose we have these two transactions, T_1 and T_2 :

T_1 :

```
read(X)
 $X \leftarrow X + N$ 
write(X)
read(Y)
 $Y \leftarrow Y - N$ 
write(Y)
```

T_2 :

```
read(X)
 $X \leftarrow X + M$ 
write(X)
```

- Let us see what may happen if T_1 and T_2 are executed concurrently in an uncontrolled way:

Suppose initially that $X = 100$; $Y = 50$; $N = 5$ and $M = 8$.

Database	T_1	T_2
$X = 100, Y = 50$	$X = ?, Y = ?$	$X = ?$
$X = 100, Y = 50$	read(X) $X = 100, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X \leftarrow X + N$ $X = 105, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X = 105, Y = ?$	read(X) $X = 100$
$X = 100, Y = 50$	$X = 105, Y = ?$	$X \leftarrow X + M$ $X = 108$
$X = 105, Y = 50$	write(X) $X = 105, Y = ?$	$X = 108$
$X = 105, Y = 50$	read(Y) $X = 105, Y = 50$	$X = 108$
$X = 108, Y = 50$	$X = 105, Y = 50$	write(X) $X = 108$
$X = 108, Y = 50$	$Y \leftarrow Y - N$ $X = 105, Y = 45$	$X = 108$
$X = 108, Y = 45$	write(Y) $X = 105, Y = 45$	$X = 108$

- At the end of T_1 and T_2 , X should be 113, Y should be 45.
- The update $X \leftarrow X + N$ has been lost.

Incorrect Summary Problem (Isolation Issue)

T_1	T_3
	$sum \leftarrow 0$
	read(A)
	$sum \leftarrow sum + A$
	\vdots
read(X)	
$X \leftarrow X - N$	
write(X)	
	\vdots
	read(X)
	$sum \leftarrow sum + X$
	read(Y)
	$sum \leftarrow sum + Y$
	\vdots
read(Y)	
$Y \leftarrow Y + N$	
write(Y)	
	\vdots

- Here the sum calculated by T_3 will be wrong by N .

The Temporary Update Problem

Database	T_1	T_2
$X = 100, Y = 50$	$X = ?, Y = ?$	$X = ?$
$X = 100, Y = 50$	read(X) $X = 100, Y = ?$	$X = ?$
$X = 100, Y = 50$	$X \leftarrow X + N$ $X = 105, Y = ?$	$X = ?$
$X = 105, Y = 50$	write(X) $X = 105, Y = ?$	$X = ?$
	FAILS	
$X = 105, Y = 50$		read(X) $X = 105$
$X = 105, Y = 50$		$X \leftarrow X + M$ $X = 113$

Recover from the disk



Several possibilities for what might happen next:

Database	T_1	T_2
X = 105, Y = 50		X = 113
X=100, Y=50	Case 1: DBMS undoes T_1	
X=113, Y=50		Write (X) X= 113

Database	T_1	T_2
X = 105, Y = 50		X = 113
X=105, Y=50	Case 2: DBMS does nothing to T_1	
X=113, Y=50		Write (X) X= 113

- Case 1&2, only half of T_1 has been executed.
- Case 3, T_1 & T_2 have been lost.

Database	T_1	T_2
X = 105, Y = 50		X = 113
X= 105, Y = 50		X = 113
X=100, Y=50	Case 3: DBMS undoes T_1	
		Write (X), X= 113 X= 100

Recover from Failures

Ensure ACID

Log-based Recovery

- Undo logging
- Redo logging
- Undo/Redo logging

System Log

- System Log
 - The system needs to record the states information to recover failures correctly.
 - The information is maintained in a log (also called journal or audit trail).
 - The system log is kept in hard disk but maintains its current contents in main memory.

System Log

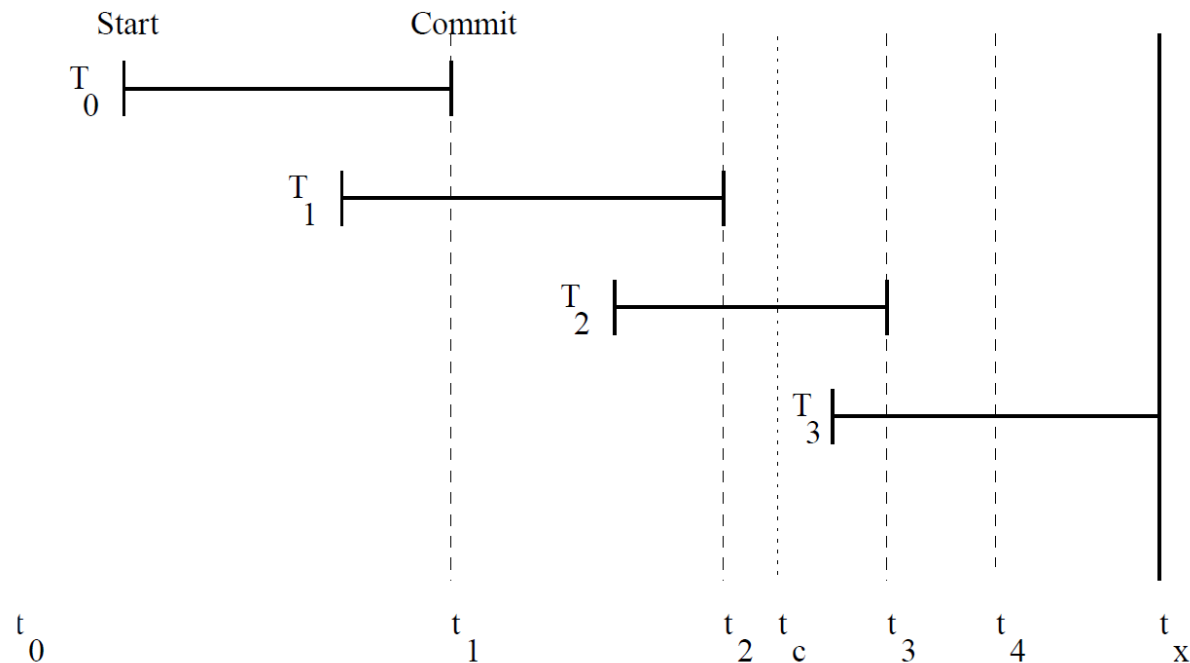
- Start transaction marker [start transaction, T]: Records that transaction T has started execution.
- [read item, T , X]: Records that transaction T has read the value of database item X .
- [write item, T , X , old value, new value]: Records that T has changed the value of database item X from old value to new value.
- Commit transaction marker [commit, T]: Records that transaction T has completed successfully, and asserts that its effect can be committed (recorded permanently) to the database.
- [abort, T]: Records that transaction T has been aborted.

System Log (Cont'd)

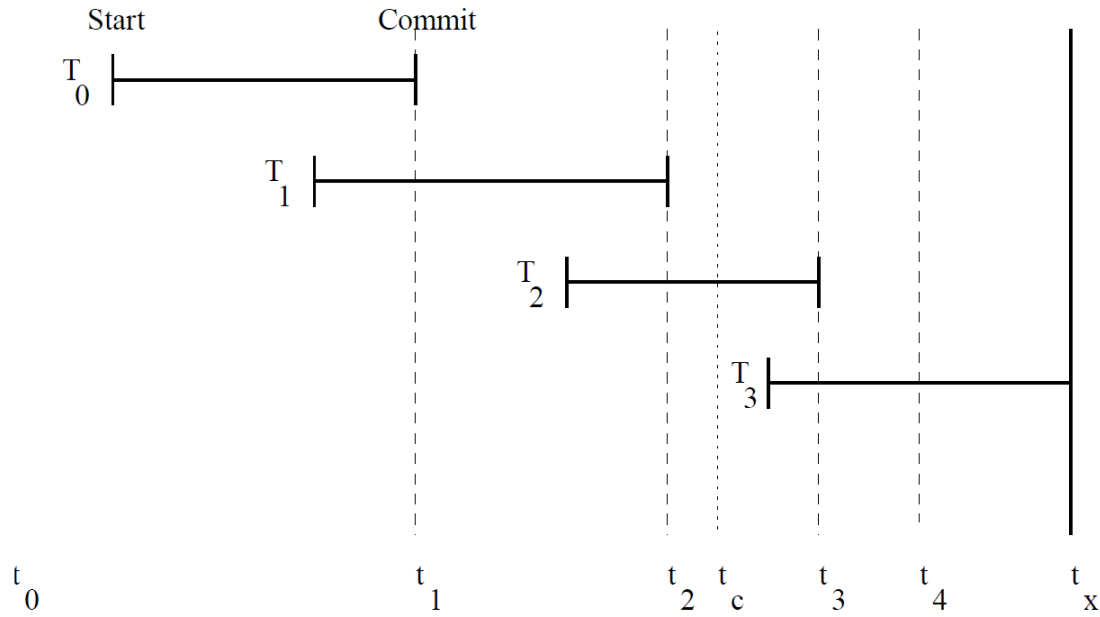
- In fact some other entries (rollback, undo, redo) are also required for a recovery method.
- These entries allow the recovery manager to *rollback* an unsuccessful transaction (undo any partial updates).

Recovery

- Let us see how the log might be used to recover from a system crash.
- The diagram below shows transactions between the last system backup and a crash.



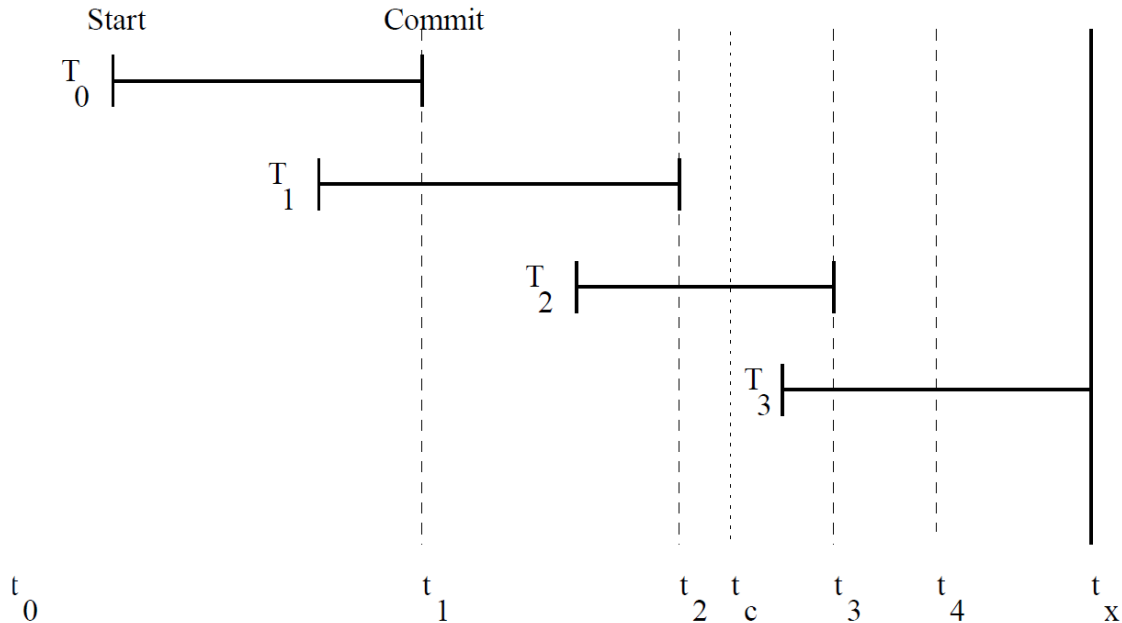
Recovery (Cont'd)



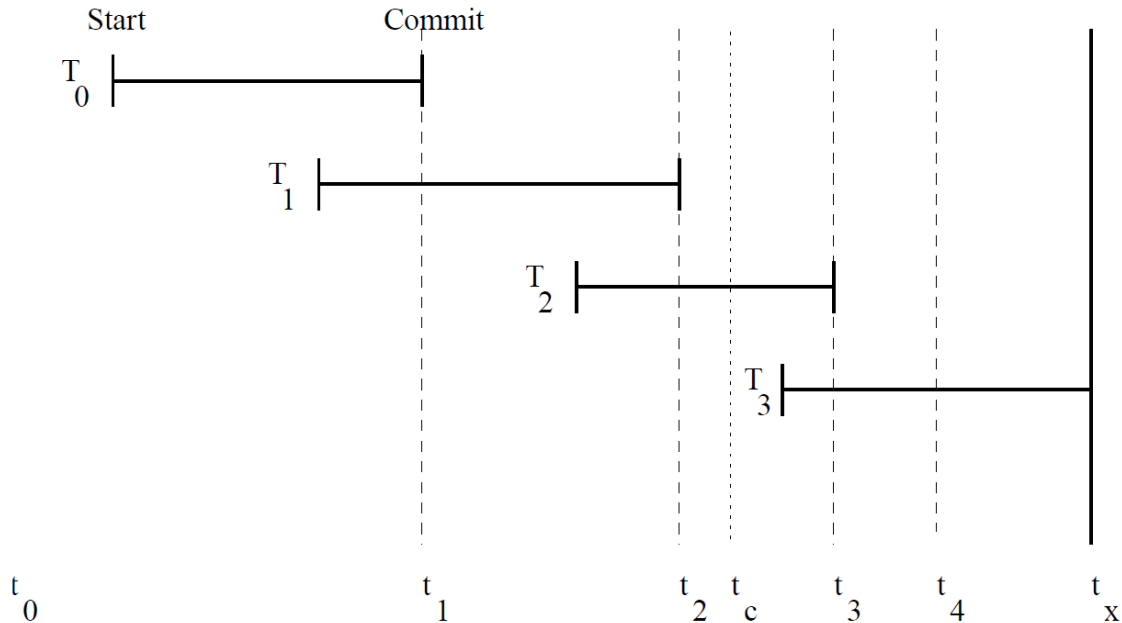
- The database on disk will be in a state somewhere between that at t_0 and the state at t_x .
- The same is also true for log entries.

Recovery (Cont'd)

- We will assume that the *write-ahead log strategy* is used. This means that
 - old data values must be force-written to the log (i.e. the buffer must be copied to disk) before any change can be made to the database, and
 - the transaction is regarded as committed when the new data values and the commit marker have been force-written to the log.
- Thus the log is force-written at least at t_1 , t_2 and t_3 in the above.



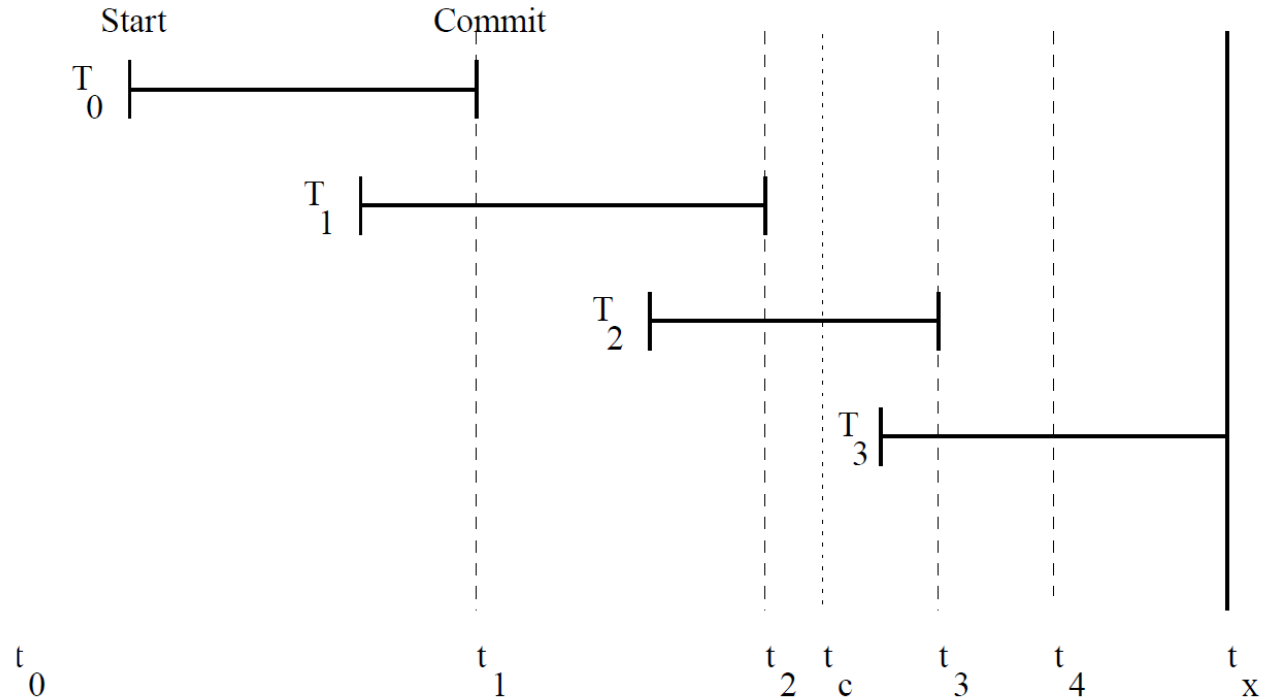
- Suppose the log was last written to disk at t_4 .
- By examining the log:
 1. We know that T_0, T_1 and T_2 have committed and their effects should be reflected in the database after recovery.
 2. But we do not know whether the effects of T_0, T_1 and T_2 were reflected at the time of the crash.
 3. We also know that T_3 has started, may have modified some data, but is not committed. Thus T_3 should be undone.



- The database can be recovered by rolling back T_3 using the old data values from the log, and redoing the changes made by $T_0 \dots T_2$ using the new data values (for these committed transactions) from the log.
- Notice that instead of rolling back, the database could have been restored from the backup. This might be necessary in the event of a disk crash for example (for this reason, the log should be stored on an independent disk pack).

Checkpoints

- Notice also that using this system, the longer the time between crashes, the longer recovery may take.
- To avoid this problem, the system may take *checkpoints* at regular intervals.
- To do this:
 - a *start of checkpoint* marker is written to the log, then
 - the database updates in buffers are force-written, then
 - an *end of checkpoint* marker is written to the log.



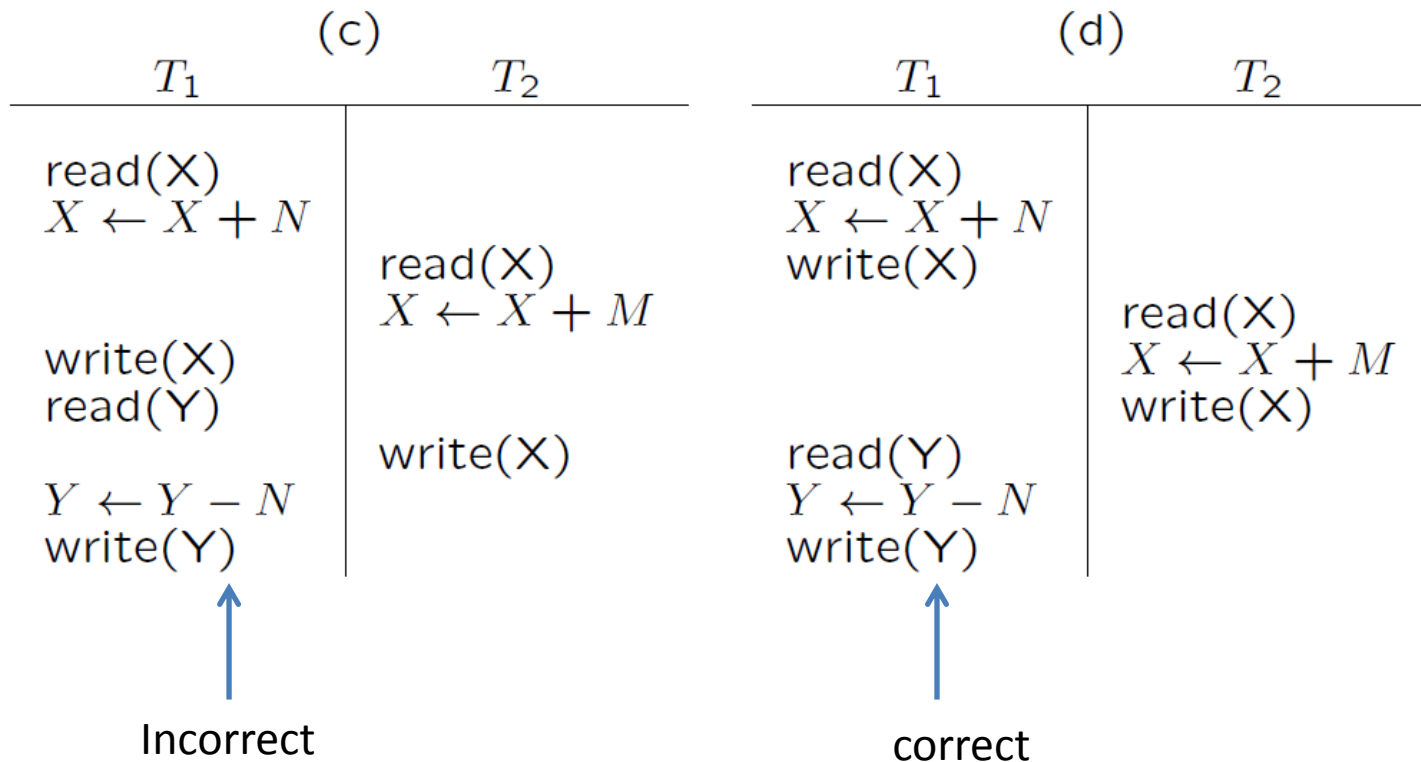
In our example, suppose a checkpoint is taken at time t_c . Then on recovery we only need redo T_2 .

Schedules of Transactions

- To fully utilise resources, desirable to interleave the operations of transactions in an appropriate way.
- For example, if one transaction is waiting for I/O to complete, another transaction can use the CPU.
- A *schedule* S of the transactions T_1, \dots, T_n
 - is a sequential ordering of the operations of T_1, \dots, T_n , and
 - preserves the ordering of operations in each transaction T_i .

(a)		(b)	
T_1	T_2	T_1	T_2
read(X) $X \leftarrow X + N$ write(X) read(Y) $Y \leftarrow Y - N$ write(Y)	read(X) $X \leftarrow X + M$ write(X)	read(X) $X \leftarrow X + N$ write(X) read(Y) $Y \leftarrow Y - N$ write(Y)	read(X) $X \leftarrow X + M$ write(X)

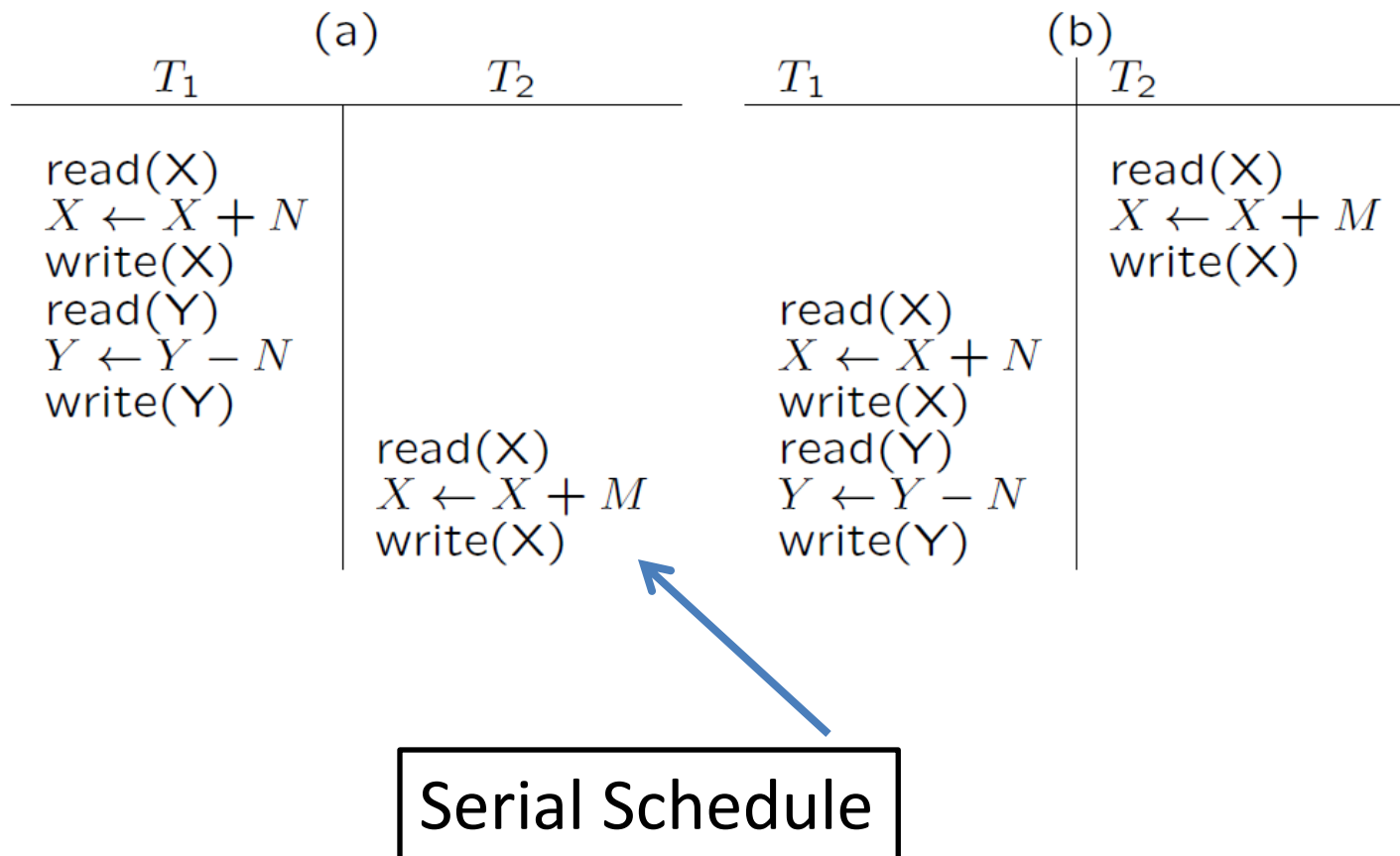
cannot swap read (X) and write (X)

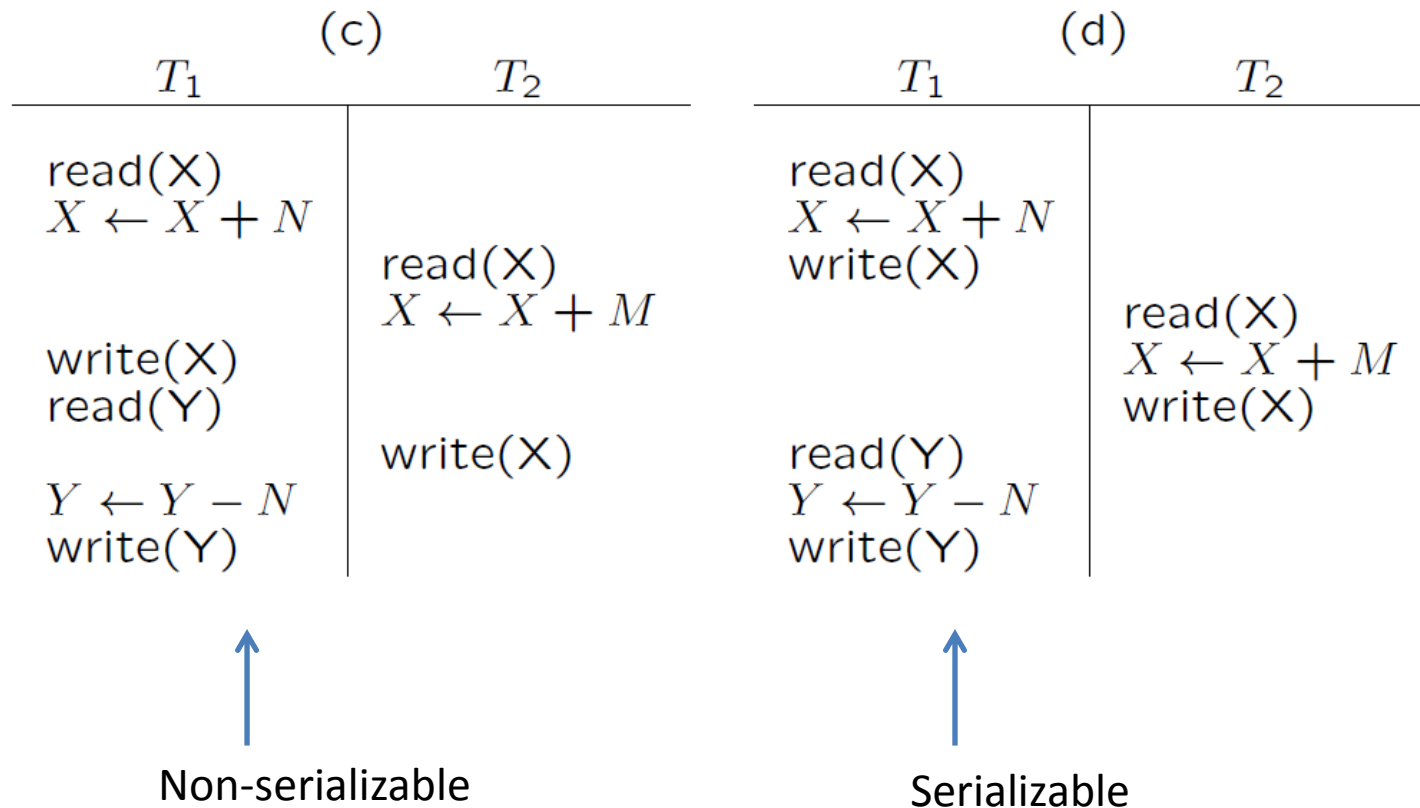


- As we have seen, if operations are interleaved arbitrarily, incorrect results may occur.
- However, it is reasonable to assume that schedules (a) and (b) in the figure will give correct results (as long as the transactions are independent).
- (a) and (b) are called *serial* schedules, and we will assume that *any serial schedule is correct*.
- Notice that schedule (d) always produces the same result as schedules (a) and (b), so it should also give correct results.
- A schedule is *serializable* if it always produces the same result as some serial schedule. (see E/N 17.5.1 for a formal definition).
- Notice that schedule (c) is not serializable.

Scheduling Transactions

- Schedule and Complete Schedule?
- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule over a set S of transactions is equivalent to some serial execution of the set of committed transactions in S .
(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)





Scheduling Transactions (Cont.)

- Recoverable schedule (RS): Transactions commit only after (and if) all transactions whose changes they read commit.

EX1: T1.R(X), T1.W(X), T2. R(X), T2.W(X), COMMIT.T2.

EX1 is not recoverable.

EX2: T1.R(X), T1.W(X), T2. R(X), T2.W(X), COMMIT.T1. Recoverable!

- Avoid cascading aborts (ACA): Transactions read only the changes of committed transactions.

EX3: T1.R(X), T1.W(X), T2. R(X), T2.W(X)... EX3 is not ACA.

EX4: T1.R(X), T1.W(X), COMMIT.T1, T2. R(X), T2.W(X)... ACA!

- Strict schedules (SS): A value written by a transaction is not read or overwritten by other transactions until T either aborts or commits.

EX5: T1.R(X), T1.W(X), T2.W(X)... EX5 is RS and ACA but not SS.

EX6: T1.R(X), T1.W(X), COMMIT.T1, T2.W(X)... EX6 is SS.

Note: SS is ACA and ACA is RS but not vice versa.

Check Serializability

- When there are only two transactions, there are only two serial schedules - for n transactions there will be $n!$.
- Fortunately there is an efficient algorithm to check whether a schedule is serializable without checking all these possibilities.

Conflict Serializable Schedules

- Two schedules are *conflict equivalent* if:
 - Involve the same actions of the same transactions
 - Every pair of conflicting actions is ordered the same way
- Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule

(d)

T_1	T_2
read(X) $X \leftarrow X + N$ write(X)	read(X) $X \leftarrow X + M$ write(X)
read(Y) $Y \leftarrow Y - N$ write(Y)	

(a)

T_1	T_2
read(X) $X \leftarrow X + N$ write(X)	
read(Y) $Y \leftarrow Y - N$ write(Y)	
	read(X) $X \leftarrow X + M$ write(X)

View Serializability

- Schedules S1 and S2 are *view equivalent* if:
 - If T_i reads initial value of A in S1, then T_i also reads initial value of A in S2
 - If T_i reads value of A written by T_j in S1, then T_i also reads value of A written by T_j in S2
 - If T_i writes final value of A in S1, then T_i also writes final value of A in S2
- A schedule is *view serializable* if view equivalent to a serial schedule.

T1:	R(A)	W(A)
T2:	W(A)	
T3:		W(A)


T1:	R(A),W(A)
T2:	W(A)
T3:	W(A)

Properties of Serizability


- View Serializability does not have monotonic property; that is, a schedule is view serializable but its sub-schedule may not necessarily view serializable.

Example:

Yes!


T1: R (A) W(A)
T2: W (A)
T3: W(A)

No!


T1: R (A) W(A)
T2: W (A)

- If no blind writes, conflict serializability is equivalent to view serializability .

Check Conflict Serializability

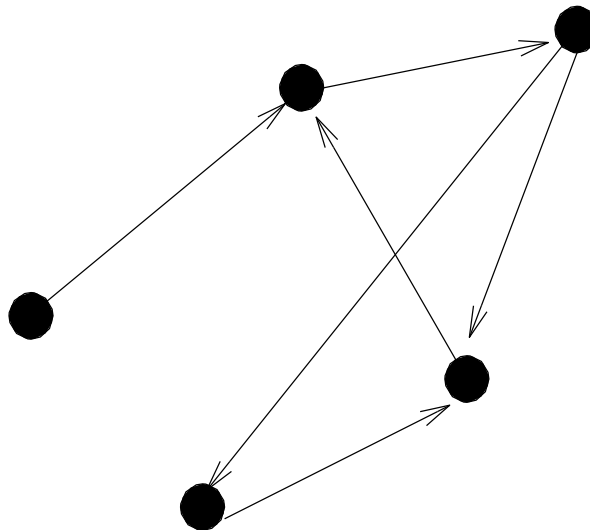
- *Algorithm*

Step 1: Construct a *schedule* (or *precedence*) graph – a *directed graph*.

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

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

- A *directed graph* $G = (V, A)$ consists of
 - a vertex set V , and
 - an arc set A such that each arc connects two vertices.
- G is *cyclic* if G contains a directed cycle.



Cyclic Graph

Construct a Schedule Graph $G_S = (V, A)$ for a schedule S

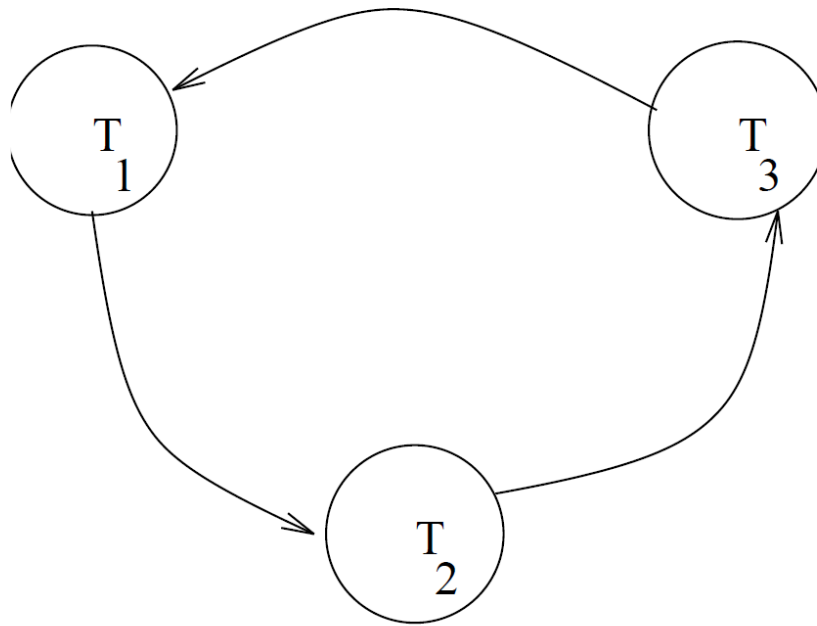
1. A vertex in V represents a transaction.
2. For two vertices T_i and T_j , an arc $T_i \rightarrow T_j$ is added to A if
 - there are two *conflicting* operations $O_1 \in T_i$ and $O_2 \in T_j$,
 - in S , O_1 is before O_2 .

Two operations O_1 and O_2 are *conflicting* if

- they are in different transactions but on the same data item,
- one of them must be a write.

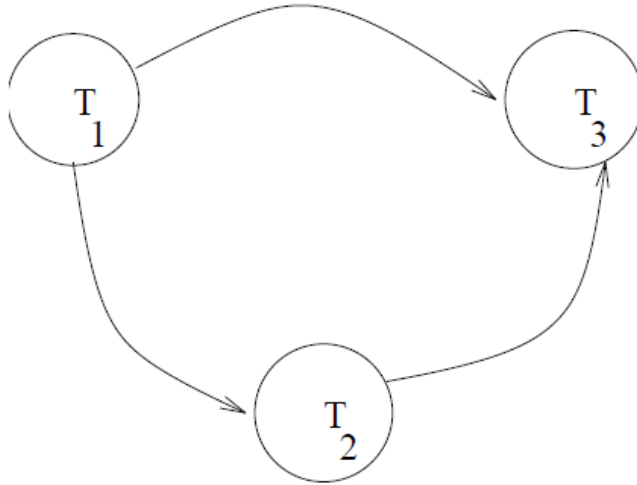
Example 1:

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 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)



- Unfortunately, testing for serializability on the fly is not practical.
- Instead, a number of protocols have been developed which ensure that if every transaction obeys the rules, then *every* schedule will be serializable, and thus correct.

- SS is serializable?

➤ irrelevant!

Example:

T1.R(X), T2.R (X), T1.W(X), COMMIT.T1,
T2.W(X), COMMIT.T2