21.9

Serial Schedule:

A schedule “S” is referred as serial, for each transaction “T” participating in schedule, the operations of T must be executed **consecutively** in schedule.

So from this perspective, it is clear that only one transaction at a time is active and whenever if that transaction is committed, then it initiates the execution of next transaction.

Serializable schedule:

The schedule is referred as “serializable schedule. When a schedule t T be a set of n transactions (T1, T2,…, Tn), is serializable and if it is equivalent to n transactions executed serially.

Consider that possibly there are “n” serial schedule of “n” transactions and moreover there are possibly non-serial schedules. If two disjoined groups of the nonserial schedules are formed then it is equivalent o one or more of the serial schedules. Hence, the schedule is referred as serializable.

Reason for the correctness of serial schedule:

A serial schedule is said to be correct on the assumption of that each transactions is **independent** of each other. So according to the “consistency preservation” property, when the transaction runs in isolation, it is executed from the beginning to end from the other transaction .Thus, the output is correct on the database. Therefore a set of transaction executed one at a time is correct.

Reason for the correctness of serializable schedule:

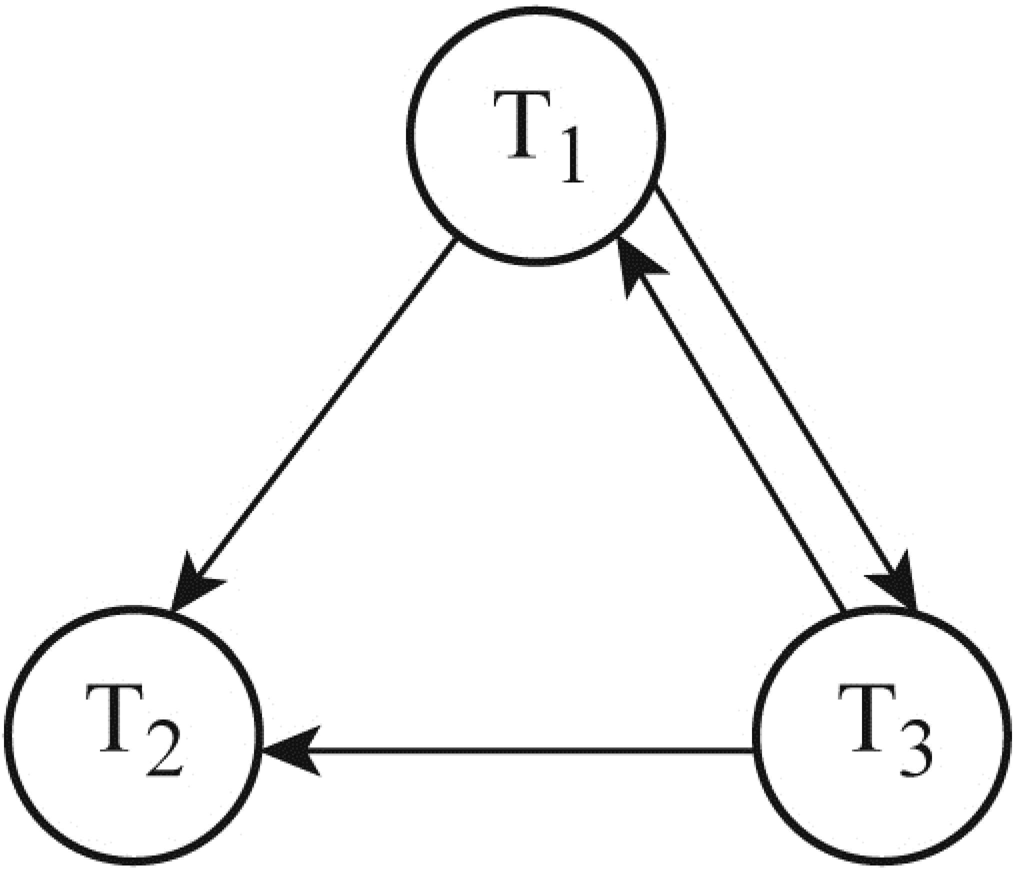
The simple method to prove the correctness of serializable schedule is that to prove the satisfactory definition. In this definition, it compares the results of the schedules on the database, if both produce same final state of database. Then, two schedules are equivalent and it is proved to be serializable.

Therefore, the serializable schedule is correct when the two schedules are in the same order.

21.22 (b, c, d)

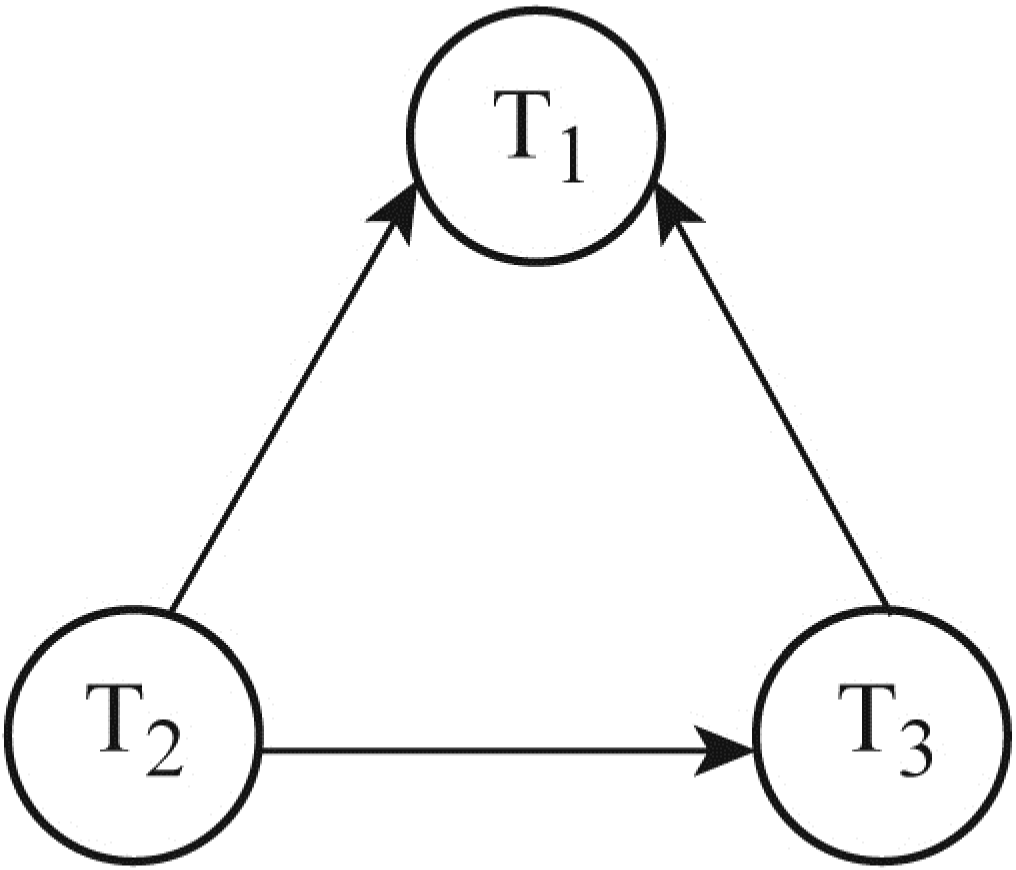
Use the sequence of conflict graph to determine whether the sequence is a serializable schedule.

b.



There is a circle between T1 and T3, thus **non-serializable**.

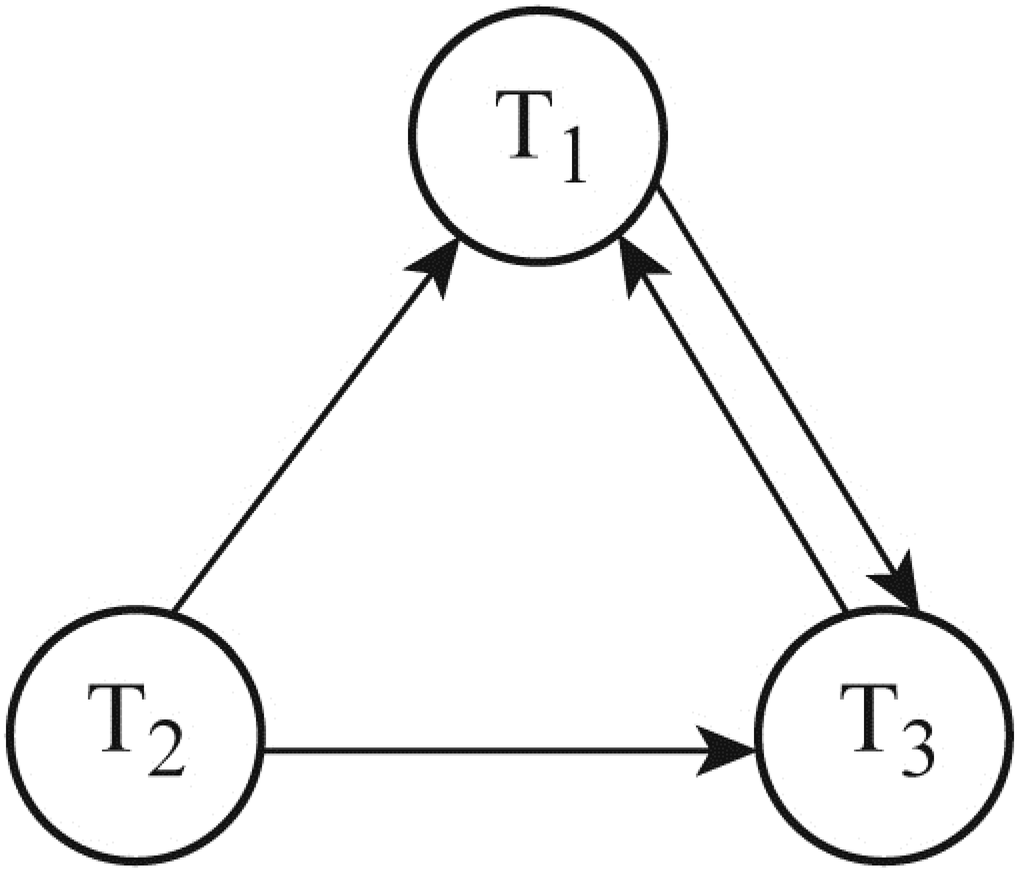
c.



Serializable. T2->T3->T1

r2(X), r3(X), w3(X), r1(X), w1(X)

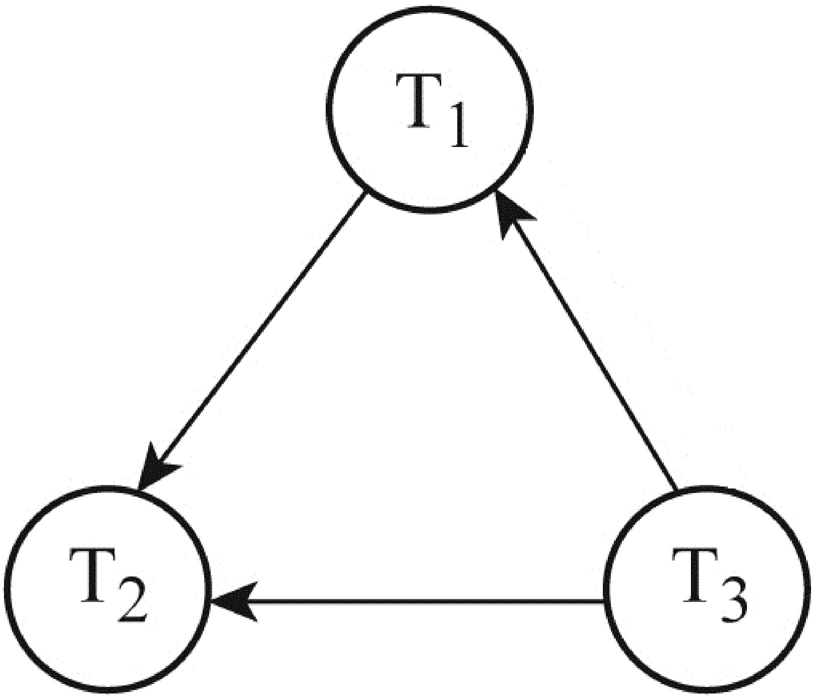
d.



There is a circle between T1 and T3, thus **non-serializable**.

21.23

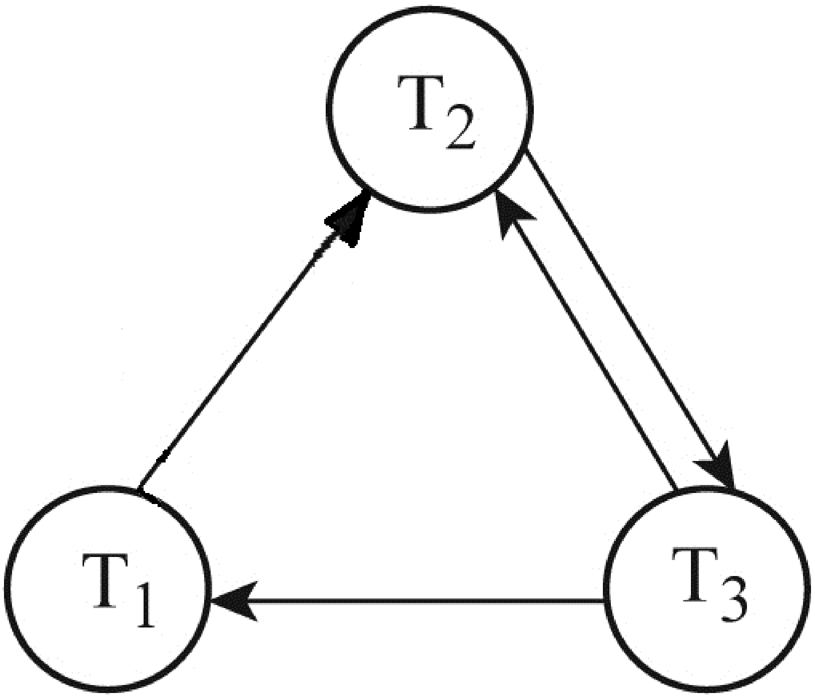
S1.



Serializable. T3->T1->T2

Thus, r3(x); r3(y); w3(y); r1(x); r1(z); r1(y); r2(z); r2(y); w2(z); w2(y)

S2.



There is a circle in between T2 and T3, thus **non-serializable**.

21.24

Strict Schedule:

S3 is **not** strict. R3(x) ->…->c1. X is writen and commited w1(x) before c1, but T3 has read x before the commit of T1 and does not re-read the x varibale again after c1.

S4 is **not** strict. R3(x) happens before w1(x), which means T3 read the intial(not writen) value of x. T3 must read x after c1.

S5 is **not** strict for the same reason in S4.

Cascadeless schedule:

S3 is **not** cascadeless because T3 r3(X) occurs before T1 c1.

S4 is **not** cascadeless for the same reason as S3.

S5 is **not** cascadeless, since the same reason as S3. S5 has other non-cascadeless conflict as well.

Recoverable or non recoverable schedule:

A1, A2, and A3 means T1 abort, T2 abort or T3 abort.

S3.

If A1 -> C3 -> C2, then schedule S3 is recoverable because rolling back of T1 does not affect T2 and T3. If C1->A3->C2. schedule S3 is not recoverable because T2 read the value of Y (r2(Y)) after T3 wrote X (w3(Y)) and T2 committed but T3 rolled back. Thus, T2 used non- existent value of Y. If C1 -> C3 -> A3, then S3 is recoverable because roll back of T2 does not affect T1 and T3.

S4.

If A1 -> C2 -> C3, then schedule S4 is recoverable because roll back of T1 does not affect T2 and T3. If C1 -> A2 -> C3, then schedule S4 is recoverable because the roll back of T2 will restore the value of Y that was read and written to by T3 (w3(Y)). It will not affect T1. If C1 -> C2 -> A3, then schedule S4 is not recoverable because T3 will restore the value of Y which was not read by T2.

S5.

If A1 -> C3 -> C2, then S5 is recoverable because neither T2 nor T3 writes to X, which is written by T1. If C1 -> A3 -> C2, then schedule S5 is not recoverable because T3 will restore the value of Y, which was not read by T2. Thus, T2 committed with a non-existent value of Y. If C1 -> C3 -> A2, then schedule S5 is recoverable because it will restore the value of Y to the value, which was read by T3. Thus, T3 committed with the right value of Y. Strictest condition of schedule S3 is C3 -> C2, but it is not satisfied by S5.

22.1

Two-phase locking:

It is a one of the locking schema which a transaction cannot request a new lock until it unlocks the operations in the transaction. It is involved in two phases: locking phase and unlocking phase.

The previous is the phase that new locks are acquired but not released. The latter is the phase that existing locks are released and no new locks are acquired.

Guarantee of serializablity:

Suppose X, Y, and Z are locked by T, then if another transaction D wants to access(read and write) any of the XYZ, D is put into the wait list until T release the lock. This mechanism limits the amount of concurrency, and hence guarantee the serialization of schedule.

22.2

1.

Conservative 2PL or static 2PL, which requires a tranx to lock all the items it would access before the transx being executed. It is a deadlock-free protocol.

Basic 2PL, which is a technique for 2PL and transx locks data items incrementally. This may cause dead lock.

2.

A transx T does not release any of its locks until it commits or aborts. As a result, no other transx could access an item that is acquired by T. Strict 2PL is not dead-lock free. Rigorous 2PL is the most restrictive variation of strict two-phase locking. It is preferred since it is easy to implement.

23.5

BFIM, the old value of the data item before updating is called the before imgae.

AFIM, the new value of the data item after updating is called the after image.

In-place updating: writes the buffer to the same original disk location, and over writing the old value of any changed data items on disk. A single copy of each database disk block is maintained. This process is called before image.

Shadowing: writes an updated buffer at a different disk location. Multiple versions of data items can be maintained. This process is called after image.

BFIM and AFIM both are kept on disk and it is not strictly necessary to maintain a log for recovery.

23.7

When in-place updating is used, then log is necessary for reovery and in this case, it must available to recovery manager.

For example, if BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before BFIM is overwritten with the AFIM in the database on disk. This total achieved by write-ahead logging protocol.

Write-ahead logging protocol:

For undo: before a data item’s AFIM is flushed to the database disk, its BFIM must be written to the log and the log must be saved on a stable store.

For redo: before a tranx executes its commit operation, all its AFIM must be written to the log and the log must be saved on a stable store.

23.9

Transaction roll back: it means that if a tranx has failed after a disk write, the write operation need to be undone.

Cascading roll back: it is where the failure and rollback of some tranx requires the roll back of other uncommitted transactions because they read updates of the failed tranx. Meanwhile, any values that are derived from the values rolled back will also be undo.

Practical recovery methods use protocols that do not permit cascading roll back because it is complex and time consumming. Pratical recovery methods guarantee cascade less or strict schedules.

Undo/Redo recovery technique is the technique that does not require any roll back in a deferred update.

23.10

Undo/Redo operations requires logging of entry information.

Undo: restore all BFIM on to disk, remove all AFIM.

Redo: restore all AFIM on to disk.

In the undo/redo algorithm, both types of log entries are combined. Cascading roll back is possible when the read item entries in the log are considered to be undo-type entries.

23.11

The main thought of this technique is, to deffer or postpone any actual updates to the database until the transaction completes its execution successfully end reaches its commit point.

Through this technique, the updates are recorded only in the log and in the cache buffers.

After the transaction reaches its commit point and the log is force written to disk and the updates are recorded in the data base.

Differed update technique is also called as NO – UNDO / REDO recovery.

Deferred update protocol. It maintains two main rules.

1. A transaction cannot change any items in the database until it commits.
2. A transaction may not commit until all of the write operations are successfully recorded in the log.

This means that we must check to see that the log is actually written to disk.

Advantages	Disadvantages
Recovery is made easier with redo	Concurrency is limited
Cascading rollback are ignored	

Deferred update technique is called as NO – UNDO / REDO recovery method because. From the second step (A transaction does not reach its commit point until all its update operations are recorded in the log and the log is force – written to disk) of this protocol is a restatement of the write – ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits. There is never a need to UNDO any operations.

Hence this is known as the NO – UNDO / REDO method.

23.13

Immediate update applies the write operations to the database as the tranx is executing. When the tranx issues an updated commend, the database can be updated with out any need to wait for the tranx to reach its commit point and the update operation must still be recorded in the log before it is applied to the database.

There are two logs: Redo log and Undo log.

Two rules are followed:

1. Tranx T may not update the database until undo entires have been written to the Undo log.

2. Tranx T is not allowed to commit until all redo and undo log entries are written.

Advantages:

Immediate updates allow higher concurrency, since tranxs write continuously to the database rather than waiting until commit point.

Disadvantages:

It can lead the cascading roll backs very time consuming or even problematic.

23.14

Outline and differences for an Undo/Redo algorithm and undo/No-redo algorithm:

Undo/Redo	Undo/No-redo
Recovery tech is immediate.	AFIM are flushed to the database disk under WAL before it commits.
Used in single user environment, no concurrency is required.	Undo all tranx during recovery.
Able to undo of a tranx if it is in the active table	Cannot redo
Able to redo of a tranx if it is in the commit table	
Recovery schemas applies undo and redo to recover the databse from failure.	

23.24

In the case of deferred update by removing the un necessary log entries , the write operations of uncommitted transactions are not recorded in the database until the transactions commit. So, the write operations of T2 and T3 would not have been applied to the database and so T4 would have read the previous values of items A and B, thus leading to a recoverable schedule.

By using the procedure RDU_M (deferred update with concurrent execution in a multiuser environment), the following result is obtained.

The list of committed transactions T since the last checkpoint contains only transaction T4. The list of active transactions T' contains transactions T2 and T3.

Only the WRITE operations of the committed transactions are to be redone. Hence, REDO is applied to:

[write_item,T4,B,15]

[write_item,T4,A,20]

The transactions that are active and did not commit i.e., transactions T2 and T3 are canceled and must be resubmitted. Their operations do not have to be undone since they were never applied to the database.