14.6

There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic.

A possible schedule is obtained by doing a topological sort, that is , T1, T2, T3, T4, T5.

14.12

List he ACIO properties. Explain the usefulness of each.

Answer: The ACID properties, and the need of each of them are:

- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are. Clearly lack of atomicity will lead to inconsistency in the database.
- **Isolation:** When multiple transactions execute concurrently, it should be the case that, for every pair of transactions, Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started transactions executing concurrently with it. The user view of a transaction system requires the isolation property, and the property that concurrent schedules take the system from one consistent table to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency preserving transactions are allowed.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

14.15

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).
T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

a.

There are 2 possible executions: T13 T14 and T14 T13

- case 1:

	A	B
initially	0	0
after T13	0	1
after T14	0	1

Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$.

- case 2:

	A	B
initially	0	0
after T14	1	0
after T13	1	0

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$.

b.

Any interleaving of T13 and T14 results in a non-serializable schedule.

T13	T14

T13	T14
read(A)	
	read(B)
	read(A)
read(B)	
if A=0 then B=B+1	
	if B = 0 the A=A+1
write(B)	
	write(A)

c.

There is no parallel execution resulting in a serializable schedule. From part a. We know that a serializable schedule results in $A = 0 \vee B = 0$. Suppose we start with T13 read(A). Then when the schedule ends, no matter when we run the steps of T2, B=1. Now suppose we start executing T14 prior to completion of T13. Then T2 read(B) will give B a value of 0. So when T2 completes, A=1. Thus $B = 1 \wedge A = 1 \rightarrow \neg(A = 0 \vee B = 0)$. Similarly for starting with T14 read(B).

15.2

Add lock and unlock instructions.

- T34:

lock-S(A)

read(A)

lock-X(B)

read(B)

if A=0

then B:=B+1

write(B)

unlock(A)

unlock(B)

- T35

lock-S(B)

read(B)

read(A)

if B=0

then A:=A+1

write(A)

unlock(B)

unlock(A)

Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T31	**T32**
lock-S(A)	
	lock-S(B)
	read(B)
read(A)	
lock-X(B)	
	lock-X(A)

The transactions are now deadlocked.

15.10

- a.

Serializability can be shown by observing that if two transactions have an I mode lock on the same item, the increment operations can be swapped, just like read operations. However, any pair of conflicting operations must be serialized in the order of the lock points of the corresponding transactions, as shown in Exercise 15.1.

- b.

The increment lock mode being compatible with itself allows multiple incrementing transactions to take the lock simultaneously thereby improving the concurrency of the protocol.

In the absence of this mode, an exclusive mode will have to be taken on a data item by each transaction that wants to increment the value of this data item. An exclusive lock being incompatible with itself adds to the lock waiting time and obstructs the overall progress of the concurrent schedule.

In general, increasing the true entries in the compatibility matrix increases the concurrency and improves the throughput.

15.21

It is relatively simple to implement, imposes low rollback overhead because of cascadeless schedules, and usually allows an acceptable level of concurrency.