18.2 Consider the following two transactions:

```
T34: read(A);
     read(B);
     if A = 0 then B := B + 1;
     write(B).

T35: read(B);
     read(A);
     if B = 0 then A := A + 1;
     write(A).
```

Add lock and unlock instructions to transactions *T*31 and *T*32 so that they observe the two-phase locking protocol. Can the execution of these transactionsresult in a deadlock?

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)
        lock-X(A)
        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:

| $T_{31}$ | $T_{32}$ |
|---|---|
| lock-S ($A$) | |
| | lock-S ($B$) |
| | read($B$) |
| read($A$) | |
| lock-X ($B$) | |
| | lock-X ($A$) |

The transactions are now deadlocked.

18.3 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

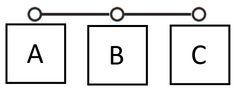
Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

18.4 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

Consider two nodes *A* and *B*, where *A*is a parent of *B*. Let dummy vertex *D* be added between *A* and *B*. Consider a case where transaction *T*2 has a lock on *B*, and *T*1, which has a lock on *A* wishes to lock *B*, and *T*3 wishes to lock *A*. With the original tree, *T*1 cannot release the lock on *A* until it gets the lock on *B*. With the modified tree, *T*1 can get a lock on *D*, and release the lock on *A*, which allows *T*3 to proceed while *T*1 waits for *T*2. Thus, the protocol allows locks on vertices to be released earlier to other transactions, instead of holding them when waiting for a lock on a child. A generalization of idea based on edge locks is described in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

18.5 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

Consider the tree-structured database graph given below.



Schedule possible under tree protocol but not under 2PL:

| $T_1$ | $T_2$ |
|---|---|
| lock $(A)$ | |
| lock $(B)$ | |
| unlock $(A)$ | |
| | lock $(A)$ |
| lock $(C)$ | |
| unlock $(B)$ | |
| | lock $(B)$ |
| | unlock $(A)$ |
| | unlock $(B)$ |
| unlock $(C)$ | |

Schedule possible under 2PL but not under tree protocol:

| $T_1$ | $T_2$ |
|---|---|
| lock $(A)$ | |
| | lock $(B)$ |
| lock $(C)$ | |
| | unlock $(B)$ |
| unlock $(A)$ | |
| unlock $(C)$ | |

18.8 In timestamp ordering, W-timestamp(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(*Q*) successfully. Would this change in wording make any difference? Explain your answer.

It would make no difference. The write protocol is such that the most recent transaction to write an item is also the one with the largest timestamp to have done so.

18.12 Consider the timestamp-ordering protocol, and two transactions, one that writes two data items p and q, and another that reads the same two data items. Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a livelock.)

Consider two transactions *T*1 and *T*2 shown below.

| $T_1$ | $T_2$ |
|---|---|
| write($p$) | |
| | read($p$) |
| | read($q$) |
| write($q$) | |

Let TS(T1) < TS(T2) and let the timestamp test at each operation except write(q) be successful. When transaction T1 does the timestamp test for write(q) it finds that TS(T1) < R-timestamp(q), since TS(T1) < TS(T2) and R-timestamp(q) = TS(T2). Hence the write operation fails and transaction T1 rolls back. The cascading results in transaction T2 also being rolled back as it uses the value for item p that is written by transaction T1. If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

18.19 Consider a variant of the tree protocol called the forest protocol. The database is organized as a forest of rooted trees. Each transaction Ti must follow the following rules:
• The first lock in each tree may be on any data item.
• The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
• Data items may be unlocked at any time.
• A data item may not be relocked by Ti after it has been unlocked by Ti.

The **forest protocol** described is a generalization of the tree protocol where the database is organized as a **forest of rooted trees** instead of a single tree. Let's break down the protocol and explore its implications.

**Rules of the Forest Protocol**

1. **First Lock in Each Tree**:
   - The first lock in a tree can be on any data item within that tree.
   - A transaction is not restricted to starting at the root of a tree.
2. **Subsequent Locks**:
   - Subsequent locks can only be acquired on child nodes of the currently locked node.
   - This enforces a hierarchical traversal within each tree.

3. **Unlocking**:
    - Data items may be unlocked at any time (there is no strict release ordering).
4. **Relocking Restriction**:
    - Once a transaction $T_i$ unlocks a data item, it cannot lock that data item again.

**Implications of the Forest Protocol**

- **Deadlock Prevention**:
    - The hierarchical nature of locks within each tree helps avoid circular wait conditions, reducing the chance of deadlocks.
- **Restrictive Traversal**:
    - Transactions are forced to traverse the data hierarchy of each tree in a top-down manner, similar to the tree protocol.
- **Concurrency Control**:
    - The protocol ensures serializability by preventing conflicting transactions from accessing parent and child nodes simultaneously.

**Comparison with Tree Protocol**

The forest protocol extends the tree protocol by allowing transactions to operate on a **forest** instead of a single tree, enabling more flexibility. However, the hierarchical traversal rules remain in place, maintaining similar guarantees.

**Sample Problem for Analysis**

**Q: Does the forest protocol guarantee serializability? Why or why not?**

**Answer:**

Yes, the forest protocol guarantees **serializability** for the following reasons:

1. **Top-Down Traversal**:
    - Transactions must lock items in a hierarchical order (from parent to child within each tree), which avoids conflicting access patterns.
2. **No Relocking**:
    - The restriction that data items cannot be relocked after being unlocked prevents cycles in the dependency graph of transactions.
3. **Isolated Access**:
    - Since transactions operate on separate trees or follow a consistent order within a tree, there is no risk of violating consistency.

The forest protocol, therefore, maintains the serializability property while allowing more flexibility than the tree protocol.