1. Consider the following transaction schedule where C stands for commit. Operations on the same line occur concurrently by different threads.

```
T1      T2      T3
R(A)    R(B)    R(A)
W(B)    R(A)
                R(B)
                W(B)
                C
        C
C
```

(a) Is this schedule possible with 2PL? If so, illustrate how this schedule could be executed under two-phase locking. Specifically, write out a complete schedule that includes lock acquisitions and releases. Use `S(A)` to indicate obtaining a shared lock on A, `X(A)` to indicate an exclusive lock, and `U(A)` to indicate the release of the lock. If not, explain why not.

Solution:

```
T1      T2      T3
S(A)    S(B)    S(A)
R(A)    R(B)    R(A)
        S(A)
        U(B)
X(B)
W(B)    R(A)
U(A,B)  U(A)
                X(B)
                R(B)
                W(B)
                U(A,B)
                C
        C
C
```

(b) Is this schedule feasible under *strict* two-phase locking? Why or why not?

Solution: T1 must release its lock on B so that T3 can read B. Yet, we also see T3 commits before T1. Thus, T1 released a lock before commit, violating strict 2PL.

(c) Is this schedule *recoverable*? Why or why not?

Solution: No because T3 reads B after T1 writes it yet T3 commits before T1. (If T1 aborted instead of committing, T3 would need to be rolled back, but it would have already committed.)

2. Consider the following transaction schedule where C stands for commit. Operations on the same line occur concurrently by different threads.

```
T1      T2      T3
R(B)    R(B)    R(B)
W(B)    R(A)
C       C
                R(A)
                W(A)
                C
```

(a) Is this schedule possible with 2PL? (Follow same instructions as previous question.)

Solution: No because T3 must release shared lock on B after the first step so that T1 can acquire an exclusive lock on B. But T3 must later acquire an exclusive lock on A – we know it's later because it must come after T2's read of A.

(b) Is this schedule feasible under *strict* two-phase locking? Why or why not?

Solution: No because it's not feasible under 2PL.

(c) Is this schedule *conflict serializable*? If so, to what serial schedule is it equivalent? If not, why not?

Solution: Yes, it is equivalent to T2 T3 T1.

3. Consider the following schedule:

T1:R(X), T3:R(X), T2:W(X), T1:W(X), T2:W(X), T3:W(Y)

(a) Draw (or describe) the precedence graph for this schedule.

Solution:

`T1 -> T2` because T1 reads X before T2 writes it.

`T2 -> T1` because T2 writes X before T2 writes it.

`T3 -> T1` because T3 reads X before T1 writes it.

`T3 -> T2` because T3 reads X before T2 writes it.

(b) Using the precedence graph, explain why this schedule is *not* conflict serializable.

Solution: It's not conflict serializable because there is a cycle.

(c) Show that this schedule is however serializable by giving an equivalent serial order.

Solution: While not conflict serializable, it is nevertheless equivalent to T3, T1, T2. This is because the write of T1 is a lost write.

4. Consider the following locking protocol: All items are numbered, and once an item is locked by a transaction, only higher-numbered items may be locked by that same transaction. Locks may be released at any time. Only exclusive locks are used.

(a) Does this protocol ensure serializability? If yes, explain briefly why. If not, give an example schedule that abides by this protocol but is not serializable.

Solution: No. Let us assume A has a lower number than B. Consider this: T1:X(A)W(A)U(A), T2:X(A)R(A)U(A), T2:X(B)W(B)U(B), T1:X(B)R(B)U(B).

(b) Does this protocol avoid deadlock? If yes, explain briefly why. If not, give an example schedule that abides by this protocol but leads to deadlock.

Solution: Yes! Proof by contradiction. Suppose there is deadlock and the waits for graph looks like this: $T_1 \to T_2 \to \cdots \to T_n \to T_1$. If T_i waits for T_j on some object X then all locks held by T_i must have lower numbers than X because they were acquired before T_i attempted to lock X. The path $T_1 \to T_2 \to \cdots \to T_n$ implies that T_1's locks have lower numbers than T_n's locks, but the path $T_n \to T_1$ implies the opposite, which is a contradiction.