1a

        T         |       U

    ---------------------------------

       Lock(i)    |

    x = read(i);  |

Unlock(i) |

|   lock(i)

                  |   write(i, 55);

lock(j)

                  |   write(j, 66);

| Unlock(i)

| Unlock(j)

Lock(j) | commit

write(j,44) |

Unlock(j) |

commit

x=read(i) and write(i,55) are in conflict and the access order is T, then U

write(j,66) and write(j,44) are in conflict and the access order is U, then T. **NOT SERIALIZED**

1b.

        T         |       U

    ---------------------------------

       Lock(i) |

Lock(j)   |

|

    x = read(i);  |

Unlock(i) |

|

write(j,44) |

Unlock(j) |

| lock(i)

| write(i,55)

|

| lock(j)

| write(j,66)

|

| unlock(i)

| unlock(j)

x=read(i) and write(i,55) are in conflict, access order is T, then U

write(j,44) and write(j,66) are in conflict, access order is T, then U. **SERIALIZED**

2a.

write(i,12) and y=read(i) are conflicting and access order is U, then T

write(k, 66) and write(k,33) are conflicting access order is U, then T. Since the conflicts have same ordering, this interleaving is **SERIALZED**

T | U

---------------------------------

| lock(i), lock(k), lock(l)

|

| write(i, 12);

| unlock(i)

Lock(j) |

x = read(j); |

lock(i) |

y = read(i); |

| write(k, 66);

| unlock(k)

Lock(k) |

write(k, 33); |

| write(l, 22);

| unlock(l)

Unlock(j) |

Unlock(i) |

Unlock(k) |

Schedule is not strict since T does a write(k,33) before U has committed write(k,66).

y=read(i) is dirty read since the write(i,12) has not yet committed.

2b.

Y=read(i) and write(i,12) are in conflict, access order is T, then U

Write(k,66) and write(k,33) are in conflict, access order is U, then T. **NOT SERIALIZED**

T | U

---------------------------------

| write(i, 12);

x = read(j); |

y = read(i); |

| write(k, 66);

write(k, 33); |

| write(l, 22);

2c.

write(i,12) and y=read(i) are in conflict, access order is U, then T.

Write(k,66) and write(k,33) are in conflict, access order is U, then T. **SERIALIZED**

T | U

---------------------------------

Lock(j) | lock(i), lock(k), lock(l)

x = read(j); |

| write(i, 12);

| unlock(i)

|

| write(k, 66);

| unlock(k)

Lock(i) |

y = read(i); |

lock(k) |

write(k, 33); |

unlock(k) |

unlock(i) |

unlock(j) |

| write(l, 22);

| unlock(l)

Schedule is not strict since T does a y=read(i) before U has committed write(i,12). Also, T does a write(k,33) before U has committed write(k,66).

y=read(i) is dirty read since the write(i,12) has not yet committed

2d.

Y=read(i) and write(i,12) are in conflict, access order is T, then U.

Write(k,33) and write(k,66) are in conflict, access order is T, then U. **SERIALIZED**

T | U

---------------------------------

Lock(j) |

Lock(i) |

Lock(k) |

|

x = read(j); |

y = read(i); |

write(k, 33); |

|

unlock(j) |

unlock(i) |

unlock(k) | lock(i)

| write(i, 12);

| lock(k)

| write(k, 66);

| lock(l)

| write(l, 22);

|

| unlock(l)

| unlock(k)

| unlock(i)

Schedule is strict since U performs write(i,12) and write(k,66) after T commits y=read(i) and write(k,33).

No dirty reads, T completes before U starts

2e.

Write(i,12) and y=read(i) are in conflict, access order is U, then T.

Write(k,66) and write(k,33) are in conflict, access order is U, then T. **SERIALIZED**

T | U

---------------------------------

| lock(i), lock(k), lock(l)

| write(i, 12);

Lock(j) |

x = read(j); |

| write(k, 66);

| write(l, 22);

| unlock(k)

| unlock(l)

| unlock(i)

|

Lock(i) |

y = read(i); |

lock(k) |

write(k, 33); |

|

unlock(i) |

unlock(k) |

unlock(j) |

Schedule is strict since T performs y=read(i) and write(k,33) after U has committed write(i,12) and write(k,66).

No dirty reads. Y=read(i) is performed after write(i,12) is committed.

3.

P1 writes 0 into x and then a 1 into x

P2 reads x and gets 1 and on a subsequent read of x, P2 gets 0. This is not sequentially consistent

4.

P2’s writes are causally related since they both write to x

P3’s read and write are causally related since the write follows the read

P3 reads 1 from x, which means that P2 has finished writing

P1 reads 2 from y, which means that p3 has finished writing

However, the second read in P1 gives 0. If this were causally consistent, then the second read would yield 1 since P2 finished writing a 1 to x and this would be seen by all processes.

5.

Monotonic read

Yes. User wants to see all the emails all the time. If user moves from city A to city B, and accesses email in city B, the user would want to see all the emails, including the emails that arrived while user was en route to B.

Monotonic write

Yes. Assume emails are replicated at different servers in different servers so that they may be accessed more quickly. If user creates email in city A and saves a draft, then goes to city B, then the copy at the NY server should be up to date with the changes made in city A.

Read your writes

Yes. If user updates the password he/she would want to access email right away with the new password and not wait for the change to propagate.

Write follows reads

Yes. User would like to see original email along with replies to it, especially if the email chain is long