Assignment 3 - Solution

Problem 1

H₁: r₁[y] r₁[x] r₂[x] w₁[y] c₁ w₂[y] c₂

H₁ is normally-strict two-phase locked:


Note that ru₁[y] isn’t needed, since ru₁[y] was converted into wu₁[y], i.e., T₁ holds only one lock on y.

H₂: r₁[y] r₁[x] r₂[x] w₂[x] w₁[y] c₁ w₂[y] c₂

H₂ is two-phase locked, but not strict two-phase locked. To run w₂[x], T₁ must have released its read lock on x before w₂[x], which means it cannot be strict two-phase locked. Moreover, to be two-phase locked, it must have gotten its write lock on y before it released its read lock on x. Thus, we have the following:


H₃: r₁[y] r₁[x] r₂[x] w₁[y] w₂[y] c₂ c₁

H₃ is two-phase locked, but not strict two-phase locked because T₁ must have released its write lock before w₂[y] executed.


H₄: r₁[y] r₁[x] r₂[x] w₂[x] r₃[y] w₁[y] c₁ w₃[z] c₃ w₂[y] c₂

H₄ is not two-phase locked. To see why, consider the following prefix of the history:

rl₁[y] r₁[y] rl₁[x] r₁[x] rl₂[x] r₂[x]

The next operation is w₂[x]. So as in H₂, T₁ must have released its read lock on x before w₂[x], so again the next few operations must have been w₁[y] ru₁[x] w₁[y] w₂[y] w₂[x], as in the following expanded prefix.

Assignment 3
There's no cycle in the SG, so the history is serializable as T_3 T_1 T_2. Note that there are no transaction handshakes in the input, so there are none to preserve.

Extra credit: Is it possible for a history to be strict two-phase locked but not normally-strict two phase locked? No. To prove it, let H be a strict 2PL history that has been augmented with lock and unlock operations to demonstrate that it’s strict 2PL. We can transform H into a history each of whose lock operations immediately precedes the operation it’s synchronizing, as follows.

- Suppose that for some operation o_i[x] in H, the corresponding lock request ol_i[x] does not immediately precede o_i[x].
- The only constraint that prevents moving ol_i[x] to the right in H so that it immediately precedes o_i[x] is an unlock operation by T_i, since that would break 2PL.
- However, since H is strict 2PL, all of T_i’s unlock operations follow c_i.
- Therefore, it’s possible to move ol_i[x] to the right in H so that it immediately precedes o_i[x].
- This can be done for all offending lock operations in H, thereby transforming it into a demonstration that H is normally-strict 2PL-ed.

Problem 2
Consider the following three transactions, each of which is a sequential program:

T_1: r_1[x] r_1[y]
T_2: r_2[x] r_2[y]
T_3: w_3[y] w_3[x]

Suppose they start executing as follows:

H_1: r_1[x] r_2[x] w_3[y]

So far, T_1 and T_2 each have a read lock on x, and T_3 has a write lock on y.

Next, each transaction tries to set a lock for its second operation: r_1[y], r_2[y], and w_3[x]. However, no matter which order the three lock requests are made, none of those lock requests can be granted, because another transaction already owns a conflicting lock. In terms of the waits-for graph, we have:

- T_1 \rightarrow T_3 because T_1 requests a read lock on y and T_3 owns a write lock on y
- T_2 \rightarrow T_3 for the same reason as above
- T_3 \rightarrow T_1 because T_3 requests a write lock on x and T_1 owns a read lock on x
- T_3 \rightarrow T_2 for the same reason as above.
Thus, there are two deadlock cycles in the graph, $T_1 \rightarrow T_3 \rightarrow T_1$ and $T_2 \rightarrow T_3 \rightarrow T_2$.

Since each transaction is sequential, it can only have one blocked operation. It is therefore tempting to say that there could only be one outgoing edge from the transaction in the waits-for graph. But the italicized implication is wrong, because a transaction may issue a write request, thereby waiting for all of the transactions holding a read lock. Therefore, it is waiting for each of those read transactions and has more than one outgoing edge. In the above example $T_3$ is waiting for both $T_1$ and $T_2$ to unlock $x$. Then $T_1$ and $T_2$ each request a lock on $y$, which causes each of them to deadlock (independently) with $T_3$.

---

**Problem 3**

Let’s hand execute each sequence by issuing a lock request for each operation as it arrives:

a) $H_1$: $r_1[x,y] r_2[x] w_1[x] w_2[z] r_3[z] r_3[y] w_3[y]$

$rl_1[x,y] r_1[x,y] r_2[x] r_2[x]$ \{ $w_1[x]$ is blocked \} $wl_2[z] w_2[z]$ \{ $T_2$ is done so it could have issued commit at this point \} $c_2 wu_2[z] wu_2[z]$ \{ now we can set $w_1[x]$ \} $w_1[x] w_1[x]$ \{ $T_1$ is done so it can commit \} $c_1 r_1[y] wu_1[x]$ \{ now there are no locks held so $T_3$ can execute and commit \}. So adding commits to $H_1$:

$H_1$: $r_1[x,y] r_2[x] w_1[x] w_2[z] c_2 r_3[z] r_3[y] w_3[y] c_3$

b) $H_2$: $r_1[x,y] r_2[x] w_1[x] r_3[z] w_2[z] r_3[y] w_3[y]$

$rl_1[x,y] r_1[x,y] r_2[x] r_2[x]$ \{ $w_1[x]$ is blocked \} $rl_3[z] r_3[z]$ \{ $w_2[z]$ is blocked \} $rl_3[y] r_3[y]$ \{ $w_3[z]$ is blocked \}

There’s a deadlock: $w_1[x]$ is waiting for $rl_2[x]$, $w_2[z]$ is waiting for $rl_3[z]$, and $w_3[y]$ is waiting for $rl_1[y]$. 