Concurrency control and scheduling

CSE 444, summer 2010 — section 5 worksheet

July 22, 2010

Our notation for actions in a schedule:

- st_k: transaction T_k begins
- $r_k(X)$: T_k reads database element X
- $w_k(X)$: T_k writes database element X
- com_k: T_k commits

Other notation will be introduced as needed.

1 Two-phase locking

For each of the following schedules, suppose that we add one lock action ($L_k(X)$) and one unlock action ($U_k(X)$) for each database element that is used.

1. $r_1(A)$, $w_1(B)$
2. $r_2(A)$, $w_2(A)$, $w_2(B)$

For each schedule, please answer the following questions:

1. Say when each lock and unlock action can appear relative to the other actions (both read/write and lock/unlock). Don't worry about ensuring two-phase locking for now, but make sure that every element is locked before use and unlocked after use.

   Solution:
   For schedule 1:

   - $L_1(A)$ can only appear before the first action, $r_1(A)$.
   - $U_1(A)$ can appear before $w_1(B)$, or after both actions.
   - $L_1(B)$ can appear before either action, but not after both.
   - $U_1(B)$ can only appear after both actions.

   For schedule 2:

   - $L_2(A)$ can only appear before the first action, $r_2(A)$.
   - $U_2(A)$ can appear before $w_2(B)$, or after all actions.
   - $L_2(B)$ can appear before any action, but not after all of them.
   - $U_2(B)$ can only appear after all actions.

2. How many possible schedules are allowed by the rules you just gave?

Solution:
We count the number of places each action can occur, and multiply all the counts together. In doing so we ignore differences caused by swapping adjacent lock and unlock actions.

Hence, for schedule 1, $(1)(2)(2)(1) = 4$ schedules are allowed, while for schedule 2, we can have $(1)(3)(2)(1) = 6$ schedules.

3. If we enforce two-phase locking, how do your rules change, and how many schedules are now allowed?

Solution:
For schedule 1, $U_1(A)$ must follow both $L_1(A)$ and $L_1(B)$. Hence, if both $L_1(B)$ and $U_1(A)$ appear between the two actions, $L_1(B)$ must appear first. However, we didn't count swaps of adjacent locks and unlocks in our previous answer, so our answer does not change — we still have 4 schedules.

Schedule 2 is similar: again, $U_2(A)$ must follow both $L_2(A)$ and $L_2(B)$. And again, this only matters when both $L_2(B)$ and $U_2(A)$ appear between the same two actions, in this case $w_2(A)$ and $w_2(B)$. Since this amounts to merely swapping two adjacent locks and unlocks, again the number of possible schedules does not change.

2 Timestamps

Each of the following schedules is presented to a timestamp-based scheduler. Assume that the read and write timestamps of each element start at 0 ($\text{RT}(X) = \text{WT}(X) = 0$), and the commit bits for each element are set ($\text{C}(X) = 1$). Please tell what happens as each schedule executes.

Correction: In the handout the commit bits were listed as initially set to 0 rather than 1; this has been corrected above.

1. st_1, st_2, st_3, $r_1(A)$, $r_2(B)$, $w_1(C)$, $r_3(B)$, $r_3(C)$, $w_2(B)$, $w_3(A)$

   Solution:
   The scheduler proceeds as follows:

   - st_1, st_2, st_3: We assign timestamps in the order the transactions start; here I let $\text{TS}(T_1) := 1$, $\text{TS}(T_2) := 2$, and $\text{TS}(T_3) := 3$.
   - $r_1(A)$: ***OK*** because element A hasn't yet been written ($\text{WT}(A) = 0$). Sets $\text{RT}(A) := (\text{TS}(T_1) = 1)$.
   - $r_2(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 2$.
   - $w_1(C)$: ***OK*** because C hasn't yet been read or written ($\text{RT}(C) = \text{WT}(C) = 0$). Sets $\text{WT}(C) := 1$ and $\text{C}(C) := 0$.
   - $r_3(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 3$.
   - $r_3(C)$: We have $(\text{WT}(C) = 1) \leq 3$, but $\text{C}(C) = 0$ because T_1 has not yet committed or aborted. ***Delay*** T_3 until $\text{C}(C) = 1$ or T_1 aborts, then recheck timestamps and retry this action.
   - $w_2(B)$: We have $(\text{RT}(B) = 3) > 2$, so allowing this write would cause a*write too late* anomaly, because T_3 should have read the value about to be written by T_2, which comes earlier in the serialization order. ***Rollback*** T_2.
   - $w_3(A)$: ***Wait*** until and unless T_3 is unblocked and $r_3(C)$ above succeeds. If T_3 is later aborted, then do not execute this action.

2. st_1, st_3, st_2, $r_1(A)$, $r_2(B)$, $w_1(C)$, $r_3(B)$, $r_3(C)$, $w_2(B)$, $w_3(A)$

   Solution:
   This schedule is the same as the one above, except now T_3 precedes T_2. The scheduler proceeds as follows:

   - st_1, st_3, st_2: Let $\text{TS}(T_1) := 100$, $\text{TS}(T_2) := 300$, and $\text{TS}(T_3) := 200$.
   - $r_1(A)$: ***OK*** because $\text{WT}(A) = 0$. Sets $\text{RT}(A) := 100$.
   - $r_2(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 300$.
   - $w_1(C)$: ***OK*** because $\text{RT}(C) = \text{WT}(C) = 0$. Sets $\text{WT}(C) := 100$ and $\text{C}(C) := 0$.
   - $r_3(B)$: ***OK*** because $\text{WT}(B) = 0$. Does not change $\text{RT}(B)$ because $(\text{RT}(B) = 300) > 200$.
   - $r_3(C)$: We have $(\text{WT}(C) = 100) \leq 200$, but $\text{C}(C) = 0$ because T_1 has not yet committed or aborted. ***Delay*** T_3 until $\text{C}(C) = 1$ or T_1 aborts, then recheck timestamps and retry this action.
   - $w_2(B)$: ***OK*** because $(\text{RT}(B) = 300) \leq 300$ and $(\text{WT}(B) = 0) \leq 300$. Sets $\text{WT}(B) := 300$ and $\text{C}(B) := 0$.
   - $w_3(A)$: ***Wait*** until and unless T_3 is unblocked and $r_3(C)$ above succeeds. If T_3 is later aborted, then do not execute this action.

3. st_1, st_2, st_3, $r_1(A)$, $r_2(B)$, $r_2(C)$, $r_3(B)$, com_2, $w_3(B)$, $w_3(C)$

   Solution:
   The scheduler does the following:

   - st_1, st_2, st_3: Let $\text{TS}(T_1) := 1$, $\text{TS}(T_2) := 2$, and $\text{TS}(T_3) := 3$.
   - $r_1(A)$: ***OK*** because $\text{WT}(A) = 0$. Sets $\text{RT}(A) := 1$.
   - $r_2(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 2$.
   - $r_2(C)$: ***OK*** because $\text{WT}(C) = 0$. Sets $\text{RT}(C) := 2$.
   - $r_3(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 3$.
   - com_2: ***Does nothing*** because T_2 didn't make any changes.
   - $w_3(B)$: ***OK*** because $(\text{RT}(B) = 3) \leq 3$ and $(\text{WT}(B) = 0) \leq 3$. Sets $\text{WT}(B) := 3$ and $\text{C}(B) := 0$.
   - $w_3(C)$: ***OK*** because $(\text{RT}(C) = 2) \leq 3$ and $(\text{WT}(C) = 0) \leq 3$. Sets $\text{WT}(C) := 3$ and $\text{C}(C) := 0$.

4. st_1, st_2, $r_1(A)$, $r_2(B)$, $w_2(A)$, com_2, $w_1(B)$

   Solution:
   The scheduler does the following:

   - st_1, st_2: Let $\text{TS}(T_1) := 1$ and $\text{TS}(T_2) := 2$.
   - $r_1(A)$: ***OK*** because $\text{WT}(A) = 0$. Sets $\text{RT}(A) := 1$.
   - $r_2(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 2$.
   - $w_2(A)$: ***OK*** because $(\text{RT}(A) = 1) \leq 2$ and $(\text{WT}(A) = 0) \leq 2$. Sets $\text{WT}(A) := 2$ and $\text{C}(A) := 0$.
   - com_2: Set $C(A) := 1$ because T_2 was the last transaction to write A, as determined by $\text{WT}(A)$.
   - $w_1(B)$: ***Rollback*** T_1, because $(\text{RT}(B) = 2) > 1$.

5. st_1, st_3, st_2, $r_1(A)$, $r_2(B)$, $r_3(B)$, $w_3(A)$, $w_2(B)$, com_3, $w_1(A)$

Solution:
The scheduler does the following:

- st_1, st_3, st_2: Let $\text{TS}(T_1) := 100$, $\text{TS}(T_2) := 300$, and $\text{TS}(T_3) := 200$.
- $r_1(A)$: ***OK*** because $\text{WT}(A) = 0$. Sets $\text{RT}(A) := 100$.
- $r_2(B)$: ***OK*** because $\text{WT}(B) = 0$. Sets $\text{RT}(B) := 300$.
- $r_3(B)$: ***OK*** because $\text{WT}(B) = 0$. Does not change $\text{RT}(B)$ because $(\text{RT}(B) = 300) > 200$.
- $w_3(A)$: ***OK*** because $(\text{RT}(A) = 100) \leq 200$ and $(\text{WT}(A) = 0) \leq 200$. Sets $\text{WT}(A) := 200$ and $\text{C}(A) := 0$.
- $w_2(B)$: ***OK*** because $(\text{RT}(B) = 300) \leq 300$ and $(\text{WT}(B) = 0) \leq 300$. Sets $\text{WT}(B) := 300$ and $\text{C}(B) := 0$.
- com_3: Set $C(A) := 1$ because T_3 was the last transaction to write A, as determined by $\text{WT}(A)$.
- $w_1(A)$: ***Ignore this write,*** because while $(\text{RT}(A) = 100) \leq 100$, we have $(\text{WT}(A) = 200) > 100$ and $\text{C}(A) = 1$, so we know that a later, committed transaction has already written to A (the Thomas write rule).

6. st_1, $r_1(A)$, $w_1(A)$, st_2, $r_2(C)$, $w_2(B)$, $r_2(A)$, $w_1(B)$

Solution:
The scheduler does the following:

- st_1: Let $\text{TS}(T_1) := 1$.
- $r_1(A)$: ***OK*** because $\text{WT}(A) = 0$. Sets $\text{RT}(A) := 1$.
- $w_1(A)$: ***OK*** because $(\text{RT}(A) = 1) \leq 1$ and $(\text{WT}(A) = 0) \leq 1$. Sets $\text{WT}(A) := 1$ and $\text{C}(A) := 0$.
- st_2: Let $\text{TS}(T_2) := 2$.
- $r_2(C)$: ***OK*** because $\text{WT}(C) = 0$. Sets $\text{RT}(C) := 2$.
- $w_2(B)$: ***OK*** because $\text{RT}(B) = \text{WT}(B) = 0$. Sets $\text{WT}(B) := 2$ and $\text{C}(A) := 0$.
- $r_2(A)$: We have $(\text{WT}(A) = 1) \leq 2$, but $\text{C}(A) = 0$ because T_1 has not yet committed or aborted. ***Delay*** T_2 until $\text{C}(A) = 1$ or T_1 aborts, then recheck timestamps and retry this action.
- $w_1(B)$: We have $(\text{RT}(B) = 0) \leq 1$, but $(\text{WT}(A) = 2) > 1$. We cannot ignore this write, however, because $\text{C}(A) = 0$ and T_2 was the last writer. We should ***delay*** T_1 until $\text{C}(B) = 1$ or T_2 aborts, then recheck timestamps and retry this action.

Notice that T_2 is blocked waiting for T_1 to commit or abort, but T_1 is also blocked, waiting for T_2 to commit or abort. This circular wait represents a deadlock between the two transactions, which must be broken externally.

3 Multi-version timestamps

Tell what happens during the following schedules if we use a *multi-version* timestamp scheduler. What happens if the scheduler does not maintain multiple versions?

Correction: Commit actions have been added after the last action of each transaction; solutions changed accordingly.

1. st_1, st_2, st_3, st_4, $w_1(A)$, com_1, $w_2(A)$, $w_3(A)$, com_3, $r_2(A)$, com_2, $r_4(A)$, com_4

   Solution:
   Each attempt to write to element A creates a new copy of A, and the write occurs on the copy, rather than on the original copy of A (call it A_0), which is never written. Hence, we have copies A_1, A_2, and A_3, representing the new data written by transactions T_1, T_2, and T_3 respectively.

   The read attempts then read from the copy of A with the highest timestamp no greater than the timestamp of the read action's transaction. Hence, $r_2(A)$ will read A_2, the most recent copy T_2 can see, while $r_4(A)$ reads A_3, which was the most recent copy prior to T_4 in the serialization order.

   If we did not use a multi-version scheduler, then the first read $r_2(A)$ would cause T_2 to be rolled back, because a later transaction (T_3) has already written to A. The second read $r_4(A)$ would succeed, however, and would read the last value written by T_3.

2. st_1, st_2, st_3, st_4, $w_1(A)$, com_1, $w_3(A)$, com_3, $r_4(A)$, com_4, $r_2(A)$, com_2

   Solution:
   The two write actions on element A create new versions A_1 and A_3, corresponding to the writes $w_1(A)$ by T_1, and $w_3(A)$ by T_3.

   Then, $r_4(A)$ reads version A_3, the version with the highest timestamp not greater than T_4's, and similarly $r_2(A)$ reads version A_1.

   Without a multi-version scheduler, the first read $r_4(A)$ would succeed because no transaction with higher timestamp than T_4 wrote A. However, the second read $r_2(A)$ would fail and force T_2 to rollback because the later T_3 had already written A.

3. st_1, st_2, st_3, st_4, $w_1(A)$, com_1, $w_4(A)$, com_4, $r_3(A)$, com_3, $w_2(A)$, com_2

   Solution:

   The first two actions on element A create new versions A_1 and A_4, corresponding to the writes $w_1(A)$ by T_1, and $w_4(A)$ by T_4. Then, $r_3(A)$ reads version A_1, the version with the highest timestamp not greater than T_3's.

   However, the last action $w_2(A)$ fails, causing T_2 to roll back. This is because if allowed, $w_2(A)$ would create a version A_2, whose immediately previous version would be A_1. But we notice from A_1's read timestamp that A_1 has already been read by T_3. Hence, if the write $w_2(A)$ were allowed, then T_3 should have read the new version A_2 instead of A_1 which it actually read. This write is thus a write too late, and so must not be allowed.

   Without a multi-version scheduler, the read $r_3(A)$ would fail and roll back T_3, because A has already been written by the later transaction T_4. The last write $w_2(A)$ would not fail, because the read $r_3(A)$ did not happen. But this write would still be ignored, because the later T_4 has already committed a new value for A.

4 Validation

For the following schedules:

- $R_k(X)$ means "transaction T_k starts, and its read set is the list of database elements X,"
- V_k means "T_k tries to validate," and
- $W_k(x)$ means "T_k finished, and its write set was X."

Clarification: Remember that each transaction must inform the scheduler of *both* its read and write sets when it begins, or when it validates (at the latest). While the notation we use implies otherwise, and hence is slightly confusing, we use it to be consistent with your textbook's notation.

Tell what happens when each schedule is processed by a validation-based scheduler.

1. $R_1(A, B)$, $R_2(B, C)$, $R_3(C)$, V_1, V_2, V_3, $W_1(A)$, $W_2(B)$, $W_3(C)$

   Solution:
   T_1 is the first transaction to try to validate; as there are no other transactions to check against, T_1 obviously validates successfully.

   Next, T_2 tries to validate. The only other validated transaction is T_1, and T_1 did not finish before T_2 started, so we need to check the read set of T_2 against the write set of T_1, to make sure that T_2 hasn't read any element for which T_1 has written an as-yet-uncommitted new value. And indeed, we find that $\text{RS}(T_2) \cap \text{WS}(T_1) = \emptyset$, so this check passes.

   Not only did T_1 not finish before T_2 started, it still isn't finished now, as we are validating T_2. Hence, we also need to check the *write* set of T_2 against the write set of T_1 to make sure that T_1, which is earlier in the serialization order, doesn't overwrite any of T_2's changes to the database. This check also passes, as we find that $\text{WS}(T_2) \cap \text{WS}(T_1) = \emptyset$. With both checks of T_2 passing, we can thus validate T_2 successfully.

   Finally, T_3 tries to validate. The other validated transactions are T_1 and T_2, and both transactions were unfinished when T_3 started, so we check T_3's read set against both T_1 and T_2's write sets. No elements are in both T_3's read set and either of the other transactions' write sets, so those checks pass. Because both T_1 and T_2 are still active now, as T_3 is being validated, we also check T_3's write set against the others' write sets, and again with no intersection, the checks pass. Hence T_3 validates successfully.

2. $R_1(A,B)$, $R_2(B,C)$, $R_3(C)$, V_1, V_2, V_3, $W_1(C)$, $W_2(B)$, $W_3(A)$

Solution:
This schedule has the same transactions and read sets as the previous one, but different write sets. As with the previous schedule, T_1 trivially validates because it's first to try to validate.

Next, T_2 tries to validate. The only other validated transaction is T_1, and T_1 did not finish before T_2 started, so we need to check $\text{RS}(T_2)$ against $\text{WS}(T_1)$. We find that $\text{RS}(T_2) \cap \text{WS}(T_1) = \{C\}$ is nonempty, so the check fails, and T_2 cannot be validated. Having failed validation, T_2 is rolled back.

Finally, T_3 tries to validate. The only other validated transaction is still T_1, which was unfinished when T_3 started, so we check T_3's read set against T_1's write set. Once again, we find that the sets' intersection is nonempty ($\text{RS}(T_3) \cap \text{WS}(T_1) = \{C\}$), so T_3 is not validated and is rolled back.

3. $R_1(A,B)$, $R_2(B,C)$, $R_3(C)$, V_1, V_2, V_3, $W_1(A)$, $W_2(C)$, $W_3(B)$

Solution:
This schedule differs from the last two only in write sets. Once again, T_1 validates because it's first to attempt it, so there are no other validated transactions to check against.

Next, T_2 tries to validate. The only other validated transaction is T_1, and T_1 did not finish before T_2 started, so we check that $\text{RS}(T_2) \cap \text{WS}(T_1) = \emptyset$. In addition, T_1 is still not finished now, so we check that $\text{WS}(T_2) \cap \text{WS}(T_1) = \emptyset$. Both checks pass, so T_2 validates.

Finally, T_3 tries to validate. The other validated transactions are T_1 and T_2. Both transactions were unfinished when T_3 started, so we check that $\text{RS}(T_3) \cap \text{WS}(T_1) = \emptyset$ and that $\text{RS}(T_3) \cap \text{WS}(T_2) = \emptyset$. The first check passes, but the second does not ($\text{RS}(T_3) \cap \text{WS}(T_2) = \{C\}$), so T_3 does not validate and gets rolled back.

4. $R_1(A, B)$, $R_2(B, C)$, V_1, $R_3(C, D)$, V_3, $W_1(C)$, V_2, $W_2(A)$, $W_3(D)$

Solution:

As always, the first transaction to try to validate (T_1 in this case) succeeds, because there are no previously validated transactions to check against.

Next, T_3 tries to validate. T_1 is the only other validated transaction, so we need to check T_3 against T_1. Because T_1 did not finish before T_3 started, we check that $\text{RS}(T_3) \cap \text{WS}(T_1) = \emptyset$, and because T_1 is still unfinished now, we also check that $\text{WS}(T_3) \cap \text{WS}(T_1) = \emptyset$. The first condition does not hold ($\text{RS}(T_3) \cap \text{WS}(T_1) = \{C\}$), so T_3 does not validate and must be rolled back.

Finally, T_2 tries to validate. The only other validated transaction is T_1, and because it did not finish before T_2 started, we check that $\text{RS}(T_2) \cap \text{WS}(T_1) = \emptyset$. (We don't need to check that $\text{WS}(T_2) \cap \text{WS}(T_1) = \emptyset$, because T_1 finished before now, the validation time for T_2.) However, element C is in both sets, so the check fails and T_2 fails to validate.