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 \( X \)
- \( \text{com}_k \): \( T_k \) commits

Other notation will be introduced as needed.

1 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 (\( RT(X) = WT(X) = 0 \)), and the commit bits for each element are set \( C(X) = 1 \). Please tell what happens as each schedule executes.

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 \( TS(T_1) := 1 \), \( TS(T_2) := 2 \), and \( TS(T_3) := 3 \).
   - \( r_1(A) \): \text{OK} because element \( A \) hasn’t yet been written \( (WT(A) = 0) \). Sets \( RT(A) := (TS(T_1) = 1) \).
   - \( r_2(B) \): \text{OK} because \( WT(B) = 0 \). Sets \( RT(B) := 2 \).
   - \( w_1(C) \): \text{OK} because \( C \) hasn’t yet been read or written \( (RT(C) = WT(C) = 0) \). Sets \( WT(C) := 1 \) and \( C(C) := 0 \).
   - \( r_3(B) \): \text{OK} because \( WT(B) = 0 \). Sets \( RT(B) := 3 \).
   - \( r_3(C) \): We have \( WT(C) = 1 \leq 3 \), but \( C(C) = 0 \) because \( T_1 \) has not yet committed or aborted. \textit{Delay} \( T_3 \) until \( C(C) = 1 \) or \( T_1 \) aborts, then recheck timestamps and retry this action.
   - \( w_2(B) \): We have \( (RT(B) = 3) > 2 \), so allowing this write would cause \textbf{aurite 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. \textbf{Rollback} \( T_2 \).
   - \( w_3(A) \): \textit{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. **st₃, st₂, r₁(A), r₂(B), w₁(C), r₃(C), w₂(B), w₃(A)**

**Solution:**
This schedule is the same as the one above, except now T₃ precedes T₂. The scheduler proceeds as follows:

- **st₁, st₃, st₂:** Let TS(T₁) := 100, TS(T₂) := 300, and TS(T₃) := 200.
- **r₁(A): OK** because WT(A) = 0. Sets RT(A) := 100.
- **r₂(B): OK** because WT(B) = 0. Sets RT(B) := 300.
- **w₁(C): OK** because RT(C) = WT(C) = 0. Sets WT(C) := 100 and C(C) := 0.
- **r₃(B): OK** because WT(B) = 0. Does not change RT(B) because (RT(B) = 300) > 200.
- **r₃(C):** We have (WT(C) = 100) ≤ 200, but C(C) = 0 because T₁ has not yet committed or aborted. **Delay** T₃ until C(C) = 1 or T₁ aborts, then recheck timestamps and retry this action.
- **w₂(B): OK** because (RT(B) = 300) ≤ 300 and (WT(B) = 0) ≤ 300. Sets WT(B) := 300 and C(B) := 0.
- **w₃(A): Wait** until and unless T₃ is unblocked and r₃(C) above succeeds. If T₃ is later aborted, then do not execute this action.

3. **st₁, st₂, st₃, r₁(A), r₂(B), r₂(C), r₃(B), com₂, w₃(B), w₃(C)**

**Solution:**
The scheduler does the following:

- **st₁, st₂, st₃:** Let TS(T₁) := 1, TS(T₂) := 2, and TS(T₃) := 3.
- **r₁(A): OK** because WT(A) = 0. Sets RT(A) := 1.
- **r₂(B): OK** because WT(B) = 0. Sets RT(B) := 2.
- **r₂(C): OK** because WT(C) = 0. Sets RT(C) := 2.
- **r₃(B): OK** because WT(B) = 0. Sets RT(B) := 3.
- **com₂:** **Does nothing** because T₂ didn’t make any changes.
- **w₃(B): OK** because (RT(B) = 3) ≤ 3 and (WT(B) = 0) ≤ 3. Sets WT(B) := 3 and C(B) := 0.
- **w₃(C): OK** because (RT(C) = 2) ≤ 3 and (WT(C) = 0) ≤ 3. Sets WT(C) := 3 and C(C) := 0.
4. $s_{t_1}, s_{t_2}, r_1(A), r_2(B), w_2(A), \text{com}_2, w_1(B)$

Solution:
The scheduler does the following:

- $s_{t_1}, s_{t_2}$: Let $TS(T_1) := 1$ and $TS(T_2) := 2$.
- $r_1(A)$: OK because $WT(A) = 0$. Sets $RT(A) := 1$.
- $r_2(B)$: OK because $WT(B) = 0$. Sets $RT(B) := 2$.
- $w_2(A)$: OK because $(RT(A) = 1) \leq 2$ and $(WT(A) = 0) \leq 2$. Sets $WT(A) := 2$ and $C(A) := 0$.
- $\text{com}_2$: Set $C(A) := 1$ because $T_2$ was the last transaction to write $A$, as determined by $WT(A)$.
- $w_1(B)$: Rollback $T_1$, because $(RT(B) = 2) > 1$.

5. $s_{t_1}, s_{t_2}, s_{t_3}, r_1(A), r_2(B), r_3(B), w_3(A), w_2(B), \text{com}_3, w_1(A)$

Solution:
The scheduler does the following:

- $s_{t_1}, s_{t_3}, s_{t_2}$: Let $TS(T_1) := 100$, $TS(T_2) := 300$, and $TS(T_3) := 200$.
- $r_1(A)$: OK because $WT(A) = 0$. Sets $RT(A) := 100$.
- $r_2(B)$: OK because $WT(B) = 0$. Sets $RT(B) := 300$.
- $r_3(B)$: OK because $WT(B) = 0$. Does not change $RT(B)$ because $(RT(B) = 300) > 200$.
- $w_3(A)$: OK because $(RT(A) = 100) \leq 200$ and $(WT(A) = 0) \leq 200$. Sets $WT(A) := 200$ and $C(A) := 0$.
- $w_2(B)$: OK because $(RT(B) = 300) \leq 300$ and $(WT(B) = 0) \leq 300$. Sets $WT(B) := 300$ and $C(B) := 0$.
- $\text{com}_3$: Set $C(A) := 1$ because $T_3$ was the last transaction to write $A$, as determined by $WT(A)$.
- $w_1(A)$: Ignore this write, because while $(RT(A) = 100) \leq 100$, we have $(WT(A) = 200) > 100$ and $C(A) = 1$, 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 TS(T\(_1\)) := 1.
- **r\(_1\)(A):** OK because WT(A) = 0. Sets RT(A) := 1.
- **w\(_1\)(A):** OK because (RT(A) = 1) \(\leq\) 1 and (WT(A) = 0) \(\leq\) 1. Sets WT(A) := 1 and C(A) := 0.
- **st\(_2\):** Let TS(T\(_2\)) := 2.
- **r\(_2\)(C):** OK because WT(C) = 0. Sets RT(C) := 2.
- **w\(_2\)(B):** OK because RT(B) = WT(B) = 0. Sets WT(B) := 2 and C(A) := 0.
- **r\(_2\)(A):** We have (WT(A) = 1) \(\leq\) 2, but C(A) = 0 because T\(_1\) has not yet committed or aborted. Delay T\(_2\) until C(A) = 1 or T\(_1\) aborts, then recheck timestamps and retry this action.
- **w\(_1\)(B):** We have (RT(B) = 0) \(\leq\) 1, but (WT(A) = 2) > 1. We cannot ignore this write, however, because C(A) = 0 and T\(_2\) was the last writer. We should delay T\(_1\) until 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.
2 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?

1. st\textsubscript{1}, st\textsubscript{2}, st\textsubscript{3}, st\textsubscript{4}, w\textsubscript{1}(A), com\textsubscript{1}, w\textsubscript{2}(A), w\textsubscript{3}(A), com\textsubscript{3}, r\textsubscript{2}(A), com\textsubscript{2}, r\textsubscript{4}(A), com\textsubscript{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 \(r_4(A)\) would succeed because no transaction with higher timestamp than \(T_4\) wrote \(A\). The second read \(r_4(A)\) would succeed, however, and would read the last value written by \(T_3\).

2. st\textsubscript{1}, st\textsubscript{2}, st\textsubscript{3}, st\textsubscript{4}, w\textsubscript{1}(A), com\textsubscript{1}, w\textsubscript{3}(A), com\textsubscript{3}, r\textsubscript{4}(A), com\textsubscript{4}, r\textsubscript{2}(A), com\textsubscript{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. st₁, st₂, st₃, st₄, w₁(A), com₁, w₄(A), com₄, r₃(A), com₃, w₂(A), com₂

**Solution:**
The first two actions on element $A$ create new versions $A₁$ and $A₄$, corresponding to the writes $w₁(A)$ by $T₁$, and $w₄(A)$ by $T₄$. Then, $r₃(A)$ reads version $A₁$, the version with the highest timestamp not greater than $T₃$’s.

However, the last action $w₂(A)$ fails, causing $T₂$ to roll back. This is because if allowed, $w₂(A)$ would create a version $A₂$, whose immediately previous version would be $A₁$. But we notice from $A₁$’s read timestamp that $A₁$ has already been read by $T₃$. Hence, if the write $w₂(A)$ were allowed, then $T₃$ should have read the new version $A₂$ instead of $A₁$ 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 $r₃(A)$ would fail and roll back $T₃$, because $A$ has already been written by the later transaction $T₄$. The last write $w₂(A)$ would not fail, because the read $r₃(A)$ did not happen. But this write would still be ignored, because the later $T₄$ has already committed a new value for $A$. 
3 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 $X$.”

Note: 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 \( RS(T_2) \) against \( WS(T_1) \). We find that \( RS(T_2) \cap WS(T_1) = \{C\} \) 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 (\( RS(T_3) \cap WS(T_1) = \{C\} \)), 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 \( RS(T_2) \cap WS(T_1) = \emptyset \). In addition, \( T_1 \) is still not finished now, so we check that \( WS(T_2) \cap WS(T_1) = \emptyset \). 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 \( RS(T_3) \cap WS(T_1) = \emptyset \) and that \( RS(T_3) \cap WS(T_2) = \emptyset \). The first check passes, but the second does not (\( RS(T_3) \cap WS(T_2) = \{C\} \)), 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 $RS(T_3) \cap WS(T_1) = \emptyset$, and because $T_1$ is still unfinished now, we also check that $WS(T_3) \cap WS(T_1) = \emptyset$. The first condition does not hold ($RS(T_3) \cap WS(T_1) = \{C\}$), 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 $RS(T_2) \cap WS(T_1) = \emptyset$. (We don’t need to check that $WS(T_2) \cap WS(T_1) = \emptyset$, 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.