Thomas Write Rule in DBMS

Timestamp Ordering Protocol states that if $R_i(X)$ and $W_j(X)$ are conflicting operations then R_i (X) is processed before $W_j(X)$ if and only if $TS(T_i) < TS(T_j)$. Whenever a schedule does not follow serializablity order according to the Timestamp, user generally rejects it and rollback the Transaction. Some operations on the other hand are harmless and can be allowed.

Thomas Write Rule allows such operations and is a modification on the Basic Timestamp Ordering protocol. In Thomas Write Rule user ignore outdated writes. Moreover, of all the Concurrency Protocols have been discussed, Concurrency is imposed on Schedules which are Conflict Serializable, in Thomas Write Rule, the most important improvement is user can achieve Concurrency with View Serializable schedules.

Thomas Write Rule –

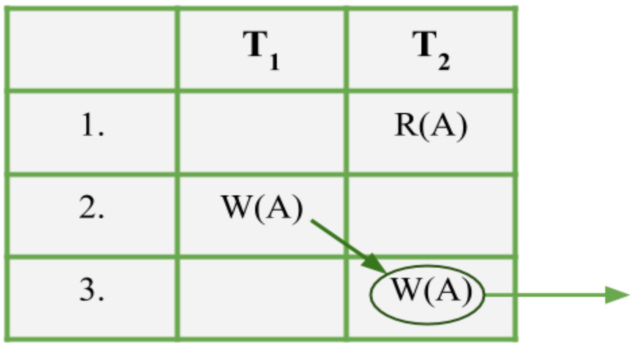
Thomas Write Rule does not enforce *Conflict Serializablity* but rejects fewer Write Operations by modifying the check Operations for W_item(X)

1. If **R_TS(X) > TS(T)**, then abort and rollback T and reject the operation.
2. If **W_TS(X) > TS(T)**, then don't execute the Write Operation and continue processing. This is a case of *Outdated or Obsolete Writes*. Remember, outdated writes are ignored in Thomas Write Rule but a Transaction following Basic protocol will abort such a Transaction.
3. If neither the condition in 1 or 2 occurs, then and only then execute the W_item(X) operation of T and set W_TS(X) to TS(T)

Outdated Write Example –

The main update in Thomas Write Rule is ignoring the Obsolete Write Operations. This is done because some transaction with timestamp greater than TS(T) (i.e., a transaction after T in TS ordering) has already written the value of X. Hence, logically user can ignore the Write(X) operation of T which becomes obsolete. Let us see this through an example:

Suppose user has a schedule in which two transactions T_1 and T_2. Now, $\mathbf{TS(T_2) < TS(T_1)}$. This means T_1 arrived after T_2 and hence has a larger TS value than T_2. This implies that serializablity of schedule allowed is $\mathbf{T_2 \rightarrow T_1}$. Consider the partial schedule given below:



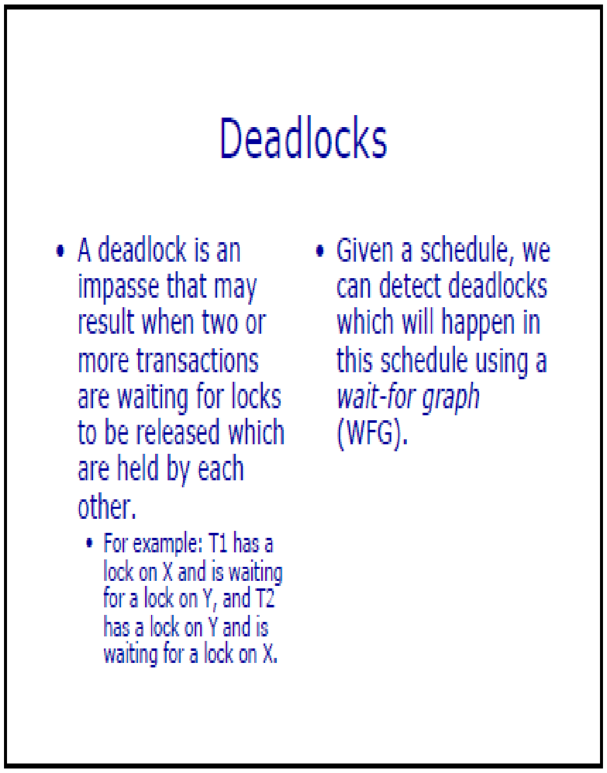
	T_1	T_2
1.		R(A)
2.	W(A)	
3.		W(A)

$T_1 \rightarrow T_2$ which is not allowed. Ignore this Outdated Write operation according to Thomas Write Rule.

Allowed TS ordering $T_2 \rightarrow T_1$

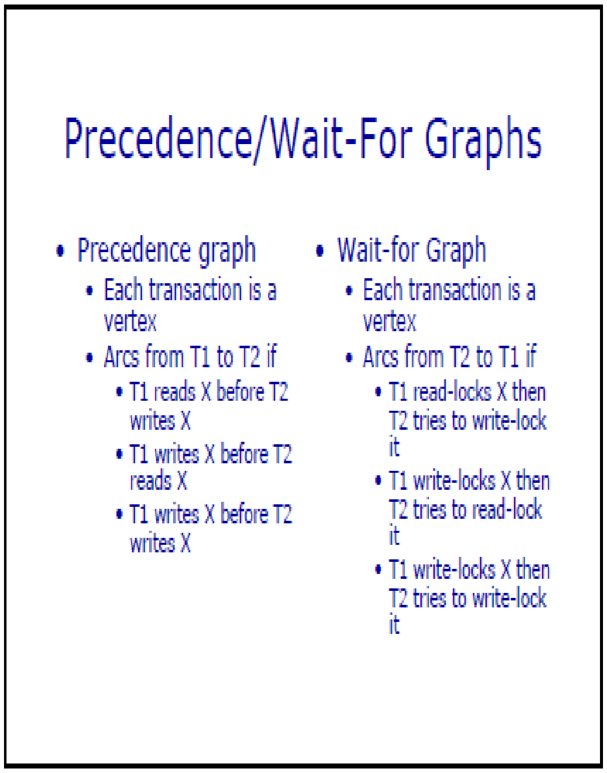
Obsolete Writes are hence ignored in this rule which is in accordance to the 2nd protocol. It seems to be more logical as user skip an unnecessary procedure of restarting the entire transaction. This protocol is just a modification to Basic TO protocol.

- **<u>Precedence / Wait-for Graph</u>**



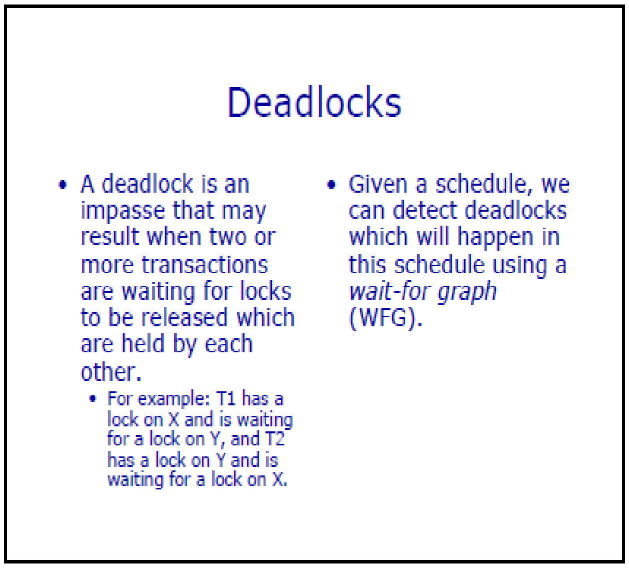
Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example: T1 has a lock on X and is waiting for a lock on Y, and T2 has a lock on Y and is waiting for a lock on X.
- Given a schedule, we can detect deadlocks which will happen in this schedule using a *wait-for graph* (WFG).



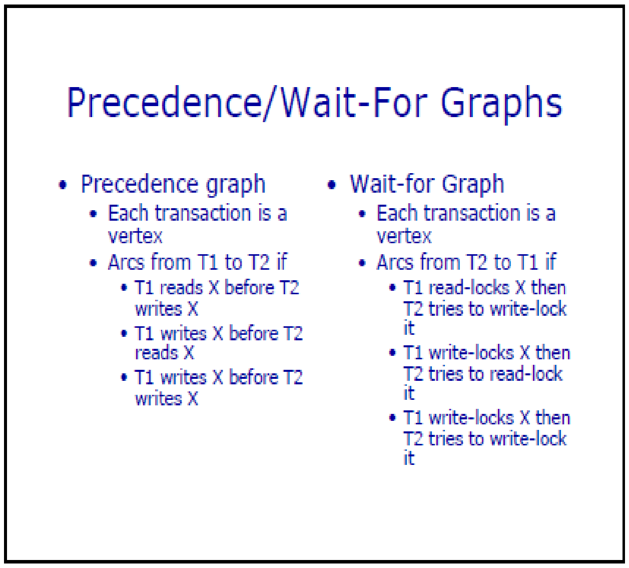
Precedence/Wait-For Graphs

- Precedence graph
  - Each transaction is a vertex
  - Arcs from T1 to T2 if
    - T1 reads X before T2 writes X
    - T1 writes X before T2 reads X
    - T1 writes X before T2 writes X
- Wait-for Graph
  - Each transaction is a vertex
  - Arcs from T2 to T1 if
    - T1 read-locks X then T2 tries to write-lock it
    - T1 write-locks X then T2 tries to read-lock it
    - T1 write-locks X then T2 tries to write-lock it



Deadlocks

- A deadlock is an impasse that may result when two or more transactions are waiting for locks to be released which are held by each other.
  - For example: T1 has a lock on X and is waiting for a lock on Y, and T2 has a lock on Y and is waiting for a lock on X.
- Given a schedule, we can detect deadlocks which will happen in this schedule using a *wait-for graph* (WFG).

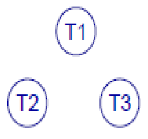


Precedence/Wait-For Graphs

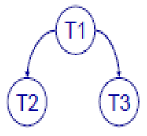
- Precedence graph
  - Each transaction is a vertex
  - Arcs from T1 to T2 if
    - T1 reads X before T2 writes X
    - T1 writes X before T2 reads X
    - T1 writes X before T2 writes X
- Wait-for Graph
  - Each transaction is a vertex
  - Arcs from T2 to T1 if
    - T1 read-locks X then T2 tries to write-lock it
    - T1 write-locks X then T2 tries to read-lock it
    - T1 write-locks X then T2 tries to write-lock it

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



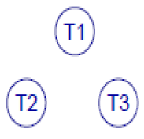
Wait for graph



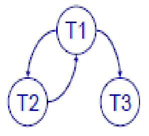
Precedence graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



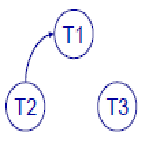
Wait for graph



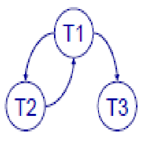
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) **tries read-lock(X)**
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



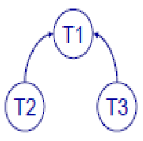
Wait for graph



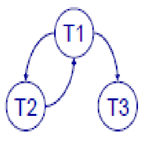
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) read-locks(Y)
T1 Write(X) **write-lock(X)**
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) **tries read-lock(X)**
T1 Write(Y)



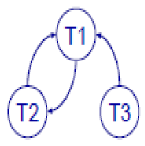
Wait for graph



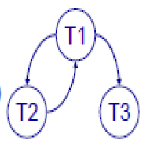
Precedence graph

Example

T1 Read(X) read-locks(X)
T2 Read(Y) **read-locks(Y)**
T1 Write(X) write-lock(X)
T2 Read(X) tries read-lock(X)
T3 Read(Z) read-lock(Z)
T3 Write(Z) write-lock(Z)
T1 Read(Y) read-lock(Y)
T3 Read(X) tries read-lock(X)
T1 Write(Y) **tries write-lock(Y)**



Wait for graph



Precedence graph

Deadlock Prevention

- Deadlocks can arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether
- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - Low 'lock utilisation' - transactions can hold on to locks for a long time, but not use them much