Lecture 47
Deadlock

Deadlock

- A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and … , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds.
- None of the transactions can make progress in such a situation.
- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock.
- Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

Deadlock Handling

- There are two principal methods for dealing with the deadlock problem.
  1. We can use a deadlock prevention protocol to ensure that the system will *never* enter a deadlock state.
  2. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme.
- Both methods may result in transaction rollback.
- Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.
- Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

Deadlock Prevention

- There are two approaches to deadlock prevention.
  1. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.
  2. The other approach is closer to deadlock recovery, and it performs transaction rollback instead of waiting for a lock whenever the wait could potentially result in a deadlock.
- The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked.
- There are two main disadvantages to this protocol:
  1. It is often hard to predict, before the transaction begins, what data items need to be locked;
  2. data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Deadlock Prevention

- Another approach for preventing deadlocks is to impose an ordering of all data items and to require that a transaction lock data items only in a sequence consistent with the ordering, one such scheme in the tree protocol, which uses a partial ordering of data items.
- A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.
- This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.
- There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

Deadlock Prevention

- The second approach for preventing deadlocks is to use preemption and transaction rollbacks.
- In preemption, when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be preempted by rolling back of T_i, and granting of the lock to T_j.
- To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins.
- The system uses these timestamps only to decide whether a transaction should wait or roll back.
- Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted.

Deadlock Prevention

- Two different deadlock-prevention schemes using timestamps have been proposed:
  1. The **wait–die** scheme is a **non-preemptive** technique. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e., T_i is older than T_j). Otherwise, T_i is rolled back (dies).

     For example, suppose that transactions T_1, T_2, and T_3 have timestamps 5, 10, and 15, respectively. If T_1 requests a data item held by T_2, then T_1 will wait. If T_3 requests a data item held by T_2, then T_3 will be rolled back.

Deadlock Prevention

- Two different deadlock-prevention schemes using timestamps have been proposed:
  2. The **wound–wait** scheme is a **preemptive** technique. It is a counterpart to the wait– die scheme. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (i.e., T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is *wounded* by T_i).

     For example, with transactions T_1, T_2, and T_3, if T_1 requests a data item held by T_2, then the data item will be preempted from T_2, and T_2 will be rolled back. If T_3 requests a data item held by T_2, then T_3 will wait.
- The major problem with both of these schemes is that unnecessary rollbacks may occur.

GATE Question

In a certain operating system, deadlock prevention is attempted using the following scheme. Each process is assigned a unique timestamp, and is restarted with the same timestamp if killed. Let Ph be the process holding a resource R, Pr be a process requesting for the same resource R, and T(Ph) and T(Pr) be their timestamps respectively. The decision to wait or preempt one of the processes is based on the following algorithm.

if T(Pr) < T(Ph)

then kill Pr

else wait

Which one of the following is TRUE?

(A) The scheme is deadlock-free, but not starvation-free

(B) The scheme is not deadlock-free, but starvation-free

(C) The scheme is neither deadlock-free nor starvation-free

(D) The scheme is both deadlock-free and starvation-free

[GATE 2004]

GATE Question

In a database system, unique timestamps are assigned to each transaction using Lamport's logical clock. Let TS(T_1) and TS(T_2) be the timestamps of transactions T_1 and T_2 respectively. Besides, T_1 holds a lock on the resource R and T_2 has requested a conflicting lock on the same resource R. The following algorithm is used to prevent deadlocks in the database system assuming that a killed transaction is restarted with the same timestamp.

if TS(T2)<TS(T1) then

T1 is killed

else T2 waits.

Assume any transaction that is not killed terminates eventually. Which of the following is TRUE about the database system that uses the above algorithm to prevent deadlocks?

(A) The database system is both deadlock-free and starvation-free.

(B) The database system is deadlock-free, but not starvation-free.

(C) The database system is starvation-free, but not deadlock-free.

(D) The database system is neither deadlock-free nor starvation-free. **[GATE 2017]**

Deadlock Prevention

- Another simple approach to deadlock prevention is based on **lock timeouts**.
- In this approach, a transaction that has requested a lock waits for at most a specified amount of time.
- If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.
- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.
- This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.

Deadlock Prevention

- The timeout scheme is particularly easy to implement, and it works well if transactions are short and if long waits are likely to be due to deadlocks.
- However, in general it is hard to decide how long a transaction must wait before timing out.
- Too long a wait results in unnecessary delays once a deadlock has occurred.
- Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.
- **Starvation** is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

Deadlock Detection and Recovery

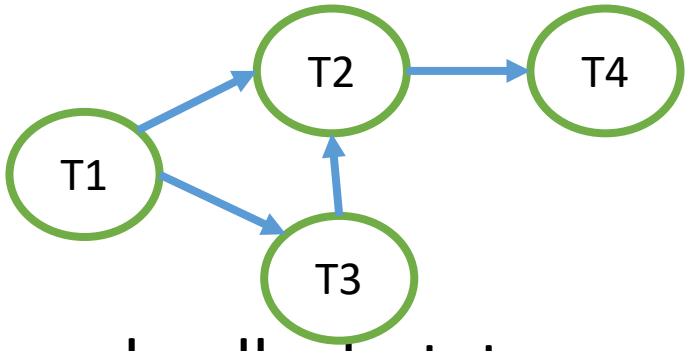
- If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred.
- If one has, then the system must attempt to recover from the deadlock.
- To do so, the system must:
  1. Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
  2. Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
  3. Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a wait-for graph.
- This graph consists of a pair $G = (V , E)$, where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.
- When transaction T_i requests a data item currently being held by transaction T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph.
- This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.

Deadlock Detection

- A deadlock exists in the system if and only if the wait-for graph contains a cycle.
- Each transaction involved in the cycle is said to be deadlocked.
- To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.
- For example, consider the wait-for graph in Figure below, which depicts the following situation:
  1. Transaction *T*1 is waiting for transactions *T*2 and *T*3.
  2. Transaction *T*3 is waiting for transaction *T*2.
  3. Transaction *T*2 is waiting for transaction *T*4.



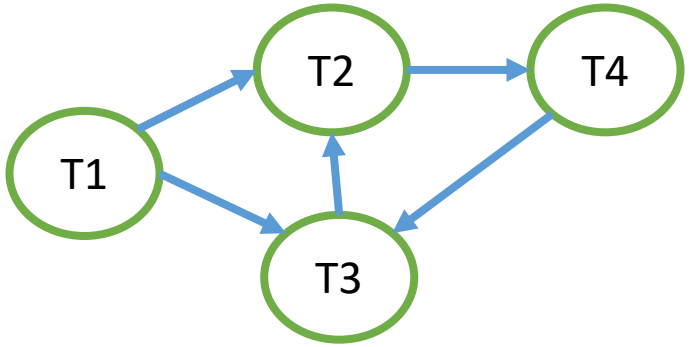
- Since the graph has no cycle, the system is not in a deadlock state.

Deadlock Detection

- Suppose now that transaction *T*4 is requesting an item held by *T*3. The edge *T*4 → *T*3 is added to the wait-for graph, resulting in the new system state in Figure below.
- This time, the graph contains the cycle:

  $$T2 \rightarrow T4 \rightarrow T3 \rightarrow T2$$

  implying that transactions *T*2, *T3*, and *T4* are all deadlocked.



Deadlock Recovery

- When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:
  1. **Selection of a victim.**
  - Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
    a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
    b) How many data items the transaction has used.
    c) How many more data items the transaction needs for it to complete.
    d) How many transactions will be involved in the rollback.

Deadlock Recovery

2. Rollback.

- Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- The simplest solution is a **total rollback**: Abort the transaction and then restart it.
- However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions.
- Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded.
- The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.
- The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point.
- The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback.

Deadlock Recovery

3. Starvation.

- In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim.
- As a result, this transaction never completes its designated task, thus there is starvation.
- We must ensure that a transaction can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.

For Video lecture on this topic please subscribe to my youtube channel.

The link for my youtube channel is

https://www.youtube.com/channel/UCRWGtE76JlTp1iim6aOTRuw?sub_confirmation=1