Problem M16.1: Transactional Memory (Spring 2015 Quiz 4, Part B)

Ben Bitdiddle wants to implement a transactional memory system with pessimistic conflict detection in a two-core processor. This system has the following characteristics:

- When a transaction starts, it is assigned a unique global timestamp.
- The memory system tracks the set of addresses read or written by each transaction (i.e., its **read set** and **write set**).
- For every transactional load, the memory system checks whether this load reads an address in the **write set** of any other transaction, and declares a conflict if so.
- For every transactional store, the memory system checks whether this store writes an address in the **read set or write set** of any other transaction, and declares a conflict if so.
- On a conflict, the transaction with the later timestamp aborts.
- An aborted transaction restarts execution 10 cycles later.

Ben runs a program with two types of transaction: X and Y, shown below.

Cycle relative to start	Transaction X
Cycle 0	Starts
Cycle 10	Read B
Cycle 20	Read A
Cycle 30	Write A
Cycle 40	Ends

Cycle relative to start	Transaction Y
Cycle 0	Starts
Cycle 10	Read B
Cycle 20	Read A
Cycle 30	Read B
Cycle 40	Ends

Problem M16.1.A

Suppose the system is executing two transactions: a type X transaction that starts at cycle 0 and receives timestamp 0, and a type Y transaction that starts at cycle 5 and receives timestamp 5. Is there a conflict between these two transactions? If so, at what cycle does this conflict happen?

There is a conflict at cycle 30 due the write A in transaction X.

Problem M16.1.B

Ben implements conflict detection by extending a conventional MSI coherence protocol. Furthermore, drawing inspiration from the delay invalidation cache coherence protocol in Quiz 3, Ben wants to optimize his transactional memory system as follows:

- When a core receives an abort for its currently running transaction, it delays the abort until the next local cache miss. If the transaction finishes without additional misses, it will commit successfully.

With this optimization, assume the same scenario as in the previous question: a type X transaction that starts at cycle 0 and receives timestamp 0, and a type Y transaction that starts at cycle 5 and receives timestamp 5. Are any of these transactions aborted? If so, when do aborts happen?

No, since the optimization delays the abort for transaction Y, and it does not miss after that, transaction Y will commit. This is logically same as Y starts before X.

Does this optimization always provide correct transactional semantics? Explain your answer in one or two sentences.

No, it does not provide correct transactional semantics. Consider the following example:

Cycle relative to start	Transaction X
Cycle 0	Starts
Cycle 20	Read A
Cycle 30	Use value A to Write C
Cycle 40	Ends

Cycle relative to start	Transaction Y
Cycle 0	Starts
Cycle 10	Write A
Cycle 20	Read B
Cycle 30	Ends

If X starts at 0, and Y starts at 5, Y will abort at cycle 25 due to read miss, but X will read the data from Y since at cycle 20, it sees the write from Y. Finally, X commits will modification that should have abort.

Problem M16.1.C

Ben believes this optimization works well and always needs fewer cycles to complete transactions. Is he correct? If so, explain why this always improves performance with one or two sentences. Otherwise, provide an example where this optimization causes a transaction to finish later.

No, Ben is incorrect. This optimization is somehow similar to optimistic conflict detection, so it's possible that it takes longer to finish transactions. For example, if a transaction should have abort at cycle 10, but delay the abort till later, it will start later and thus finish later.

Problem M16.2: Transactional Memory (Spring 2016 Quiz 4, Part D)

You are designing a hardware transactional memory (HTM) system that uses pessimistic concurrency control (i.e., on each load/store, the HTM checks for conflicting accesses to the same address made by other transactions). Comment on whether the following conflict resolution policies suffer from either livelock (i.e., the system may reach a state where *no single transaction* makes forward progress) or starvation (i.e., the system may reach a state where *at least one transaction* does not make forward progress). State your reasoning.

1. **Requester wins**: Upon a conflict, the transaction whose request initiated the conflict check is granted access to the data, and any conflicting transactions are aborted. After aborting, transactions immediately restart execution.

   This policy can livelock. Transactions A and B that conflict, can end up aborting each other similar to the scenario discussed in L23-19. This policy is also prone to starvation if a transaction gets aborted by conflicting transactions repeatedly.

2. **Timestamp-based, retain timestamp on abort**: Each transaction is assigned a unique timestamp when it first begins execution. Timestamps are monotonically increasing. Upon a conflict, if the requesting transaction's timestamp is lower than the timestamps of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original timestamp when they restart execution.

   Cannot livelock or starve. At some point, a transaction becomes the oldest transaction in the system (i.e. with the lowest timestamp), and can proceed to completion (commit) at that point.

3. **Timestamp-based, discard timestamp on abort**: Like the previous policy, except that aborted transactions discard their previous timestamp and acquire a new one when they restart execution.

   This policy cannot livelock since the lowest timestamp transaction at any point can commit. However, this policy can lead to starvation, since an aborted transaction acquires a new timestamp on restarting execution. It is possible that it repeatedly conflicts with lower timestamp transactions, and is aborted.

4. **Random-number-based, retain random number on abort**: Each transaction is assigned a unique random number when it first begins execution. Upon a conflict, if the requesting transaction's random number is lower than the random numbers of all other conflicting transactions, the requester is granted access to the data, and other conflicting transactions are aborted. Otherwise, the requesting transaction is aborted.

   After aborting, transactions immediately restart execution. Aborted transactions retain their original random number when they restart execution.

   This policy cannot livelock. The lowest timestamp transaction will complete unless a new conflicting transaction with lower timestamp arrives in the system (and issues a conflicting memory access) before completion of this transaction. Eventually, we should generate a transaction with minimum random number allowing it to complete. The policy can however lead to starvation if a transaction is assigned the maximum possible random number.

5. **Random-number-based, discard random number on abort**: Like the previous policy, except that aborted transactions discard their previous random number and acquire a new one when they restart execution.

   This policy cannot livelock (reason similar to the previous question). Since an aborted transaction receives a new timestamp on restarting execution, this policy avoids starvation.

Problem M16.3: Transactional Memory (Spring 2020 Quiz 4, Part C)

Ben Bitdiddle is designing a hardware transactional memory (HTM) system. He is concerned about three potential issues arising in his system:

1. *Deadlock*: Some transactions stay stalled indefinitely on a cyclic waiting pattern, so they neither commit nor abort.

2. *Livelock*: Some transactions can execute, but no transaction ever commits (e.g., due to repetitive aborts and re-execution). Thus, the system does not make forward progress.

3. *Starvation*: Some transactions can commit, but at least one other transaction is prevented from committing indefinitely. Thus, one or a subset of transactions does not make forward progress.

Ben wants to classify each of the 4 HTM systems in Questions 1 to 4 as one of four types, according to the forward progress guarantees they provide:

A. May deadlock
B. May livelock, but cannot deadlock
C. May starve, but cannot deadlock or livelock
D. Cannot deadlock, livelock, or starve

For **Questions 1 to 4, write down the letter A, B, C, or D and explain your choice**. You can either explain intuitively why an issue cannot arise, or use an example to show that the system suffers from the issue.

When you choose a particular option, you only need to explain the issues it differentiates between. For example, if you choose B, you should explain why the system cannot deadlock, and describe an example of how it may livelock.

Problem M16.3.A

C.

This system cannot livelock, because transactions can only be aborted by committing transactions, so forward progress for at least one transaction is guaranteed.

But it suffers from starvation. Example:
A long transaction can be repetitively aborted by many short transactions that update the data it reads.

Problem M16.3.B

B.

This system cannot deadlock because no transaction is stalled due to conflicts.

But it may livelock. Example:
Two conflicting transactions that write the same data can repetitively abort each other

Problem M16.3.C

A.

This system can deadlock. Example:

TX0: LD A ST B (TX0 stalls, waiting for TX1's commit/abort)
TX1: LD B ST A (TX1 stalls, waiting for TX0's commit/abort)

Problem M16.3.D

D.

This system is free from deadlocks, livelocks, and starvations.
The transaction with the lowest/oldest timestamp is guaranteed to commit. Because timestamps are assigned when beginning execution, starvation cannot happen: each transaction is guaranteed to eventually become the oldest one in the system.

Problem M16.3.E

HTM 1: Potentially serialized commits
HTM 4: Potentially serialized timestamp assignment
Lazy versioning is slow during commits but it happens locally and does not involve serialization among multiple transactions.

Problem M16.3.F

HTM 1: Yes, (N-1)N/2
HTM 2: No, due to livelocks
HTM 3: No, due to deadlocks

EDIT: The question is not clear about how many memory locations receive conflicted accesses. If there is only one such location, deadlock won't happen on HTM 3. We've adjusted the scores for students who reason in this way