Concurrent Algorithms	December 3, 2019

Solutions to Exercise 9

Problem 1.

Below are transactional memory executions. For each execution:

- Specify whether it is *opaque* or not.
- If it is not, suggest a modification to make it *opaque*.
- Specify an equivalent sequential execution of transactions.

Reminder: An execution is *opaque* if it is equivalent to some sequential execution in which every transaction, even aborted or unfinished, observes a consistent state of the memory. Transaction T in a sequential execution observes a consistent state of the memory if for every transactional variable x every read operation on x within the transaction returns: (I) the value written by the last write operation on x in the transaction T, or (II) the value written by the last write operation on x within a committing transaction, if there are no write operations on x in T, or (III) the initial value of x if there are no write operations on x within the execution.

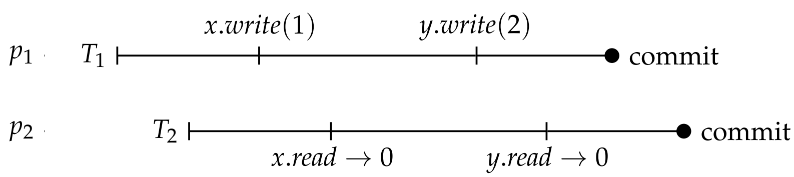

Figure 1: Transactional execution 1.

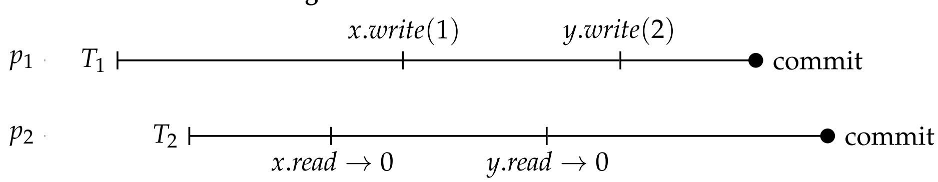

Figure 2: Transactional execution 2.

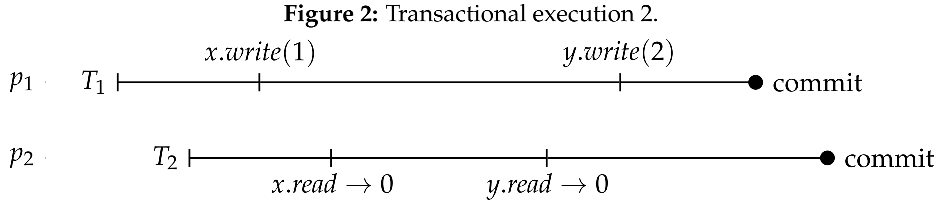

Figure 3: Transactional execution 3.

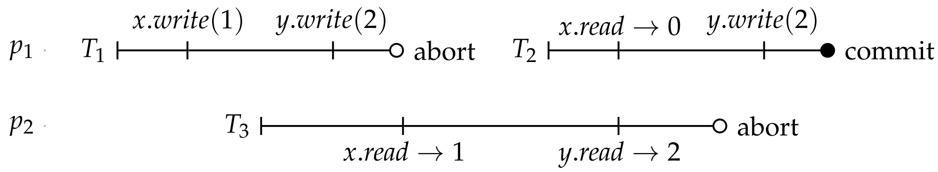

Figure 4: Transactional execution 4.

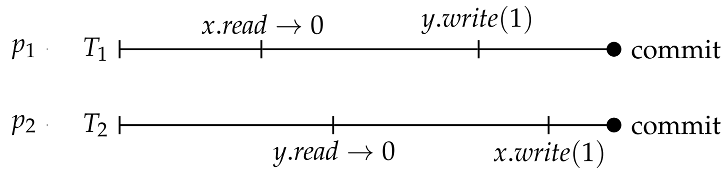

Figure 5: Transactional execution 5.

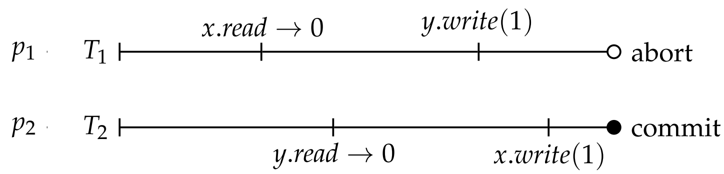

Figure 6: Transactional execution 6.

Solution

- Figure 1. Yes. An equivalent serial execution is $T_2 \cdot T_1$.
- Figure 2. Yes. An equivalent serial execution is $T_2 \cdot T_1$.
- Figure 3. Yes. An equivalent serial execution is $T_2 \cdot T_1$.
- Figure 4. No. The execution is not opaque because T_3 observes results of T_1's actions even though T_1 is aborted. One way to make it opaque is to have the read operations in T_3 return 0. In this case an equivalent sequential execution is $T_1 \cdot T_3 \cdot T_2$.
- Figure 5. No. The execution is not opaque because if T_1 is serialized before T_2, then T_2 does not observe the write to y; and if T_2 is serialized before T_1, then T_1 does not observe the write to x. One way to make the execution opaque is to abort one of the transactions. Another is to have read operation in T_1 return 1. In this case an equivalent serial execution is $T_2 \cdot T_1$.
- Figure 6. Yes. An equivalent sequential execution is $T_1 \cdot T_2$.

Problem 2.

Implement the following objects using transactional memory:

- A snapshot.
- A strong counter.
- A compare-and-swap that works on several locations in an array. It takes the indices of the locations, expected old values and the new values as parameters. Only if all the locations in the array have the expected values, it swaps them with the new ones.

Solution

To implement these objects using transactional memory, we only need to enclose their sequential specification in an atomic block.

Snapshot:

uses: $array[M]$

upon $Snapshot$ **do**
- $begin_{transaction}$;
- **for** $i = 1$ *to* M **do**
  - $ret[i] \leftarrow array[i]$;
- $end_{transaction}$;
- **return** ret

Counter:

initially: $count = 0$

upon Inc **do**
- $begin_{transaction}$;
- $ret \leftarrow count$;
- $count \leftarrow count + 1$;
- $end_{transaction}$;
- **return** ret

CASN:

uses: $array[M]$

upon $CASN(idx, oldv, newv)$ **do**
- $begin_{transaction}$;
- $L \leftarrow length(idx)$;
- **for** $i = 1$ *to* L **do**
  - **if** $array[idx[i]] \neq oldv[i]$ **then**
    - $end_{transaction}$;
    - **return** $array$
- **for** $i = 1$ *to* L **do**
  - $array[idx[i]] \leftarrow newv[i]$
- $end_{transaction}$;
- **return** $array$