COMPSCI 2DB3 Assignment 8

Prakhar Saxena

MacID: saxenp4

Student number: 400451379

3 April 2024

Contents

Question 1

The proposed approach does **not** follow strict two-phase locking and also does **not** follow two-phase locking. To see this, we consider the execution of a transaction

$$\tau_1 = \text{transfer}(Aple, \text{Bo, Alicia});\ \text{transfer}(Pear, \text{Eva, Celeste})$$

```
1: −− for-loop of Line 2 with transfer(Apple, Bo, Alicia).
2: $Lock_{\tau 1}(Bo)$. −− EXECUTE-TRANSACTION($\tau$, Bo, Apple).
3: $Read_{r1}(Bo)$. −− checks if Bo holds a copy of Apple.
4: $Release_{r1}(Bo)$. −− At Line 5 of Execute transaction.
5: −− Store the operation “update-inventory($\tau$, Bo, Apple, True)” in Commit.
6: −− Store the operation “update-inventory($\tau$, Alice, Apple, False)” in Commit.
7: −− for-loop of Line 2 with transfer(Pear, Eva, Celeste).
8: $Lock_{r1}(Eva)$. −− EXECUTE-TRANSACTION($\tau$, Eva, Pear).
9: $Read_{r1}(Eva)$. −− checks if Eva holds a copy of pear.
10: $Release_{r1}(Eva)$. −− At Line 9 of Execute transaction.
11: −− At Line 13 of Execute transaction, perform everything in commit.
12: $Lock_{r1}(Bo)$. −− Update-TRANSACTION($\tau$, Bo, Apple, True).
13: $Read_{r1}(Bo)$. −− checks if remove == True holds, which it does.
14: $Write_{r1}(Bo)$. −− removes a copy of Apple from Bo.
15: $Release_{r1}(Bo)$. −− At Line 7 of Update transaction.
16: $Lock_{r1}(ALicia)$. −− Update-TRANSACTION($\tau$, Alicia, Apple, False).
17: $Read_{r1}(Alicia)$. −− checks if remove == True holds, which it doesnt.
18: $Write_{r1}(Alicia)$. −− add a copy of Apple from Alicia.
19: $Release_{r1}(Alicia)$. −− At Line 7 of Update transaction.
```

Question 2

A deadlock happens when a transaction obtains a lock while already having another one. For example, if we have 2 transactions τ and, τ' and they are trying to lock onto 2 objects, O' and O respectively then, τ will be waiting on τ' to release its deadlock on O' while subsequently τ' is wait for τ to release its deadlock on O. In this scenario no deadlocks occur as no transaction will obtain a lock.

Question 3

The designed approach minimizes conflicts really well. However, there are still some instances of interleaved transactions where conflicts may arise. We use the notation "UP" for update inventory and "ET" for execute transaction which checks whether the "from" has a copy of the object or not. We design 2 transactions that clearly show a read-write conflict and a write-read conflict below. τ_2 tries to write an object to Alicia which was read in τ_1, this shows a read-write conflict. Also, τ_2 writes a value for Bo which is also read by τ_2 for Bo without being commited, this is a write read conflict.

$$\tau_1 = \text{transfer}(\mathit{Apple}, \text{Alicia, Bo});\ \text{transfer}(\mathit{Pear}, \text{Eva, Celeste})$$

$$\tau_2 = \text{transfer}(\mathit{Apple}, \text{Alicia, Celeste})$$

Player	**Inventory**
Alicia	1 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	0 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{ET(Apple, Alicia, Bo)}}$

Player	**Inventory**
Alicia	1 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	0 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{ET(Apple, Alicia, Celeste)}}$

Player	**Inventory**
Alicia	1 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	0 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{ET(Pear, Eva, Celeste)}}$

Player	**Inventory**
Alicia	1 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	0 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{UP(Apple, Alicia, Bo)}}$

Player	**Inventory**
Alicia	0 Apple, 0 Pear
Bo	1 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	0 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{UP(Apple, Alicia, Celeste)}}$

Player	**Inventory**
Alicia	-1 Apple, 0 Pear
Bo	1 Apple, 0 Pear
Eva	0 Apple, 1 Pear
Celeste	1 Apple, 0 Pear

$\xrightarrow[\text{True}]{\text{UP(Pear, Eva, Celeste)}}$

Player	**Inventory**
Alicia	-1 Apple, 0 Pear
Bo	1 Apple, 0 Pear
Eva	0 Apple, 0 Pear
Celeste	1 Apple, 1 Pear

τ_1	τ_2
$Lock_{\tau1}(Bo)$	
$Read\tau1(Bo)$	
$Release\tau1(Bo)$	
	$Lock_{\tau2}(Bo)$
	$Read\tau2(Bo)$
	$Release\tau2(Bo)$
Lock$_{\tau1}$(Eva)	
Read$_{\tau1}$(Eva)	
Release$_{\tau1}$(Eva)	
Lock$_{\tau1}$(Bo)	
Read$_{\tau1}$(Bo)	
Write$_{\tau1}$(Bo)	
Release$_{\tau1}$(Bo)	
	Lock$_{\tau2}$(Bo)
	Read$_{\tau2}$(Bo)
	Write$_{\tau2}$(Bo)
	Release$_{\tau2}$(Bo)
	Commit$_{\tau2}$
Lock$_{\tau1}$(Eva)	
Read$_{\tau1}$(Eva)	
Write$_{\tau1}$(Eva)	
Release$_{\tau1}$(Eva)	
Commit$_{\tau1}$	

Question 4

Dirty reads happen when one transaction reads data that has not been committed of the other transaction and vice-versa. Referring to the examples done in Q3, Also, τ_2 writes a value for Bo which is also read by τ_2 for Bo without being committed. This shows that the proposed approach has dirty reads.

Question 5

Unrepeatable reads are when a transaction reads an object and get a value and when it reads the same object again, it gets a different value every time. The proposed approach suffers from unrepeatable reads. We refer to the transactions τ_1 and τ_2 used in the solution of Evaluation Question 3 and the second interleaved execution and schedule. In the example provided in Q3 τ_2 read the value of 1 Apple from Bo and then after re-reading it gets the value of 0 Apples from Bo proving unrepeatable reads.

Question 6

$$\tau_1 = \text{transfer}(Aple, \text{Alicia, Bo})$$

$$\tau_2 = \text{transfer}(Aple, \text{Bo, Eva})$$

Initial database:

Player	**Inventory**
Alicia	1 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	0 Apple, 0 Pear
Celeste	0 Apple, 0 Pear

First τ_1 then τ_2:

Player	**Inventory**
Alicia	0 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	1 Apple, 0 Pear
Celeste	0 Apple, 0 Pear

First τ_2 then τ_1:

Player	**Inventory**
Alicia	0 Apple, 0 Pear
Bo	0 Apple, 0 Pear
Eva	1 Apple, 0 Pear
Celeste	0 Apple, 0 Pear

Absolutely not: we have already seen that the proposed approach suffers from both dirty reads and unrepeatable reads. Alternatively, we shall show a transaction execution that is not equivalent to any serial execution. We refer to the transactions τ_1 and τ_2 used in the solution of Evaluation Question3. The only two serial executions of τ_1 and τ_2 are first fully executing τ_1 then fully executing τ_2 or, first fully executing τ_2 then fully executing τ_1. Both end up with the same final databases as seen from the tables above. However, if we consider the interleaved execution, the initial transaction returns a fail when first read as Bo has not received the apple yet as the database has not changed yet. Hence, the approach is not serializable.

Question 7

C1

No account should ever receive a negative balance (assuming that all accounts start with a positive balance).
As seen from the example transactions given in Q3, Alicia ended up with -1 Apples in the inventory even though she had a positive count for apples at the start of the transaction. Hence, the rule is violated.

C2

As the transfers only move items between inventories, no items should be lost or created. Hence, if at any time t no transactions are being executed, then the sum of the items of all inventories at that time t should be equivalent to the initial sum of the items of all inventories.
This constraint holds. This is because if an item is removed from account A and added to account B then both, the addition of item in account B and removal of item from account A are done in the same commit phase due to the way the code of execute transaction is written.

C3

Successful transactions must have their lasting effects, while failed transactions must not have lasting effects. Hence, if at any time t no transactions are being executed, then the inventory of each account should reflect the inventory updates due to all transactions that were executed successfully before t.
This constraint holds as, successful transactions execute balance removals through execute transaction operations. both removal and addition of items in inventory are done by update balance operations which has its lasting effects.