Operating Systems Workbench V2.1
Threads (Threads and Locks) Activities and Experiments

Richard Anthony January 2005

This laboratory sheet accompanies the '***Threads – Threads and Locks***' application within the Operating Systems Workbench.

1. Prerequisite knowledge

You should have a clear understanding of the following concepts: Operating System, Scheduling, Process, Thread.
You should have a basic understanding of the following concepts: Lock, Mutual Exclusion, Transaction.
You should be familiar with the '***Threads – Introductory***' application within the Operating Systems Workbench.

2. Introduction

This emulation has been designed to introduce you to the fundamentals of locking and mutual exclusion whilst keeping the complexity low.
Transactions can be executed with and without locks. You can choose at what point within a transaction to apply a lock, and at what point to release a lock. You can witness the 'lost-update' problem in action and can correct the problem by applying locks appropriately to ensure mutually exclusive access to a shared variable.
Two threads can be executed separately or simultaneously. Each thread executes update transactions (one decrements a variable, the other increments it). Without locking, the transactions of the two threads can interfere with each other.

Figure 1 shows the Threads (Threads and Locks) interface during an emulation in which both threads are concurrently accessing the shared variable without using locks.

The display is divided into a number of sections. Each section is briefly explained:

The 'ADD' Thread – This thread executes a sequence of one thousand update transactions on a shared variable. Each transaction reads the current value into thread-local storage, increments the value locally, and then writes the new value to the shared variable. There is a button to permit running this thread independently. A transaction counter is updated to allow monitoring of progress.

The 'SUBTRACT' Thread – This thread executes a sequence of one thousand update transactions on a shared variable. Each transaction reads the current value into thread-local storage, decrements the value locally, and then writes the new value to the shared variable. There is a button to permit running this thread independently. A transaction counter is updated to allow monitoring of progress.

Locking Configuration – Locking can be enabled or disabled (the default). When locking is enabled the user can elect the point during transactions at which a lock is applied on the shared variable, and the point at which the lock is released.

Data Field (shared resource) – This shows the value of the shared variable. The value is initially 1000. A button is provided to reset the value to 1000 before re-running emulations.

Start both threads button – This starts both threads at the same time. Since they both carry out the same number of simple updates they tend to complete at approximately the same time too (but this is not guaranteed or enforced). No synchronization is applied to the operation of the two threads (or to their access to the shared variable) unless the user explicitly configures locking.

Done button – This button exits the application immediately without preserving statistics or configuration settings. Control is returned to the top-level menu.

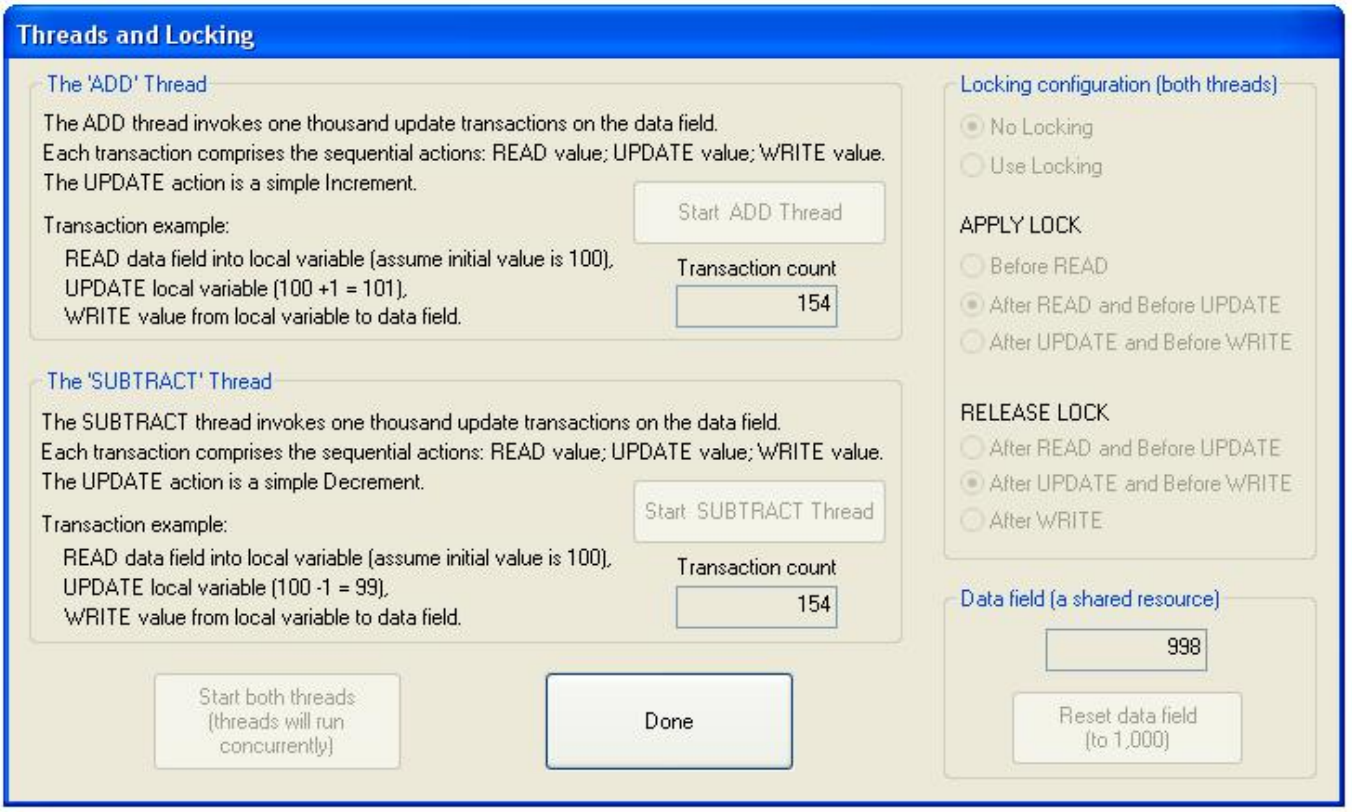


Figure 1. The Threads (Threads and Locks) interface.

The following 'lab activities' sections describe specific experiments and investigations to help you maximise the benefit of the software.

Please take care to read the instructions carefully for each step and try to follow instruction sequences carefully. Try to predict the outcome of experiments in advance if possible. If the outcome is not as expected try to determine why not. You can repeat experiments as often as required and can work at your own pace. Try repeating experiments with slight changes in the parameters – sometimes just changing one parameter slightly can lead to big differences in the results and can shed light on the relative importance of a particular aspect of the emulation.

For each activity the configuration settings are explained. In each case it is assumed that you have already started the 'Introductory Scheduling Algorithms' simulation application. To do this:
1. Start the **Operating Systems Workbench**.
2. From the main menu bar, select **Threads**.
3. From the drop-down menu, select **Threads and Locks**.

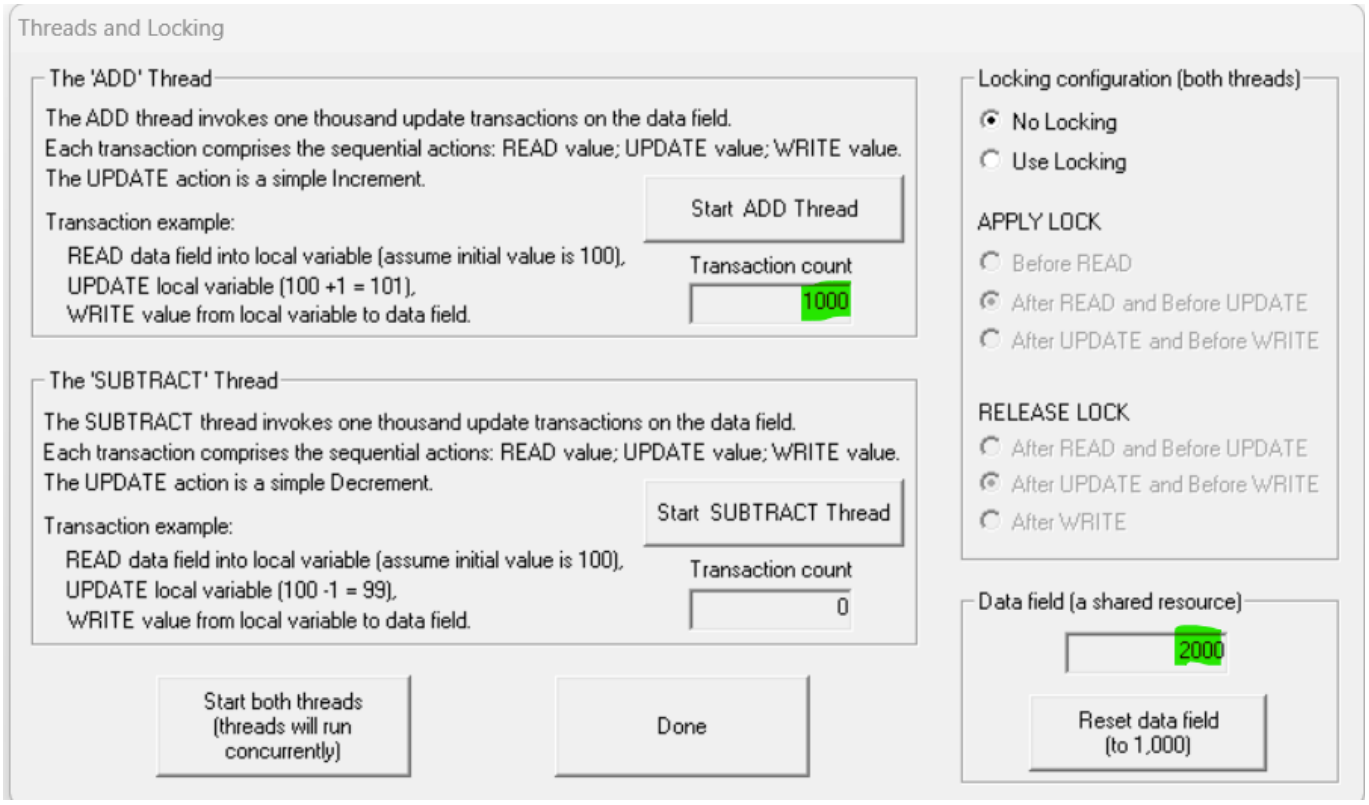
Lab Activity: Threads: Introductory: Lost-Update 1
Understanding the 'lost-update' problem

1. Inspect the initial value of the data field.

Q1. If one thousand increment updates are performed, what value should it end up at?

2000

2. Click the 'Start ADD Thread' button to run the ADD thread in isolation.



Q2. Was the result as expected? (if not, check your maths!).

Yes, it was.

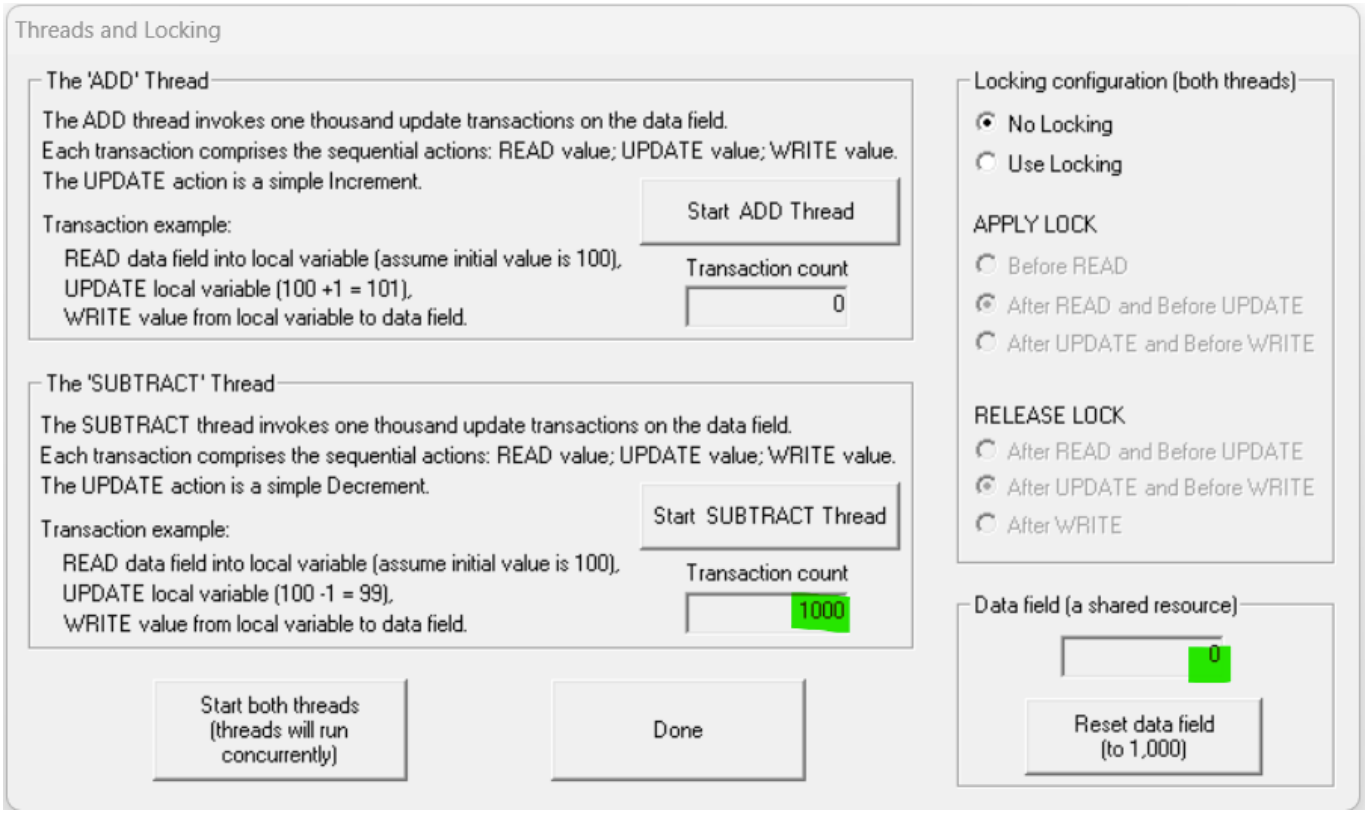
3. Note the value of the data field now.

2000

Q3. If one thousand decrement updates are performed, what value should it end up at?

0

4. Click the 'Start SUBTRACT Thread' button to run the SUBTRACT thread in isolation.



Q4. Was the result as expected? (if not, check your maths!).

Yes, it was.

Q5. If the threads are run sequentially (as above), are there any issues concerning access to the shared variable that can lead to its corruption?

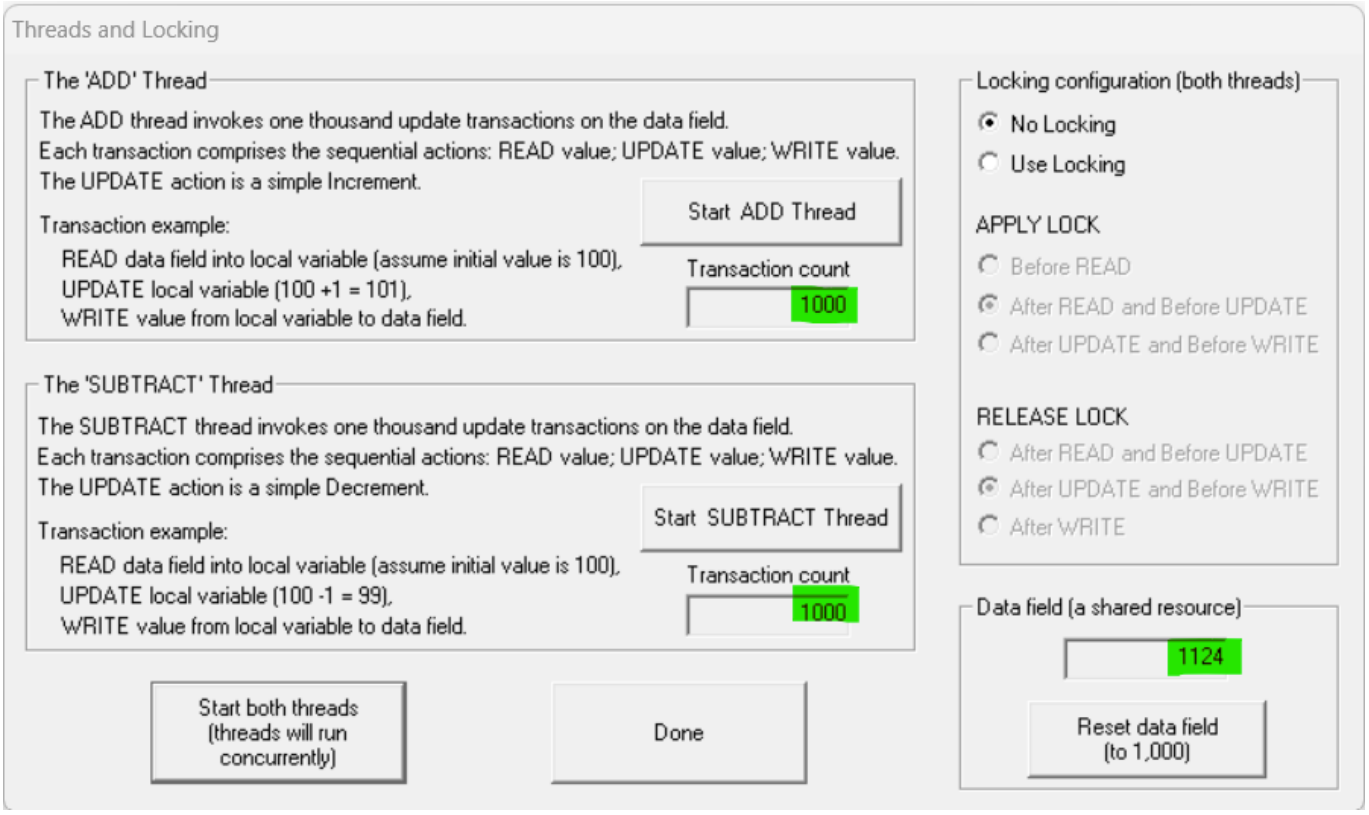
Yes, if multiple threads are accessing and modifying a shared variable concurrently without proper synchronization, it can lead to data corruption and various issues, commonly known as a race condition.

5. Click the ‘Reset data field’ button to reset the value of the data variable to 1000.

Q6. Given a starting value of 1000, if one thousand increment updates are performed, and one thousand decrements are performed, what should the final value be?

1000

6. Click the ‘Start both threads’ button to run the two threads concurrently.



7. When both threads have finished, check their transaction counts to ensure both executed 1000 transactions, hence determining the correct value of the data variable.

1124

Q8. Is the actual data variable value correct? If not what could have caused the discrepancy?

No, because of the lack of atomicity in the increment operation. The sequence of instructions from different threads can overlap, leading to unexpected and incorrect results

NGUYỄN MINH ĐỨC - ITITIU21045

Lab Activity: Threads: Introductory: Lost-Update
2 Understanding the 'lost-update' problem

The ***Lost-Update 1*** activity exposed the lost-update problem.

We saw that a discrepancy occurred between the expected value and the actual final value of a shared variable.
Some of the updates were lost. This is because one thread was allowed to access the variable whilst the other had already started a transaction based on the value of the variable.

Q1. How does this problem relate to transactions in database applications?

In database applications, if multiple transactions (similar to threads from example) are allowed to read and write data concurrently without proper concurrency control mechanisms, similar issues can occur.

For example, two transactions might read the same data simultaneously, make changes based on that data, and then both try to commit their changes, resulting in lost updates or other inconsistencies.

Q2. How many of the ACID properties of transactions have been violated?

Just one and it is Isolation.

Q3. How could this problem affect applications such as:
An on-line flight booking system?
A warehouse stock-management system?

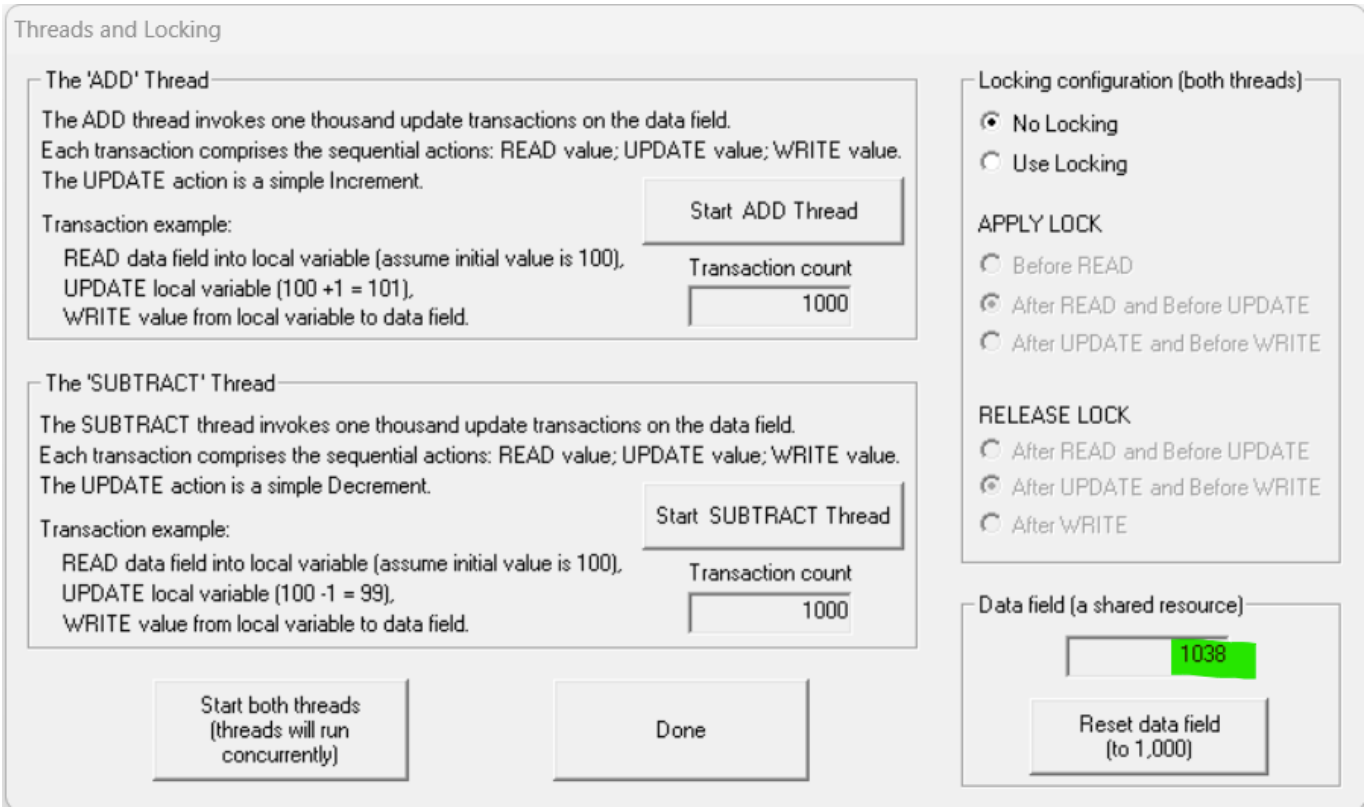
For an on-line flight booking system, it may cause lost reservation (many flights can be booked at the same time), inconsistent pricing (prices and seats can be changed between the time the users try to make a reservation), etc.

For a warehouse stock-management system, inventory inaccuracy (lost updates, discrepancies between actual and recorded stock levels), inconsistent data for users (users might see different quantities, which may lead to confusion and errors in decision-making).

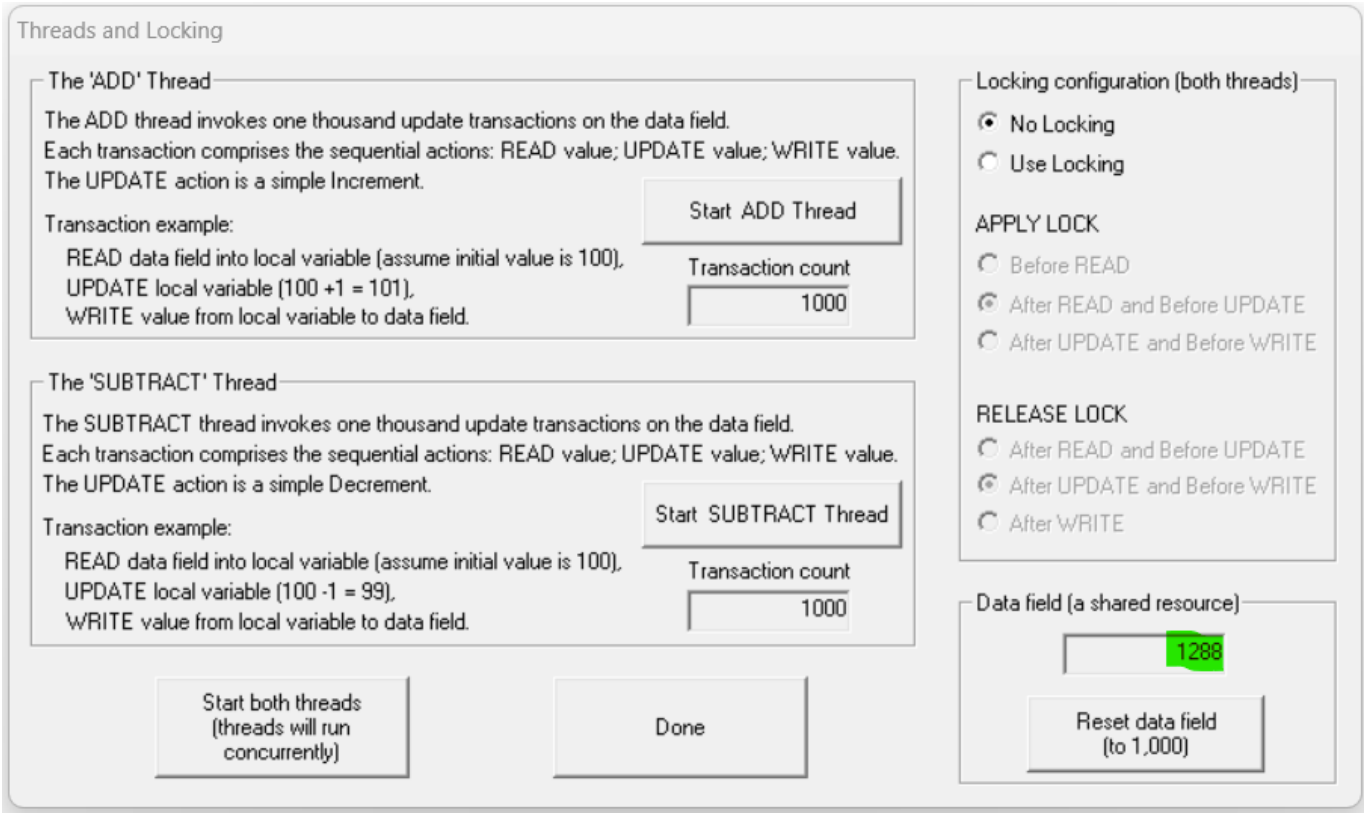
Q4. Think of at least one other real-world application that would be affected by the lost-update problem?

Cinema ticket-booking, online shopping order-management, courses assignment on websites.

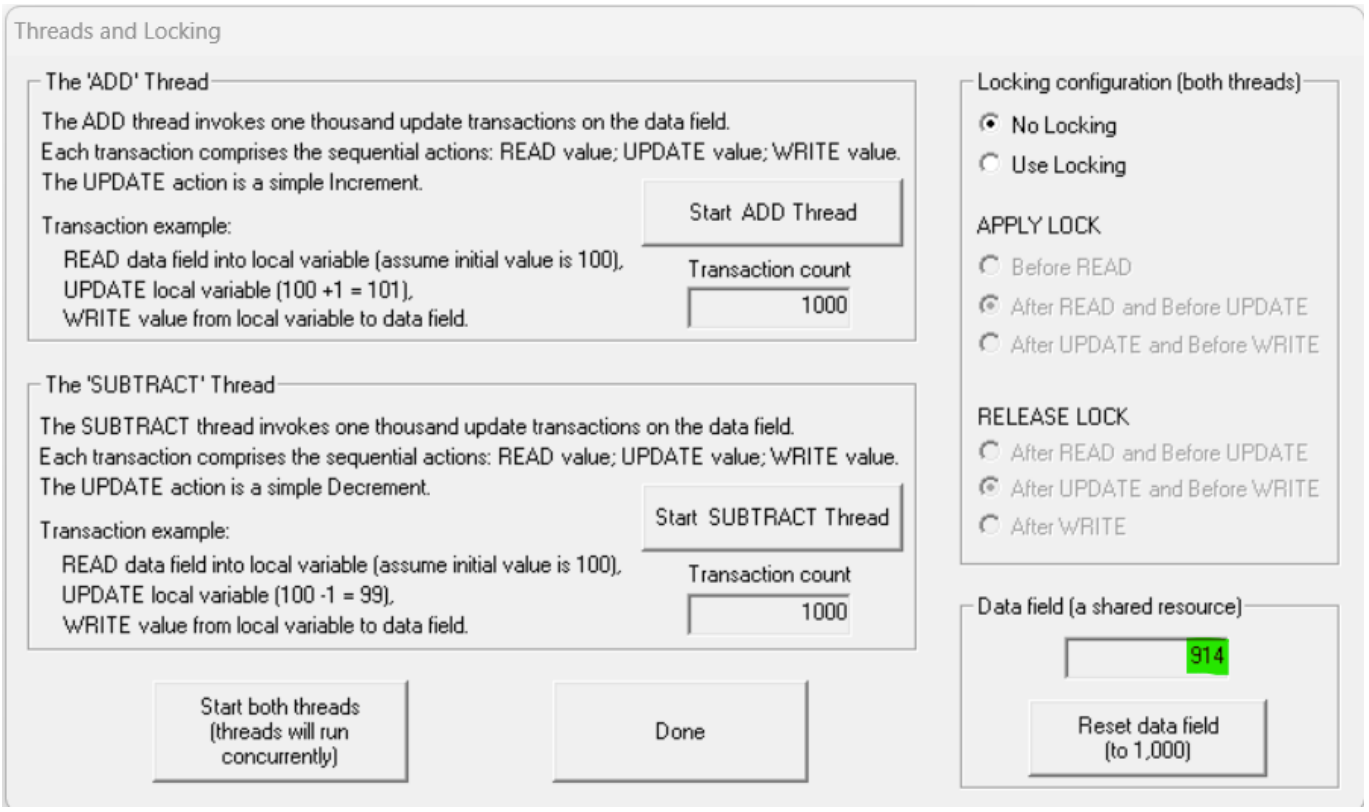
1. Investigate if the problem has a random aspect, or if the discrepancy is predictable. Repeat the following steps three times:
   A. Ensure the value of the data variable is 1000 (Click the 'Reset data field' button if necessary).
   B. Click the 'Start both threads' button to run the two threads concurrently.
   C. When both threads have finished check their transaction counts to ensure both executed 1000 transactions, hence determine the correct value of the data variable.
   D. Make a note of the extent of the discrepancy.



1st time: 1038 (Discrepancy: +38)



2nd time: 1288 (Discrepancy: +288)



3rd time: 1288 (Discrepancy: -86)

Q5. Compare the three discrepancies. Is the actual discrepancy predictable, or does it seem to have a random element?

It seems to have a random element.

Q6. If there is a random element, where does it come from (think carefully about the mechanism at play here – two threads are executing concurrently but without synchronisation – what could go wrong)?

It comes from the the lack of atomicity in the increment operation, which is due to a race condition. The sequence of instructions from different threads can overlap, leading to unexpected and incorrect results. It also may cause interleaved execution, randomness in execution timing, unpredictable thread preemption, inconsistency in execution order, etc.

Q7. Is the problem more serious if the discrepancy is predictable? Or if the discrepancy is not predictable?

In either case, the solution remains the same: implementing proper synchronization mechanisms to ensure atomicity and consistency in concurrent operations.

Predictable discrepancies may be somewhat easier to identify, but the underlying issue of non-atomic and unsynchronized operations still needs to be addressed to ensure the application's correctness and reliability.

NGUYỄN MINH ĐỨC - ITITIU21045

Lab Activity: Threads: Introductory: Locks 1
The need for locks

The ***Lost-Update 1*** and ***Lost-Update 2*** activities exposed the lost-update problem, and showed us that the extent of the problem is unpredictable. Therefore, we must find a mechanism to prevent the problem from occurring.

Q1. Could a locking mechanism be the answer? If so, how would it work?

Yes, it ensures that only one thread can access a critical section of code (which involves the shared resource) at any given time

Q2. Would it be adequate to apply the lock to only one of the threads, or does it have to apply to both?

To ensure the correct and safe synchronization of shared resources, the lock must be applied to both threads. If you apply the lock to only one of the threads, you won't achieve the desired synchronization, and race conditions may still occur.

Q3. Is it necessary to use a read-lock, a write-lock, or is it important to prevent both reading and writing while the lock is applied, to ensure that no lost-updates occur (think about the properties of transactions)?

In the context of preventing lost updates and ensuring the integrity of shared data, the most common approach is to use a write lock (or an exclusive lock). To ensure data integrity, prevent lost updates, and align with transaction properties, a write lock (exclusive lock) is often the preferred choice when dealing with shared resources that undergo both read and write operations.

Q4. At what point in this sequence should the lock be applied to the transaction?

The lock should be applied during the entire sequence to ensure the atomicity of the entire transaction. This ensures that the entire sequence is treated as a single, indivisible operation, preventing interleaving of operations by other threads.

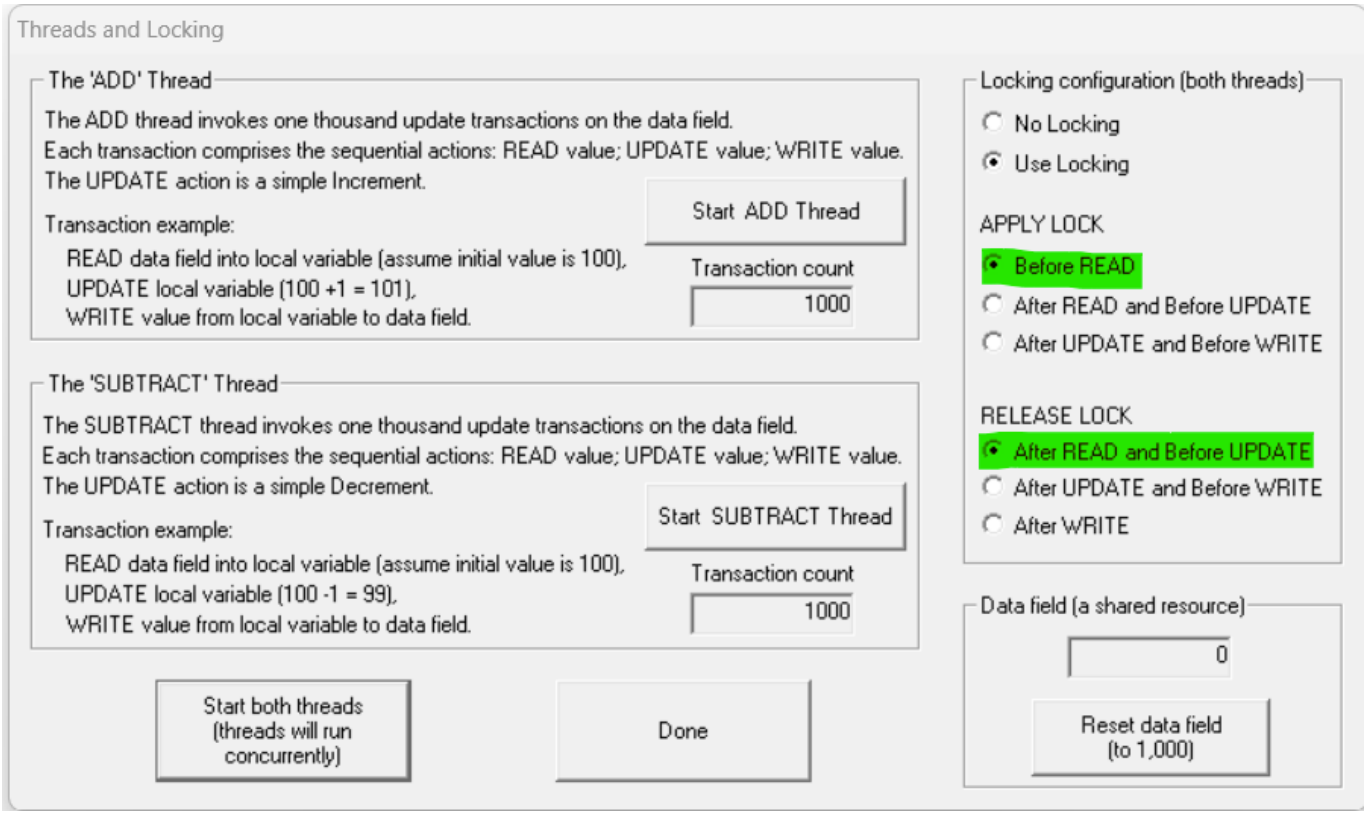
Q5. At what point should the lock be released?

The lock should be released after the entire transaction is completed. This means releasing the lock after the write operation, ensuring that the critical section protected by the lock is only held for the duration of the atomic transaction.

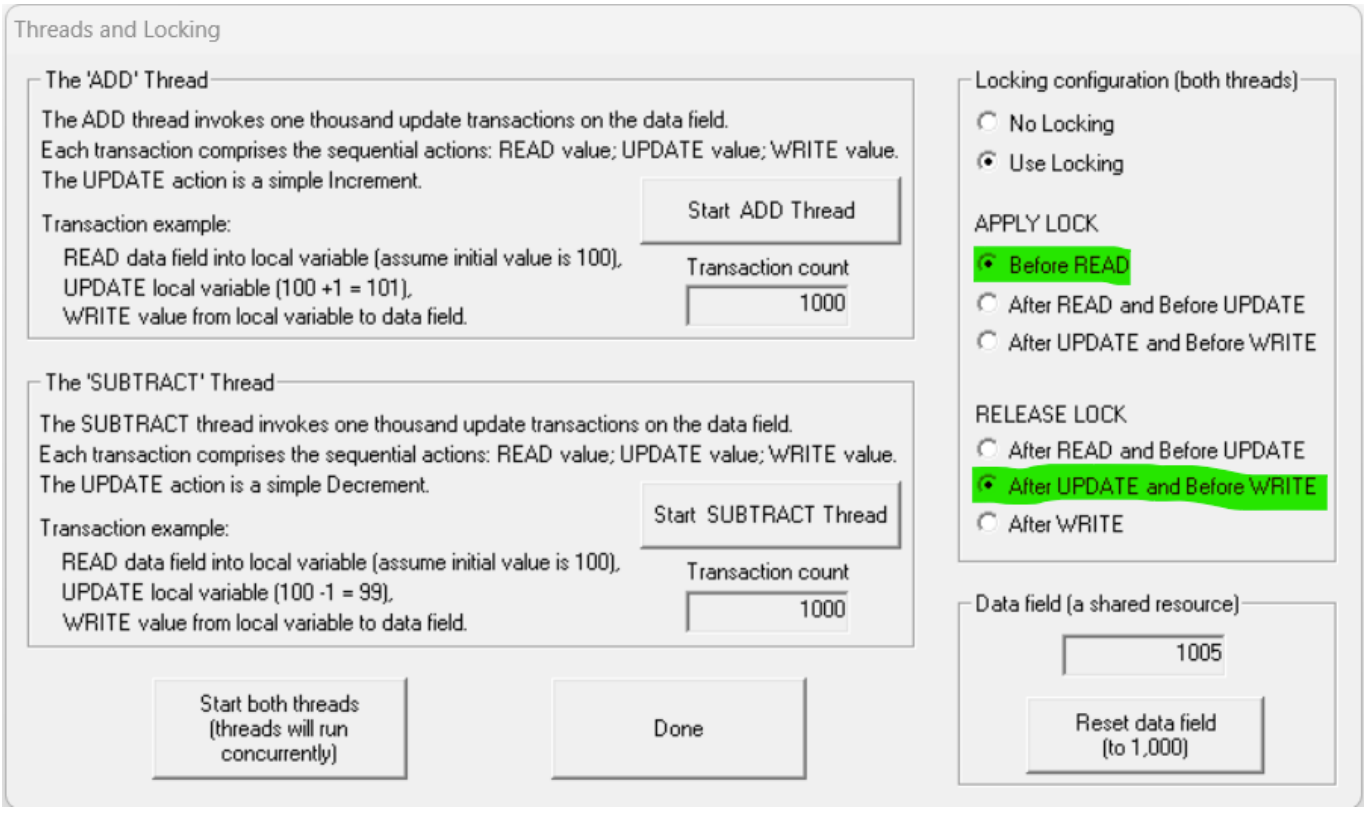
Investigate the various combinations of locking strategies:

1. Click the ‘Use Locking’ button.
2. Repeat the following steps until all combinations of lock apply and release have been tried:
    A. Ensure the value of the data variable is 1000 (Click the ‘Reset data field’ button if necessary).
    B. Select an option from the APPLY LOCK choices.
    C. Select an option from the RELEASE LOCK choices.
    D. Click the ‘Start both threads’ button to run the two threads concurrently.
    E. When both threads have finished check their transaction counts to ensure both executed 1000 transactions, hence determine the correct value of the data variable.
    F. Make a note of the extent of the discrepancy.

NGUYỄN MINH ĐỨC - ITITIU21045

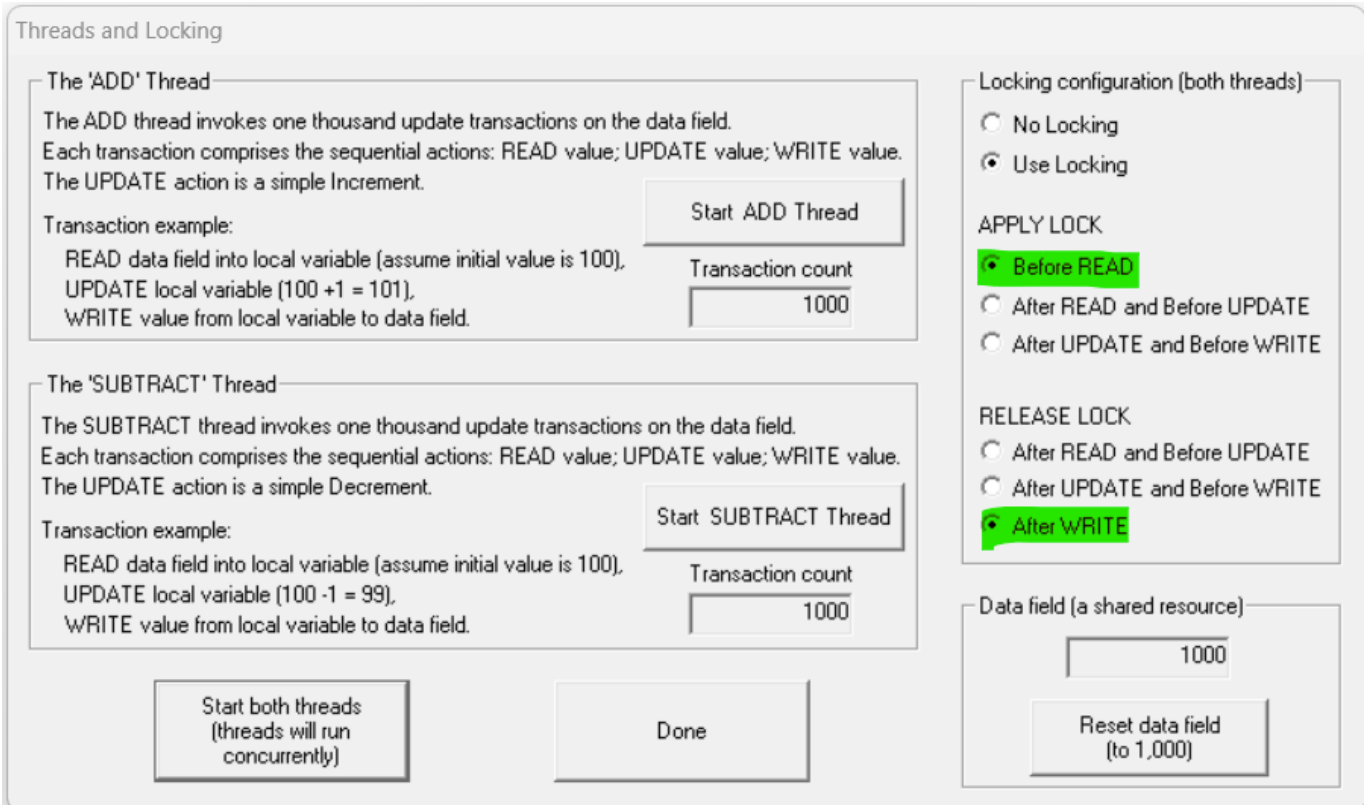


Discrepancy: -1000

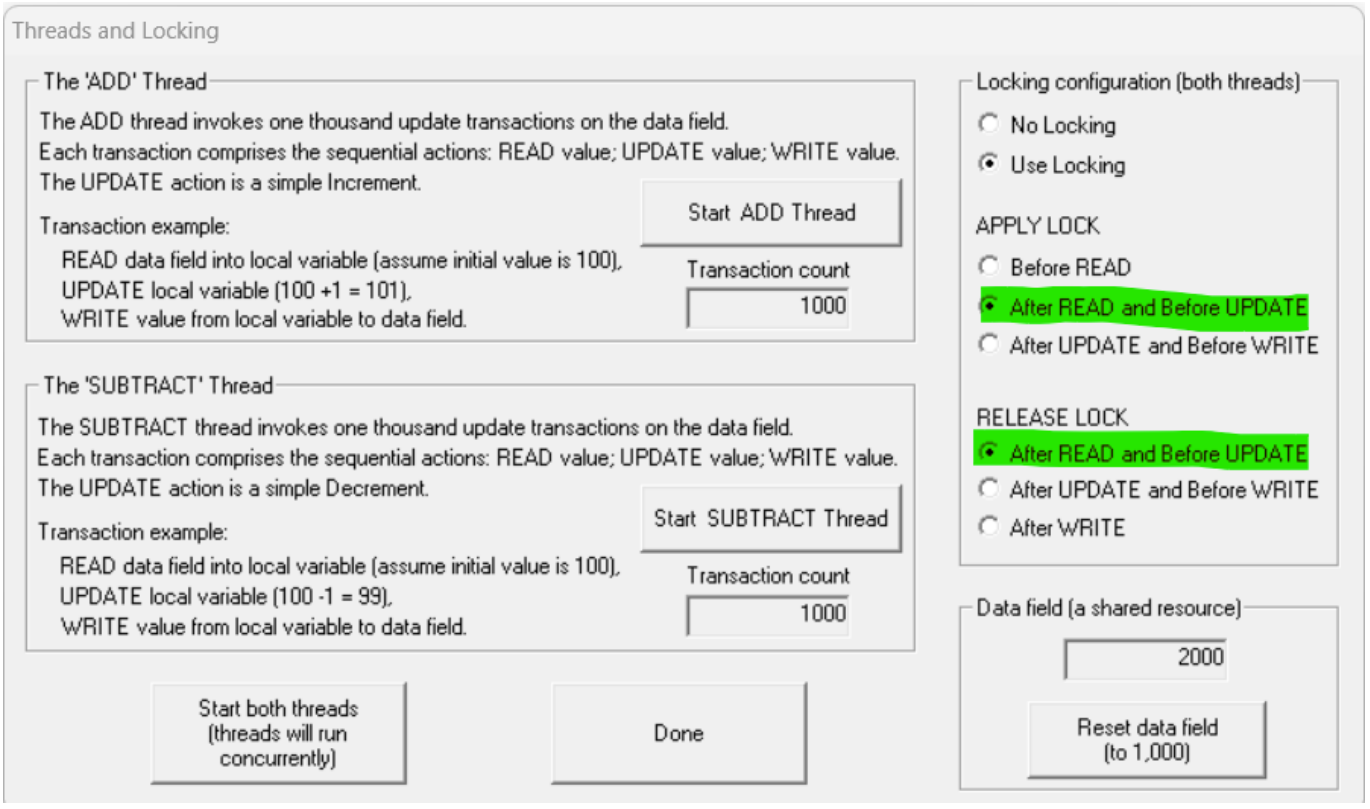


Discrepancy: +5

NGUYỄN MINH ĐỨC - ITITIU21045

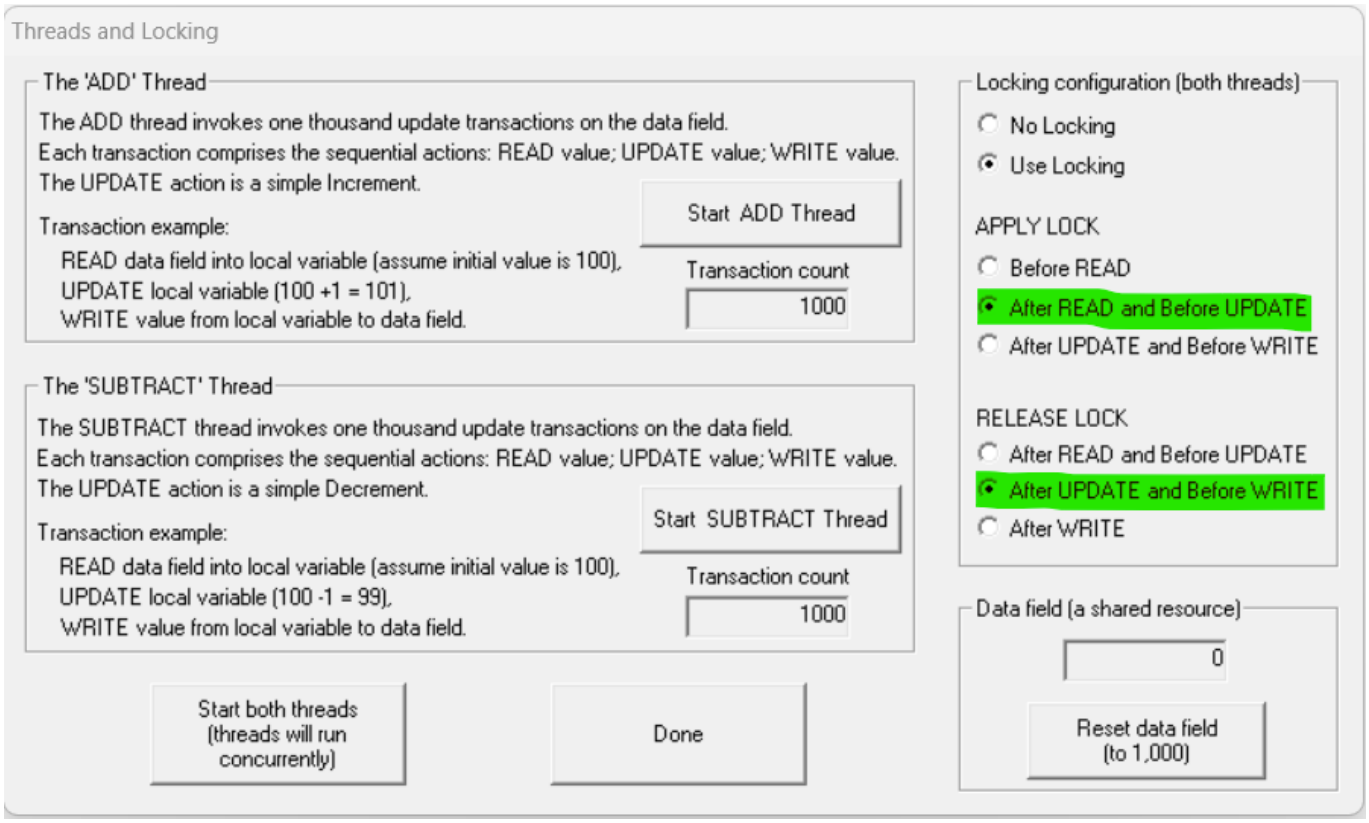


Discrepancy: 0

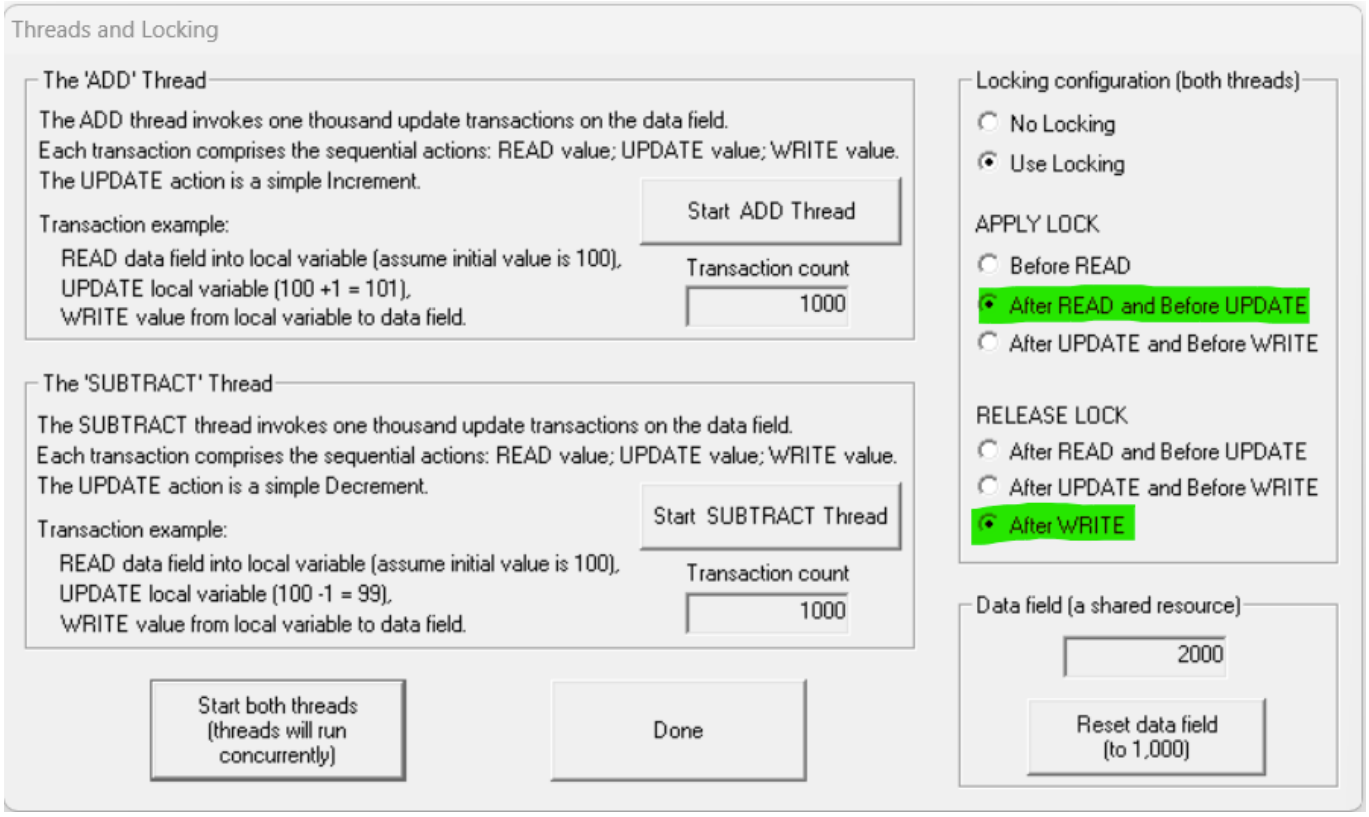


Discrepancy: +1000

NGUYỄN MINH ĐỨC - ITITIU21045

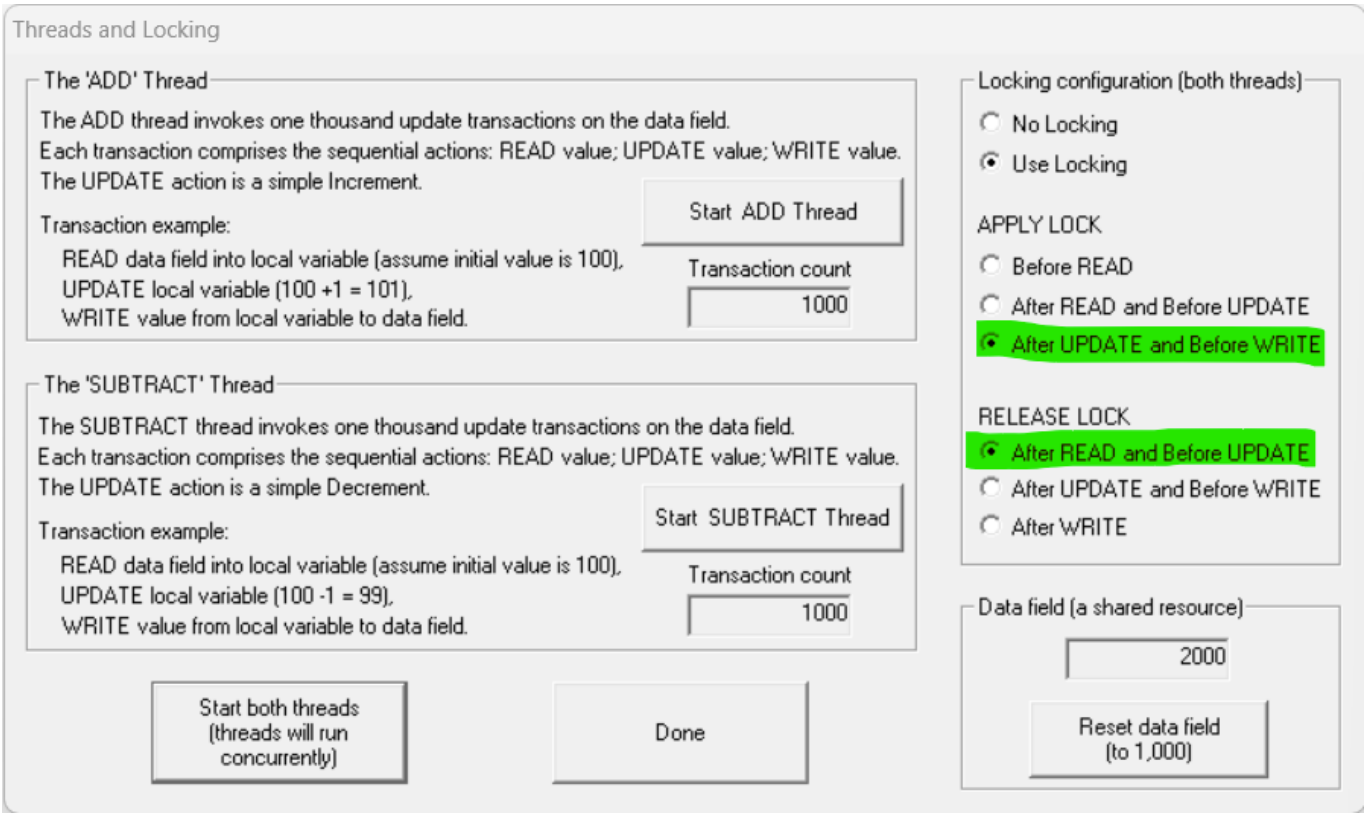


Discrepancy: -1000

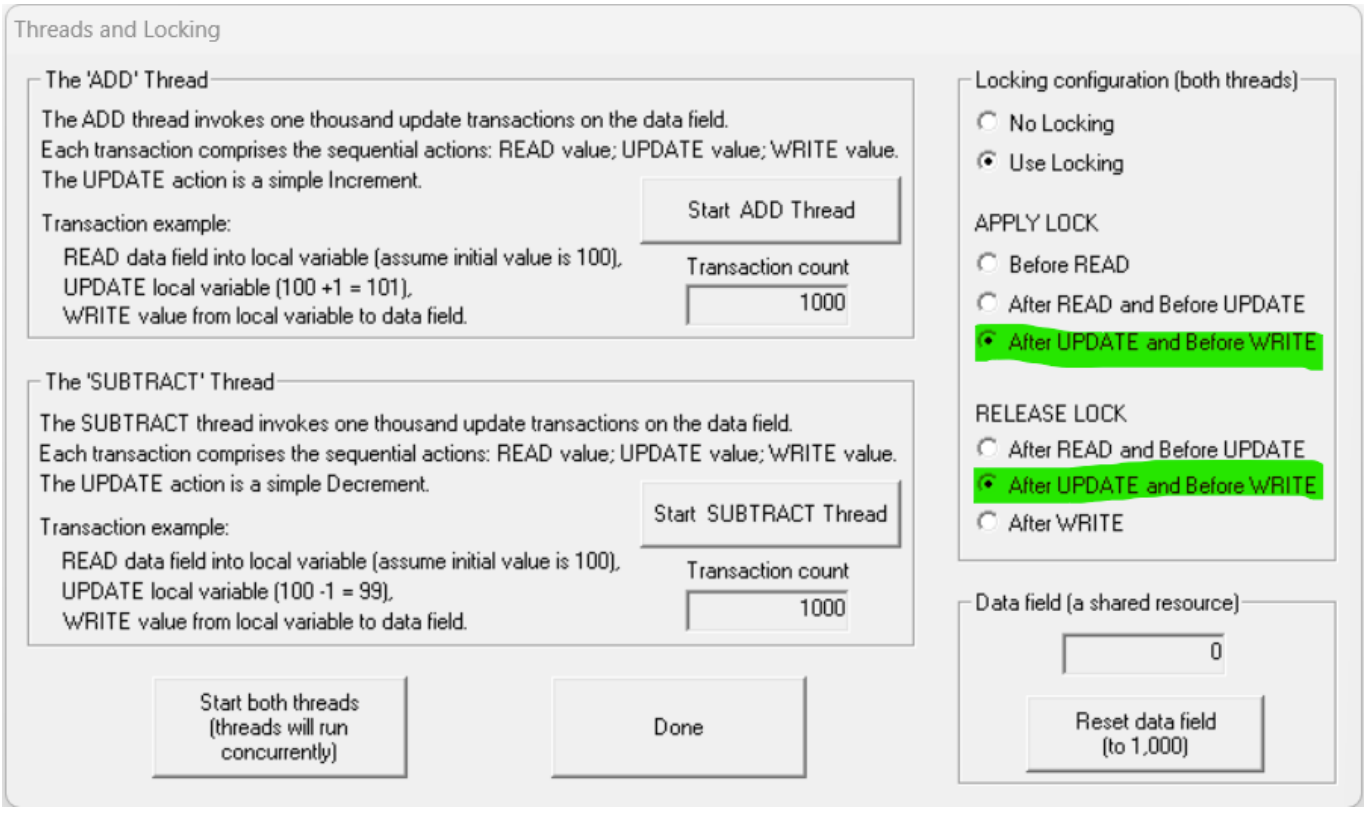


Discrepancy: +1000

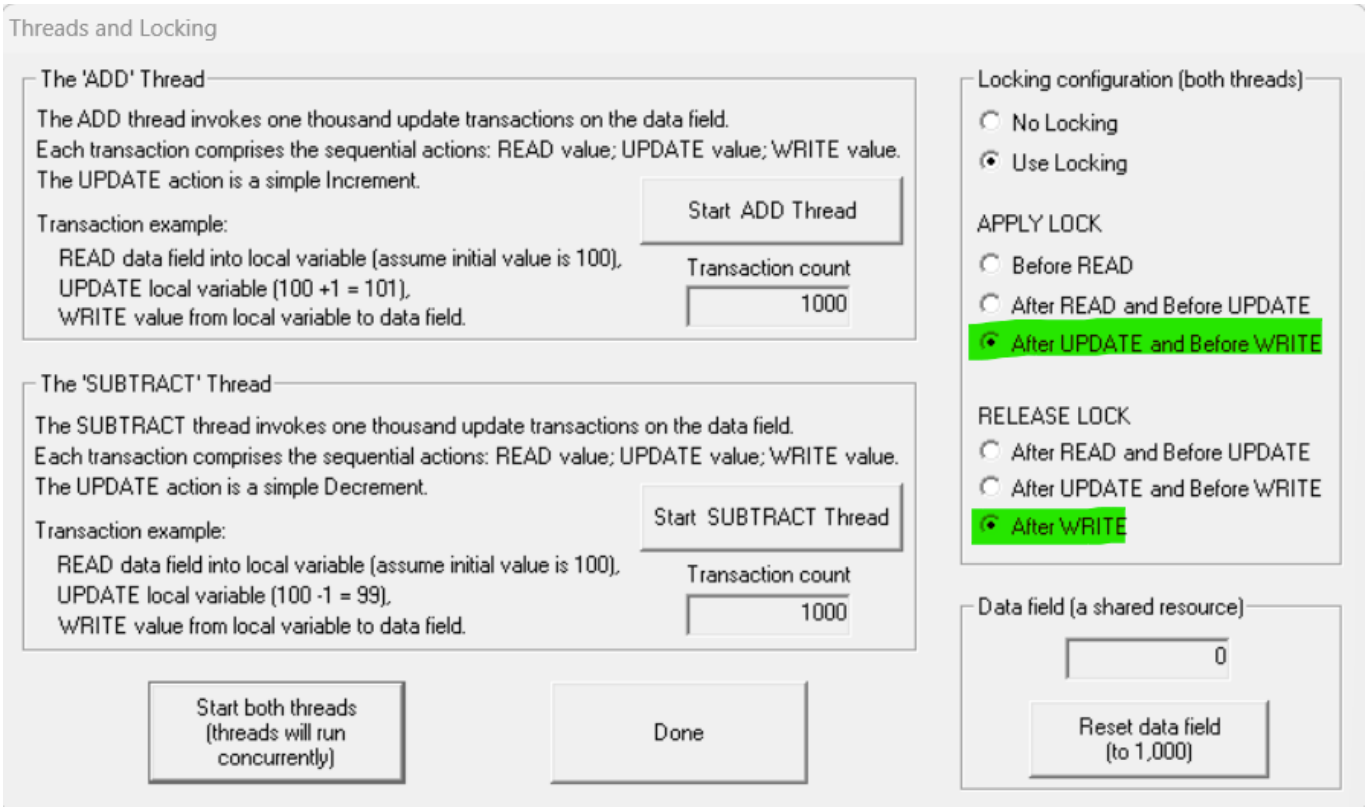
NGUYỄN MINH ĐỨC - ITITIU21045



Discrepancy: +1000



Discrepancy: -1000



Discrepancy: -1000

Q6. Have you found a combination of applying and releasing locks that always ensures that there is NO discrepancy? Are you sure – repeat the emulation with these settings a few times – does it ALWAYS work?

Yes, that is APPLY LOCK BEFORE READ and RELEASE LOCK AFTER WRITE and it always works.

Q7. Could you explain clearly to a friend what the above statement means – try it?

Imagine I have a shared piggy bank (let's call it "counter") that me and my friend both want to put coins into. Now, I want to make sure that when me or my friend puts a coin into the piggy bank, nobody else can interrupt it at the same time.

Now, the "lock" is a magical key that only one person can have at a time. When I have the key, it means I'm in charge of the piggy bank. So, when I'm about to put a coin in, I grab the key (acquire the lock), do my thing (read, increment, and write), and then I give the key back (release the lock).

Here's where the "mutually exclusive access" comes in. While I have the key, my friend can't grab it and vice versa.

So, by using the lock, we make sure that any action (like putting a coin into the piggy bank) is done separately, not at the same time.

Mutual exclusion prevents the lost-update problem.

Q8. Could you explain clearly to a friend what the above statement means – try it?

Imagine me and my friend are both keeping track of the score in a game on a shared piece of paper. The score is initially 0. Now, we both try to update the score simultaneously because something exciting happened in the game.

Without mutual exclusion, it's like we are trying to write the new score on the paper at the same time. Here's where the lost-update problem can occur. When I see the score is 0 and want to add 1 because my team scored a point. At the same time, my friend also sees the score is 0 and wants to add 2 because something else exciting happened on their end. We both write your new scores on the paper at the same time.

Now, instead of the score being 1 (my update) or 2 (my friend's update), we end up with a mix of both, like 1+2=3. The original scores are lost, and we have this weird, incorrect total.

Now, imagine using a "lock" as a rule: only one of us can write on the paper at a time. If I want to update the score, I first grab the lock, update the score, and then give the lock back to my friend.

With this rule (mutual exclusion), we can make sure that only one person can update the score at any given time. This prevents the lost-update problem because there's no chance of your updates getting mixed up