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.

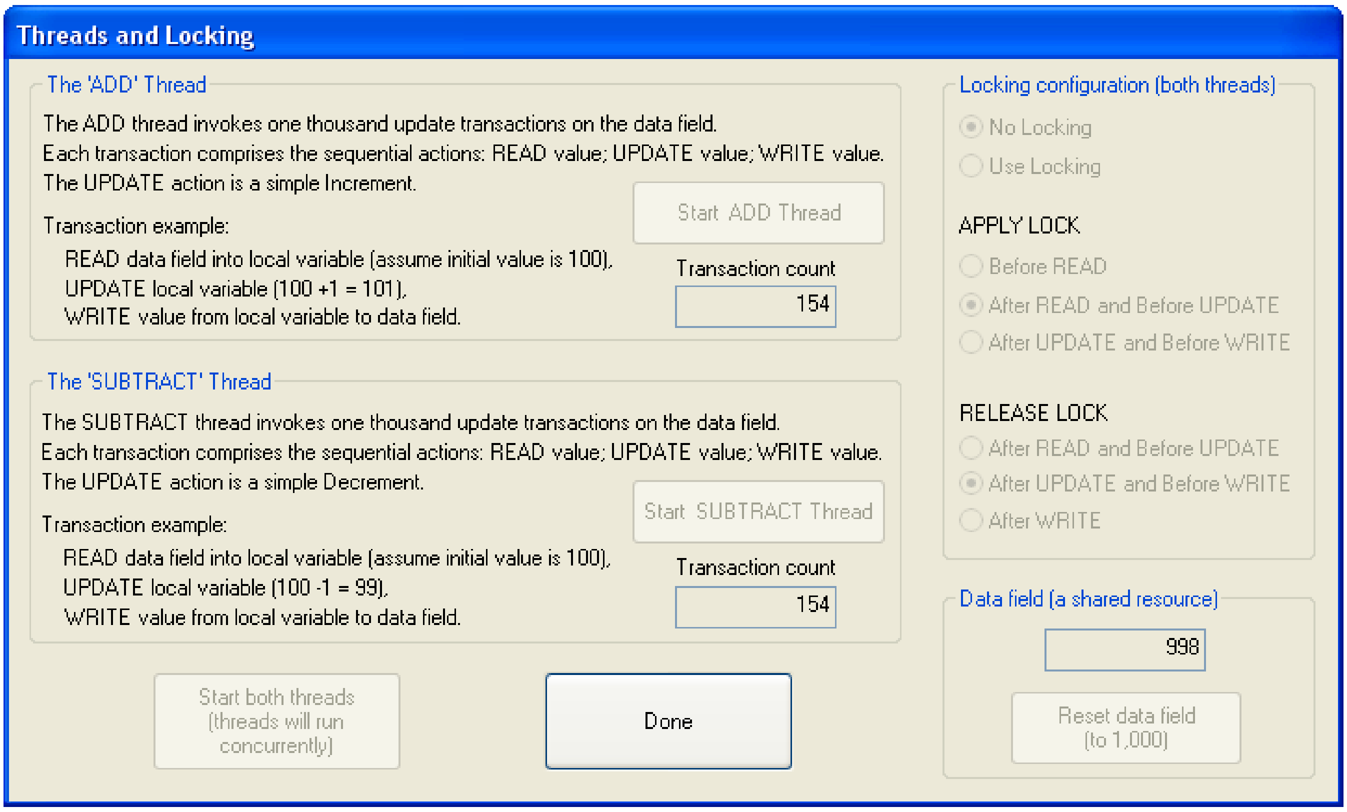
2. Introduction

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 synchronisation 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.


Threads and Locking

The 'ADD' Thread

The ADD thread invokes one thousand update transactions on the data field.
Each transaction comprises the sequential actions: READ value; UPDATE value; WRITE value.
The UPDATE action is a simple Increment.

Transaction example:

READ data field into local variable (assume initial value is 100),
UPDATE local variable (100 +1 = 101),
WRITE value from local variable to data field.

Transaction count: 154

The 'SUBTRACT' Thread

The SUBTRACT thread invokes one thousand update transactions on the data field.
Each transaction comprises the sequential actions: READ value; UPDATE value; WRITE value.
The UPDATE action is a simple Decrement.

Transaction example:

READ data field into local variable (assume initial value is 100),
UPDATE local variable (100 -1 = 99),
WRITE value from local variable to data field.

Transaction count: 154

Locking configuration (both threads)

● No Locking
○ Use Locking

APPLY LOCK

○ Before READ
● After READ and Before UPDATE
○ After UPDATE and Before WRITE

RELEASE LOCK

○ After READ and Before UPDATE
● After UPDATE and Before WRITE
○ After WRITE

Data field (a shared resource): 998

Start ADD Thread
Start SUBTRACT Thread
Start both threads (threads will run concurrently)
Done
Reset data field (to 1,000)

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?

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!).

3. Note the value of the data field now.

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

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!).

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?

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?

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 determine the correct value of the data variable.

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

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?

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

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

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

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.

Q5. Compare the three discrepancies. Is the actual discrepancy predictable, or does it seem 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)?

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

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?

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

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)?

The transactions are a sequence of three stages:

Read the current value into thread-local storage,
Increment (or decrement) the value locally,
Write the new value to the shared variable.

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

Q5. At what point should the lock be released?

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.

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?

The use of the lock forces mutually exclusive access to the variable.

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

Mutual exclusion prevents the lost-update problem.

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