Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Solution 7: Master Solution
Spring Semester 2017

Banking System

See also `solution7.zip`.

Task 1 - Problem identification

The assertion

```
assertThat("Did not loose any money.", bs.totalMoneyInBank(), is(sum));
```

fails because transferMoney is not yet thread-safe. Several threads can execute it simultaneously leading to incorrect amounts of total money in the bank. Consider the following scenario where a thread A calls transferMoney(from, toA), and a thread B calls transferMoney(from, toB).

```
Thread A: Execute from.getBalance() -> 2
Thread B: Execute from.getBalance() -> 2
Thread A: 2 - 1 = 1
Thread B: 2 - 1 = 1
Thread A: from.setBalance(1)
Thread B: from.setBalance(1)
Thread A: toA.setBalance(toA.getBalance() + 1)
Thread B: toB.setBalance(toB.getBalance() + 1)
```

The scenario here shows how we have withdrawn only 1 CHF, but overall we did add 2 CHF to the accounts.

Task 2 - Synchronized

The problem is that the synchronized keyword will lock the whole BankingSystem during a transfer. This means that even with multiple threads, only one thread can perform a transaction at a time, there is no parallelism at all. Even worse, the lock introduces a new bottleneck, all eight threads will always try to acquire the lock, which can be an expensive operation, especially if there are many threads.

Task 3 - Locking

Question 1: What if a transaction happens from and to the same account (i.e., `transferMoney(a, a, X)`;)?

The Java intrinsic locks (i.e., the **`synchronized`** keyword) are reentrant, this means a thread can acquire the same lock multiple times. The same is true for `ReentrantLock` class for explicit locks.

Question 2: Explain what measures you took in order to ensure that your code does not suffer from deadlocks?

To avoid deadlocks, you can introduce an ordering between the lock objects, e.g., always lock the account with the lower account id first.

Task 4 - Summing up

Question 3: Find out what is wrong with the current implementation by describing a scenario that can lead to an incorrect summation of the accounts.

Consider the following case: The sum of three accounts should be calculated: A, B and C, each with an amount of 1000. Assume that there is a concurrent transfer from account A to C with the transfer amount of 100. The correct sum is 3000. In the current implementation it can happen that the `sumAccounts` has already calculated $A + B$, but then the transfer takes place and C now has a balance of 1100, while A has a balance of 900. The broken implementation of `sumAccounts` will not notice that the balance of A has changed, and return to the wrong result of 3100.

If there was an additional transfer with an amount of 100 from B to an external account D (which is not part of the sum), then the correct result would be 3000 (the sum before the transfer from B to D has happened) or 2900 (the sum after the transfer has happened). Both are valid, but they imply a different view on what has happened first.

Question 4: Change the implementation such that it now works for an arbitrary number of accounts.

The solution locks each account before it reads out its balance, but it does not release the accounts until all accounts are summed up. This guarantees that after an account is part of the sum, the balance of the account will not change any more. This means the calculated sum is indeed the sum of all accounts at the time before all the locks are released.

This strategy is called two-phase locking and is used for example in databases to ensure correctness. The idea is that there is a phase were all locks are acquired and no locks are released and there is a phase where all locks are released and no locks are acquired.

Note that you still have to take care of deadlocks. This means that during the locking phase you should take the locks in the same order as you do in the `transferMoney` method, e.g. lock accounts with smaller IDs first.

Question 5: Are there ways to parallelize the summation? If so, describe how you could do it.

Calculating a sum is an associative operation. Therefore, one could split the list of accounts into multiple subsets and calculate the sum of the subsets in parallel. At the end, all subtotals are accumulated to the total sum. However, we need to guarantee that none of the threads will release the lock of an account before the locks of all the other accounts have been acquired by the other threads. Otherwise, a transaction between an account that will be released and an account that has not been acquired by an other thread will result in incorrect total balance. In fact, all the accounts must be locked sequentially in the beginning. Then, we can split the accounts into subsets, and assign the computation of the balance for each subset to a task. Every task will compute the partial balance and release the lock for each account. In the end we can compute the total balance.