CS 61, Fall 2011
Assignment 5: Simulating a Multithreaded Banking System

Assigned: Thursday, November 1

Assignment due: **Thursday, November 17, 11:59PM**

1 Introduction

The purpose of this assignment is to become more familiar with multi-threaded programs and the synchronization challenges that arise in such environments. You have been hired by Globally Synchronized Bank (GSB), whose name conveys their lofty expectations of you, to tackle the tough synchronization problems associated with their ATM system.

2 Getting Started

You may work in a group of up to two people for this assignment. Submission is entirely electronic; any clarifications or revisions to the assignment will be posted on the course Web page.

You can do this homework in groups of one or two students. Please go to the form at

`http://tinyurl.com/CS61-Fa11-bank-groups`

and let us know your group. (This form is also linked to from the CS 61 web page.) *Please fill out the form even if you are working on this assignment by yourself*—simply leave the "Partner 2" fields blank. Please fill in this form by **11:59pm, Sunday November 6**.

In your home directory on your CS 61 VM, you should have a directory called `bank`. Please contact course staff if you do not have this directory. The directory `bank` contains several files. **The only file that you will modify and submit is the file** `bank.c`**.**

Please type your team member names and FAS login IDs in the header comment at the top of `bank.c`.

3 Your Assignment

Overview

Globally Synchronized Bank (GSB) has given you their code base, including the routines that perform the account updates and return the account balances. You will be improving the synchronization of the system, and helping implement functionality for detecting suspicious account activity.

GSB's original system is fully functional, albeit very simple. Access to every bank account is protected by a single global lock. Any thread attempting to modify an account or read the balance out of an account must first acquire this lock, and then release it after doing so.

However, as GSB grew and acquired new clients, they realized their simple system did not scale very well. It became extremely slow for their clients to perform even the simplest transactions, such as reading the balance out of an account that nobody else was even trying to access. So, they turned to you to develop a fine-grained synchronization scheme that will enable their business to scale more effectively while still correctly handling attempts at simultaneous transactions on the same accounts.

The code in `bank.c` already contains calls to GSB's functions that deal with transactions; all they've asked you to do is to add locking around these calls as needed. GSB's functions also include calls to their original system, the *reference implementation*. Comparing the results of your fine-grained synchronization schemes with the reference implementation will help you determine whether your synchronization is correct.

Requirements

- The ONLY file you should modify (and the only file you will submit) is `bank.c`
- You will need to add locking to the following operations.

```
open_account(id, account_number)
close_account(id, account_number)
deposit(id, account_number, amount)
withdraw(id, account_number, amount)
get_balance(id, account_number)
transfer(id, int from, int to, amount)
report_balances(id)
```

- The most basic requirement is of course that your system must enable all calls to be completed correctly. Transactions may fail—e.g., a withdrawal from an account that does not exist should return an error code—but the ultimate behavior of the system must be the same as if the transactions occurred under GSB's original global locking scheme, in which no two transactions are allowed to occur at the same time. As a corollary, your system must execute all transactions eventually (no starvation).

Tasks to Complete

You must implement 4 locking modes, selected at runtime by the `-l` flag (that's a lowercase `L`). The selected locking mode is accessible in `bank.c` in the `locking_type` variable. The locking modes are as follows:

1. **No Locks:** This is the assignment-as-provided. All you need to do is make sure that when `locking_type==NO_LOCKS`, you don't actually lock anything. This will give you a way to compare the broken, but fast, behavior of an unsynchronized implementation with later designs.
2. **One Big Lock:** When `locking_type==BIG_LOCK`, you should synchronize the entire system using one lock. This will simply duplicate the behavior of the reference system, but it should work and be easy to implement. **Do this first.**

3. **Per-Account Locks:** (`locking_type==FINE_GRAINED_LOCKS`) Now you'll add separate locks for each account. This will make it so that unrelated transactions to separate accounts won't interfere with each other. For example, if somebody is attempting to deposit into account 0, it should not interfere with (i.e., block) somebody else from attempting to withdraw from account 1. Similarly, transfers should only interfere with other transactions that involve the accounts participating in the transfer. **Do this second.**

4. **Multi-Reader Locks:** (`locking_type==MULTI_READ_LOCKS`) GSB has noticed that there will often be many simultaneous requests to read the balance of an account (e.g., a large corporate account), but very few requests to actually change that balance. Thus, you will next implement a locking scheme that allows any number of concurrent readers on the same account without having them block each other. For example, if two threads are both attempting to read the balance out of account 0, they should both be able to do so at the same time; if another thread comes along and wishes to change that balance (through a deposit, transfer, or withdrawal), it must wait until all of the reading threads have exited before doing so.

   Your multi-reader locks should be *writer-priority*: this means that if a writer thread is waiting for some readers to finish, new readers shouldn't get to jump ahead of it. This is necessary to prevent a continuous stream of read operations from blocking account updates.

   You **MAY NOT** use any of the `pthread_rwlock` routines (they aren't guaranteed to be writer-priority, in any case). **Do this only after you have per-account locks working.**

Suspicious Activity: An additional service GSB would like to provide is *suspicious transaction notification*: they have clients that wish to be alerted every time a suspicious transaction takes place on their account. GSB handles these notifications with a dedicated thread that runs the function `suspicious_activity_thread`. When the `-u` flag is set and a transfer is suspicious (as defined by the function `transfer_is_suspicious`), the thread handling the transfer should call the function `report_suspicious_transfer`, which stores the suspicious operation in a buffer and then notifies the handler thread. The handler thread should then wake, call `log_suspicious_transfer` on every stored operation, and go back to sleep until the next time it is needed. At least, that's how it should work. GSB has had some trouble getting the synchronization right.

5. You need to modify the functions:

- `report_suspicious_transfer(id, from, to, amount)`
- `suspicious_activity_thread(void* arg)`

   so that, when a suspicious transfer is reported, the operation is safely stored in the buffer, the handler thread is correctly notified, it safely logs all suspicious operations, and then goes back to sleep until needed. **Do this only after you have completed the multi-reader locks.**

There are many ways to implement this scheme, but some are better than others. You will almost certainly need to use a more sophisticated synchronization mechanism than locks, such as semaphores or condition variables. This is an instance of the famous "producer-consumer problem".

Design and Commenting

You **MUST** provide well-structured and commented code, as GSB will be reading it thoroughly! The correctness of multi-threaded code depends on its behavior under all possible interleavings of all the different threads. It is difficult (usually impossible) to verify the correctness of such code with normal testing. Reading and reasoning about the code's behavior is very important; GSB must be able to convince themselves that your code works just by reading it. If your code is difficult to understand, then it will be difficult for GSB to trust it.

4 Testing Your Code

To help test your synchronization, GSB has given you a simulator, and several transaction logs. Once you have written the necessary synchronization code, you will be able to run the transaction logs using your system. Each log will create some accounts and then attempt to perform simultaneous (and sometimes conflicting) operations on those accounts, including withdrawals, deposits, transfers, and balance checks. Crucially, these transactions will be handled by multiple threads (3 by default).

GSB has included calls to their original system, which will be paired with the calls that your improved system executes. The simulator's output will contain two lines of output for each attempted transaction: one indicating the outcome of the transaction under your system, and one indicating the outcome with their original locking scheme (labeled "REFERENCE"). Your solution must agree with all of their original output on all traces in order to be correct (although note that just because your system agrees with theirs on a few individual runs, doesn't mean it's necessarily correct in all cases!).

Running the Simulator

First, compile your code by executing `make all` on the command line. This will produce the simulator, which is an executable file called `bank`, that can take a transaction log, and simulate those transactions using the code you have written. (Look in the file `Makefile` to see other targets you can give to `make`, including targets to perform some testing.)

To run the simulator, execute `bank`, with a transaction log as input, use `./bank < traces/TRACETOTEST.txt`

Run `bank -h` to get a list of command line parameters that you can set. You can set the locking type on the command line using the `-l` flag. Use the `-n` and `-r` flags to set the number of threads and the time between requests. Due to load issues, please don't test your code with more than 20 threads on CS61 virtual machines.

Checking the Output

When `locking_type != NO_LOCKS`, your code should output the same results as the reference solution. Note that the exact order of the messages may not be the same when multiple threads are involved. If your locking is correct, however, the only reorderings should be between operations where the order doesn't matter to the result. For example:

```
REFERENCE:  <193> transfer 2 1 318 -- 628
GSB:        [181] GET_BALANCE 0 -- 1230
REFERENCE:  <201> withdraw 0 327 -- 903
GSB:        [201] WITHDRAW 0 327 -- 903
GSB:        [193] TRANSFER 2 1 318 -- 628
```

Each operation prints its result; operations with the same ID ([ID] for your code, <ID> for the reference) should always have the same result.

At the end of any trace, there is also a final balance check that will detect if any accounts have the wrong open/closed status or the wrong balance. Here's an example from a correct run:

```
==== Final check: =======
Final balance for account 0: 903
Final balance for account 1: 803
Final balance for account 2: 946
```

Your locking scheme should be correct for any number of threads, and any request rate.

You can use the provided `valid.sh` script to help you check whether your code agrees with the reference implementation. See the `Makefile` for an example of how to run `valid.sh`. The script calls the unix command `diff` to see if there is a difference between the output of your solution and the reference solution, so if your solution is correct you should just see something like the following:

```
Running 'diff trace1.log.ref trace1.log.gsb'...
Done
```

5 Advice

- Plan out your locking scheme in advance. Decide on a locking protocol, write it down, and if you're working in a pair, make sure you and your partner are both on the same page about it. Then you can start writing code. There is no formal design document due for this assignment, but that does **not** mean that design is unimportant! You will save yourself a lot of hassle if you are systematic about your approach rather than haphazardly putting in locks.
- Review the lecture notes and the textbook chapter on Concurrent Programming (Chapter 12 in the 2nd edition, Chapter 13 in the 1st edition). Be sure you understand how to avoid deadlock in situations where multiple threads all need to acquire multiple locks simultaneously.
- Comment, comment, comment. Not only will it be critical for understanding the complex interactions between your locks, but you will be graded partially on the clarity of your comments. This does not mean that you have to comment every line, but rather that you should add a brief and explanatory sentence or two next to any potentially confusing code.
- Use helper functions to factor out your locking code. Try to keep the number of places that have to worry about the different `locking_types` to a minimum. To give you a sense of scale, our reference

implementation has about 150 lines of helper functions (including whitespace and comments), and only a couple of extra lines in each of the actual operations, which just call the helpers.

- Check the return value of every single system call. Yes, we mean it: every single one. Use wrapper functions to make this easier.
- To check that your fine-grained locks and multi-reader locks are working, you should look at performance, as reported by the simulator. When running with multiple threads, fine-grained locking should result in much better performance than a single big lock. On traces with many reads and only a few writes, multi-reader locks should do even better.
- Feel free to write more traces to test different aspects of your implementation.
- Don't forget about gdb.

6 Evaluation

Your score will be based primarily on correctness, but will also take into account the sensibility of your design and the clarity of your code and comments.

70 points: Correctness Your solution must pass all the tests that are run against it. We **WILL** be running your solution against traces that we do not provide you! They will be similar in style, but different in their particulars. Thus, you must satisfy yourself that your solution is correct not only through testing, but also through reading and reasoning about your code.

We will be reading and reasoning about your code, as well. A solution that appears correct in testing may nevertheless contain deadlocks, race conditions, and other nondeterministic bugs. A portion of your correctness score will be based on a thorough reading of your code; we must be able to convince ourselves via reading and reasoning that your code is correct.

30 points: Design and Comments Because the correctness of your code depends on a thorough reading, your code must be clear, concise, and commented. We should be able to easily understand what you've done. If you find yourself adding tons of code to every function, something is wrong.

7 Submission Instructions

To submit your assignment, execute the command **make submit** in your `bank` directory. This will send a copy of the file `bank.c` to the CS 61 staff. Remember, the only source file you should modify is the file `bank.c`.

You may submit multiple times; however, only the most recent submission is graded. Only one submission per group is required (i.e., both members should not submit).

When testing your assignment, make sure to use a CS 61 VM. This will ensure that the output you get is representative of what will be graded.