Overview

You have been assigned the task of retrofitting the bank's single-threaded software to support multiple threads running in parallel. The hope is that this will help the bank's software handle more transactions per second to keep up with demand.

Purpose

The purpose of this assignment is to give you hands-on experience with synchronizing multiple threads operating on shared data. In particular, you'll get to debug race conditions and deadlocks and try to fix them while still maintaining a reasonable amount of concurrency in the program.

The assignment will also introduce you to the Posix threading package and its synchronization routines. Finally, we hope you'll learn to think about the tradeoffs of different locking disciplines in terms of complexity and concurrency.

Building

cd to code directory and type make to build.

Running

Inside the code directory, run:

./bankdriver -w4 -t1
to create 4 workers and run test #1. (There are 7 different tests. You'll want to make sure that each one works with multiple workers.)

Add -r to perform fewer operations. You'll probably find this flag to be useful when running your code inside Helgrind.

See the nondeterminism section for details.

Run

./bankdriver -h

to get more information on the arguments.

The Existing Bank Software

We've given you the source code for the single-threaded version of the bank's software; your job is to modify it to run safely in a multithreaded environment while maintaining an acceptable level of concurrency between threads.

The bank itself is subdivided into branches. Each branch has some number of accounts, and each account belongs to exactly one branch.

Tellers process requests against the bank. A request can be:

- a deposit or withdrawal from an account,
- a transfer between two accounts,
- a query for a branch's balance, or
- a query for the bank's balance.

Transfers can cross branch boundaries; that is, we can request a transfer between any two accounts A and B, even if A and B belong to different branches.

The bank also contains a reporting system that performs the government-mandated tracking of transfers above a specified size and generates nightly reports.

Assignment Details

The code we give you divides the workload of the bank among one or more worker threads, which execute their commands concurrently.

Since the program was originally intended to run on a single thread, the authors did not consider the implications of multiple threads executing the code. For example, since the code doesn't make use of locks or other synchronization primitives, you're likely to observe errors due to race conditions when you run using multiple threads. Your assignment is to add synchronization to eliminate the race conditions while allowing worker threads to process commands concurrently.

Another fallout of multiple threads executing the code concurrently is incorrect behavior of the reporting functionality. The intended behavior is to have a single report generated every night. However, when multiple threads attempt to generate the report, these semantics break as each thread tries to generate its own nightly report.

You may make any modifications you want to both the modules and the interfaces to these files. For example, you are welcome to add interface routines to the modules and data items (e.g. locks) to the data structures. You may add entire modules and files if you deem it necessary.

The test harness uses the Posix threads package (pthreads) to spawn and control the worker threads so you should use the pthread synchronization primitives. See the appendix for a description of routines you may use.

To summarize, the lack of synchronization in the starter code causes two classes of problems:

- First, two workers may read or write shared state concurrently while processing commands. These race conditions may result in incorrect account balances or

errors elsewhere in the bank's data. You'll need to add locks to prevent these bad interleavings.

- Second, the nightly report module doesn't work properly in conjunction with multiple worker threads. You'll need to add code to ensure that all workers finish the day's work before the bank begins to generate that day's report and that no worker begins the next day's work until the report is fully calculated. You'll also need to ensure that threads that finish the day's work early wait for the other threads and don't start on the next day's work until the report is done. All worker threads call Report_DoReport() when they finish the day's work; this function may be a good place to coordinate between threads to ensure that they all begin the next day's work at the appropriate time.

Concurrency requirements

Given that the bank code works when run with a single thread of control, it would be trivial to eliminate all bad interleavings when running with multiple threads by *serializing* the threads—that is, by allowing only one thread to execute at a time.

Since the entire point of using multiple worker threads is to improve performance, your solution must support concurrent execution of commands. In particular, a correct solution will have the following properties:

Branches operate independently from one another.
Worker threads should be able to operate concurrently on accounts in different branches. Similarly, the managers of two different branches should be able to concurrently compute their branches' balances.

Transfers between different accounts in the same branch can proceed concurrently.
If accounts *A*, *B*, *C*, and *D* are all in the same branch, one worker thread should be able to run a transfer from *A* to *B* concurrently with another worker thread transferring from *C* to *D*.

Note: It's entirely possible that your multithreaded bank code won't run any faster than the single-threaded code. This is fine, so long as you meet the above requirements.

Other requirements

Besides supporting concurrent execution, your solution must execute the commands correctly!

Aside from ensuring that deposits, withdrawals, and transfers work properly, you also need to ensure users can never observe the result of a partially-completed transaction. Your solution will need to use locks to ensure that commands are executed *atomically*.

Testing

Our test harness works by pseudorandomly generating a series of requests to the bank and running your code in multithread mode. It then generates the same series of requests and runs your code with one thread. The test passes if the single-threaded output matches the multithreaded output.

We provide you with seven different testcases (-t1 through -t7). Make sure you run each one, as they each test different things!

Dealing with nondeterminism: The -y flag

Testing a concurrent program is very hard, since bugs may only appear when threads interleave in a specific way, and thread interleavings are nondeterministic—that is, they may (and probably will) be different the next time you run your program.

On a machine with just one CPU (such as our VM), threads will interleave only when the operating system decides to stop executing one thread and start executing another. This may happen every few milliseconds. In contrast, on a machine with multiple CPUs (such as corn or myth), two or more threads will actually run at the same time, generating many interleavings.

To help generate a richer collection of interleavings, we have sprinkled throughout the starter code calls to a special Y; function. (Well, it's actually a macro. But close enough.) This function flips a coin and, with some probability, instructs the operating system's scheduler to stop executing the current thread and start executing another thread, if some other thread is ready to run.

If you run bankdriver with the default flags, these Y; calls won't do anything. To turn them on, you need to pass the -y flag, as in

```
./bankdriver -t1 -w2 -y
```

Because the call to yield to another thread is slow, the default behavior is to yield execution on only 5% of calls to Y;. You can change this by passing a numeric parameter to -y. For instance, -y50 will yield on half of the calls to Y;.

We recommend that you run with -y if you're testing on a VM or on a heavily-loaded cluster machine, since without it, the tests may not find many errors in your code.

Concurrency requirements

Unfortunately, our automated tests can't tell whether your code meets our concurrency requirements. You'll have to check for this by inspecting your code.

To help guide you in judging the concurrency of your solution, here are two thought-experiments you can perform against your implementation to check that it meets our requirements.

Account actions

Suppose we have five distinct accounts, *A*, *B*, *C*, *D*, and *X*.

A, *B*, *C*, and *D* are in branch 1, and *X* is in branch 2.

In order for two commands to proceed concurrently, they cannot run through the same critical section. This means that two commands do not run concurrently if they acquire the same lock.

Because we require that branches can operate independently from one another, a worker depositing into *A* may not acquire any of same the locks that a worker depositing into *X* acquires.

Since we require that transfers within a branch proceed concurrently and since *A*, *B*, *C*, and *D* are all in the same branch, a worker executing a transfer from *A* to *B* must not acquire any of the locks that a worker transferring from *C* to *D* acquires. However, a transfer from *A* to *B* may share locks with a transfer from *A* to *C*.

Branch and bank balances

Although we have a test for bank balance (-t7), we don't have one for branch balance. You'll want to inspect your code to make sure both these functions are correct.

In particular, the value returned by a balance command must not reflect a balance which exists only during the execution of some other command. Put another way, a returned balance must always correspond to a balance which might have existed *between* two commands if we'd executed them in some order on a single thread.

For instance, suppose our bank's balance starts out at $100. We transfer $5 from account *A* to *X* and finally deposit $10 into account *B*.

In this case, bank balance might correctly return $100 or $110. But $95 or $105 would not be correct, as those values reflect the $5 as it's in flight between *A* and *X*.

Hints and Suggestions for the Assignment

Getting started

Sometimes it is possible to add synchronization to an existing program by simply looking at individual routines or modules and adding some locks to make some multistep operations atomic. Frequently this is not the case. Unfortunately, this particular assignment falls into the camp of requiring a more global understanding of the functions

being performed and coming up with a locking discipline.

For starters, be sure you understand what code needs to support concurrent threads. The worker threads only call into a subset of the functions in the files you will modify. These include the functions in report.c, teller.c as well as the bank and branch balance functions. Other routines such as the module initialization functions will only be called from a single thread.

For routines that are called by concurrent threads, make sure you understand what shared data is being accessed by these functions. It's not enough to understand just the data internal to the module. You need to understand all the shared data access regardless of what module is actually touching it. We suggest to trace out the calls and identify what is read and written for each of the different commands. From this information you can determine what can and what shouldn't be executed concurrently. Aside from correctly executing the code, you will need this information to answer the discussion questions.

Deadlock is a real possibility in a system like this. Be sure you have a plan to avoid deadlocks. You will need this plan for answering the discussion questions as well.

Since a random number generator generates the workload of commands, you may encounter corner cases that would be unlikely in a real bank. These include 0 amount operations and transfers from an account to itself. You need to handle this.

The report functions need different handling than the bank account manipulation code. Report_DoReport() is a scheduling problem. Worker threads can call the function at different times and the workers must wait until all have called before doing the report work. Similarly, Report_DoReport() should not allow a worker thread to continue until the report is completed. Since Report_Transfer() is called on every transaction any solution that requires acquiring a single lock can not meet the concurrency requirements. You will need to change the implementation of the Report module to get acceptable concurrency.

Unlike serial programs, parallel programs can behave differently depending on the load on the system. Be aware of the machine you are on and what else is going on at the same time. Commands like top and uptime give you some insight to how much contention your run is likely to receive.

The testing infrastructure ensures that each worker thread calls Report_DoReport() the same number of times for every test run. Each call to Report_DoReport() signifies the end of the day for that thread.

Although the test driver reports performance speed ups, the shared machines we used for the assignment will cause your performance to be all over the map. Most of the test cases we provide are designed to be high contention to help you find race conditions in your code. It's unlikely you are going to get a lot of speed up on these even with the best locking possible. In summary, don't worry about performance but do worry about races and the assignment concurrency requirements. We plan to use a combination of running

tests, code inspection, and your answers to Discussion question 1 to evaluate your submissions.

Debugging

DPRINTF

We provide you a a command-line-controllable printf for debugging. If you write

DPRINTF('n', ("num is %d", num));

then when you run bankdriver with the -d n flag, the program will print out the value of num.

To enable multiple DPRINTF flags, just pass them together on the command line. For instance, you could run

./bankdriver -d xyz -t1 -w4

GDB

GDB is an excellent tool, particularly for diagnosing deadlocks. See our FAQ for more information on using GDB with this assignment.

Helgrind

Helgrind is a Valgrind module which tries to detect race conditions in your code. Because running a program under Helgrind is much slower than running natively, you'll want to pass the -r flag when you run bankdriver under Helgrind. This flag reduces the number of transactions the bank driver executes.

Here's a sample command-line:

valgrind --tool=helgrind ./bankdriver -r -t1 -w4

Note that Helgrind doesn't work on myth. It should work on corn and within the VM.

See the Valgrind manual for more information on Helgrind.

Extra Credit

The extra credit part of this assignment is to code a solution using non-blocking synchronization for the transactions. Mutual exclusion (e.g. locks) should not be used during normal transactions but you can use locks and conditional variables for implementing the Report_DoReport() functionality. For the extra credit part of the

assignment your solution does not need to handle any requests that span multiple branches including bank balance requests and transfers between accounts in different branches. The solution can be optimized for test cases with many branches with smaller numbers of accounts. Test cases 5 and 6 generate workloads with no multiple branch transactions.

If you do the extra credit you need to build it as a separate program in a subdirectory of your code directory called nolocks. Note your submission should include both the original program using locks as well as this one. The submit script on the original program directory will submit the nolocks subdirectory as well.

Note this extra credit is challenging due to both the existing code not being structured to support it and modern machines and programming environments having minimal support for the atomic operation operations necessary for the non-blocking algorithms. The extra credit section will likely require more invasive modifications to the starter code than did the first part of the assignment.

The compilation environment we use (gcc) has support for accessing the atomic instructions of the x86 architecture using some built-in routines. The routine __sync_bool_compare_and_swap() provides the basic compare and swap functionality on word-sized memory locations. Documentation on these routines can be found at the following URL: http://gcc.gnu.org /onlinedocs/gcc/Atomic-Builtins.html.

There's actually quite a bit more to this extra credit than meets the eye. For instance, you can't just replace all the account and branch balance update commands with atomic operations. You need to make sure that when you deposit into an account, the branch's balance is updated atomically with the deposit. We shouldn't be able to observe either new balance until both have been updated.

A similar problem occurs when we transfer between accounts. Suppose I have accounts *A* and *B*, each starting with $100. I transfer $10 from *A* to *B*. If I observe that the balance of *A* is $90, then I must observe that the balance of *B* is $110. It would be incorrect to see that the balance of *B* is $100, since I've already observed the transfer out of *A*.

You'll have to think very carefully about your implementation in order to avoid these problems.

Discussion Questions

Please answer the following questions in a plain-text file named discussion.txt in your submit directory.

1. Answer the following questions about the system you built:

a. Compare the concurrency supported by your system with the system we

provided you without locks. Ignore the fact that the original system had race conditions that resulted in incorrect executions. Include a description of what commands or parts of commands are able to proceed concurrently in your system. Also discuss what can no longer be executed concurrently with your synchronization in place.

b. Describe your approach to eliminating deadlock possibilities from your system.

2. Assume you are told your system needs to support two orders of magnitude more accounts and tellers but the same number of branches. What changes would you recommend to the bank software structure to support the additional concurrency to support address additional workload? Hint: consider what functionality reduction you could make to greatly increase the concurrency.

3. Assume of the following bank scenario where the account balance and the amounts are 64bit unsigned integers (i.e. unsigned long long with our compilers). Assume you are running on a 64bit machine. The withdrawal function looks like:

```
int Withdrawal(Account *account, unsigned long long amount){
    if (account->balance < amount) {
      return ERROR_INSUFFICIENT_FUNDS;
    }
    account ->balance -= amount;
    return ERROR_SUCCESS;
  }
```

a. Shortly after releasing the multithreaded version of the software, you read in the newspaper that an unemployed banker whose checking account was just about out of money discovers that he is now the world's richest person. Describe the race condition that could cause this to happen. Provide an explicit timing flow that describes the interleaving necessary for the race condition to appear.

b. Would there be any difference in the answer to this question if the code was running on a 32bit machine?

4. Did you complete any of the extra credit for this assignment? If so, please *briefly* describe your work.

Appendix

These are examples of pthread routines you might find useful for this assignment. You can look up more details via the man pages (see man pthread_mutex_lock, for instance) or at this page (particularly sections 6 and 7).

Our one restriction in this assignment is you may not use the pthread barrier routines

(e.g. pthread_barrer_wait()).

```
#include <pthread.h> /* Be sure to include in any file that you add locks */


pthread_mutex_t lock;    /* Declare a lock name "lock"*/

pthread_mutex_init(&lock, NULL); /* Initialize locks before first use */

pthread_mutex_lock (&lock); /* Acquire a lock */
pthread_mutex_unlock (&lock); /* Release a lock */


pthread_cond_t cond; /* Declare a condition variable */

pthread_cond_init(&cond, NULL); /* Initialize the condition var */

pthread_cond_wait(&cond, &lock); /* Wait on cond, release lock. */
pthread_cond_signal(&cond); /* Wake up one waiter on cond. */
pthread_cond_broadcast(&cond); /* Wake up all the waiters on cond. */
```