# Homework 4

As part of this homework, you will practice your understanding of concurrency. You will implement a ledger that maintains information about the account and balance of each account. We will explore how to use a singly linked list to store the ledger and different approaches to synchronize access to the ledger. Each solution must implement the following interface:

int create\_account(unsigned int account, void \*ledger);

int modify\_balance(unsigned int account, unsigned int amount, void \*ledger);  
void list\_accounts(void \*ledger);

* The create\_account is used to create new accounts to the ledger. If the account already exists, then the method returns -1.
* The modify\_balance is used to add the amount from the balance of the specified account. Note that subtractions from the balance can be implementing by passing a negative amount. The method returns the updated balance or -1 in the case when the account does not exist.
* The list\_accounts prints the account and the balance in that account.

All functions have access to the ledger via the ledger pointer. Note that since we want to support multiple implementations, we use a void \* data type.

***Step 1 (bank.c):*** As the first step, we will implement the ledger as a single linked list. The linked list and ledger will be defined as follows:

typedef struct account {

unsigned int account;

int balance;

struct account \*next;

} account\_t;

typedef struct ledger {

account\_t \*head;

account\_t \*tail;

} ledger\_t;

You will have to implement the create\_account, modify\_balance, and list\_accounts methods using the above defined data structures.

***Step 2 (bank\_global\_lock.c)***: The implementation that you have so far does not support concurrent access i.e., if multiple threads access the same bank account the balance may be incorrect. You have seen examples of how this may occur in class. You will have to use the pthreads to implement multiple clients accessing the bank and handle concurrency. The simplest approach to ensure concurrency is to implement a global lock that lets a single thread at a time modify the linked list storing the ledger (i.e., all accounts are locked when a single thread wants to update a balance). To this end, you will update the ledger\_t data type to include a lock as follows:

typedef struct ledger {

account\_t \*head;

account\_t \*tail;

pthread\_mutex\_t lock;

} ledger\_t;

You will have to update the create\_account, modify\_balance, and list\_accounts to use the global lock appropriately. You should check out pthread\_mutex\_init, pthread\_mutex\_lock, and pthread\_mutex\_unlock. To verify that your implementation is correct, you’ll have to use the pthreads library to create multiple clients each one using a different thread.

***Step 3 (bank\_account\_lock.c):*** A limitation of the above data structure is that at most one thread can modify it. We want to remove this limitation. Towards this goal, we will make two changes:

* First, we will transform the global lock into a read-write lock (rwlock) (<https://pubs.opengroup.org/onlinepubs/9699919799/>). A rwlock has different semantics than a regular lock to allow multiple reader threads or a single writer thread enter a critical section. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. The rwlock will allow us to have multiple threads iterate through the list (i.e., reading threads) and a single thread adding elements to the list (i.e., the write thread).
* Second, to protect the balance of each account we will add a pthread lock to each account. The redefined account\_t is:

typedef struct account {

unsigned int account;

int balance;

struct account \*next;

pthread\_mutex\_t lock;

} account\_t;

Update the implementation of create\_account, modify\_balance, and list\_accounts to correctly handle the rwlock and the mutexes of each account.

Step 4: Now that we have implementing the two versions of providing concurrency, let’s analyze their performance. I have provided benchmark.c and benchmark.h to help you benchmark your applications. The benchmark\_driver controls the experiments that you will run. At a high-level an experiment works as follows:

* An experiment involves multiple threads with the number of threads being specified in the num\_threads argument supplied to the function.
* Each thread attempts to create an account between 0 … num\_accounts – 1.
* Next, each thread runs several transactions. A transaction involves pick at random one of the previously created accounts and modifying its balance by a random value. The number of transactions each thread creates is equal to num\_transactions arguments
* The function returns the number of seconds required by all threads to complete.

Modify the main function to run the benchmark with the following parameters for each of the two programs.

For bank\_global\_lock:

|  |  |  |  |
| --- | --- | --- | --- |
| num\_threads[down] /  num\_accouts[right] | 1 | 10 | 100 |
| 1 |  |  |  |
| 10 |  |  |  |
| 100 |  |  |  |
| 1000 |  |  |  |

For bank\_account\_lock:

|  |  |  |  |
| --- | --- | --- | --- |
| num\_threads[down] /  num\_accouts[right] | 1 | 10 | 100 |
| 1 |  |  |  |
| 10 |  |  |  |
| 100 |  |  |  |
| 1000 |  |  |  |

Using the above tables answer the following questions:

* Is there one implementation that works better than the other one under all settings?
* If this is the case, why is this implementation having better performance?
* If this is not the case, then under what settings is one implementation performing better than the other one and why?
* For bank\_global\_lock and bank\_account\_lock, plot the run-time of the functions (y-axis) when the number of accounts is 1 and the number of threads is 1, 10, 100, and 1000 (x-axis).
  + Fit a line through the points. What is the slope of the line?
  + How can we interpret the slope of each line?
  + What is the highest value of the slope that we might expect?
  + What does it say that when there is a single thread bank\_global\_lock has shorter runtime than bank\_account\_lock?
* For bank\_global\_lock and bank\_account\_lock, plot the run-time of the functions (y-axis) when the number of accounts is 100 and the number of threads is 1, 10, 100, and 1000 (x-axis).
  + Fit a line through the points. What is its slope?
  + Discuss how the threads changed from the previous case of bank\_global\_lock.

Please turn in the following:

1. Modify the files provided in handout.zip to implement each of the steps described above
2. Provide a word document that includes the performance tables for bank\_global\_lock and bank\_account\_lock, the graphs, and answers to the above questions.