Brian Reber
CprE308
Project 02
Lab Section D

In this lab, I have learned a lot about mutexes, and how they can be used to provide thread safety. I have also learned how they can lead to deadlocks in programs if implemented improperly. In addition to the major topic of mutexes, I learned about how to write moderately complex multi-threaded programs in C, and then some C-specific techniques.

When I first implemented the project, I locked a mutex for each account, in the order the account numbers were provided in the input. This led to some problems with the program appearing to hang. Some of the input transactions used the same accounts, but specified them in a different order, which would cause one thread to obtain a lock on the earlier accounts, while allowing another thread to obtain the locks on some of the accounts that the first thread needed. This caused a deadlock to occur. To fix this, I implemented a sorting function that would sort the accounts by their account number.

---

See the my submission tar for the output of running the test script with 10 1000 0.

---

**Coarse Grained vs Fine Grained Locking**

These are the timing results from running the regular (fine grained locking) and the coarse grained locking. From running my two programs, it is clear that the fine-grained lock is significantly quicker than the coarse grained locking for this case. I ran the test multiple times, and the results were very consistent.

```
time ./testscript.pl ./appserver 10 1000 0
3m39.400s
3m36.322s
```

```
time ./testscript.pl ./appserver-coarse 10 1000 0
8m31.620s
8m32.293s
```

The fine grained method was faster because it allowed for more concurrent transactions. If there are two transactions with completely different accounts, there is no reason they can't be performed concurrently. With fine grained locking, we are able to perform multiple transactions at the exact same time. With coarse grained locking, this isn't the case. We lock the entire bank when we want to perform a transaction, preventing other transactions from happening until we are finished. This means that even though there may be two unrelated transactions that want to be performed, only one can be done at a time since we locked the entire bank.

The coarse grained locking will be better if there are a lot of overlapping transactions, each with a lot of accounts. This would mean that instead of having to lock each account individually, we would just have to perform one lock. Each lock operation isn't necessarily cheap, so by reducing the number of locks we have to perform, we can improve performance.

The performance would likely improve if we had a lock for every 10 accounts.  This would be the case if we had a lot of transactions with accounts that are close to eachother.  Then instead of having to lock multiple sequential accounts, we wouldn't only need to lock once for every 10 accounts.  This could also degrade performance if we have multiple transactions that could happen concurrently because they don't use the same accounts, but aren't able to because another transaction has a lock on the block of accounts in which we need to lock.

I think the optimal granularity kind of depends on the type of application.  If you know you have a lot of transactions with conflicting accounts, and these are large transactions, it might be better to have coarse.  But if you just have a lot of transactions with few accounts in each, the fine grained would probably be better.