Name: Mubassir Serneabat Sudipto
Section: 01
University ID: 814828068

Programming Project 2 Report

Summary (10pts):

In the project, I gained significant insights. To begin with, I decided to implement pthread's feature of read-write locks instead of mutex locks. I believed this would be more effective, primarily since operations on bank accounts are distinctively split between reading and writing, and it seemed easier to apply. During the project, I encountered a deadlock situation where my threads were stuck because several transactions (TRANS) were waiting for other transactions' locks to be released. As I'm detailing this, it's clear that a trylock for the locks is a more suitable solution than the one I had initially planned. If a transaction cannot acquire a trylock, it releases all its locks and attempts to lock them again. Nonetheless, this method isn't without its drawbacks; I've observed occasions where a transaction takes excessively long to complete.

6.2:
(5pts) Average runtime for each program (use the "real" time)

For the section on the average runtime of each program, where I'm supposed to use the "real" time, I'm not exactly sure what "real" entails. However, I've taken the times from the outputs of the test program. For the program with fine-grained locking, the transaction requests (TRANS) averaged around 4020 seconds, and the check requests averaged 120 seconds, giving me a total average time of approximately 4150 seconds. In contrast, the program with coarse-grained locking had a much higher average time; TRANS requests took about 105000 seconds, and check requests took close to 390000 seconds, with a total average time coming to about 495000 seconds. In practical terms, the fine-grained program terminates promptly after the test program signals it to end, whereas the coarse-grained program often makes me wait several minutes to finish. I'm curious about the method used to calculate these times.

6.3:

1. **(3 pts) Which technique was faster – coarse or fine-grained locking?**

   When asked which locking technique was faster, coarse or fine-grained, I found that fine-grained locking is significantly faster.

2. **(3 pts) Why was this technique faster?**

   Fine-grained locking proved to be quicker because it allows multiple threads to work on different bank accounts simultaneously, unlike the coarse-grained method, which restricts the operation to one thread at a time, processing each account sequentially.

3. **(3 pts) Are there any instances where the other technique would be faster?**

   As for the scenarios where the coarse method might be superior, I don't see many. It seems it could be somewhat better when there's a small number of accounts to manage, which might not justify the overhead of maintaining many locks. In such cases, fewer threads can work in parallel without much contention. However, this advantage diminishes as the number of parallel threads increases and the likelihood of them accessing the same accounts decreases.

4. **(3 pts) What would happen to the performance if a lock was used for every 10 accounts? Why?**

   If a lock were implemented for every ten accounts, there wouldn't be a significant drop in performance. There might be less overhead with fewer locks, but it also raises the probability of several threads trying to lock the same accounts simultaneously. For example, a transaction could involve locking a hundred accounts if they're all in the same ten blocks. Or it could be as simple as locking just the ten needed accounts if they are adjacent.

5. **(3 pts) What is the optimal locking granularity (fine, coarse, or medium)?**

   I think the most effective locking granularity in a practical, real-world scenario would likely be medium. Accounts should be categorized into groups where they'll probably be accessed concurrently, like accounts belonging to the same owner. This way, in a bank setting, if a person has multiple accounts at the same bank, all their accounts would be locked and unlocked in unison. I'm not entirely clear on the best approach to grouping accounts at a lower level.