**Final Project Report**

Name: Truyen Truong

Student Number: 100516976

Course: CSCI 4060U

**Description**

This project implements a bank transaction system where the tellers are run concurrently to simulate a real life bank where tellers are inserting transactions at the same time instead of sequentially inserting transactions. A monitor thread is put in place to log money laundering in the back system. Money laundering that is caught is logged into a text file where it can be viewed. The system is implemented using POSIX Threads and OpenMP. In the OpenMP system, POSIX Threads are used for the monitor thread as OpenMP does not support thread signalling. The results between these two will be compared for performance. The performance will be tested using various combinations of number of tellers(threads) and number of transactions. The number of bank accounts will be tested at 100. Using the various combinations will test the performance of both types of threads in small and large scale systems. The transactions that are supported in the bank transaction system are deposit, withdrawal, and transfers.

In the POSIX Thread version, pthread mutexes are used to lock the bank accounts and other shared variables. The locking and unlocking are done using pthread\_mutex\_lock, pthread\_mutex\_trylock, and pthread\_mutex\_unlock. In the OpenMP version, omp locks are used to lock the bank accounts. They are locked using omp\_set\_lock, omp\_test\_lock, and omp\_unset\_lock. The locks/mutexes and functions to manage them were chosen because locks and mutexes are closest in function in POSIX Threads and OpenMP. Monitor threads in both versions were implemented using POSIX Threads mutexes, and conditions. This was used as OpenMP does not support signalling.

**Java Concurrency**

This bank system could be implemented using Java Concurrency as it supports all that is needed to implement the bank system. For the locks/mutexes, Java Concurrency has locks that can be locked and unlocked simply with lock/unlock functions. For try lock function, an outer and inner re-entrant locks are present. These can be used for the try lock that is required for the transfer function. Java Concurrency also supports thread signalling that could be used for the monitor thread. Thread signalling and waiting can be done in Java Concurrency using notify and wait.

**Results**

These results were calculated on a Linux system using the time command in terminal. The real time uses the real time from the time command while the CPU time is the user time plus system time from the time command.

OR = OpenMP real time OC = OpenMP CPU time

PR = POSIX Thread real time PC = POSIX Thread CPU time

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 100 Bank  Accounts | | Number of Transactions | | | | |
| 10 | 100 | 1000 | 1,000,000 | 1,000,000,000 |
| Number  of  Tellers  (threads) | 2 | OR: 0.001s  OC: 0.000s  PR: 0.001s  PC: 0.000s | OR: 0.003s  OC: 0.000s  PR: 0.001s  PC: 0.000s | OR: 0.004s  OC: 0.004s  PR: 0.002s  PC: 0.000s | OR: 0.416s  OC: 0.952s  PR: 0.301s  PC: 0.708s | OR: 8m + 49.131s  OC: 19m + 22.684s  PR: 8m + 17.092s  PC: 17m + 38.720s |
| 10 | OR: 0.001s  OC: 0.000s  PR: 0.002s  PC: 0.000s | OR: 0.001s  OC: 0.000s  PR: 0.002s  PC: 0.000s | OR: 0.002s  OC: 0.000s  PR: 0.002s  PC: 0.000s | OR: 0.553s  OC: 2.124s  PR: 0.543s  PC: 2.084s | OR: 8m + 49.036s  OC: 31m + 22.148s  PR: 8m + 54.839s  PC: 32m + 4.420s |
| 100 | OR: 0.003s  OC: 0.000s  PR: 0.004s  PC: 0.004s | OR: 0.003s  OC: 0.000s  PR: 0.004s  PC: 0.004s | OR: 0.003s  OC: 0.004s  PR: 0.003s  PC: 0.004s | OR: 0.571s  OC: 2.268s  PR: 0.553s  PC: 2.196s | OR: 9m + 13.735s  OC: 34m + 42.708s  PR: 9m + 3.545s  PC: 33m + 21.688s |
| 1000 | OR: 0.021s  OC: 0.024s  PR: 0.020s  PC: 0.016s | OR: 0.022s  OC: 0.020s  PR: 0.020s  PC: 0.020s | OR: 0.027s  OC: 0.020s  PR: 0.020s  PC: 0.020s | OR: 0.573s  OC: 2.224s  PR: 0.563s  PC: 2.208s | OR: 9m + 50.347s  OC: 37m + 56.436s  PR: 9m + 33.895s  PC: 35m + 48.268s |

**Result Discussion**

From the results overall, When the number of transactions increases, the time it requires to complete the transactions increases. This is expected as there is more work overall to complete. When looking at the number of tellers, it starts off low when there is just 2 tellers. When it increases to 10 tellers, the time it requires to complete the transactions decreases. Increasing from 10 tellers increases the time it requires to complete the transactions. A thought on why it does that is because After 10 threads, the processor executing the program does not have that many cores. It then has to wait and execute in sequential instead of in parallel. Another thought would be that since there are more threads, there would be more thread collisions where they have to wait for another thread to release a lock or finish executing certain parts of the program. Overall comparing OpenMP and POSIX Threads, it looks like both programs are very similar in terms of execution time. When doing 1,000,000 and 1,000,000,000 transactions, the real times and CPU times using OpenMP takes longer than times using POSIX threads.