CIS 415 Operating Systems

Project <3> Report Collection

Submitted to:

Prof. Allen Malony

Author:

*<Sophia Zhang>*

*<951870312 - sophiaz@uoregon.edu>*

*<sophiaz>*

**Report**

**Introduction**

*In this project, a Duck Bank system built using a multithreaded solution was implemented, designed to handle thousands of client requests. The primary objective is to ensure the system remains thread saif while processing varies types of transactions. This included transferring funds, making deposits, withdrawals, and checking account balances. This most important part during these request was to make sure that all threads processing their specific transactions did not interfere with other threads that had the chance of manipulating the same data. This step was implemented using pthread mutex locks and identifying each critical section during data manipulation.*

*The project included 4 parts, and each part was built off of eachother to create a more robust solution. With solution one being a simple single threaded implementation. In part 2, threads were introduced to the program. Allowing a division of labor for a fast execution time using multithreading of 10 worker threads. Part 3 introduced a further robust solution by adding a threshold of updating account balance every 5000 transactions. This way, clients and customers would get their data slightly faster than the first approach. This section was more tricky as it included using signals and thread communication. Lastly, part 4 allowed each user to have a savings account that used shared memory. This was made possible using mmap to synchronize account data between Duck Bank and Puddles Bank. Simply, the Duck Bank updated the balances, and the Puddles Bank applied a flat rate reward to the individual savings accounts.*

**Background**

*In this project, I utilized several system level techniques to develop a multithreaded and interprocess solution for the Duck Bank System. One of the key algorithms I implemented revolved around the tools the pthread library provided. This allowed me to create thread synchronization using mutex locks and conditional variables. This most importantly allowed me to lock critical sections to prevent issues like dead locks and race conditions. Deadlocks meant that your program is waiting for eachother, in a scenario that both thread needs to update something where a certain info they need will not be available because they are waiting for each other to release it. This would prevent your program from running. Another important factor to consider was race conditioning. This meant that the threads were manipulating data based on who reached the transaction first. In this scenario, I would be resulted with inconsistent data because each thread is racing to determine who can reach a certain data manipulation first to update, hence producing different results each time. The pthread\_mutex\_t was used to protect these critical sections. Especially during the if the transaction request was a transfer. Where accounts would access and update at the same time, we would lock both destination accounts and current accounts first to prevent deadlocks and race conditioning. This ensures that multiple threads can safely access shared resources without problems.*

*During the implementation of part 3, where I incorporated thread signalling to provide faster updates to customers. I included the use of pthread\_cond\_wait and pthread\_cond\_broadcast to synchronize the threads. This allowed the bank to update accounts balances once a certain number of transactions threshold was reached. This method ensures that all of the worker threads properly paused to notify the bank to apply reward rate updates, after which they resume back to their transaction processing.*

*Another important method used in part 4 was memory mapping with mmap() to enable inter process communication between Duck Bank and Puddles Bank. Shared memory allowed me to create the common space that allowed both processes to access and modify account information in a synchronized manner. I used mmap() to allocate memory for account data, ensuring that changes made by the Duck Bank process was almost immediately available to Puddles Bank, without requiring a heavy overhead of expensive data transfers. Hence to manage the shared memory I used munmap() to properly deallocate resources after the processes had finished their work. This method allowed a clean way for interprocess communication between the two banks.*

**Implementation**

*For the implementation of this project, I worked through several stages of development, gradually transforming the bank system from a simple single threaded version to a multithreaded solution. In part 1, I started with a straightforward approach single threaded solution that handled transaction in the order it appeared on the input document. This was the foundation of my project so it is free of memory leaks, confirmed with valrgind.* 

*In part 2, the implementation that required thread synchronization, was a long journey for me. I chose to proceed with not storing all of the transactions in a memory space. But instead initialize a different file pointer for each worker thread that would then move the file pointer to the part of the file that they would read from. Beyond this involved me creating a separate transaction struct to store information to pass into the pthread\_create to spawn worker threads. Further developing into the process transaction function, it would then identify the critical sections and use a lock to prevent any race conditioning. This part was especially challenging to implement as it was hard for me to identify why a program wasn’t working due to an error of a deadlock or race conditionings.*

*This most challenging part actually came down to part 3, which required complex thread signaling and coordination. The goal was to synchronize the worker threads with a newly created bank thread so that after processing a certain amount of transaction equivalent to the threshold, one of the worker threads would then communicate and send a signal to the banker thread to execute the applyRewards() function. During this process all worker threads should wait to receive a signal from the banker thread before proceeding with the next transaction. I ran to an issue where they cycle would appear only to run once. It would only run through transactions and signal the bank thread once before it would stop executing. My suspicion would be that the worker threads were not proper stopped and signaled correct to resume back to their previous task. Thus, causing the program to not run. I believe that I could possibly be not sending the correct signal to resume the worker thread. I used a lock to protect the banker thread which then would use broadcast to wake up all of the worker threads, I believe this part was working fine. But the part where I checked if all 10 worker threads signalled a stop was implemented incorrectly.*

*Since my part 3 was not implemented correctly, I tried my best to complete parts of part 4 using the memory mapping. My code unfortunately did not run due to issues in part 3, but I did my best to implement parts of the requirement.*

**Performance Results and Discussion**

*In terms of performance, my project was designed to handle multiple transactions concurrently. Everything from part 1 and part 2 is working as expected. But unfortunately my part 3 was not able to synchronize well together. There are uses of signalling, but I believe that I am not using the threads calculation correctly which prevented my program from running after the first iteration. My program produces an output different from the initial balance which means that some form of transaction processing was done. As mentioned in the previous paragraph, since part 3 was not working correctly, it was very difficult to proceed onto part 4. I used the best of my knowledge and man page information to implement memory sharing, and is unsure of the result since my program could not run.*

**Conclusion**

*In conclusion, this project was semi successful in implementing multithreading bank systems that handled concurrent transactions with proper synchronization to insure that no accounts or transactions are running into deadlocks or race conditions. Despite the difficult level of this project, it was very enjoyable and I learned a lot from it.*