CS 415 Operating Systems

Project 3 Report

Submitted to:

Prof. Allen Malony

Author:

Nathan Gong

951892158

ngong

Report

Introduction

For this project, I was tasked with the objective of creating a bank that processed different transactions, where the number of "workers" (threads) is equal to the number of accounts existing. Obviously, this isn't realistic, but the point of the project is creating something that attempts to create the same interactions between processes and threads in real processors. The processes interact, as do the individual threads (workers) within their process.

The difficulty of the project comes in the form of attempting to prevent deadlocks and other setbacks that may occur with real threads when they share resources between each other. The program that I wrote progressively became more advanced, with the end goal of improving functionality while preventing deadlocks and keeping in mind race conditions and other issues that real processors must go through.

Background

This project became more abstract/ambiguous with each part. While the functionality and implementation of the original bank is quite simple, the introduction of the "pthread" library introduces many potential problems. Deadlocks and race conditions are chief among them. Timing the processes and threads so that shared resources are protected but not deadlocked is complicated and takes some thinking, and timing everything so that the processes continue without fault is also slightly difficult.

No explicit algorithms were used in my implementation, but the closest thing that comes close is the interaction of mutex lock/unlocks on certain accounts. Because each accounts has its own personal mutex lock, and because currency transfers between accounts is a possible type of transaction, deadlocks can occur if the accounts aren't locked/unlocked in a precise, consistent manner. We wouldn't want two accounts to be locked between two processes, each vying for the other's account to unlock their respective mutexes.

Implementation

As stated before, deadlocks and race conditions are most concerning in the implementation of this program. Implementing the conditions, signals/broadcasts, mutex unlocking/locking, and process creation/joining all with the "pthread" library was a challenging but incredibly interesting process. It was quite interactive, figuring out for myself what order process should be stopped/started/created/joined. It was interesting what portions of code were considered critical sections, what resources needed to be shared, and thus what needed to be protected.

I had initially thought it was just the transactions that needed to be protected, in essence the accounts, but many other things had to be protected to ensure threads weren't running and transactions weren't being processed when they shouldn't have been. To an extent, figuring out what processes/threads should signal/broadcast what conditions was also interesting and fun.

Performance Results and Discussion

I feel like my project performed quite well considering everything. I spent a lot of time on part 1 because I thought I was getting the output wrong when in reality I hadn't realized the update_balance function should've been called at the end, but every other part wasn't as bad. I never actually hit a roadblock in terms of deadlocks giving me trouble for a reason I couldn't figure out relatively quickly, and in the end, I felt like my program performed quite well in terms of thread/process interactions.

Conclusion

This project was pretty fun. It's hard to say which project this term was most interesting, but this was no slacker. The most interesting part of this project, in my opinion and my own experience, was the implementation of the pthread mutexes, conditions, and barriers in such a way where they're functional but don't cause the program to worsen in performance.