CIS 415 Operating Systems

Project 3 Report Collection

Submitted to:
Prof. Allen Malony

Author:
Abhinav Palacharla
abhinavp
951979070

Report

Introduction

The primary purpose of this project was to further the parallel programming from the last project but instead learn about implementing parallel programming through multithreading. I found this to be much harder than the last project as threads seemed a lot more sensitive to timing related issues.

Background

Parts 1 and 2 are straightforward and I am not going to talk about them as there is nearly no manual synchronization that we needed to implement. For part 3 I had a very tough time getting all my threads to synchronize, and getting them to have predictable behavior. Up until about an hour ago (I am writing this at 6pm on Dec 1st) I thought my program was working completely fine. I was doing all my testing on an Ubuntu v22.04 Docker Container. When manually testing I found no deadlock and I was producing the correct output every time. I had my program working for 3,000+ iterations with no deadlock and I only tested 3000 times because my battery was draining. However when I tested my program in an Ubuntu VM I found that I was going into deadlock far more often. However the times that I did not go into deadlock my program had the right output. I really thought there would be little to no difference in an Ubuntu docker container versus an Ubuntu virtual machine, I guess that wasn't the case. I will try and debug why this is happening and resubmit if I can fix it. I hope you can see that when my program does not go into a deadlock it is working and I do receive credit for the other portions of part 3 & 4. I do understand that I should've done my due diligence and tested in the Ubuntu VM a lot sooner.

For part 4 my program works with some deadlocking surely because of the same reason from part 3. But when it does not deadlock it works perfectly.

Implementation

Implementation for parts 1 and 2 are fairly straightforward. The only thing I think I can talk about for these parts is that I decided to write my own parser, structs, and general base from scratch without using anything from previous projects. I found this helped keep my code a lot more organized and tailored to this project.

For Part 3 I will just provide a high level overview of how I handled the synchronization as I think that is the interesting part. In worker_thread.c you can see all my logic for the worker threads. Essentially I first have all my threads wait at a barrier before an infinite while loop. This gets everyone to sync before any of the threads start processing transactions.

The while loop is how I implemented the concept of a thread pool. They all stay alive until a certain shared condition is met at which point they all exit. Inside the while loop there is a large if statement. The first part is the logic to do if the total transactions processed is below 5000 AND if the worker thread is not done. If either of these conditions fail then the thread will go into the else statement where it will sit at a barrier until the other threads join it. Inside the first half of the if statement processes continually dequeue from their allocated transaction queues until they find a transaction to run at which point they run it. In order to run a transaction they must first acquire a token (a mutex lock) which will guarantee that they will run a valid transaction. If the transaction they run is invalid they will acquire the lock again and decrement the counter. When all threads reach 5000 transactions they all go sit at the barrier in the else statement. This is the same barrier where threads go sit if they are done with all their transactions (when they dequeue from their own queue and it is NULL). After syncing at this barrier we check again if the number of transactions processed is still 5k. If it is not they all go try again. If it is 5k then all the threads go sit at a barrier except for worker thread 0. Worker thread 0 will signal the bank thread to start, reset num transactions processed to 0, and then will wait at a pthread_cond_wait. Once the bank thread is finished updating then it will signal worker thread 0 to resume, at which point it will go join the other worker threads at a barrier. Then the worker threads will continue doing work.

For the exit condition which will occur when no threads have any work. This is determined by a shared counter which each thread decrements when it dequeues a NULL transaction. When this counter reaches 0 all threads will be waiting at the barrier in the else statement. T0 will signal the bank and the bank will see there are no more threads with work left. The bank will do its last update, signal T0 and then the bank will exit. T0 will wakeup, see that there are no threads with work left, and will flip a shared variable to tell them all to exit. All the threads will then meet at a barrier. After they pass, they will then each pass an if statement because of the flipped variable and will exit.

Part 4 was pretty simple, I just set up shared memory, then forked at the very beginning. Puddles waits at a sigwait for duck bank. Duck bank will load account info into shared memory then signal Puddles. Puddles will then read shared memory, copy it, then unmap it. Puddles then go to an infinite while loop with a sigwait in it. Each time duck bank's bank thread updates it signals puddles. Puddles will get one iteration of the while loop and update its accounts. If the bank thread is going to exit it sends a different signal to puddles. This signal also lets puddles have one more iteration in the while loop; however at the end of the loop it will enter an if statement which will break it out of the loop. Then everything will free and exit. I am pretty happy with this implementation as it is very clean.

Conclusion

This project showed me just how finicky/sensitive threads can be. It also made me understand why they are almost never used in production since they can be very unpredictable and hard to maintain. It is difficult for one person to work on a multithreaded application much less a team, as everyone on the team needs to be aware of the entire system. I am pretty happy that I got everything working with a very high reliability rate on my personal setup but am pretty disappointed to see it be less reliable on the VM, for me this really stressed just how sensitive multithreaded applications can be.