CIS 415 Operating Systems

Assignment 3 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Brad Bailey

Report

Introduction

In this project I am implementing text processing with a multithreaded application to simulate online banking. The program reads in information about accounts from an input file, and then processes instructions to change said information accordingly. In its final form, the program uses multithreaded processing to achieve this task, using ‘worker’ threads to process instructions and a ‘bank’ thread to coordinate the workers and update account balances.

Background

This project relies heavily on the pthreads library. This library allows a user to create and manage threads to allow for multicore programming functionality. Crucial to multithreaded applications, the library provides methods for locking mutexes to prevent race conditions in critical sections of code, and methods for synchronizing and communicating between threads during their execution. In this project, the ‘algorithmic design’ is very straightforward - at the end of the day, the task is just text processing. Implementing the synchronized multithreaded functionality is the real challenge.

Implementation

The main technical challenges faced were race conditions, deadlocks, and thread communication. For race conditions, simple mutex locks are sufficient to avoid the critical section problem. Mutexes are necessary for both updating account information and updating shared counter variables. If either of these happened to be modified by more than one thread at a time, incorrect and/or inconsistent behavior could result. To avoid deadlocks, it was sufficient to avoid nesting mutexes. If mutexes are nested, there is a possibility that a ‘cycle’ can arise where two threads each own a mutex that the other needs to continue, and the program cannot proceed. Lastly, my thread communication relied entirely on calls to the function pthread_barrier_wait. While it is entirely possible that other pthreads functions (ie pthread_cond_wait, among others) could have achieved the same result, in my work barrier_wait proved the most straightforward function to use. The pseudo structure for the worker threads is roughly as follows:

```
    For (lines of instructions)
        If (# of instructions processed across all threads < 5000)
            If (password matches account)
                Process instruction
                Increment count
        Else (# instructions >= 5000)
            Decrement line counter
            Barrier_wait()
            (Bank processes here)
            Barrier_wait()
            Resume for loop
    For loop exited
    Increment 'all done' counter
    While (some threads are still working)
        Barrier_wait()
    If (last thread to finish for loop)
        Barrier_wait()
    Done
```

And the bank thread is roughly:

```
    Barrier_wait (waiting for workers)
    Update account balances
    Reset count of instructions processed
    Barrier_wait (workers resume)
    If some threads are still working, repeat
```

With the barrier initialized to the number of workers + 1 for the bank thread, the barrier provides all the functionality needed to implement the worker - bank communication needed. Especially since pthread_cond_wait suffers from 'spurious returns', where the function returns even though the condition is not met, I found this approach much easier to understand and implement.

Performance Results and Discussion

My application, to the best of my knowledge, meets the specifications as outlined in the project instructions. There are no leaks or deadlocks, the final output is correct, and the 'intermittent' results in the account*.txt files are different on every run due to multithreading, as expected.

I was getting some valgrind errors of ‘still reachable’ blocks from pthread_create, but could not solve that problem at first. I believe I was correctly creating and destroying threads as demonstrated on the man page for pthread_create. Threads were created, waited on to finish, called pthread_exit(), and were joined. However, memory problems persisted when I used pthread_join. The only way I found to solve this problem was to forgo joining finished threads entirely, and instead call pthread_detach at the end of each thread’s execution. While this may not have been the intended implementation of the project, without more time I cannot find a better solution. At least it works!

Conclusion

Multi threading is HARD. Even with just this simple task of text processing, there are so many ways a multithreaded application can fail, and debugging is much harder than normal. I had to rely on many printf statements throughout my work to make sure threads were behaving correctly during their execution, which was tedious at best and at times useless. However, the benefits of multithreading are worth it - if this project was scaled up by several orders of magnitude, the multithreaded versions would perform much better than their single threaded counterparts.