CIS 415 Operating Systems

Project 3 Report Collection

Submitted to:

Prof. Allen Malony

Author:

Ethan Reinhart

ereinha3

951909814

Report

Introduction

This project was to create a program that would create threads to parse the input to decrease runtime. The pthread library was used in this project to create pthreads that would execute allocated chunks of the total work. Mutexes were also used so that individual threads would not simultaneously access different pieces of data. We were given a struct that represented an account and had to read in input data from the file and create accounts from this input data. There are many ways of doing this such that the pthreads would have access to these accounts but I opted to use a global variable. Furthermore, we were required in to create an update_balance function, that would be called every 5000 transactions as well as on termination. This update balance would read from the account array and update each account based on its reward rate and its current balance. The particularly challenging part of this project is that mutexes and conditions needed to be carefully allocated such that a deadlock would not occur. A deadlock can occur in a variety of ways, but in this specific project, it would most commonly occur when a thread is stuck waiting on a condition that will never arrive or if two threads are requesting to lock without any unlock being called. Deadlocks are one of the hardest things to debug because valgrind will not detect them and gdb has few features to assist this.

Background

Nearly all of my references for this project were derived from LINUX manuals and StackExchange forums regarding uses of functions and mutexes. This was the first time that I had worked with mutexes and I found them to be particularly challenging. It was difficult at first to see their functionality but ultimately, I believe I understand them now. They serve to lock before a specific variable is accessed so that any other thread trying to access the variable will see that the corresponding lock is locked and be forced to wait before accessing the variable. Pthread conditions are also very important. Two functions, pthread_cond_broadcast/pthread_cond_signal and pthread_cond_wait proved to be very beneficial in these case. Pthread_cond_broadcast will change a given input condition for all threads who have called pthread_cond_wait with that specific condition. Pthread_cond_signal will do the same but for only one thread. Neither of these will directly lock or unlock mutexes. Pthread_cond_wait takes in two arguments, a condition and a mutex. The functionality is as follows: First you must called pthread_mutex_lock on the mutex to lock it. Next, you should call pthread_cond_wait with the mutex and a condition, which unlocks the mutex and relocks it once the condition is changed. In another thread, pthread_mutex_lock is called which locks the mutex after is had been unlocked. Pthread_cond_signal/broadcast will broadcast the condition has been changed and pthread_mutex_unlock will be called in that same thread. Next, in the original thread, pthread_cond_wait will see that the mutex has been unlocked and relock it. Next, some code can be executed and finally pthread_mutex_unlock will called, yet again freeing the mutex. Many things can go wrong throughout these functions making debugging incredibly difficult. This was the predominant problem I had in my project.

Implementation

My biggest struggle in this project was implementation of the mutexes and condition variables. Initially, had read in lines from the buffer and parsed them as I went. This served to not be of use in part2, so I scrapped this idea and created a large array of command_line variables. From here, I created a function that would count the total number of transactions, and divided this by the number of accounts to get the interval. When I created the pthreads, I passed in a pointer to i in my for loop, such that thread 0 would receive 0, thread 1

would receive 1, and so on. This allowed me to multiply this by the line_count / num_accounts so that I could attain the correct starting line in my for loop. From here, I iterated from 0 to the interval, meaning each account would process command_line[curr_line:curr_line+line_interval-1]. I implemented the parsing using string_parser from project2. This allowed me to easily process transactions by reading the number of tokens per line and data from the individual tokens. I suffered many errors with this but ultimately found it best to free the entire command line array at the end of the process execution instead of dynamically. Using mutexes, I created four different mutexes, to guard the transactions, the number of actives threads, the update condition, and the number of waiting threads. I additionally created two conditions, one to call the update and one to signal that balances had been updated.

```
if (transaction_count >= 5000){
        pthread_mutex_lock(&wait_lock);
        //printf("Waiting thread count increased.\n");
        waiting++;
        if (threads_alive == waiting){
                pthread_mutex_lock(&update_lock);
                printf("Signaling to update...\n");
                pthread_cond_broadcast(&time_to_update);
                pthread_mutex_unlock(&update_lock);
        }
        pthread_cond_wait(&updated, &wait_lock);
        printf("Resuming thread %d\n", position);
        pthread_mutex_unlock(&wait_lock);
}
```

This code shows how I signaled the update_balances function to update

```
pthread_mutex_lock(&update_lock);
pthread_cond_wait(&time_to_update, &update_lock);
pthread_mutex_unlock(&update_lock);
printf("Updating balanced based on reward rates...\n");

pthread_mutex_lock(&wait_lock);

for (int i = 0; i<NUM_ACCTS; i++){
        acts[i].balance += acts[i].reward_rate*acts[i].transaction_tracter;
        acts[i].transaction_tracter = 0;
        //printf("Balance updated in account %s.\n", acts[i].account_number);
        FILE* fp;
        fp = fopen(acts[i].out_file, "a");
        char* string = malloc(sizeof(char)*124);
        sprintf(string, "Current Balance:\t%f\n", acts[i].balance);
        fprintf(fp, string);
        free(string);
        fclose(fp);
}
*count += 1;
transaction_count = 0;
waiting = 0;
//printf("Waiting thread and transaction counts reset.\n");
pthread_cond_broadcast(&updated);
pthread_mutex_unlock(&wait_lock);
if (threads_alive == 0){
        break;
}
sched_yield();
```

This code shows how it was implemented in the update_balances function

Performance Results and Discussion

After much debugging, many full calendar days of hours, and multiple submissions, I believe that I have finally achieved the exact required final output. I had many errors that remained unrealized in part1 that carried over into later parts. One of these was that I was also increasing the transaction_tracter in the thread money would be added into for Transfer transactions. This caused errors later and incorrect output when printing updated account balances, but was ultimately fixed in part1. I believe no deadlocks occurred in part2 as I ran it with a bashscript 10000 times. Initially, I was very excited as I ran the executable and finally, after countless hours, achieved no deadlocks. In OH however, Alex told me that the code would be run 10000 times and that I should create a bashscript to do so. I did this and ultimately ran into deadlocks. I created countless print statements, trying to find exactly where the deadlock was occurring but this proved useless. I even tried

using fflush() to force the buffer to print but this also did not prove useful. Troubleshooting this error probably took legitatemelty 20 HOURS. What I discovered was that, due to my implementation, in incredibly rare cases, the thread signaling to update could call lock on the update lock, signal, and then unlock the update lock BEFORE the lock was relocked and cond_wait was called in bank thread. This resulted in the bank thread inifitely waiting for a conditional change that would not arrive. I mitigated this by created a truth variable that served to only hold true if the bank thread has locked this update lock. Otherwise, the signaling thread would wait until the bank thread had in order to signal. Then, I ran the program using my bash script and no deadlocks occurred over 10000 iterations. This took probably 15 minutes to run. The final part also undertook multiple troubleshooting errors. Initially, I wrote working code but did not use mmap(). Truthfully, I do not understand why mmap is useful considering I used global variables, but I still implemented it in my own way. On large error that I had was that I was calling _exit() in the child process, when everything allocated in the main process must still be freed as a copy is sent to the child process. I fixed this by getting rid of _exit() and implementing conditionals to check the current pid in whatever process was running. I also struggled with string allocations for output files here for some reason, but ultimately I believe that I achieved good working output.

Conclusion

I am finally, after multiple submissions and the most work I have yet put into a project in college, happy with my final product. I gained a strong understanding regarding the implementation of pthreads, mutexes and conditions. I feel confident that I could explain these concepts to an introductory programmer, yet still struggled in nuanced implementation. I understand the functionality of pthreads, how they decrease runtime and increase efficiency, and how they access memory / what memory they can access. I additionally understand the importance of mutexes and finally understand the concept of the Dining Philosophers Problem, which is directly related to mutexes. With that being said, I still struggled with implementation. The same can be said for pthread conditions, I understand how essential they are now but still struggle with implementation. There were so many individual moving parts in this project that it was incredibly difficult to keep track of all mutexes and conditions individually. I am very proud of the all I have accomplished this term (lesser so of this project but as I said with more time I believe I could've done better) and I hope you also see the great benefit I've reaped from all of your guidance and help!