CIS 415 Operating Systems

Project <3> Report Collection

Submitted to:
Prof. Allen Malony

Author:
<Zachary Weisenbloom >
<zweisenb>
<951911117 >

Report

Introduction

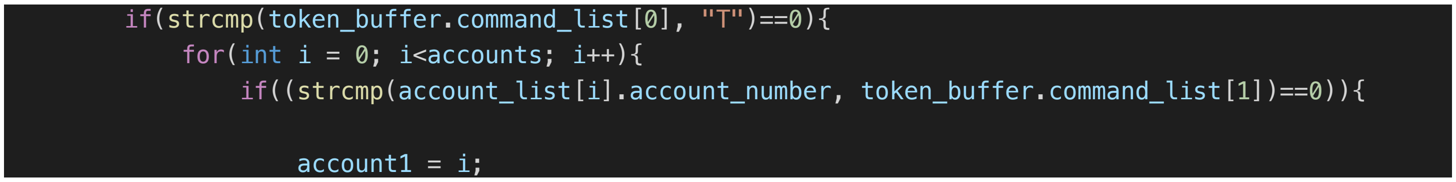
This project simulates a bank by processing transactions in parallel and calculating reward rates based on transactions. This project contains 4 parts each building on the last by adding additional functionality such as multithreading, additional synchronization control and extra processes. The first part of the project implements a single threaded solution implementing transfer, deposit and withdrawal functionality as well as implementing an update balance method that adds the reward rate. In this implementation the transaction file is opened, and each line is parsed using the parser from the previous projects and converted into relevant data to be processed. The second part implements basic multithreading functionality by spawning 10 different threads and creating critical sections to prevent race conditions when multiple threads write to the same peace of memory in the bank. The total transactions were split up into 10 different sections so that each thread processes the same number of lines. After all lines are processed, a barrio is triggered to update the balances with the reward rate. The 3rd part implements additional synchronization by starting all the processes by a single from main and updating the balances with there reward rates after 5000 lines have been processed. When the counter reaches 5000 lines all the threads wait and the update balance function starts updating all the accounts. The 4th part of the project implements an additional bank the puddles savings bank using a separate process. The processes communicate using mmap allowing the puddles bank to access all the account information in the duck bank. The puddles bank updates its bank account savings every time the duck bank updates all of the balances.

Background

Each process I created using pthreadcreate in a for loop passing in my process transaction function and the indices for that thread to processes. The main thread waited for each of the other threads to finish using join in a four loop. I used pthread_mutex_lock and used pthread_mutex_unlock to lock and unlock each account mutex to ensure that no race conditions created incorrect balances. To prevent deadlocks in the scenario where the transfer needs to access two different accounts a "dining philosophers problem" I made sure to always grab the left lock (in terms of the the account list index). I used pthread_cond_wait in conjunction with my bank mutex to make sure that no threads can start a new line while it is time for the bank to update all the accounts on the 5000th balance. pthread_cond_wait is especially useful because it releases the lock it is trying to acquire while it waits for a conditional to be true. In this case my conditional was a is bank boo lion that would prevent missed signals. In order to signal the bank to start updating balances I used pthread_cond_signal which would wake up the bank waiting on the bank conditional. To wake up all of the threads after the bank had finished updating I used pthread_cond_broadcast which sends out a signal to every thread with that conditional. for part three to prevent processes from starting before all threads had been created and signaled from main I used pthread_barrior_wait to make sure that all threads, the bank thread, and the main thread(main thread reaching the barrier is the signal) before allowing all threads to continue execution. For the last section, I used fork to create a new the puddle bank and running the puddle bank function. Instead of using malloc to allocate memory for my bank account list I used mmap to create the allocation so that both the parent process (duck bank) and the child processes (puddles bank) could access the information.

Implementation

My general approach to the project was to give each thread its own set of indices and using a while loop each thread would skip to its line before processing its range. I used getline to grab each new successive line, then I would parse it, and grab the locks for all accounts in the transaction. Below is my implementation of how I grabbed all of the locks. For transfer I first grab all of the indexes (related to my accounts list) of each account. I then grabbed the leftmost account first before grabbing the right as described above to prevent dining philosophers.



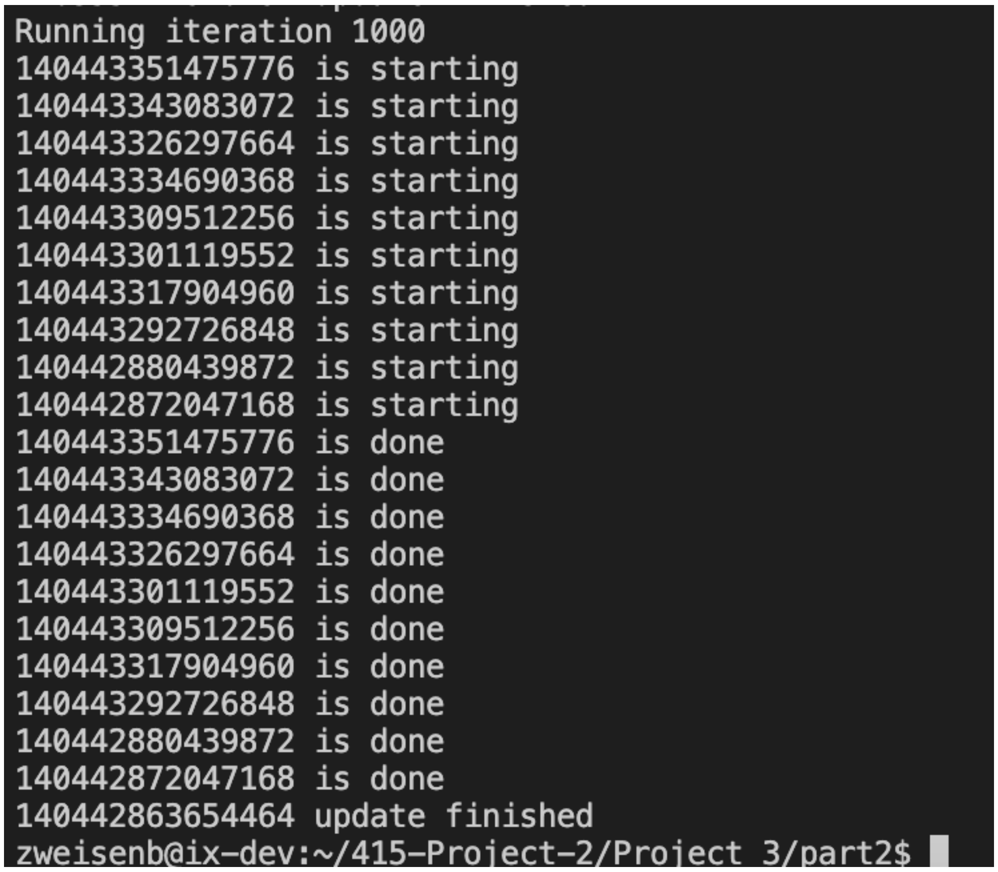
```
        if(strcmp(token_buffer.command_list[0], "T")==0){
            for(int i = 0; i<accounts; i++){
                if((strcmp(account_list[i].account_number, token_buffer.command_list[1])==0)){

                    account1 = i;
                    for(int j = 0; j<accounts; j++){
                        if(strcmp(account_list[j].account_number,
token_buffer.command_list[3])==0){
                            //printf("second accoutn is %s \n", token_buffer.command_list[1]);
                            account2 = j;
                        }
                    }
                }
            }
            if(account1 < account2){
                pthread_mutex_lock(&account_list[account1].ac_lock);
                pthread_mutex_lock(&account_list[account2].ac_lock);

            }else{
                pthread_mutex_lock(&account_list[account2].ac_lock);
                pthread_mutex_lock(&account_list[account1].ac_lock);
            }
        }else{
            for(int i = 0; i<accounts; i++){
                if((strcmp(account_list[i].account_number, token_buffer.command_list[1])==0)){
                    account1 = i;
                }
            }
            pthread_mutex_lock(&account_list[account1].ac_lock);
        }
```

Here is part of my implementation for my duck bank for part 4. Before the start of what you can see I lock the account mutex before looping through all of the accounts and updating the reward rate. After this is finished I set the count to be zero so the next 5000 counts can be reached. The next section sets is_bank to zero letting threads that have not yet hit the signal to continue execution without missing the signal and waiting. the has_sig is set to zero so that a new signal can be sent. The next section restarts puddles bank and sets a mmaped sigsnet boolean to 1 to prevent a missed signal if the puddles has not yet started waiting. bank_mutex has also been mmaped to prevent race conditions between the two banks.

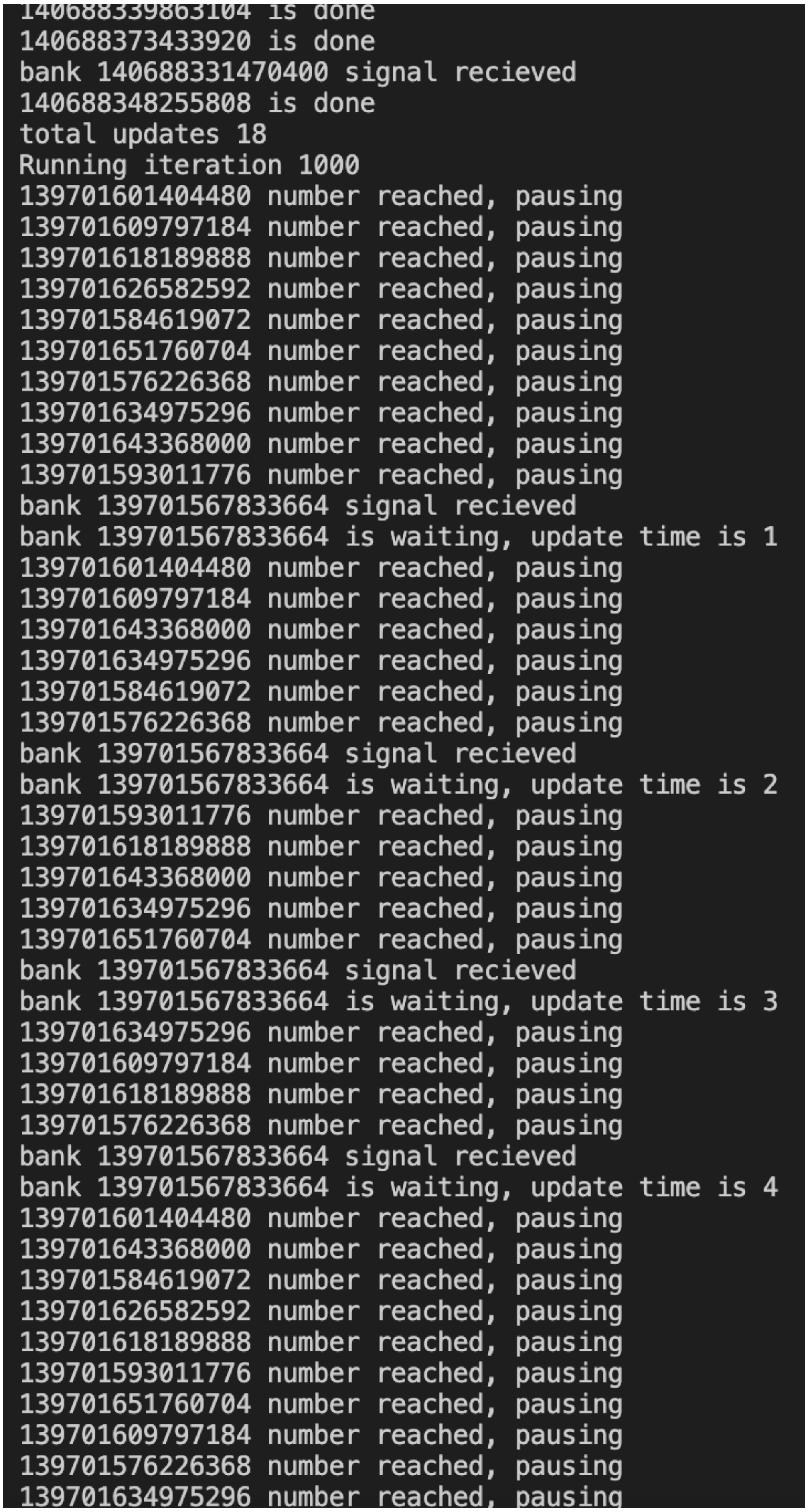
```
for(int i = 0; i<accounts; i++){
            pthread_mutex_lock(&account_list[i].ac_lock);
            account_list[i].balance  =  account_list[i].balance + (account_list[i].reward_rate *
account_list[i].transaction_tracter);
            account_list[i].transaction_tracter = 0;


            snprintf(filename, sizeof(filename), "./output/act_%d.txt", i);
            file = fopen(filename, "a");
            fprintf(file, "Balance: %.2f\n", account_list[i].balance);
            fclose(file);

            pthread_mutex_unlock(&account_list[i].ac_lock);
        }
        pthread_mutex_lock(&count_mutex);
        count = 0;
        //printf("line count: %d\n", lines_processed);
        pthread_mutex_unlock(&count_mutex);

        is_bank = 0;
        has_sig = 0;
        pthread_cond_broadcast(&cond);

        kill(pid, SIGCONT);
        pthread_mutex_lock(bank_mutex);
        *sigsent = 1;
        pthread_mutex_unlock(bank_mutex);
        if(active_threads == 0){
            *puddles_bool = 2;
            break;
        }
```

Performance Results and Discussion

I have tested parts 2 and 3 1000-5000 times without deadlock. My code additionally passes helgrind without any errors. Part 4 also runs without getting any deadlocks, but I have not tested it as much. I have screenshots of my output for 1000 runs from part 2 and part 3 below. I used the command "for i in {1..1000}; do echo "Running iteration $i"; ./bank input-1.txt; done".



```
Running iteration 1000
140443351475776 is starting
140443343083072 is starting
140443326297664 is starting
140443334690368 is starting
140443309512256 is starting
140443301119552 is starting
140443317904960 is starting
140443292726848 is starting
140442880439872 is starting
140442872047168 is starting
140443351475776 is done
140443343083072 is done
140443334690368 is done
140443326297664 is done
140443301119552 is done
140443309512256 is done
140443317904960 is done
140443292726848 is done
140442880439872 is done
140442872047168 is done
140442863654464 update finished
zweisenb@ix-dev:~/415-Project-2/Project 3/part2$
```



```
140688339863104 is done
140688373433920 is done
bank 140688331470400 signal recieved
140688348255808 is done
total updates 18
Running iteration 1000
139701601404480 number reached, pausing
139701609797184 number reached, pausing
139701618189888 number reached, pausing
139701626582592 number reached, pausing
139701584619072 number reached, pausing
139701651760704 number reached, pausing
139701576226368 number reached, pausing
139701634975296 number reached, pausing
139701643368000 number reached, pausing
139701593011776 number reached, pausing
bank 139701567833664 signal recieved
bank 139701567833664 is waiting, update time is 1
139701601404480 number reached, pausing
139701609797184 number reached, pausing
139701643368000 number reached, pausing
139701634975296 number reached, pausing
139701584619072 number reached, pausing
139701576226368 number reached, pausing
bank 139701567833664 signal recieved
bank 139701567833664 is waiting, update time is 2
139701593011776 number reached, pausing
139701618189888 number reached, pausing
139701643368000 number reached, pausing
139701634975296 number reached, pausing
139701651760704 number reached, pausing
bank 139701567833664 signal recieved
bank 139701567833664 is waiting, update time is 3
139701634975296 number reached, pausing
139701609797184 number reached, pausing
139701618189888 number reached, pausing
139701576226368 number reached, pausing
bank 139701567833664 signal recieved
bank 139701567833664 is waiting, update time is 4
139701601404480 number reached, pausing
139701643368000 number reached, pausing
139701584619072 number reached, pausing
139701626582592 number reached, pausing
139701618189888 number reached, pausing
139701593011776 number reached, pausing
139701651760704 number reached, pausing
139701609797184 number reached, pausing
139701576226368 number reached, pausing
139701634975296 number reached, pausing
```

Conclusion

Give any concluding remarks here. If you learned anything talk about that here as well. If you discovered anything interesting, then talk about it here too.
One thing I learned was that it is really important to map out all of the possible race conditions and deadlocks beforehand to plan out your implementation. It is really easy to increase the complexity of your project which also greatly increases the difficulty. For my implementation I relied on boolean variables in addition to pthread_cond_wait which helped to remove deadlocks by adding an additional layer of redundancy.