Project 3

(Duck bank)

CIS 415 - Operating systems

Fall 2023 - Prof. Allen Malony

Due date: **11:59 pm, Friday, December 1st, 2023**

Introduction:

Today, we can do almost anything on the Internet including banking. In this project we are going to build a multithreaded solution for the Duck bank to handle hundreds of thousands of requests from our clients. The ultimate goal is to come up with a thread safe solution which can be applied on even larger traffic volumes while ensuring the correctness of the clients' information.

Project Details:

Our base tasks are pretty simple in this project; read in all the accounts information, verify and process all the transaction requests. These requests will include “Transfer funds”, “Deposit”, “Withdraw”, and “Check balance”. By now, I am sure you are all familiar with single thread programming, however, in this project you will be coordinating 10 worker threads and a bank thread to accomplish this job together.

To make this project even more challenging, the bank has some additional work to do while the requests are being handled. To encourage people spending and saving more money through the Duck bank, everyone has a custom reward rate. Whenever a deposit, withdrawal, or transfer fund is initiated by the account, the amount will be added to a tracker. Once the number of transaction requests handled (excluding check balance) across all worker threads have reached a threshold, the worker threads will all pause and notify the bank thread that it is time to update all accounts balance based on their reward rate and transaction tracker. When the bank is done updating, it will then notify all the worker threads to go back to work. This process will repeat until all the worker threads are done with their tasks, and your program will end with the bank thread updating the balance one last time.

To avoid any complications, in this project, there will be no case of error handling, the input file and testing file will have the exact same format. However, the length could differ. In the case of “Withdraw” and “Transfer funds”, an overdraw will never happen (even if in extreme cases that somehow withdraw and transfer funds are processed before all other requests). You may also use project 1’s string_parser as an external tool to help you with file I/O.

There are 4 parts to the project, each building on the other. For the first 2 parts, you do not have to worry about calculating the reward until the end of the program.

The objective is to get experience with a combination of OS techniques in your solution, mainly threading, synchronization, and file I/O.

Function & lib requirements:

Remember, manpage is your friend.

- Library
  - <pthread.h>
- Functions you need
  - pthread_create()
  - pthread_exit()
  - pthread_join()
  - pthread_mutex_lock()
  - pthread_mutex_unlock()
  - pthread_cond_wait()
  - pthread_cond_broadcast() or pthread_cond_signal()
  - pthread_barrier_wait()
  - pthread_barrier_init()
  - sched_yield() (optional but strongly recommended)
  - mmap()
  - munmap()
  - memcpy()
- Functions you have to write
  - void* process_transaction (void* arg)
    - This function will be run by a worker thread to handle the transaction requests assigned to them.
    - This function will return nothing (optional)
    - This function will take in one argument, the argument has to be one of the following types, char**, command_line*, struct (customized).
  - void* update_balance (void* arg)
    - This function will be run by a dedicated bank thread to update each account's balance base on their reward rate and transaction tracker.
    - This function will return the number of times it had to update each account
    - This function does not take any argument

Input file structure:

- First line

Line 1: n total # of accounts

- Account block

Line n:	index #	indicating the start of account information
Line n + 1:	#.....#	account number (char*)
Line n + 2:	******	password (char*)
Line n + 3:	######	initial balance (double)
Line n + 4:	#.##	reward rate (double)

- Transaction lines (separated by space)

Transfer funds:	"T src_account password dest_account transfer_amount"
Check balance:	"C account_num password"
Deposit:	"D account_num password amount"
Withdraw:	"W account_num password amount"

- Additional information

In these requests, there are some invalid requests (wrong password). Getting to know the total number of each type of request and invalid requests will help you debug down the road.

Part 1: Single threads solution

A header file is given to you named "account.h" In this header, you will find the following struct.

```
#ifndef ACCOUNT_H_
#define ACCOUNT_H_

typedef struct
{
    char account_number[17];
    char password[9];
    double balance;
    double reward_rate;

    double transaction_tracter;

    char out_file[64];

    pthread_mutex_t ac_lock;
}account;
```

Since we do not care about overdraw, regardless of the order of processing the requests, we should end up with the same result for each account. In part 1, you will be implementing a

single threaded solution which produces an output file in the following format. (This is also the correct answer for the input file given). Keep track of transaction tracter.

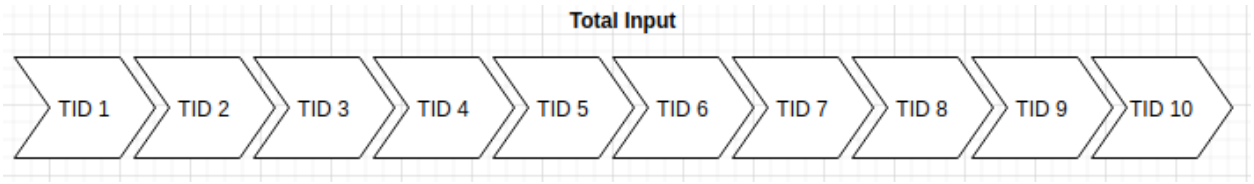
```
0 balance:  3111685.13
1 balance:  2016708.08
2 balance:  3248744.20
3 balance:  3889910.50
4 balance:  2164242.04
5 balance:  2119930.00
6 balance:  2206168.18
7 balance:  2306013.02
8 balance:  2788273.79
9 balance:  2011539.14
```

You are free to modify the header file and add any function you deem necessary to help you solve this problem. Even though part 1 will only be worth a very small portion of your grade, it is the foundation of your project. Part 1 also provides the correct solution for your end goal. Make sure you make every function correct and robust.

Part 2: Multi-thread solution with critical section protection

Now that you have completed part1, you are now ready to transform your solution to a thread-safe solution.

You have to first identify the critical sections in your code and utilize the `pthread_mutex_t` to ensure that only one thread can access the section at a time. Write `process_transaction` function, and use `pthread_create` to start all the worker threads (evenly slice the number transactions for each one of the worker threads to handle based on the number of workers (10) and number of requests). Write `update_balance` function, and use `pthread_create` to start the bank thread. In this part, your bank thread will only update the balance once all the worker threads have finished. Your main thread should not exit until all threads are finished and all dynamic memories are released. You should reach the same solution as part1. Each thread needs to process all of their assigned transactions before exiting.

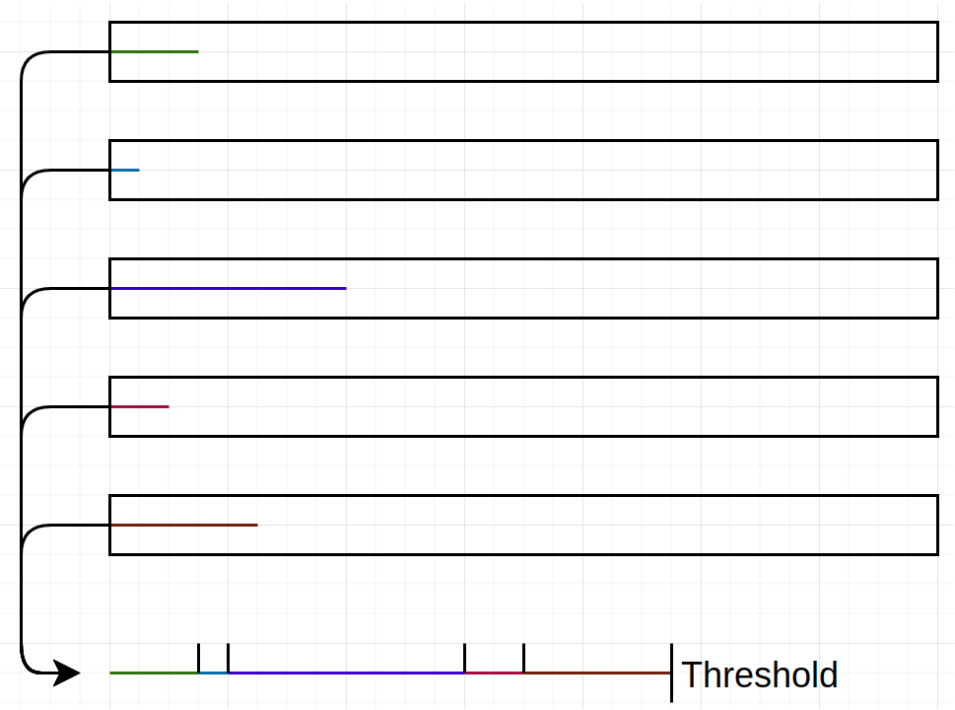


Part 3: Coordinating the threads to work together

In part2, you have completed a multithreaded solution to our problem set. However, no customers will be satisfied with a slow update. Therefore, we are upgrading our solution to update the reward regularly by using `pthread_barrier_wait & pthread_cond_wait` to have the worker threads communicate with the bank threads.

Here are the goals for part 3. First, no thread can start processing the requests before all threads are created and given a signal by main. Second, when the number of requests processed (excluding checking balance, and invalid request) across all threads reaches 5000, all threads have to pause and notify the bank thread to wake up and update the balance of each account balance. Third, when the bank is done updating the balances, it will append each account's balance to output files name after each account number (act_#.txt). The bank thread will then tell all other threads to continue processing before going back to a waiting state ready for the next round of updates.

The following diagram shows an example of five threads reaching the threshold (You will have 10 threads).



One of the indicators of the correct solution will output a different act_#.txt with same amount of line numbers every time, while the resulting balance remains the same.

```
account 2:
Current Balance:        4140001.91
Current Balance:        4004946.05
Current Balance:        3974129.41
Current Balance:        3935093.95
Current Balance:        3926891.98
Current Balance:        3854780.65
Current Balance:        3771173.10
Current Balance:        3658910.58
Current Balance:        3666098.07
Current Balance:        3639894.43
Current Balance:        3599369.34
Current Balance:        3549592.29
Current Balance:        3509038.76
Current Balance:        3467133.15
Current Balance:        3420467.68
Current Balance:        3367566.48
Current Balance:        3304670.96
Current Balance:        3248744.20
```

Example of account at index 2

Part 4: Transferring to another bank

Each user now also has savings account at puddles bank (represented by a separate process). So there are two banks total, each represented by a process – two processes total. When an account is initialized at duck bank, the information is shared with puddles bank, where they have a savings account. Each account will be "duplicated" at puddles bank, except their initial balance will be equal to 20% of their initial balance at duck bank, and everyone will have a flat reward rate of 2%. The duck bank process will write the account information to shared memory, and the puddles bank process will read from shared memory, you will use mmap() https://man7.org/linux/man-pages/man2/mmap.2.html.

Every time the banker thread from part3 applies interest to their duck bank account, the puddles process applies the puddles bank savings reward rate to their puddles bank account. The schedule for the banker thread will remain the same as part3. Balances at puddles bank are printed in the same manner as at duck bank but in a new directory called "savings", your account file names will be the same – see "account_7_example_output_savings.txt".

Remarks:

Race conditions are going to play a huge role while implementing part3. An important question you should ask yourself is how do you make sure one thread will reach a certain part of the code before another?

Deadlocks could make your program stuck, and it is extremely difficult to figure out exactly what happened, and how to resolve it. Think about what variables you could keep track of to signal a deadlock.

Project Structure Requirements:

For a project to be accepted, the project must contain the following files and meet the following requirements: (You must use the C programming language with the pthread library for this assignment. No projects written in another programming language will be accepted.)

bank.c: This is the main program.

Makefile: Your project must include a standard make file. It must produce the executable with the following names: **bank**

How to run: ./bank input.txt

What to output: An "output.txt" includes all the accounts final balance. An "output" directory contains all "account#.txt" which all accounts balance for each update. On the terminal, you should also print some useful information indicating the state of your program. Before the program exits, it should print the update times which should match the line number in each "account#.txt" (line number - 1).

```
bank 241519 waiting now, update time is 16
241509 is done
241517 number reached, pausing
241510 number reached, pausing
241514 number reached, pausing
241518 number reached, pausing
241516 number reached, pausing
241513 number reached, pausing
241512 number reached, pausing
241511 number reached, pausing
241515 number reached, pausing
bank 241519 signal received
bank 241519 waiting now, update time is 17
241513 is done
241515 is done
241518 is done
241511 is done
241512 is done
241516 is done
241510 is done
241517 is done
241514 is done
bank 241519 signal received
total updates 18
```

Report: Write a 1-2 page report on your project using the sample report collection format given. Feel free to go over the limit if you wish. Report format and content suggestions are given in the report collection template. If you are not able to complete all 3 parts, state in your report which part you finished, so partial credit can be given.

Note: Additionally, you are allowed to add any other *.h and *.c files you wish. However, when we run your code we will only be running the server file. Make sure your code runs in the VM before submission.

Submission Requirements:

Once your project is done, do the following:
Your executable should be able to run with the following command "./bank input.txt"

1. Open a terminal and navigate to the project folder. Compile your code in the VM with the -g, -pthread, and - lpthread flag.
2. Run your code and take screenshots of the output as necessary (of each part).
3. Create valgrind logs of each respective part:
    a. “**valgrind --leak-check=full --tool=memcheck ./a.out > log*.txt 2>&1** ”

4. Tar/zip the project folder which should contain the following directories and content. Your project should have part1, part2 and part3 folders.
    a. part1
        i. bank.c
        ii. Any additional header file and their corresponding ".c" file
        iii. Makefile
        iv. Output (directory)
        v. valgrind log
    b. part2
        i. bank.c
        ii. Any additional header file and their corresponding ".c" file
        iii. Makefile
        iv. Output (directory)
        v. valgrind log
    c. part3
        i. bank.c
        ii. Any additional header file and their corresponding ".c" file
        iii. Makefile
        iv. Output (directory)
        v. valgrind log
    d. par4
        i. bank.c
        ii. Any additional header file and their corresponding ".c" file
        iii. Makefile
        iv. Output (directory)
        v. Savings (directory)
        vi. valgrind log
    e. Report pdf

Valgrind can help you spot memory leaks in your code. As a general rule any time you allocate memory you must free it. Points will be deducted in both the labs and the project for memory leaks so it is important that you learn how to use and read Valgrind's output. See (https://valgrind.org/) for more details.

The naming convention of the zip/tar file while uploading to canvas UoID_duckID_LAB/ProjectX (an example is given below)

- firstname: alex
- lastname: summers
- image: ex. DebainUTM, UbuntuVbox, etc.
- Submission for: Project1
- So the name of the zip/tar file should be: alex_summers_DebainUTM_Project3.zip

Grading Rubric:

Parts	Points	Description
Part 1	10	10 the correct answer is reached
Part 2	20	5 All critical sections are well protected with each account lock 5 Correct usage of pthread_create according to the specification 5 Correctly return a value using pthread_join for bank thread 5 the correct answer is reached
Part 3	40	10 the correct answer is reached and “account#.txt” is different every run. However, “account#.txt” should have the same number of lines every run. 10 correct usage of pthread_barrier_wait 10 program does not deadlock 10 correct usage of pthread_cond_wait and pthread_cond_broadcast / pthread_cond_signal
Part 4	15	5 the correct answer is reached and “account#.txt” is different every run. However, “account#.txt” should have the same number of lines every run. 10 correct usage of mmap() munmap()
Valgrind	10	No memory leak/errors 1 point each until all 10 points are gone
README	5	1 - 2 page report

Note: Some sections may have more detail points than the total points, meaning there are more than 1 way you can get a 0 in that section.

1. 0/100 if your program does not compile.
2. 10 points deduction if your makefile does not work.

3. 30 points deduction for part3, if pthread_barrier_wait, pthread_cond_wait, pthread_cond_broadcast / pthread_cond_signal are used but did not contribute to the actual functionality.
4. Missing functionality caused by chain effects will not receive credit. (correctly implemented but does not work due to other mistakes)

Late Homework Policy:

- 5% penalty (1 day late)
- 10% penalty (2 days late)
- 20% penalty (3 days late)
- 100% penalty (>3 days late) (i.e. no points will be given to homework received after 3 days)