PRACTICE 3

Sheila Gallardo Redondo
NIA: 100472913
Mail: 100472913@alumnos.uc3m.es
Group: 121

Alexandra Perruchot-Triboulet Rodríguez
NIA: 100453024
Mail: 100453024@alumnos.uc3m.es
Group: 121

Contents

Description of the Code

bank.c

Introduction

This script has 3 main parts: the main, the ATM thread's function (ATM_function) and the worker's thread's function (worker_function). During this lab, we have been controlling the concurrency and synchronization of the threads, implementing mutexes in the thread's functions.

Global Variables

We have created 5 global variables (declared outside the main function) to keep track of their values within the program and so every thread can access them independently. The first one is *client_numop*, it stores the number of operations obtained from the *list_client_ops* array and stored in the circular buffer, it's incremented by the ATMs. Then, we have *bank_numop*, it's like the previous one, but it stores the number of operations performed, (incremented each time a worker takes an operation from the circular buffer and it's about to perform it). Furthermore, we have the *global_balance*, which stores the total balance of the bank, modified when operations like: "Deposit" or "Withdraw" are performed. The *account_balance*, which is a vector of integers, storing the balance of each account (the account ID corresponds to the position of the vector) and it has been initialized to -1 in the main function. Finally, *list_client_ops*, which is an array of struct element* elements, which correspond to the operations to be executed and we have allocated its necessary memory space (using *malloc*) in the main function.

main

In this part of the program, we have extracted the inputs passed as arguments: the *max_accounts* (argv[4]), which corresponds to the maximum number of accounts allowed by the bank and has been used to allocate memory for the account_balance vector. Then, we have obtained the number of ATMs (*num_ATMs*) and the number of workers (*num_workers*), from argv[2] and argv[3] respectively, this has been used to determine how many ATM threads and Worker threads must be created. Also, we have extracted the circular buffer size (its capacity) from argv[5] and we have use it to initialize the circular buffer with the

queue_init function from the queue.c script. Finally, we extracted the file's name from the first input (argv[1]) and used it to open and read the file.

In the process of reading the file, we looked at the first line of the file and stored its content (number of operations) in the array *num,* then, we transform it into an integer (using *atoi*) and store it in *max_operations.* We check that this number doesn't exceed 200, and if so, we continue reading the file. Once we have the *max_operations* we implemented a for loop to read each line from the file (which corresponds to each operation that needs to be performed). Inside the loop, we have created a struct element* variable (*op*) which will be used to store each operation. We declare the variable *j* and initialize to zero (used to keep track of each element of the line, which are the operators of the operation). Then, we implement a while loop, which reads the file until we find a '\n' . Inside this loop, we use '(blank space)' to separate each operator, until we find one, we store the operator we are reading in *vect* (vector of characters). When we find a blank space, we look at the value of *j* to check in which attribute of the structure *op* we should store the operator (in case of j =2, we store the value stored in *vect* in an auxiliary vector *x*). Once outside the while loop (still in the for loop), we check the value of *n* (the number of bytes read from file), if it's zero, it means that there are less lines in the file (less operations) than the number in *max_operations*, then, there's an error. We also look at the value of *j* to check where to store the last operator of the line (if j = 3, we store the value of vect in the *money* attribute of *op* and the value stored previously in *x* in the *account_ID2* attribute of *op*). Moreover, we store the operation number (*operation_num* attribute) which corresponds to the index of the for loop, we store the structure in the corresponding position of the *list_client_ops* array and we reset *op* to store next operation (next line). Finally, once outside the for loop, we check if there's any other operation in the file (reading the next character), and if so, we send an error message.

The next step is to create the ATM and worker threads. To do that, we implement a for loop, to create *num_ATMs* threads and *num_workers* threads. The corresponding function for the ATMs (*ATM_function*) receives as arguments the *circular buffer*, the *list_client_ops* and *max_operations*, which we have stored in *args_atm* so that they can be passed to the function later. For the workers function (*worker_function*), we send as arguments the *circular buffer*, *max_operations*, *max_accounts*, which we have stored in *args_worker* to then pass them to the function. Furthermore, we initialize the mutex (*mut_buffer*).

Finally, we wait for all ATM threads and worker threads to finish execution, we destroy the circular buffer (with the *queue_destroy* function) and the mutex (*mut_buffer*) to release the memory space used, and free the resources (*list_client_ops* and *account_balance*).

ATM_function

In this function, we first extract the inputs from *args_atm* and store them in the variables: *circular_buffer, list_client_ops* and *max_op,* which we have created inside the function in

order to store the circular buffer (where we are going to store the operations), the list_client_ops array (where we have all operations from the file stored as struct element* elements) and max_operations (which is the number from the first line of the file indicating the number of operations to be performed).
Once we have all the variables ready, we implement a while loop, to insert all elements from the *list_client_ops* array into the *circular_buffer* until we have a value of *client_numop* greater or equal than *max_op*. When one thread enters the loop, we lock the mutex (*mut_buffer*) so that no other thread can access the circular buffer or modify global variables at the same time. Moreover, we check if the buffer is full or not (with *queue_full* function), if it's not, then, we can add an element (with *queue_put* function), but first we must check if the element we are going to insert has its operation number (*operation_num*) equal to the current value of *client_numop*, this condition allows the ATMs to store the operations in order. If that is the case, we insert the element in the circular buffer with the *queue_put* function, we add one to the *client_numop* value and we unlock the mutex so that another thread can access the circular_buffer.
When the value of *client_numop* is greater or equal to the maximum number of operations to be performed, we end the execution.

worker_function

In this other function, we first extract all inputs passed as arguments and store them in variables (just like we did in the *ATM_function*). We store the *max_accounts* (number of maximum accounts the bank has) in the variable *max_acc*.
Once we have all our variables, we perform a while loop to extract all elements from the circular buffer and execute them, until we have a value of *bank_numop* greater or equal than *max_op*. When one thread enters the loop, we lock the mutex (*mut_buffer*) so that no other thread can access the circular buffer or modify global variables at the same time.
Furthermore, we check if the buffer is empty or not (with *queue_empty* function), if it's not, then we can extract an element from it (using *queue_get* function) and store it in an struct element* variable: *element_buffer.* Then, we must check that the order of the operations is correct. We do that by comparing the current *bank_numop* value with the corresponding *element_buffer's operation_num,* if they are the same, then, order is correct, and we can continue with the execution. Moreover, we start looking which operation we must perform by comparing the *element_buffer's operation_name* with the names of possible operations ('CREATE', 'DEPOSIT', 'WITHDRAW', 'BALANCE', 'TRANSFER').
If the operation is '**CREATE**', we first need to check that the *max_op* accounts permitted is not exceeded, so that we don't create more accounts than the ones allowed by the bank. We also check that the account hasn´t been created yet by looking at the number stored in *account_balance* at the position corresponding to that account ID. If this number is equal to -1, then, we can create the account.
We set the *account_balance* of this account to zero and the *global_balance* remains unchanged.

If the operation is '**DEPOSIT**', we first check if the account has been created, if so, we add the amount of money stored in the *element_buffer's money* attribute with the amount stored in the *account_balance* of that account. Then, we update the *global_balance* by adding the *element_buffer's money* attribute to it.
If the operation is '**WITHDRAW**', we check if the account has been created, and if it's the case, we remove the amount of money stored in the *element_buffer's money* attribute to the amount stored in the *account_balance* of that account.
Then, we update the *global_balance* by subtracting the *element_buffer's money* attribute from it.
If the operation is '**BALANCE**', we check if the account has been created, and if so, we simply print as in the previous cases the output:
<operation_num> <BALANCE> <Account_ID1> <BALANCE= €> <TOTAL= *global_balance*>.
Finally, if the operation is '**TRANSFER**', we check if both accounts have been created, and if it's the case, we add the amount of money stored in the *element_buffer's money* attribute to the amount stored in the *account_balance* of *account ID2* and remove it from the amount of money stored in the account_balance of *account ID1*. The *global_balance* remains the same.
Once we have perform one of these operations, we add one to the value of *bank_numop*.
When the value of *bank_numop* is greater or equal to the maximum number of operations to be performed, we end the execution.

queue.c

Functions

This script has 6 functions: *queue_init*, *queue_put*, *queue_get*, *queue_empty*, *queue_full*, and *queue_destroy*.

- *queue_init* function initializes a circular buffer. It takes as input an integer (*size*) which will be the capacity of the buffer and returns a pointer to the circular buffer.
- *queue_put* function inserts an element in the circular buffer. It takes as input a circular buffer (a *queue*) and a struct element* element and returns an integer. If the returned integer is zero, then, the element has been inserted successfully.
- *queue_get* function extracts elements from the circular buffer. It takes as input a circular buffer (a *queue*) and returns a struct element* element.
- *queue_empty* function tells whether the circular buffer is empty or not. It takes as input a circular buffer and returns an integer. If the returned integer is 1, then, the buffer is empty, but, if the returned integer is 0, then, the buffer is not empty.

- *queue_full* function tells whether the circular buffer is full or not. It takes as input a circular buffer and returns an integer. If the returned integer is 1, then, the buffer is full, but, if the returned integer is 0, then, the buffer is not full.
- *queue_destroy* function destroys a circular buffer and releases its resources. It takes as input a circular buffer and returns an integer. If the returned integer is zero, then, the circular buffer has been destroyed successfully.

queue.h

Structures

This script has 2 data structures and will then be imported to the *queue.c* script to be able to use them. These two structures are: struct *element* and struct *queue*.

- *struct element* is the data type of the elements we will insert in the circular buffer. The operations of the bank are stored as *struct elements.* Its attributes are *operation_num*, *operation_name*, *account_ID1*, *account_ID2*, and *money*.
- *struct queue* is the data type of the circular buffer. Its attributes are *size* (actual size of the buffer), *capacity* (the maximum capacity of the buffer), **items* (addresses of the elements stored in the buffer), *head* (index of the first element from the buffer), and *tail* (index of the last element from the buffer).

Test Cases

Base Case: ./bank file.txt 2 2 10 30

File with 20 as number of operations (max_operations = 20) and 20 operations.

- **Expected output:**

1 CREATE 1 BALANCE=0 TOTAL=0
2 DEPOSIT 1 100 BALANCE=100 TOTAL=100
3 CREATE 2 BALANCE=0 TOTAL=100
4 DEPOSIT 2 50 BALANCE=50 TOTAL=150
5 TRANSFER 1 2 30 BALANCE=80 TOTAL=150
6 TRANSFER 2 1 20 BALANCE=90 TOTAL=150
7 WITHDRAW 1 40 BALANCE=50 TOTAL=110
8 WITHDRAW 2 60 BALANCE=0 TOTAL=50
9 BALANCE 1 BALANCE=50 TOTAL=50
10 BALANCE 2 BALANCE=0 TOTAL=50
11 CREATE 3 BALANCE=0 TOTAL=50
12 DEPOSIT 3 20 BALANCE=20 TOTAL=70
13 CREATE 4 BALANCE=0 TOTAL=70
14 DEPOSIT 4 500 BALANCE=500 TOTAL=570
15 TRANSFER 3 4 10 BALANCE=510 TOTAL=570
16 TRANSFER 4 3 200 BALANCE=210 TOTAL=570
17 WITHDRAW 3 5 BALANCE=205 TOTAL=565
18 WITHDRAW 4 100 BALANCE=210 TOTAL=465
19 BALANCE 3 BALANCE=205 TOTAL=465
20 BALANCE 4 BALANCE=210 TOTAL=465

- **Actual output:**

1 CREATE 1 BALANCE=0 TOTAL=0
2 DEPOSIT 1 100 BALANCE=100 TOTAL=100
3 CREATE 2 BALANCE=0 TOTAL=100
4 DEPOSIT 2 50 BALANCE=50 TOTAL=150
5 TRANSFER 1 2 30 BALANCE=80 TOTAL=150
6 TRANSFER 2 1 20 BALANCE=90 TOTAL=150
7 WITHDRAW 1 40 BALANCE=50 TOTAL=110
8 WITHDRAW 2 60 BALANCE=0 TOTAL=50
9 BALANCE 1 BALANCE=50 TOTAL=50
10 BALANCE 2 BALANCE=0 TOTAL=50
11 CREATE 3 BALANCE=0 TOTAL=50

12 DEPOSIT 3 20 BALANCE=20 TOTAL=70
13 CREATE 4 BALANCE=0 TOTAL=70
14 DEPOSIT 4 500 BALANCE=500 TOTAL=570
15 TRANSFER 3 4 10 BALANCE=510 TOTAL=570
16 TRANSFER 4 3 200 BALANCE=210 TOTAL=570
17 WITHDRAW 3 5 BALANCE=205 TOTAL=565
18 WITHDRAW 4 100 BALANCE=210 TOTAL=465
19 BALANCE 3 BALANCE=205 TOTAL=465
20 BALANCE 4 BALANCE=210 TOTAL=465

Invalid number of ATMs: ./bank file.txt -3 2 10 10

- **Expected output:** "Error: invalid number of ATMs: Success" (Return value -1)
- **Actual output:** "Error: invalid number of ATMs: Success" (Return value -1)

Invalid number of workers: ./bank file.txt 1 0 10 10

- **Expected output:** "Error: invalid number of workers: Success" (Return value -1)
- **Actual output:** "Error: invalid number of workers: Success" (Return value -1)

Less operations than the number of operations: ./bank file.txt 3 2 10 10
File with 10 as operation number and 3 operations.

- **Expected output:** "Error: too few operations for the given number: Success" (Return value -1)
- **Actual output:** "Error: too few operations for the given number: Success" (Return value -1)

More operations than the number of operations: ./bank file.txt 3 2 10 10
File with 3 as operation number and 6 operations.

- **Expected output:** "Error: more operations than expected: Success" (Return value -1)
- **Actual output:** "Error: more operations than expected: Success" (Return value -1)

More than 200 operations: ./bank file.txt 3 2 10 10
File with 250 operations.

- **Expected output:** "Error: number of operations out of range: Success" (Return value -1)
- **Actual output:** "Error: number of operations out of range: Success" (Return value -1)

Too many arguments: ./bank file.txt 3 2 10 10 3

- **Expected output:** "Too many arguments: Success" (Return value -1)
- **Actual output:** "Too many arguments: Success" (Return value -1)

Too few arguments: ./bank file.txt 3 2 10

- **Expected output:** "Too few arguments: Success" (Return value -1)
- **Actual output:** "Too few arguments: Success" (Return value -1)

Some operations cannot be performed: ./bank file.txt 6 2 5 5

File with 20 operations.

- **Expected output:**

"Error: Account hasn't been created yet: Success"
2 CREATE 1 BALANCE=0 TOTAL=0
3 CREATE 2 BALANCE=0 TOTAL=0
4 DEPOSIT 2 50 BALANCE=50 TOTAL=50
5 TRANSFER 2 1 20 BALANCE=20 TOTAL=50
6 WITHDRAW 1 40 BALANCE=-20 TOTAL=10
7 BALANCE 2 BALANCE=30 TOTAL=10
8 CREATE 3 BALANCE=0 TOTAL=10
9 DEPOSIT 3 20 BALANCE=20 TOTAL=30
10 CREATE 4 BALANCE=0 TOTAL=30
11 DEPOSIT 4 500 BALANCE=500 TOTAL=530
12 TRANSFER 3 4 10 BALANCE=510 TOTAL=530
13 WITHDRAW 3 5 BALANCE=5 TOTAL=525
14 WITHDRAW 4 100 BALANCE=410 TOTAL=425
15 BALANCE 3 BALANCE=5 TOTAL=425
16 BALANCE 4 BALANCE=410 TOTAL=425
Error: Account hasn't been created yet: Success
18 CREATE 5 BALANCE=0 TOTAL=425
Error: exceeded the maximum number of accounts: Success
Error: Account already created: Success

- **Actual output:**

Error: Account hasn't been created yet: Success
2 CREATE 1 BALANCE=0 TOTAL=0
3 CREATE 2 BALANCE=0 TOTAL=0

4 DEPOSIT 2 50 BALANCE=50 TOTAL=50
5 TRANSFER 2 1 20 BALANCE=20 TOTAL=50
6 WITHDRAW 1 40 BALANCE=-20 TOTAL=10
7 BALANCE 2 BALANCE=30 TOTAL=10
8 CREATE 3 BALANCE=0 TOTAL=10
9 DEPOSIT 3 20 BALANCE=20 TOTAL=30
10 CREATE 4 BALANCE=0 TOTAL=30
11 DEPOSIT 4 500 BALANCE=500 TOTAL=530
12 TRANSFER 3 4 10 BALANCE=510 TOTAL=530
13 WITHDRAW 3 5 BALANCE=5 TOTAL=525
14 WITHDRAW 4 100 BALANCE=410 TOTAL=425
15 BALANCE 3 BALANCE=5 TOTAL=425
16 BALANCE 4 BALANCE=410 TOTAL=425
Error: Account hasn't been created yet: Success
18 CREATE 5 BALANCE=0 TOTAL=425
Error: exceeded the maximum number of accounts: Success
Error: Account already created: Success

Conclusions and Errors encountered

During this Lab we have experienced some trouble. The first one was while reading the file, specifically storing the words that we were reading in the corresponding attribute of *structure element* op*. Since the number of parameters of each type of operation is not the same, we had to create an auxiliar variable (*x*) to assign the right parameter to the correct attribute later.
Moreover, we have experienced some problems passing arguments to the ATMs and the workers functions (threads). We tried to pass the arguments directly through the thread but to do that we had to create a vector with all the arguments instead. Then, when we tried to extract the arguments from the vector of arguments inside the ATMs and the workers function, we didn't do it properly and it causes a lot of problems while implementing the functions.
Another problem was with the mutex. We implemented it outside the main while loop and it causes that two or more threads cannot work at the same time. Finally, we implemented the mutex just before and after entering to the critical sections (*cicular_buffer* and global variables), then the threads do their jobs simultaneously without compromising the critical sections.
The last problem we had during this lab was checking if an account was already created. To solved that, we decided to fix -1 as the value of an empty position in the *account_balance* vector (the corresponding account hasn't been created yet). With that, we initialized the vector with -1 in every position and change the value to 0 when an account is created. We tried first assigning NULL to each position of the vector, but we realised that we cannot do that with an integer vector.

In our opinion, we found the work very useful to be able to see first-hand how to work with multiple threads. In addition, we have also learned how to work with thread concurrency, and it was very helpful to understand the theory about it.