LAB ASSIGNMENT 3

-

Iván Miguélez García - 100383387

100383387@alumnos.uc3m.es

José Manuel Frías Salvachúa - 100383533

100383533@alumnos.uc3m.es

TABLE OF CONTENTS

1. Introduction
2. Description of the main methods
   1. Broker method
   2. Operation executer method
   3. Stats reader method
   4. Market
3. Control mechanisms
   1. Initialize and destroy concurrent mechanisms
   2. Concurrency mechanism between broker and executer
   3. Check exit method
4. Tests
   1. Multiple brokers, reader and 1 executer
   2. Inexisting batch operations txt file
   3. Bad format in batch operations txt file
   4. Invalid stocks txt file
5. Conclusions and main problems found

**INTRODUCTION**

The market we are presented can be viewed as a variation of the producer-consumer problem. A proper use of the mutex and condition signals to manage the different threads (each one a method that can operate in the market) will be key to proper functioning. In the following points, there will be a more extensive explanation of how the different methods (threads) work avoiding concurrency problems between them.

**DESCRIPTION OF THE CODE**

After extracting the information from the argument in each method, they proceed to work as follows:

* Broker: the *broker* method reads operations from a file and put them into a queue whenever possible).First we extract information passed as argument pointer. We will also create an iterator that will help in the extraction of operations passed in the operations txt file. While we can read new operations from the file, we parse their info a nd create a new operation ready to enqueue. Before adding an operation to the queue it checks whether there are any operations to add (for this is useful to use the function operations\_queue\_full), and if the queue admits more operations to be added. If it is full we will execute a wait until we receive a signal from the executer thread telling us that we can add more things. If an operation is added, then a signal is sent to the operation executer, saying that there are operations in the queue that can be processed. If there are no operations left to add, the iterator in charge of parsing the operations in the text file is deleted and the method finishes.
* Operation executer: This method is in charge of pulling and processing the operations of the queue. A flag called *exit* is in charge of telling the method that the broker method(s) are done. This flag can only be accessed when the mutex *exit\_mutex* is locked. As *exit* is a condition variable inside a while loop, that iterates whenever the flag is not activated, we decided to implement an auxiliary method which is in charge of telling the while loop if the next iteration should be done (this method is *check exit* and will be explained later). In the same fashion as before, we must first lock the mutex and check if there is any operation in the queue. If there is we will proceed to dequeue it and process it ( a malloc is applied due to a problem we had with previous implementations where the operations became mixed one with the other, thus, changing the attributes). after processing, we will unlock.

Once the *exit* flag is activated, *check exit* tells the while loop to stop iterating and the rest of the operations inside the queue are processed (the same way as explained before but without the need of the concurrency mechanisms) until the queue is empty.

* Stats reader: the method *stats reader,* in charge of printing the status of the market every n microseconds (the frequency), is also controlled by the *exit*  flag. As in the *operation executer*  method, the while loop is controlled by the *check exit* method, so whenever the exit flag is activated, the method finishes. Every n miliseconds, the reader will try to enter in the market (thus stopping the brokers and the executer from changing the market) and print it’s status. after printing, it will free the broker and executer until next time.
* Market: First, we indicate the number of threads we are going to create and initialize the market and the exit variable. While initializing the market we also initialize the concurrency mechanisms we are going to use on the threads. Next initialize all the distinct components (brokers, executer and readers) and create the threads that will execute each of them. Create a for loop in charge of joining the broker threads. Once this is finished we can activate the exit variable that will tell the readers to stop reading and the executer that all brokers have ended (so itś free to dequeue the rest of operations without fear of concurrency issues). We join the rest of the threads with a for loop. at last, we print the status of the market, delete it and destroy the concurrency mechanisms.

**CONTROL MECHANISMS**

* Init concurrent mechanism and destroy concurrent mechanism: both methods are in charge of managing the mutex *enqueue\_mutex* used for the producer-consumer mechanism and the condition signals, which are not full and not empty. At the beginning of the concurrent market file, this mutex are initialized, and once all the threads finish, the mutex are destroyed.
* Check exit method: this method is used to properly check that the *exit mutex* is locked once the *exit* flag is check. The method locks *exit mutex* and reads the flag. If *exit*  is not activated, the mutex is unlocked and the method returns true. Otherwise, the method is unlocked and the method returns false.
* Concurrency Mechanisms: We have applied 1 mutex for controlling access to the queue and 2 conditions for controlling when the queue is empty or full. The mutex lock is always applied before checking the conditions and unlock after the operation is enqueue/dequeue. The protects the shared area from collisions (which will avoid segmentation faults). If the queue is empty, operation executer is locked and waiting for a signal from the brokers. In the other hand, when the queue is full, brokers get locked until we received a signal from the executer telling us that the queue is full no more.
* Check exit: We apply this function in the condition of the while loops. This simply applies the lock/unlock methods for checking the variable exit without problems of concurrency.

**TESTS**

* Valid input with multiple brokers, 1 operation executerand multiple readers: Checks the correct functioning of the concurrency mechanisms, as well as, the correct functioning of the methods (well applying of all the operations written in the batch files). After executing it multiple times we found that the final result it’s always the same and follows the operations we have defined.
* Inexistent batch operation txt: An error should appear telling the user that it couldn’t open the batch file and program should finished. The error appears and program finishes
* Bad format of operations: An error should appear when executer tries to process it. It should specify what is the first part of the format that is invalid. We were informed that sdsgh wasn’t a valid ID in our stock market
* Non existing stock market file: Market shouldn’t initiate and when trying to process an operation we should receive an exception and program should end. Program ended with a floating point exception

**CONCLUSION**

Practice was an interesting way of exercising the use of mutex and learning about this type of problems (producer-consumer). It wasn’t too difficult but some parts force us to think once or twice how we had to apply it and force redesign of the solution.

Our main problem appeared before the use of malloc. Before applying this function our operations got mixed one with each other when applying multiple brokers due to the shared memory even though our concurrency mechanisms worked well. This delayed a lo our work and kept us stuck for days without any advance. But, once this was solved everything worked well.