Universidad Carlos III de Madrid

OPERATING SYSTEMS P3

Lab 3 – Stock market functionality

GROUP 88

AUTHORS:

Rafael Pablos Sarabia (100372175) 100372175@alumnos.uc3m.es

Juan Sánchez Esquivel (100383422) 100383422@alumnos.uc3m.es

Table of Contents

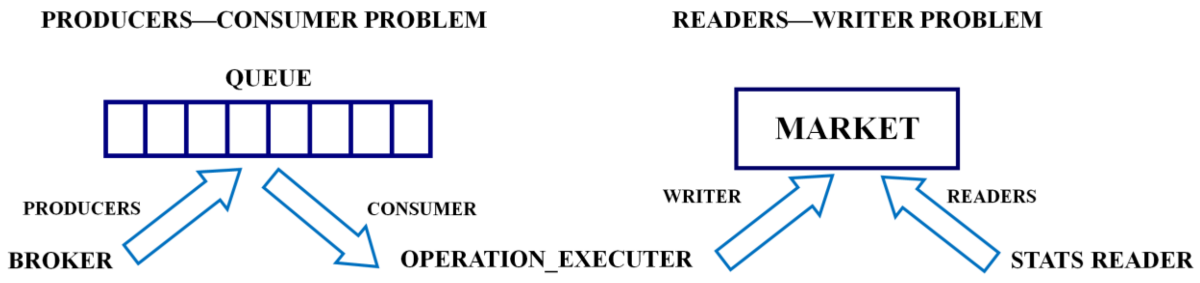
1. Introduction

The purpose of this lab assignment is to fully understand the management of threads with concurrency mechanisms, how to implement mutexes and condition variables and determine critical areas of code that should not be shared. The main objective of this assignment is focused on creating a good solution, by design, that will behave always as expected and not lead to race conditions nor scheduling assumptions.

To do so we will code the `concurrency_layer.c` file which will handle concurrency on a given stock_market. This file also has a given library in which a definition for all the functions we must have implemented are defined. We have also modified `concurrent_market.c` to be able to perform tests quickly by setting the number of threads of each type and storing the input files in a given directory. This way, we do not need to manually handle each thread or change the name of the input files in the code.

2. Design

In order to have a valid implementation for the concurrency layer, which did not depend upon race conditions or scheduling policies, we had to design a good solution which considered critical sections. After analyzing the statement and taking notes on the key thing to consider, we realized the problem was a similar to a combination of the two types of concurrency problems we had studies in class: producer-consumer and readers-writers. The following schema summarizes this idea.



Then, using the examples from class we had to think about the needed resources to control the critical sections of our program. Since the statement specifies that only one thread type is executing simultaneously (explained in the concurrency requirements section), a mutex called `accesses` will be used for controlling the type of thread currently executing (broker, executer or reader). Also, a global variable `stats_readers` will be used to keep track of the simultaneous readers and it will be regulated by mutex `readers_mutex` so that only one reader is active when executing some parts of the code. Finally, two condition variables (`broker_write` and `executer_write`) are used to control the state of the queue and whether broker or executer can be active (broker can be active if queue is not full and executer if queue is not empty). In the implementation section, the implemented functions will be explained in great detail paying special attention to the concurrency techniques.

Concurrency control mechanisms:

- *Mutex accesses:* controls that only one broker, the executer or readers are active. Therefore, it is locked when the first broker, the executer or the first reader is activated. It is unlocked when the broker enqueues the operation, the executer processes the operation or the last current reader is slept.
- *Mutex readers_mutex*: controls that only one reader is modifying the variable stats_readers and deciding whether to lock or unlock mutex accesses.

- *Condition broker_write*: notifies the broker that it can execute. This signal is activated when the queue is not full because the executer has just processed an operation or when the last reader is slept.
- *Condition executer_write*: notifies the executer that it can execute. This signal is activated when the queue is not empty because the broker has just enqueued an operation or when the last reader is slept.

3. Implementation

Before coding the functions, we included the library `concurrency_layer.h` and declared the integer variable `stats_readers` in which the number of simultaneous readers will be stored. Also, mutex `accesses` and mutex `readers_mutex` are declared, they will control current thread type executing and shared variable `stats_readers` respectively. Lastly, thread condition variables `broker_write` and `executer_write` are declared, where `broker_write` will allow broker to execute if the queue is not full while `executer_write` will allow the executer to be active if queue is not empty.

void init_concurrency_mechanisms()

As the function name states, we initialize all mutexes and conditions to NULL so that default attributes are used. Also, `stats_readers` is initially set to 0 as there are no readers initially executing.

void destroy_concurrency_mechanisms()

As the name suggest, we destroy all mutexes and conditions. This function will be called just before ending the execution, in order to free memory and resources.

void broker(void * args)

This function will implement the functionality of broker threads with the needed concurrency control mechanisms.

First, we obtain the desired information from args by casting the argument to the struct type broker_info. Then, an iterator for the batch file passed in the structure is created. After doing so, we declare variables `id, type, num_shares` and `price` that will be later used to add the operation to the operations queue. Also, some memory is allocated for the operation to be stored in.

Secondly, we will execute a while loop that will keep looping until there are no more operations to be stored in the queue. If there are still operations to be enqueued (we will be inside the while loop) it will store the values inside the declared variables (`id, type, num_shares` and `price`) and it will create a `new_operation` with the variables and store it in the memory area referenced by `op`. Then we will lock the `accesses` mutex to guarantee that no thread will be executing when the next lines are executed by the current broker. In case the queue is full, the thread will block until it receives the signal `broker_write`, indicating that the queue is no longer full and broker can be executed. Once the broker can execute, we will enqueue our operation, signal that the queue is not empty and that the executer can operate, and we unlock the `accesses` mutex as we end the critical section and we want to allow other threads to execute.

Lastly, before ending the function, we will free and destroyed the used resources as they will no longer be used. This means we free the memory allocated for the operations and the iterator is destroyed.

void operation_executer(void * args)

This function implements the functionality of the thread that processes the operations from the queue, including the concurrency mechanisms for control.

To begin, as we did on the previous function, we will obtain and store the desired information from `args` in variables (`exec_info` `param` and `operations_queue * queue`). We will also allocate memory for the operations obtained from the queue. After doing so, we will declare int variable `exitFlag` that will be updated in each loop and store whether brokers have finished enqueueing or not. Then, we will lock `exit_mutex` from the `param` variable previously obtained, store the value into the `exitFlag` variable, and finally we unlock the mutex for other to use the shared variable exit. We have to lock and unlock the mutex to obtain the data in order to preserve data integrity.

Next, we will execute a while loop that will keep looping until the queue is empty and `exitFlag` is equal 1 (which will mean that brokers have finished). If the queue is empty, the value of `exitFlag` will be updated. In the case of having the queue empty and the flag active, the thread will be exited. Otherwise, the thread will keep waiting for the `executer_write` signal notifying that the queue is no longer empty.

When the conditions are met, we will dequeue an operation, store it in our operation variable and process it. Then, we will signal that the queue is not full as we will have dequeued one operation, unblock since we have reached the end of the critical section, update the `exitFlag` value and continue in the loop.

To end, before exiting the thread, we will free the used resources as they will no longer be used.

void stats_reader(void * args)

This function implements the functionality of threads for readers and their concurrency control mechanisms.

Firstly, we will obtain the desired information from `args` and store into a variable (`reader_info param`). After doing so and similarly to the previous function, we will declare variable `exitFlag` that will be updated before each loop, which will store whether we have to exit or not. As we previously did, every time we want to obtain the `exitFlag` value we will lock its mutex (located inside `param` variable), read and then unlock the mutex.

Next, we will execute a while loop that will keep looping until `exitFlag` is activated (value equal to 1). Inside this loop, we will increase the number of current readers by one. If it is the first reader, lock mutex `accesses` so that brokers and executer are blocked. These operations are executed in a critical section controlled by mutex `readers_mutex` because only one reader should be executing this part of code to prevent concurrency issues. Then, once the reader is added, print the market statistics using the provided method. After the reader has performed its action, decrease number of current readers. If current readers are 0, signal to broker and executer so that the y can operate and unlock the mutex `accesses`. These actions are again performed under the control of `readers_mutex` to prevent concurrency problems. The thread is then slept for a given time and the exit flag updated before the next loop iteration.

Lastly, we will exit the thread once `exitFlag` has been activated and the loop finished.

4. Test cases

This section focuses on executing different tests to try to verify that the implementation is correct and complies with the statement. In order to ease this process, concurrent_market.c was modified to allow automatic testing by only modifying the four declared variables at the top. For each test, the input files `<testID>_stocks.txt` and `<testID>_batch_operations_<brokerID>.txt` have to be created under ./tests/ with `brokerID` starting at 1. The four variables to modify represent the test ID and the number of threads for each type of agent involved (brokers, operations_executers and stats_readers). The main focus of the tests is to verify a correct implementation by design, so the same test was executed multiple times to verify constant output. Output was also verified for every test. After successfully passing all these tests, we believe our program is well designed and developed. For the tests, we attempt to verify the correct design and implementation of `concurreny_layer.c`.

Test ID: 1

- *Description*: basic test with the provided data. Verifies final market state is correct and execution logic is valid.

- *Input:* one thread for broker and another one for operations executer. batch_operations file contains 6 transactions for different companies. Initial stock market has 26 companies.

- *Output (expected and obtained):* initially, the market statistics will be shown after adding each company with the respective shares. The 26 companies should be shown, the total value of the market should be increased, and the average value properly updated. Next, only the broker thread can execute, enqueueing operations since the executer is blocked due to an initially empty queue. At some point, the executer thread will inlock, blocking the broker thread, and start performing the updates on the market after dequeuing from the operation queue. It will never deque more operations that present on the queue. If queue becomes empty and the broker is active, it will block and allow the broker to keep adding operations to the queue. Otherwise, the executer thread will exit and program will end, showing the final state of the market. The final state of the market is constant for multiple executions of the program and is valid according to the initial state and operations performed. The stats market have also been updated correctly, and the number of enqueued and dequeued operations corresponds to the number of operations in the batch file.

Test ID: 2

- *Description*: test with similar data as before but with more operations in batch_file so that we can verify that the broker thread executes for a while and enqueues some operations and then the executer thread dequeues them and updates the market. Also, verifies final market state is correct and execution logic valid.

- *Input:* one thread for broker and another one for operations executer. batch_operations file contains 60 transactions for different companies. Initial stock market has 3 companies.

- *Output (expected and obtained):* the output is the same as before, but since there are more operations it is easier to verify that the broker thread executes for a while and when it stops, the executer thread starts operating. The market is correctly updated and final result is constant and valid for several trials.

Test ID: 3

- *Description*: test with similar data as before but with only one operation which is useless and not valid because it buys 0 shares at cost 0 from a company which does not exist.

- *Input:* one thread for broker and another one for operations executer. batch_operations file contains only an invalid and useless transaction as explained before. Initial stock market has 3 companies.

- *Output (expected and obtained):* broker adds to the queue, the operation executer dequeues and attempts to perform operation but error message is displayed notifying that the stock with the given ID was not found. Market statistics are valid and program correctly terminated.

Test ID: 4

- *Description*: test to verify correct execution with multiple brokers, specifically 3.

- *Input:* one thread for each broker and another one for operations executer. batch_operations file will be created for each broker with the previously explained format. Each broker will have the 6 transactions, involving buy and sell. The initial market will contain 10 companies with different stock.

- *Output (expected and obtained):* brokers add to the queue (one at a time) and the operation executer dequeues and performs the operations. Market final state and statistics are valid and program correctly terminated. The execution seems valid as the brokers add when the queue is not full and the executer is not operating. Similarly, the executer operates when no broker is active and queue is not empty.

Test ID: 5

- *Description*: similar test to test 1, which uses the provided data but has a stats_reader thread.

- *Input:* one thread for each role (broker, operations_executer and stats_reader). As explained, the provided files have a broker with 6 transactions and a stock market with 26 companies. The reader will have a frequency in the range of 10,000 to 100,000 microseconds.

- *Output (expected and obtained):* brokers add to the queue (one at a time) and the operation executer dequeues and performs the operations. Market final state and statistics are valid and program correctly terminated just like for Test 1. However, we did not observe any difference with respect to test 1 after adding a reader thread. This is due to the fact that the exit flag is activated before the reader thread is executed. The functionality of readers will need to be tested on test 6, where more brokers are added.

Test ID: 6

- *Description*: test to verify the correct implementation for stats_readers after the results of test 5. More brokers will be added so that the reader threads can execute. The input files used will be the ones from test 2.

- *Input:* one thread for each role (broker, operations_executer and stats_reader). As explained in test 2, the provided files have a broker with 60 transactions and a stock market with 3 companies. The reader will have a frequency in the range of 10,000 to 100,000 microseconds.

- *Output (expected and obtained):* we can verify that the output is the same as for test 2, but adding the reader thread, sometimes, after the execution of the broker or the executer, the reader thread will execute and print the current market status on the standard output.

Test ID: 7

- *Description*: test to verify the correct implementation for multiple stats_readers with multiple brokers. Files used correspond to test 4.

- *Input:* three threads for brokers, one thread for operations_executer and another one for stats_reader. As explained in test 5, the provided files have 6 transactions in each batch file for each broker and a stock market with 10 companies. The readers will have different frequencies in the range of 10,000 to 100,000 microseconds.

- *Output (expected and obtained):* we can verify that the output is the same as for test 4, but adding the output of the readers which execute after the execution of the broker or the executer. Once a reader is executing, more readers can execute without waiting.

5. Conclusion

This lab assignment has allowed us to familiarize and get more practice with concepts regarding threads, concurrency, condition variables and mutexes. After doing it we believe to have a better understanding of threads, mutexes, as well as condition variables regarding how to use them and their behavior as how to localize and control critical sections of code. We believe the hardest part of the practice was not to develop and use the mutexes but to understand the already given code and how to work in the critical sections with the data coming from other classes and functions already implemented. Debugging and testing were also difficult tasks since we had never before performed them in a multi-thread program.

We have used as support materials the provided course slides as well as the given exercises to understand the behavior of threads, mutexes and condition variables as well as online external material and the provided code, which was well commented.