

Computer Architecture and Technology Area

Universidad Carlos III de Madrid





OPERATING SYSTEMS

Lab 3. Programming of a Stock Market functionality (Multithread)

Bachelor's Degree in Computer Science and Engineering

Year 2018/2019

	<p>Informatics Department Bachelor's Degree in Computer Science and Engineering Operating Systems (2018-2019) Lab 3 - Concurrency</p>	
---	---	---

Contents

1.	Lab statement.....	3
1.1.	Lab description.....	3
2.	Assignment submission.....	15
2.1.	Deadline and method.....	15
2.2.	Submission.....	15
2.3.	Files to be submitted.....	15
3.	Rules.....	16
4.	Appendix (man function).....	18
5.	Bibliography.....	18

1. Lab statement

This lab allows the student to familiarize with the services for the management of threads that is provided by POSIX. The objective is that the student understands the importance of synchronizing the threads when they work concurrently.

For the thread management, you will use the related system calls of POSIX, such as `pthread_create` and `pthread_join`. For the thread synchronization, you will use *mutex* and *conditional variables* using calls such as `thread_mutex_init`, `pthread_mutex_lock`, `pthread_mutex_unlock`, `pthread_cond_init`, `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_destroy`. **Other methods for synchronization are not allowed.**

The student must design and program, in C language and over UNIX/Linux Operating System, a simulation of the functionality of a factory with different roles represented by threads that will work concurrently.

1.1. Lab description

It is necessary to write program that simulates a stock exchange with three main different components. These three roles must work concurrently managing the stock. These three roles are:

- **Broker:** It is responsible for inserting new stock trades on the queue of market operations. It takes the operations from a batch file. There may be n concurrent threads with this role, but only runs one operations at a time to maintain data integrity.
- **Operation_executer:** It is in charge of processing operations one by one (operations that were inserted into the market operations queue by the brokers). There will only be a thread running this role.
- **Stats_reader:** It is responsible for evaluating the current state of the market. There may be n concurrent readers in the system and all can run simultaneously. During this read process no task that modifies the market (operation_executer or broker) can be executed.

Broker and operation_executer make use of the queue operations, while readers can access directly to the state of the market.

Information of the available market shares can be loaded from a file. The operations to be performed by a particular broker also can be loaded from a file consisting of batch operations, which contains an operation on each file line.

Library	PARSER	OPERATIONS_QUEUE
	STOCK_MARKET_LIB	
	CONCURRENCY_LAYER	
main program	CONCURRENT_MARKET	

Figure 1: Functionality.

The base code provided, consists of several layers that simulate the behavior of different market components. The top layer, implemented in the parser.c / parser.h files and operations_queue.c / operations_queue.h respectively include auxiliary functions to simplify coding functionality market.

1.1.1. Parser

It includes auxiliary functions that operate on the operation batch file line by line and display on screen the current status of all actions of the stock market.

- `iterator * new_iterator(char * file_name)`
 - o Description: generates a new iterator for input file.
 - o Input: file name on which you want to iterate (a file in batch operations).
 - o Output: pointer to new iterator or NULL on case of error.
- `void destroy_iterator(iterator * iter)`
 - o Description: Destroy the last iterator passed as input parameter.
 - o Input: pointer to the iterator to destroy.
 - o Output: void.
- `int next_operation(iterator * iter, char * id, int * type, int * num_shares, int * price)`
 - o Description: given a iterator passes as input parameter pointers, this function sets the id, type, and price num_shares data with the current file associated to the iterator.
 - o Input: pointer to the iterator and pointers to the parameters that has to be set.

- o Output: Sets the variables associated to the pointers and returns: 0 → ok, -1 → error.

1.1.2. Operations queue

It includes data structures and functions necessary to work with a queue of operations. An operation contains:

- ID of the company/share associated to the operation that is being executed.
- type: type of the operation to be performed (buy (BUY) or sell (SELL)).
- num_shares: number of shares to buy / sell.
- share_price: price to be paid for each share you want to buy / sell.

Functions (description of the functions available in lib / operations_queue.c and lib / operations_queue.h)

```
operations_queue * new_operations_queue(int max_items);
```

```
void delete_operations_queue(operations_queue *q);
```

```
int operations_queue_empty(operations_queue *q);
```

```
int operations_queue_full(operations_queue *q);
```

```
int dequeue_operation(operations_queue *q, operation *op);
```

```
int enqueue_operation(operations_queue *q, operation *op);
```

```
void new_operation(operation *op, char id[ID_LENGTH], int type, int num_shares, int share_price);
```

stock_market_lib

It contains the core functionality of the market. A stock market will have the following characteristics:

- stocks [NUM_STOCKS] array containing NUM_STOCKS shares. They will be the shares on which may operate in this market.
- total_value: the total value of this market (the total value of all shares of the market).
- avg_value: the average value of each share available in this market.
- num_active_stocks: how many shares are currently available on the market.
- stock_operations: the queue of pending operations. The broker threads will add new operations on it and they will be processed by the thread operation_executer in FIFO order.

Each of the actions included in the array stocks, contains the following information about the value:

- ID: Unique identifier (maximum 10 characters) of this value.
- name: company name (maximum 255 characters).
- total_shares: number of shares of the company.
- total_value: total value of the company.
- current_share_value: the value of the company divided among all the shares available ($\text{total_value} / \text{total_shares}$).

Functions (description of the functions available in lib / stock_market_lib.c and lib / stock_market_lib.h)

```
int init_market(stock_market * market, char * file_name);
```

```
void delete_market(stock_market * market);
```

```
int new_stock(stock_market * market, char id[ID_LENGTH], char
name[STOCK_NAME_LENGTH], int current_share_value, int
total_shares);
```

```
stock * lookup_stock(stock_market * market, char
id[ID_LENGTH]);
```

```
void update_market_statistics(stock_market * market);
```

```
int process_operation(stock_market * market, operation * op);

void print_market_status(stock_market * market);
```

1.1.3. Work to be done

The aim of the concurrency layer is to allow simultaneous access of multiple brokers to the service for processing market operations. Thus, this layer contains the concurrency control mechanisms necessary to prevent concurrency errors. THE STUDENT HAS TO IMPLEMENT THE FILE lib / concurrency_layer.c KEEPING THE lib / concurrency_layer.h DEFINITION OF FUNCTIONS AND STRUCTURES. FILE lib / concurrency_layer.h MUST NOT BE MODIFIED.

The lib / concurrency_layer.c file must contain at least the implementation of the five functions in lib / concurrency_layer.h. You can include as many auxiliary functions and global variables as necessary (for example, mutex, condition variables, etc.).

- `void init_concurrency_mechanisms()`: initializes the concurrency control mechanisms necessary for the correct operation of the application with multiple threads. It receives no input parameters and has no return value. It will always be called by the main () function of the program before creating any thread.

- `void destroy_concurrency_mechanisms()` destroys all concurrency control mechanisms used during program execution. It will be always called by the main () function after the completion of all threads and before completion of the program.

- `void* broker(void * args)` implements the functionality of broker threads (concurrency control included). Receives `broker_info` as input parameter that contains:

- `char batch_file[256]`: name of the batch file containing transactions to be made by the broker.
- `stock_market * market`: pointer to the market on which trading broker will operate with.

Pseudocode function (concurrency control has to be designed by the student):

Extract information from data received in the pointer **void * args**

Create the iterator on the batch file (**new_iterator**)

While there are pending file operations

Read a new operation from the file with the iterator (**next_operation**)

Create a new operation with the information returned by the file (**new_operation**)

Queue the new operation in the operation queue (**enqueue_operation**)

Destroy the iterator (**destroy_iterator**)

- **void* operation_executer(void * args)** implements the functionality of the thread that processes the operations (including concurrency control). It receives as input parameter the **exec_info** type structure containing:
 - o **int *exit** pointer to flag termination of the application. When the flag has a true value the thread will process requests pending in the queue and terminate.
 - o **stock_market * market**: pointer to the market on which the processor will perform trading operations.
 - o **pthread_mutex_t *exit_mutex**: pointer to mutex that protects the variable **exit**. It must access the variable **exit** always with a locked mutex.

Pseudocode function (concurrency control has to be designed by the student):

Extract information from data received in the pointer **void * args**

While the **exit flag** is not active

Dequeue operation from the operations queue (**dequeue_operation**)

Processing the operation (**process_operation**)

- **void* stats_reader(void * args)** implements the functionality of the threads that evaluate the stock exchange market (including concurrency control). It receives **reader_info** as input parameter containing:
 - o **int *exit**: pointer to flag termination of the application. When the flag passes true value the thread will terminate.
 - o **stock_market * market**: pointer to the market on which the consultant will read the stock market statistics.
 - o **pthread_mutex_t *exit_mutex**: pointer to the mutex that protects the variable **exit**. This variable has to be accessed always with the mutex locked.
 - o **unsigned int frequency**: time that the thread should sleep after each query (sampling frequency determines the market).

Pseudocode function (concurrency control has to be designed by the student):

```
Extract information from data received in the pointer void * args  
While the exit flag is not active  
    View stock market statistics (print_market_status)  
    Sleep until the next round of information analysis (usleep  
    (frequency))
```

1.1.4. Concurrency requirements

For the proper functioning of the market there are certain operations that cannot run at the same time and make use of shared resources. These operations are:

- Write operations: both the broker operations and processor operations (operation_executer) are considered global write operations on the system. Only one of them may be running simultaneously.
 - There will only be a thread running processing operations. Each operation will run in an isolated way (no broker, no reader can operate simultaneously).
 - You can have multiple threads to insert system operations (brokers), but each operation will be executed in an isolated way (no other broker or operation_executer or reader can operate simultaneously).
- Insertion / processing operations: the threads that insert operations in the queue must stop when it is full and restore its execution when possible (queue is not full). Similarly, the processor must stop operating when the queue is empty and restore execution when possible (nonempty queue). **It is not allowed to display any error message during program execution.**
- Read operations: there may be n reader threads running on the system at any instant of time and read operations may be concurrent. During the processing of a read operation, any write operation will not be allowed to be performed. However other read operations are allowed.

SEMAPHORES are not allowed to be used as concurrency control mechanism.

The libraries provided include error messages to help the debug process. A correct program must show no trace that includes the word **ERROR**, it may be penalized if it happens. The student **MUST NOT INCLUDE ANY TRACE** different to those included in the base code.

1.1.5. File configuration

For the correct execution of the application, there are two types of files that facilitate the introduction of information.

File with the description of the stock market. It contains information about all companies listed on a stock market. Within the base code a sample file called stocks.txt is included.

The file format includes a company and market information in each new line in the file. Each line contains the following information in this format (separators will always be blank spaces and values are always integers):

<ID> <name> <number_of_shares> <price_per_share>

- o <ID>: unique identifier of the company up to 10 characters.
- o <name>: full company name up to 255 characters.
- o < number_of_shares>: initial number of shares of the company.
- o < price_per_share >: starting value of each share of the company (the total value of the company will be: < number_of_shares > * < price_per_share >).

File with the description of the batch operations. It contains the operations to be performed by a broker. Within the base code sample file called batch_operations.txt is included. The file format includes information for each operation on a new line in the file. Each line contains the following information in this format (separators will always be blank space and values are always integers):

<ID> <type> <number_of_shares> <price_per_share>

- o <ID>: unique identifier of the company on which you want to apply the operation (up to 10 characters). The company must exist in the stock market on which the broker works.
- o <type>: type of operation: purchase or sale. 0 corresponds to the value of the constant BUY file lib / stock_market_lib.h (share purchase transaction) and 1 corresponds to the constant SELL (operation sale of shares).
- o < **number_of_shares** >: number of shares of the company to buy or sell.
- o < **price_per_share** >: price to be paid for each of the shares are purchased / sold.

1.1.6. Main function implementation and execution

The main programs must be implemented in concurrent_market.c to be compiled by the Makefile without requiring any modification. If you want to include more programs, you must modify the Makefile to be compiled (**but only to create your own tests**). In order to guarantee a correct submission, the original Makefile must be sent.

The main program must include at least:

- Include layer concurrency library implemented by the student
 - o Example: #include "include/concurrency_layer.h"
- Initialization market through a description file market, including information on the actions that are part of the market.
 - o Example: init_market (& market, "stocks.txt");
- Initialization concurrency mechanisms before creating any thread.
 - o Example: init_concurrency_mechanisms ();
- Creating threads interacting with the stock market. At least there should be a thread broker to introduce new transactions in the system and a thread operation_executer that process the transactions.
 - o Example:
 - pthread_create(&(tid[0]), NULL, &broker, (void*) &info_b1);
 - pthread_create(&(tid[1]), NULL, &operation_executer, (void*) &info_ex1);

NOTE: info_b1 and info_ex1 variables are instances of the broker_info structures and exec_info respectively and are properly initialized.

- join the broker threads. It is expected first for the completion of all broker threads.
 - o Example: pthread_join (tid [0], & res);
 - Activating the exit flag.
 - Join ALL other threads created (operation_executer and stats_reader).
 - Destruction of the mechanisms of competition and the market.
 - o Example:
destroy_concurrency_mechanisms ();
delete_market (& market_madrid);
 - OPTIONAL: print the final state of the market to ensure that all transactions are properly made (before the destruction of the market).
 - o Example: print_market_status (& market);
- NOTE: a correct final state does not ensure that the practice is 100% correct. You must ensure the proper functioning of all threads in each part of the execution. However, it can be useful to print the final state.

You can find an example of basic main program in the concurrent_market.c file that uses configuration files stocks.txt (description of the stock market) and batch_operations.txt (description of transactions in the stock market for a broker). This file can serve as a basis for implementing more advanced operations.

To run the program it must be compiled and launched:

```
make  
./concurrent_market
```

1.1.7. Support code

To facilitate the realization of this practice p3_concurrency_2016.tar.gz file is available. It contains the support source code. To extract its contents, execute the following:

```
tar zxvf tar p3_concurrency_2016.tar.gz
```

To extract its contents, the ssoo_p3_stock is created. Within this directory we have included the following files:

Makefile

Source file for the make tool. It must not be modified. It allows the automatic compilation of the source files.

concurrent_market.c

Sample file of the main program. You can modify it.

stocks.txt

Description file of companies and shares of a stock market. You can modify it.

batch_operations.txt

Example file batch operations. A thread broker must add them to the system. It has to be changed.

include / concurrency_layer.h

Header file with definitions of the functions and data structures. The student must implement the functions described in this file. You cannot modify it.

include / operations_queue.h

Header file that contains the functions that can be used to interact with the operation queue. You cannot modify it.

include / parser.h

Header file that contains auxiliary functions for data processing (iterator over the file and print operations statistics). You cannot modify it.

include / stock_market_lib.h

Header file that contains the functions that can be used to interact with the market. You cannot modify it.

lib / concurrency_layer.c

This file should contain the implementation of the functions that use the application threads. Implements the file include / *. h of the same name. In this file, the student must implement their practice. It has to be modified.

lib / operations_queue.c

Code file containing the implementation of the functions described in the file include / *. h of the same name. You cannot modify it.

lib / parser.c

Code file containing the implementation of the functions described in the file include / *. h of the same name. You cannot modify it.

lib / stock_market_lib.c

Code file containing the implementation of the functions described in the file include / *. h of the same name. You cannot modify it

1.1.8. Output format

No additional output to the one provided in the base code is allowed.

2. Assignment submission

2.1. *Deadline and method*

The deadline for the delivery of this programming assignment in AULA GLOBAL will be Monday 6th May 2019 (until 23:55h).

2.2. *Submission*

The submission must be done using Aula Global using the links available in the first assignment section. The submission must be done separately for the code and using TURNITIN for the report.

2.3. *Files to be submitted*

You must submit the code in a zip compressed file with name ssoo_p3_AAAA_BBBB_CCCC.zip where A...A, B...B and C...C are the student identification numbers of the group. A maximum of 3 persons is allowed per group, if the assignment has a single author, the file must be named ssoo_p3_AAAAAAAAAA.zip. The file to be submitted must contain:

Makefile
concurrent_market.c
stocks.txt
batch_operations.txt
include/concurrency_layer.h
include/operations_queue.h
include/parser.h
include/stock_market_lib.h
lib/concurrency_layer.c
lib/operations_queue.c
lib/parser.c
lib/stock_market_lib.c
authors.txt

authors.txt contains the name of each student belonging to the group and NIA in two different lines. The NIA is separate from full name by a tab. Example:

```
Name Surname1 Surname2 <tab> NIA
Name Surname1 Surname2 <tab> NIA
Name Surname1 Surname2 <tab> NIA
```

The report must be submitted in a PDF file. Notice that only PDF files will be reviewed and marked. The file must be named ssoo_p3_AAAA_BBBB_CCCC.pdf. A minimum report must contain:

- **Description of the code** detailing the main functions it is composed of. Do not include any source code in the report.

- **Tests cases used and the obtained results.** All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to take into account:
 - Avoid duplicated tests that target the same code paths with equivalent input parameters.
 - Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the number of tests per program.
 - Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Additionally, marks will be given attending to the quality of the report. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain an index.
- Every page except the title page must be numbered.
- Text must be justified.

The PDF file must be submitted using the TURNITIN link and the maximum size is 8 pages. Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

NOTE: Only the last version of the submitted files will be reviewed.

3. Rules

- 1. Programs that do not compile or do not satisfy the requirements will receive a grade of zero.**
- 2. All programs should compile without reporting any warnings. In case any warning is reported during the compilation, the final mark will be reduced according to its severity.**
- 3. Programs without explanations will receive a grade of 0.**
- 4. The assignment must be submitted using the available links in Aula Global 2. Submitting the assignments by mail is not allowed without prior authorization.**
- 5. The programs implemented must work in the computers of the computer lab of the university or the guernika platform using Linux. It is the student responsibility to be sure that the delivered code works correctly in those places.**
- 6. It is mandatory to follow the input and output formats indicated in each program implemented.**
- 7. It is mandatory to implement a minimum level of error handling.**
- 8. It is mandatory to follow the format requirements. In case this is not fulfilled there will be a penalization to the mark obtained.**
- 9. Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be filled.**

Failing to follow these rules will translate in receiving a grade of zero in the affected programs.

4. Appendix (man function).

man is a command that formats and displays the online manual pages of the different commands, libraries and functions of the operating system. If a *section* is specified, man only shows information about name in that section. Syntax:

```
$ man [section] name
```

A man page includes the synopsis, the description, the return values, example usage, bug information, etc. about a *name*. The utilization of man is recommended for the realization of all lab assignments. **To exit a man page, press q.**

The most common ways of using man are:

1. **man section element:** It presents the element page available in the section of the manual.
2. **man -a element:** It presents, sequentially, all the element pages available in the manual. Between page and page you can decide whether to jump to the next or get out of the pager completely.
3. **man -k keyword** It searches the keyword in the brief descriptions and manual pages and present the ones that coincide.

5. Bibliography

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (man function)