Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Lab 3. Multithreading

Bachelor's Degree in Computer Science & Engineering
Bachelor's Degree in Applied Mathematics & Computing
Dual Bachelor in Computer Science & Engineering & Business Administration

Year 2023/2024

Contents

1 Lab Statement

This programming assignment allows the student to become familiar with services used for process management provided by POSIX.

For the management of lightweight processes (threads), the functions `pthread_create`, `pthread_join`, `pthread_exit`, will be used and, **mutex** and **conditional variable** will be used for the synchronization between them.

- **pthread_create**: creates a new thread that executes a function that is indicated as an argument in the call.
- **pthread_join**: waits for a thread that must end and that is indicated as an argument in the call.
- **pthread_exit**: ends the execution of the process that makes the call.

The student must design and implement in C and for LINUX operating systems, a program that acts as a profit and stock manager for a store.

1.1 Lab Description

The objective of this practice is to code a concurrent multi-threaded system that calculates the profit and stock of a store. Given a file with a specific format, the profit (difference between the cost of acquiring the products and the revenue from their sale) as well as the remaining stock of each product must be calculated.

To realize the functionality, it is recommended to implement two basic functions, which represent the roles of the program (following the behavior of the **Figure 1**):

- **Producer**: It will be the function executed by the threads in charge of adding elements in the shared circular queue.
- **Consumer**: It will be the function executed by the threads in charge of extracting elements from the shared circular queue.

1. The **main thread** will be the one in charge of:
   (a) Read the input arguments.
   (b) Load the data from the file provided into memory.
   (c) Distribute the file load equally among the number of producers threads indicated.
   (d) Launch the **N** producers and the **M** consumers.
   (e) Wait for the finalization of all the threads and show the profit and stock of each product at the end of the operations.

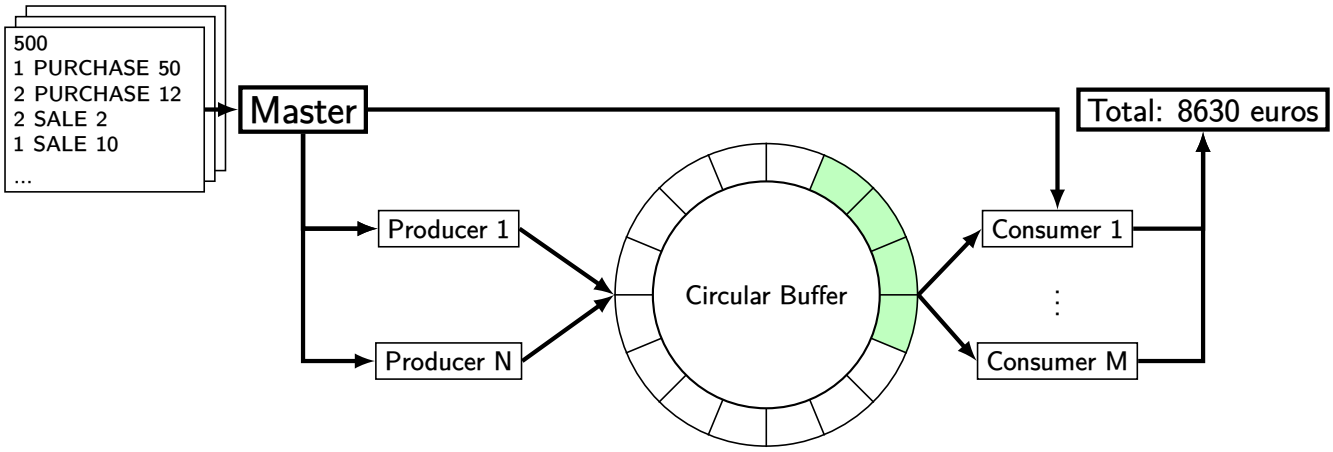

Figure 1: Example of operation with N producers, M consumers and one buffer.

2. Each **producer thread** must:
   (a) Obtain the data extracted from the file and insert them, **one by one**, in the circular buffer.
   (b) **This task must be executed concurrently with the other producers, as well as the consumers**. Under no circumstances may the threads be left **manually** blocked or an order forced between them (e.g. waiting for a thread to insert all its elements, or those that fit in the queue, and then giving way to the consumer to remove them; then giving way to the next producer, etc.).
3. Each **consumer thread** must:
   (a) Obtain concurrently, the elements inserted in the queue.
   (b) Each element extracted represents a product transaction. These operations consist of the purchase of a given number of products and the sale of a given number of products. Therefore, the benefits of carrying out these operations must be calculated, as well as the stock of each product once the operations have been carried out.
   (c) Once all the elements have been processed, each thread will finish its execution and **will return the calculated profit and partial stock to the main process.**

1.1.1 Store costing and stock manager

The main program will import the arguments and data from the indicated file. To do this, it must be considered that the execution of the program will be as follows:

```
./store_manager <file_name><num_producers><num_consumers><buff_size>
```

The **file_name** label corresponds to the name of the file to be imported. The label **num_producers** is an integer that represents the number of producer threads that you want to generate. The label **num_consumers** is an integer that represents the number of producer threads that you want to generate. Finally, the **buff_size** label is an integer that indicates the size of the circular queue (maximum number of elements that can be stored).

Additionally, the input file must have the following format:

```
500 #Num max operations to process
1 PURCHASE 50
2 PURCHASE 12
2 SALE 2
1 SALE 10
...
```

The first line of the file represents the number of operations to be calculated. There can be more operations in the file, but **only those indicated by this value should be processed. Under no circumstances there may be fewer operations in the file than operations indicated by the first value.** The rest of the lines in the file represent an operation:

<product_id><operation_type><units>

These are three values separated by a space and ending in a line break. The **product_id** is the product identifier and has a purchase cost and a unit selling price associated with it:

product_id	**Purchase cost** (€/unit)	**Sales price** (€/unit)
1	2	3
2	5	10
3	15	20
4	25	40
5	100	125

Table 1: Purchase cost and selling price of each product.

The second value indicates the type of operation to be performed on the product. It can be the acquisition of a product (PURCHASE) or the sale of a product (SALE). Finally, the last value indicates the number of units that have been purchased or sold of a given product.

The main process must load the information contained in the file into memory for further processing by the producers. To do this, it is recommended to use the `scanf` function and the dynamic memory reserve with `malloc` (and `free` for later free up). The idea is:

1. Obtain the number of operations (first value in the file).
2. Reserve memory for all those operations with `malloc`.
3. Store the operations in the array.
4. Distribute **equally** the operations among the producers:
   (a) To simplify the task, it is recommended to make a distribution of the operations, so that each producer thread knows in which position to start processing and in which position to finish.
   (b) In this way each thread is aware of when it must finish its execution.
   (c) To do this, arguments can be passed to each producer thread at launch with **pthread_create**.
5. After processing the threads, free the reserved memory with `free`.

NOTE

To store the data from the file, an array of structures can be generated. It is also recommended to use a structure for passing parameters to the threads.

An example of the program output is shown below:

```
Total: 234234 euros
Stock:
    Product 1: 10 units
    Product 2: 2 units
    Product 3: 50 units
    Product 4: 22 units
    Product 5: 33 units
$>
```

1.1.2 N-Producers & M-Consumers

The problem to be implemented is a classic example of process synchronization: when sharing a shared queue (**circular buffer**), it is necessary to control the concurrency when depositing objects in it, and when extracting them.

For the implementation of the producers threads it is recommended that the function follows the following scheme for simplicity:

1. Obtain the indexes to be accessed from the data in the file. It is recommended to pass parameters to the thread.
2. Loop from the beginning to the end of the operations that should be processed:
   (a) Obtain the operation data.
   (b) Create an element with the operation data to insert in the queue.
   (c) Insert element in the queue.
3. End the thread with **pthread_exit**.

For the implementation of the consumers thread it is recommended to follow a similar scheme to the previous one for simplicity:

1. Extract element from the queue.
2. Calculate the profit and stock and accumulate it.
3. When all operations have been processed, end the thread with pthread_exit **returning the profit and partial stock calculated by each**.

NOTE

Mutex and **condition variables** must be used for concurrency control. Concurrency can be managed in the producer and consumer functions, or in the circular queue code (in **queue.c**). The choice is of the practice group.

Queue on a circular buffer Communication between producers and consumers will be implemented through a circular queue shared. Since changes to this element will constantly occur, mechanisms for the control of concurrency must be implemented for light processes.

The circular queue and its functions must be implemented in a file called **queue.c**, and it must contain at least the following functions

- **queue* queue_init (int num_elements)**: function that creates the queue and reserves the size specified as a parameter.
- **int queue_destroy (queue* q)**: function that removes the queue and frees up all assigned resources.
- **int queue_put (queue* q , struct element * ele)**: function that inserts elements in the queue if there is space available. If there is no space available, it must wait until the insertion can be done.
- **struct element * queue_get (queue* q)**: function that extracts elements from the queue if it is not empty. If the queue is empty, it must wait until an element is available.
- **int queue_empty (queue* q)**: function that consults the status of the queue and determines if it is empty (return 1) or not (return 0)
- **int queue_full (queue* q)**: function that consults the status of the queue and determines if it is full (return 1) or still has available positions (return 0).

The implementation of this queue must be done in such a way that there are no concurrency problems between the threads that are working with it. To do this you should use the proposed mechanisms of **mutex** and **condition variables**.

The object to be stored and extracted from the circular queue must correspond to a defined structure with at least the following fields:

- **int product**: represents the product identifier.
- **int op**: represents the type of operation.
- **int units**: represents the number of units involved in the operation.

1.2 Initial Code

To facilitate the realization of this lab you have the file p3_multithread_2024.zip which contains supporting code. To extract the content, you can execute the following:

```
unzip p3_multithread_2024.zip
```

To extract its content, the directory p3_multithread_2024/ is created, where you must develop the lab. Inside that directory the next files are included:

- **Makefile**
  It must NOT be modified. File used by the `make` tool to compile all programs. Use *make* to compile the programs and *make clean* to remove the compiled files.

- **store_manager.c**
  This file must be modified. C source file where the students must code the profit and stock manager of a store.

- **queue.c**
  This file must be modified. Headers file where students will have to define the data structures and functions used for the management of the circular queue.

- **queue.h**
  This file must be modified. C source file where the students will have to implement the functions that allow the management of the circular queue.

- **checker_os_p3.sh**
  It must NOT be modified. Shell script that carries out an auto-correction of the student's code. It shows, one by one, the test instructions to be carried out, as well as the expected result and the result obtained by the student's programme. At the end, a tentative score is given for the code of practice (excluding manual review and memory). To run the script the students must change the permissions:

  ```
  chmod +x checker_os_p3.sh
  ```

  And run it:

  ```
  ./checker_os_p3.sh <zip_file>
  ```

- **authors.txt**
  This file must be modified. `txt` file where to include the authors of the practice.

- **file.txt**
  Support file.

1.3 Results checker

Along with the support code, a **checker.xlsx** file is attached, which is an excel document to check if the obtained results are correct or not. The **file.txt** file provided, contains the same data as the first one, but without formulas and checks (it has the specified format to be able to process it). If the number of operations to be processed in *file.txt* is modified, the result obtained can be checked by modifying the number of operations in the document *checker.xlsx*.

NOTE

It is also possible to generate new files (**recommended**), and to perform new tests on them.

2 Assignment submission

2.1 Deadline and method

The deadline for the submission of the lab in AULA GLOBAL will be **May 10th, 2024 (until 23:55h)**.

2.2 Submission

The submission must be done using Aula Global using the links available in the first assignment section and **by a single member of the group**. **The submission must be done separately for the code and report**. The report will be submitted through the TURNITIN tool.

2.3 Files to be submitted

You must submit the code in a zip compressed file with name:

`os_p3_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip`

Where A...A, B...B, and C...C are the student identification numbers of the group. A maximum of 3 members is allowed per group, if the assignment has a single author, the file must be named **os_p3_AAAAAAA.zip**. **The zip file will be delivered in the deliverer corresponding to the code of the lab.** The file to be submitted must contain:

- **store_manager.c**
- **queue.c**
- **queue.h**
- **Makefile**
- **authors.txt: Text file with the information of the authors in different lines, with the following format (csv): NIA, Surname, Name**

NOTE

To compress such files and be processed correctly by the provided tester, it is recommended to use the following command:

```
zip os_p3_AAA_BBB_CCC.zip Makefile Makefile store_manager.c queue.c
queue.h authors.txt
```

The report must be submitted in a PDF file. The file must be named:

`os_p3_AAAAAA_BBBBBB_CCCCCC.pdf`

Note that only PDF files will be reviewed and marked. The report must contain at minimum:

- **Cover** with the authors (including the complete name, NIA, and email address).
- **Table of contents**
- **Description of the code** detailing the main functions it is composed of. Do **NOT** include any source code in the report (it will be ignored).
- **Tests cases** used and the obtained results. Higher scores will be given to advanced tests that cover edge cases, and, in general, to those tests that guarantee the correct operation of the program in all cases. In this regard, the following things must be taken into account:
  1. Avoid duplicated tests that target the same code paths with equivalent input parameters.
  2. 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.
  3. Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describe the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the completion 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 a table of contents.
- Every page except the title page must be numbered.
- Text must be justified

The PDF file must be submitted using the TURNITIN link. The length of the report should not exceed 15 pages (including cover page and table of contents). Do not neglect the quality of the report as it is a significant part of the grade.

NOTE

You can submit the lab code as many times as you wish within the deadline, the last submission will be considered the final version. **THE LAB REPORT CAN ONLY BE SUBMITTED ONCE ON TURNITIN.**

3 Rules

1. Programs that do not compile or do not satisfy the requirements will receive a mark of **zero**.

2. Special attention will be given to detecting copied functionalities between two practices. In case of finding common implementations in two practices, the students involved (copied and copiers) will lose the grades obtained by continuous evaluation

3. All programs should compile without reporting any warnings.

4. The programs must run under a Linux system, the practice is not allowed for Windows systems. In addition, to ensure the correct functioning of the practice, its compilation and execution should be checked in the university computer labs or on the guernika.lab.inf.uc3m.es server. If the code presented does not compile or does not work on these platforms the implementation will not be considered correct.

5. Programs without comments will receive a grade of 0.

6. The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.

7. It is mandatory to follow the input and output formats indicated in each program implemented. In case this is not fulfilled there will be a penalization to the mark obtained.

8. It is mandatory to implement error handling methods in each of the programs.

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 be translated into zero marks in the affected programs.

4 Appendix

4.1 Manual (man command)

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] open
```

The pages used as arguments when running man are usually names of programs, utilities or functions. Normally, the search is performed on all available man sections in a predetermined order, and only the first page found is presented, even if that page is in several sections.

A manual page has several parts. These are labeled NAME, SYNOPSIS, DESCRIPTION, OPTIONS, FILES, SEE ALSO, BUGS, and AUTHOR. The SYNOPSIS label lists the libraries (identified by the #include directive) that must be included in the user's C program in order to make use of the corresponding functions. 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`)