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 2022/2023

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 code, in C language and on the UNIX / Linux operating system, a program that acts as a bank that provides operations on accounts from ATMs, so that the balance of the accounts is always correct.

1.1 Lab Description

The objective of this computer lab is to program a concurrent multi-threaded system that acts as a bank, managing account operations from ATMs, so that the balance of the accounts is always correct. A file is provided to the students that indicate the following:

- A list of account operations, including: CREATE, DEPOSIT, WITHDRAW, BALANCE, TRANSFER.

The bank program must read the file and create the threads of the ATMs and workers. In addition, it must create a circular queue on which the ATMs write operations and from which the workers extract operations to execute. The operations on the requested accounts must be executed, ensuring that the balance is correct at the end of all operations on each account, regardless of the order of execution.
For the implementation of the functionality, it is recommended to implement two basic functions that represent the role of the program (following the behavior of **Figure 1**):

- **ATM**: It will be the function executed by the threads responsible for reading the operations from the file and adding elements into the shared circular queue.
- **Worker**: It will be the function executed by the threads responsible for extracting elements from the shared circular queue and actually performing the bank operations.

1. The **bank** (main thread) will be responsible for:
   (a) Reading the input arguments and creating the ATMs and workers as indicated.

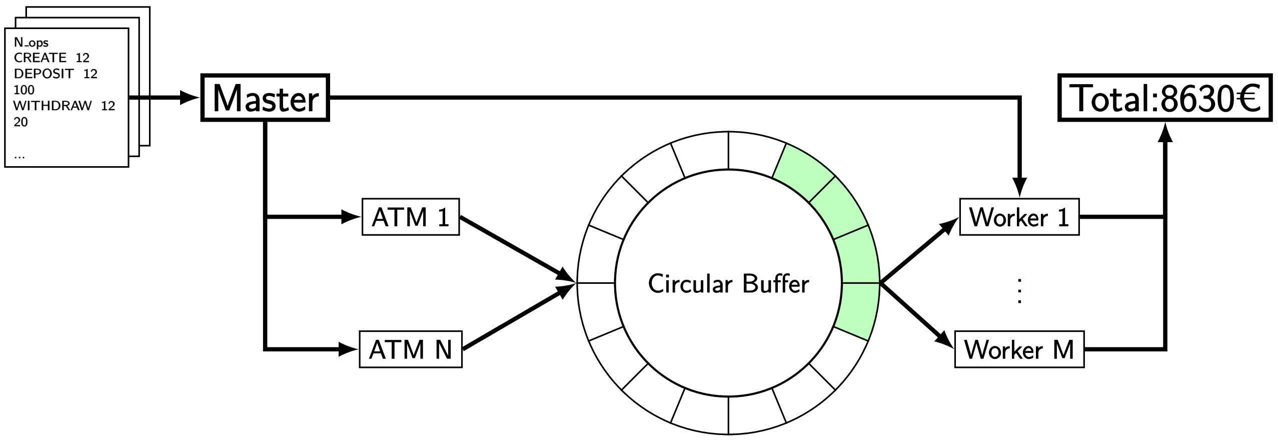

Figure 1: Example of operation with N cashiers, M workers and a buffer

(b) Creating three global variables: **client_numop**, which is initialized to zero and incremented each time a customer performs an operation from a ATM; **bank_numop**, which is initialized to zero and incremented each time a worker performs an operation taken from the circular queue shared with the ATMs; **global_balance**, which is initialized to zero and updated with the operations that are performed **(it can be positive or negative)**.

(c) Loading the list of operations from the provided file into memory in an array **list_client_ops**.

(d) Launching the **N** ATMs and the **M** workers.

(e) Waiting for the completion of all threads.

2. Each **ATM** (producer thread) must:

   (a) Obtain operations from the array **list_client_ops** one by one, increment the number of customer operations (client_numop), and insert the operation into the circular queue.

   (b) **This task must be performed concurrently with the rest of the ATMs, as well as with the workers**. Under no circumstances should the threads be **manually blocked or force an order between them** (for example, waiting for a thread to insert all its elements or those that fit in the queue, and letting the consumers extract them; then letting the next producer through, etc.). The order will be done by the operation number (client_numop).

3. Each **worker** (consumer thread) must:

   (a) Increment the bank operation number (**bank_numop**) and extract the operation from the queue with that number.

   (b) For each extracted element, perform the indicated banking operation on the associated accounts and also update the global balance.

   (c) Print the type of operation, its parameters, and the resulting balance on the screen.

   (d) Once all customer operations have been processed, each thread will end its execution.

1.1.1 Bank Program

The main program (bank) 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:

```
./bank <file_name> <num_ATMs> <num_workers> <max_accounts> <buff_size>
```

The label **file_name** corresponds to the file's path to be imported. The label **num_ATMs** is an integer representing the number of ATMs (producer threads) to be generated. The label **num_workers** is an integer representing the number of workers (consumer threads) to be generated. The label **max_accounts** is a positive integer representing the maximum number of accounts that the bank can have. Finally, the label **buff_size** is an integer indicating the size of the circular queue (maximum number of elements it can store).

Below is an example of the contents of the input file:

```
50 #Max number of operations to be processed
CREATE 12
DEPOSIT 12 100
WITHDRAW 12 20
CREATE 25
TRANSFER 12 25 30
BALANCE 25
...
```

The first line indicates the number of operations in that session (**max_operations**). **It must be checked that the number of operations indicated in this line is not exceeded and that the maximum of 200 operations is not exceeded. Also, there cannot be fewer operations in the file than the number of operations indicated in the first value**. Each of the following lines indicates the type of operation and its arguments, which are integer values separated by a space. It should be noted that there is only one operation per line, and all of them always have the operation type and the account number on which to perform that operation.

The main process must load the information in the file into memory - in a vector or a queue - for later processing by the ATMs. The `scanf` function is recommended to read the file. The idea is to:

1. Reserve memory for the maximum number of allowed operations (max_operations) with `malloc`.
2. Store the operations in the array, **list_client_ops**, one per element.
3. Pass the array argument to the ATMs at the time of launch with **pthread_create** so that they read the operation indicated in **client_numop**.
4. After processing the operations of the threads, release the reserved memory with `free`.

NOTE

An array of structures can be generated to store the data from the file. Using a structure for passing parameters to the threads is also recommended.

Each worker must print on the screen the operations performed, indicating first the order of the operation (which must be global for all workers and consecutive), the type of operation, the arguments, the account balance, and the global bank balance at each moment.

Next, an example of the program output for the operations file of the previous example is shown:

```
$> ./bank input_file 5 3 50 20
1 CREATE 12 BALANCE=0 TOTAL=0
2 DEPOSIT 12 100 BALANCE=100 TOTAL=100
3 WITHDRAW 12 20 BALANCE=80 TOTAL=80
4 CREATE 25 BALANCE=0 TOTAL=80
5 TRANSFER 12 25 30 BALANCE=30 TOTAL=80
6 BALANCE 25 BALANCE=30 TOTAL=80
...
$>
```

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 into it and when extracting them.

For the implementation of producer threads, it is recommended that the function follow the following outline for simplicity:

1. Loop from the start to the end of the bank operations to be processed:
   (a) Obtain the data of the operation, safely increase the client operation variable (client_numop).
   (b) Create an element with the operation data to insert in the queue, always including the operation number in the first place.
   (c) Insert the element into the shared queue.
2. Finish the thread with **pthread_exit**.

For the implementation of consumer threads, it is recommended to follow a similar outline as above for simplicity:

1. Increase the global bank operation counter (bank_numop).
2. Extract the element from the queue that corresponds to the previous counter.
3. Process the operation by the worker.

4. Update the corresponding account balance and the global bank balance (bank_balance) for each consumer thread to obtain the total balance.
5. When all operations have been processed, finish the thread with **pthread_exit**.

NOTE

To control concurrency, **mutex** and **condition variables** must be used. Concurrency can be managed in the producer and consumer functions, or in the code for the circular queue (**queue.c**). The choice is up to the lab group.

Global Variables In addition to the circular queue, the following global variables must be managed with mutexes and condition variables:

- `list_client_ops`, array with the list of operations to be executed. All ATMs access to this variable.
- `client_numop`, which starts at zero and is incremented concurrently each time a client is about to perform an operation from an ATM.
- `bank_numop`, which starts at zero and is incremented concurrently each time a worker is about to perform an operation extracted from the shared circular queue with the ATMs.
- `global_balance`, which is modified each time a deposit or withdrawal is made.
- `account_balance[i]`, update the balance of account [i]. It is recommended to generate a vector that can hold the maximum number of accounts and keep the balance indexed by account number.

Queue on a circular buffer Communication between producers and consumers will be implemented through a shared circular queue. A shared circular buffer should be created for producers and the consumer. Since modifications to this element will constantly occur, mechanisms for controlling concurrency must be implemented for lightweight 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 among the threads that are working with it. For this, the proposed mechanisms of **mutex** and **condition variables** must be used.

The object that must be stored and extracted from the circular queue **must correspond to a structure** (struct element).

1.2 Initial Code

To facilitate the realization of this lab, the file is provide:

```
os_p3_multithread_2023.zip
```

Which contains supporting code. To extract the content, you can execute the following:

```
unzip os_p3_multithread_2023.zip
```

After extracting its content, the directory p3_multithread_2023/ 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.
- **bank.c**
  Should be modified. C source file where the students should code the requested program.
- **queue.h**
  Should be modified. Header file where the students should define data structures and functions used to manage the circular queue.
- **queue.c**
  Should be modified. C source file where the students should implement functions for the management of the circular queue.
- **checker_os_p3.sh**
  Should not be modified. Shell script that performs a guided self-correction of the assignment. It shows one by one, the test instructions to be performed, the expected result, and the result obtained by the student's program. Ultimately, a tentative grade for the assignment's code is given (without considering manual review or the report). To execute it, the file must be given execution permissions:

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

And run it with:

```
./checker_os_p3.sh <code_zip_file>
```

- **authors.txt**
  Should be modified. `txt` file where the authors of the assignment should be included.
- **file.txt**
  Support file.

NOTE

New files can also be generated (**recommended**), and new tests can be performed on them.

2 Assignment submission

2.1 Deadline and method

The deadline for the submission of the lab in AULA GLOBAL will be **May 12th, 2023 (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_BBBBBBBB_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:

- **bank.c**
- **queue.c**
- **queue.h**
- **Makefile**
- **authors.txt: Text file in CSV format with the authors' information in different lines. The format should be: 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 bank.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`)