uc3m | Universidad Carlos III de Madrid

Degree in Applied Mathematics and Computing

*Operating Systems*

2022-2023

*Lab 3*

# Multithreading

---

Ilan Halioua Molinero | 100472908
100472908@alumnos.uc3m.es

Alejandro Fernández - Paniagua | 100473072
100473072@alumnos.uc3m.es

Alex David Slover | 100500292
100500292@alumnos.uc3m.es

Group 121

# TABLE OF CONTENTS

# 1. Description of the code

The code represents a multi-threaded banking system simulation, where multiple producer threads (ATMs) produce bank operations, and multiple consumer threads (workers) execute these operations from a circular queue. The objective is to process these operations correctly, maintain account balances and print them accordingly.

We begin by including the necessary header files and defining some constants and variables. We also declare the structures and global variables required for the simulation.

The main program starts by parsing the command-line arguments and reading input from a file. The input file contains a list of bank operations, each represented by a line. The first line of the file specifies the total number of operations (max_operations), and the subsequent lines contain the details of each operation, including the operation type (CREATE, DEPOSIT, TRANSFER, WITHDRAW, or BALANCE) and the associated account numbers and amounts.

After reading the input file, the program initializes the circular queue and creates producer and consumer threads. The number of producer threads is determined by the second command-line argument (num_ATMs), and the number of consumer threads is determined by the third command-line argument (num_workers).
Each producer thread is responsible for producing a certain number of operations specified by the max_operations variable.

Inside the ATM function, a producer thread first signals its start and waits for other threads to acknowledge the start. Then, it enters a loop to produce operations and enqueue them in the circular queue.
Within the production loop, the producer thread acquires a mutex lock to ensure exclusive access to the shared variables. It checks if the circular queue is full, and if so, it waits using a condition variable notFull until a consumer thread signals that the queue is not full anymore.

While the queue is not full, the producer thread dequeues an operation from the list_client_ops array and enqueues it into the circular queue using the queue_put function. It then signals the consumers using the notEmpty condition variable to indicate that the queue is not empty anymore. Finally, the producer thread releases the mutex lock.

The consumer threads execute the Worker function. Similar to the producer threads, each consumer thread signals its start and waits for other threads to acknowledge the start. The consumer threads enter a loop and continue executing specified operations until the worker_numop variable reaches the specified number of operations entered by the user.

Within the loop, the consumer thread acquires the mutex lock and checks if the circular queue is empty. If it is, the consumer thread waits using the notEmpty condition variable until a producer thread signals that the queue is not empty anymore. If the queue is empty and the ‘end’ flag is set, it means all operations have been executed, and the consumer thread exits.
Once the queue is not empty, the consumer thread dequeues an operation from the circular queue using the queue_get function. It processes the operation based on its type (CREATE, DEPOSIT, TRANSFER, WITHDRAW, or BALANCE), updates the account balances and the total(global) balance, and prints relevant information about the operation.

After processing the operation, the consumer thread increments the worker_numops counter, signals the producers using the notFull condition variable to indicate that the queue is not full anymore, and releases the mutex lock.

Once all producer and consumer threads have completed their execution, the program cleans up the allocated resources and exits.

## 2. Test cases

| **Test_ nº** | **Input Data:** MSH>>... | **Test description:** *(Motivation)* | **Expected Output:** | **Actual Output:** |
|---|---|---|---|---|
| ERROR TESTS | | | | |
| 1 | ./bank file.txt -3 4 10 10 | negative num_ATMs argument | "Insert positive arguments please" return -1 | "Insert positive arguments please" return -1 |
| 2 | ./bank file.txt 8 -3 10 10 | negative num_workers argument | | |
| 3 | ./bank file.txt 8 34 -5 10 | negative max_accounts argument | | |
| 4 | ./bank file.txt 8 34 10 -6 | negative buff_size argument | | |
| 5 | ./bank file.txt 8 34 | less arguments than expected | "Invalid number of arguments. Usage: ./bank <file name> <num ATMs> <num workers> <max accounts> <buff size>" return -1 | "Invalid number of arguments. Usage: ./bank <file name> <num ATMs> <num workers> <max accounts> <buff size>" return -1 |
| TESTS WITH FILES *See files used in appendix | | | | |
| 6 | ./bank notExistentFile.txt 8 3 10 10 | input file does not exist | “Error opening file: No such file or directory” return -1 | “Error opening file: No such file or directory” return -1 |
| 7 | ./bank emptyFile.txt 8 34 10 10 | input file is empty | “Error retrieving data from input file” exit(-1) | “Error retrieving data from input file” exit(-1) |

| | | | | |
|---|---|---|---|---|
| 8 | ./bank 230.txt 8 5 10 10 | max_operations > 200 | "Maximum number of operations is: 200" return -1 | "Maximum number of operations is: 200" return -1 |
| 9 | ./bank 50-20.txt 8 5 10 10 | max_operations > actual number of operations in file | "Number of operations in file should match with the value in the first line" return -1 | "Number of operations in file should match with the value in the first line" return -1 |
| TESTING CONCURRENCY WITH DIFFERENT VALUES OF num_ATMS AND buff_size *file 20-20 in appendix 4.1 | | | | |
| 10 | ./bank 20-20.txt 8 5 10 10 | More producers than consumers | See appendix 4.5 | As expected |
| 11 | ./bank 20-20.txt 5 8 10 11 | More consumers than producers | See appendix 4.5 | |
| 12 | ./bank 20-20.txt 8 8 10 12 | Equal number of producers and consumers | See appendix 4.5 | |
| 13 | ./bank f m n a q<br>./bank f n m a q | For any existing file, accounts and queue size, the program works correctly, resulting in the same output given that there are m producers and n consumers or n producers and m consumers. | ./bank f m n a q = ./bank f n m a q | ./bank f m n a q = ./bank f n m a q |

# 3. CONCLUSION

## 3.1. Problems found, how they have been solved, and personal opinions.

Working on this project has provided us with valuable lessons in multi-threaded programming, synchronization, and data sharing. We have learned how to efficiently manage shared buffers between multiple producer and consumer threads through the implementation of a circular queue. Creating the threads for producers and consumers was not so difficult but we had some issues with getting the threads to communicate properly with each other. At one point, we had a producer thread who would not unlock mutex after the buffer was full and denied control to the consumer thread in order to do the operations. We fixed this by writing this code: `pthread_cond_wait(&notFull &mutex)` which made the producer thread wait when the circular buffer is full in order to continue functioning when buffer space becomes available. We did not think initializing the variables and taking input from the file or creating the threads were very difficult. The difficulty was in trying to get the threads to communicate properly with each other using the mutex locks and signals so that the process was completed in the proper order. Coordinating our threads using proper synchronization mechanisms, such as mutex locks and condition variables, has ensured thread safety and prevented race conditions. This lab has also emphasized the importance of error handling and graceful termination, as well as effectively utilizing command-line arguments and file input for program configurability. Overall, this project has offered us practical experience in handling concurrent systems, emphasizing thread coordination, shared data management, and error handling techniques.

# 4. APPENDIX

## 4.1. 20-20.test

```
20
CREATE 1
DEPOSIT 1 100
CREATE 2
DEPOSIT 2 50
TRANSFER 1 2 30
TRANSFER 2 1 20
WITHDRAW 1 40
WITHDRAW 2 60
BALANCE 1
BALANCE 2
CREATE 3
DEPOSIT 3 20
CREATE 4
DEPOSIT 4 500
TRANSFER 3 4 10
TRANSFER 4 3 200
WITHDRAW 3 5
WITHDRAW 4 100
BALANCE 3
BALANCE 4
```

## 4.2. 20-50.test

```
20
CREATE 1
DEPOSIT 1 100
CREATE 2
DEPOSIT 2 50
TRANSFER 1 2 30
TRANSFER 2 1 20
WITHDRAW 1 40
WITHDRAW 2 60
BALANCE 1
BALANCE 2
CREATE 3
DEPOSIT 3 20
CREATE 4
DEPOSIT 4 500
TRANSFER 3 4 10
TRANSFER 4 3 200
WITHDRAW 3 5
WITHDRAW 4 100
BALANCE 3
BALANCE 4
CREATE 5
DEPOSIT 5 1000
CREATE 6
DEPOSIT 6 250
```

```
TRANSFER 5 6 600
TRANSFER 6 5 150
WITHDRAW 5 80
WITHDRAW 6 300
BALANCE 5
BALANCE 6
CREATE 7
DEPOSIT 7 300
CREATE 8
DEPOSIT 8 30
TRANSFER 7 8 25
TRANSFER 8 7 50
WITHDRAW 7 250
WITHDRAW 8 5
BALANCE 7
BALANCE 8
CREATE 9
DEPOSIT 9 300
CREATE 10
DEPOSIT 10 30
TRANSFER 9 10 25
TRANSFER 10 9 50
WITHDRAW 9 250
WITHDRAW 10 5
BALANCE 9
BALANCE 10
```

### 4.3. 50-20.test

```
50
CREATE 1
DEPOSIT 1 100
CREATE 2
DEPOSIT 2 50
TRANSFER 1 2 30
TRANSFER 2 1 20
WITHDRAW 1 40
WITHDRAW 2 60
BALANCE 1
BALANCE 2
CREATE 3
DEPOSIT 3 20
CREATE 4
DEPOSIT 4 500
TRANSFER 3 4 10
TRANSFER 4 3 200
WITHDRAW 3 5
WITHDRAW 4 100
BALANCE 3
BALANCE 4
```

### 4.4. 230.test

```
230
CREATE 1
DEPOSIT 1 100
```

```
CREATE 2
DEPOSIT 2 50
TRANSFER 1 2 30
TRANSFER 2 1 20
WITHDRAW 1 40
WITHDRAW 2 60
BALANCE 1
BALANCE 2
CREATE 3
DEPOSIT 3 20
CREATE 4
DEPOSIT 4 500
TRANSFER 3 4 10
TRANSFER 4 3 200
WITHDRAW 3 5
WITHDRAW 4 100
BALANCE 3
BALANCE 4
.
.
.
```

### 4.5. 20-20.test expected output (and actual output)

1 CREATE 1 BALANCE=0 TOTAL=0
2 DEPOSIT 1 100 BALANCE=100 TOTAL=100
3 CREATE 2 BALANCE=0 TOTAL=100
4 DEPOSIT 2 50 BALANCE=50 TOTAL=150
5 TRANSFER 1 2 30 BALANCE=80 TOTAL=150
6 TRANSFER 2 1 20 BALANCE=90 TOTAL=150
7 WITHDRAW 1 40 BALANCE=50 TOTAL=110
8 WITHDRAW 2 60 BALANCE=0 TOTAL=50
9 BALANCE 1 BALANCE=50 TOTAL=50
10 BALANCE 2 BALANCE=0 TOTAL=50
11 CREATE 3 BALANCE=0 TOTAL=50
12 DEPOSIT 3 20 BALANCE=20 TOTAL=70
13 CREATE 4 BALANCE=0 TOTAL=70
14 DEPOSIT 4 500 BALANCE=500 TOTAL=570
15 TRANSFER 3 4 10 BALANCE=510 TOTAL=570
16 TRANSFER 4 3 200 BALANCE=210 TOTAL=570
17 WITHDRAW 3 5 BALANCE=205 TOTAL=565
18 WITHDRAW 4 100 BALANCE=210 TOTAL=465
19 BALANCE 3 BALANCE=205 TOTAL=465
20 BALANCE 4 BALANCE=210 TOTAL=465