OPERATIVE SYSTEMS

GROUP 121

APPLIED MATHEMATICS AND COMPUTING

Lab Report 03

Authors	*NIA*
IGNACIO AGUADO SOBRADILLO	100496991
LUCAS GUZMÁN BASTIDA	100496813
OLAYA MARTÍNEZ FERNÁNDEZ	100495748

Emails

100496991@alumnos.uc3m.es (Ignacio)
100496813@alumnos.uc3m.es (Lucas)
100495748@alumnos.uc3m.es (Olaya)

Table of Contents

1 Description of the code

1.1 queue.c

In this file a number of functions are defined. The first function is a function used for creating queues. After allocating memory for the array, it sets attributes head, tail, and size of the queue to 0 and sets the attributes max size to an input parameter, size. Furthermore, it allocates memory for the element of the queue and finally, it returns a pointer to the queue.

The second function's goal is to enqueue an element. Firstly, it checks that the queue is not full, in which case it copies the element received by the function in the position of the array indicated by the attribute tail, as it must be inserted at the end of the queue. Then, it updates the position of the attribute tail to the next position of the queue following a circular scheme: if the tail position exceeds array limits, the index goes back to the beginning of the queue, as it is given by the remainder between the next position and maximum size of the queue. Finally, it increments the size of the queue. In the case in which the queue is full, it returns -1.

The next function is used to dequeue an element. Initially, it checks if the queue is empty. If it is not, it creates a pointer element pointing to the head of the queue and updates its position to the next position of the queue, without exceeding array limits in the same way as in the previous function. Finally, it decreases the size of the queue and returns the element. In the case in which the queue is empty, it returns NULL.

The two following functions are quite similar. The first one evaluates whether the queue is empty by returning 1 if the size is 0, or returning 0 if it is not. The second one evaluates whether the queue is full or not, returning a 1 if the size is equal to the maximum size, and 0 otherwise.

The next function's goal is to destroy the queue. If the queue pointer is not NULL, it frees the memory allocated by the elements as well as for the queue.

Lastly the function `print_queue` enters a loop iterating from the first to the last position of the queue. In each iteration, it creates a pointer element pointing at the position `i` of the queue, which is reached by summing the number `i` to the head of without exceeding array limits in the manner specified in the other functions. Then, it prints all the information of the element.

1.2 store_manager.c

Firstly, all global variables are declared, the array for operations, the queue shared buffer, and the integer profits; which is initialized to 0, the product stock array, mutexes, condition variables, the number of total operations, the number of operations processed and a variable to indicate whether the producers have finished or not.

The producer's routine starts with the conversion of the range pointer from void to integer in order to obtain elements from the array with it. `start` and `end` integer variables are defined and used to define a range for the following while loop. At the beginning of the loop, the element at the index `start` is taken from the operations array. Then, it enters a critical part in which the mutex object is locked in order to store correctly operation in the shared buffer. Next, the program enters a while loop until the queue is not full, in which it unlocks the mutex and waits. Once the loop is finalized, the element is added to the buffer and then unblocks the threads waiting for the queue to be non-empty with the broadcast function. Lastly in the critical part, it unlocks the mutex. Finally, it increments the `start` in order to access the next element in the array.

The consumer routine starts directly with a `while`. It stays in the loop while there exist threads that are executing. The `while` stops when the thread ends. It then enters a critical part in which the mutex object is locked to retrieve correctly the operation in the shared buffer. But before this, we use a variable called `end_prods`, which takes the value 1 if all the producer threads have finished. If all have finished, the consumer threads will be terminated by the other consumer threads. Then, if the shared buffer is empty, it unlocks the mutex until there can be something to read in the buffer. Once we can read something from the buffer, we use the `queue_get` operation of the shared buffer to get the element we are going to process. If the shop purchases, we increment the stock and subtract the cost from the profit. If the shop sells, we decrease stock and sum the price of the item times the units to the profit. Finally, it increments by one the `consumer_processed_ops` and if the value of `consumer_processed_ops` is greater or equal to `num_ops` (which is the total number of operations), it changes to 1 the value of `end_prods`, unlocks the mutex, sends the signal that the buffer is not empty and the thread exits.

2 Test Cases

Various cases with file.txt

```
./store_manager file.txt  10 10 50
Total: 120 euros
Stock:
  Product 1: 20
```

```
  Product 2: 60
  Product 3: 20
  Product 4: 18
  Product 5: 2
```

```
./store_manager file.txt  1 1 50
Total: 120 euros
Stock:
  Product 1: 20
  Product 2: 60
  Product 3: 20
  Product 4: 18
  Product 5: 2
```

```
./store_manager file.txt  10 10 10
Total: 120 euros
Stock:
  Product 1: 20
  Product 2: 60
  Product 3: 20
  Product 4: 18
  Product 5: 2
```

```
./store_manager file.txt  1 1 1
Total: 120 euros
Stock:
  Product 1: 20
  Product 2: 60
  Product 3: 20
  Product 4: 18
  Product 5: 2
```

```
./store_manager file.txt 4 10 27
Total: 120 euros
Stock:
  Product 1: 20
  Product 2: 60
  Product 3: 20
  Product 4: 18
  Product 5: 2
```

Non-valid arguments

```
./store_manager
[ERROR]: usage ./store_manager <file_name><num_producers><
    num_consumers><buff_size>
```

```
./store_manager file.txt  10 10
[ERROR]: usage ./store_manager <file_name><num_producers><
    num_consumers><buff_size>
```

```
./store_manager file.txt  10 10 50 15
[ERROR]: usage ./store_manager <file_name><num_producers><
    num_consumers><buff_size>
```

```
./store_manager file.txt 4 10 0
[ERROR]: usage ./store_manager <file_name><num_producers><
    num_consumers><buff_size>
```

File that does not exist

```
./store_manager file33.txt  10 10 50
Error opening the file: No such file or directory
```

Various cases with customCase.txt

We've also made a custom file with 10 operations:

```
10
1 PURCHASE 39
2 PURCHASE 3
3 PURCHASE 13
4 PURCHASE 4
5 PURCHASE 50
1 SALE 6
2 SALE 3
3 SALE 11
4 SALE 3
5 SALE 48
```

`customCase.txt`

If we operate, we get that the benefit of these operations is 1.000€, which is what we get in our program

```
./store_manager customCase.txt 4 3 5
Total: 1000 euros
Stock:
  Product 1: 33
  Product 2: 0
  Product 3: 2
  Product 4: 1
  Product 5: 2
```

```
./store_manager customCase.txt 5 5 10
Total: 1000 euros
Stock:
  Product 1: 33
  Product 2: 0
  Product 3: 2
  Product 4: 1
  Product 5: 2
```

Then we changed the 9^{th} line of our file by `3 OBTAIN 11`, which is not a valid operation, to check that our program handles this error:

```
./store_manager customCase.txt 5 5 2
Some operation is not valid
```

Conclusion

During the completion of the assignment, we have encountered a series of challenges. Working with threads usually involves issues such as having different results because of the race conditions or the threads not closing properly because of never reaching the condition to exit. One of the major issues has been when consumers tried to get information from the buffer when there were no producers left. The solution was to create a variable to indicate whether producers have ended or not so that in the affirmative case, consumers could be unlocked.