Operating Systems

Third Assignment Report
Multithread

Members: Carlos Iborra Llopis / Pablo Brasero Martínez / Marcos Caballero Cortés
NIAs: 100451170 / 100451247 / 100451047
Group: 89

Table of Contents

1. Code Description

In this third Operating System project, we have been asked to create a program that allows the students to understand and become familiar with the services used for the administration of processes provided by POSIX. In addition, we are given some functions (*pthread_create*, *pthread_join*, and *pthread_exit*) in order to use them throughout the development of our program, in addition, we were also recommended the use of mutexes and conditional variables. Each of these functions has a different utility which will help us to accomplish the different required tasks asked for in this project.

1.1. costCalc.c

costCalc.c file is in charge of calculating the price that it will cost us to use a machine in a given time. We have three types of machines that have their individual cost of use per minute. The first machine, “common node” has a cost of €3 per minute, the second machine, “computation node” has a cost of €6 per minute, and the last machine, “super-computer” has a cost of €15 per minute. With this information, the program calculates the total cost of the calculation, aggregating all different machines with their types and their running time.

costCalc.c is composed of the three following functions:
The main function which is in charge of executing all the processes, a cproducer function in charge of adding elements to the shared circular queue, and the cconsumer function in charge of extracting elements from the shared circular queue.

1.1.1. Main

Our main function is responsible for executing all possible actions within the program, it begins by allocating memory for the name of the file to be used. Then the program checks that the amount of arguments is the correct one (5) and assigns argument 1 to the variable *path_file*, the second to the variable *producers_number*, the third to the variable *consumers_number*, and the fourth argument to the *queue_size* variable. If the entered number of arguments was other than 5, it will raise an error. Later the program tries to open the main file and in case it can not be opened an error will be raised. It also counts the number of lines in the file, which value is stored into a variable and else, raises an error. The following part is in charge of raising errors, in the first case if there are more operations than lines, this is coded in line 1 of *m_file* being specifier of the number of operations to be performed, in the second case, an error is raised if the number of producers or consumers is wrong, in the third case, if there are more producers than operations an error will also be raised, in the fourth error case, an error will be raised if the queue size is smaller than zero. In the case either the mutex initialization or the *mutex_file* initialization is wrong, an error will be raised individually for each of them. Two more errors can happen when initializing the conditional variables, one when *non_full* conditional variables are being initialized and the other when non_empty conditional variables are being initialized, making the program raise an error for each of them independently.

For the next block of this function, the operations per producer are defined, then, the threads for the producer and the consumer are created, as well as the parameters. Then, it starts the

creation of the producers' threads. The process starts at *id_operation*, then the operations number of the producer is defined and finally, the producers' threads are created. In the case there is an error when creating these threads, an error message will be raised. Finally, the program goes to the next operation with the last producer.

The following process is to create the consumers' threads, and an error will be raised in case joining producers' threads does not work.
The next part of the code consists of the termination of the different processes. In the first place, it waits for all the threads to finish, then the threads of the producers are joined, in case of error, an error message will be raised. Having finished this process, the same action is done for consumer threads, which will join it, and in case of error, a message will be raised.

After the previous processes, the *m_file* is closed, and once again if the process is not successful an error message will be raised.

The final part of the code is dedicated to the destruction of both the mutexes and the conditional variables and the queue (circular buffer), in the first place the mutexes are destroyed, if any operation goes wrong, the program will raise the corresponding errors. We continue with the destruction of the conditional variables, which destroys both the *non_empty* conditional variable and the *non_full* conditional variable. Once again the program will raise the corresponding errors if any of those processes are not successful. Next, the queue is destroyed and a message with the total value in euros is printed on the screen. Finally, we return a 0 meaning the program has been executed correctly (as when errors occur we return a -1).

1.1.2. CProducer

The CProducer (circular producer) function is executed by several threads (as there are from 1 to many producers) and will be in charge of adding the elements to the shared circular buffer (queue). It can be seen that within the CProducer function, first, the function receives the necessary parameters, later within the function itself, errors will be raised in case either the mutex file is locked or the file can't be opened, the program also counts the lines in the file. Afterward, while the buffer is full, the program will wait until there is a hole to insert the data, when a hole is found, the following piece of code is executed for inserting data into the queue, therefore fulfilling all the functions required for this function.

1.1.3. CConsumer

This second function, CConsumer (circular consumer), is executed by several threads (as there are from 1 to many consumers as well) and is in charge of extracting the elements of the shared circular buffer (queue). Observing that the realization of this process is given in this case, the CConsumer function performs the opposite action to that previously seen in CProducer. In this case, it is executed in such a way that the function first waits in case the buffer is empty, that is, there is no element to extract from the queue.

1.2. queue.c

Inside this file, we created six different functions, each one of them destined to carry a different task. We first create a function for initializing the queue (*"queue_init"*), allocating space for it, and creating the different values of the queue, such as the size, the values in it, the number of entries, and the head and tail. Our second function is used to enqueue elements (*"queue_put"*), which means to insert elements inside the queue, this function modifies the number of values, the tail, and the number of entries. We continue with a function whose purpose is to dequeue an element (*"queue_get"*), this function also modifies the number of elements and the number of entries. We also have a function that checks the queue state (*"queue_empty"*), which checks whether the queue is empty. Next, we have created a function that checks if the queue is full (*"queue_full"*). Last but not least, we created a function in charge of destroying the queue (*"queue_destroy"*).

1.3. queue.h

This file is a header file and it is dedicated to defining the variables and calling the functions inside the *queue.c* file so they can be used.

Firstly, we define the structure element which contains the type of machine used, and the use time of the given machine.

Then we define the structure queue which contains the previous structure called as *values*, and the integers *head*, *tail*, number of entries (*num_entries*), and *size*.

Finally, we call the different queue operations created inside *queue.c*.

2. Created Test Cases

Test case 1: Valid case

Input	./calculator file.txt 20 25 40
Expected output	1020360 euros.
Output	Total: 1020360 euros.
Description of output	This is a valid case.

Test case 2: Fewer arguments than expected

Input	./calculator file.txt 20 25
Expected output	We expect an error.
Output	Error: Invalid number of arguments: Success
Description of output	Here we wanted to test what would happen when the input has fewer parameters than expected. As you observe, an error occurs

Test case 3: More arguments than expected

Input	./calculator file.txt 5 5 20 21
Expected output	We expect an error related to an invalid input.
Output	Error: Invalid number of arguments: Success
Description of output	Here we wanted to check what would happen if we insert in the input more than 5 parameters. As you observe, an error occurs.

Test case 4: Negative time on the file

Input	./calculator file.txt 5 5 20
Expected output	We expect 1008444 euros as there are no specifications on negative time in the lab statement.
Output	Total: 1008444 euros.
Description of output	Here we wanted to check if we would get an error if there was an operation with a negative time on file.txt. As you can observe, there was no error and the total amount gets subtracted by the ones with negative time.

Test case 5: Negative time on our file2

Input	./calculator file2.txt 1 1 20
Expected output	We expect -333.
Output	Total: -333
Description of output	In this test, we checked the same thing as the one checked previously for file.txt would happen in file2.txt, but with only one operation. As you can see this test did not raise any error.
Content of file2.txt	1 1 1 -111

Test case 6: Wrong machine type

Input	./calculator file.txt 5 5 20
Expected output	We expect an error related to the machine type.
Output	ERROR: wrong type format: Success
Description of output	Here we see what happens when in the file.txt there is one operation with a machine type different than 1, 2, or 3. As you can observe, an error was raised.

Test case 7: Wrong machine type on our file2

Input	./calculator file2.txt 1 1 20
Expected output	We expect an error related to the machine type.
Output	ERROR: wrong type format: Success
Description of output	Here we tried to check the same test as the previous one but with file2.txt instead of file.txt. This file only has one operation so we could check that it would raise an error independently of the number of operations and file used. As you can observe, an error was raised.
Content of file2.txt	1 1 4 111

Test case 8: Negative number of producers

Input	./calculator file.txt -5 5 20
Expected output	We expect an error related to the expected number of producers.
Output	ERROR: wrong producers OR consumers number: Success
Description of output	In this test we check what would happen if we created an input with a negative number of producers. This way we make sure that a negative number of producers is not allowed. As you can observe, an error was raised.

Test case 9: Negative number of consumers

Input	./calculator file.txt 5 -5 20
Expected output	We expect an error related to the expected number of consumers.
Output	ERROR: wrong producers OR consumers number: Success
Description of output	In this test we check what would happen if we created an input with a negative number of consumers. This way we make sure that a negative number of consumers is not allowed. As you can observe, an error was raised.

Test case 10: Negative size of the buffer

Input	./calculator file.txt 5 5 -20
Expected output	We expect an error related to an invalid input.
Output	ERROR: queue size must be greater than 0: Success
Description of output	Here we tested what happens when a negative size buffer is tried to be implemented, an error is raised as the size of the buffer must be bigger than 0. As you can observe, an error was raised.

Test case 11: Size of the buffer is 0

Input	./calculator file.txt 5 5 0
Expected output	Error related to an invalid input.
Output	ERROR: queue size must be greater than 0: Success
Description of output	Here we tested what happens when a buffer of size 0 is tried to be implemented, similarly to what happened in the previous test, an error was raised as the size of the buffer must be bigger than 0.

Test case 12: Fewer number of operations than defined

Input	./calculator file.txt 5 5 20
Expected output	Error related to the number of operations defined.
Output	ERROR: wrong operations number: Success
Description of output	Here we tried to test what would happen if we defined fewer operations than the number of operations defined at the beginning of the file. As you can observe, an error was raised. For example: **300** 1 1 20 … **100** 1 30

Test case 13: More number of operations than defined

Input	./calculator file.txt 5 5 20
Expected output	We expect 61587 euros.
Output	Total: 615867 euros.
Description of output	Here we tried to test what would happen if we defined more operations than the number of operations defined at the beginning of the file. What happens here is that the program only calculates the cost for the number of operations defined at the beginning of the file. For example: **300** 1 1 20 … **400** 1 30

Test case 14: More producers than operations in our file2

Input	./calculator file2.txt 5 5 20
Expected output	We expect an error related to the number of producers and operations.
Output	ERROR: more producers than operations: Success
Description of output	Here we check what would happen if the number of producers is greater than the number of operations defined. In our input, file2.txt is a file with only 1 operation. As you can observe, an error was raised.
Content of file2.txt	1 1 1 111

3. Teacher Tests

Apart from our own created tests, we have also checked the test file given to us by the teacher. As a result of the execution, we have obtained a grade of 10 out of 10 when running it in the terminal console as you can observe below:

```
*** TESTING P3
File : ssoo_p3_100451170_100471247_100451047.zip
Archive:  ssoo_p3_100451170_100471247_100451047.zip
  inflating: authors.txt
  inflating: costCal.c
  inflating: Makefile
  inflating: queue.c
  inflating: queue.h
Compiling
OK
File:  500-500.test
Nprod:  1
MCons:  2
Buffsize:  3
Extra arg:  4
 OK
1


File:  500-500.test
Nprod:  0
MCons:  1
Buffsize:
 OK
2


File:  500-500.test
Nprod:  1
MCons:  0
Buffsize:
 OK
3


File:  500-500.test
Nprod:  -1
MCons:  1
Buffsize:
 OK
4


File:  500-500.test
Nprod:  1
MCons:  -1
Buffsize:
 OK
5


File:  500-500.test
```

```
Nprod:  3
MCons:  2
Buffsize:  10
 OK
Total: 1030146 euros.
6


File:  2000-500.test
Nprod:  3
MCons:  2
Buffsize:  10
 OK
Total: 1030146 euros.
7


File:  2000-2000.test
Nprod:  3
MCons:  2
Buffsize:  10
 OK
Total: 4120584 euros.
8


File:  500-2000.test
Nprod:  3
MCons:  10
Buffsize:
 OK
9


File:  2000-2000.test
Nprod:  1
MCons:  1
Buffsize:  10
 OK
Total: 4120584 euros.
10


File:  2000-2000.test
Nprod:  1
MCons:  1
Buffsize:  2000
 OK
Total: 4120584 euros.
11


File:  2000-2000.test
Nprod:  3
MCons:  3
Buffsize:  10
 OK
Total: 4120584 euros.
12
```

```
File:  2000-2000.test
Nprod:  4
MCons:  4
Buffsize:  50
 OK
Total: 4120584 euros.
13


File:  2000-2000.test
Nprod:  5
MCons:  5
Buffsize:  500
 OK
Total: 4120584 euros.
14


File:  2000-2000.test
Nprod:  6
MCons:  6
Buffsize:  2000
 OK
Total: 4120584 euros.
15


File:  2000-2000.test
Nprod:  4
MCons:  4
Buffsize:  2000
 OK
Total: 4120584 euros.
16


File:  2000-2000.test
Nprod:  5
MCons:  5
Buffsize:  200
 OK
Total: 4120584 euros.
17


File:  2000-2000.test
Nprod:  6
MCons:  6
Buffsize:  10
 OK
Total: 4120584 euros.
18


Grade: 10.00
```

4. Conclusion

In conclusion, we wanted to mention how this project along with the previous ones has made us increase our knowledge regarding the C language and all the different Operating System functionalities we have used throughout all the three projects as well as an increase in knowledge on how to use Linux Ubuntu 20.4 LTS terminal and system.

In addition, this project has helped us a lot in the understanding of threads, mutexes, and conditional variables (which was very helpful in the exam) as well as how buffers (specifically circular buffers) work when used in the consumer-producer problem. We remark how useful it was for us to have all the operating system Aula Global documents along with the teachers' help and knowledge in order to complete this project.

To finish the conclusion, we wanted to highlight how we have organized all the work, we all have done part of the code and part of this document sharing with the others all the information gained, this way we assured we all understood everything and that made sense for the three of us. An example of this teamwork organization is how we arranged to use just one programming style, using snake_case for naming the different methods, variables, and functions names and using the "Allman" indentation method for the brace placement in compound statements for our project code readability. We also fixated the indentation length to 4 spaces. This shows how we have organized as a whole in order to achieve our goals faster, more efficiently, and better.