Bachelor in Computer Science and Engineering

Operating Systems 2024-2025
Grupo 89

Laboratory 3

"Multi-thread programming"

Jorge Adrian Saghin Dudulea – 100522257
Denis Loren Moldovan – 100522240
Antonio Nicolas Lemus Yeguas – 100522110

Índice

Parte I

Description of the code

The laboratory is composed of three source files, all compiled into the same one. Therefore, multiple header files had been used, each one having their respective import protections and function declarations.

1. queue.c

This file is responsible for the manipulation of the different queues created during the duration of the program. A single queue can be used at the same time, following the statement of the exercise, therefore, without the need of implementing a more complex algorithm to control more queues.

1.1. queue_init

Function responsible for intializing the queue used by the current belt, doing the necessary checks of the input data, starting the semaphores and the mutex.

1.2. queue_destroy

Destroys everything created by the afore mentioned function, clearing the remaining elements from the buffer that could not have been freed corectly during the belt processing.

1.3. queue_put

Puts an element into the queue, following the respective synchronization mechanisms. For this, three static global variables are going to be used, one for the position of the head, another for the tail, and the total count of elements inside the buffer.

To calculate the insertion position, we use the operation:

$$(head\_pos + 1)\,\%\ \text{current\_belt -> size}$$

Where the head position is the last place where we inserted an element, and we perform the modulus with the size of the belt to remain inside the belts boundaries.

1.4. queue_get

As putting an element, we are going to be calculating the position of the element to be getting, first retrieving said element from the buffer, and then updating the tail position usign a similar operation as the `queue_put` function:

$$(tail\_pos + 1)\,\%\,\text{current\_belt -> size}$$

1.5. queue_empty

Checks if the queue is empty by using the `count` variable.

1.6. queue_full

Same as `queue_empty`, but checks comparing it to the belt struct.

2. process_manager.c

This file contains the code for the threads responsable for manufacturing said belts passed as argument to a void pointer. It is divided into the manager thread `process_manager`, the producer and the consumer. All these make use of the queue functions.

2.1. process_manager

Initializes the producer and consumer threads, while parsing the input data and synchronizes them with the parent thread from the `factory_manager`. It is also responsable of creating and destroying the belts when the production of elements finishes.

2.2. producer

Produces the elements from the belts, while synchronizing with the consumer thread using a mutex to avoid both from modifying the belt or the global variables at the same time, and semaphores to have extra control over the amount of elements that are being produced and the ones that must be retrieved.

2.3. consumer

As already commented in the producer function, uses synchronization mechanisms to avoid modifying the belt incorrectly and global variables to retrieve the elements.

3. factory_manager.c

These functions are just responsible from parsing the inputted file, checking if the data is correct, and starting starting the required threads for running each belt *in order*.

3.1. main

As already mentioned, parses the input file, checks for errors, and starts each thread appropiately, while synchronizing with them to ensure the correct order. This was the most challenging part, as we had to implement multiple semaphores and mutexes to guarantee this requirement.

3.2. tokenizar_linea

Reused function from previous labs, modified to generalize two cases of lines to tokenize and improving error management.

3.3. parse_file

Another reused function, modified to correct a memory leak and multiple errors that were arising from not closing the source file, therefore leading to undefined behavior.

Parte II

Tests

When running the tester, we were encountering times where we were getting 73/74 and other ones with 74/74. We cheked and realized that the Test 6 was failing sometimes. Even though, we reached to the conclusion that sometimes the scheduling of the operating system was putting one of the threads before another one, therefore changing the output and failing the test.

Aside from the base tests provided in the tester, we are going to use the following ones to check manually if the solution follows the basic description of the statement.

Test ID	Description	Inputs	Expected Outputs
TC01	Single belt with valid values	Input file: `1` `1 5 5`	Producer and consumer complete 5 elements successfully
TC02	Two belts concurrently	Input file: `2` `1 5 5 2 10 10`	Both belts produce/consume all elements correctly
TC03	Belt with buffer size 1	Input file: `1` `1 1 3`	Queue operates with minimal size; producer/consumer synchronize correctly
TC04	Belt with zero size (invalid)	Input file: `1` `1 0 5`	Error message, program exits with failure
TC05	Belt with zero elements to produce (invalid)	Input file: `1` `1 5 0`	Error message, program exits with failure
TC06	Belt with negative ID (invalid)	Input file: `1` `-1 5 5`	Error message, program exits with failure
TC07	Belt with negative size (invalid)	Input file: `1` `1 -5 5`	Error message, program exits with failure
TC08	Belt with large number of elements	Input file: `1` `1 100 1000`	Program runs and processes all elements correctly
TC09	Multiple belts with varied sizes and loads	Input file: `3` `1 5 10 2 3 6 3 7 14`	All belts complete independently without interference
TC10	Invalid file with missing fields	Input file: `1` `1 5`	Error message, program exits with failure
TC11	Queue full scenario handling	Input file: `1` `1 2 5`	Producer waits for consumer when queue is full, and completes successfully

Test ID	Description	Inputs	Expected Outputs
TC12	Queue empty scenario handling	Input file: `1` `1 5 2`	Consumer waits for producer and completes correctly
TC13	Even belt count mismatch	Input file: `2` `1 5 5 2 10`	Error message, program exits with failure
TC14	File with non-numeric values	Input file: `1` `a b c`	Error message, program exits with failure
TC15	Empty input file	Input file: (empty)	Error message, program exits with failure
TC16	Input with extra tokens beyond expected	Input file: `1` `1 5 5 99`	Error message, program exits with failure
TC17	Producer fails memory allocation	Simulate malloc failure in `producer`	Error message, thread exits with failure
TC18	Consumer fails integrity check	Simulate mismatch in `id_belt`	Error message, thread exits with failure
TC19	Stress test with 100 belts	Input file: `100` followed by 100 belt definitions	All belts process correctly in parallel
TC20	Producer produces only 1 element	Input file: `1` `1 5 1`	Single element produced and consumed successfully

Parte III

Conclusion

This laboratory helped us understand the importante of concurrency and synchroniaztion mechanisms, while also teaching us how to approach them. The work with the tester is also an important part to mention, as it helped us find all the errors that we weren't able to acknowledge beforehand.

To conclude, it was a really interesting exercise, which we are looking forward to see more in other subjects.