# Operating systems: Laboratory practice 3 Multi-thread programming. Control of fabrication processes.


**Héctor Molina Garde.**

.
Computer Science Engineering Degree

**Nicolás Maire Bravo.**

Computer Science Engineering Degree

**Guillermo González Avilés.**

Computer Science Engineering Degree

# Code description:

## 1. **General design**:

The concurrency design of the manager is divided into two parts, the factory manager that handles manager thread order execution and the productor-consumer problem.

The first concurrency synchronization is made by 3 semaphores: the Waiting, the Finish and the Execute semaphore. First the semaphores are initialized in zero, except the first element of Waiting, where the factory manager will enter the critical section of creating all the threads and print the creation of them. After creating all of them, the factory manager posts the semaphore. Subsequently, a chain of semaphores starts in which each of the threads will wait for the activation of each Waiting semaphore as they instantly enter in a wait for Waiting. They then print that they are waiting and the semaphore of the next thread in the array order is posted so they are printed orderly. Once the last thread has printed that it is waiting, the first element of the Execute semaphore is posted. This starts the execution of the belt process and after finishing, the semaphore Finish is posted. The factory manager that is waiting for it can join the thread and print that the process ended successfully. And then, the next thread can be executed due to the post of the next element of Execute.

Additionally, the concurrency synchronization between the consumer and the producer is made through the prototype code of the problem. In this prototype a mutex is created and two conditional variables are made for indicating whether the circular queue is either full or empty with the corresponding management of it.

Structures for passing parameters to functions and the semaphores are created as global variables, so in other ‘.c’ files they can be called using ‘extern’. More clarifications of how concurrency systems are implemented in the code are given in the functions descriptions

## 2. **‘*factory_manager.c*’ File**:

a. tokenize_file().

We have decided to implement a copy of the tokenize system of the bash terminal when reading the file. The 'tokenize_file()' function passes all the arguments of the file into an array of strings as when they are typed in the terminal. They are then tokenized and passed from 'ASCII' to an integer type variable.

This is performed by firstly opening the file, which is passed as an argument of the function and the creation of a dynamic buffer in which it is intended to store the whole line for later modifications. Afterwards, the function enters into a loop. The first thing that the loop does is read the file, it has been planned in such a way that the pointer of the buffer and a variable for the total read size let the file information be extracted correctly into the buffer. Then a check is performed to see whether the content read is '-1' or '0'. At this point an error is raised or the end of file has been reached, so the loop is exited. The size of the total read of the file is updated with the read size, if the total read is equal to the buffer, the buffer is reallocated in order to follow more Reading iterations.

After the loop, another check of the total read size and the buffer size is checked in order to ensure space for the null terminator in the buffer line if ‘\n’ is the last character in the file. However, the file must be in just one line so the function detects when a new line exists

and raises an error. Following this, the file is closed and a new dynamic array 'argvv' is created. It will be in charge of the tokenization of the buffer line. Using 'strtok_r()' the buffer is stripped and another loop for turning each value to integer type is done. Additionally, after the casting of 'strtol(token, &endptr, 10)' the 'endptr' pointer checks if the token is a valid integer and exits the function with an error. Then, a pointer to the variable 'argc' which is passed as a parameter is checked if it is higher than the actual size of the 'argvv' array capacity, if the case is done a reallocation is performed. Finally, the 'argvv' array with the index 'argc' is given the value of the token of the loop's iteration. Finally, the buffer array is freed and the 'argvv' array is returned, as it is a dynamic array it must be freed after a successful function call.

### b. parse_file()

The 'parse_file()' function handles the conversion of a matrix from a plane array created by the 'tokenize_file()' function. The arguments of the function are: a string with the file path, a triple integer pointer to the matrix with the processes information, a pointer to the maximum number of processes and another pointer to the number of processes.

First, the 'tokenize_file()' function is called by passing the file and the count ('argc') (which were created previously by the call) as arguments. The returned array is assigned to a newly created array. If after the call, the pointer is still 'NULL' an error is returned. Then, the variable 'max_belts' is assigned the first value in the array of 'argvv', which is, as given by the statement, the maximum number of possible belts. Subsequently, it is checked whether the maximum number of belts is lower than one. If the argument count (‘argc’) after subtracting one is a multiple of three, as three values are given per process and if the number of processes is higher than the maximum number of belts. If any of these three conditions occur, an error is returned.

Next, the pointer of the processes matrix is allocated with the size of the maximum number of belts, to then allocate in a loop each row of the matrix with a size of three elements (the ID, the maximum belt size and the number of products). The matrix is populated in a double loop with the elements of the 'argvv' array. Before the value assigned in the matrix, if the maximum number of processes is higher than the actual number, the surplus of process rows are populated with zeros. Otherwise in the process assigning, the 'process ID' column is checked if in the previous iterated rows there are no ID duplications with the new assigning row.

The value to the process number pointer is given by the (argc-1)/3, the 'argvv' array is freed and '0' is returned.

### c. create_sync_struct()

The 'create_sync_struct()' function performs all the semaphore and thread executions of the program. It receives the matrix with the processes, the maximum number of belts and the process number. The execution of the threads are made possible by the use of semaphore arrays. Three dynamic arrays are created with the size of the process number. After the memory allocation of every array, the semaphores are initialized at ‘0’ except the first element of the waiting semaphore, the semaphore planning is previously explained.

The array of threads is then declared and as the program enters a critical section, the first element of the Waiting semaphore is waited by the factory manager, in order to make the threads wait before the creation of all of them. A pointer with the structure created for

passing all the arguments of the function of the thread is created and allocated in the heap, to later pass all the values to the structure. The thread is created with ‘pthread_create()’ passing the function of ‘process_manager’ and the structure pointer. After all the iterations the semaphore is posted.

A variable ‘status’ is declared to check if the threads end correctly. The program enters in a loop for each thread where they wait for the Finish semaphore and join the ended thread. Then, it checks for the status of the joined thread to check if it ended successfully. If the thread was not the last one, the next element of the Execute semaphore array is posted. Finally, all the semaphores are destroyed and the arrays freed.

### d. main()

The main function checks for the quantity of the arguments passed, creates the variables of the maximum number of belts, process number and matrix of processes. Calls to the ‘parse_file()’ function and then calls the ‘create_sync_struct()’ for the execution of the threads, Then the matrix is freed and is printed the factory manager has finished.

## 2. ‘*process_manager.c*’ File:

All the structures for passing function arguments, the mutex and the conditional variables are declared as global variables. Also the semaphores are declared as extern variables

### e. Producer()

The producer function just unpacks the arguments from the ‘ConsProdArgs’ structure and free it. Then for each of the elements to produce, the producer's prototype of code for producer-consumer problems is implemented. With the unique change that as in the queue code you cannot check for if the size is equal to one. Previously the insertion on the belt is stored if the belt is empty in order to set the conditional variable of empty as false after the insertion of an element on the belt, the element is allocated in the heap. As previously mentioned, the function is the same as the producer’s code.

### f. Consumer()

The consumer function just unpack the arguments from the ‘ConsProdArgs’ structure and free it. Then for each of the elements to produce, the consumer's prototype of code for producer-consumer problems is implemented. With the unique change that as in the queue code you cannot check for if the size is equal to the full size minus one. Previously, the consumption on the belt is stored if the belt is empty in order to set the conditional variable of full as false after the consumption of an element on the belt, the element is then freed from the heap. As previously mentioned, the function is the same as the consumer’s code.

### g. Process_manager()

The process manager function unpack all the elements from the structure which is passed as a pointer and pass its values to variables. The process waits the Waiting semaphore in order to print that “the process manager is waiting to produce” to then post the semaphore of the next ordered thread so the waitings are printed in order. If the thread is the

last one, the first semaphore of the Execute array is posted so the first thread could now start executing the production process.

Subsequently, the two threads are declared and the queue mutex and the conditional variables for ‘full’ and ‘empty’ are initialized. Afterwards, the queue is created and the structures’ elements whose pointers are the arguments for the threads of consumer and producer are created, and the values of them are given, so the threads are then created.

Despite having the same values as both structures (id and elements to produce), in order to avoid conflicts, it has been planned to create two different structure instances. Once the threads end, they are joined once they finish. The queue is destroyed and it is printed that the process has produced all the elements; the semaphore element from the array Finish is posted, the initial argument struct is freed and the function returned. Also, when an error occurs the Finish semaphore is also posted before returning ‘-1’.

## 3. ‘*queue.c*’ File:’

The queue system implemented is a  single linked queue of the structure ‘element’.

h. queue_init()

If the size is fewer than 0, an error is returned, the max size is setted to the input argument and the size is setter to zero, the head and tail are null and the count is zero.

i. queue_destroy()

The queue elements are freed one by one the head, the tail, the count and the queue size are setted to null and 0.

j. queue_put()

Performs a simple insert in the linked queue and prints the introduced element.

k. queue_get()

Performs a simple pop in the linked queue and prints the returned element.

l. queue_empty()

Checks if the queue count is 0

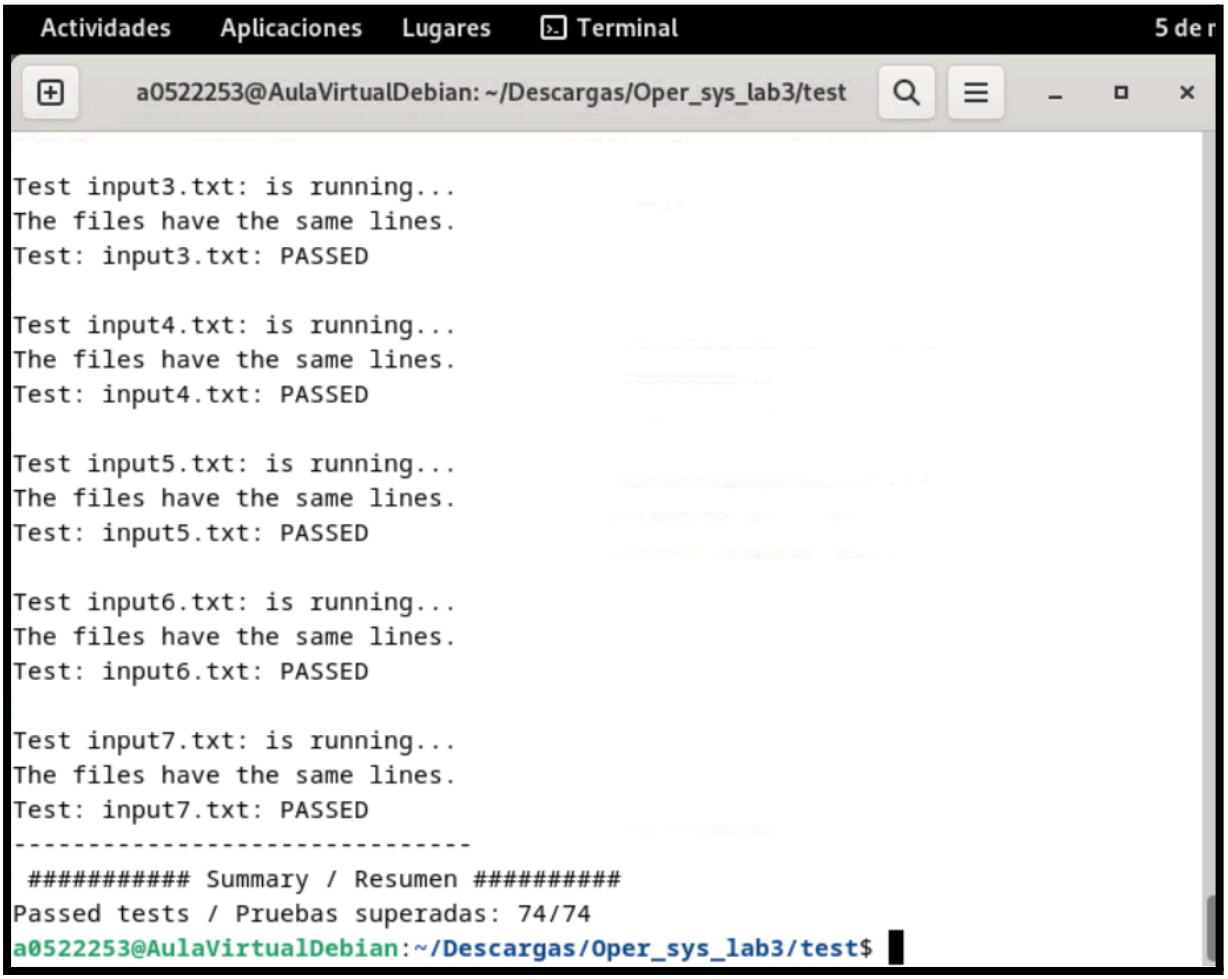
m. queue_full()

Checks if the queue count is equal to the queue size

# Test cases:

As a starting point, the programs pass the tests provided by the teachers in tester.sh. The environment of execution is a virtual machine of Ubuntu x64 in VMWare Workstation, in the Bash`s terminal. The program was tested in the Picasso LabINF to check the correctness of the code, the repository where the code resided was cloned and the first

iteration of the tester.sh used. However, due to the slowness of it and the geolocational constraints, the code was decided to be tested on another machine for the following tests.




| **'*factory*' Tests** | | | | |
|---|---|---|---|---|
| Description | Planned input | Expected output. | Terminal input/output | Status |
| More than 1 arguments | ./factory test1.txt | [ERROR][factory_manager] Invalid file. | $ ./factory t/output.txt name.sh<br>[ERROR][factory_manager] Invalid file. | ✅ |
| Check if your program correctly detects too many processes in the input file. | A file like input_invalid.txt containing: 2 1 2 3 2 2 3 3 2 3(3 processes, but max allowed is 2) | [ERROR][factory_manager] Invalid file. | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |
| Ensure the producer correctly blocks when the buffer is full. | 1 2 2 5 (buffer size 2, but 5 products to produce) | The producer should block after inserting 2 items until the consumer removes one. | $ ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id 1 has been created.<br>[OK][process_manager] Process_manager with id 1 waiting to produce 5 elements.<br>[OK][process_manager] Belt with id 1 has been created with a maximum of 2 elements.<br>[OK][queue] Introduced element with id 0 in belt 1.<br>[OK][queue] Introduced element with id 1 in belt 1.<br>[OK][queue] Obtained element with id 0 in belt 1. | ✅ |

| | | | | |
|---|---|---|---|---|
| | | | [OK][queue] Obtained element with id 1 in belt 1.<br>[OK][queue] Introduced element with id 2 in belt 1.<br>[OK][queue] Introduced element with id 3 in belt 1.<br>[OK][queue] Obtained element with id 2 in belt 1.<br>[OK][queue] Obtained element with id 3 in belt 1.<br>[OK][queue] Introduced element with id 4 in belt 1.<br>[OK][queue] Obtained element with id 4 in belt 1.<br>[OK][process_manager] Process_manager with id 1 has produced 5 elements.<br>[OK][factory_manager] Process_manager with id 1 has finished.<br>[OK][factory_manager] Finishing. | |
| Test the smallest valid case (1 process, buffer size 1, 1 product).<br>Check if the consumer waits when there's nothing in the buffer yet.<br>Make sure elements are produced and consumed in the correct order (num_edition).<br>Multiple process_manager Threads | 1 3 5 3 | The consumer waits for the producer to add products. Order of queue_put and queue_get messages confirms synchronization. | $ ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id 3 has been created.<br>[OK][process_manager] Process_manager with id 3 waiting to produce 3 elements.<br>[OK][process_manager] Belt with id 3 has been created with a maximum of 5 elements.<br>[OK][queue] Introduced element with id 0 in belt 3.<br>[OK][queue] Introduced element with id 1 in belt 3.<br>[OK][queue] Introduced element with id 2 in belt 3.<br>[OK][queue] Obtained element with id 0 in belt 3.<br>[OK][queue] Obtained element with id 1 in belt 3.<br>[OK][queue] Obtained element with id 2 in belt 3.<br>[OK][process_manager] Process_manager with id 3 has produced 3 elements.<br>[OK][factory_manager] Process_manager with id 3 has finished.<br>[OK][factory_manager] Finishing. | ✅ |
| Try zero process | 0 1 2 3 | [ERROR][factory_manager] Invalid file. | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |
| Tests invalid belt (circular buffer) size. | 1 2 0 2 | [ERROR][queue] There was an error while using queue with id: 2. | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |
| Stress test — large number of products on a moderate buffer size.Ensure: No memory leaks, No thread crashes No race conditions | 1 100 10 1000 | ExpectedOutput/input6.txt file | The test from the tester.sh is passed | ✅ |
| Multiple Producers with Same ID (invalid) | 2 1 5 2 1 5 2 | Invalid file | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |
| No Products to Produce. Producer should not produce anything. Consumers should terminate gracefully. | 1 2 2 0 | Invalid file | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |

| | | | | |
|---|---|---|---|---|
| Max Buffer, Min Production to test memory allocation/deallocation for oversized buffers | 1 5 1000 1 | Correct process | $ ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id 5 has been created.<br>[OK][process_manager] Process_manager with id 5 waiting to produce 1 elements.<br>[OK][process_manager] Belt with id 5 has been created with a maximum of 1000 elements.<br>[OK][queue] Introduced element with id 0 in belt 5.<br>[OK][queue] Obtained element with id 0 in belt 5.<br>[OK][process_manager] Process_manager with id 5 has produced 1 elements.<br>[OK][factory_manager] Process_manager with id 5 has finished.<br>[OK][factory_manager] Finishing. | ✅ |
| Input with Negative Values | 1 -1 5 2 | Correct process | $ ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id -1 has been created.<br>[OK][process_manager] Process_manager with id -1 waiting to produce 2 elements.<br>[OK][process_manager] Belt with id -1 has been created with a maximum of 5 elements.<br>(…).<br>[OK][process_manager] Process_manager with id -1 has produced 2 elements.<br>[OK][factory_manager] Process_manager with id -1 has finished.<br>[OK][factory_manager] Finishing. | ✅ |
| | 1 1 -5 2 | Arguments invalid | $ ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id 1 has been created.<br>[ERROR][process_manager] Arguments not valid. | ✅ |
| | 1 1 5 -2 | Argument invalid | ./factory t/output.txt<br>[OK][factory_manager] Process_manager with id 1 has been created.<br>[ERROR][process_manager] Arguments not valid. | ✅ |
| force input buffer reallocation | file with a size higher than 128 bytes | Valid run | (...)<br>[OK][factory_manager] Process_manager with id 49 has finished.<br>[OK][factory_manager] Finishing. | ✅ |
| Force the creation of 50 processes to check reallocations and large amount of processes | file with a creation of 50 processes | | | |
| Input characters letters | 1 a b c | Invalid file | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |
| Input character special symbols | 2 ; : ) | Invalid file | $ ./factory t/output.txt<br>[ERROR][factory_manager] Invalid file. | ✅ |

# Conclusions:

In conclusion, the development of this practice proceeded without any significant difficulties. All tasks were completed successfully, and minor questions were quickly resolved in the dedicated classes. Overall, the practice was valuable for reinforcing the theoretical concepts learned in class and the auxiliary materials like Professor Carretero's book applying them in a practical setting. Additionally, it contributed to strengthening our teamwork and problem-solving abilities, which are crucial in real-world projects