OPERATING SYSTEMS LABORATORY 3. MULTI-THREAD PROGRAMMING. CONTROL OF FABRICATION PROCESSES.

Mazin Bersy	100558650	100558650@alumnos.uc3m.es
Malek Mahmoud	100558649	100558649@alumnos.uc3m.es
Laila Sayed	100558440	100558440@alumnos.uc3m.es

Table of Contents

Code Discussion

The code for this project is divided into 3 files: queue.c, factory_manager.c, and process_manager.c.

Queue.c file:

It is the basic file for the queue structure and the main one that both factory_manager.c and process_manager.c depend on for process execution.

This C code consists of four main functions: queue.init, queue.put, queue_get, and queue_destroy.

Queue.init:

The main job is to start a new queue with a specific size. It starts with allocating memory for the queue and sends a “failure” message if the allocation didn’t work.Then, it allocates a space for the array elements of the queue and returns Null if the allocation fails.

After this function the code initializes two pointers: head and tail, elements count and maximum size of the belt and returns a pointer to the initialized queue.

Queue.put:

This function adds an element to the queue in a thread and prevents race conditions by preventing access to the queue. The function checks the space in the queue, if it’s full it blocks the queue until any space is emptied. The elements get inserted at the tail of the queue, and after the insertion the queue is incremented, and the count is updated. If insertion is prevented for any case, the function signals are waiting for consumers. Finally, it releases the mutex and returns successful operation.

Queue_get:

This function retrieves elements from the queue. It starts by checking if the queue is empty or not, if it is empty, it locks it and waits. In case the queue is not empty, the function allocates a memory for the retrieved element from the head of the queue. After that it prints a debug message, the count gets decremented, and pointer moves to next element at the header. Like queue.put, this function signals the waiting producers and releases the mutex and returns successful operation.

Queue_destory:

This function frees all the resources used by the queue and destroys synchronization primitives. It also clears everything in the memory used by the queue and all the array elements of that queue.

Process_manager file:

This C file contains three main functions and the struct: Producer, Consumer and process_manager. The struct defines id.belt, the belt size, the number of items to be produced,and pointer to the queue.

Producer“thread function”:

This function produces some elements and adds them to the queue. It starts by defining a pointer to “process_manager_args*”. Then it deals with items one by one, for each item it initializes element, assigns the current index value to “num_edition”, assigns last flag to 1 “for final item”, and then puts the element in the queue. It also terminated the thread using “pthread_exit (NULL)”.

Consumer“thread function”:

Since we have a function that adds elements to the queue, this function retrieves elements from the queue. It starts by converting the argument type to the specified one by “ process_manager_args* args = (process_manager_args*) arg”. Then it uses a while loop to retrieve elements and if one of the queues returns NULL, it skips the iteration. When the function reaches the last items, it frees the memory and ends the loop. It also frees the memory for all the elements and exits using” free(elem)” and “ pthread_exit(NULL);“.

Process_manager:

This function creates a process using queues, producers and consumers. It outputs initialization message and initialize queue “prints an errors message if initialization failed”. It also declares threads: Producer and consumer prepares the arguments needed for that thread” prints a failure message if thread is not created successfully. IT also has a C code part to manage the waiting for two threads to finish via “ pthread_join(producer_thread, NULL);

pthread_join(consumer_thread, NULL)”. Print a successful message if the log is completed.

Factory_manager.c

This file contains four main functions.

process_manager:

This function Declares an external function (defined in process_manager.c) that models a producer-consumer queue system. It defines a buffer size for reading the input file. It also Stores the information provided to each “process_manager_thread”. And have a pointer “start_all” points to a semaphore used for synchronizing the start of execution.

process_manager_thread:

This function is used to launch process_manager() in a thread. It checks if the semaphore has been posted before it starts. Then it executes the main functionality for a single belt through this line” int result = process_manager(args->id, args->belt_size, args->items);”. It prints a successful message if the belt works correctly and a failure message if not. There is a built_info struct which is used to store all the belt’s information: id, built size and items.

parse_input_file:

This function reads the input file that contains the belt data. It starts by reading the input file rows into the buffer where the first value is the number of belts to be managed. Then it allocated memory for the belt array and Processes sets of three values: id, belt size, and item count for each belt.

Main Function:

It takes a file name “only one” as an input, opens the file, takes the data from it and loads belts into memory. It also initializes a semaphore for thread startup and allocated the thread array in memory. Then for each Records a creation message, initializes the argument structure, and creates a thread running “process_manager_thread”. Finaly, it Posts the semaphore once for each thread, allowing them all to start simultaneously and wait for all of them then when the operation is done it frees all the resources and exists.

Test Cases

For the test cases we have done all types of possible tests to make sure the code is working successfully with all cases.

First, we compiled all the files using this command:

```
aulavirtual@Ubuntu22v3006:~/Documentos$ gcc -o factory_manager factory_manager.c process_manager.c queue.c -pthread
```

Test case 1:

First test case “1 belt (1 process manager):

We used “echo -e” to create test cases files from the terminal

```
aulavirtual@Ubuntu22v3006:~/Documentos$ echo -e "1\n0 3 5" > test1.txt
```

Result:

```
aulavirtual@Ubuntu22v3006:~/Documentos$ ./factory_manager test1.txt
[OK][factory_manager] Process_manager with id 0 has been created.
[OK][process_manager] Process_manager with id 0 waiting to produce 5 elements.
[OK][process_manager] Belt with id 0 has been created with a maximum of 3 elemen
ts.
[OK][queue] Introduced element with id 0 in belt 0.
[OK][queue] Introduced element with id 1 in belt 0.
[OK][queue] Introduced element with id 2 in belt 0.
[OK][queue] Obtained element with id 0 in belt 0.
[OK][queue] Obtained element with id 1 in belt 0.
[OK][queue] Obtained element with id 2 in belt 0.
[OK][queue] Introduced element with id 3 in belt 0.
[OK][queue] Introduced element with id 4 in belt 0.
[OK][queue] Obtained element with id 3 in belt 0.
[OK][queue] Obtained element with id 4 in belt 0.
[OK][process_manager] Process_manager with id 0 has produced 5 elements.
[OK][factory_manager] Process_manager with id 0 has finished.
[OK][factory_manager] Finishing.
aulavirtual@Ubuntu22v3006:~/Documentos$
```

It is clear that the normal test for one producer-consumer works correctly.

Test case 2:

```
aulavirtual@Ubuntu22v3025:~/Documentos$ echo -e "2\n0 3 5\n1 2 6" > input2.txt
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager input2.txt
```

```
[OK][factory_manager] Process_manager with id 1 has been created.
[OK][process_manager] Process_manager with id 0 waiting to produce 5 elements.
[OK][process_manager] Process_manager with id 1 waiting to produce 6 elements.
[OK][process_manager] Belt with id 1 has been created with a maximum of 2 elements.
[OK][process_manager] Belt with id 0 has been created with a maximum of 3 elements.
[OK][queue] Introduced element with id 0 in belt 1.
[OK][queue] Introduced element with id 1 in belt 1.
[OK][queue] Introduced element with id 0 in belt 0.
[OK][queue] Introduced element with id 1 in belt 0.
[OK][queue] Introduced element with id 2 in belt 0.
[OK][queue] Obtained element with id 0 in belt 1.
[OK][queue] Obtained element with id 1 in belt 1.
[OK][queue] Introduced element with id 2 in belt 1.
[OK][queue] Introduced element with id 3 in belt 1.
[OK][queue] Obtained element with id 0 in belt 0.
[OK][queue] Obtained element with id 1 in belt 0.
[OK][queue] Obtained element with id 2 in belt 0.
[OK][queue] Introduced element with id 3 in belt 0.
[OK][queue] Introduced element with id 4 in belt 0.
[OK][queue] Obtained element with id 2 in belt 1.
[OK][queue] Obtained element with id 3 in belt 1.
[OK][queue] Introduced element with id 4 in belt 1.
[OK][queue] Introduced element with id 5 in belt 1.
[OK][queue] Obtained element with id 4 in belt 1.
[OK][queue] Obtained element with id 5 in belt 1.
[OK][process_manager] Process_manager with id 1 has produced 6 elements.
[OK][factory_manager] Process_manager with id 1 has finished.
```

```
[OK][queue] Obtained element with id 3 in belt 0.
[OK][queue] Obtained element with id 4 in belt 0.
[OK][process_manager] Process_manager with id 0 has produced 5 elements.
[OK][factory_manager] Process_manager with id 0 has finished.
[OK][factory_manager] Finishing.
```

This test was for ensuring the functionality of multiple threads working concurrently and it successfully did.

Test Case 3:

echo -e “4\n1 2 10\n2 3 6\n3 1 4\n4 5 5” > input3.txt

```
[OK][queue] Obtained element with id 1 in belt 4.
[OK][queue] Obtained element with id 2 in belt 4.
[OK][queue] Obtained element with id 3 in belt 4.
[OK][queue] Obtained element with id 4 in belt 4.
[OK][queue] Obtained element with id 0 in belt 3.
[OK][queue] Obtained element with id 0 in belt 1.
[OK][process_manager] Process_manager with id 4 has produced 5 elements.
[OK][factory_manager] Process_manager with id 4 has finished.
[OK][queue] Introduced element with id 1 in belt 3.
[OK][queue] Obtained element with id 1 in belt 3.
[OK][queue] Obtained element with id 0 in belt 2.
[OK][queue] Introduced element with id 2 in belt 3.
[OK][queue] Obtained element with id 2 in belt 3.
[OK][queue] Obtained element with id 1 in belt 2.
[OK][queue] Obtained element with id 2 in belt 2.
[OK][queue] Introduced element with id 3 in belt 3.
[OK][queue] Introduced element with id 2 in belt 1.
[OK][queue] Obtained element with id 3 in belt 3.
[OK][queue] Introduced element with id 3 in belt 2.
[OK][process_manager] Process_manager with id 3 has produced 4 elements.
[OK][factory_manager] Process_manager with id 3 has finished.
[OK][queue] Introduced element with id 4 in belt 2.
[OK][queue] Introduced element with id 5 in belt 2.
[OK][queue] Obtained element with id 3 in belt 2.
[OK][queue] Obtained element with id 4 in belt 2.
[OK][queue] Obtained element with id 5 in belt 2.
[OK][queue] Obtained element with id 1 in belt 1.
```

```
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager input3.txt
[OK][factory_manager] Process_manager with id 1 has been created.
[OK][factory_manager] Process_manager with id 2 has been created.
[OK][factory_manager] Process_manager with id 3 has been created.
[OK][factory_manager] Process_manager with id 4 has been created.
[OK][process_manager] Process_manager with id 4 waiting to produce 5 elements.
[OK][process_manager] Belt with id 4 has been created with a maximum of 5 elements.
[OK][process_manager] Process_manager with id 3 waiting to produce 4 elements.
[OK][queue] Introduced element with id 0 in belt 4.
[OK][queue] Introduced element with id 1 in belt 4.
[OK][queue] Introduced element with id 2 in belt 4.
[OK][process_manager] Process_manager with id 1 waiting to produce 10 elements.
[OK][queue] Introduced element with id 3 in belt 4.
[OK][queue] Introduced element with id 4 in belt 4.
[OK][process_manager] Belt with id 1 has been created with a maximum of 2 elements.
[OK][process_manager] Belt with id 3 has been created with a maximum of 1 elements.
[OK][process_manager] Process_manager with id 2 waiting to produce 6 elements.
[OK][process_manager] Belt with id 2 has been created with a maximum of 3 elements.
[OK][queue] Introduced element with id 0 in belt 1.
[OK][queue] Introduced element with id 1 in belt 1.
[OK][queue] Introduced element with id 0 in belt 3.
[OK][queue] Introduced element with id 0 in belt 2.
[OK][queue] Introduced element with id 1 in belt 2.
[OK][queue] Introduced element with id 2 in belt 2.
[OK][queue] Obtained element with id 0 in belt 4.
```

```
[OK][queue] Introduced element with id 4 in belt 2.
[OK][queue] Introduced element with id 5 in belt 2.
[OK][queue] Obtained element with id 3 in belt 2.
[OK][queue] Obtained element with id 4 in belt 2.
[OK][queue] Obtained element with id 5 in belt 2.
[OK][queue] Obtained element with id 1 in belt 1.
[OK][queue] Introduced element with id 3 in belt 1.
[OK][queue] Obtained element with id 2 in belt 1.
[OK][process_manager] Process_manager with id 2 has produced 6 elements.
[OK][factory_manager] Process_manager with id 2 has finished.
[OK][queue] Introduced element with id 4 in belt 1.
[OK][queue] Obtained element with id 3 in belt 1.
[OK][queue] Introduced element with id 5 in belt 1.
[OK][queue] Obtained element with id 4 in belt 1.
[OK][queue] Introduced element with id 6 in belt 1.
[OK][queue] Obtained element with id 5 in belt 1.
[OK][queue] Introduced element with id 7 in belt 1.
[OK][queue] Obtained element with id 6 in belt 1.
[OK][queue] Introduced element with id 8 in belt 1.
[OK][queue] Obtained element with id 7 in belt 1.
[OK][queue] Introduced element with id 9 in belt 1.
[OK][queue] Obtained element with id 8 in belt 1.
[OK][queue] Obtained element with id 9 in belt 1.
[OK][process_manager] Process_manager with id 1 has produced 10 elements.
[OK][factory_manager] Process_manager with id 1 has finished.
[OK][factory_manager] Finishing.
aulavirtual@Ubuntu22v3025:~/Documentos$
```

This is a stress test to make sure the code works for mixed different size belts under concurrency.

Test case 4:

Echo -e “1\n9 2 10” > input4.txt

```
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager input4.txt
[OK][factory_manager] Process_manager with id 9 has been created.
[OK][process_manager] Process_manager with id 9 waiting to produce 10 elements.
[OK][process_manager] Belt with id 9 has been created with a maximum of 2 elements.
[OK][queue] Introduced element with id 0 in belt 9.
```

```
[OK][factory_manager] Process_manager with id 9 has been created.
[OK][process_manager] Process_manager with id 9 waiting to produce 10 elements.
[OK][process_manager] Belt with id 9 has been created with a maximum of 2 elements.
[OK][queue] Introduced element with id 0 in belt 9.
[OK][queue] Introduced element with id 1 in belt 9.
[OK][queue] Obtained element with id 0 in belt 9.
[OK][queue] Obtained element with id 1 in belt 9.
[OK][queue] Introduced element with id 2 in belt 9.
[OK][queue] Introduced element with id 3 in belt 9.
[OK][queue] Obtained element with id 2 in belt 9.
[OK][queue] Obtained element with id 3 in belt 9.
[OK][queue] Introduced element with id 4 in belt 9.
[OK][queue] Introduced element with id 5 in belt 9.
[OK][queue] Obtained element with id 4 in belt 9.
[OK][queue] Obtained element with id 5 in belt 9.
[OK][queue] Introduced element with id 6 in belt 9.
[OK][queue] Introduced element with id 7 in belt 9.
[OK][queue] Obtained element with id 6 in belt 9.
[OK][queue] Obtained element with id 7 in belt 9.
[OK][queue] Introduced element with id 8 in belt 9.
[OK][queue] Introduced element with id 9 in belt 9.
[OK][queue] Obtained element with id 8 in belt 9.
[OK][queue] Obtained element with id 9 in belt 9.
[OK][process_manager] Process_manager with id 9 has produced 10 elements.
[OK][factory_manager] Process_manager with id 9 has finished.
[OK][factory_manager] Finishing.
```

This test was to make sure that producer blocks successfully if the queue is full.

Test case 5:

```
aulavirtual@Ubuntu22v3025:~/Documentos$ echo -e "1\n0 100 100000" > input5.txt
```

```
[OK][queue] Obtained element with id 99986 in belt 0.
[OK][queue] Obtained element with id 99987 in belt 0.
[OK][queue] Obtained element with id 99988 in belt 0.
[OK][queue] Obtained element with id 99989 in belt 0.
[OK][queue] Obtained element with id 99990 in belt 0.
[OK][queue] Introduced element with id 99991 in belt 0.
[OK][queue] Introduced element with id 99992 in belt 0.
[OK][queue] Introduced element with id 99993 in belt 0.
[OK][queue] Introduced element with id 99994 in belt 0.
[OK][queue] Introduced element with id 99995 in belt 0.
[OK][queue] Introduced element with id 99996 in belt 0.
[OK][queue] Introduced element with id 99997 in belt 0.
[OK][queue] Introduced element with id 99998 in belt 0.
[OK][queue] Introduced element with id 99999 in belt 0.
[OK][queue] Obtained element with id 99991 in belt 0.
[OK][queue] Obtained element with id 99992 in belt 0.
[OK][queue] Obtained element with id 99993 in belt 0.
[OK][queue] Obtained element with id 99994 in belt 0.
[OK][queue] Obtained element with id 99995 in belt 0.
[OK][queue] Obtained element with id 99996 in belt 0.
[OK][queue] Obtained element with id 99997 in belt 0.
[OK][queue] Obtained element with id 99998 in belt 0.
[OK][queue] Obtained element with id 99999 in belt 0.
[OK][process_manager] Process_manager with id 0 has produced 100000 elements
[OK][factory_manager] Process_manager with id 0 has finished.
[OK][factory_manager] Finishing.
```

This test was to make sure built works for large values.
Note: The terminal was counting for many numbers, so I provided a screenshot for the most important last part that shows that the count is done successfully till the end.

Test case 6:

```
aulavirtual@Ubuntu22v3025:~/Documentos$ echo -e "1\n0 1 50" > input6.txt
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager input6.txt
[OK][factory_manager] Process_manager with id 0 has been created.
[OK][process_manager] Process_manager with id 0 waiting to produce 50 elements.
[OK][process_manager] Belt with id 0 has been created with a maximum of 1 elements.
[OK][queue] Introduced element with id 0 in belt 0.
[OK][queue] Obtained element with id 0 in belt 0.
[OK][queue] Introduced element with id 1 in belt 0.
[OK][queue] Obtained element with id 1 in belt 0.
[OK][queue] Introduced element with id 2 in belt 0.
[OK][queue] Obtained element with id 2 in belt 0.
[OK][queue] Introduced element with id 3 in belt 0.
[OK][queue] Obtained element with id 3 in belt 0.
[OK][queue] Introduced element with id 4 in belt 0.
[OK][queue] Obtained element with id 4 in belt 0.
[OK][queue] Introduced element with id 5 in belt 0.
[OK][queue] Obtained element with id 5 in belt 0.
[OK][queue] Introduced element with id 6 in belt 0.
[OK][queue] Obtained element with id 6 in belt 0.
[OK][queue] Introduced element with id 7 in belt 0.
[OK][queue] Obtained element with id 7 in belt 0.
[OK][queue] Introduced element with id 8 in belt 0.
[OK][queue] Obtained element with id 8 in belt 0.
[OK][queue] Introduced element with id 9 in belt 0.
[OK][queue] Obtained element with id 9 in belt 0.
```

```
[OK][queue] Obtained element with id 38 in belt 0.
[OK][queue] Introduced element with id 39 in belt 0.
[OK][queue] Obtained element with id 39 in belt 0.
[OK][queue] Introduced element with id 40 in belt 0.
[OK][queue] Obtained element with id 40 in belt 0.
[OK][queue] Introduced element with id 41 in belt 0.
[OK][queue] Obtained element with id 41 in belt 0.
[OK][queue] Introduced element with id 42 in belt 0.
[OK][queue] Obtained element with id 42 in belt 0.
[OK][queue] Introduced element with id 43 in belt 0.
[OK][queue] Obtained element with id 43 in belt 0.
[OK][queue] Introduced element with id 44 in belt 0.
[OK][queue] Obtained element with id 44 in belt 0.
[OK][queue] Introduced element with id 45 in belt 0.
[OK][queue] Obtained element with id 45 in belt 0.
[OK][queue] Introduced element with id 46 in belt 0.
[OK][queue] Obtained element with id 46 in belt 0.
[OK][queue] Introduced element with id 47 in belt 0.
[OK][queue] Obtained element with id 47 in belt 0.
[OK][queue] Introduced element with id 48 in belt 0.
[OK][queue] Obtained element with id 48 in belt 0.
[OK][queue] Introduced element with id 49 in belt 0.
[OK][queue] Obtained element with id 49 in belt 0.
[OK][process_manager] Process_manager with id 0 has produced 50 elements.
[OK][factory_manager] Process_manager with id 0 has finished.
[OK][factory_manager] Finishing.
```

This test cases check for the successful alternating between producer and consumer.

Test case 7:

```
aulavirtual@Ubuntu22v3025:~/Documentos$ echo -e "1\n0 0 10" > input7.txt
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager input7.txt
[ERROR][factory_manager] Invalid file.
```

This test case validates the correct input checking for belt configuration.

Test case 8:

```
aulavirtual@Ubuntu22v3025:~/Documentos$ echo -e "hello world" > input8.txt
aulavirtual@Ubuntu22v3025:~/Documentos$ ./factory_manager > input8.txt
```

This test case validates that the program doesn't work for corrupted or invalid input files.

Problems encountered

Problem 1:
Ensuring correct synchronization between threads was challenging, particularly with respect to the placement of print statements. Without proper coordination, log messages appeared out of order, making it difficult to verify correct execution and debug the program.

> Solution:
> To resolve this, mutexes and semaphores were utilized to regulate execution timing and order. Print statements were inserted only once locks secured critical operations, ensuring the logs matched the real event sequence.

Problem 2:

Initially, we attempted to use a shared global queue for communication between threads. However, this approach led to synchronization issues, including race conditions and data inconsistencies, due to concurrent access from multiple threads.

> Solution:
> To resolve this, we refactored the design so that each thread manages its own independent queue. This eliminated contention over a shared resource and simplified synchronization, resulting in more stable and predictable thread behavior.

Problem 3:
Properly initializing mutexes, semaphores, and queues before access was a subtle yet crucial requirement, as failing to do so could lead to undefined behavior.

Overall, the assignment served as an insightful exploration of real-world multithreading complexities, requiring expertise in synchronization, thread-safe design, and shared-resource management. Initial hurdles—such as inconsistent test outputs and race conditions—were resolved through systematic debugging, resulting in a reliable final implementation.

Conclusion

This laboratory project provided valuable hands-on experience with multi-threaded programming and the control of fabrication processes through a simulated producer-consumer model. By implementing independent queues per belt and managing synchronization using mutexes and semaphores, the project successfully addressed common concurrency challenges such as race conditions, deadlocks, and thread coordination. Rigorous testing across a variety of scenarios, including stress tests and input validation, confirmed the robustness and correctness of the implementation. Ultimately, the assignment deepened our understanding of thread-safe design principles and demonstrated the critical importance of proper synchronization in concurrent systems.