University Carlos III of Madrid

BSc in Applied Mathematics and Computing

Lab 3: Multi-thread programming. Control of fabrication processes.

Member 1: Sabela Rubert Docampo
100523102
100523102@alumnos.uc3m.es

Member 2: Roberto Hogas Goras
100523139
100523139@alumnos.uc3m.es

Member 3: Alejandro Tarbay Guadalupe
100523065
100523065@alumnos.uc3m.es

Academic Year 2024/25 Group 121

With love, made in LaTeX for Catherine, our Operating Systems' lab teacher.

Click here to see our Overleaf code.

Contents

1 Description of the codes

1.1 `factory_manager.c`

1.1.1 Libraries used

- **#include <stdio.h>** – Provides input/output functionality, like printing messages using `printf`.
- **#include <stdlib.h>** – Offers functions for memory management, such as `malloc`, `realloc`, and `free`, as well as general utilities.
- **#include <unistd.h>** – Grants access to POSIX API functions like `read`, `close`, and `write`, useful for working with files and system calls.
- **#include <fcntl.h>** – Used for file control operations, including the `open` system call used to open files.
- **#include <stddef.h>** – Defines useful types and macros such as `size_t` and `NULL`.
- **#include <pthread.h>** – Enables multithreading support with functions like `pthread_create` and `pthread_join`.
- **#include <semaphore.h>** – Provides semaphore functions like `sem_init`, `sem_post`, and `sem_wait`, used to synchronize threads.
- **#include <sys/stat.h>** – Supplies constants and functions for file status operations, commonly used alongside `open`.
- **#include "process˙manager.h"** – Includes the custom header that declares the `process_manager` function and the `process_data_t` structure.
- **#include "factory˙manager.h"** – Includes definitions related to the factory manager logic, shared across modules.

1.1.2 Explanation of code

In this first code we were asked to program a C program . The program must recognize input files that follow this structure:

$$\text{<Max number belts>}\,[\text{<belt ID>}\,\text{<belt size>}\,\text{<No. elements>}]^{+}$$

Where:

- **Max number belts:** Represents the maximum number of `process_managers` that can be created. If, after reading the file, more processes than the declared maximum are detected, the file is considered invalid, and the program must terminate by returning `-1`.
- **Belt ID:** A unique identifier assigned to each `process_manager` to track its products in the traces.

- **Belt size:** The maximum number of elements that a transport belt can contain. It represents the size of the circular buffer between producer and consumer.
- **No. of elements:** The number of elements to be generated by the assigned belt.

To start programming, we downloaded the folder called **Support code** in Aula Global. From there, we obtained the first lines of code to know how to start. We begin with a structure where we stored the information of each process. This information is made up of an ID, a size, and the number of items it needs to generate. We also make sure that no error comes up when introducing the number of arguments or opening the file. If such an error occurs, a message will be shown.

The next part of code reads line by line, character by character, saving it in a **buffer** (which grows if needed). The program is coded to replace a newline character with a null terminator. During this while loop, there's also error handling, and if any error reading the line or such occurs, a message pops up as before.

Then, we get the number of processes from the first line and create space in memory to store their data. Afterward we start reading the information in groups of three numbers (the previous information stored in the beginning of the code: ID, a size, and the number of items it needs to generate). All this for each process. Then we make sure we handle the corresponding errors and continue.

Before continuing to the next section of the code, we declare a few variables to store the maximum number of processes and a counter to keep track of the processes that have been already read. We then try to get the maximum number of **belts**. If the line is empty or the number of belts is not bigger than 0, we return an error.

To continue, we go line by line reading the sets of three numbers and store them in an array called `belts`. We use `malloc` to dynamically allocate memory based on the maximum number of processes' value. We go through a process of error checking, to make sure that we don't have more belts than the maximum number of belts, or that we do read at least one belt, for example. If so, the appropriate message appears. Once we are done reading, we free the memory used to store the line. Afterward, we assign each element of the array a thread and a semaphore. If there's any error while doing this, we print an error, free the memory, and close the file.

We then initialize the **semaphores** to 0, which means each process manager will be blocked at first. After that, we create a thread for each process manager with `pthread_create`, passing in the data. Again, if there's any problem while doing this, we print it.

Once the thread is created, we signal the semaphore with `sem_post` to let the process manager start. Finally, if everything goes well, a success message is printed for each process manager.

Now, we wait for all the threads to finish their execution. We use `sem_wait` (in `process_manager.c`) and wait for the signal from each semaphore, indicating that the process manager is ready to execute. If this fails, we handle it as always.

Then, we use `pthread_join` to wait for each thread to finish. If any of the threads cannot join, we again print an error, free the memory, and close the file.

After all threads have finished, we destroy the semaphores with `sem_destroy`. If any semaphore cannot be destroyed, we handle the error in the same way as before.

Finally, we free the dynamically allocated memory for the threads, processes, and semaphores, and print a message indicating the successful completion of the process. The function then returns 0 to indicate everything has finished without errors.

1.2 `process_manager.c`

1.2.1 Libraries used

- **#include <stdio.h>** – Enables standard input and output operations, primarily used here for `printf` to print messages and errors.
- **#include <stdlib.h>** – Provides general-purpose functions including memory allocation with `malloc` and deallocation with `free`.
- **#include <unistd.h>** – Offers access to POSIX operating system API, although in this file it's likely included for consistency or possible future use (e.g., `read`, `write`).
- **#include <fcntl.h>** – Generally used for file control operations, not directly used here but may be included for consistency with other files in the project.
- **#include <stddef.h>** – Defines standard types and macros like `NULL` and `size_t`, which are used in memory handling and function parameters.
- **#include <pthread.h>** – Provides support for multithreading using POSIX threads, including functions like `pthread_create` and `pthread_join`.
- **#include <semaphore.h>** – Adds support for POSIX semaphores, used for synchronizing access between threads with functions like `sem_wait` and `sem_post`.
- **#include "queue.h"** – Includes the custom queue implementation header, which defines functions like `queue_init`, `queue_put`, `queue_get`, and `queue_destroy`.
- **#include "process˙manager.h"** – Declares structures and function prototypes specific to the process manager logic.
- **#include "factory˙manager.h"** – Gives access to definitions from the factory manager module, such as shared types or synchronization variables.

1.2.2 Explanation of code

The goal of `process_manager.c` is to manage a producer-consumer system using threads. Each process manager controls a **conveyor belt**, where the producer adds items to the belt and the consumer removes them. All of that while making sure no error occurs.

The `producer` function is in charge of creating the items that will be added to the belt. First, it takes the input argument and makes sure it's not null. If it is, it prints an error and exits the thread. Then, it gets the belt ID and the number of items it needs to produce from the data. After that, it enters a loop that runs once for each item it needs to generate. Inside

the loop, it creates a new element, fills in its fields (like the edition number and the belt ID), and checks if it's the last item and if it is, it sets the last field to 1. Then it tries to add the element to the queue using `queue_put`. If that fails, it prints an error, frees the memory, and exits the thread. If everything goes fine, it just keeps going until all items are produced. Once it's done, it exits the thread normally.

The `consumer` function is the one that takes the items off the belt and processes them. Just like in producer, it first checks that the input argument isn't null. If it is, it prints an error and exits the thread. Then, it gets how many items it's supposed to consume from the data. It keeps a counter called `consumed`, and while that number is smaller than the total items it needs to get, it keeps going. Inside the loop, it tries to take an item from the queue using `queue_get`. If something goes wrong and it gets null, it means there was a problem with the queue, so it prints an error and exits the thread. If it gets an item successfully, it increases the counter and frees the memory of the item since it's done using it. When all the items are consumed, it just exits the thread.

The `process_manager` receives some data that tells it which belt it's managing, how big the belt is (how many items it can hold), and how many items it needs to handle. First, it checks if the input data is valid. If it's not, it prints an error and exits. Then, it waits on a semaphore. This is basically a way to pause the function until it's allowed to start. When it gets the green light, it prints a message saying it's ready to produce.

Next, it initializes the queue (the belt) with the size given. If that fails, it prints an error and exits. If the belt is created with no problems, it creates two threads: the first one for the producer and the other for the consumer. If either thread fails to start, it prints it, destroys the belt and exits.

If the threads are created successfully, it waits for both of them to finish using `pthread_join`. That just means it's not going to move on until the producer and consumer are both done. If it fails, it exits. After both threads finish, it destroys the queue and posts to the semaphore. Finally, it exits the thread.

1.3 `queue.c`

1.3.1 Libraries used

- **#include <stdio.h>** – Used for standard input and output, like printing debug messages with `printf`.
- **#include <stdlib.h>** – Needed for functions like `malloc` and `free` to manage dynamic memory.
- **#include <pthread.h>** – Brings in support for working with threads, including mutexes and condition variables.
- **#include <stddef.h>** – Provides definitions for common macros and types like `NULL` and `size_t`.

- **#include "queue.h"** – This pulls in the custom header file that defines the queue structure and its related functions.

1.3.2 Explanation of code

Important note for the design of this code: We decided to create a single global queue that will be reused by each process. For us, it was easier to synchronize all the processes this way, rather than creating a different queue for each one of them. Each `process_manager` runs sequentially: to start a new queue, the one which is running must finish first.

Before we began with the functions, we decided to create a single structure that holds all the relevant information according to the queue.

In the first function, `queue_init`, we initialize the queue by allocating memory for it, according to the size given as an input. Then we set the indexes for head, tail, and count. Only then we set the mutex and the condition variables.

Then, in `queue_put`, we lock the mutex so nobody else can use the queue while we are using it. Sequentially, we wait for the queue to be not full, and when we have the chance, introduce the element into it, updating the indexes. If the queue is full after this operation, notify it and unlock the mutex.

`queue_get` follows the same logic as `queue_put`, but in reverse. The only thing we have to make sure is that we allocate the right amount of memory before getting the element.

`queue_full` and `queue_empty` are simple checks that are used in the previous functions.

Lastly, `queue_destroy` cleans the queue and frees the buffer, deallocating the memory so the next element can use it.

2 Test cases

The test cases provided by the teacher have been tried and they worked perfectly (100%). We did not include them in this subsection because we were not the ones to come up with them.

Data to enter	**test.txt**	**Description of the test**	**Expected result**	**Obtained result**
./factory	—	Incorrect number of arguments	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt test.txt	—	Incorrect number of arguments	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.

./factory test.txt	-1 5 9 6	Maximum number of process_managers must be greater than 0	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	5 -2 5 6	Belt ID of process_manager must be greater than or equal to 0	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	3 2 -4 3	Belt size of any process_manager must be greater than 0	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	3 2 8 -3	Number of elements to be generated by the belt must be greater than 0	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	2 2 8 3 9 8 6 9 5 9	The number of process_managers is greater than the maximum number of process_managers	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	2 5 a 9	An input is non-numeric	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	5 9 8(two spaces)9	There is two spaces between two inputs	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	5 6 8 9\n 8 9 9 9	There are two lines of belts	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.
./factory test.txt	5	No process_managers (there must be at least 1)	[ERROR] [factory_manager] Invalid file.	[ERROR] [factory_manager] Invalid file.

./factory test.txt	9 2 2 3 6 1 2	Multiple belts with varied sizes and loads	[OK][factory_manager] Process_manager with id 2 has been created. [OK][process_manager] Process_manager with id 2 waiting to produce 3 elements. ... [OK][queue] Introduced element with id 2 in belt 2. [OK][queue] Obtained element with id 2 in belt 2. [OK][process_manager] Process_manager with id 2 has produced 3 elements. [OK][process_manager] Process_manager with id 6 waiting to produce 2 elements. ... [OK][queue] Obtained element with id 1 in belt 6. [OK][process_manager] Process_manager with id 6 has produced 2 elements. [OK][factory_manager] Process_manager with id 6 has finished. [OK][factory_manager] Finishing.	[OK][factory_manager] Process_manager with id 2 has been created. [OK][process_manager] Process_manager with id 2 waiting to produce 3 elements. ... [OK][queue] Introduced element with id 2 in belt 2. [OK][queue] Obtained element with id 2 in belt 2. [OK][process_manager] Process_manager with id 2 has produced 3 elements. [OK][process_manager] Process_manager with id 6 waiting to produce 2 elements. ... [OK][queue] Obtained element with id 1 in belt 6. [OK][process_manager] Process_manager with id 6 has produced 2 elements. [OK][factory_manager] Process_manager with id 6 has finished. [OK][factory_manager] Finishing.

./factory test.txt	1 1 100 500	Belt with large number of elements	[OK][factory_manager] Process_manager with id 1 has been created. [OK][process_manager] Process_manager with id 1 waiting to produce 500 elements. [OK][process_manager] Belt with id 1 has been created with a maximum of 100 elements. ... [OK][queue] Introduced element with id 2 in belt 1. [OK][queue] Obtained element with id 0 in belt 1. ... [OK][queue] Obtained element with id 497 in belt 1. [OK][queue] Obtained element with id 498 in belt 1. [OK][queue] Obtained element with id 499 in belt 1. [OK][process_manager] Process_manager with id 1 has produced 500 elements. [OK][factory_manager] Process_manager with id 1 has finished. [OK][factory_manager] Finishing.	[OK][factory_manager] Process_manager with id 1 has been created. [OK][process_manager] Process_manager with id 1 waiting to produce 500 elements. [OK][process_manager] Belt with id 1 has been created with a maximum of 100 elements. ... [OK][queue] Introduced element with id 2 in belt 1. [OK][queue] Obtained element with id 0 in belt 1. ... [OK][queue] Obtained element with id 497 in belt 1. [OK][queue] Obtained element with id 498 in belt 1. [OK][queue] Obtained element with id 499 in belt 1. [OK][process_manager] Process_manager with id 1 has produced 500 elements. [OK][factory_manager] Process_manager with id 1 has finished. [OK][factory_manager] Finishing.

3 Conclusion

This last laboratory gave us a great understanding on how to synchronize threads and developed even more our knowledge of C. The queue was our biggest problem, as we did not know if we should approach it as several different ones or only the global one. After trial and error, we found much trouble while allocating the memory with the different queues, so we decided to go for the unique and global one. The time spent on this project was about 35 hours.