Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Laboratory 3. Multi-thread programming. Control of fabrication processes.

Bachelor's Degree in Computer Science & Engineering

Bachelor's Degree in Applied Mathematics & Computing

Dual Bachelor's in Computer Science & Engineering & Business Administration

Academic Year 2024/2025

Índice

1. Laboratory statement **1**
2. Description **2**
2.1. Manufacturing Process Manager 4
2.2. Manufacturing processes 4
2.2.1. Queue on a circular buffer 4
2.3. Factory Manager input file 5
2.4. Integration and implementation examples 6
3. Submission **9**
3.1. Submission Deadline 10
3.2. Submission Procedure 10
2.3 Files to be submitted 10
4. Rules **11**
5. Annexes **12**
5.1 man function 13
5.2 Semaphores 13
6. References **14**

1. Laboratory statement

This practice allows the student to become familiar with the process management services provided by POSIX.

For the management of lightweight processes (threads), the system calls *pthread_create, pthread_join,* and *pthread_exit* will be used. Whereas *mutex* and *conditional variables* will be used for their synchronization:

- **Pthread_create:** creates a new thread that executes a function given as an argument in the call.
- **Pthread_join:** performs a wait for a thread that must terminate and that is indicated as an argument of the call.
- **Pthread_exit:** terminates the execution of the calling process.

The student must design and code, in C language and on the Linux operating system, a program that acts as a manufacturing process manager including several threads in charge of managing different phases of the factory, and a thread that performs the planning of the phases.

2. Description

The objective of this practice is to code a simulation of the operation of a factory in which workers have different roles. The roles must work concurrently, allowing the ts used in the factory to have correct management and movement.

Roles are:

- **Factory manager:** This thread is destined to start the threads in charge of factory production. Its main function is to start the manufacturing process by starting the process_manager threads. The factory_manager can start as many process_manager threads as indicated in the load file.

- **Process manager:** It is the thread in charge of starting up the light processes that will work together with each conveyor belt to generate a producer-consumer system. The elements involved in this subsystem are:

    - **Producer:** It is a light process that will be in charge of producing as many elements as the process_manager tells it to and making them available to a consumer on a conveyor belt between the two of them.

    - **Consumer:** It is a light process that will pick up as many elements as its producing partner sends it on the conveyor belt.

Therefore, the *factory_manager* is the head of the factory, who communicates to his n *process_managers* how many elements each one must produce. Finally, each *process_manager* will send two workers (producer and consumer) to perform the tasks entrusted by the boss.

NOTE: The conveyor belts will be implemented using circular tails.

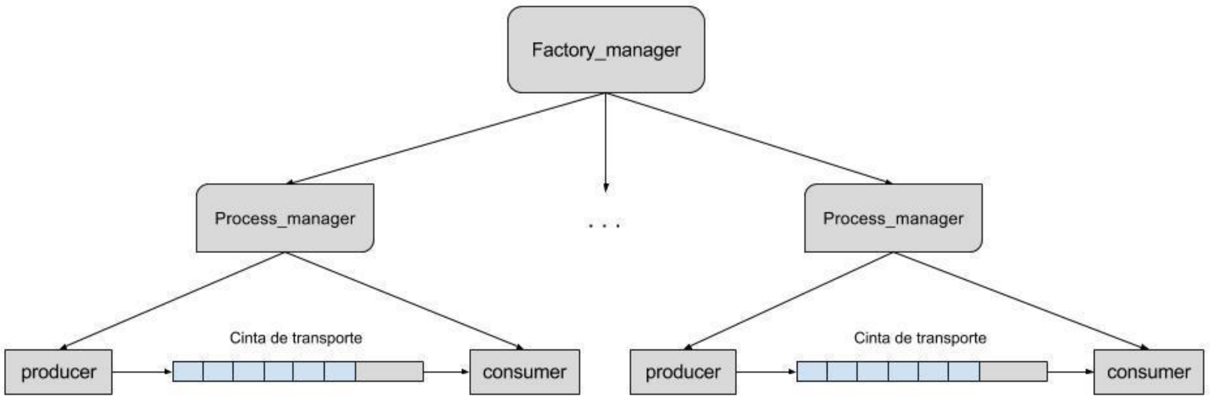

Figure 1. Example with two conveyor belts.

2.1. Manufacturing Process Manager

The manufacturing process manager corresponds to the *factory_manager* functions and must be implemented in a file named factory_manager.c. The tasks of this role are to capture the factory input parameters, which are entered using a file, and to launch the *n* process_managers indicated by the input parameters. The program will have only one argument: the path to a file, which will be the load file (described in section 2.3).

When the factory_manager creates a thread of type process_manager, it must indicate the following data: **<Id> <maximum belt size><No. of products to generate>**, where:

- **Id:** is the belt identifier (determined by the data included in the load file).
- **Maximum belt size:** this is the maximum number of elements that can be stored on the assigned belt (circular buffer size).
- **Number of products to be generated:** each producer-consumer work group is assigned a number of products to be generated, indicated by the input file and assigned to the process_manager by parameter from the factory_manager parent process.

Synchronization between the factory_manager and the process_managers is done using semaphores controlled by the *factory_manager*. The function of the semaphores that manage the processes is to control the operation of a maximum number of manufacturing processes and their order. The factory will have a maximum size, defined in the load file, and there cannot be more manufacturing processes than enter the factory (but there can be fewer). In addition, the manufacturing processes must wait at a semaphore to be woken up by the factory_manager.

2.2. Manufacturing processes

The threads of the manufacturing processes are going to be in charge of performing the tasks of launching the factory workers, and their functions are those of *process_manager*. The functionality corresponding to this point must be implemented in a file called process_manager.c. Its arguments will be those defined in section 2.1.

Each process_manager created must generate a producer-consumer system implemented with threads. The producer process will insert elements into a conveyor belt, while the consumer process will pick up elements from the belt.

This problem to be implemented is a classic example of process synchronization: when sharing a conveyor belt (circular buffer), concurrency control must be carried out when depositing objects on the belt and when extracting them.

2.2.1. Queue on a circular buffer

The communication between producers and consumers will be done using conveyor belts. These belts correspond to queues on circular buffers. A circular queue must be created for each producer-consumer. Since modifications will constantly occur on this element, concurrency control mechanisms must be implemented for lightweight processes.

The circular queue and its functions must be implemented in a file named queue.c, and must contain at least the following functions:

- **int queue_init (int num_elements):** function that creates the queue and reserves the size specified as a parameter.
- **int queue_destroy (void):** function that destroys the queue and frees all allocated resources.
- **int queue_put (struct element * ele):** function that inserts elements into the queue if space is available. If there is no space available, it must wait until the insertion can be performed.
- **struct element * queue_get (void):** function that extracts elements from the queue if there are elements in it. If the queue is empty, it must wait until an element is available.
- **int queue_empty (void):** function that queries the queue status and determines if it is empty (return 1) or not (return 0).
- **int queue_full (void):** function that queries the queue status and determines if the queue is full (return 1) or still has available items (return 0).

Implementing this queue must ensure that there are no concurrency problems among the threads working with it. For this purpose, the proposed **mutex and condition variable mechanisms must be used**.

The object to be stored and extracted from the different belts must correspond to a structure defined with the following fields:

- **int num_edition**: represents the order of creation within the belt.
- **int id_belt**: represents the ID of the belt in which the object is created (the factory_manager indicates it to the process by parameter).
- **int last**: it will be 0 if the inserted element is not the last one, and it will be 1 if the object is the last one to be created by the producer.

2.3. *Factory Manager input file*

The main program (factory_manager) must recognize input files in the following format. The information contained is represented as follows: <No. max belts> [<belt ID> <belt size> <No. elements>]+, where:

- **Nº max 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 invalid and the program must terminate by returning -1.
- **Belt ID:** a belt number is assigned to each process_manager process to identify its products in the traces.
- **Belt size:** this is the maximum value of elements that a transport belt can contain (size of the circular buffer between producer-consumer).
- **Number of elements:** Number of elements to be generated by the assigned belt.

The loading of the file data must be performed in the factory_manager process, performing a data integrity check (e.g., you cannot have a number of belts less than or equal to 0).

An example of a valid input file would be:

```
4 5 5 2 1 2 3 3 5 2
```

- A maximum of 4 process_managers can be created.
- The first process_manager, with ID 5, will create a belt with a maximum size for 5 elements, and will have to produce 2 elements.
- The second process_manager, with ID 1, will create a belt with a maximum size for 2 elements, and will have to produce 3 elements.
- Finally, the third process_manager, with ID 3, will create a belt with a maximum size for 5 elements, and will have to produce 2 elements.

NOTE: process_manager IDs do not have to be consecutive. The order must be translated by the factory_manager and preserved using synchronization mechanisms.

2.4. Integration and implementation examples

An example of the factory execution flow could be the following:

- Factory_manager execution:
    - Reads the file with the factory specification, whose name is passed as a parameter.
    - Opens the file, gets the information and closes the file.
    - It creates the synchronization structures necessary for the correct operation of the factory.
    - Creation of the process_manager threads.

- For each thread, control that they are executed in the same order as specified in the input file.
- When all the process_manager threads have finished, the resources are released and the program is finished.

- Execution of the process_manager:
  - Obtaining input parameters.
  - Wait until the factory_manager gives the command to start, by means of a semaphore.
  - Create and initialize the conveyor belt.
  - Create the two light processes: producer and consumer. The first one creates the elements and inserts them into the conveyor until there are no more to produce. The consumer gets the elements from the belt until there are no more to be produced.
  - When the threads are finished, it will release the used resources and terminate the program.

To ensure the correct functioning of the operations, the program must print the following traces (via the standard output, unless otherwise specified) at each specified time:

- *Factory_manager*:
  - When an error occurs in the opening, reading, closing or parsing of the input file, the following message must be returned by the standard error output and return the value -1:
    - "[ERROR][factory_manager] Invalid file."
  - When a process_manager thread is created:
    - "[OK][factory_manager] Process_manager with id <id> has been created."
  - When a process_manager has finished successfully:
    - "[OK][factory_manager] Process_manager with id <id> has finished."
  - When a process_manager has terminated with errors, use the error output:
    - "[ERROR][factory_manager] Process_manager with id <id> has finished with errors."
  - Before the end of the program:
    - "[OK][factory_manager] Finishing."

- *Process_manager*:

- When an error occurs reading the arguments, the error output is used and -1 is returned:
  - "[ERROR][process_manager] Arguments not valid."
- When the arguments have been processed:
  - "[OK][process_manager] Process_manager with id <id> waiting to produce <number of elements> elements."
- When the belt is created:
  - "[OK][process_manager] Belt with id <id> has been created with a maximum of <maximum number of elements> elements."
- When all the elements have been produced:
  - "[OK][process_manager] Process_manager with id <id> has produced <number of elements> elements."
- If any errors have occurred (threads with termination errors, initialization problems or belt destruction, etc.), using the error output:
  - "[ERROR][process_manager] There was an error executing process_manager with id <id>."

- *Queue:*
  - When an element is inserted in the belt:
    - "[OK][queue] Introduced element with id <num_edition> in belt <id cinta>."
  - When an element is removed from the belt:
    - "[OK][queue] Obtained element with id <num_edition> in belt <id cinta>."
  - If any error occurs, it should be displayed by the error output and return -1:
    - "[ERROR][queue] There was an error while using queue with id: <id>."

In general, if the program detects an error, it should immediately return -1. If the execution is correct, it must end with a return 0.

An example of execution could be (taken from the example file in the previous section):

```
shell>./factory_manager input_file.txt
[OK][factory_manager] Process_manager with id 5 has been created.
[OK][factory_manager] Process_manager with id 1 has been created.
[OK][factory_manager] Process_manager with id 3 has been created.
[OK][process_manager] Process_manager with id 5 waiting to produce
2 elements.
[OK][process_manager] Process_manager with id 1 waiting to produce
3 elements.
```

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

[OK][factory_manager] Finishing.
```

EXCEPTION: If an error occurs in any process_manager, the factory_manager should detect it, but not terminate the execution. It should jump to the next process_manager and allow its execution.

NOTE: THESE MESSAGES ARE THE ONLY ONES ACCEPTED. NO DIFFERENT MESSAGES MUST BE INCLUDED IN THE FINAL DELIVERY (they can be added for debugging, but those messages must be removed for delivery).

3. Submission

3.1. Submission Deadline

The laboratory's delivery deadline in AULA GLOBAL is **May 5th, 2025 until 23:55h.**

3.2. Submission Procedure

The delivery of the laboratory must be done electronically. In AULA GLOBAL, links will be enabled through which you will be able to submit the laboratory. Specifically, a submitter will be enabled for the code of the laboratory and another TURNITIN type for the memory of the laboratory.

2.3 Files to be submitted

A compressed file in ZIP format must be submitted with the name:

ssoo_p3_NIA1_NIA2_NIA3.zip

With the NIAs of the members of the group. In case of doing the laboratory alone, the format will be: **ssoo_p2_NIA1.zip.** The file must contain:

- factory_manager.c
- process_manager.c
- queue.c
- queue.h.
- Makefile
- autores.txt

The file memory.pdf must be delivered to the TURNITIN deliverer. The report must contain at least the following sections:

- **Portada:** Cover page: with authors' full names, NIAs, group, and e-mail addresses.
- **Index:** with navigation options to each section of the memory.
- **Description of the code:** detailing the main functions implemented. Do NOT include source code from the lab in this section. Any code will be ignored.

- **Set of tests** used and results obtained for both programs (Scripter and mygrep). Higher scores will be given to advanced tests, extreme cases, and to those tests that guarantee the correct functioning of the laboratory in all cases. It must be considered:
    - If a program compiles correctly and without *warnings*, it is not guaranteed that the program works correctly.
    - Avoid duplicate tests that assess the same program streams. The score for this section is not measured based on the number of tests but by covering a wide range of scenarios. Few tests assessing different cases are better than many considering the same case.
- **Conclusions** describe the problems encountered, how they have been solved, and opinions.

The following aspects relating to the **presentation** of the laboratory will also be evaluated:

- It must have a cover page, with the authors of the practice and their NIAs.
- It must have a navigable index.
- The report should have page numbers on all pages (except the title page).
- The text of the report must be justified.

The report has a page limit of 15 pages (cover page and table of contents included). You must obtain an approbatory score on the report to get an approbatory score on the lab, so the quality of the report must not be neglected.

NOTA: The report can only be submitted once via TURNITIN.

4. Rules

1) **Programmes that do not compile or do not satisfy the functionality and requirements will receive a score of 0.**
2) **Special attention will be paid to detect copied functionalities between two laboratories. In case of finding common implementations, the students involved (copied and copiers) will lose the grades obtained for continuous assessment.**
3) **Programs must compile without warnings.**
4) **The programs must run under a Linux system; the laboratory is not allowed to be carried out on Windows systems. In addition, to ensure the correct functioning of the laboratory, you must check its compilation and execution in the university's computer laboratories (on the server guernika.lab.inf.uc3m.es). If the code submitted does not compile or does not work on these platforms, the implementation will not be considered correct.**
5) The laboratory will be delivered through Aula Global, as detailed in this document's "**Submission**" section. Submissions via email are not allowed without prior authorisation.
6) Students must follow the guidelines on the input and output formats specified for each program.
7) Error checking must be performed for each program.

5. Annexes

5.1 man function

man is the pager for the system manual, i.e. it allows you to search for information about a program, a utility or a function. See the following example:

man 2 fork

The pages used as arguments when running *man* are usually names of programs, utilities or functions. Normally, the search is carried out in all available manual sections in a predetermined order, and only the first page found is presented, even if that page is found in multiple sections.

To exit the displayed page, simply press the 'q' key.

A manual page has several parts. These are labelled NAME, SYNOPSIS, DESCRIPTION, OPTIONS, FILES, SEE ALSO, BUGS, and AUTHOR. The SYNOPSIS tag contains the libraries (identified by the *#include* directive) that must be included in the user's C program in order to make use of the corresponding functions.

The most common ways of using *man* are as follows:

- **man item section:** Displays the item page available in the manual section.
- **man -a element:** This displays, sequentially, all the pages of elements available in the manual. Between pages, you can choose to jump to the next page or exit the pager completely.
- **man -k keyword:** Searches for the keyword among the short descriptions and man pages and displays all matching ones.

5.2 Semaphores

POSIX semaphores help processes synchronize their actions. A semaphore is an integer whose value will never be less than 0. On this value, two actions are allowed: incrementing the semaphore (sem_post), and decrementing the semaphore (sem_wait). If the semaphore value is 0, the sem_wait action will block until the semaphore is greater than 0 again. When several processes want to access a resource controlled by a semaphore, they reduce the counter by 1, and when it reaches 0 no process will be able to access the resource. Once the task is completed, a process must reset its counter by adding 1 to the semaphore so that others can access it.

In order to use these elements, semaphore.h must be included in the file that will require the semaphores and the linking with the pthread library must be executed.

6. References

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (man function)