Computer Architecture and Technology Area

Universidad Carlos III de Madrid

Operating Systems

Programming assignment 3. Multithread. Manufacturing process control.

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

Year 2016/2017

uc3m	**Bachelor’s Degree in Computer Science and Engineering** **Operating Systems (2016-2017)** **Programming assignment 3 - Multithread**	ARCOS

Índice

Introduction

This programming assignment allows the student to become familiar with services used for process management provided by POSIX.

For the management of heavy processes, the functions *fork, wait* and *exit* will be used. For the synchronization of the different processes the student must use **named semaphores.**

For the management of lightweight processes (threads), the functions *pthread_create, pthread_join, pthread_exit,* will be used and, mutex and conditional variable will be used for the synchronization between them.

The objective of this assignment is to design and implement in C and for LINUX operating systems, a program that acts as a manufacturing process manager. It will include several processes that will be in charge of managing different phases of the factory and a process that will be in charge of planning the different phases.

1. Description of the programming assignment

The objective of this programming assignment is to implement a simulation of the operation of a factory in which there exist different roles. The roles should work concurrently allowing a correct movement and management of the elements generated in the factory.

The existing roles are:

- **Factory manager:** Is a heavy process whose main function is to create and synchronize the processes that will be producing elements inside the factory (*process_manager)*. The *factory_manager* can initiate as many *process_manager* processes as indicated in the parameters file.

- **Process manager:** Is a heavy process in charge of initiating the lightweight processes that will work along the transport belts to generate a *producer-consumer* system. The elements that intervene in this subsystem are:

    - **Producer:** Is a lightweight process that will be in charge of producing as many elements as indicated by the process manager, and put them at disposal of the *consumer* through a transport belt that will communicate both processes.

    - **Consumer:** Is a lightweight process that will be in charge of collecting as many elements as the *producer* produces in the transport belt.

Therefore, the *factory_manager* acts as the chief of the factory, that communicates with his *n* managers *(process_manager)* and gives them information about how many elements each of them has to produce. Finally, each *process_manager* will dispatch two workers (*producer and consumer)* that will perform the tasks commissioned by the chief.

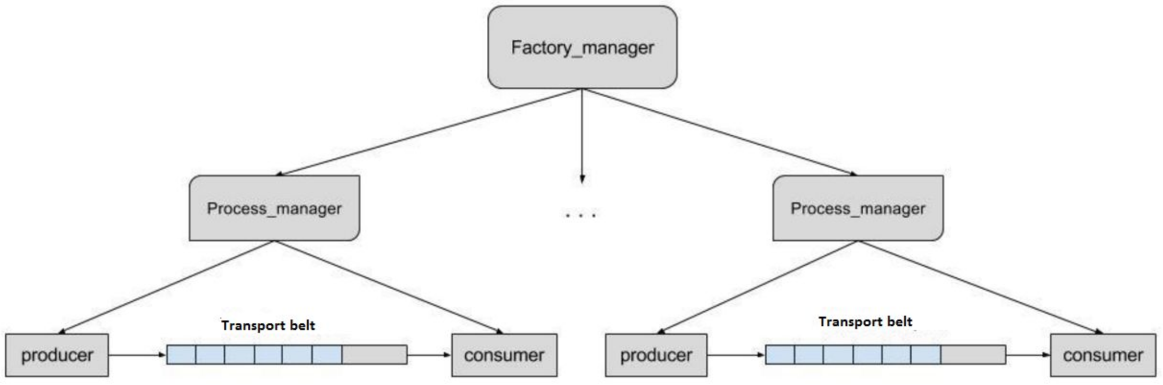

Ilustración 1. Example of the operation of the programs.

1.1. Factory manager

The functionality of the *factory_manager* must be implemented in a file named **factory_manager.c**. It will only receive one argument: the path to a file which will contain the input parameters that will configure the operation of the factory (described in section 2.3). The tasks of this role are:

- To retrieve the input parameters of the factory that are **introduced in a text file.**
- To launch a number of *process_manager* processes indicated by the parameters.
- To synchronize the execution of the different *process_manager* processes following the order given by the parameters.

When the *factory_manager* creates a process *process_manager,* it should indicate the following information: **<id> <name of the semaphore> <maximum size of the belt> <number of products to be generated>**, where:

- **Id:** is the identifier of the assigned belt (indicated by the data included in the input file).
- **Id of the semaphore:** the synchronization between the heavy processes is done using a named semaphore controlled by the *factory_manager*, whose name must be given to the children processes as parameter.
- **Maximum size of the belt:** is the maximum number of elements that can be stored in the transport belt (size of the circular buffer).
- **Number of products to be generated:** each worker *process_manager* has a number of products to be generated, indicated by the input file and assigned using this parameter.

The factory will have a maximum size defined in the input file, and there will be no more processes than the ones specified there (there could be less).

1.2. Process manager

The manufacturing processes are the ones in charge of performing the task of launching the lightweight processes (*producer-consumer*). The functionality must be implement in a file named **process_manager.c**. Their arguments have been defined in the previous section.

Each *process_manager* created must generate a *producer-consumer* system implemented with *threads.* The thread producer will insert elements into a transport belt, while the consumer thread will be in charge of receiving the elements from the belt.

This part implies a synchronization between the threads over a shared resource (the transport belt), so it is necessary to control the access to this resource when inserting and receiving from the belt. When all the elements have been inserted and received, both *producer and consumer* threads should finalize.

1.2.1 Queue

The communication between producers and consumers must be implemented using the transport belt. These belts correspond to queues implemented with a circular buffer. One circular buffer must be created for each *producer-consumer* system. Concurrency mechanisms must be implemented to control the access to the elements of the belt. For this purpose the student must use two mechanisms: *mutex* and *conditional variables*.

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

- **Int Queue_init(int num_elements)**: function that creates the queue and reserves memory for the size specified as parameter.
- **Int Queue_destroy(void)**: function that removes the queue and frees the used resources.
- **Int Queue_put (struct element * ele)**: function that inserts an element into the queue if the queue is not full. If there is no space left, it should wait until the insertion can be executed.
- **Struct element * Queue_get (void)**: function that extracts an element from the queue. If the queue is empty it should wait until an element is available.
- **Int Queue_empty()**: function that checks the state of the queue and determines if it is empty (returns 1) or not (returns 0).
- **Int Queue_full()** : function that checks the state of the queue and determines if the queue

is full (returns 1) or not (returns -1).

The object that must be stored and extracted from the belts must be implemented as a struct (*struct element*) with the following fields:

- **Int num_edition**: represents the order of creation inside the belt. It will start from 0 and will be incremented by one for each new element.
- **Int id_belt**: represents the id of the belt in which the object is created (this identifier is passed as a parameter to the program).
- **Int last**: it will be 0 if it is not the last element to be inserted in the belt and 1 if it is the last one to be inserted.

1.3. Input file of the factory_manager

The main program (*factory_manager*) must be able to read the input files that have the following format. The information is represented in the following way:: **<maximum number of process managers> [<id process manager> <size of the belt> <number of elements>]**$^+$, where:

- **Maximum number of belts:** is the maximum number of processes *process_manager* that can be executed in parallel. If after reading the file it is detected that more *process_manager* processes are needed than the maximum specified, the file is invalid and the program should exit with a -1.
- **Id process manager:** one id is assigned to each process_manager to identify their products.
- **Size of the belt:** is the maximum number of elements that can be stored in the transport belt (size of the circular buffer).
- **Nº elements:** number of elements that must be generated in this *process_manager.*

This file must be read in the process *factory_manager*, checking that the data is correctly structured (for example, there can not exist a number of belts less or equal to 0).

An example of a valid input file would be:

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

In this case, a factory of a maximum of 4 *process_manager* would be created:

- First, the *process_manager* number 5 would create a belt with a maximum of 5 elements and the producer and consumer would move 2 elements.
- Then, the *process_manager* number 1 would create a belt with a maximum of 2 elements and the producer and consumer would move 3 elements.
- Finally, the *process_manager* number 3 would create a belt with a maximum of 5 elements and the producer and consumer would move 2 elements.

NOTE: the id included for each *process_manager* does not have to consist of consecutive numbers. The order in which these elements are introduced in the file will be translated in the order of execution that the *factory:manager* will preserve using the synchronization mechanisms.

1.4. Integration, messages, error codes and execution examples

The operation of the whole factory would follow the next flux of execution:

- Execution of *factory_manager*:
  - It reads the path of the file with the parameters.
  - It opens the file, acquire the information and closes the file.
  - Creation of the synchronization structures necessary for the correct operation of the factory (check semaphores).
  - Creation of all the *process_manager* processes.
  - For each process, make each of the *process_manager* execute following the same order that is given in the input file.
  - When all *process_manager processes* have finished, free the resources and end the program.
- Execution of *process_manager:*
  - Acquires the input parameters (id, name of the semaphore, size of the belt and number of elements). The process should wait until the *factory_manager* indicates that it must start to produce.
  - Create and initialize the belt.
  - Create two lightweight processes *producer* and *consumer*. The first will produce an element and insert it in the belt until all specified elements have been produced. The consumer will obtain an element from the belt until there exist no more elements (last==1). When they have finished producing and consuming they will finish.
  - After the threads have finished it will destroy all the associated resources and end the program.

To assure the correct operation of your assignment the programs implemented must print the following traces (**through standard output unless specified otherwise**) at the moments specified:

- *Factory_manager:*
    - When an error occurs when opening, reading, closing or analyzing the input file, it must print the following message using **standard error** and exit with code -1:
        - "[ERROR][factory_manager] Invalid file".
    - When a process is created:
        - "[OK][factory_manager] Process_manager with id <id> has been created."
    - When a process has finished (successfully):
        - "[OK][factory_manager] Process_manager with id <id> has finished."
    - When a process has finished (with errors) using **standard error**:
        - "[ERROR][factory_manager] Process_manager with id <id> has finished with errors."
    - Before finishing;
        - "[OK][factory_manager] Finishing."
- *Process_manager*:
    - When an error occurs reading the arguments using **standard error** and return -1:
        - "[ERROR][process_manager] Arguments not valid."
    - When the arguments have been read (before factory_manager asks to produce):
        - "[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 elements have been produced:
        - "[OK][process_manager] Process_manager with id: <id> has produced <number of elements> elements."
    - If there was an error (threads finish with errors, problems initializing or destroying the belt...etc) through standard error and returning -1:
        - "[ERROR][process_manager] There was an error executing process_manager with id: <id>."
- *Queue:*
    - When an element is introduced in the queue:
        - "[OK][queue] Introduced element with id: <num_edition> in belt <id of the belt> .“
    - When an element is obtained from the queue:
        - "[OK][queue] Obtained element with id: <num_edition> in belt <id of the belt> .“
    - If there was an error through **standard error** and returning -1:
        - "[ERROR][queue] There was an error while using queue with id: <id>."

In general if a program detects and error it must return immediately with a -1. If the execution was successful it should finish with 0.

An example of execution would be (taken from the example file shown 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 occurred in a *process_manager* the *factory_manager* should detect the error but not finish the execution. It would just jump to the following *process_manager* to be executed.
NOTE: These messages are the only ones accepted. No other messages should appear in the screen.

2. Submission

2.1. Deadline

The deadline for the delivery of this programming assignment in AULA GLOBAL will be **Tuesday 28th April (until 23:55h).**

2.2. Submission

The submission must be done using Aula Global using the links available in the first assignment section. The submission must be done separately for the code and using **TURNITIN** for the report.

2.3. Files to be submitted

You must submit the code in a zip compressed file with name ssoo_p3_AAAAAAAAA_BBBBBBBBB.zip where A…A and B…B are the student identification numbers of the group. A maximum of 2 persons is allowed per group, if the assignment has a single author, the file must be named ssoo_p2_AAAAAAAAA.zip. The file to be submitted must contain:

- factory_manager.c
- process_manager.c
- queue.c

- Queue.h: DO NOT MODIFY
- Makefile: DO NOT MODIFY

The report must be submitted in a PDF file through TURNITIN. Notice that only PDF files will be reviewed and marked. The file must be named **ssoo_p2_AAAAAAAAA_BBBBBBBBB.pdf**. A minimum report must contain:

- **Cover** with the authors (including the complete name, NIA, number of group and email address).
- **Table of contents**
- **Description of the code** detailing the main functions and functionalities it is composed of. Do not include any source code in the report.
- **Tests cases** used and the obtained results. All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to take into account.
    - Avoid duplicated tests that target the same code paths with equivalent input parameters.
    - Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the number of tests per program.
    - Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Additionally, marks will be given attending to the quality of the report. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain an index.
- Every page except the title page must be numbered.
- Text must be justified.

The PDF file must be submitted using the TURNITIN link. Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

NOTE: Only the last version of the submitted files will be reviewed.

NOTE: **The maximum length of the report is 15 pages including cover and index.**

3. Rules

1) **Programs that do not compile or do not satisfy the requirements will receive a mark of zero.**

2) **All programs should compile without reporting any warnings.**

3) **Programs without comments will receive a grade of 0.**

4) **The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.**

5) **The programs implemented must work in the computers of the informatics lab in the university or the Guernika (guernika.lab.inf.uc3m.es) platform. It is the student responsibility to be sure that the delivered code works correctly in those places.**

6) **It is mandatory to follow the input and output formats indicated in each program implemented. In case this is not fulfilled there will be a penalization to the mark obtained.**

7) **It is mandatory to implement error handling methods in each of the programs.**

8) **Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be filled.**

Failing to follow these rules will be translated into zero marks in the affected programs.

4. Appendix

4.1. Manual (man command).

man is a command that formats and displays the online manual pages of the different commands, libraries and functions of the operating system. If a *section* is specified, man only shows information about name in that section. Syntax:

$ man [section] name

A man page includes the synopsis, the description, the return values, example usage, bug information, etc. about a *name*. The utilization of man is recommended for the realization of all lab assignments. **To exit a man page, press *q*.**

The most common ways of using man are:

1. **man section element:** It presents the element page available in the section of the manual.

2. **man –a element:** It presents, sequentially, all the element pages available in the manual. Between page and page you can decide whether to jump to the next or get out of the pager completely.

3. **man –k keyword** It searches the keyword in the brief descriptions and manual pages and present the ones that coincide.

4.2. Semaphores

The semaphores in POSIX help processes to synchronize their actions. A semaphore is an integer number whose value is never going to be less than 0. About this value, two actions are allowed: to increment the semaphore (*sem_post*), and decrement the value of the semaphore (*sem_wait).* If the value is 0, the action *sem_wait* will block the execution until the semaphore has again a value greater than 0.

There are two types of semaphores in POSIX. IN this programming assignment only **named semaphores** will be used. This type of semaphore is identified by a name of type "/*somename*" and the processes can use them passing as parameter this name to the function *sem_open*. The rest of functions that can be used are: *sem_close*, *sem_destroy*, *sem_getvalue, sem_init, sem_unlink.*

Bibliografía

- 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)