LAB – C2

SISTEMES OPERATIUS

Bru Gladiador

Rubén Gómez

ÍNDEX

Tabla de contenido

[Activity 1: Processes, shared memory and semaphores 2](#_Toc130557807)

[Compilation and Executation 2](#_Toc130557808)

[Code Explanation 3](#_Toc130557809)

[Activity 2: Processes and Named pipes 6](#_Toc130557810)

[Compilation and Executation 6](#_Toc130557811)

[Code Explanation 6](#_Toc130557812)

[Activity 3: Threads and mutexes 7](#_Toc130557813)

[Compilation and Executation 7](#_Toc130557814)

[Code Explanation 7](#_Toc130557815)

# Activity 1: Processes, shared memory and semaphores

To make this activity we need to create two processes inside a program named activity2.c, one shared memory space, and two named semaphores.

* To create the shared memory space, you can use the **shm\_open** function call.
* To limit the space to 4 bytes you can use the **ftruncate** function call.
* To map the memory space in the process, you can use the **mmap** function.

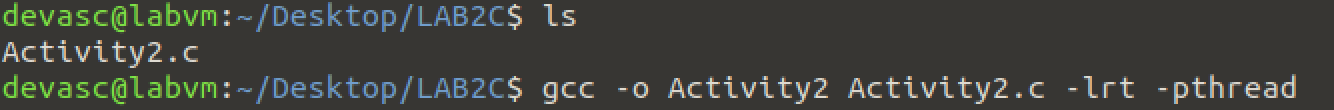
The semaphores will be named "/activity2\_sem1" and "/activity2\_sem2” and will be used to indicate each process’s turn to write to shared memory.

* To create and initialize semaphores, you can use the **sem\_open** and **sem\_init** calls.
* To get and release the semaphores, you can use the **sem\_wait** and **sem\_post** calls.
* Finally, remember to release the semaphores with **sem\_close**.

## **Compilation and Executation**

First of all, we need to open a terminal in the directory where we have the "activity2.c" file.

Next we are going to enter the following command to compile the source code:

****This command compiles the "activity2.c" file and creates the "activity2" executable file. The "-lrt" option is necessary to use the functions related to shared memory, and the "-pthread" option is necessary to use the functions related to semaphores.

Once the program has been compiled successfully, we are going to execute with the following command:

**Texto

Descripción generada automáticamente**

This command runs the "activity2" file and starts the simulation of the processes that communicate through shared memory and semaphores.

Additionally, after each execution, it is important to free the resources used by the program through the cleanup commands that are at the end of the code.

## **Code Explanation**

The code includes several header files. These header files provide various functions and data types needed for the program.

Texto

Descripción generada automáticamente

The main function starts by declaring two semaphores and a pointer to a shared memory region.

Texto

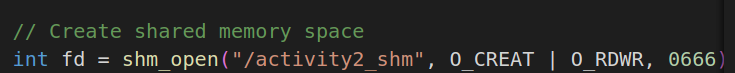
Descripción generada automáticamente

The first semaphore is created using sem\_open() with the name "/activity2\_sem1", and the initial value of 1. The second semaphore is created similarly, with the name "/activity2\_sem2" and the initial value of 0.

Texto

Descripción generada automáticamente

The shared memory region is created using shm\_open() with the name "/activity2\_shared\_mem. This creates a shared memory region that can be accessed by multiple processes.



The shared memory region is truncated to the size of a 32-bit integer using ftruncate().



The shared memory region is mapped into the process's address space using mmap(). The mmap() function returns a pointer to the mapped memory.

Texto

Descripción generada automáticamente

The parent process generates a random number between 1 and 100 using rand().

Texto

Descripción generada automáticamente

The parent process creates a child process using fork().

Imagen de la pantalla de un celular

Descripción generada automáticamente con confianza media

Both parent and child processes enter into an infinite loop, taking turns decreasing the value of the number written to shared memory until it reaches zero.

The parent process waits on sem1 using sem\_wait(), which blocks the parent process until sem1 is released.

After the parent process is unblocked by sem1, it decreases the value in the shared memory region by 1 and prints out the updated value.

The parent process releases sem2 using sem\_post(), which allows the child process to acquire sem2 and write to shared memory.

The child process waits on sem2 using sem\_wait(), which blocks the child process until sem2 is released.

After the child process is unblocked by sem2, it decreases the value in the shared memory region by 1 and prints out the updated value.

The child process releases sem1 using sem\_post(), which allows the parent process to acquire sem1 and write to shared memory.

Both parent and child processes continue taking turns decreasing the value of the number in shared memory until it reaches zero.

Once the value in shared memory reaches zero, both parent and child processes exit the infinite loop.

The parent process waits for the child process to exit using wait().

The shared memory region is unmapped from the process's address space using munmap().

The shared memory region is closed using close().

The shared memory region is unlinked from the file system using shm\_unlink().

Both semaphores are closed using sem\_close().

Both semaphores are unlinked from the file system using sem\_unlink().

The program exits with a return value of 0.

# Activity 2: Processes and Named pipes

This program uses named pipes to allow communication between the parent and child processes. The parent process generates random operands and an operation, writes them to the pipe, and repeats this process for the specified number of operations. The child process reads the operands and operation from the pipe, performs the corresponding mathematical operation, and prints the result to the screen.

## **Compilation and Executation**

First, we need to open a terminal in the directory where we have the "activity1.c" file.

Next, we are going to enter the following command to compile the source code:



The command "gcc activity1.c -o activity1" compiles the source code file "activity1.c" and links it to an executable file called "activity".

The "gcc" command invokes the GNU Compiler Collection.

The "-o" option specifies the name of the output file. In this case, the output file will be named "activity1".

Once the program has been compiled successfully, we are going to execute with the following command:



This command runs the "activity1" file and starts the simulation of the processes that communicate through shared memory and semaphores.

## **Code Explanation**

The program includes standard C libraries such as "stdio.h", "stdlib.h", and "stdint.h". It also includes POSIX libraries such as "unistd.h", "fcntl.h", "sys/stat.h", and "sys/types.h".

Texto

Descripción generada automáticamente

The program defines a named pipe and declares three global variables.

Texto

Descripción generada automáticamente

The "get\_operation()" function generates random values for "number1", "number2", and "operation". The values for "number1" and "number2" are randomly generated integers between 1 and 100, while "operation" is randomly selected.

Texto

Descripción generada automáticamente

The "parent()" function creates a named pipe using "mkfifo()" and opens the pipe for writing using "open()". It then generates "total\_operations" arithmetic operations by repeatedly calling the "get\_operation()" function and writing the values of "operand1", "operand2", and "operation" to the pipe using "write()". Finally, it closes the pipe using "close()".

Texto

Descripción generada automáticamente

The "child()" function opens the named pipe for reading using "open()" and repeatedly reads the values of "number1", "number2", and "operation" from the pipe using "read()". It then performs the arithmetic operation based on the value of "operation" and prints the result to the console using "printf()". This process continues until there are no more values to read from the pipe.

Texto

Descripción generada automáticamente

The "main()" function first checks if the program is called with one argument using "argc" and "argv[]". If the program is not called with exactly one argument, it prints a usage message and exits. Otherwise, it converts the argument to an integer using "atoi()" and stores it in "total\_operations". It then creates a child process using "fork()" and executes the "child()" function in the child process and "parent()" function in the parent process.

Texto

Descripción generada automáticamente

# Activity 3: Threads and mutexes

In this program, we define a thread\_data\_t structure that contains a pointer to the shared variable turns, as well as pointers to the two mutexes that will be used for synchronization. We define two thread functions thread\_func1 and thread\_func2 that take a thread\_data\_t argument and use the mutexes to synchronize the threads and access the shared variable. We also define the main function, which initializes the mutexes, creates the threads, waits for their completion using pthread\_join, and destroys the mutexes.

To compile the program, we need to link it with the pthread library by adding -lpthread to the compiler flags in the Makefile.

## **Compilation and Executation**

First, we need to open a terminal in the directory where we have the "activity3.c" file.

Next, we are going to enter the following command to compile the source code:



This command compiles the "activity3.c" file and creates the "activity3" executable file. The "-lrt" option is necessary to use the functions related to shared memory, and the "-pthread" option is necessary to use the functions related to semaphores.

Once the program has been compiled successfully, we are going to execute with the following command:



This command runs the "activity3" file and starts the simulation of the processes that communicate through shared memory and semaphores.

## **Code Explanation**

This code demonstrates the use of mutexes in a multithreaded program to synchronize access to a shared resource. The program creates two threads that alternate printing numbers to the console until a specified number of turns has been completed.

The program starts by parsing a command-line argument to determine the number of turns to perform. It then initializes two mutexes using pthread\_mutex\_init.

Two structures of type thread\_data\_t are defined to store data that is passed to each thread. The thread\_data\_t struct contains a pointer to the shared resource (data\_ptr) and pointers to two mutexes (mutex1 and mutex2).

Two threads are created using pthread\_create, each with its own thread\_data\_t structure. The first thread executes thread\_func1, and the second thread executes thread\_func2.

thread\_func1 and thread\_func2 are similar in structure. They both run in a loop until the specified number of turns has been reached. They both lock a mutex (either mutex1 or mutex2, depending on the thread), check if there are any turns left to perform, print a message to the console, decrement the turn counter, and unlock the other mutex.

The pthread\_mutex\_lock and pthread\_mutex\_unlock functions are used to lock and unlock the mutexes, respectively. These functions ensure that only one thread can access the shared resource at a time.

Finally, the main thread waits for the two threads to finish using pthread\_join, and then destroys the mutexes using pthread\_mutex\_destroy.