TecnoCampus
Escola Superior
Politècnica

Centre adscrit a:

Universitat
Pompeu Fabra
Barcelona

Operating Systems

PRAC 3: Concurrent programming

Dr. Léonard Janer
Degree in Computer Engineering for Management Information Systems
Escola Superior Politècnica del TecnoCampus

Course 2022-2023

Introduction

Presentation

This third PRAC consists of three activities focused on how to use the libraries of the GNU/Linux operating system to:

- Create, use and destroy processes and threads.
- Create, use and destroy pipes and shared memory.
- Create, use and destroy semaphores and mutexes.

Responsible Lecturers

- Dr. Léonard Janer <leonard@tecnocampus.com>

Competencies

The specific and transversal competences that are worked on in this PRAC are those described below:

- Specific:
  - Ability to analyze a problem at the appropriate level of abstraction for each situation and apply the skills and knowledge acquired to address and solve it.
- Transversal:
  - Capacity for written communication in the academic and professional field.
  - Ability to communicate in a foreign language.
  - Ability to adapt to new technologies and environments.

Objectives

The objectives of this PRAC are:

- Learn to develop concurrent software using the C programming language and using the GNU/Linux operating system and its libraries to create, use and destroy processes, threads, pipes, shared memory, semaphores and mutexes.

Resources

The following resources are available for the correct development of this PRAC:

- Basics:
  - Subject materials available on the Virtual Campus
- Complementary:
  - Devasc VM (provided on eCampus)

Evaluation criteria

The weight of each question is indicated in the statement of each exercise and/or in the rubric of the activity on campus. The clarity and justification of the answers presented will be valued.

Format and delivery date

A PDF document must be submitted with the answer to the questions clearly written, as well as **all** the programmed source files. The delivery will be made in the GIT repository that will be accepted to perform the PRAC.

The PDF report document must have a cover, with a name, title, date, an index, a section for each activity and some bibliographical references.

Screenshots should be included, with: examples how to compile all the codes, how to execute all the codes, and execution examples to show the correct features of the solution proposed. The relevant elements of the code should be explained. It is not enough to include the code, if it is not explained in the report: you have to write a code, and write

a report. The report is not just the code; the report must be a document to explain the code, with the structure of the program explained and screenshots.

Activities

To carry out this PRAC, you are asked to read and respond to the activities that are presented below, taking into account the following aspects.

Regarding the implementation of the different activities, it is requested to take into account:

- The folders with the files of each activity and a document in PDF format with the response to the activities must be delivered to the repository, which will be the explanatory and demonstrative report of each activity
- In the repository the activity source files must be inside a folder named `activity_x`, where x is the number of the activity, and each folder must contain the source files (`. c` and `.h`), as well as a `Makefile` file that allows you to compile the activity code using the parameterless `make` command (if used).
- In the PDF document it is necessary to include for each activity a description of the operation of the code, the process of compilation and execution of the program, and a description of the result obtained (including screenshots). You have to show (clearly, with screenshots) the procedure to obtain the executables of each activity, and you have to show (clearly, with screenshots) the procedure for executing the code. In fact, for the executions you must attach screenshots of the executions where you have to show the correct operation of the proposed solution.

Regarding the implementation of the programs that make up each activity, it is requested to take into account the following good practice tips:

- Be careful not to use *magic numbers*, so all but trivial numerical values that appear in your code should be put in a `#define` directive.
- It is recommended to initialize all variables with a default value, including vectors. In this case, you can use the `memset` function.

- To convert operating system numeric values to network numeric values you have to use the `htons` and `htnonl` functions. In the reverse direction, you have to use the `ntohs` and `ntohl` functions.
- At the end of the execution, it is necessary to release the resources used. Specifically, it is necessary to free the file descriptors associated with the communication *socket* and the log file.
- In case of using dynamic memory, it is also important to free any reservations that have been made using the call to `malloc`. This requires a to `free`.

The practical score will be based on the campus rubric.

As can be seen, both the code and the comments are valued to understand the proposed solution, and the screenshots that show how it is compiled, and the execution of the code, where you have to show the management of input parameters and the correct operation of the code.

Activity 1: Processes, shared memory and semaphores [40 %]

Write a program named `activity2.c` that creates two processes (i.e., parent and child), one shared memory space, and two named semaphores.

The program should generate a random number which should be written to shared memory. After that, the two processes will take turns decreasing the value of the number written to shared memory until it reaches zero. Once it reaches zero, the program must end, and it is necessary to free the resources (i.e., shared memory space and semaphores).

To carry out the implementation, take into account that the shared memory space must store a 32-bit variable (4 bytes). 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. Finally, remember that at the end of the program execution it is necessary to unmap, close and free the shared memory space and the file descriptor with the functions `munmap`, `close` and `unlink`.

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`.

Below is an example of executing the program of Activity 1.

```
./activity1
main: bouncing for 10 times.
parent (pid = 4066) begins.
parent (pid = 4066) bounce 9.
child (pid = 4067) begins.
child (pid = 4067) bounce 8.
parent (pid = 4066) bounce 7.
child (pid = 4067) bounce 6.
parent (pid = 4066) bounce 5.
child (pid = 4067) bounce 4.
parent (pid = 4066) bounce 3.
child (pid = 4067) bounce 2.
parent (pid = 4066) bounce 1.
child (pid = 4067) bounce 0.
child (pid = 4067) ends.
parent (pid = 4066) ends.
```

Code 1: Execution example for the `activity1.c` program.

Activity 2: Processes and Named pipes [40 %]

Write a program named `activity1.c` that creates two processes (i.e., parent and child) and a named pipe that allows them to communicate. Once created, the parent process must generate three random numbers that will correspond to two operands and the operation of the formula axb, where a is the first operand, x is the mathematical operation and b is the second operand. The operands must be positive numbers between 0 and 100, while the mathematical operation can be addition $(+)$, subtraction $(-)$, multiplication $(\cdot)$, or division $(/)$. To generate values between the specified ranges, the modulo operation can be used.

Once these values are generated, the parent process will write them one by one through the pipe. Remember that each operand can be represented by a number `uint8_t` (8-bit unsigned, see library `stdint.h`) and that the operation can be represented by a `char` (8-bit), so you can perform three write operations on the pipe with a length of 1 byte each.

On its behalf, the child process must perform three 1-byte read operations, so it

will retrieve the byte corresponding to the first operand, the operation, and the second operand. After that, the child process must perform the mathematical operation with the operands and write the result on the screen.

Finally, the process of generating random numbers and sending them through the pipe (by the parent) and reading the pipe and resolving the operation (by the child) must be repeated a number of times passed by parameter to the program via arguments (i.e., `argc` and `argv`). Remember that the child process inherits the parent's variables, so there is no need to pass parameters between the parent and child process if you use a global variable.

Below is an example of executing the program of activity 2.

```
./activity2 3
main: created pipe.
main: open pipe for read/write.
parent (pid = 751) begins.
parent (pid = 751): iteration 0.
child (pid = 752) begins.
parent (pid = 751): 98 * 43 = ?
child (pid = 752): 98 * 43 = 4214
parent (pid = 751): iteration 1.
parent (pid = 751): 74 - 29 = ?
child (pid = 752): 74 - 29 = 45
parent (pid = 751): iteration 2.
parent (pid = 751): 63 + 34 = ?
child (pid = 752): 63 + 34 = 97
child (pid = 752) ends.
parent (pid=751) ends.
```

Code 2: Execution example for the `activity2.c` program.

Activity 3: Threads and mutexes [20 %]

Write a program named `activity3.c` that implements the same functionality as Activity 2, but using threads instead of processes and mutexes instead of semaphores.

To pass information to the threads it is necessary to use a data structure called `thread_data_t` that contains a pointer to the shared variable that contains the number of turns to perform until reaching zero. In addition, this data structure must also contain pointers to the two mutexes that will be used to synchronize the execution of the two threads, as well as ensure mutual exclusion when accessing the shared variable. The

definition of the data structure `thread_data_t` is as follows:

```
typedef struct {
    uint32_t * data_ptr;
    pthread_mutex_t * mutex1;
    pthread_mutex_t * mutex2;
} thread_data_t;
```

Code 3: Declaration of the `thread_data_t` data type.

To create the threads you will need to use the `pthread_create` function, while to create the mutex you will need to use the `pthread_mutex_init` function. Once created, to perform synchronization between threads and ensure mutual exclusion with mutex you will need to use the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. Also, in order for the main program to wait for the completion of threads you will need to use the `pthread_join` function. Finally, to destroy the mutex you will need to use the `pthread_mutex_destroy` function. Finally, remember that for the program to compile correctly, you must link it with the `pthread` library. To do this, you need to add the `-l pthread` option in the Makefile.

Below is an example of executing the program of Activity 3.

```
./activity3
main: bouncing for 11 times.
thread1 begins, 11.
thread1 bounce 10.
thread2 begins, 11.
thread2 bounce 9.
thread1 bounce 8.
thread2 bounce 7.
thread1 bounce 6.
thread2 bounce 5.
thread1 bounce 4.
thread2 bounce 3.
thread1 bounce 2.
thread2 bounce 1.
thread1 bounce 0.
thread2 ends.
thread1 ends.
```

Code 4: Execution example for the `activity3.c` program.