Sysdev - POSIX programming

Objective: Experiment the basic POSIX multi thread APIs and inter-process communication APIs under Linux

Setup

Go in the `/home/<user>/felabs/sysdev/posix/` directory, the recommended location to store your lab files.

Threads

To start, create a simple application that runs two threads, each printing a different message indefinitely, by using `pthread_create()` and compiling with `-lpthread`. What happens when you run the program ?

The main() function exits as soon as the two threads are created. You need to join the threads in order to wait for their completion, using `pthread_join()`. Do it and try again the application.

Now, change the code so that the two threads increment the same global counter 1.000.000 times, and display the value of the global counter at the end. What happens ?

Threads and synchronization

Add synchronization with mutexes around accesses to the global counter, and make sure that the result is now correct.

Threads and conditions

Implement a program with two threads:

- the first thread increments a global counter when it is not a multiple of 10;
- the second thread increments the global counter when it is a multiple of 10, and displays the value.

Of course, both threads must synchronize their actions, preferably using conditions. The incrementation should be stopped when the counter reaches 1.000.000.

Processes and shared memory

Write two separate applications that share memory using the `shm_open()` function call followed by a memory mapping using `mmap()`. The first application will write a different string to the shared memory region every 3 seconds and the second application will read the string from the shared memory region every second and display it on the screen.

Processes and notification

Let's add a notification mechanism between these two processes: when the writer process has written a new message, it sends a signal to the reader process so wake it up.

The writer program should:

- Take as argument the PID of the reader process, in order to be able to send a signal to it;
- Every time a string is written to the shared memory segment, a signal is sent to the reader process using the `kill()` function. The `SIGUSR1` signal will be used for the notification.

The reader program will:

- Block the delivery of SIGUSR1 with `sigprocmask()` so that the signal handler is not called;
- Wait in an infinite loop for the signal using `sigwaitinfo()` and display the message stored in the shared memory segment.