Universitat Pompeu Fabra Barcelona

Lab 05: Delivery Exercise

Understanding Threads and Synchronization

In this lab, we will implement a chained producer-consumer system using threads, finite-size buffers, mutexes, and condition variables. The goal is to correctly synchronize multiple threads while ensuring efficient communication between them, avoiding race conditions and active waiting.

You have to implement a producer-consumer model with two buffers, using three different types of threads. The program will continuously read input from the keyboard and process it through multiple threads before outputting the results to the console.

The program will work as follows:

A producer thread will read space-separated integers from the keyboard, convert each read value to an integer, and store it in `buffer_1`. A consumer-producer thread will read from `buffer_1`, compute its *Fibonacci* number (the function is given below), and store the result in `buffer_2`. Finally, a consumer thread will read from `buffer_2` and print the final computed values to the standard output.

The program must use two finite-size buffers: `buffer_1`, which stores the binary representations of integers, and `buffer_2`, which holds the computed Fibonacci numbers. Access to these buffers must be properly synchronized using *mutexes* and *semaphores* to prevent race conditions and avoid active waiting. Producers must wait when their buffer is full, and consumers must wait when their buffer is empty.

To obtain the maximum grade, make sure to avoid active waiting and prevent starvation (HINT: use semaphores).

When the program terminates (i.e., when "exit" is entered), all threads should be properly woken up and exit cleanly. Make sure that the program ends correctly and does not leave any active threads running.

Universitat Pompeu Fabra Barcelona

Fibonacci number computation:

```
int fib(int n){
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    int fib_i_1 = 1;
    int fib_i_2 = 0;
    int fib_i = 1;
    for (int i = 1; i < n; ++i){
        int aux = fib_i_1 + fib_i_2;
        fib_i_2 = fib_i_1;
        fib_i_1 = fib_i;
        fib_i = aux;
    }
    return fib_i;
}
```

Delivery Instructions:

For this laboratory we will again use the platform gradescope (www.gradescope.com) to submit the exercise. Upload in practice 5 all documents associated with the practice as usual:

- Submit the following files using the same specified naming conventions below:
  - producer_consumer_threads.c
  - integrants.txt include all participants of the group, separated by lines, in the format *NIA Name Lastname1 LastName2 , and the practice group number.*
- Ensure each program compiles successfully and adheres to the CLI argument structure provided. Include the instruction you used for compiling.
- Take a screenshot showing your program successfully compiled without errors.
- Submit the programs as a compressed file named through Gradescope `practice5_<student1_student2>.zip`, replacing `<student1>` and `<student2>` with your full names (first and last names). Along with the compressed file, include screenshots showing the program successfully compiled and implemented without errors.
- Only 1 submission per pair is required. You need to indicate, when doing the submission in gradescope, the members of the group.
- Ensure that all programs are written in the C programming language and follow the provided naming conventions.
- Allowed function calls: open, close, read, write, dup2, fork, wait, waitpid, pipe, getpid, getppid, printf, scanf, malloc, free, signal, kill, and these seen in class for thread management and synchronization. You are not allowed to use **any other functions** that directly or indirectly use a system call (i.e. fwrite).