UNIVERSIDADE DE COIMBRA
UNIVERSITY OF COIMBRA

DEI
DEPARTMENT OF INFORMATICS ENGINEERING

Operating Systems [2024-2025]

Assignment 06 – Threads and synchronization II (condition variables)

Introduction

Thread programming causes substantial synchronization hazards due to the shared environment where threads execute. Threads share the process (to which they belong) address space, global variables, file descriptors, child processes, signals, signal handlers and accounting information.

The *pthread* library has locks and condition variables to help with the synchronization. Condition variables allow the programmer to control the execution according to one or multiple conditions, being able to block the execution of a thread while a condition is not met. Blocked threads are notified whenever the value associated to a variable is modified.

Objectives

Students concluding this work successfully should be able to:

- Create, destroy and execute multiple threads
- Use condition variables

Support Material

- K. A. Robbins, S. Robbins, “Unix Systems Programming: Communication, Concurrency, and Threads”, Prentice Hall:
  - Chapter 12 – POSIX Threads
  - Chapter 13 – Thread Synchronization
  - Chapter 14 – Critical Sections and Semaphores
- “Programming in C - UNIX System Calls and Subroutines using C” (http://www.cs.cf.ac.uk/Dave/C/CE.html):
  - Threads: Basic Theory and Libraries
  - Further Threads Programming: Synchronization

Exercises

Note: Only some of the exercises provided in this assignment will be done during the practical classes. The extra exercises should be done by the student as homework and any questions about them should be clarified with the teacher.

1. Count monitor

Create a program with NUM_WORKER_THREADS worker threads and one monitor thread. Each worker thread increases a common counter a random number of times (between 2 and 5), sleeping for 1 second between each change, and then exits. A monitor thread is waiting for all worker threads to finish the changes, so that it can print the final counter value. Before leaving, each worker increases the number of exited threads and notifies the monitor thread. If the number of threads that have left equals the number of worker threads created, the monitor thread prints the value of the common counter and leaves.

A main thread creates all threads. After both workers and monitor threads finish, it cleans the resources used and leaves. Use a condition variable to allow the monitor thread to know when to leave.

Complete the code in *count.c*. The *count* executable is also included.

2. Shop

The program to be implemented will simulate a shop with several cashiers and one foreman. The shop employees sell items for €10 and put the money in a shared safe. The foreman controls the amount of money in the safe and, when it reaches a predefined limit (CASH_LIMIT), he removes the same amount (CASH_LIMIT) of money from the safe, for security reasons. Each sell lasts for 1 second. The foreman only checks the safe after a sale has been made.

Use threads to represent each of employees and the foreman, and condition variables to control the execution of the program. The main thread should create all the employees and foreman threads. The program ends after the employees make NSALES.

Complete the code in *shop.c* using condition variables. The *shop* executable is also included.

3. Producer/Consumer

In a Producer/Consumer problem there are one or more entities producing some data, which is later consumed by other entities. Fig. **1** depicts the Producer/Consumer to implement in this exercise, using threads and condition variables.

The program will have N producer threads and M consumer threads. In between producers and consumers, a circular array will keep all the data previously written (produced) by the producers that was not yet read (consumed) by the consumers. When a consumer thread consumes a piece of data, the data is removed from the array and will no longer be available to other consumer threads.

To control this system, two condition variables must be created to control when the buffer is full and when it is empty. Producers can only write when the value of `empty` is higher than 0. Consumers can only read when the value of `full` is higher than 0. Together with the two condition variables, two locks (mutexes) must be used, respectively the `lock_full` and the `lock_empty`.

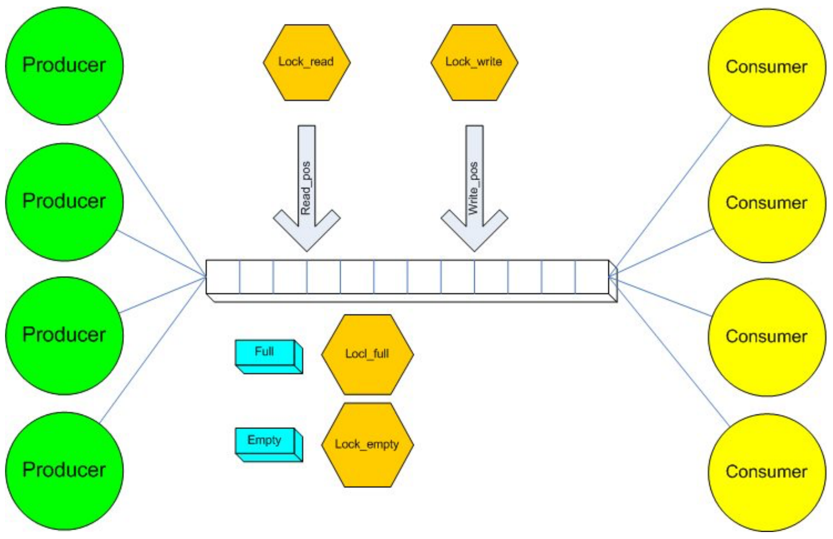

Fig. 1 – Producer/Consumer to implement

In order to control the writing and read positions in the array, two global variables, shared among producers and consumers, will be created. The `read_pos` and `write_pos` variables indicate to consumers the next position to read, and to producers the next position where to write. Any access to each of these variables must be done in mutual exclusion, i.e, only one thread can access these variables at the same time. To control these accesses two mutexes must be used, the `lock_read` and `lock_write`. Variables `read_pos` and `write_pos` must be incremented respectively by consumer threads and producer threads, after retrieving each of their values.

Only `MAX_PRODUCED_ITEMS` are produced and consumed. Two variables named `produced_items` and `consumed_items` will control this condition in each existing thread. There is no needed to further locks beside the ones stated before. When all threads verify the exiting condition, the program will terminate.

Complete the code given in *prodcon.c* . The *prodcon* executable is also included.

1) Initialize condition variables and mutexes in #Place1

2) Destroy condition variables and mutexes in #Place2

Places #3 and #4 synchronize producer threads and verify the exiting condition. Do not forget to update variables `empty`, `full` and `produced_items`.

Places #5 and #6 synchronize consumer threads and verify the exiting condition. Do not forget to update variables `empty`, `full` and `consumed_items`.