Lab 4: Synchronizing Threads

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2022

1 Instructions

This lab is not graded. The problems are to be solved on machine.

2 The Peruvian and Bolivian trains (follow-up)

This exercise is about the famous Peruvian and Bolivian train problem discussed during the previous session.

We recall that the problem can be summarized as follows. At the border, two railroad lines, one for Peruvian trains and one for Bolivian trains, share a section of track. To allow both Peruvian and Bolivian trains to use this shared section, while ensuring the safety of the passengers, some synchronization mechanisms have to be introduced at the entrance of the shared track.

The provided code `peru_and_bolivia.c` aims at simulated the problem. It creates 2 threads, one corresponding to a Peruvian train and one corresponding to a Bolivian train. To show that it is currently using the shared section of track, a train will periodically print the first letter of its home country on the standard output. When it is done with the shared section, it inserts a new line. *This implies that in a safe implementation there should never be letters of both countries on the same line.*

In this exercise, you are asked to modify `peru_and_bolivia.c` (and especially, functions `enter_shared_track()` and `exit_shared_track()`) to make the solution *safe*. Create a copy of the initial code to answer each question.

Question 2.1: *Implement a* safe *solution to the problem using the synchronization primitives introduced during Lecture 11.*

Question 2.2: *Propose another implementation. If you used semaphores to answer previous question, propose a solution without semaphores. If you did not, it is time to do so. Here you are still only asked to ensure safety.*

Question 2.3: *Based on your observation of the behavior of the proposed solution (and/or based on your understanding of the semantic of the synchronization primitives), point out the main problem of the solutions implemented in the two previous questions.*

Note that the problem raised in previous question is not easy to solve. We will study similar problems in more details during next lab.

Question 2.4: (bonus) (Come back to it at the end of the lab)*: Try to provide a solution to the problem raised in the previous question.*

3 Passing objects from one thread to another

In this exercise, we are considering two threads: one thread generates objects and *passes them* to another one that will process/use them. To make the problem simple, we consider that the generated objects are integers with values between 0 and 9, and that processing an object is simply displaying its value on the standard output. To allow passing objects from one thread to another, we will use a fixed-size buffer.

Question 3.5: *To which well-known synchronization pattern does this problem correspond?*

The file `passing_objects.c` is a partial implementation of the problem. Functions `pushing()` (called by the thread generating objects to insert them in the buffer) and `fetching()` (called by the user thread to fetch objects from the buffer) remain to be implemented.

Question 3.6: *Complete `passing_objects.c`. You should implement a solution that does not use semaphores. As for the previous question, work on a copy of the provided code.*

Question 3.7: *Complete `passing_objects.c`, this time using semaphores.*

Modify `passing_objects.c` to have multiple threads generating and using objects.

Question 3.8: *Are your previous solutions still correct with more than two threads? Explain.*

4 Memory allocation and multithreading

In a multithreaded process, what happens if a thread calls `malloc` while another thread is already executing `malloc` (or `free`)? Briefly explain the potential risks through an example (assuming that the heap is implemented using a linked list of free blocks).
It is also important to understand whether these potential risks are internally managed by the implementation of the allocator or must be addressed by application programmers. To find out the answer, look at the man page of `malloc`[1].

[1] See `https://man7.org/linux/man-pages/man3/malloc.3.html`.

5 Barriers

Barriers are another thread synchronization mechanism. A call to `barrier()` blocks the calling thread until all other threads have also called `barrier()`.

Question 5.9: *Complete file `barrier.c` and implement the barrier using condition variables. Note that we will assume that the main thread is not involved in the synchronization: the barrier only has to synchronize `NB_THREADS` threads.*
Important: we will also assume that the barrier function can only be called once.

Note that the POSIX interface already provides a barrier synchronization primitive (see `man pthread_barrier_wait`).

Question 5.10: *What is the main difference between the barrier that you have implemented and the POSIX barrier?*

Question 5.11: (bonus) *What is the major problem to be handled if one wants to be able to use the same barrier multiple times (*reusable barrier*)?*

Question 5.12: (bonus) *Implement a reusable barrier.*