Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2023-2024)

Guiões das aulas práticas

Quiz #IPC/03 Threads, mutexes, and condition variables

Summary

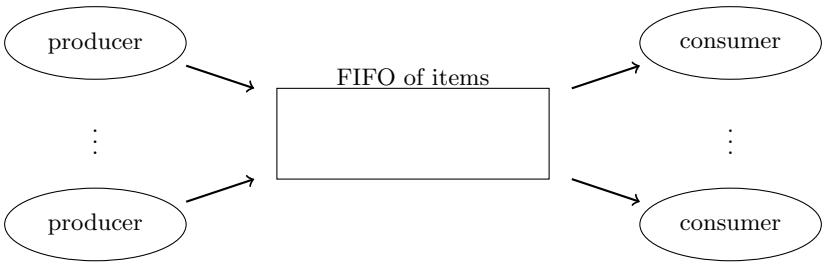
Understanding and dealing with concurrency using threads.

Using mutexes and condition variables to control access to a shared data structure, using the `pthread` library.

Previous note

In the code provided, the `pthread` library is not used directly. Instead, equivalent functions provided by the `thread.{h,cpp}` library are used. The functions in this library deal internally with error situations, either aborting execution or throwing exceptions, thus releasing the programmer of doing so. This library will be available during the practical exams.

Question 1 Implementing a bounded-buffer application, using a shared FIFO and semaphores.



Directory `bounded_buffer` provides an example of a simple producer-consumer application, where interaction is accomplished through a buffer with bounded capacity. The application relies on a FIFO to store the items of information generated by the producers, that can be afterwards retrieved by the consumers. Each item of information is composed of 3 integer

values, one used to store the id of the producer and the other two general purpose. Directory `bounded-buffer` contains the support source code for this exercise.

(a) **Understanding the fifo data type definition**

File `fifo.h` defines a `FIFO` data type and the signature of some manipulating functions.

- Analyse it and try to understand the purpose of the different fields. Note that thare are fields that are only used when synchronozation is introduced.

(b) **A first implementation of the fifo**

File `fifo-unsafe.cpp` implements a first version of the fifo manipulating functions.

- Analyse it and try to understand the implementation of the different functions.

(c) **Understanding how the concurrency is launched**

File `main.cpp` implements the main program, which launches child threads to execute the producer and consumer procedures.. Analyse it.

- Try to understand how shared fifo is created and used. Compare it with the process-based implementtion.
- Try to understand how thread creation is done.
- Try to understand how the main code waits for the child threads to finish.

(d) **See race conditions showing up**

Generate the *unsafe* version of the program (`make bb-unsafe`), execute it (`./bb-unsafe`) and analyse the results. Race conditions appear in red color. If they do not show up, use option `-d` of the main program to add some delay (a value of 100 should be enough).

- Point out an execution scenario that result in a race condition.
- Why doesn't the program end? You can press CONTROL-C to abort the execution.
- Look again at the code of the *unsafe* version, `fifo-unsafe.cpp`, and try to understand why race conditions can appear. What should be done to solve the problem?

(e) **Understanding the safe implementation of the fifo**

Generate the *safe* version of the program (`make bb-safe`), execute it (`./bb-safe`) and analyse the results. Race conditions should no longer appear.

- Look at the code of the *safe* version, `fifo-safe`, analyse it and try to understand how the mutex and conditions variables are initializaed and used to implement the safe version of the fifo.
- What is the purpose of the `access` mutex?
- What is the purpose of the `notFull` and `notEmpty` conditions variables?

(f) **Training exercise**

In the *safe* version, the program still does not terminate. Imagine a form of clean termination and implement it. One possibility is the insertion, by the main process, after all producers' termination, of dummy items, understood by consumers as exit notifications.

Question 2 Implementing an up-down counter application, using a shared integer variable.

The idea is to implement a concurrent program, involving main thread and a child thread, that, in collaboration, first increment and then decrement a counter shared between both threads. The conjugate behaviour should be the printing in the terminal of values from N_1 to N_2, follow by values from N_2-1 to 1, where N_1 and N_2 are values read from the terminal.

(a) The main thread (main function) should:

- ask the user for a value N_1 between 1 and 9, validating the value read;
- create an integer variable in static or dynamic memory and start it at N_1;
- launch a child thread, whose functionality in given below;
- wait until the child thread terminate;
- decrement the value in the shared variable until it reaches 1, printing its value at every iteration;
- terminate;

(b) The child thread should:

- ask the user for a value N_2 between 10 and 20, validating the value read;
- increment the value in the shared variable until it reaches N_2, printing its value at every iteration;
- terminate.

Question 3 Implementing a decrementer application, using a shared integer variable.

The idea is to implement a concurrent program, involving two child threads, that, in collaboration, decrement a counter in shared memory. The conjugate behaviour should be that the shared variable is alternately decremented by the two child threads.

(a) The main thread (main function) should:

- ask the user for a value between 10 and 20, validating the value inserted;
- create an integer variable in static or dynamic memory and start it at the value read;
- create and initialize the mutex and condition variables required to synchronize the activity of the two child processes;
- launch two child threads, whose functionality in given below;
- wait until both child threads terminate;
- release the resoures and terminate.

(b) Each child thread should:

- wait until it is its turn to decrement;
- terminate if the value in the shared variable is 1.
- decrement it, if not;
- print the value saying who made the decrement (PID);
- terminate if value is 1; otherwise repeat from top.