Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2024-2025)

Guiões das aulas práticas

script #04 Threads, mutexes, and condition variables

Summary

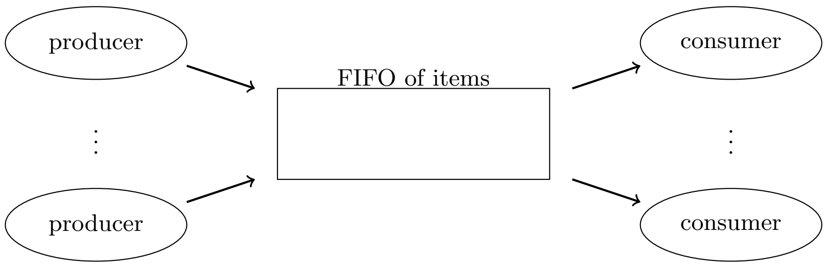
Understanding and dealing with concurrency using threads.

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 threads and a monitor.



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 it.

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

File `fifo-unsafe.cpp` implements a first version of the fifo manipulating functions, that does not take into consideration thread/concurreny safeness.

- 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 life cycle procedures. Analyse it.

- Try to understand how the shared data structure is created and used. Compare it with the process-based implementation.
- 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**

As said before, every item is composed of 3 values. When they are generated (by a producer), fields `v1` and `v2` are made equal and, in addition, they contains the `id` in their value (`v1 = v2 = id * 1000000 + i`). So, if everything goes alright, an item retrieved by a consumer must meet these restrictions. By default, a consumer only prints (in red color) an item if it does meet the restrictions, meaning there was a **race condition**.

- Generate the *unsafe* version of the program (`make bb-unsafe`), execute it (`./bb-unsafe`) and analyse the results. Hopefully, a print in red appears. If not, execute it again (you can press CONTROL-C to abort the previous execution). It may take several runs for the effect to show up. It may even be necessary to change the number of items produced per producer. You can execute `./bb-unsafe -h` do see the possible command line options.
- Point out an execution scenario that might result in a race condition.
- Why doesn't the program end?
- 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-mon-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 1**

In the *safe* version, the program still does not terminate. A simple form, although not entirely correct, of doing it is to kill the consumers after all the producers have terminated, and some time has passed. Alter the `main.cpp` program to accomplish it.

(g) **Training exercise 2**

Sending a kill to the consumer threads may not allow them to complete their activity. Imagine a form of clean termination and implement it. One possibility is the insertion, by the main thread, 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 the 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:

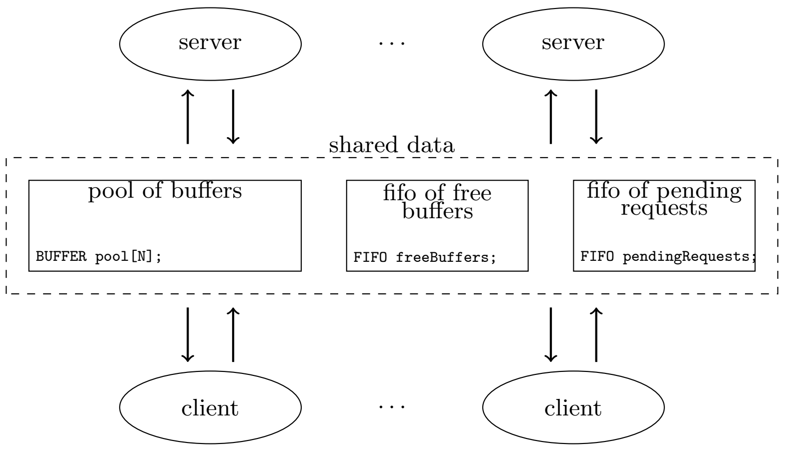
- ask the user for a value between 10 and 20, validating the value inserted;
- create an integer variable in static or dynamic memory and start it at the value read;
- create and initialize the mutex and condition variables required to synchronize the activity of two child threads;
- launch two child 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.

Question 4 Designing and implementing a simple client-server application

The figure below represents a simplified representation of a client-server concurrent system based on shared memory. The supporting (shared) data structure consists of a pool of N buffers of communication, individually identified by a number (between 0 and $N-1$), and two fifos, one of ids of buffers available and one of ids of buffers with pending orders. The same buffer is used for a client to place a request and the server to place the response to that request.



On the client side, interaction with the server takes place according to the following pseudo-code:

```
id = getFreeBuffer();           /* take a buffer id out of fifo of free buffers */
putRequestData(data, id);       /* put request data on buffer */
submitRequest(id);              /* add buffer id to fifo of pending requests */
waitForResponse(id);            /* wait (blocked) until the response is available */
resp = getResponseData(id);     /* take response out of buffer */
releaseBuffer(id);              /* buffer is free, so add its id to fifo of free buffers */
```

On the server side, the interaction is described by the pseudo-code:

```
id = getPendingRequest();       /* take a buffer id out of fifo of pending requests */
req = getRequestData(id);       /* take the request */
resp = produceResponse(req);    /* produce a response */
putResponseData(resp, id);      /* put response data on buffer */
notifyClient(id);               /* so client is waked up */
```

This is a double producer-consumer system, requiring three types of synchronization points:

- the server must block while the fifo of pending requests is empty;
- a client must block while the fifo of free buffers is empty;
- a client must block while the response to its request is not available in the buffer.

Note that in the last case there is a synchronization point per buffer. Note also that, as long as the fifos' capacities are at least the pool capacity, there is no need for a fifo full synchronization point.

Finally, consider that the purpose of the server is to process a sentence (string) to compute some statistics, specifically the number of characters, the number of digits and the number of letters.

(a) Using the *safe* implementation of the fifo, used in the previous exercice, as a guideline, design and implement a safe solution to the data structure and its manipulation functions.

(b) Implement the server thread, assuming there is only one.

(c) Implement the client thread.

(d) Does your solution work if there are more than one server?