Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2022-2023)

Guiões das aulas práticas

Quiz #IPC/04 A simple client-server concurrent application

Summary

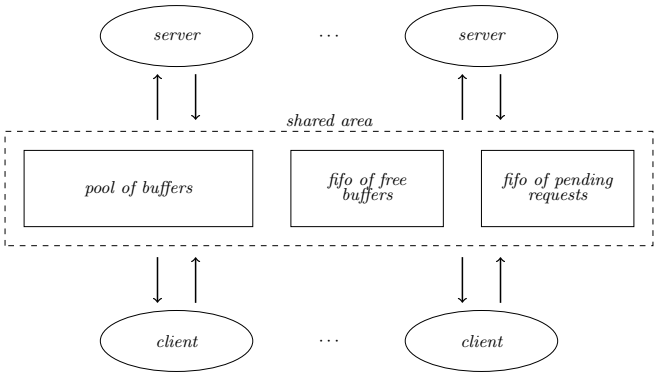
Developing a client-server concurrent application based on shared memory.

Previous note

Instead of directly using the `pthread` library and/or the Sys V or POSIX system calls, you can use convenient wrappers provided with this code. They correspond to libraries `thread.{h,cpp}`, to be used in thread-based approaches, and `process.{h,cpp}`, to be used in process-based approaches. The functions in these libraries deal internally with error situations, either aborting execution or throwing exceptions, thus releasing the programmer of doing so. These libraries will be available during the practical exams.

Question 1 *Designing and implementing a simple client-server application*

The figure below represents a simplified representation of a client-server concurrent application 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. Every buffer has room 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:

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

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

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

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

- *a 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) *The code provided implements the functionality of the application, but all code related to concurrency is missing.*

- *The role of a server entity is concentrated in function* `server`*, which is an infinite loop of* `processRequest`*s.*
- *Function* `processRequest` *implements the pseudo-code from the server side.*
- *The role of a client entity is concentrated in function* `client`*, which call for a service a given number of times.*
- *Function* `callService` *implements the pseudo-code from the client side.*
- *Module* `sos.{h,cpp}` *should provide all the required functionality and is only partially implemented.*
- *Comments with the word TODO represents points in the code where action is or may be required.*

(b) *Go through the code (*`main.cpp`*,* `sos.{h,cpp}`*) and try to understand it.*

(c) *Complete the implementation using threads, mutexes and condition variables.*

(d) *Complete the implementation using processes and semaphores.*