Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2019/2020)

Guiões das aulas práticas

Quiz #IPC/01 — Threads, mutexes, and condition variables

Summary

Understanding and dealing with concurrency using threads.
Programming using the `pthread` library.

Question 1 *Understanding race conditions in the access to a shared data structure.*

(a) Directory ***`incrementer`*** *provides an example of a simple data structure used to illustrate race conditions in the access to shared data by several concurrent threads. The data shared is a single integer value, which is incremented by the different threads. Each thread makes a local copy of the shared value, takes some time (delay) to simulate a complex operation on the value and copies the incremented value back to the shared variable. Three different operations are possible on the variable: set, get and increment its value.*

(b) Generate the unsafe version (`make incrementer_unsafe`), execute it and analyse the results.

- *If N threads increment the variable M times each, why is the final value different from $N \times M$?*
- *Why is the final value different between executions?*
- *Macros `UPDATE_TIME` and `OTHER_TIME` represent the times taken by the update operation and by other work. Change the value of `OTHER_TIME` to 1000 and verify if it affects the final value. Why?*

(c) Module ***`inc_mod_unsafe`*** *implements the unsafe version of the operations. Analyse the code and try to understand why it is unsafe.*

- *What should be done to solve the problem?*

(d) Generate the safe version (`make incrementer_safe`), execute it and analyse the results.

(e) Module ***`inc_mod_safe`*** *implements the safe version of the operations. Analyse the code and try to understand why it is safe.*

Question 2 *Implementing a monitor of the Lampson/Redell type.*

(a) Directory `bounded_buffer` *provides an example of a simple producer-consumer application, where interaction is accomplished through a buffer with limited 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 a pair of integer values, one representing the id of the producer and the other a value produced. For the purpose of easily identify race conditions, the two least significant decimal digits of every value contains the id of its producer. Thus the id appears twice in each item of information. The number of producers are limited to 100.*

There are 3 different implementations for the fifo: `fifo_unsafe`, `fifo_bwsafe`, *and* `fifo_safe`.

(b) Generate the unsafe *version (*`make bounded_buffer_unsafe`*), execute it and analyse the results. Race conditions appear in red color.*

(c) Look at the code of the unsafe *version,* `fifo_unsafe.cpp`*, analyse it, and try to understand why it is unsafe.*

- *What should be done to solve the problem?*

(d) Generate the bwsafe *version (*`make bounded_buffer_bwsafe`*), execute it and analyse the results.*

(e) Look at the code of the bwsafe *version,* `fifo_bwsafe.cpp`*, analyse it, and try to understand why it is safe.*

- *However, there is still a problem:* busy waiting. *Can you identify it?*

(f) Generate the safe *version (*`make bounded_buffer_safe`*), execute it and analyse the results.*

(g) Look at the code of the safe *version,* `fifo_safe.cpp`*, analyse it, and try to understand why it is safe and busy waiting free.*

- *Safeness is obtained by the use of condition variables. Try to understand how they are used.*

Question 3 *Designing and implementing a simple client-server application*

(a) Consider a simple client-server system, with a single server and two or more clients. The server consumes requests and produces responses to the requests. The clients produce requests and consume the corresponding responses.

A solution to this system can be implemented using a pool (array) of service slots and two fifos of slot ids, one representing the free slots and the other the pending requests. A client calls for service by:

1. *getting a slot id from the fifo of free slots;*
2. *puting its request in the slot;*
3. *inserting the slot id in the fifo of pending requests;*
4. *waiting until the response is available;*
5. *retrieving the response;*
6. *and freeing the slot used.*

Cyclically, the server:

1. *retrieve an id (representing a pending request) from the fifo of pending requets;*
2. *process the request in the corresponding slot;*
3. *put the response in the slot;*
4. *and notify the client.*

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 slots is empty;*
- *a client must block while the response to its request is not available in the slot.*

Note that in the last case there is a synchronization point per slot. Note also that, as long as the fifoas' 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.

(b) Using the safe *implementation of the fifo, used in the previous exercice, as a guideline, design and implement a solution to the data structure and its manipulation functions. Consider, for example, the following two main functions:*

```
void callService(ServiceRequest & req, ServiceResponse & res);
void processService();
```

The former is called by a client when it wants to be served; the latter is called by the server, in a cyclic way. Apart from the fifos, you will also need a pool of slots, each one containing the request data structure, the response data structure, and the support for synchronization (is this case, probably, a boolean and a condition variable). As auxiliary functions one can propose the following:

```
// insert an id into a fifo
void insert(FIFO * fifo, uint32_t id);

// retrieve an id from a fifo, blocking if necessary
uint32_t retrieve(FIFO * fifo);

// signal client that the response is available
void signalResponseIsAvailable(uint32_t id);

// block client until its response is available
void waitForResponse(uint32_t id);
```

(c) *Implement the server thread.*

(d) *Implement the client thread.*

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