Sistemas de Operação

(Ano letivo de 2018/2019)

Guiões das aulas práticas

Quiz #IPC/02 — Processes, shared memory, and semaphores

Summary

Understanding and dealing with concurrency using shared memory.
Using semaphores to control access to a shared data structure, by different processes.

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 processes. The data shared is a single pair of integer variables, which are incremented by the different processes. Three different operations are possible on the variables: set, get and increment their values. The increment is done in such a way to promote the occurrence of race conditions in one of the variables.*

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

- *If N processes increment both variables M times each, why can the final values be different from $N \times M$?*
- *Why can the two variables have different values?*
- *Why are the final value different between executions?*
- *Macros* `INC_TIME` *and* `OTHER_TIME` *represent the times taken by the increment operation and by other work. Change their values and understand what happens. Why?*

(c) Look at the code of the unsafe *version,* `inc_mod_unsafe`*, and analyse it.*

- *Try to understand how shared memory is used.*
- *Try to understand why race conditions can appear.*
- *What should be done to solve the problem?*

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

(e) Look at the code of the safe *version,* `inc_mod_safe`*, and analyse it.*

- *Try to understand how semaphores are used to avoid race conditions, thus implementing mutual exclusion.*

(f) The current implementation used the System V IPC system calls (see `man ipc` *and* `man svipc`*). Reimplement the safe version of the given code using the POSIX versions of shared memory and semaphores resources (see* `man shm_overview` *and* `man sem_overview`*).*

Question 2 *Implementing producer-consumer application, using a shared FIFO and semaphores.*

(a) Directory `prodcon` *provides an example of a simple producer-consumer application. 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 is the id of its producer. Thus the number of producers are limited to 100.*

There are 2 different implementations for the fifo: `fifo_unsafe`*, and* `fifo_safe`*.*

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

If execution does not terminate normally and you have to force it (for instance, pressing CRTL+c), you may need to remove the shared memory and/or semaphore resource afterwards. To do that, use command `ipcrm -M 0x1111 -S 0x1111`*. We can use command* `ipcs` *to see IPC resources in use.*

(c) Look at the code of the unsafe *version,* `fifo_unsafe`*, and analyse it.*

- *Try to understand how shared memory is used.*
- *Try to understand why race conditions can appear.*
- *What should be done to solve the problem?*

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

(e) Look at the code of the safe *version,* `fifo_safe`*, and analyse it.*

- *Try to understand how semaphores are used to avoid both race conditions and busy waiting.*

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 item slots and a fifo of items' ids. A client sends a request to the server, choosing an empty slot, puting its request there, inserting the slot's id in the fifo, and waiting for the response. The server retrieves requests (actually slot ids) from the fifo, process them, put the responses in the slot, and notify the client.

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

- *the server must block while the fifo is empty;*
- *A client must block while the fifo is full;*
- *A clients must block while the response to its request is not in the slot.*

Finally, consider that the purpose of the server is to convert a sentence (string) to upper case.

(b) Using the safe *implementation, used in the previous exercice, as a guide line, design and implement a solution to the data structure and its manipulation functions. Consider the following possible functions:*

```
HANDLE sendRequest(ITEM);
ITEM getResponse(HANDLE);

HANDLE retrieveRequest();
ITEM getRequest(HANDLE);
setResponse(HANDLE, ITEM);
```

If you prefer, you can use POSIX resources instead.

(c) Implement the server process.

(d) Implement the client process, adding a random delay between sending the request and receiving and displaying the response.

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