Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2019/2020)

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 `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 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 bounded_buffer_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 bounded_buffer_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 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. retrieves an id (representing a pending request) from the fifo of pending requets;

2. processes the request in the corresponding slot;

3. put the response in the slot;

4. and notifies 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, a semaphore). 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 process.

(d) Implement the client process.

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