Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2023-2024)

Guiões das aulas práticas

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

Summary

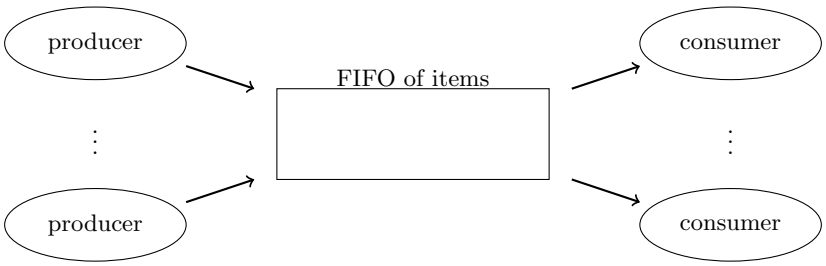
Understanding and dealing with concurrency using shared memory.

Using semaphores to control access to a shared data structure by different processes.

Previous note

In the code provided, system calls are not used directly. Instead, equivalent functions provided by the `process.{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 a shared FIFO and semaphores.



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 the purpose of the different fields.

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

File `fifo-unsafe.cpp` implements a first version of the fifo manipulating functions.

- 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 processes to execute the producer and consumer procedures.. Analyse it.

- Try to understand how shared memory is created and used.
- Try to understand how process creation is done.
- What is the purpose of the `exit(EXIT_SUCCESS)` after the calls to the producer and consumer procedures?
- Try to understand how the main code wait for the child processes to finish.

(d) **See race conditions showing up**

Generate the *unsafe* version of the program (`make bb-unsafe`), execute it (`./bb-unsafe`) and analyse the results. Race conditions appear in red color.

- Point out an execution scenario that result in a race condition.
- Why doesn't the program end? You can press CONTROL-C to abort the execution.
- The `FIFO` data structure is 1224 bytes long. Command `ipcs` shows which system V resources are in use. Execute it and identify the shared memory used by the program.
- Command `ipcrm -m` ≪`shmid`≫ can be used to release a block of shared memory. Used it to remove the shared memory used by the program. Re-execute the `ipcs` command to see the result.
- 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-safe`, analyse it and try to understand how semaphores are used to implement the safe version of the fifo.
- What is the purpose of the `LOCKER` semaphore?
- What is the purpose of the `SLOTS` and `ITEMS` semaphores?

(f) **Training exercise**

In the *safe* version, the program still does not terminate. Imagine a form of clean termination and implement it. One possibility is the insertion, by the main process, 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 a parent process and a child process, that, in collaboration, first increment and decrement a counter in shared memory. The conjugate behaviour should be the printing in the terminal of values from 1 to N, follow be values from N-1 to 1.

(a) The main program (parent process) should:

- *create an integer variable in shared memory and start it at 1;*
- *launch a child process, whose functionality in given below;*
- *wait until the child process terminate;*
- *decrement the value in the shared variable until it reaches 1, printing its value at every iteration;*
- *release the shared memory and terminate;*

(b) The child process should:

- *ask the user for a value between 10 and 20, validating the value inserted;*
- *increment the value in the shared variable until it reaches that value, 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 processes, 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 processes.

(a) The main program (parent process) should:

- *ask the user for a value between 10 and 20, validating the value inserted;*
- *create an integer variable in shared memory and start it with the value read;*
- *launch two child processes, whose functionality in given below;*
- *create the semaphores required to synchronize the activity of the two child processes;*
- *initialize the semaphores with the appropriate values;*
- *wait until both child processes terminate;*
- *release the shared memory and the semaphores and terminate.*

(b) Each child process 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.*