Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2024-2025)

Guiões das aulas práticas

script #02 IPC — Processes and signals

Summary

Understanding and dealing with processes and signals in Linux.

Changing the default response to signals.

Previous note

In the code provided, some 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.

Exercises

Exercise 1 Understanding the `fork` and `wait` system calls.

(a) ***Creation of processes in Unix/Linux***

- Read the `fork1.cpp` program, generate its binary version (`make fork1`), execute it and analyse the results, taking into consideration the behavior of system calls `fork`, `getpid` and `getppid` (see `man fork`, `man getpid` and `man getppid`).
- Every `printf` in `fork1.cpp` prints a single line of text and you can verify that in the program there are 3 occurrences of it. However, the number of lines of text that appear in the output (terminal screen) after executing the program is 5. Why?
- If you run the program several times, in some of them, the prompt of the bash may appear before one of the program lines. Why? You may need to change the delays for this to happen.

- In this program there are 2 processes involved. Which process is responsible for each line of text that appears in the output? Use the values of PID and PPID to get to the answer.
- Message *"Was I printed by the parent or by the child process? How can I know it?"* appears twice. Can you say which process print each one of them?

(b) ***Distinction between parent and child processes***

- Read the `fork2.cpp` program, generate its binary version (`make fork2`), execute it and analyse the results.
- System call `fork` returns a value which is assigned to variable `ret` in the program. Message *"Was I printed by the parent or by the child process? How can I know it?"* is prefixed with the return value of the `fork`. Can you now say which process (parent or child) print each one of them?
- Can you figure out how the return value of the `fork` can be used to condition how the program runs in the parent and child processes?
- Program `fork3.cpp` shows how the distinction can be done. Read it, generate its binary version (`make fork3`), execute it and analyse the results. Run it several times.

(c) ***A kind of synchronization between parent and child processes***

- Read the manual of system call `wait` (`man 2 wait`) to understand its function.
- Edit program `fork3.cpp` and uncomment the line that calls the `wait` system call, recompile and execute. What happens? Was it what you were waiting for?
- If you run the program several times, the order of messages may be different. You may need to change the delays for this to happen.

(d) ***Training exercise***

Implement a concurrent program, involving a parent process and a child process, that, in collaboration, print to the standard output values from 1 to 20, such that:

- values should be printed in ascending order;
- values from 1 to 10 should be printed by the child process;
- values from 11 to 20 should be printed by the parent process;

Continue in next page

Exercise 2 Understanding the `exec` family of functions.

Often, one wants to execute a different program in the child process. This can be done using system call `execve`. However, usually, an `exec` function that is layered on top of `execve` is used instead. As said in the manual of these functions (see, for example, `man execlp`), ≪the exec() family of functions replaces the current process image with a new process image≫.

(a) ***Executing a different program in the child process***

- Read `fork4.cpp` and `child.cpp` programs, generate their binary version (`make fork4 child`), execute `fork4` (only it) and analyse the results, taking into consideration the behavior of system call `execl`.
- The message associated to the `printf` instruction placed after the `execl` did not show up. Why? What kind of message should be placed after the `execl`?
- The first and second arguments of the call to `execl` are equal. Why?
- The two messages printed by the child process may be different. Play with the values of the arguments of the `usleep` function in `fork4.cpp` and `child.cpp`, in order to make it to happen. Then, try to answer to the following questions.
  - The difference is on the values of variable PPID. Why may the values of PPID be different?
  - When they are different, which process does the second value of PPID represent?
  - Why may the bash's prompt appear before the second message?

(b) ***Training exercise***

Implement a concurrent program, involving a parent process and a child process, that, in collaboration, do the following:

- The parent process should print a line of 40 equal (=) characters.
- Then the child process should exec command `ls -l`.
- Finally, the parent process should print another line of 40 equal (=) characters.

The global behaviour would be an `ls -l` with a top and a bottom delimiting line.

Continue in next page

Exercise 3 Understanding the handling of signals in Unix/Linux.

(a) ***Signals and the interruption of process execution***

- Read the `sig1.cpp` program, generate its binary version (`make sig1`), execute it and analyse the results.
- Repeat the execution and press CRTL+c at the middle of the counting. What happens?
- When you press CRTL+c, you are actually sending the `SIGINT` signal to the process, which as response terminates its execution. You can make the same through another terminal. Execute again the program, memorize the process ID (displayed before the counting) and execute the following command in another terminal:

  `kill -2 ≪PID≫`

  Alternatively, you can execute:

  `kill -INT ≪PID≫`

  You may increase the counting limit in order to give you time to execute the kill command.
- The `kill` system call (see `man 2 kill`) allows you to send a signal to a process. Develop a C/C++ program that accepts a number as command line argument, representing the PID of a process, and sends the `SIGINT` signal to that process.

(b) ***Changing the signal handling routine***

- Read the manual of system call `sigaction` (`man 2 sigaction`) to understand its purpose.
- Read the `sig2.cpp` program, generate its binary version (`make sig2`), execute it, press CRTL+c after a while and analyse the results.
- Execute again the program, memorize the process ID (displayed before the counting) and execute the following command in another terminal:

  `kill -15 ≪PID≫`

  Alternatively, you can execute:

  `kill -TERM ≪PID≫`
- Change the `sig2.cpp` program in order to associate the same handling routine to the `SIGTERM` signal.