Sistemas de Operação / Fundamentos de Sistemas Operativos

(Ano letivo de 2022-2023)

Guiões das aulas práticas

Quiz #IPC/01 — 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.

Exercice 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 5 occurrences of it. However, the number of lines of text that appear in the output (terminal screen) after executing the program is different. Why?*
- *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?*

- *Edit program* `fork1.cpp`*, change the second argument of the* `bwRandomDelay` *function to 2000, save the changes, recompile and try to answer again the previous questions. Run it more than once.*

(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?" *appears twice. 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` *may be used to distinguish between parent and child processes?*
- *Example* `fork3.cpp` *show how the distinction can be done. Read the* `fork3.cpp` *program, generate its binary version (*`make fork3`*), execute it and analyse the results.*

(c) **Associating a different program to 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` *(see* `man 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?*
- *Are the two messages printed by the child process equal? They should be by now. If not, increase the argument of the* `usleep` *function in* `fork4.cpp`*.*
- *Now, increase the value of the argument of the* `usleep` *function in* `child.cpp` *until the two messages printed by the child process are different (in the value of PPID). A value of 100000 should be enough. Also, the second message must appear after the prompt.*
  - *Why are the values of PPID different?*
  - *What does the second value of PPID represent?*
  - *Why does the prompt appear before the second message?*

(d) **A kind of synchronization approach between parent and child processes.**

- *Read the manual of system call* `wait` *(*`man 2 wait`*) to understand its purpose.*
- *Edit program* `fork4.cpp` *and uncomment the line that calls the* `wait` *system call, recompile and execute. What happens? Was it what you were waiting for?*

Exercice 2 *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. 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.*