Processes in Linux

Søren Hansen < *shan@iha.dk* >
V1.0

Introduction

In this exercise you will use the process creation primitives `fork()`, `wait()`, `_exit()` and `execv()` to create a sub processes.

Goal

Upon successful completion of this exercise, you will:

- Understand the process creation primitives.
- Understand the difference between processes and programs.
- Explain what happens when a process is spawned/forked.
- How shared memory works and is used
- How to synchronize between processes
- Pitfalls that are inherent to process spawning and shared memory

---

Exercise 1 Processes

Exercise 1.1 Simple process creation

Create a program that, when executed, spawns one child process.

- The *child* process should output "Hi! I am the child process", the value returned from `fork()` and the Process ID (PID) of the process (hint: use `getpid()`).
- The parent process should output "Hi! I am the parent process", the value returned from `fork()` and the PID of the process.
- The parent process should wait for the child to die. Note that you have to include the header file `sys/wait.h`. Use the man pages to get help on the function `wait()`.

Explain the value of the PIDs and the return values from `fork()` - how can one function return two different values?

Explain what happens when a process is forked:

- What happens to the memory from the parent passed to the child
- What is passed to the child process and what is not?
- Why must it be `_exit()` that is called from within the child and not `exit()`?

Exercise 1.2 Single program - multiple processes

Repeat the above but having 3 separate programs instead of one.

- Program 1 should spawn programs 2 & 3.
- Program 2 should output "tick" every second for 5 seconds.
- Program 3 should output "tock" every second for 8 seconds.

- The parent program should wait for both child processes to finish. Then it should output "Done."

Explain what happens when a process is spawned via the `exec()` function familily:

- What is passed to the "child" process and what is not?
- What should one be vary about when spawning processes in a security context (in relation to the previous question)?

Exercise 2 Shared Memory

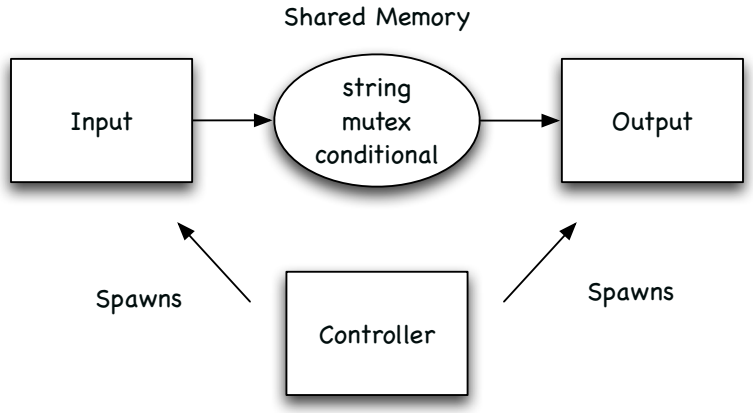

Figure 2.1: Simple diagram that illustrates data flow and processes

The basic idea is to create 3 programs as shown in figure 2.1. The main program is tasked with setting op shared memory, hereby also creating a mutex and a conditional variable in shared memory[1]. Having done this two child program must be spawned.

The first program, *Input*, must wait on input[2] from `stdin` place it in shared memory and signal the conditional variable.

The second program, *Output*, must wait on the aforementioned conditional variable and once signaled retrieve the read string from shared memory and print out via `stdout`.

Questions to answer:

- Describe the concept behind shared memory this includes which steps to go through and what to remember from a purely functional point of view.
- Under which circumstances would you consider shared memory a viable solution?
- Consider a scenario where two programs communicate via shared memory and program B has taken a mutex that program A also waits for. In this unfortunate moment program B crash whilst having acquired said mutex.

[1] This obviously includes releasing all resources when done!
[2] String of some sort

- What is the problem?
- How would you resolve it? (See pthread ROBUST - Google it)
- But using said approach what does this mean for what I may do in the critical section part, namely the part where I have acquired the mutex?