Assignment: Applied Fork and Exec

Guillermo Pérez Trabado *©2017*

Operating Systems

Dept. of Computer Architecture – University of Malaga

Writing useful programs with fork() and exec() syscalls

1. Object of this exercise

The purpose of this exercise is to learn how to write C code which performs the following functions:

- Launches another program as a child process.
- Prepares the environment that the child program will inherit with useful strategies like standard I/O channels redirection or pipes.
- Controls the execution of the child process and supervises the finishing status.

As these are important functions of any shell, this exercise will contribute to clarify how to write that part of the exercise of implementing your own shell program.

2. Problems to solve

1. Write a C program that launches a very simple command redirecting its input and output from/to files. The idea is to achieve the same result than writing in the command line of a shell the expression `command < filein >fileout`. A proposed example is to mimic this command:

   ```
   sort < /etc/services > ./output.txt
   ```

   To write such command you could write your own fork/exec schema or use the code skeleton given in this assignment. Pay attention to details:

   - To set the input (descriptor 0), output (descriptor 1) or error output (descriptor 2) to a given file, you have to open the file and then copy the returned descriptor to the given one using the *dup2()* system call. Be careful closing unneeded descriptors.

     ```
     int f=open(file,O_RDONLY)
     close(0);
     dup2(f,0);
     close(f);
     ```

   - Use the proper open modes for each file.
   - Don't modify original stdin/stdout descriptors in the parent to be able to read/write the terminal when needed.
   - Avoiding sharing unnecessary resources (close unneeded descriptors when possible).

2. Write a C program that launches two very simple commands connecting both of them through a pipe. The idea is to achieve the same result than writing in the command line of a shell the expression `command1 | command2`. A proposed example is to mimic this commands which read from the terminal cutting the first four characters of each line and then add a line number for each line:

   ```
   cut -c1-4 | nl
   ```

   To write such command you could write your own fork/exec schema or use the code skeleton given in this assignment. Pay attention to details:

   - The **unnamed** pipe is created with the *pipe()* syscall which takes as argument an array with two integers. The return value are two descriptor representing both sides of the pipe. Note that the created pipe can't be opened from other processes, so that you can only *pass the pipe* through **inheritance**. The pipe descriptors are copied with the *dup2()* syscall in the same way than normal file descriptors.
   - Preserve original stdin/stdout descriptor in the parent to be able to read/write the terminal.
   - Avoiding sharing unnecessary resources (close unneeded descriptors when possible). Process termination won't work normally if you keep open duplicates of the descriptors of the pipe in other processes.

3. Technical Note: Pipes and processes termination

The pipe is a communication mechanism that offers the same interface than files. However, reading/writing from/to a pipe means talking with another process and it introduces additional sceneries due to concurrency.

The most common is the case in which one of the two processes has to end for any reason: normal termination, conditions that imply ending the program or even programming errors like NULL pointers. The question is how to know about the termination of the other process and avoid waiting forever reading or writing the pipe?

1. **The producer (writer of the pipe) ends first.** Whenever a process is terminated, all its file descriptors are closed automatically, and so are pipes. By definition of pipes, when a process reads a pipe and the other side is closed, an EOF is returned immediately.
   So, if we control EOF condition while reading stdin, we can detect the termination of the other process.
2. **The consumer (reader of the pipe) ends first.** In this case, EOF is not useful as the EOF condition is only defined for reading and never for writing. Then we need a different mechanism to know about this condition. By definition of pipes, when a process writes a pipe with no other open descriptor, a SIGPIPE signal is sent to the writer process.
   As the default behavior of the SIGPIPE is to terminate the process, the writer process is *killed* by the operating system when it tries to write data that will never be consumed. If we want to end in a smart way, we can program the SIGPIPE signal to do cleaning work before ending our program.

Remark: These conditions are also valid to detect termination while using **TCP sockets**. You can **reuse the same** code to communicate both through pipes and TCP sockets.