Laboratory Assignment 4: Processes and threads

Table of contents

1 Objectives

In this laboratory we are going to solve several exercises oriented to reinforce our knowledge of the POSIX API for processes and threads, and how they affect to the handling of files.

The student is advised to create a directory for the laboratory with one subdirectory per exercise. The instructions assume that exercise N is done in a subdirectory named exercise*N* within the common directory for the laboratory.

The file files_p4.tar.gz contains a series of files that can be used as a starting point for the development of the exercises of this laboratory, as well as some makefiles that can be used for the compilation of the different projects.

2 Exercises

Exercise 1: Process creation and program execution

Develop a *run_commands* program that executes commands specified by the user, and waits for their termination. A code skeleton with a Makefile is provided, as well as two input files to check the correct operation of the program to be developed.

A test code `run_commands.c` is provided, which must be modified to develop the desired functionality. This program accepts a single argument where a command is specified. The program parses that command, and constructs an array of characters NULL-terminated (`argv` format), which is then printed to the screen, properly freeing the reserved memory with `malloc()`. The purpose of this program is to illustrate the use of the provided `parse_command()` function, which should be studied and reused in the implementation of the exercise.

The *run_commands* program to be developed will accept as parameter a set of options, which must be processed using the `getopt()` function, used in previous practices. The following are the options to be accepted by the program, which it is recommended to implement and test in the order in which they are described:

1. `-x <command>`: When the program is invoked with this option, a child process will be created to execute the command. To do this, the main program will invoke the function `launch_command()` (to be implemented), which will create a child process with `fork()`, and make it execute the command passed as parameter using `execvp()`. The function will return the PID of the child process (without waiting for it to finish its execution).

The main program will wait for the termination of the created child process. The `launch_command()` function will have the following prototype:

```
pid_t launch_command(char** argv);
```

2. `-s <file>`: This option will allow the user to indicate as argument the path of a file with the commands to be executed. This file will be interpreted by lines, taking each line as a command to be executed with the `launch_command()` function. The commands will be executed sequentially, waiting for one command to finish before executing the next one. Hint: use `fgets()` to read from the file by lines. See `man 3 fgets`.

3. `-b`: This program option will only take effect if it is passed together with the `-s` option. In this case, the commands in the file indicated by the `-s` option will be executed one after the other without waiting for the previous command to finish. Only when all the commands indicated in the file (each one by a child process) have been launched for execution will wait for all of them to finish. Likewise, each time one of the commands launched terminates, the program will print on the standard output the number of the command that has terminated, its PID and termination code – e.g., *@@ Command #3 terminated (pid: 11576, status: 0)*, as shown in the execution example. For this purpose, the program must maintain a data structure -for simplicity, an array of predefined maximum length- that allows to associate the PIDs of the children, with the number of each command in the launching order.

   Example:

```
# Testing -x switch
$ ./run_commands -x ls
Makefile run_commands run_commands.c run_commands.o test1 test2
$ ./run_commands -x "echo hello"
Hello

# Testing -s switch
$ ./run_commands -s test1
@@ Running command #0: echo hello
hello
@@ Command #0 terminated (pid: 1439, status: 0)
@@ Running command #1: sleep 2
@@ Command #1 terminated (pid: 1440, status: 0)
@@ Running command #2: ls -l
total 88
-rw-r--r--@ 1 user staff 267 Oct 20 11:34 Makefile
-rwxr-xr-x 1 user staff 9960 Oct 20 11:58 run_commands
-rw-r--r-- 1 user staff 4332 Oct 20 11:57 run_commands.c
-rw-r--r-- 1 user staff 8984 Oct 20 11:58 run_commands.o
-rw-r--r--@ 1 user staff 31 Oct 20 11:34 test1
-rw-r--r--@ 1 user staff 41 Oct 20 11:46 test2
@@ Command #2 terminated (pid: 1443, status: 0)
@@ Running command #3: false
@@ Command #3 terminated (pid: 1444, status: 256)

# Testing -bs switch
$ ./run_commands -b -s test2
@@ Running command #0: echo one
@@ Running command #1: sleep 6
@@ Running command #2: sleep 3
@@ Running command #3: echo two
one
two
@@ Running command #4: sleep 1@@ Command #3 terminated (pid: 1457, status: 0)
@@ Command #0 terminated (pid: 1454, status: 0)
@@ Command #4 terminated (pid: 1458, status: 0)
@@ Command #2 terminated (pid: 1456, status: 0)
```

```
@@ Command #1 terminated (pid: 1455, status: 0)
```

After you have finished developing the run_commands program, answer the following questions:

1. When using the `-x` option of the program, the command given as an argument is passed enclosed in double quotes in case it, in turn, accepts arguments, such as `ls -l`. What happens if the `-x` argument is not passed enclosed in quotes? Does the launch of the `ls -l` program work correctly if it is enclosed in single quotes instead of double quotes? Note: To see the differences try executing the following command: `echo $HOME`.
2. Is it possible to use `execlp()` instead of `execvp()` to execute the command in the `launch_command()` function ? If yes, indicate possible limitations arising from the use of `execlp()` in this context.
3. What happens when executing the command `echo hello > a.txt` with the *run_commands* program ? and `grep intn| run_commands.c`? In case the commands are not executed correctly indicate why. What happens when executing `./run_commands "/bin/bash -c 'echo hello > a.txt \'"`?

Exercise 2: Creating and passing parameters to threads

In this exercise we are going to use the *pthreads* library, so it will be necessary to compile and link with the `-pthread` option.

Write a program *threads.c* that creates a thread for each user, passing as input argument a pointer to a structure containing two fields: an integer, which will be the user number, and a character, which will indicate whether the user has priority (P) or not (N).

The program must create a struct variable for the argument of each thread using dynamic memory. Initialize this variable with the user number and its priority (the even ones will have priority and the odd ones will not), create the threads and wait for them to finish.

Each thread will copy its arguments into local variables, free up the dynamic memory reserved for them, find out their identifier and print a message with their identifier, user number and priority.

The student should check the manual pages of: `pthread_create`, `pthread_join`, `pthread_self`.

Try creating only one variable for the argument of all threads, giving it the corresponding value to each thread before the call to `pthread_create`. Explain what happens and what is the reason.

Exercise 3: Signal handling

In this exercise we are going to experiment with signals between processes. A process creates a child, waits for a signal from a timer and, upon receiving it, terminates the execution of the child.

The main program will receive as argument the executable that you want the child process to execute.

The parent process will create a child, which will change its executable with a call to `execvp`.

The parent will then set the `SIGALRM` signal handler to be a function that sends a `SIGKILL` signal to the child process and will set an alarm to send the signal after 5 seconds.

Before terminating, the parent will wait for the child to terminate and will check the cause of the child termination (normal termination or signal reception), printing a message on the screen.

The student should check the manual pages of: `sigaction`, `alarm`, `sigalarm`, `kill`, `wait`.

To check the correct operation of our program we can use as argument an executable that finishes in less than 5 seconds (like `ls` or `echo`) and one that does not finish until a signal arrives (like `xeyes`).

Once the program is running, modify the parent to ignore the `SIGINT` signal and check that it does so.

Exercise 4: File handling with multiple processes and threads

We intend to create a program that uses 10 processes (the original and 9 child processes) to write concurrently an "output.txt" file. The idea is that each process writes a character string with a decimal number repeated 5 times. So the initial process will write 5 zeros ("00000"), the first child process will write 5 ones ("11111"), the second 5 twos ("22222") and so on. Thus the content of the file at the end will be: 00000111112222233333444445555566666777778888899999

A first programmer with little experience in system programming proposes the following implementation (file *practice_2_5_initial.c*)

```c
int main(void)
{
    int fd1,fd2,i,pos;
    char c;
    char buffer[6];

    fd1 = open("output.txt", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    write(fd1, "00000", 5);
    for (i=1; i < 10; i++) {
        pos = lseek(fd1, 0, SEEK_CUR);
        if (fork() == 0) {
            /* Child */
            sprintf(buffer, "%d", i*11111);
            lseek(fd1, pos, SEEK_SET);
            write(fd1, buffer, 5);
            close(fd1);
            exit(0);
        } else {
            /* Parent */
            lseek(fd1, 5, SEEK_CUR);
        }
    }

    //wait for all children to finish
    while (wait(NULL) != -1);

    lseek(fd1, 0, SEEK_SET);
    printf("File contents are:\n");
    while (read(fd1, &c, 1) > 0)
        printf("%c", (char) c);
    printf("\n");
    close(fd1);
    exit(0);
}
```

After this implementation the programmer checks the operation, executing the program 10 times in a row in the hope that no races occur.

The result, on the programmer's machine is:

```
$ for i in $(seq 10); do ./practica_2_5_inicial ; done
File contents are:
000001111122222333335555566668888899999
File contents are:
000001111122222555556666668888899999
File contents are:
000001111122222333335555566666888899999
File contents are:
000001111122222444446666667777799999
File contents are:
000004444477777555556666668888899999
```

```
File contents are:
000002222244444555557777778888899999
File contents are:
00000111112222244444555557777778888899999
File contents are:
000001111122222555554444488887777799999
File contents are:
00000111112222244444555557777778888899999
File contents are:
000001111122222555554444488887777799999
```

It seems that the program has some errors, since races are produced and the result is incorrect in all cases.

Question A:

Solve the initial implementation, maintaining concurrent writing to the file. That is, the parent process will write the five initial zeros, the child one the five ones, etc, without the need to synchronize the processes. We don't want to impose an order in the execution of the processes.

Question B:

Propose a solution in which the parent writes its number between the writing of the children, so that the contents of the file at the end will be:

00000111110000022222000003333300000444440000055555000006666600000777770000088888000009999 9