LABORATORY REPORT

Lab 1 - Programmer's view - Creating Processes

Autors :

Maxime Heurtevent
Maxence Lecoq

Lab instructor :

Khoury Christian

- Prerequisites –

- PART I. Creating and Running a Process (1) –

A. Understanding fork( ), getpid( ) and getppid( ) functions

B. Creating and running a child process

- PART II. Creating and Running a Process (2) -

A. Understanding the functions of the exec family
B. Running a different application with exec

- PART III. Writing your own Shell -

A. Understanding the system( ) function
B. Implementing our system function using fork( ) and exec( )
C. Organizing the shell's tasks in a menu

- BONUS - Using Rust -

Prerequisites:

How to compile a program from the command line:

```
$ gcc program.c -o program
```

figure 1.1: Compile program command

How to execute a program from the command line:

```
$ ./program
```

figure 1.2: Execute program command

Creating and Running a Process (1):

Understanding fork( ), getpid( ) and getppid( ) functions:

What happens after a fork call? How are parents and children differentiated? After a fork system call is made, a new child process is created as a duplicate of the parent process. Here's what happens and how parent and child processes are differentiated

1. **Creation of Child Process:** When the fork system call is executed, the operating system creates a new process that is an exact copy of the parent process. This includes duplicating the code, data, file descriptors, and other attributes of the parent process.

2. **Return Value:** In the parent process, the fork system call returns the child's process ID (PID), which is a positive integer greater than zero. In the child process, it returns 0.

3. **Differentiation:**
    - In the parent process, you can differentiate it from the child process by checking if the return value of fork is greater than 0. This indicates that it's the parent process.
    - In the child process, you can differentiate it from the parent process by checking if the return value of fork is 0. This indicates that it's the child process.

Creating and Running a child process:

Write a small C program in which the parent process creates a child processand each displays a different message : I'm the parent vs I'm the child. Display the process id and the parent process id for every running process.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork(); // Create a child process

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // This code is executed by the child process
        printf("PID value: %d\n", pid);
        printf("I'm the child (PID: %d, Parent PID: %d)\n", getpid(), getppid());
    } else {
        // This code is executed by the parent process
        printf("PID value: %d\n", pid);
        printf("I'm the parent (PID: %d, Child PID: %d)\n", getpid(), pid);
    }

    return 0;
}
```

figure 1.3: Small C program create a child

Functions explanation:

- **fork( )** is used to create a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
- **getpid( )** returns the process ID (PID) of the calling process.
- **getppid( )** returns the process ID of the parent of the calling process.

Output:

```
PID value: 36556
I'm the parent (PID: 36552, Child PID: 36556)
PID value: 0
I'm the child (PID: 36556, Parent PID: 36552
```

figure 1.4: Output from C program create a child

As a proof of order execution, we can modify the parent process execution with adding a **sleep(3)**.

```
printf("PID value: %d\n", pid);
sleep(3);
printf("I'm the parent (PID: %d, Child PID: %d)\", getpid(), pid);
```

figure 1.5: sleep function

Output:

```
PID value: 36937
PID value: 0
I'm the child (PID: 36937, Parent PID: 36935)
I'm the parent (PID: 36935, Child PID: 36937)
```

figure 1.6: result of sleep function

First the parent process is executed, when it reaches the **sleep(3)** function, the child process executed in parallel will be executed before the second instruction in the parent process. After 3 seconds, the parent process will be executed again.

Is data shared between parent and child?

No, data is not shared between parent and child. Each process has its own address space. The parent process and child process have separate copies of the data. The child process gets a copy of all the memory segments of the parent process. The child process does not share the stack with the parent process. The child process gets a copy of the stack of the parent process. The child process does not share the heap with the parent process. The child process gets a copy of the heap of the parent process.

```
int main() {
    int i = 5;
    if (fork() == 0) {
        i++;
        printf("Child value: %d\n", i);
    } else {
        sleep(3);
        printf("Parent value: %d\n", i);
    }
    return 0;
}
```

figure 1.7: test of data shared between child and parent

Output:

```
Child value: 6
Parent value: 5
```

figure 1.8: result of the data shared test

Child is executed before for the same reason as the question above.

Is it possible to create more than one child process ? Show how using a simple program that creates 2 children for the 1st-level process (main parent) and a child for one of the 2nd-level processes (children).

```c
int main() {
    printf("1st-level parent process. PID: %d\n", getpid());

    pid_t child1_pid = fork(); // Create the first child

    if (child1_pid == 0) {
        printf("1st-level child process 1. PID: %d, Parent PID: %d\n", getpid(), getppid());

        pid_t grandchild_pid = fork(); // Create a child for the first child

        if (grandchild_pid == 0) {
            printf("2nd-level grandchild process. PID: %d, Parent PID: %d\n", getpid(), getppid());
        }
        return 0; // Exit the first child
    }

    pid_t child2_pid = fork(); // Create the second child

    if (child2_pid == 0) {
        printf("1st-level child process 2. PID: %d, Parent PID: %d\n", getpid(), getppid());
    }

    // Wait for all child processes to finish
    for (int i = 0; i < 2; i++) {
        wait(NULL);
    }

    return 0; // The main parent process exits
}
```

figure 1.9: Create more than 1 child program

wait( ) is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In our case, we wait for a child to terminate.

Output:

```
1st-level parent process. PID: 50154
1st-level child process 1. PID: 50157, Parent PID: 50154
1st-level child process 2. PID: 50158, Parent PID: 50154
2nd-level grandchild process. PID: 50159, Parent PID: 50157
```

figure 1.10: result of more than 1 child

Creating and Running a Process (2) :

Understanding the functions of the exec family

The **exec( )** family of functions replaces the current process image with a new process image.

Functions used:

- **execv( ):** provides an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer.
- **execvp( ):** Same as **execv( )** but the **execvp( )** function does not use the PATH environment variable to find the executable file, so the file name must contain the full path name.

Running a different application with exec

Executing a program using execv( )

```c
#include "stdio.h"
#include "unistd.h"
#include "sys/wait.h"

int main( ) {
    pid_t child_id;
    int status;

    child_id = fork( );

    if (child_id < 0) {
        perror("Fork failed");
    } else if (child_id == 0){
        char *argv[4] = {"mkdir", "-p", "sample-dir", NULL};
        execv("/bin/mkdir", argv);
        perror("execv");
    } else {
        while ((wait(&status)) > 0);

        char *argv[3] = {"touch", "sample-dir/sample-touch.txt", NULL};
        execv("/usr/bin/touch", argv);
        perror("execv");
    }
    return 0;
}
```

figure 2.1: Executing a program using execv()

This code will create a directory named **sample-dir** and wait until the directory is created. Then it will create a file named **sample-touch.txt** inside the directory.

Executing a program using execvp()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid;
    int status;

    child_pid = fork();

    if (child_pid == -1) {
        perror("Fork failed");
        return 1;
    }

    if (child_pid == 0) {
        // This code runs in the child process
        char *program = "/Applications/Figma.app/Contents/MacOS/figma"; // Specify the full path to
Figma
        char *args[] = {program, NULL};
        execvp(program, args);

        // If execvp() fails, it will reach this point
        perror("Execvp failed");
        return 1;
    } else {
        // This code runs in the parent process
        printf("Parent process. PID: %d\n", getpid());
        printf("Child process. PID: %d\n", child_pid);

        // Wait for the child process to finish
        wait(&status);
    }

    return 0;
}
```

figure 2.2: Executing a program using execvp()

```
$ gcc main.c -o main && ./main
Parent process. PID: 60051
Child process. PID: 60054
INFO  [url_config] starting with app url: https://www.figma.com
[fonts] read font_cache from disk.
[fonts] found 446 paths on disk
...
```

figure 2.3: Result of execvp() program

This code will open *Figma* application, and wait until the application is closed. It will print the parent process ID and the child process ID.

NB: The full path to Figma application may be different on your machine.

Is data shared by the parent and child processes and to what extent ? Explain

Shared between Parent and Child Processes:

1. Program Code: Initially, the child process shares the same program code (instructions) as the parent process. This means that both the parent and child start executing at the same point in the program.

2. Global Variables: Global variables in the program are shared between the parent and child processes. Any changes made to these variables in one process will be visible to the other process.

3. Open File Descriptors: If a file is opened by the parent process before forking, the child process inherits the open file descriptors. This allows both processes to read from or write to the same file.

4. Signal Handlers: Signal handlers installed by the parent process are inherited by the child process.

5. Environment Variables: Environment variables are shared, but processes can modify their own environment variables without affecting the other.

Not Shared between Parent and Child Processes:

1. Process ID (PID): Each process has a unique PID. The parent and child processes have different PIDs.

2.Stack: Each process has its own stack. Changes made to the stack in one process do not affect the other.

3. Heap: Processes have their own heap memory for dynamic memory allocation. Changes to the heap in one process do not affect the other.

4. File Descriptors: File descriptors created after forking are not shared between parent and child processes. Each process has its own set of file descriptors.

5. Register Values and Program Counter: The CPU register values (e.g., program counter) are independent in each process, so they can execute different instructions.

6. Resource Usage and Process State: Each process has its own set of resource usage statistics (e.g., CPU time, memory usage) and process state (e.g., process status, exit status).

7. Parent Process ID (PPID): The PPID of the child process is set to the PID of the parent process.

Explain what happens in the following program. What is the main difference with the previous version ?

```c
int main() {
    int status;
    int i = 5;
    if (fork() == 0){
        execlp("echo", "echo", "Hello World", NULL);
        perror("Exec failed");

        printf("%c\n", i); //is this line executed? why?
    } else {
        printf("Process ID: %d\n", getpid());

        wait(&status);
    }
    return 0;
}
```

figure 2.4: Execlp() program

Output:

```
Process ID: 62299
Hello World
```

figure 2.5: Result of execlp() program

execlp( ) replaces the current process image with a new process image.

So the first **printf( )** statement before the **execlp( )** call is not executed because the **execlp( )** function replaces the current process image with the "echo" command, and any code that follows it is effectively unreachable

Writing your own Shell:

Understanding the system( ) function

The **system( )** library function behaves as if it used **fork(2)** to create a child process that executed the shell command specified in command using **execl(3)** as follows:

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL);
```

figure 3.1: C line for execute shell command

system( ) returns after the command has been completed.

Implementing our system function using fork( ) and exec( )

Implement your own system function (call it « mySystem »)

```
int mySystem(const char *command){
    if (command == NULL || command[0] == '\0'){
        return -1;
    }

    int status;
    pid_t child_pid;

    child_pid = fork();

    //Error management
    if (child_pid < 0){
        perror("Fork failed");
        printf("Error Code: %d\n", errno);
    }

    if (child_pid == 0) {
        execl("bin/sh", "sh", "-c", command, NULL);
        perror("Exec failed");
        _exit(2);
    } else {
        waitpid(child_pid, &status, 0);

        if (WIFEXITED(status)) {
            return WEXITSTATUS(status);
        } else {
            return -4;
        }
    }
}
```

figure 3.2: Create mySystem function in C

In the code above, we can see that we return the same error codes as **system( )** function according to the documentation. If the function succeeds, it returns the exit status of the command executed.

- **waitpid( )** is the same as wait() but it allows to specify which child process to wait for.
- **WIFEXITED(status)** returns true if the child terminated normally, that is, by calling **exit(3)** or **_exit(2)**, or by returning from main().
- **WEXITSTATUS(status)** returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to **exit(3)** or **_exit(2)** or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.
- **errno** is a global variable that is set by system calls and some library functions in the event of an error to indicate what went wrong. It is defined in the header file errno.h.

Organizing the shell's tasks in a menu

Write a program that displays the following menu in a loop :

1. *run a program*
2. *kill a process (hint : lookup the kill manual)*
3. *list the files in the current folder (hint :lookup the ls manual)*
4. *quit*

Use the «mySystem » function to implement the different options of your menu (except for quit of course).

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "errno.h"

int mySystem(const char *command);

int main() {
    int choice = 0;
    char command[100];
    char buffer;

    while (choice != 5) {
        //Display menu
        printf("\n---Menu---\n");
        printf("1. Run program\n");
        printf("2. Kill a process\n");
        printf("3. List the file in the current folder\n");
        printf("4. Open application (On MacOS)\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        // Consommer le caractère de nouvelle ligne restant
        while ((buffer = getchar()) != '\n' && buffer != EOF);

        switch (choice) {
            case 1:
                printf("Enter the command name to run: ");
                scanf("%[^\n]", command);
                mySystem(command);
                break;
            case 2:
                printf("Enter the process ID to kill: ");
                int pid;
                scanf("%d", &pid);
                snprintf(command, sizeof(command), "kill %d", pid);
                mySystem(command);
                break;
            case 3:
                mySystem("ls");
                break;
            case 4:
                printf("Enter the application name to open: ");
                char app[100];
                scanf("%[^\n]", app);
                snprintf(command, sizeof(command), "/Applications/Figma.app/Contents/MacOS/%s",
app);
                mySystem(command);
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice, try again\n");
        }
    }
    return 0;
}
```

figure 3.3: Create menu in C

Output:

```
---Menu---
1. Run program
2. Kill a process
3. List the file in the current folder
4. Open application (On MacOS)
5. Exit
Enter your choice: 1
Enter the command name to run: ls
CMakeLists.txt          Lab - Processes.pdf     README.md               cmake-build-debug
main                    main.c
```

figure 3.4: Result of the part 3

scanf(%[^\n], command) is used to read a string with spaces until the condition defined after the **^**. In our case, we read a string until we reach a new line character.

```
while ((buffer = getchar()) != '\n' && buffer != EOF);
```

figure 3.5: Buffer line in C

This line is used to consume the newline character left in the buffer after the **scanf( )** call.

snprintf( ) is used to write formatted output to the string pointed to by str. It is similar to **printf( )** but it writes the output to a character string str rather than sending it to the screen.

Using Rust:

If you already have Rust installed, you can skip this section.

Installing Rust on MacOS or Linux:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

figure 4.1: Linux command to install Rust

Installing Rust on Windows: https://www.rust-lang.org/tools/install

Run the following command to check if Rust is installed correctly:

```
$ rustc --version
$ cargo --version
```

figure 4.2: Return the Rust and Cargo version

To execute a Rust program, you can run the following command:

```
rustc program.rs && ./program
```

figure 4.3: Compile and Execute a Rust file

Here the previous my_system function written in Rust:

```rust
use std::io::{self, Write};
use std::process::{Command, exit};

fn my_system(command: &str) -> i32 {
    if command.is_empty() {
        return -1;
    }

    let output = Command::new("sh")
        .arg("-c")
        .arg(command)
        .output();

    match output {
        Ok(output) => {
            if output.status.success() {
                //Print output
                io::stdout().write_all(&output.stdout).expect("Failed to write to stdout");
                io::stderr().write_all(&output.stderr).expect("Failed to write to stderr");
                return output.status.code().unwrap_or(0);
            } else {
                eprintln!("Command failed: {:?}", output.status);
                return -4;
            }
        }
        Err(e) => {
            eprintln!("Error executing command: {}", e);
            return -2;
        }
    }
}
```

figure 4.4: my_system function

```rust
fn main() {
    let mut choice = String::new();
    let mut command = String::new();

    loop {
        // Display menu
        println!("\n---Menu---");
        println!("1. Run program");
        println!("2. Kill a process");
        println!("3. List files in the current folder");
        println!("4. Open application (On MacOS)");
        println!("5. Exit");

        print!("Enter your choice: ");
        io::stdout().flush().expect("Failed to flush");

        io::stdin().read_line(&mut choice).expect("Failed to read line");
        choice = choice.trim().to_string();

        match choice.as_str() {
            "1" => {
                print!("Enter the command name to run: ");
                io::stdout().flush().expect("Failed to flush");
                io::stdin().read_line(&mut command).expect("Failed to read line");
                command = command.trim().to_string();
                let result = my_system(&command);
                println!("Command returned: {}", result);
            }
            "2" => {
                print!("Enter the process ID to kill: ");
                io::stdout().flush().expect("Failed to flush");
                let mut pid = String::new();
                io::stdin().read_line(&mut pid).expect("Failed to read line");
                pid = pid.trim().to_string();
                let command = format!("kill {}", pid);
                let result = my_system(&command);
                println!("Command returned: {}", result);
            }
            "3" => {
                let result = my_system("ls");
                println!("Command returned: {}", result);
            }
            "4" => {
                print!("Enter the application name to open: ");
                io::stdout().flush().expect("Failed to flush");
                io::stdin().read_line(&mut command).expect("Failed to read line");
                command = command.trim().to_string();
                let command = format!("/Applications/Figma.app/Contents/MacOS/{}", command);
                let result = my_system(&command);
                println!("Command returned: {}", result);
            }
            "5" => {
                println!("Exiting...");
                exit(0);
            }
            _ => println!("Invalid choice, try again"),
        }
        choice.clear();
        command.clear();
    }
}
```

figure 4.5: main function

Functions explanation:

- **io::stdout( ).flush( ).expect(“Failed to flush”);** is used to flush the output buffer. This is necessary because Rust uses buffered output by default.
- **io::stdin().read_line(&mut choice).expect("Failed to read line");** is used to read a line from the standard input and store it in the choice variable.
- **choice = choice.trim().to_string();** is used to remove the newline character from the end of the string and convert it to a String.
- **match choice.as_str() { ... }** is used to match the value of the choice variable with the different cases.
- **format!("kill {}", pid)** is used to format a string. In this case, it will return a string with the value of the pid variable appended to the string "kill ".
- **exit(0)** is used to exit the program with a status code of 0.
- **eprintln!("Command failed: {:?}", output.status);** is used to print an error message to the standard error output.
- **output.status.code().unwrap_or(0)** is used to get the exit code of the command. If the command fails, it will return 0.
- **io::stdout().write_all(&output.stdout).expect("Failed to write to stdout");** is used to write the output of the command to the standard output.
- **io::stderr().write_all(&output.stderr).expect("Failed to write to stderr");** is used to write the error output of the command to the standard error output.
- **output.status.success()** is used to check if the command was executed successfully.
- **output.status.code().unwrap_or(0)** is used to get the exit code of the command. If the command fails, it will return 0.

In Rust, we don't need to explicitly create a child process (as with **fork()** in C) or use **wait()** to wait for a command to finish executing. The standard Rust library, via the **std::process::Command** module, handles this in a cleaner, more abstract way.

When we use Command, it automatically creates a new process to execute the specified command. You can then wait for the child process to finish (or retrieve its output, if appropriate) using Command methods such as **.spawn()**, **.output()**, or **.status()**.

The project is entirely available on GitHub and the report is written under the name of README.md in the folder associated with Lab1 - For confidentiality reasons, the repository is defined as private. However, it is possible that you will be added as a collaborator in order to have access to the repository. Please let us know your username at maxime.heurtevent@edu.ece.fr so we can invite you.