The University of Prishtina "Hasan Prishtina"

The Faculty of Electrical and Computer Engineering

Technical documentation

Course: Operating Systems

Project Title: Communication between two processes

Name of Professor / Assistant **Name & Surname of students / email address**

Prof. Dr. Artan MAZREKAJ Msc. Dalinë VRANOVCI	1. Arlinda Kastrati	arlinda.kastrati4@student.uni-pr.edu
	2. Alberiana Tofaj	alberiana.tofaj@student.uni-pr.edu
	3. Fortesa Mujaj	fortesa.mujaj@student.uni-pr.edu
	4. Veranda Blakaj	veranda.blakaj@student.uni-pr.edu

Prishtinë, 2022

Table of Contents

Abstract

This report reports a detailed summary regarding the description of the design, implementation and enforcement of a communication program between two parent / child processes. This project was implemented using Ubuntu operating methods and the C programming language.

The main objective of the report is to present the principles behind the programming of operating systems or processes and the synchronization of the operating system that belongs to the Linux family.

Introduction

Operating systems are software programs that are used to manage computer devices such as: smartphones, tablets, computers, supercomputers, Internet servers, machines, smartwatches, etc. Operating systems are those that eliminate the need to know the coding language to interact with computer devices. Operating systems are a layer of graphical user interface (GUI), which acts as a platform between the user and the computer hardware. Furthermore, the operating system manages the software side of a computer and controls the execution of programs.

Without an operating system, each application will need to include its own UI, as well as the comprehensive code needed to handle all the basic low-level functions of the underlying computer, such as disk storage, network interfaces, and so on. Given the large range of basic hardware available, this would greatly increase the size of any application and make software development impractical.

The operating system today offers a comprehensive platform that identifies, configures and manages a range of hardware, including processors, memory devices and their management, chipsets, storage, networking, communication between ports such as Video Graphics Array (VGA), High-Definition Multimedia Interface (HDMI), Universal Serial Bus (USB) and subsystem interfaces such as Peripheral Component Interconnect Express (PCIe).

The five most common operating systems are: Microsoft Windows, Apple macOS, Linux, Android and Apple iOS.

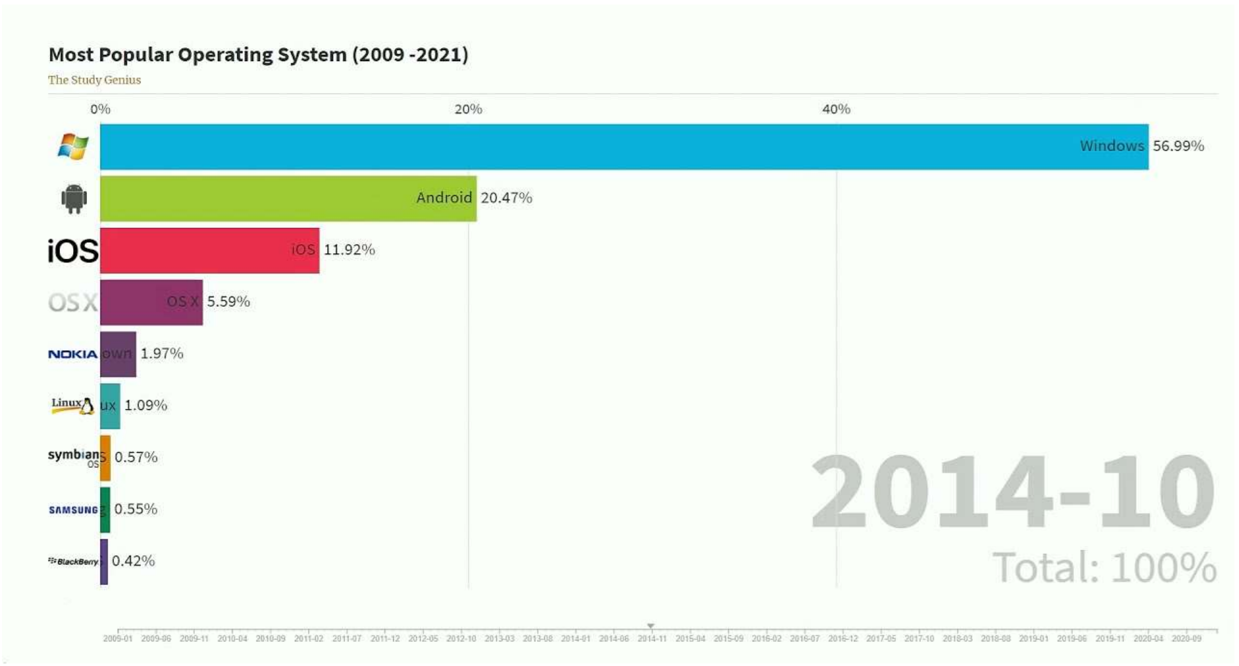


Figure 1: The most used operating systems (2009-2021)

1. The purpose of the project

Interprocess communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. It also enables the exchange of resources and data between processes without interference.

Processes that are running simultaneously on the operating system can be:

- independent process
- cooperating process

A process is independent and may or may not be affected by other processes running on the system. Any process that does not share data with any other process is independent.

Suppose if a process is cooperating then, it may be affected by other processes running on the system. Any process that shares data with another process is called a collaborative process.

To realize the communication between the two processes (parent / child) in such a way that the parent process writes in a file and the child process reads the contents of the file we can use pipes.

Pipe is a technique used for inter-processor communication. A pipe is a mechanism by which the output of one process is directed to the inlet of another process. Thus it provides a one-way flow of data between two related processes. Pipes communicate with the FIFO method. If the pipe is full, the process is blocked until the pipe condition changes. Similarly, a read process is blocked if it tries to read more bytes that are currently in the pipe, otherwise the read process is executed. Only one process can enter a pipe at a time.

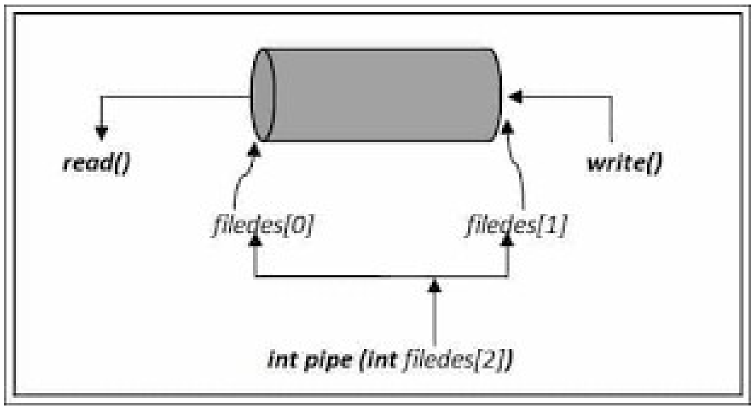
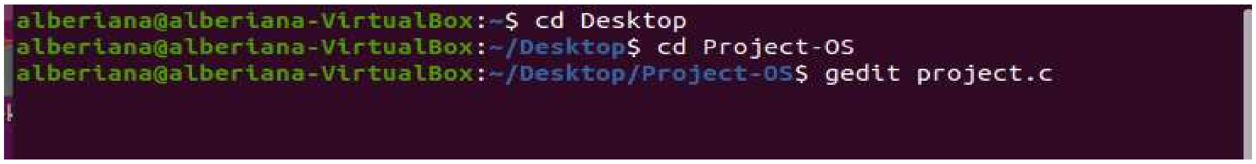


Figure 2: Pipe()

2. The main part

Since the **Ubuntu** operating system is already installed in the **Virtual Machine**, through the terminal we create a new file in the directory where the file we accessed with the **cd** command will be saved.

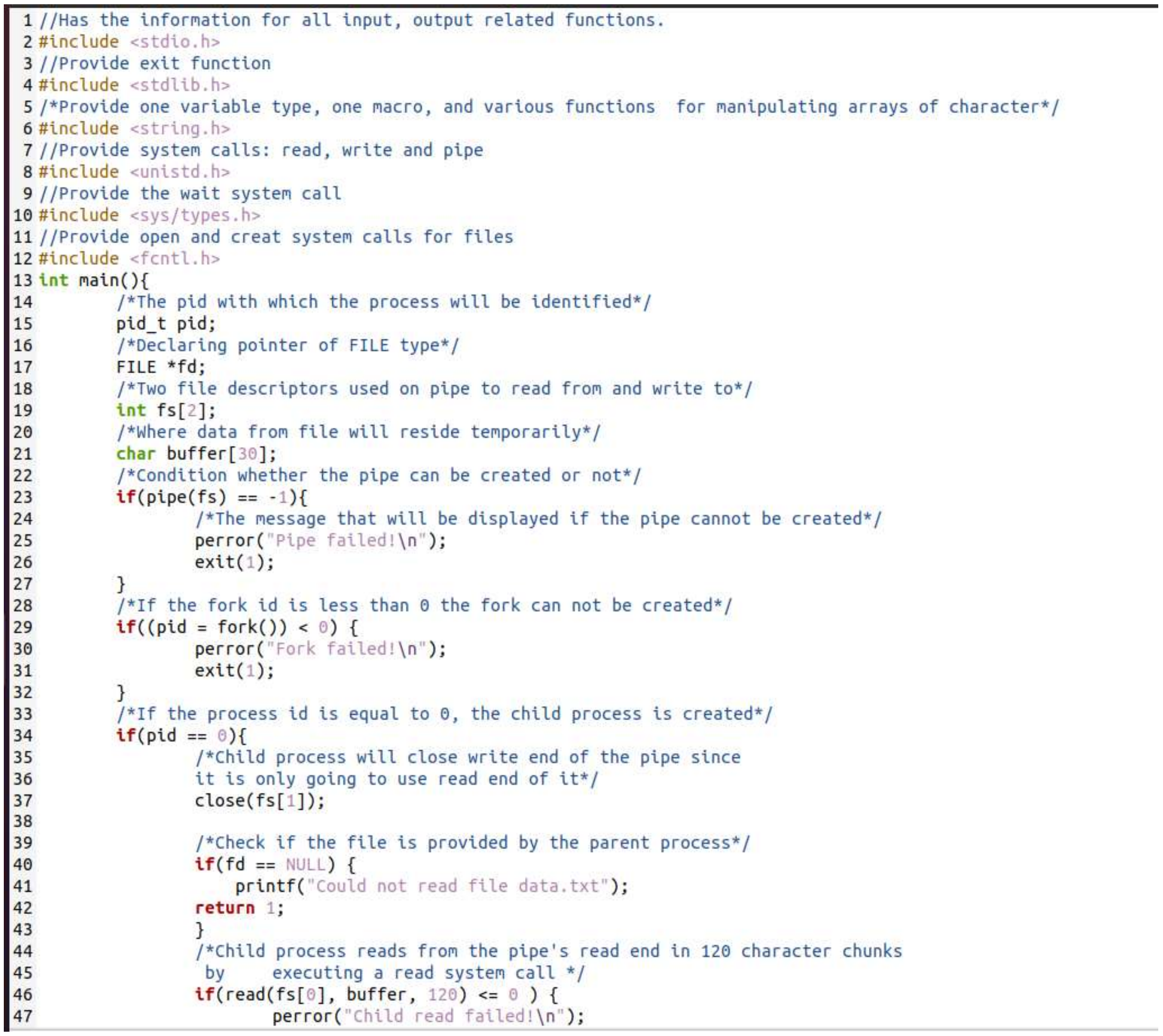


```
alberiana@alberiana-VirtualBox:~$ cd Desktop
alberiana@alberiana-VirtualBox:~/Desktop$ cd Project-OS
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ gedit project.c
```

Figure 3: The file creation process

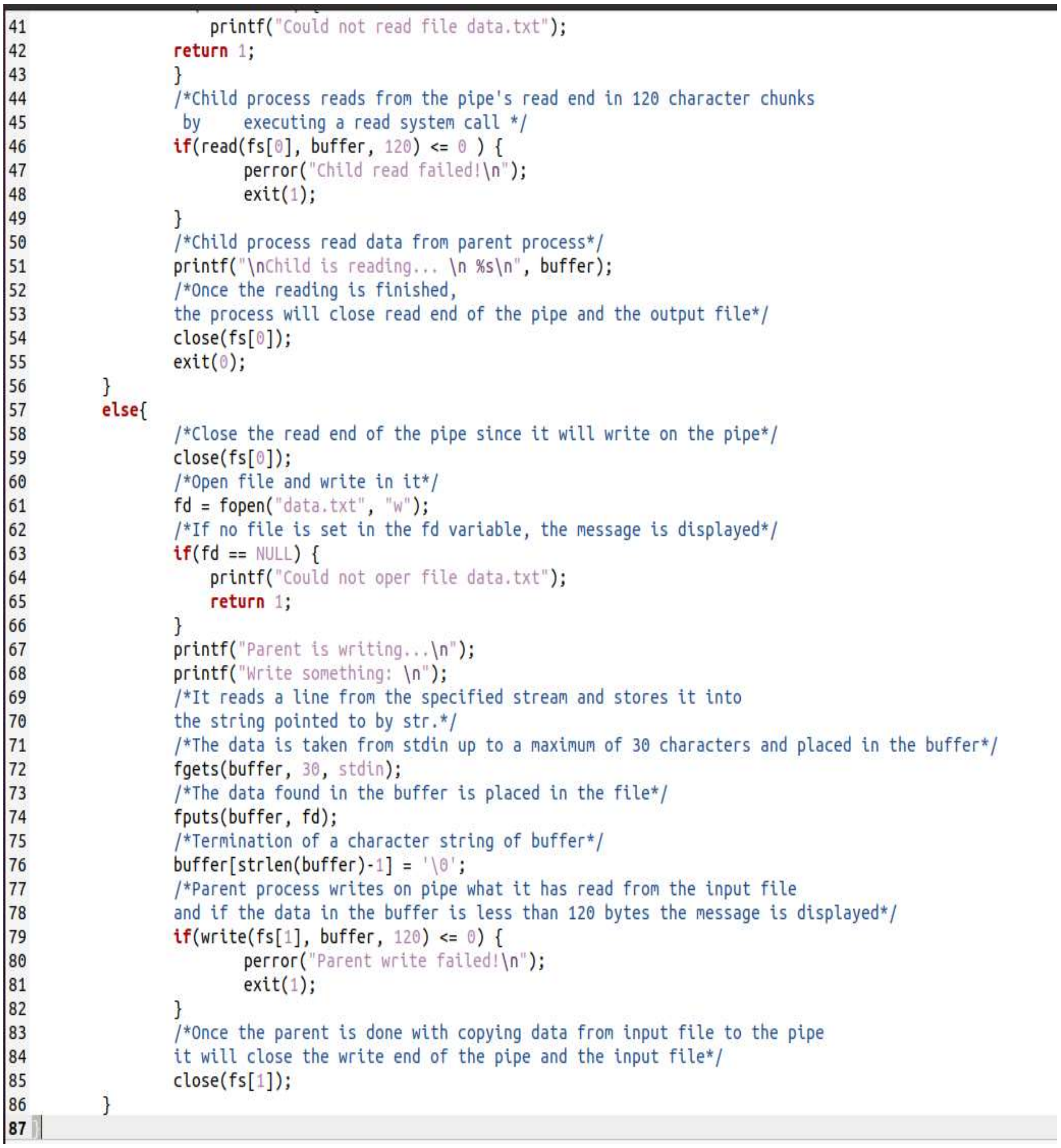
2.1 The Code

Below we have presented the code in the C programming language.



```
//Has the information for all input, output related functions.
#include <stdio.h>
//Provide exit function
#include <stdlib.h>
/*Provide one variable type, one macro, and various functions  for manipulating arrays of character*/
#include <string.h>
//Provide system calls: read, write and pipe
#include <unistd.h>
//Provide the wait system call
#include <sys/types.h>
//Provide open and creat system calls for files
#include <fcntl.h>
int main(){
        /*The pid with which the process will be identified*/
        pid_t pid;
        /*Declaring pointer of FILE type*/
        FILE *fd;
        /*Two file descriptors used on pipe to read from and write to*/
        int fs[2];
        /*Where data from file will reside temporarily*/
        char buffer[30];
        /*Condition whether the pipe can be created or not*/
        if(pipe(fs) == -1){
                /*The message that will be displayed if the pipe cannot be created*/
                perror("Pipe failed!\n");
                exit(1);
        }
        /*If the fork id is less than 0 the fork can not be created*/
        if((pid = fork()) < 0) {
                perror("Fork failed!\n");
                exit(1);
        }
        /*If the process id is equal to 0, the child process is created*/
        if(pid == 0){
                /*Child process will close write end of the pipe since
                it is only going to use read end of it*/
                close(fs[1]);

                /*Check if the file is provided by the parent process*/
                if(fd == NULL) {
                    printf("Could not read file data.txt");
                return 1;
                }
                /*Child process reads from the pipe's read end in 120 character chunks
                 by     executing a read system call */
                if(read(fs[0], buffer, 120) <= 0 ) {
                        perror("Child read failed!\n");
```

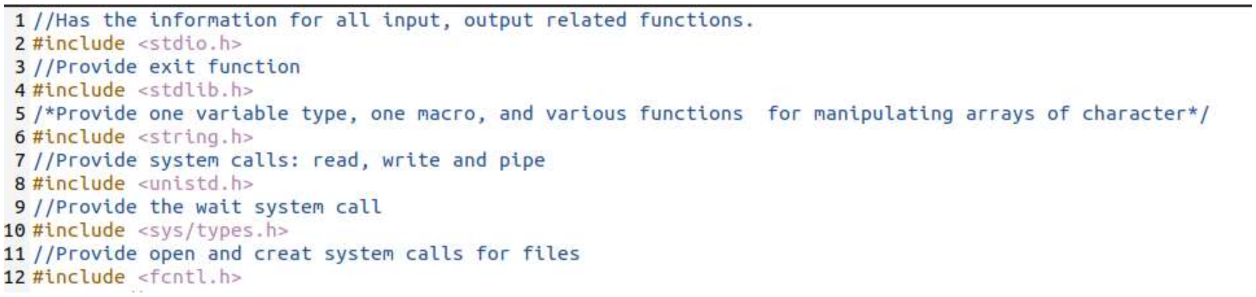
Figure 4: The first part of the code



```
                    printf("Could not read file data.txt");
                return 1;
                }
                /*Child process reads from the pipe's read end in 120 character chunks
                 by     executing a read system call */
                if(read(fs[0], buffer, 120) <= 0 ) {
                        perror("Child read failed!\n");
                        exit(1);
                }
                /*Child process read data from parent process*/
                printf("\nChild is reading... \n %s\n", buffer);
                /*Once the reading is finished,
                the process will close read end of the pipe and the output file*/
                close(fs[0]);
                exit(0);
        }
        else{
                /*Close the read end of the pipe since it will write on the pipe*/
                close(fs[0]);
                /*Open file and write in it*/
                fd = fopen("data.txt", "w");
                /*If no file is set in the fd variable, the message is displayed*/
                if(fd == NULL) {
                    printf("Could not oper file data.txt");
                    return 1;
                }
                printf("Parent is writing...\n");
                printf("Write something: \n");
                /*It reads a line from the specified stream and stores it into
                the string pointed to by str.*/
                /*The data is taken from stdin up to a maximum of 30 characters and placed in the buffer*/
                fgets(buffer, 30, stdin);
                /*The data found in the buffer is placed in the file*/
                fputs(buffer, fd);
                /*Termination of a character string of buffer*/
                buffer[strlen(buffer)-1] = '\0';
                /*Parent process writes on pipe what it has read from the input file
                and if the data in the buffer is less than 120 bytes the message is displayed*/
                if(write(fs[1], buffer, 120) <= 0) {
                        perror("Parent write failed!\n");
                        exit(1);
                }
                /*Once the parent is done with copying data from input file to the pipe
                it will close the write end of the pipe and the input file*/
                close(fs[1]);
        }
```

Figure 5: The second part of the code

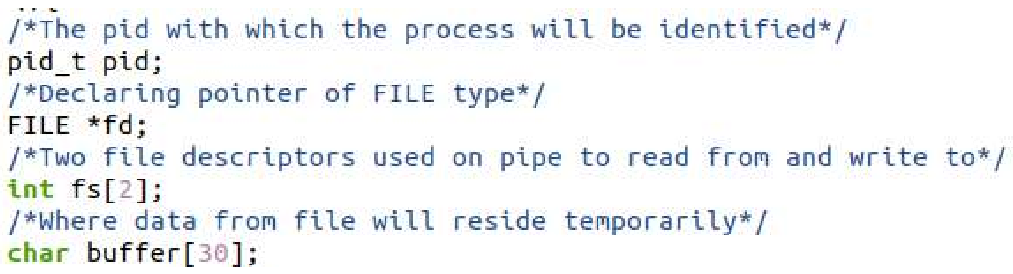
Below are the libraries that will be used to enable parent-child communication.



```
//Has the information for all input, output related functions.
#include <stdio.h>
//Provide exit function
#include <stdlib.h>
/*Provide one variable type, one macro, and various functions  for manipulating arrays of character*/
#include <string.h>
//Provide system calls: read, write and pipe
#include <unistd.h>
//Provide the wait system call
#include <sys/types.h>
//Provide open and creat system calls for files
#include <fcntl.h>
```

Figure 6: Libraries

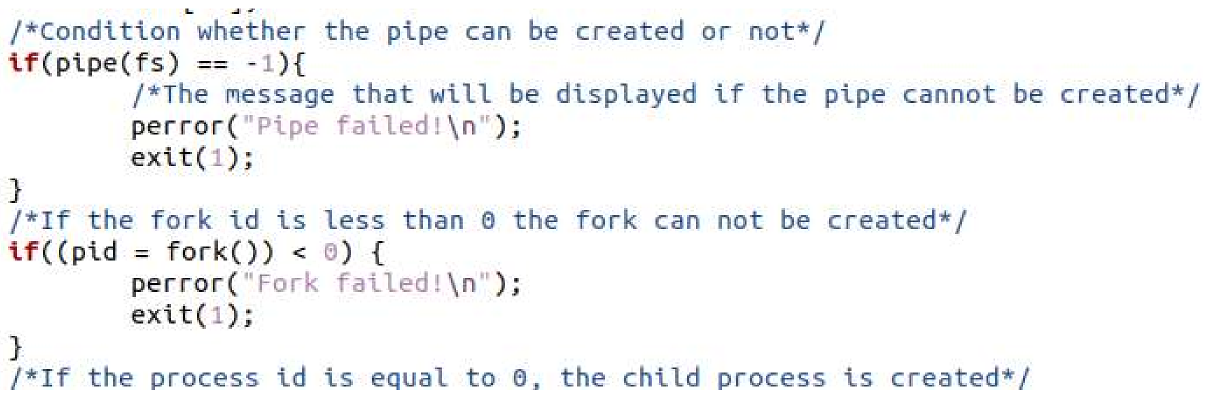
Declaration of variables that will be used later in the program.



```
/*The pid with which the process will be identified*/
pid_t pid;
/*Declaring pointer of FILE type*/
FILE *fd;
/*Two file descriptors used on pipe to read from and write to*/
int fs[2];
/*Where data from file will reside temporarily*/
char buffer[30];
```

Figure 7: Declared variables

Creating conditions that if the pipe and process can not be created the message will be displayed as a notification.

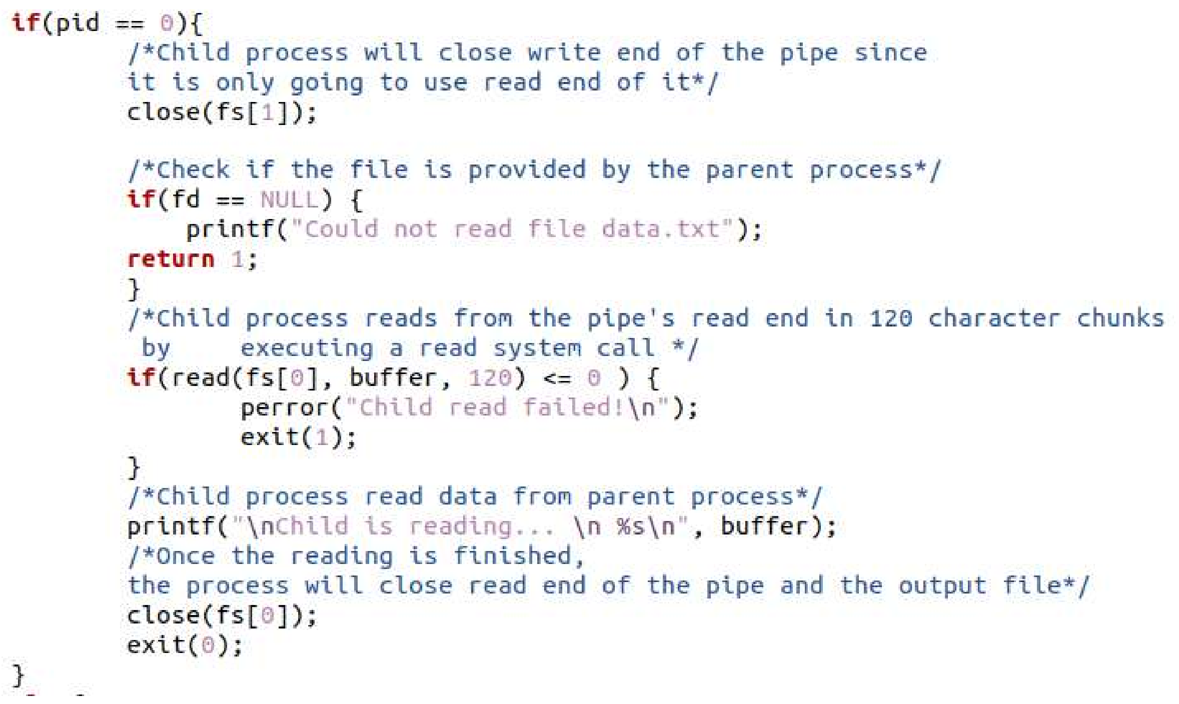


```
/*Condition whether the pipe can be created or not*/
if(pipe(fs) == -1){
        /*The message that will be displayed if the pipe cannot be created*/
        perror("Pipe failed!\n");
        exit(1);
}
/*If the fork id is less than 0 the fork can not be created*/
if((pid = fork()) < 0) {
        perror("Fork failed!\n");
        exit(1);
}
/*If the process id is equal to 0, the child process is created*/
```

Figure 8: Pipe() and fork()

In this part of the code is presented the child process means that if the condition for **pid == 0** is met then the program continues with the execution of the code for the child process. First the pipe closes because the child process will only read the file sent by the parent process then the condition that if the parent process has not placed the file in the declared variable the corresponding message is displayed.

In the other condition **if**, we look at whether the child process can read or not in the declared file, if it can display the message it has read if not then the corresponding message is displayed. Finally the reading pipe is closed.



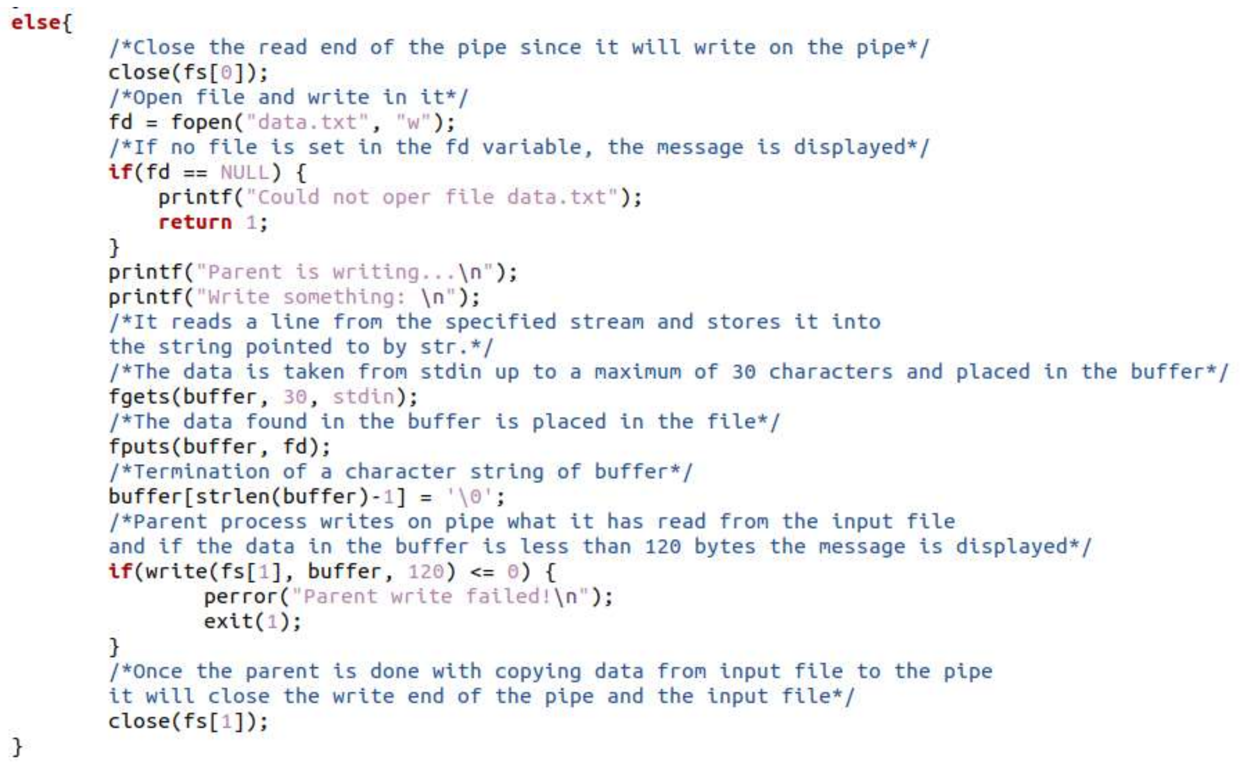
```
if(pid == 0){
        /*Child process will close write end of the pipe since
        it is only going to use read end of it*/
        close(fs[1]);

        /*Check if the file is provided by the parent process*/
        if(fd == NULL) {
            printf("Could not read file data.txt");
        return 1;
        }
        /*Child process reads from the pipe's read end in 120 character chunks
         by     executing a read system call */
        if(read(fs[0], buffer, 120) <= 0 ) {
                perror("Child read failed!\n");
                exit(1);
        }
        /*Child process read data from parent process*/
        printf("\nChild is reading... \n %s\n", buffer);
        /*Once the reading is finished,
        the process will close read end of the pipe and the output file*/
        close(fs[0]);
        exit(0);
}
```

Figure 9: The child process

The following is the part of the code for the parent process. First the pipe for reading gets closed then the pipe for writing gets open.

The condition if the file can not be opened presents the corresponding message while if the file can be opened the process starts reading the data written by the parent process in the file.

If no data is written to the file the message is displayed. And finally closes the pipe for writing on it.



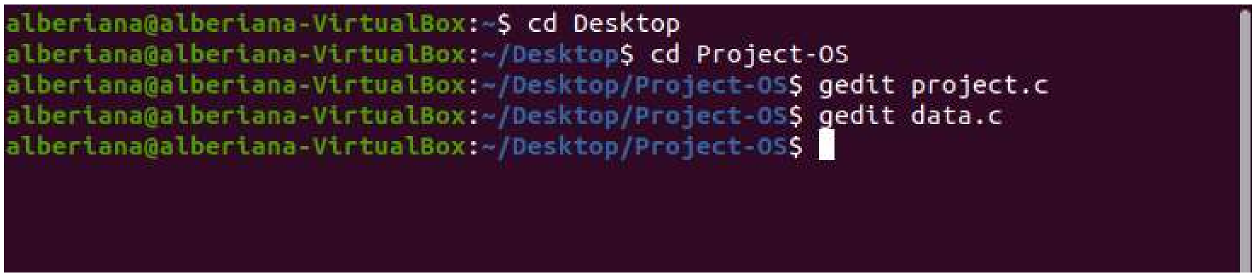
```
else{
        /*Close the read end of the pipe since it will write on the pipe*/
        close(fs[0]);
        /*Open file and write in it*/
        fd = fopen("data.txt", "w");
        /*If no file is set in the fd variable, the message is displayed*/
        if(fd == NULL) {
            printf("Could not oper file data.txt");
            return 1;
        }
        printf("Parent is writing...\n");
        printf("Write something: \n");
        /*It reads a line from the specified stream and stores it into
        the string pointed to by str.*/
        /*The data is taken from stdin up to a maximum of 30 characters and placed in the buffer*/
        fgets(buffer, 30, stdin);
        /*The data found in the buffer is placed in the file*/
        fputs(buffer, fd);
        /*Termination of a character string of buffer*/
        buffer[strlen(buffer)-1] = '\0';
        /*Parent process writes on pipe what it has read from the input file
        and if the data in the buffer is less than 120 bytes the message is displayed*/
        if(write(fs[1], buffer, 120) <= 0) {
                perror("Parent write failed!\n");
                exit(1);
        }
        /*Once the parent is done with copying data from input file to the pipe
        it will close the write end of the pipe and the input file*/
        close(fs[1]);
}
```

Figure 10: The parent process

2.2 Results from testing the code

In the following we have presented the whole process of testing parent-child communication through **pipes ().**

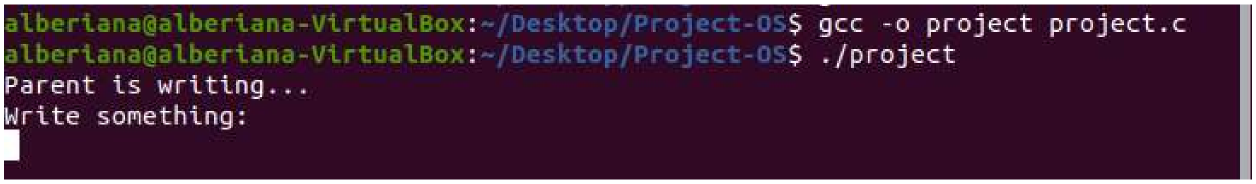
The following figure shows the commands for creating the file in the programming language and the text file in which we will enter the data.



```
alberiana@alberiana-VirtualBox:~$ cd Desktop
alberiana@alberiana-VirtualBox:~/Desktop$ cd Project-OS
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ gedit project.c
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ gedit data.c
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ 
```

Figure 11: Creating files

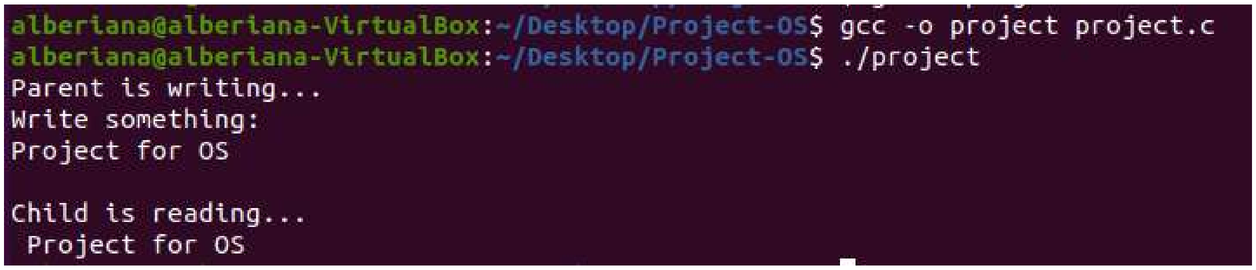
The data in the file is written by the parent process. The process expects the data to be recorded in the corresponding declared file.



```
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ gcc -o project project.c
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ ./project
Parent is writing...
Write something:
```

Figure 12: The process waits for the input data

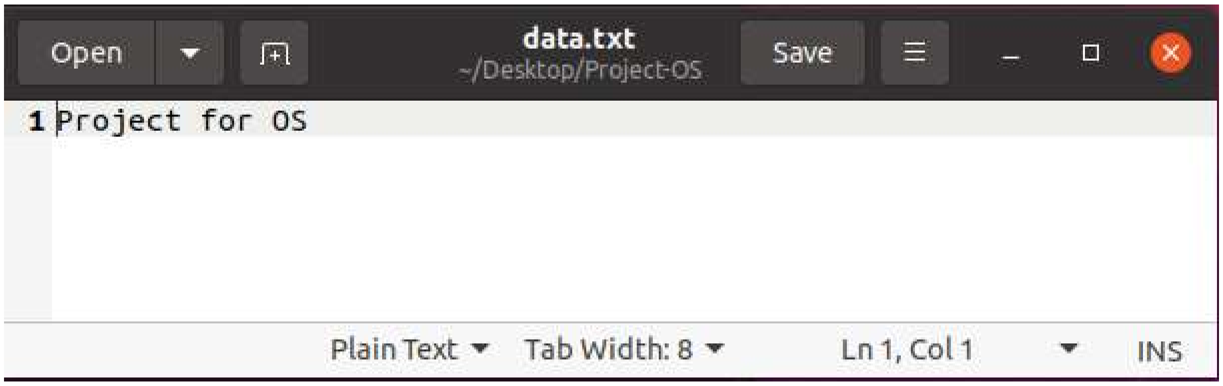
The data are written by the parent process through the terminal in the declared file and also the data is read by the child process appearing in the terminal.



```
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ gcc -o project project.c
alberiana@alberiana-VirtualBox:~/Desktop/Project-OS$ ./project
Parent is writing...
Write something:
Project for OS

Child is reading...
 Project for OS
```

Figure 13: The data written in the file and getting read from the child process

The data details in the data.txt file are as follows:



```
Project for OS
```

Figure 14: File data.txt

3. Conclusion

The realization of the whole project was quite challenging but at the same time it was fun. This project is the first project we have worked on of this nature that belongs to the operating systems, especially about processes and their communication. As a beginner student in this field we have encountered some obstacles during the implementation of the project. But, with the help of literature from foreign books, basic concepts taken from lectures and exercises from the course of Operating Systems, cooperation, common ideas and numerous researches we believe that we have successfully achieved the realization of communication between the two processes. All of this has also resulted into greater reinforcement of concepts that include architecture and process communication of the operating systems.

The program implemented above realizes the communication between two processes (parent / child) in that way:

• The parent process writes to a file

• The child process reads the contents of the file

and works properly.

References

[1] "Stack Exchange," 26 May 2021. [Online]. Available: https://unix.stackexchange.com/questions/446131/the-file-used-by-parent-and-child-process..

[2] "Geeksforgeeks," 14 May 2020. [Online]. Available: https://www.geeksforgeeks.org/ipc-technique-pipes/#:~:text=A%20Pipe%20is%20a%20technique,data%20between%20two%20related%20processes...

[3] "Tutorialspoint," [Online]. Available: https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm..

[4] H. Darvis, "Quick Adviser," 4 November 2021. [Online]. Available: https://quick-adviser.com/how-parent-processes-communicate-with-child-process/..