Operating Systems

Second Assignment Report

Members: Carlos Iborra Llopis / Pablo Brasero Martínez / Marcos Caballero Cortés
NIAs: 100451170 / 100451247 / 100451047
Group: 89

Table of Contents

1. Code Description

The purpose of this assignment is to introduce and practice the new process management services provided by POSIX. One of the main goals is to understand how the shell works on UNIX/Linux.
Briefly, the shell allows the user to interact with the operating system's kernel using single or combined commands. POSIX system calls such as *“fork()”*, *“wait()”*, or *“exit()”* have been used, also, *“pipe()”*, *“dup()”*, *“close()”* and *“signal()”* system calls.

For the correct development of this lab project a parser has been given to us, the functionality of the parser is to read what is going to be introduced by the user, some of these commands are a space, *a string, a command, a command sequence, redirection, background, the prompt*, each one of them with its own functionality and specific application for the program.

As a group, we have decided to use just one programming style throughout this project, thus making it much easier and more readable, especially when working as a team because it makes pair-programming easier. Therefore, we have agreed on using Snake Case (snake_case) throughout the program for naming our variables, functions, and other method names, in addition, we have decided to use the "Allman" indentation method for brace placement in compound statements for coherence and code readability. The indentation length has been kept constant to 4 spaces, regardless of the preferred indentation depth of each style.

Our code begins with the required elements specified at the beginning of the "project statement", such as checking that the number of commands is adequate, meaning, that the maximum number allowed is not exceeded, carried out as follows with a conditional that checks that the condition is met correctly and if it is not, it will raise a standard output error on the screen and then we use “*goto when_error;”* to return to the top.

The organization of this document consists of the exhaustive explanation of the internal commands: mycalc and mycp as well as the different applications and functionalities the functions have along with explaining what we have done with simple commands, command sequences, and background, Then we continue by doing the Test Cases section in which we will present the different tests carried out on the program where we will show the input, the received output and the functionality of each of the tests. Finally, we will finish with a conclusion in which we will collect all the elements seen, as well as the problems encountered and the impressions we have received when doing the project.

1.1. mycalc

mycalc works as a simple calculator in the command line. It takes two operands and an operator, displayed in the following way: “*operand_1*” “*operator*” “*operand_2*”; where an operand is an integer number and the operator can be either “add” or “mod”.
Note how in mod operator, we used *dividend* instead of *operand_1* and *divisor* instead of *operand_2* for it to be more legible.

In this function we only need to implement those two functionalities, the addition (add) with the “*Acc*” variable (which stores the aggregated sum results of the different add operations, note how we created “result_acc” outside the while(1), so it is equal to 0 only at start) and the module (mod) with its corresponding quotient.

To make sure it works correctly we will check that the two operands are numbers, if one or both of the parameters aren’t numbers, they will be transformed into integer 0 in our calculator because of “*atoi*” function, this way we assure the correct execution of the calculator even if one or both parameters are not correct.

To start with, *mycalc* will check the input, if the operands are not *NULL* and the operator is either “*add*” or “*mod*”.
The expected error message will be stored in a variable just in case an error occurs while performing any of the operations, the standard error message which is like this: “*[ERROR] The structure of the command is mycalc <operand_1> <add/mod> <operand_2>*”, will be raised.

1.1.1. add operator

In order to use *add* we will have to check that the third argument is ”*add*”, we check it by using: “*if (strcmp(argv_execvp[2], "add") == 0)*”, then, we store the operands into two different variables and add them with the “Acc” environment variable, that will start with value 0, and store it into the same variable.

Then we create the buffer “*ok_add*” and store the resulting string in the buffer (using “*snprintf()*”) and we print the value in the buffer using “*fprintf(stderr, "%s", ok_add);*”.

Then a message with this syntax will be printed if there is no printing error: "*[OK] <Operand 1> + <Operand 2> = <Result>; Acc <Acc Value>*".

1.1.2. mod operator

In order to use *mod* we will have to check that the third argument is “*mod*”, we check it by executing: “*if (strcmp(argv_execvp[2], "mod") == 0)*”, then, we store the second argument in the *dividend* variable and the fourth argument in the *divisor* variable.

Next, we create the buffer “*ok_mod*” and store the resulting string in the buffer (using “*snprintf()*”) and we print the value in the buffer using “*fprintf(stderr, "%s", ok_mod);*”.

Then a message with this syntax will be printed if there is no printing error: "*[OK] <Dividend> % <Divisor> = <Remainder>; Quotient <Quotient>*". If an error occurs when printing, it raises the error: "*ERROR when writing*".

1.2. mycp

This second function works in a similar way as the command *cp* in windows. It copies the content of an origin file and pastes it into the destination file.

First of all we check we have entered *mycp* as the first argument: "*if (strcmp(argv_execvp[0], "mycp") == 0)*".
We also define inside two variables the two possible error messages: (*error_message1[100] = "[ERROR] Error opening original file";)* when occurred an error while opening the original file and (*error_message2[100] = "[ERROR] The structure of the command is mycp <original file> <copied file>";)* due to a structural error.

Then we make sure both files are not *NULL*. If not, we open the origin file in read-only mode, as well as the destination file and a buffer to transfer the information from one to the other file. With help of a while, we move and read through the origin file, copy the data into the buffer, and then we write it into the destination file.

In this last part, there are several error messages that will be printed if the destination file can't be closed, if the command can't be written correctly, if there is an error reading the file or if there was an error closing the file (using "*perror"* and then going to "*when_error")*. All of these error checks will be done throughout the program to check everything works alright.

When we finish the process, we close both of them and this message will be raised: "*[OK] Copy has been successful between <origin_file> and <destination file>*" (using "*fprintf(stderr, …)")*. In any other case, an error message will be printed on the screen.

1.3. Simple Commands, Command Sequences, and Background

To execute a simple command, it should be checked if redirections have been made in the parameters filev[0], filev[1], and filev[2], in case of redirections have been made, the descriptor will be changed taking into account that it will have to readjust the code to the standard descriptors.

Since it is a **simple command**, there is no need to create pipes or close them later on. The following process will be to create a "*fork()*" (using *int PID = fork();*) and execute the command that we have been working on; throughout the entire process all possible errors have been taken into account and will be raised (for example "ERROR when fork", "ERROR when closing descriptor\n", "ERROR when opening file\n", "ERROR when executing\n", “ERROR when duplicating descriptor\n"...).

Command connected with pipes (“*pipe(fd)”*), tries to execute numerous commands at the same time, at the start, it is more complex but it does not differ too much. To begin with, all possible redirections must be checked, since if there are any, the corresponding redirections will be established in the parent process, this is done so that later the children inherit the redirections directly.
We took into account that within the child process, redirections must be made in order to connect all the children through pipes, and in this way, we ensure that the descriptors are adjusted properly.

To sum up, in each iteration the child process will be executed, and the result will be correctly directed to the next pipe or the final output if it is the last process.

As for the **background**, we could say that it is an element that allows processes to be executed, "behind" the main processes, so depending on the input that is received, it will be run in one way or another, that is to say, if the variable "in_background" takes as value, 1, the parent process will not wait for the child process to be executed in the background, however, if the value it takes is 0, the parent will wait for the child using a *wait() (“while (wait(&stat) > 0)”),* since no process will run in the background.

2. Test cases

In order to verify the correct functioning of our program, we have performed 23 tests plus another 10 extra tests that verify the multiple inputs received by the program. The objective of these tests is to put the program under the biggest pressure possible by means of extreme cases that can be received as input. In addition to checking all possible cases, from an input where the structure is wrong to another one where some element of the command is missing and therefore, the received output is neither the expected nor the correct one.

Test	Description	Input	Expected output	Result
Test1	1 command	*wc -l foo.txt*		OK
Test2	1 command + input redirection	*wc -l < foo.txt*		OK
Test3	1 command + output redirection	*cat foo.txt > salida_msh*	cat foo.txt > salida_bash	OK
Test4	1 command + error redirection	*cat noexiste.txt >& res.txt*		OK
Test5	2 commands	*cat foo.txt | grep a*		OK
Test6	2 commands + input redirection	*grep 1 | grep a < foo.txt*	10a	OK
Test7	2 commands + output redirection	*cat foo.txt | grep a > salida_msh*	cat foo.txt | grep a > salida_bash	OK
Test8	2 commands + error redirection	*cat noexiste.txt | grep a >& salida_msh*	cat noexiste.txt | grep a	OK
Test9	3 commands	*cat foo.txt | grep a | grep 1*		OK
Test10	3 commands + input redirection	*grep 1 | grep a | wc -l < foo.txt*	1	OK
Test11	3 commands + output redirection	*cat foo.txt | grep a | wc -l > salida_msh*	cat foo.txt | grep a | wc -l > salida_bash	OK
Test12	3 commands + error redirection	*cat noexiste.txt | grep a | wc -l >& salida_msh*	cat noexiste.txt | grep a | wc -l	OK
Test13	N commands	*cat foo.txt | grep 1 | grep 2 | grep 3 | grep 4*		OK

Test14	Calculator 1	*mycalc 3 add -8*	[OK] 3 + -8 = -5; Acc -5	OK
Test15	Calculator 2	*mycalc 5 add 13*	[OK] 3 + -8 = -5; Acc -5	OK
Test16	Calculator 3	*mycalc 10 mod 7*	[OK] 10 % 7 = 3; Quotient 1	OK
Test17	Calculator 4	*mycalc 10 pp 7*	[ERROR] The structure of the command is mycalc <operand_1> <add/mod> <operand_2>	OK
Test18	Calculator 5	*mycalc mycalc 8 mas*	[ERROR] The structure of the command is mycalc <operand_1> <add/mod> <operand_2>	OK
Test19	Calculator $Acc			OK
Test20	Backup 1	*mycp*	[ERROR] The structure of the command is mycp <original_file> <copied_file>	OK
Test21	Backup 2	*mycp noexiste.txt .*	[ERROR] Error opening original file	OK
Test22	Backup 3	*mycp msh.c msh.c_bak*	[OK] Copy has been successful between msh.c and msh.c_bak	OK
Test23	Background	*ps | grep sleep | wc -l*	ps | grep sleep | wc -l	OK

As it can be noticed, all the test results have the value "OK", this is because all the results given by the tests have been successfully solved, and as consequence, the expected output is equal to the one obtained.
These previous tests, as have been verified, bring the code to the limit by testing all kinds of possible inputs, one command, two or three commands, and both input or output redirection, that is, covering all the possibilities that may occur in the course of the program.

In order to develop the correctness of our code, we have also implemented another 10 extra tests. These tests complement the previous ones so that we cover more possibilities that the program can receive.

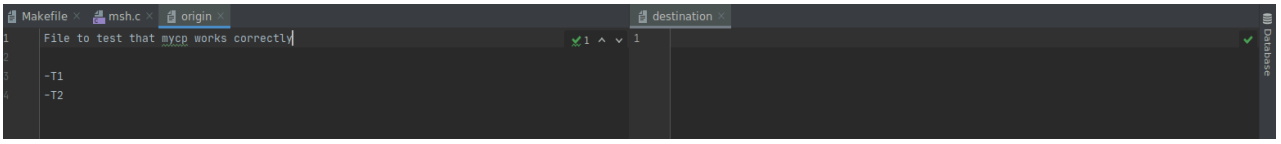
Extra Test1	Calculator 1 extra	mycalc a add 10	[OK] 0 + 10 = 10; Acc 10	OK
Extra Test2	Calculator 2 extra	mycalc a add b	[OK] 0 + 0 = 0; Acc 10	OK
Extra Test3	Calculator 3 extra	*mycalc 12 add 4 add 6*	*[OK] 12 + 4 = 16; Acc 26	OK

Extra Test4	Calculator 4 extra	*mycalc 3 add 5 mod 2*	[OK] 3 + 5 = 8; Acc 8	OK
Extra Test5	Calculator 5 extra	*mycalc 10 mod 2 mod 1*	[OK] 10 % 2 = 0; Quotient 5	OK
Extra Test6	Calculator 6 extra	*mycalc 20 mod 5 add 4***	[OK] 20 % 5 = 0; Quotient 4	OK
Extra Test7	Calculator 7 extra	*mycalc a mod 10*	[OK] 0 % 10 = 0; Quotient 0	OK

***Note:** Acc is the accumulated value of the sum, which explains why sometimes its value does not correspond to the result of the operation.
****Note:** Everything written after operand_2 is omitted by the MSH.

Extra Test8	mycp	*Mycp origin destination*	[OK] Copy has been successful between origin and destination	OK
Extra Test9	mycp	*mycp origin doesnotexist*	[OK] Copy has been successful between origin and destination	OK
Extra Test10	mycp	*mycp doesnotexist destination*	[OK] Copy has been successful between origin and destination	OK

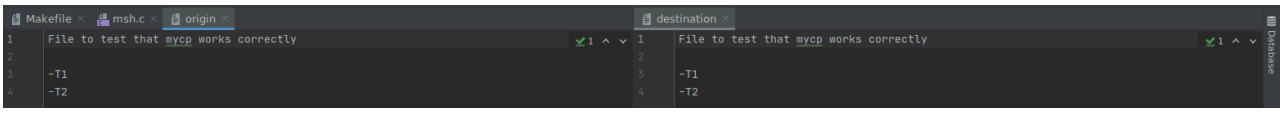
In order to see the last test more clearly, we have added some pictures below. For the first test for *mycp*, we created 2 new files, one with the name of origin and the other with the name of the destination. In origin, we wrote some lines as a way of testing and as you can see in the following screenshot, the origin file has some data in it while the destination file is empty.



Afterward, we executed the command as can be seen in the next screenshot and we obtained the following result.

```
MSH>>mycp origin destination
[OK] Copy has been successful between origin and destination
```

So after executing the code, this result is obtained.



The second test consists of sending the information from an origin file that exists to a destination file that does not exist, this returns us the following message.

```
MSH>>mycp origin doesnotexist
[OK] Copy has been successful between origin and doesnotexist
```

The third test consists of sending the information from a file that does not exist to a destination file that does exist so this is what happens and this message is returned.

```
MSH>>mycp doesnotexist destination
[OK] Copy has been successful between doesnotexist and destination
```

There we also verified the functionality of mycalc. Mycalc works as a simple calculator on the command line. It takes two operands and an operator, displayed in this way: *“operand_1” “operator” “operand_2”*; where an operand is an integer number and the operator can be “*add*”, where the two numbers will be added, or “*mod*”, in which the quotient, as well as the reminder, must be computed.

If the operation is successful, the command must show the result of the computation preceded by the label: “*[OK]*”. If the operator does not correspond with the ones previously described, or not all the terms of the equation were introduced, or if any of the source/destinations cannot be opened, or if the function is not called with two parameters an error message will be shown.

Finally, we executed the checker_os_p2.sh, which returned us the following:

```
aulavirtual@Linuxv3074:~/Escritorio/p2_minishell_2022$ ./checker_os_p2.sh
ssoo_p2_100451170_100471247_100451047.zip

*** TESTING THE MINISHELL
Archive:  ssoo_p2_100451170_100471247_100451047.zip
replace Makefile? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
  inflating: Makefile
  inflating: authors.txt
  inflating: checker_os_p2.sh
  inflating: libparser.so
  inflating: msh.c
File : ssoo_p2_100451170_100471247_100451047.zip
Compiling
OK
Test1: 1 command -> Test command:  test1
 OK
Test2: 1 command + input redirection -> Test command:  test2
```

```
 OK
Test3: 1 command + output redirection ->  OK
Test4: 1 command + error redirection ->  OK
Test5: 2 commands -> Test command:  test5
 OK
Test6: 2 commands + input redirection ->  OK
Test7: 2 commands + output redirection ->  OK
Test8: 2 commands + error redirection ->  OK
Test9: 3 commands -> Test command:  test8
 OK
Test10: 3 commands + input redirection ->  OK
Test11: 3 commands + output redirection ->  OK
Test12: 3 commands + error redirection ->  OK
Test13: N commands -> Test command:  test11
 OK
Test14: Calculator 1 ->  OK
Test15: Calculator 2 ->  OK
Test16: Calculator 3 ->  OK
Test17: Calculator 4 ->  OK
Test18: Calculator 5 ->  OK
Test19: Calculator $Acc ->  OK
Test20: Backup 1 ->  OK
Test21: Backup 2 ->  OK
Test22: Backup 3 ->  OK
Test23: Background -> Test command:  test21
 OK
Summary of tests: ->
grade: 10.00
```

3. Conclusions

During this practice, the main problems we have encountered are related to the use of Linux, as this is our second practice and we still have to manage the differences between this operating system and the ones we are used to (mainly windows 10/11), not only that but we had trouble with the Virtual Box Linux Ubuntu 20.04 LTS “Focal Fossa” we had installed because Virtual Box did not let us drag and drop between the main desktop and the virtual one, so it made it useless. As a consequence we needed to use the Aula Virtual Linux environment although it can be quite slow and messy sometimes, we managed to finish all the coding within the given time.

We also had some trouble with the implementation of the *“fork()”* used to create new processes (child processes) along with the creation of pipes as we needed to review all the theory and laboratory content in order to understand and recreate those parts of the code.

But the most difficult thing we have faced was comprehending and correcting the few errors that have been appearing in the code. When we were testing, we had some problems and we didn’t know how to solve them, but after a long time of trying different things, doing some research on the internet, and asking the teachers, we managed to solve them.

We also had a big problem with the brackets, as we are not used to coding with them (as we use mostly python), this ended up with us having to stay for hours looking for unpaired brackets and problems with the indentation of them, that is why we decided to use the "Allman" indentation method for brace placement in compound statements, as mentioned in the code description because it made easier for us to detect the different bracket relations.

Furthermore, we also had some problems related to tests 17 & 18: ‘calculator 4’ & ‘calculator 5’ respectively, where an operand disappeared. This was solved using the *fprintf()* function to print the standard output errors stored previously in the different buffers.

Additionally, in test 2: ‘backup 3’ we also had a problem similar to the one explained above, and was solved using the same function, (*fprintf()*)

Just as an additional comment on our problems, we had an issue with the mycalc, specifically with the add operand, as it printed two times the same code. This was due to us wanting to test if an error occurred when using *fprintf()* but we did not notice that the code “*if ((write(1, …, strlen(...))) < 0){ perror ("ERROR when writing"); goto when_error;}*” we had written to test the errors already did the same as the new function *fprintf()* but also checking if a writing error occurred. Due to this, we decided to use only the function *fprintf()* for the sake of simplicity.

As for the rest of the practice, we have not found any more errors, this has allowed us to successfully complete all the required elements without unexpected problems.

To sum up, this assignment has made us work as a team in a great way, organizing everything and helping us to review and understand everything we have seen in class and specifically in the last exam, so now we know how to apply it to the real world and not just as theory.