

# **LAB 2: MINISHELL**

## **AUTHORS:**

**LAURA MARTÍN DOMÍNGUEZ : 100496779 / 100496779@alumnos.uc3m.es**

**LUCIA CALDADO REIN: 100496904/100496904@alumnos.uc3m.es**

**PAULA GARCÍA NADAL: 100496715/ 100496715@alumnos.uc3m.es**

## TABLE OF CONTENTS

CONTENT	PAGE
Description of the code	2-6
Test Cases	6-14
Conclusions	14-15

# INTRODUCTION

This lab is focused on the implementation of a minishell capable of interpreting commands both simple or multiple, with redirections and background. Moreover, two internal functions must be also implemented, both *mycalc* and *myhistory*.

We have organized this project following a set of steps:

1. Code for simple commands.
2. Code for multiple commands.
3. Redirections.
4. Internal commands.

## DESCRIPTION OF THE CODE

### **structure:**

Our program is structured the following way:

Inside while(1), the program checks the arguments that are introduced: first checks if it is an internal command (*myhistory* or *mycalc*), and if not, saves it in a structure called *history* and executes the command, checking first whether it is simple or multiple.

### **simple commands**

- **purpose:** execution of simple commands, simple commands in background (&) and with input and output error redirections.
- **code description:**

Firstly , if the number of commands is equal to 1 , the program will create a new child process using *Fork()*.

Secondly, it will redirect input, output and error streams. For this purpose, the code will check any of which is specified. Depending on which of them is specified , the program will execute a different process.

1. if there is an input *filev[0]*, the standard input is closed and the specified file for reading will be open.
2. if there is an output *filev[1]*, the standard output is closed and the specified file for writing is opened.If this file exists , it will truncate it. If it does not exist , the program will create the file
3. If there is an error *filev[2]*, the standard error is closed and the specified file for writing will be open. As stated before, if the file for writing exists it will truncate it, if not, the program will create it.

Lastly, the program will check which process of the fork () is being executed.

1. If *pid = -1*, this means an error has occurred with forking. An error message will be printed.

2. If `pid == 0` , is the child process. In this case, using `execvp()` the current process image will be replaced with a new one specified by that command. If an error occurred an error message will be also printed.
3. By default , it is the parent process. There are two possibilities.If it is not running in background, it will return the child ID (`pid`) and both processes will be executed at same time. On the other hand, it will wait for the child process to end in case it is not running in the background(&).

### multiple commands

- **purpose:** Execution of multiple commands with or without background, input , output and error redirections. It will mainly use *pipes* for connecting these multiple processes.
- **code description:**  
 Firstly, if the number of commands is greater than 1, this part of the code will be executed. Initially, it will redirect input , output and error streams, following the same structure and procedure as the one specified previously in simple commands.  
 Secondly, for each of the commands except the last one , it will create a pipe; `pipe()`. The pipes are created for each of the commands except the last one the output of the first command needs to be redirected to the input of the second command, the output of the second command to the input of the third, and so on, until the output of the N-1 command is redirected to the input of the Nth command. If then , a child process will be created using `fork()`.
  - child process: Nextly, the program will be focused on the child process. The standard input will be redirected using `dup2()`, to the *newinput*, which is updated to the read end of each pipe (except the last one). If it is already the last command it will execute the program using `execvp()`.
  - parent process: Here, unnecessary files will be closed and *newinput* will be updated to the read end of the pipe , so that the next iteration can be executed.

Lastly, if it is not running in background it will wait until the child process has ended.

When all commands have been executed, it will close all file descriptors that have been open while the redirections.

### internal commands

- **purpose:** We have developed two internal commands. The first one is '**mycalc**' , it performs calculations as addition, multiplication and division of two numbers in the command line, giving its result (for the division it shows the quotient and remainder)  
 The second internal command is '**myhistory**' , it works with a data structure or list called **history** that saves the commands when they entered in the command line during the current execution of our minishell. 'myhistory' will show the last 20 commands entered and if we passed a number N (that belongs to `[0,history.size)` ) as argument ( `myhistory N`) it will run the command that is stored in history at index N.
- **code description:**  
 First of all, after reading the command from command line and store it in `argv`.

We compare the first word entered in the command line, that is stored in `argvv[0][0]`, with 'mycalc' and 'myhistory' using the function `strcmp`. In the case where it is equal to mycalc or myhistory it will develop the specific internal command.

- When `argvv[0][0]` is neither mycalc nor myhistory, we have a regular command that needs to be stored in the structure described before called history.  
It first checks if the number of elements in history (**n\_elem**) is less than 20, in that case we have space in history to store our new command. We will check if `n_elem=0` to initialize both head and tail to index 0 (head: index of first element in history ; tail: index of last element in history) , then we will store our new command in history at the tail using the function **store\_command()**. Then it increments the tail to point to the next available space in history.  
If history has already 20 commands, we will eliminate the command stored in the head of history using the function **free\_command()** and then we will store the new command at the tail using **store\_command()**
- When `argvv[0][0]` is **mycalc** .  
First, it will check that we have the three needed arguments (stored in `argvv[0][1]`, `argvv[0][2]`, `argvv[0][3]` ) .  
If the second argument (`argvv[0][2]`) is 'add', 'mul' or 'div', it will perform the specified operation in the first and third arguments and it will update the environment variable **acc** with the result of the operation. Then it will print with standard error output the result of the operation and the updated value of acc.  
If the second argument is not valid, it will print an error message using the standard output. For the division operation, it will also give an error if the third argument is equal to 0.  
In the case where are not specified one of the three arguments, it also will print an error message.
- When `argvv[0][0]` is **myhistory** .  
First, it will check if there is any argument (it will be stored in `argvv[0][1]`).  
In the case where there is no argument, it will print the list of the last 20 commands executed. To print the list we have used a for loop that iterates over the commands stored in **history** , it will print the command index and then accessing to the elements of the pointed command it will print the arguments of the command , if there are input/output redirections or background execution and their details.  
To print the arguments of the command, it will use the elements 'argvv' and 'args' of the structure command.  
To indicate if there are input/output redirections, it will look at the element 'filev' of the structure command and compare its first/second argument with 0 using the function strcmp . In the case where it has input redirection (`strcmp(history[i].filev[0], "0") != 0`), it will print '<' and the file's name. In the case where it has output redirection (`strcmp(history[i].filev[1], "0") != 0`), it will print '>' and the file's name. And for the standard error (`strcmp(history[i].filev[2], "0") != 0`), it will print '!>' and the file's name.  
To indicate if there is background execution, it will print '&' if there is an element 'in\_background' in the current command.

When we have an argument, it will check if the argument is an integer number N within the range of executed commands (from 0 to n\_elem - 1). In the case where it is, it will print 'Running command N' and it will set the run\_history flag to 1, indicating that the shell should execute the command from the history in index N.

In the case where the argument is not valid, it prints an error message indicating that the command is out of range.

## TEST CASES

### 1- SIMPLE COMMANDS:

- With no redirections, nor background :  
minishell:

```
MSH>>ls
authors.txt      example2.txt  example.txt  file2.txt  file4.txt  grade.txt  Makefile  msh.c  os_p2_100496779_100496904_100496715.zip
checker_os_p2.sh example3.txt  file1.txt    file3.txt  file5.txt  libparser.so msh       msh.o
```

we can compare the output with the one of the shell:

```
a0496779@guernika:~/p2_minishell_2024$ ls
ls: no se puede acceder a 'example.txt': No existe el fichero o el directorio
authors.txt      example2.txt  example.txt  file2.txt  file4.txt  grade.txt  Makefile  msh.c  os_p2_100496779_100496904_100496715.zip
checker_os_p2.sh example3.txt  file1.txt    file3.txt  file5.txt  libparser.so msh       msh.o
```

- With background:

minishell:

```
MSH>>pwd &
Pid = [449907]
MSH>>/home/alumnos/a0496779/p2_minishell_2024
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ pwd &
[1] 450278
/home/alumnos/a0496779/p2_minishell_2024
```

- With input redirection:  
minishell:

```
MSH>>cat < authors.txt
100496779, Martin, Laura
100496904, Caldado, Lucia
100496715, Garcia, PaulaMSH>>
```

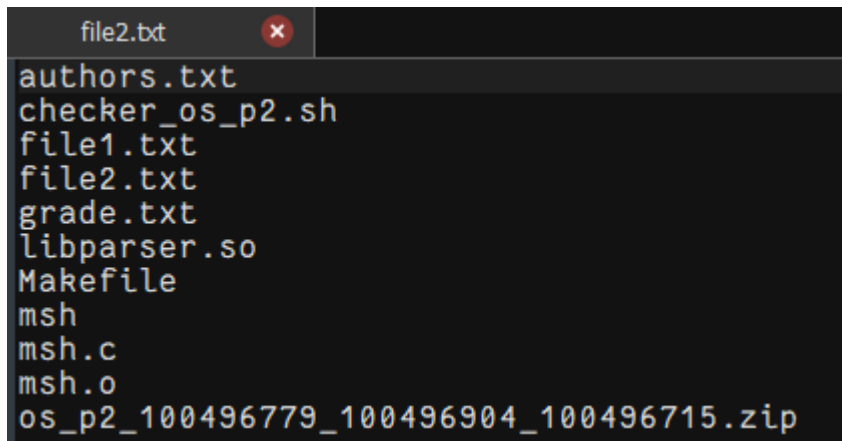
shell:

```
a0496779@guernika:~/p2_minishell_2024$ cat < authors.txt
100496779, Martin, Laura
100496904, Caldado, Lucia
100496715, Garcia, Paulaa0496779@guernika:~/p2_minishell_2024$
```

- With output redirection:  
minishell:

```
MSH>>ls > file2.txt
```

content of file2.txt:

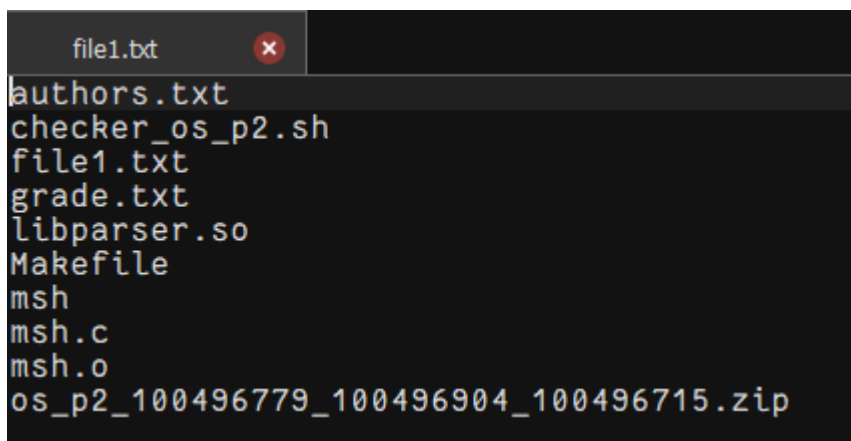


```
file2.txt
authors.txt
checker_os_p2.sh
file1.txt
file2.txt
grade.txt
libparser.so
Makefile
msh
msh.c
msh.o
os_p2_100496779_100496904_100496715.zip
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ ls > file1.txt
```

content of file1.txt:



```
file1.txt
authors.txt
checker_os_p2.sh
file1.txt
grade.txt
libparser.so
Makefile
msh
msh.c
msh.o
os_p2_100496779_100496904_100496715.zip
```

- With error redirection :  
minshell:

```
MSH>>wc nofile.txt !> file3.txt
```

(where nofile.txt does not exist)

content of file4.txt:

```
file3.txt x
MSH>>ofile.txt: No existe el fichero o el directorio
```

## 2- MULTIPLE COMMANDS:

- 2 commands with not background nor redirections:  
minishell:

```
MSH>>ls | wc
17      17      205
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ ls | wc
17      17      205
```

- more than 2 commands with no background nor redirections:

minishell:

```
MSH>>cat example2.txt | sort | wc
3        11       56
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ cat example2.txt | sort | wc
3        11       56
```

- with background:  
minishell:

```
MSH>>ls | wc &
Pid = [574479]
MSH>>      17      17      205
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ ls | wc &
[3] 574685
a0496779@guernika:~/p2_minishell_2024$      17      17      205
```

- with input redirection:  
minishell:

```
MSH>>cat | sort | wc < example2.txt
3        11       56
```

shell:

```
a0496779@guernika:~/p2_minishell_2024$ cat | sort | wc < example2.txt
3 11 56
```

- with output redirection:  
minishell:



```
MSH>>ls | sort | wc > output.txt
```

content of output.txt:

output.txt		
18	18	216

shell:

```
a0496779@guernika:~/p2_minishell_2024$ ls | sort | wc > output2.txt
```

content of output2.txt:

output2.txt		
19	19	228

- with error redirection:  
minishell:

```
MSH>>ls nofile.txt | sort !> errorfile.txt
```

(nofile.txt does not exist)

content of errorfile.txt:

```
errorfile.txt
ls: no se puede acceder a 'nofile.txt': No existe el fichero o el directorio
MSH>>
```

### 3 - MYCALC:

- addition:

```
MSH>>mycalc 2 add 3
[OK] 2 + 3 = 5; Acc 5
MSH>>mycalc 3 add -6
[OK] 3 + -6 = -3; Acc 2
```

- multiplication:

```
MSH>>mycalc 2 mul 6
[OK] 2 * 6 = 12
MSH>>mycalc -3 mul 7
[OK] -3 * 7 = -21
```

- division:

```
MSH>>mycalc 4 div 2
[OK] 4 / 2 = 2; Remainder 0
MSH>>mycalc 5 div 2
[OK] 5 / 2 = 2; Remainder 1
MSH>>mycalc 3 div 0
ERROR. Division by 0: Success
```

- error case not correct arguments:

```
MSH>>mycalc 4 mod 2
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
MSH>>mycalc 2 + 4
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
MSH>>mycalc * 5
[ERROR] The structure of the command is mycalc <operand 1> <add/mul/div> <operand 2>
```

#### 4 - MYHISTORY:

- list of commands:

```
MSH>>ls -las
total 248
0 drwx----- 2 a0496779 alumnos 0 abr 16 22:20 .
0 drwx----- 2 a0496779 alumnos 0 abr 16 20:19 ..
8 -rw-r--r-- 1 a0496779 alumnos 75 abr 16 16:01 authors.txt
16 -rwxr-xr-x 1 a0496779 alumnos 11843 feb 22 13:30 checker_os_p2.sh
8 -rwx----- 1 a0496779 alumnos 82 abr 16 22:20 errorfile.txt
8 -rwx----- 1 a0496779 alumnos 56 abr 16 21:15 example2.txt
8 -rwx----- 1 a0496779 alumnos 244 abr 16 21:04 example3.txt
8 -rwx----- 0 a0496779 alumnos 21 abr 16 21:00 example.txt
0 -rwx----- 0 a0496779 alumnos 0 abr 16 20:41 file1.txt
8 -rwx----- 0 a0496779 alumnos 137 abr 16 20:20 file2.txt
8 -rwx----- 1 a0496779 alumnos 53 abr 16 20:30 file3.txt
8 -rwx----- 1 a0496779 alumnos 75 abr 16 20:33 file4.txt
8 -rwx----- 0 a0496779 alumnos 39 abr 16 20:40 file5.txt
8 -rwx----- 1 a0496779 alumnos 6 abr 16 15:06 grade.txt
28 -rw-r--r-- 1 a0496779 alumnos 20696 feb 21 16:27 libparser.so
8 -rw-r--r-- 1 a0496779 alumnos 244 feb 21 16:27 Makefile
36 -rwx--x--x 1 a0496779 alumnos 29944 abr 16 16:15 msh
20 -rwx----- 1 a0496779 alumnos 14112 abr 16 16:15 msh.c
32 -rwx----- 1 a0496779 alumnos 27840 abr 16 16:15 msh.o
12 -rwx----- 1 a0496779 alumnos 4968 abr 16 15:06 os_p2_100496779_100496904_100496715.zip
8 -rwx----- 1 a0496779 alumnos 24 abr 16 21:40 output2.txt
8 -rwx----- 1 a0496779 alumnos 24 abr 16 21:39 output.txt

MSH>>pwd
/home/alumnos/a0496779/p2_minishell_2024
MSH>>date
mar 16 abr 2024 22:30:31 CEST
MSH>>ls
authors.txt      errorfile.txt  example3.txt  file1.txt  file3.txt  file5.txt  libparser.so  msh  msh.o  output2.txt
checker_os_p2.sh example2.txt  example.txt  file2.txt  file4.txt  grade.txt  Makefile      msh.c os_p2_100496779_100496904_100496715.zip output.txt
MSH>>ls | wc
 20      20      242
MSH>>cat authors.txt
100496779, Martin, Laura
100496904, Caldato, Lucia
100496715, Garcia, PaulaMSH>>myhistory
```

after inserting all these commands, output of myhistory is:

```
0 ls -las
1 pwd
2 date
3 ls
4 ls | wc
5 cat authors.txt
```

- execute a command (with previous list of commands):

```
MSH>>myhistory 1
Running command 1
/home/alumnos/a0496779/p2_minishell_2024
```

```
MSH>>myhistory 2
Running command 2
mar 16 abr 2024 22:33:30 CEST
```

```
MSH>>myhistory 5
Running command 5
100496779, Martin, Laura
100496904, Caldado, Lucia
100496715, Garcia, PaulaMSH>>
```

- error executing a commands out of the list:

```
MSH>>myhistory 7
ERROR: Command not found
```

- free command when there are more than 20 commands at the list:

in this example we have 20 commands in history:

```
MSH>>myhistory
0 ls -1
1 pwd
2 date
3 cat example2.txt
4 sort example2.txt
5 wc authors.txt
6 ls -las
7 cat authors.txt
8 date
9 pwd
10 ls -1
11 pwd
12 date
13 cat example2.txt
14 ls | wc
15 ls | sort | wc
16 cat file2.txt
17 mkdir file.txt
18 wc -1
19 sort authors.txt
```

after one more command:

```
MSH>>mkdir newfile.txt
```

now the output of myhistory replaces the first command by the last one introduced:

```
MSH>>myhistory
0 mkdir newfile.txt
1 pwd
2 date
3 cat example2.txt
4 sort example2.txt
5 wc authors.txt
6 ls -las
7 cat authors.txt
8 date
9 pwd
10 ls -1
11 pwd
12 date
13 cat example2.txt
14 ls | wc
15 ls | sort | wc
16 cat file2.txt
17 mkdir file.txt
18 wc -1
19 sort authors.txt
```

furthermore, if we try to execute command 20 (does not exist):

```
MSH>>myhistory 20
ERROR: Command not found
```

finally, observe that if we execute mycalc, this is not saved in history as it is not considered a normal command:

```
MSH>>mycalc 2 add 3
[OK] 2 + 3 = 5; Acc 5
MSH>>myhistory
0 mkdir newfile.txt
1 pwd
2 date
3 cat example2.txt
4 sort example2.txt
5 wc authors.txt
6 ls -las
7 cat authors.txt
8 date
9 pwd
10 ls -1
11 pwd
12 date
13 cat example2.txt
14 ls | wc
15 ls | sort | wc
16 cat file2.txt
17 mkdir file.txt
18 wc -1
19 sort authors.txt
```

# CONCLUSIONS

During the large process that has been writing all the code we have faced many problems.

Firstly, simple commands had no much complication, but commands with background did since at first we had to find more information about the meaning of executing a command in background.

Then we started with multiple commands, which was harder because we had to analyze the complexity of chained inputs and outputs, which involved the use of pipes in order to connect processes. Here we needed to check a lot of errors that were finally solved. In the same way, implementing multiple commands with background was again a hard task since we did not understand at first what pid of processes were needed to be printed.

Next, we added redirections to both simple and multiple commands, which was the easiest part. Finally, the implementation of internal commands was the hardest task, specially with *myhistory*.

Starting with *mycalc*, the difficult part was to understand the use of the environment variable "Acc", which we realized that it had to be initialized outside the main function in order to make the program work correctly. Moreover, we got confused with the standard error output and the standard output and mixed them, until we finally fixed it.

To end up, implementing *myhistory* was the most difficult part because we first needed to understand how the structure history worked, and how commands were saved there. Moreover, understanding how to free commands from history when more than 20 were reached was tricky also because we did not have clear what command we had to delete, finally we decided to delete the first one in the list. In order to list the commands, we had some problems regarding the function *print\_command*, since we thought we had to use it but it was incorrect, we just had to print the name of the command. And finally, executing a command with *myhistory* came as a huge problem, until we realized that using variable *run\_history* could make the program much more easy.

As a final conclusion, this lab has been much more demanding both technically and in terms of knowledge regarding the subject. It requires a vast amount of information and an excellent level of programming, involving various data structures and functions such as *fork*, *execvp*, etc., which we have not used before.