OPERATIVE SYSTEMS

GROUP 121

APPLIED MATHEMATICS AND COMPUTING

# Lab Report 02


| *Authors* | *NIA* |
| --- | --- |
| IGNACIO AGUADO | 100496991 |
| SOBRADILLO LUCAS GUZMÁN | 100496813 |
| BASTIDA OLAYA MARTÍNEZ FERNÁNDEZ | 100495748 |

*Emails*

100496991@alumnos.uc3m.es (Ignacio)
100496813@alumnos.uc3m.es (Lucas)
100495748@alumnos.uc3m.es (Olaya)

# Table of Contents

# 1 Description of the code

Before getting down to coding, we took a couple of days to really grasp the statement and the provided code. Then, for the minishell development, we pretty much followed the road map outlined in the statement, which proved quite helpful.

## 1.1 Minishell development

Step 1 (execution of simple commands) wasn't too complicated - it mainly involved setting up the `fork()` and running the command in the child using `execvp()`.

After that, step 2 (simple commands in the background) wasn't overly difficult either, but we did encounter a few hiccups initially. We tried a couple of different approaches before hitting with the solution. In the end, we used a conditional block `if(!in_background)` for the parent to wait for the foreground processes, and for the background processes simply print its *pid* and not wait so the shell keeps running. But, we needed to ensure that when the process finishes, its parent cleans its status so it does not become a zombie. For this, we implemented a signal handler for `SIGCHLD`, so that when the parent receives that signal it properly "kills" the process.

Then, sequences of commands (step 3) were probably the most difficult part, since we needed to handle pipes and re-directions very carefully. First, we separated the case of a simple command (`command_counter == 1`), where we don't want to create pipes, with the case of a command sequence. We started by implementing the execution of a sequence of 2 commands communicated by a pipe (`'|'`) and then tried to scale that to any number of commands. We've done this by putting our code in a `for` loop and in each iteration the parent creates a child for each command we want to execute. It also creates a pipe (except in the last iteration) to make communication between the children. The thing that gave us more problems was the appropriate closing of pipes, since we needed to be aware of what pipes the father has open at the moment of each `fork()` since the child is going to inherit them. We used a matrix `fd` to store the file descriptors of the pipes so that each process reads from `fd[i-1][READ_END]` and writes into the pipe `fd[i][WRITE_END]`, except the first and last commands that are treated separately.

Then to implement background execution to sequences of commands was quite straightforward since it was quite similar to the case of simple commands. We just make the father wait only if the process must be executed in the foreground. If not, the father does not wait. However, it's still aware of receiving

SIGCHILD for cleaning the status of the child so that it does not become a zombie. For the file redirections, we just put the redirection in case that filev[] had some nonzero entry, of course with the required error control if open does not work correctly. In case we need a redirection, we use dup2(). If it is for a simple command, we check all 3 possible redirections (input, output, and error), but for a sequence, we check redirections of input only in the first command, output only in the last one, and error in all commands. For the output and error redirection, we used the flags O_CREAT, O_WRONLY and O_TRUNC so that we open the file to write (or create it if needed) and we truncate its content. For the input, the only flag needed was O_RDONLY since we only want to read from the file.

## 1.2 Internal commands

For executing internal commands, we checked if argvv[0][0] was mycalc or myhistory so we don't use fork(), since they must be executed in the parent process. We did this in the case of command_counter was 1 because it cannot be executed in a sequence, as stated in the exercise.

### 1.2.1 mycalc

First, we count the number of arguments that the command has received. Then in case the number is different from 4 (3 arguments plus the name of the command), we print an error. We've also added to this conditional the cases when there is a trial to execute the command in the background or with file redirections.

Then, if everything is fine, we go to perform the operation. If the operand is not one of the valid ones, we print an error. But, if the operand coincides with any of the valid ones, we perform the corresponding operation. We needed to be extremely careful with where to show the result or error, and for that, we used fprintf() so that we could choose stderr for results or stdout for errors, as the statement required.

We also initialized the environment variable Acc to 0 out of the minishell loop, and each time we perform an addition, we recover its value with getenv() to print it and then update it correctly with its new value. We realized that environment variables ar stored as strings, and that is why we used the variable acc_value_str and transformed it to an integer only if getenv() worked properly, using a ternary operator. Additionally, we ensured to convert the integer back into a string after updating its value before setting it again into the environment.

### 1.2.2 myhistory

If no arguments are provided, the program prints out the entire command history. This is achieved by counting the number of arguments passed to `myhistory` and iterating through the history array to display each command.

If a numerical argument is provided after `myhistory`, it's interpreted as the index of a command in the history that the user wants to execute. The program retrieves the command at that index and executes it using `execvp()` after forking a new process.

If there's an attempt to redirect output or execute in the background with `myhistory`, the program outputs an error message because `myhistory` doesn't support these features.

Lastly, if the arguments are invalid or the number of arguments is incorrect, the program outputs an error message.

Regarding command history management, the program utilizes a circular queue technique to store commands. This ensures efficient storage and retrieval of recent commands while avoiding overflow.

The `commandToString()` function converts commands and their arguments into a readable string format. It handles tasks like appending pipe character "|" for multiple commands, indicating background execution with "&", and managing file redirection for input, output, and error streams.

# 2 Test Cases

## 2.1 Minishell

**Simple commands**

```
MSH>>ls
Makefile authors.txt checker_os_p2.sh libparser.so msh msh.c msh.o

MSH>>cat authors.txt
100496991, Aguado, Ignacio
100496813, Guzman, Lucas
100495748, Martinez, Olaya
```

**Commands that do not exist**

```
MSH>>la
Error in execvp: No such file or directory

MSH>>whoi
Error in execvp: No such file or directory
```

### Executing simple commands in background

```
MSH>>sleep 10 &
[48579]
MSH>>ls
Makefile  authors.txt  checker_os_p2.sh  libparser.so  msh  msh.c
   msh.o
MSH>>ps
    PID TTY          TIME CMD
    804 pts/6    00:00:00 bash
  45523 pts/6    00:00:00 msh
  48579 pts/6    00:00:00 sleep
  48645 pts/6    00:00:00 ps
```

The shell didn't block (we could run `ls`) but the process `sleep 10` was running (we can see it in the process list printed from `ps`).

### Command sequence

```
MSH>>ls -o | grep msh
-rwxrwxrwx 1 lucasguzmanb 23520 Apr  8 11:38 msh
-rwxrwxrwx 1 lucasguzmanb 10673 Apr  8 11:41 msh.c
-rwxrwxrwx 1 lucasguzmanb 18512 Apr  8 11:38 msh.o

MSH>>ls -o | grep h | grep Apr | grep t
-rwxrwxrwx 1 lucasguzmanb    78 Apr  8 12:02 authors.txt

MSH>>ls -o | grep h | grep Apr | grep t | wc
      1       9      70
```

### Command sequence in background

```
MSH>>ls -o | grep h | sleep 10 &
[68991]
MSH>>ls
Makefile  authors.txt  checker_os_p2.sh  libparser.so  msh  msh.c
   msh.o
MSH>>ps
    PID TTY          TIME CMD
    375 pts/0    00:00:00 bash
  68661 pts/0    00:00:00 msh
  68991 pts/0    00:00:00 sleep
  69016 pts/0    00:00:00 ps
```

The shell didn't block (we could run `ls`) but the process `sleep 10` was running (we can see it in the process list printed from `ps`). Also, the *pid* printed is `[68991]`, the one corresponding to `sleep` (last command)

### File redirection

```
MSH>>wc < authors.txt
 2  9 78
MSH>>ls
Makefile  authors.txt  checker_os_p2.sh  libparser.so  msh  msh.c
   msh.o
```

```
MSH>>ls -o | grep msh | grep o > test.txt
MSH>>ls
Makefile  authors.txt  checker_os_p2.sh  libparser.so  msh  msh.c
    msh.o  test.txt
MSH>>cat test.txt
-rwxrwxrwx 1 lucasguzmanb 22192 Apr  9 12:43 msh.o
MSH>>ls -o | greo msh !> test.txt
MSH>>cat test.txt
Error in execvp: No such file or directory
```

Works as expected: `wc` takes as input the content of authors.txt, then, the sequence `ls -o | grep msh | grep o > test.txt` creates the file test.txt (we can see it was not there from the previous `ls`). Then, with `cat`, we see that the output of the first sequence is correctly stored in test.txt. Then we introduce a sequence with a non-existing command, `greo`, which should raise an error that we redirect to test.txt. Again with `cat`, we see that the previous content of test.txt has been replaced by the error (the file must be truncated).

## 2.2 Internal Commands

### 2.2.1 mycalc

#### Normal execution

```
MSH>>mycalc 3 add 5
[OK] 3 + 5 = 8; Acc 8
MSH>>mycalc 8 mul 3
[OK] 8 * 3 = 24
MSH>>mycalc 2 add 1
[OK] 2 + 1 = 3; Acc 11
MSH>>mycalc 35 div 3
[OK] 35 / 3 = 11; Remainder 2
```

We can see that every operation is performed well and the `Acc` value is maintained correctly.

#### Wrong structure of the command

```
MSH>>mycalc 34 / 4
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc 78 mult 4
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc 2 add 4 add 5
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
```

#### Background execution or file redirection

```
MSH>>mycalc 3 add 3 &
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc 4 add 3 < file
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc 4 add 3 > file
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
MSH>>mycalc 4 add 3 !> file
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/
    div> <operand_2>
```

### 2.2.2 myhistory

#### Normal Execution

```
MSH>>myhistory
0 ls -o | grep msh
1 ls
2 echo hello
3 sleep 5 &
4 echo hello2 > file
5 cat file
MSH>>myhistory 1
Running command 1
Makefile  authors.txt  checker_os_p2.sh  file  libparser.so  msh  msh
    .c  msh.o
```

#### Wrong structure or illegal position

```
MSH>>myhistory 6
ERROR: Command not found
MSH>>myhistory 24
ERROR: Command not found
MSH>>myhistory 7 8
ERROR: Command not found
```

#### Background or file redirection

```
MSH>>myhistory 4 &
ERROR: Command not found
MSH>>myhistory 3 > file
ERROR: Command not found
```

# Conclusion

We encountered a few difficulties whilst coding the program.

First, we had an issue in the third step: the child processes did not finalize correctly as all the pipes were opened. As a solution, we created a conditional

for each case: the first, the last, and a middle command. For the first command, the program sets up the input direction if specified and writes output to the pipe. If the command is the last of the sequence, the program sets up the output direction if specified and reads input from the pipe. Lastly, if the command is in the middle of the sequence, the program reads input from the previous pipe and writes output to the next one.

The second issue that we had was the lack of an environment variable in `mycalc`. The solution was to use the function `setenv` for the `Acc` variable, and to be careful that environment variables are stored as strings.

Our final and one of the most complicated difficulties was the storage of the commands introduced in the minishell in the array `history`. To save the last 20 commands was not possible with the normal functionality of the array, as it only had access to 20 memory positions. To solve this issue, we implemented an array that follows a circular structure and works as a queue with a head and a tail. In this way, the array continues to be rewritten (once more than 20 numbers are being read), and as the numbers are being introduced in the array, the position of the head and tail advance one position simultaneously. Hence, when the process is finished there the numbers between the head and tail represent the last 20 and are saved in the array.

In the end, this project allowed us to be in touch with the main components that operate in a computer. Although we had already studied what is a shell, we did not have any idea about how it works or how to implement it, and this has changed with the development of this project. Same case with pipes, signals, or redirections. To sum up, it has been a valuable experience to properly learn some aspects of Operating Systems and to improve our abilities in problem-solving, whenever we encounter one.