

Universidad Carlos III de Madrid



OPERATING SYSTEMS

Bachelor's Degree in Computer Science & Engineering

Lab 2: Programming a shell (Minishell)

Academic Year 2023/2024

Inés Fuai Guillén Peña | 100495752 | 100495752@alumnos.uc3m.es | g89
Salvador Ayala Iglesias | 100495832 | 100495832@alumnos.uc3m.es | g89
Iván García Maestre | 100472753 | 100472753@alumnos.uc3m.es | g89

TABLE OF CONTENTS

Code description	2
<i>mycalc</i>	2
Test cases	3
<i>myhistory</i>	4
With no arguments	4
With arguments	5
Test cases	5
Simple commands	7
Simple commands	7
Sequence of commands	8
Test cases	9
Conclusion	13

Code description

The first check in order to know which command (if any) has to be executed is to check the variable `command_counter`. Only if it is higher than 0 we check further. If not, we pass, since no instruction was given.

In case it is higher than 0, we check it is lower than the maximum number of commands allowed (`MAX_COMMANDS`), in our case 8. If not, we print an error message indicating the max commands were exceeded.

If the amount of commands is within this values, we check what command must be executed: `mycalc`, `myhistory` or a simple command. In order to achieve this, we check the first argument (`argv[0][0]`).

For both `mycalc` and `myhistory`, as requested in the instructions, regular outputs are printed in the standard error output. Errors are printed in the standard output, using the function `"fprintf"` and as a first argument `"stderr"`, since `"perror"` function prints extra information that causes errors while running the checker.

1. Mycalc

If `mycalc` is the command executed, the first check to do is the format. Therefore, there should be at least three arguments (`argvv[0][1/2/3]`) after `"mycalc"`. If there are not three arguments, we print the error message described in the instructions.

If there are at least three arguments, we check the operator is a correct one: `add`, `mul` or `div`. If not, we print the error message. Otherwise, we can execute the proper operations.

Both operators are converted to integers using the `"atoi"` function, which does not provide error control. Therefore, a string without any digit (f.e. `"hola"`) is converted to a 0. If there are digits before a regular character (f.e. `"12hola3"`), the received value is just those digits (for this example, just 12). This behavior is intended, although it could be changed to recognize if the value is composed of just integers or also characters.

For `add`, we create and set the environment variable `"Acc"` requested to 0 initially. We retrieve the last accumulated value, calculate the new value (using the current result) and overwrite the variable with the new value. We print the result on screen. If while setting the environment variable we get an error message, we print it using `perror` and return -1, since it is an error while doing a system call.

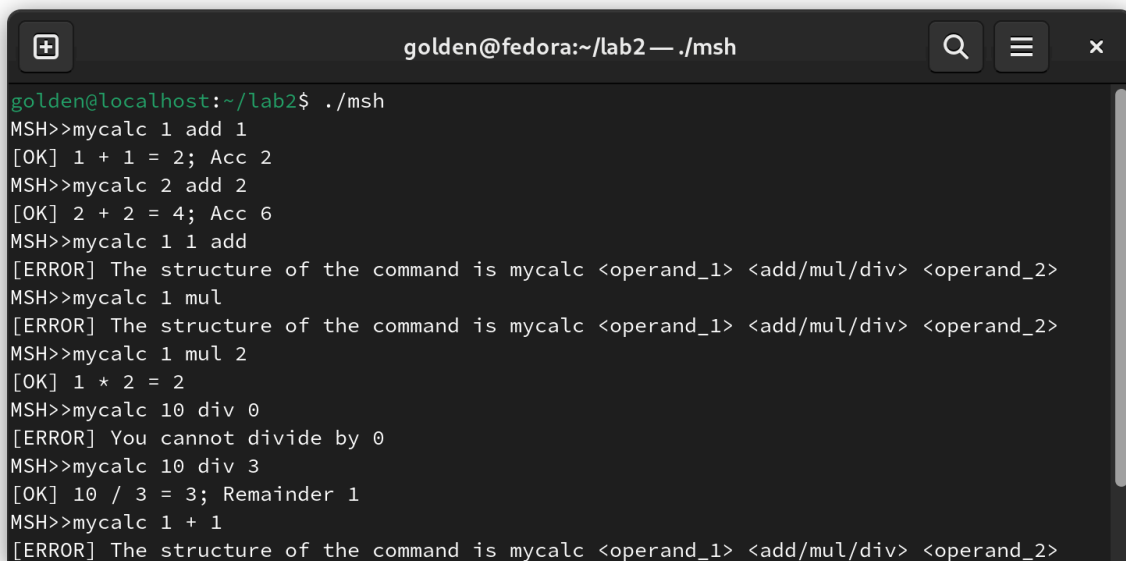
For mul, we just retrieve the values and multiply them. We print the obtained value.

For div, we have to check if the second operator is 0. For that case, we print an error as specified in the instructions. If not, we execute the operation and print both the result and the remainder, as requested.

Test Cases

Several executions were made in order to check the proper behavior of the function:

- Simple sum, check accumulated sum updates properly
- Another sum, check the value changes accordingly
- Incorrect structure, prints error
- Not enough arguments, prints error
- Correct multiplication
- Division by 0, prints special error
- Correct division
- Incorrect operand, prints error



```
golden@fedora:~/lab2 — ./msh
golden@localhost:~/lab2$ ./msh
MSH>>mycalc 1 add 1
[OK] 1 + 1 = 2; Acc 2
MSH>>mycalc 2 add 2
[OK] 2 + 2 = 4; Acc 6
MSH>>mycalc 1 1 add
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
MSH>>mycalc 1 mul
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
MSH>>mycalc 1 mul 2
[OK] 1 * 2 = 2
MSH>>mycalc 10 div 0
[ERROR] You cannot divide by 0
MSH>>mycalc 10 div 3
[OK] 10 / 3 = 3; Remainder 1
MSH>>mycalc 1 + 1
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
```

2. Myhistory

If myhistory is the command executed, we have to check if there is another argument or not (`argvv[0][1] == NULL`), since the behavior changes.

Myhistory with no arguments

If there is no argument, we have to print the full history, up to 20 commands. For this purpose, we have to previously save them in the history, an array of commands (structs). Therefore, everytime we have to execute a simple command (different to mycalc and myhistory) we save it in the history before its execution.

For the management of the history, we are given some variables: head, tail and n_elem. Head will save the position of the first command saved (initially 0, when the history gets full, it will advance positions to match the position of the oldest saved command). Tail will save the position of the last command saved. N_elem will save the number of commands stored, from 0 to 20 (when the history is full).

Using these variables, we will start filling the history (each time we execute a simple command), using the given function “store_command”, increasing both n_elem and tail until they reach 20. Once they reach that value, the history is full, therefore, the tail is set to 0 (first element of the history), so in the next execution of a simple command, the command stored in that position will be replaced with the new command. This time, we have to advance both head and tail to point to the oldest command saved, but keep n_elem at 20, since the number of commands will not decrease.

Once the history contains at least one command, executing myhistory without any parameter will print on screen the full history of simple commands entered previously. It will iterate through all elements of the history that are not null. For each element it will iterate through each command and for each command through all arguments. It will also check if there was a file redirection or if the command was executed on background and print it accordingly.

Myhistory with arguments

If there is an argument, we will convert it to integer using `atoi`. As explained before, this function does not control if the value is an integer or not, and this behavior is intended.

If the given argument is not within 0 and `n_elem` (number of commands stored in the history at the moment), an error message will be printed as requested.

If it is within the range, we will execute the proper command by copying the values of the command stored in the history (requested position in history) to the variable `argvv`, as if it was executed by the user. By changing the value of `argvv`, the first argument is also changed, so the shell will enter in the simple command execution part. Before that, a message is printed on screen to indicate that the command will be executed. Also, the variable `run_history` is changed to 1 in order to avoid the saving of this command in the history, since it was executed from the history. It will be set to 0 on the next execution.

Test Cases

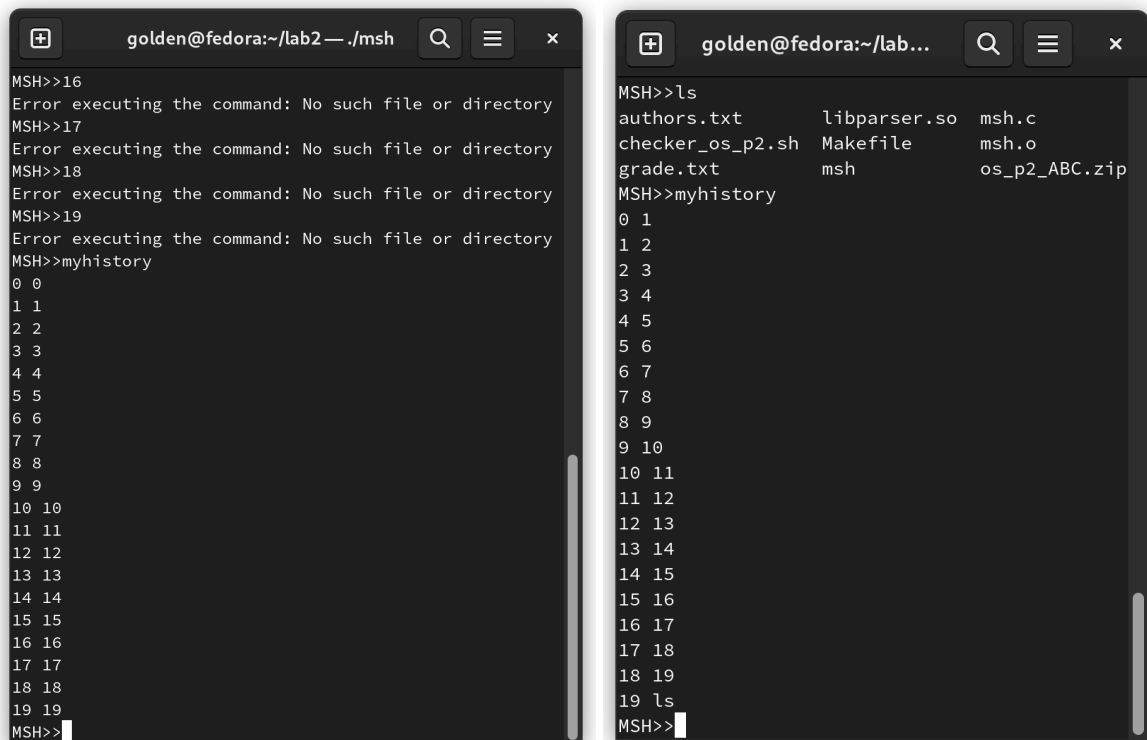
Several cases simulated:

Commands save in order and can be printed properly when executing `myhistory` with no arguments. Incorrect commands are also saved, as expected. `Myhistory` execution is not saved in history.



```
golden@localhost:~/lab2$ ./msh
MSH>>ls
authors.txt      grade.txt      Makefile      msh.c      os_p2_ABC.zip
checker_os_p2.sh libparser.so   msh           msh.o
MSH>>ls | grep m | sort
msh
msh.c
msh.o
MSH>>myhistory
0 ls
1 ls | grep m | sort
MSH>>abc
Error executing the command: No such file or directory
MSH>>myhistory
0 ls
1 ls | grep m | sort
2 abc
MSH>>
```

After the history is full, replace the oldest command and print properly:

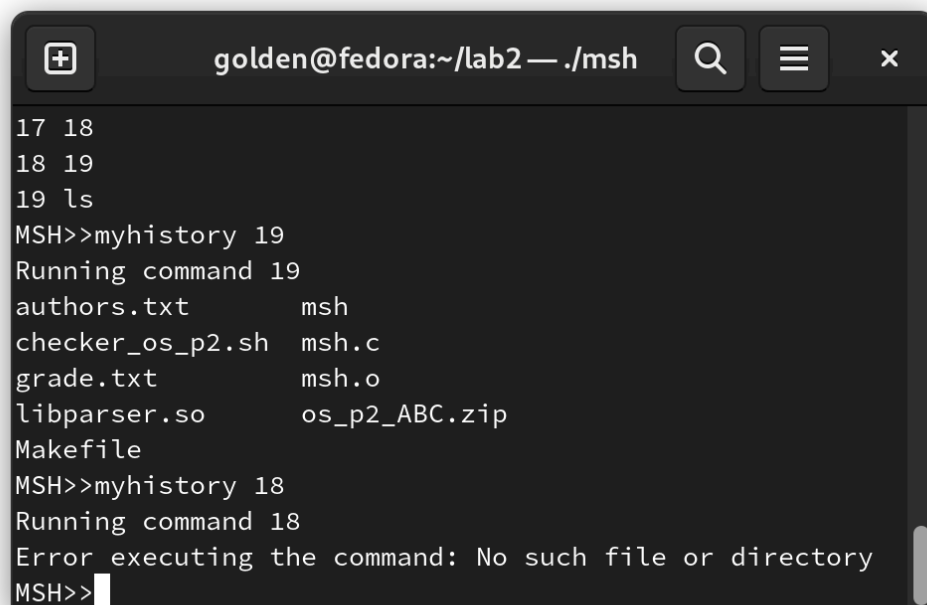


The image shows two terminal windows side-by-side. The left window shows a sequence of commands from 16 to 19, each followed by an error message: "Error executing the command: No such file or directory". The right window shows the same sequence of commands, but with the output of the 'ls' command at index 19, which lists files in the current directory: authors.txt, checker_os_p2.sh, grade.txt, libparser.so, Makefile, msh, msh.c, msh.o, and os_p2_ABC.zip.

```
golden@fedora:~/lab2 — ./msh
MSH>>16
Error executing the command: No such file or directory
MSH>>17
Error executing the command: No such file or directory
MSH>>18
Error executing the command: No such file or directory
MSH>>19
Error executing the command: No such file or directory
MSH>>myhistory
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
MSH>>

golden@fedora:~/lab...
MSH>>ls
authors.txt      libparser.so    msh.c
checker_os_p2.sh Makefile        msh.o
grade.txt        msh             os_p2_ABC.zip
MSH>>myhistory
0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
19 ls
MSH>>
```

Execution of commands when the history has been filled. Ls works properly while 19 does not (not a valid command):



The image shows a terminal window with the command history from index 17 to 19. The user then enters 'myhistory 19', which displays the output of the 'ls' command. Next, the user enters 'myhistory 18', which results in an error message: "Error executing the command: No such file or directory".

```
golden@fedora:~/lab2 — ./msh
17 18
18 19
19 ls
MSH>>myhistory 19
Running command 19
authors.txt      msh
checker_os_p2.sh msh.c
grade.txt        msh.o
libparser.so     os_p2_ABC.zip
Makefile
MSH>>myhistory 18
Running command 18
Error executing the command: No such file or directory
MSH>>
```

Execution of simple and composed commands from history:

```
golden@fedora:~/lab2 — ...
golden@localhost:~/lab2$ ./msh
MSH>>ls
authors.txt      libparser.so    msh.c
checker_os_p2.sh Makefile        msh.o
grade.txt        msh             os_p2_ABC.zip
MSH>>ls | sort
authors.txt
checker_os_p2.sh
grade.txt
libparser.so
Makefile
msh
msh.c
msh.o
os_p2_ABC.zip
MSH>>ls | sort | grep m
msh
msh.c
msh.o

MSH>>myhistory 0
Running command 0
authors.txt      libparser.so    msh.c
checker_os_p2.sh Makefile        msh.o
grade.txt        msh             os_p2_ABC.zip
MSH>>myhistory 1
Running command 1
authors.txt
checker_os_p2.sh
grade.txt
libparser.so
Makefile
msh
msh.c
msh.o
os_p2_ABC.zip
MSH>>myhistory 2
Running command 2
msh
msh.c
msh.o
MSH>>
```

3. Simple commands

For the execution of simple commands, first we have to check if the command was run from the history or not using the variable `run_history` (as explained previously). If it is not run from history, we will have to store it on the history (`store_command`). If the history is full, we first free the command memory (`free_command`) and then store the new one.

After that, we have to check if it is a simple command or a sequence of commands. That is done by checking the variable `command_counter` (1 for simple, higher for a sequence).

Simple command

If it is a simple command, we need to duplicate the process (`fork`) and execute the command in the child, while the father waits for the finalization of the child process, cleaning other processes while waiting. If it is asked to run on background, it will print the child pid and continue without waiting.. For the child process will just need to get the command using the `getCompleteCommand` function and execute it using `execvp`. The next line raises an error, which should not be run if the `execvp` worked properly.

If any file is given for the redirection of input (0), output (1) or error output (2) , `filev[i]` should contain anything different than "0". If that condition is met, we do a simple redirection by closing the standard file descriptor (`input=0`, `output=1`, `errors=2`) and opening the file (with `open` function). By doing this, the input/output/errors are

redirected to the respective files, meaning the file descriptor of the opened file now should match the closed file descriptor (0, 1 or 2). This is checked, and if it does not match, an error message is displayed and we return -1.

Sequence of commands

If it is a sequence of commands, we need to use pipes to connect the output of a command with the input of the next one. Only special cases are the ones described below:

- First command should not redirect its input unless an input redirection is requested.
- Last command should not redirect its output unless an output redirection is requested.
- Error redirection, if requested, is done in the parent process, so all child processes (all commands in the sequence) inherit this redirection, but the standard error output is later restored after the execution of the command is done.
- Parent process will wait for the termination of the last command of the sequence (last child), terminating other finished processes while waiting. If it is asked to run on background, it will print the pid of the last process, the one corresponding to the last command of the sequence.

The implementation of a sequence of commands done by our group does not have a maximum command limit. It allows any amount of commands in a single sequence.

Before forking, we create a pipe. If we are on an even iteration (starting at 0), we use “pip1” array to store the file descriptor of writing and reading ends of the new pipe. If we are on an odd iteration, we use “pip2”. This can be done since for each iteration (distinguishing by even or odd iterations), the respective pipe is used in the child process and closed in the parent process. Therefore, when we create a pipe using the array, that pipe has already been used or closed fully. We don’t create a new pipe if it is the last iteration, since output will go to std_out.

Once the pipe is created, we fork.

For the child processes, we have 3 cases:

1. First iteration (child). We only redirect the output to pip1 write end, applying the learnt process (close std_out fd, duplicate pipe write end, close pipe end once it has been assigned properly). We closed the unused read end. In case we had an

input redirection, we redirect the `std_in` to that file in the same way as we explained in the simple command code description.

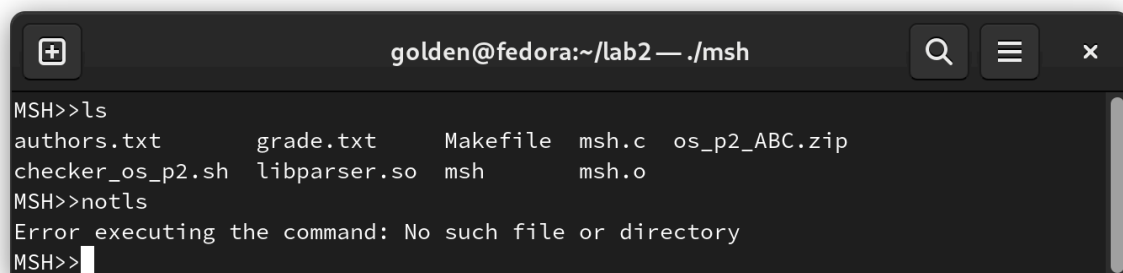
2. Intermediate iterations (children). Distinguishing between even and odd iterations, we redirect input and output to pipes.
 - a. For even iterations, `std_output` is redirected to the writing end of `pip1` and `std_in` is redirected to `pip2` reading end. Before that, we close the `pip1` reading end, since it is not used here. `Pip2` writing end was closed on the parent process in the last iteration.
 - b. For odd iterations, we redirect `std_out` to `pip2` writing end and `std_in` to `pip1` reading end. Before that, we close `pip2` reading end, since it is not used here. `Pip1` writing end was closed on the parent process in the last iteration.
3. Final iteration (child). We only redirect the input to the last created pipe reading end, differing for even and odd iterations. We don't need to close the writing end since it was closed on the parent process' last iteration.

For the parent process, after we have done the fork, we have to close the pipe ends used by the child process of that iteration. That means, for even iterations, close `pip1` write end and `pip2` read end. For odd iterations, close `pip2` write end and `pip1` read end. If we are at the first iteration, we only close the write end. If we are at the last iteration, we only close the read end (no other pipe was created).

Additionally, if we are on the last iteration, the parent process should either wait for the last child process to finish or print its pid and continue without waiting (if requested so). Also, if an error redirection was requested, we redirect the error output to its original value (stored at the start).

Test Cases

Simple command: existing and not existing.



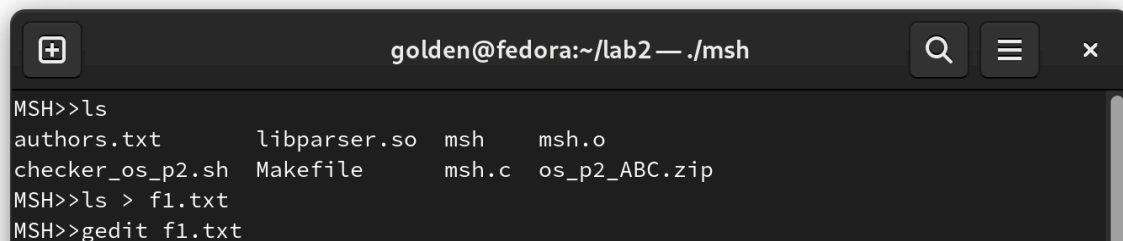
```
golden@fedora:~/lab2 — ./msh
MSH>>ls
authors.txt      grade.txt      Makefile      msh.c      os_p2_ABC.zip
checker_os_p2.sh libparser.so  msh           msh.o
MSH>>notls
Error executing the command: No such file or directory
MSH>>
```

Sequences of commands: 2, 3 4 and 5 command sequence.



```
golden@fedora:~/lab2 — ./msh
MSH>>ls | grep m
msh
msh.c
msh.o
MSH>>ls | sort | grep m
msh
msh.c
msh.o
MSH>>ls | sort | grep m | grep h.c
msh.c
MSH>>ls | sort | grep s | grep m | grep h.c
msh.c
MSH>>
```

Redirection of output:



```
golden@fedora:~/lab2 — ./msh
MSH>>ls
authors.txt      libparser.so  msh      msh.o
checker_os_p2.sh Makefile      msh.c    os_p2_ABC.zip
MSH>>ls > f1.txt
MSH>>gedit f1.txt
```



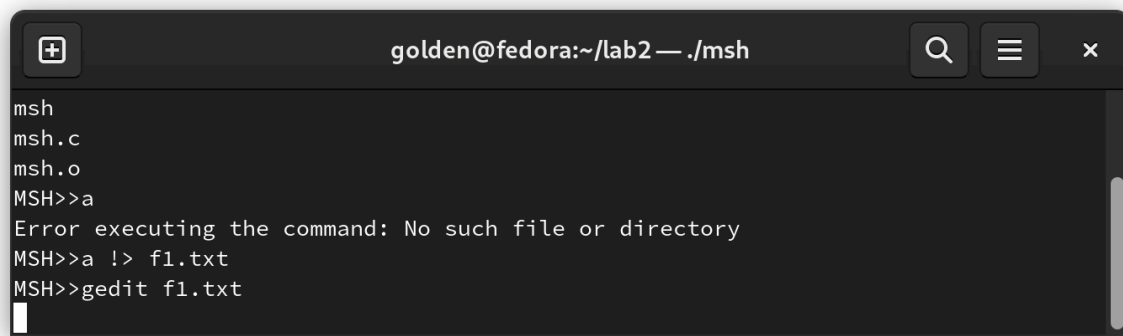
```
f1.txt
~/lab2
1 authors.txt
2 checker_os_p2.sh
3 f1.txt
4 libparser.so
5 Makefile
6 msh
7 msh.c
8 msh.o
9 os_p2_ABC.zip
Plain Text Tab Width: 8 Ln 1, Col 1 INS
```

Redirection of input:




```
golden@fedora:~/lab2 — ./msh
MSH>>ls
authors.txt      f1.txt      Makefile  msh.c  os_p2_ABC.zip
checker_os_p2.sh libparser.so msh      msh.o
MSH>>grep m < f1.txt
msh
msh.c
msh.o
MSH>>
```

Redirection of errors:



```
golden@fedora:~/lab2 — ./msh
msh
msh.c
msh.o
MSH>>a
Error executing the command: No such file or directory
MSH>>a !> f1.txt
MSH>>gedit f1.txt
```



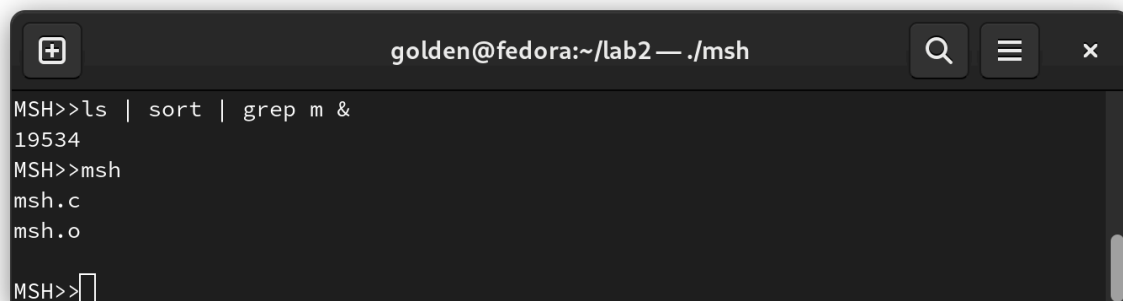
```
f1.txt
~/lab2
1 |Error executing the command: No such file or directory
Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 INS
```

Run on background:



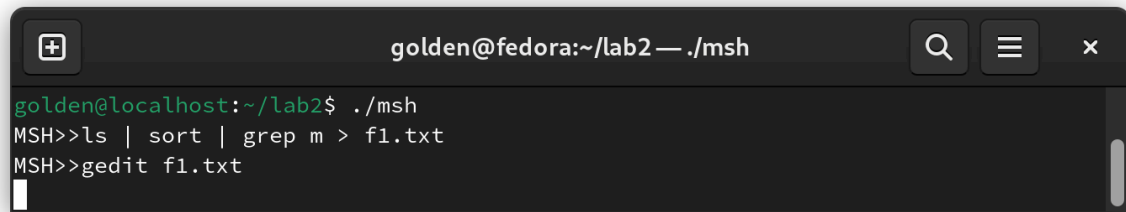
```
golden@fedora:~/lab2 — ./msh
Error executing the command: No such file or directory
MSH>>a !> f1.txt
MSH>>gedit f1.txt
MSH>>ls &
19502
MSH>>authors.txt          f1.txt          Makefile  msh.c  os_p2_ABC.zip
checker_os_p2.sh  libparser.so  msh          msh.o
```

Run on background a sequence of commands:

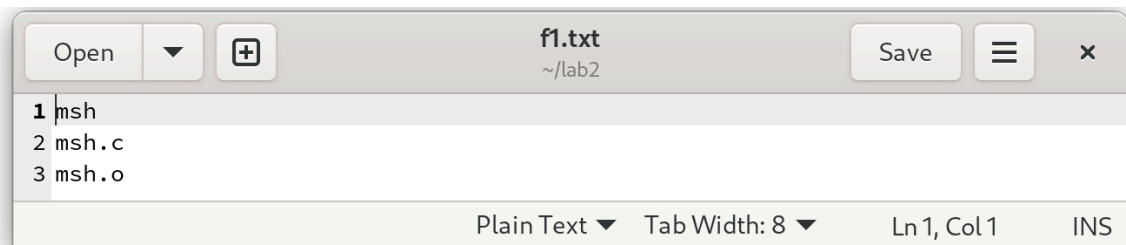


```
golden@fedora:~/lab2 — ./msh
MSH>>ls | sort | grep m &
19534
MSH>>msh
msh.c
msh.o
MSH>>
```

Sequence of commands output redirection:

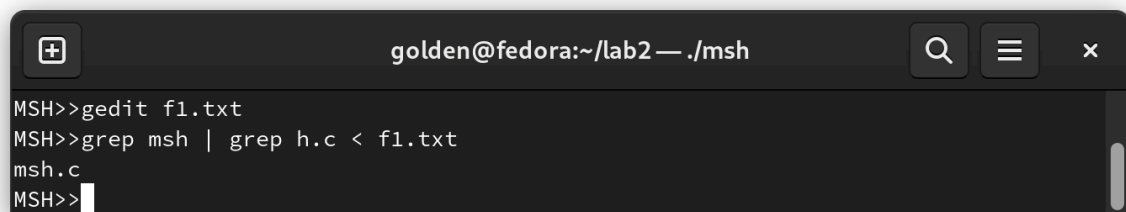


```
golden@fedora:~/lab2 — ./msh
golden@localhost:~/lab2$ ./msh
MSH>>ls | sort | grep m > f1.txt
MSH>>gedit f1.txt
```



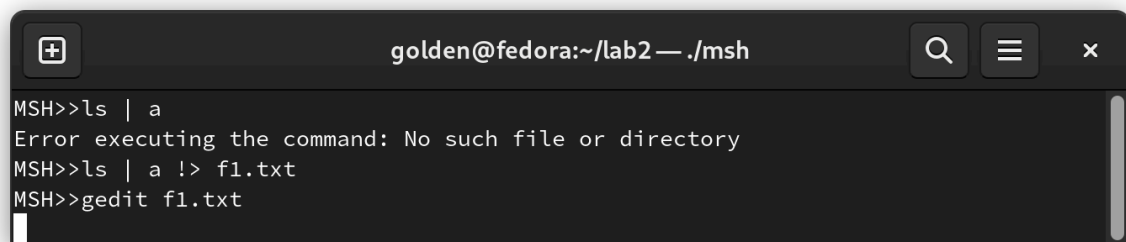
```
Open ▼ + f1.txt ~/lab2 Save ≡ ×
1 msh
2 msh.c
3 msh.o
Plain Text ▼ Tab Width: 8 ▼ Ln 1, Col 1 INS
```

Sequence of commands input redirection:



```
golden@fedora:~/lab2 — ./msh
MSH>>gedit f1.txt
MSH>>grep msh | grep h.c < f1.txt
msh.c
MSH>>
```

Sequence of commands error redirection:



```
golden@fedora:~/lab2 — ./msh
MSH>>ls | a
Error executing the command: No such file or directory
MSH>>ls | a !> f1.txt
MSH>>gedit f1.txt
```



```
Open ▼ + f1.txt ~/lab2 Save ≡ ×
1 Error executing the command: No such file or directory
Plain Text ▼ Tab Width: 8 ▼ Ln 1, Col 1 INS
```

Conclusion

The main problems encountered during the development of the laboratory were related to pipes. Forgetting to close a pipe end file descriptor caused some trouble and some rethinking had to be done in order to achieve the proper behavior of the sequence of commands. Redirection of input/outputs to files was not complicated since the procedure is simple and identical each of the times.

The voluntary implementation of a sequence of commands of any length was the most difficult task, since being able to properly close all pipes resulted in some trouble. However, with enough time to think about the proper approach, the implementation worked properly.

The implementation of myhistory execution of commands also gave some trouble since we had to extract all parameters from the history and insert them in the proper variables in order to achieve the execution of the commands. However, after some fixes, it worked properly.

When we executed the checker script, we found some trouble, since the messages to be printed in mycalc and myhistory were different to the expected. After fixing that, we got the perfect grade, verifying the correctness of our code.

In conclusion, this practice helped us understand the internal behavior of the shell, how to use pipes and how to work with structs. We consider this laboratory is essential for our knowledge gain and we feel more confident with the course after practicing with this lab over the last week.