



OPERATING SYSTEMS

Laboratory 2. Programming a shell script interpreter



Mazin Bersy	100558650	100558650@alumnos.uc3m.es
Malek Mahmoud	100558649	100558649@alumnos.uc3m.es
Laila Sayed	100558440	100558440@alumnos.uc3m.es

Table of Contents

Description of the code:	2
1. Scripter	2
2. Mygrep:	3
Testing:	4
Scripter file test:	4
Test case 1:	4
Test case 2:	5
Test case 3:	5
Test case 4:	5
Test case 5:	6
Mygrep file test:	7
Test case 1	7
Test case 2	8
Test case 3	8
Test case 4	8
Test case 5	8
Test case 6	9
Conclusion	10

Description of the code:

1. Scripter:

The code has five functions and the main one.

1. The first function is **“read_line”**: this function is mainly for reading one character from the file each time and ensures the capacity limit of the reading to leave a space for the null termination by sitting a while loop that reads as long as the bytes read don't exceed max_len -1.

The reading terminates when a new line is reached “\n” and the string is terminated using null and returns the numbers of bytes read.

If there is no new line and some data are read, the program terminates and returns the number of bytes read.

2. The second function is **“tokenizer_linea”**: this function splits each line into tokens based on the given delimiters and stores the tokens in an array. The number of tokens stored are limited by the max_tokens -1 size in the while loop to ensure a space for the null terminator.
3. The third function is **“procesar_redirecciones”**: this function initializes the redirection file list by checking for “<,>,!>” and redirecting the file based on the sing. If “<” is found file [0] is redirected and set to the next argument. If “>” is found, file [1] is being redirected instead. And finally, if “!>” is found, file [2] is set to the next argument.
4. The fourth function is **“execute_command”**: This is one of the most important functions in the program since it creates the child process and handles the fork failure. This function has five cases executed through if statement.
 - Case 1:** If input redirection occurs, it duplicates in_fd to STDIN_FILENO, then closes the original file descriptor.
 - Case 2:** When output is redirected, it copies out_fd to STDOUT_FILENO and then closes the original file descriptor.
 - Case 3:** If an input file is specified, it opens the file and directs it to STDIN_FILENO.
 - Case 4:** If an output file is specified, it opens the file and directs it to STDOUT_FILENO.
 - Case 5:** If an error file is specified, it redirects STDERR_FILENO.

Finally, if the execution fails it prints an error message and exits the program.

5. The fifth function is **“procesar_linea”**: It splits the line into commands for pipelining and initializes file descriptors for pipes. It processes redirections and creates pipes for

the next command, executes it and redirects the input/output. Finally, it closes the write-end of the pipe and passes the read-end to the next command.

Finally, the main program function **main()**: This function opens a script file and runs each command line by line, validates the script format, and invokes `procesar_linea` for every line.

2. Mygrep:

The mygrep c program has mainly two parts: The `search_file` function and the **main ()** function.

Search_file function: This function takes a file path and search for a specific term as an input and prints the line numbers and lines content that has this term. The function uses **open ()** to open the file and if there is an error opening the file it prints an error message and exits the program.

- The function has four main variables, **buffer** for reading parts of the file, **line** to store the current line being read, **line_index** acts as a pointer on the current line being processed and **current_line** to keep track of the line numbers.
- A for loop is used to loop over all characters in the buffer, and “\n” is used for marking the end of a line and starting a new one. If there is no new line, the function continues storing characters in the variable **line**.
- Then the **strstr ()** checks if the search term is in the current line or not, if yes it prints the line and its numbers.

The **main ()** function:

- It has two arguments passed to it: **argc** is the number of arguments provided and **argv** is an array of argument strings.
- The function checks if the correct number of arguments is provided (**argc != 3**), if not, it prints -1 error and exits.
- The main function also calls the **search_file()** function with the file path and search term provided by the user.

Testing:**Scripter file test:**

Test case 1:

This test cases aims to verify that commands with & run in the background.

```
## Script de SSOO
```

```
echo "Foreground 1"
```

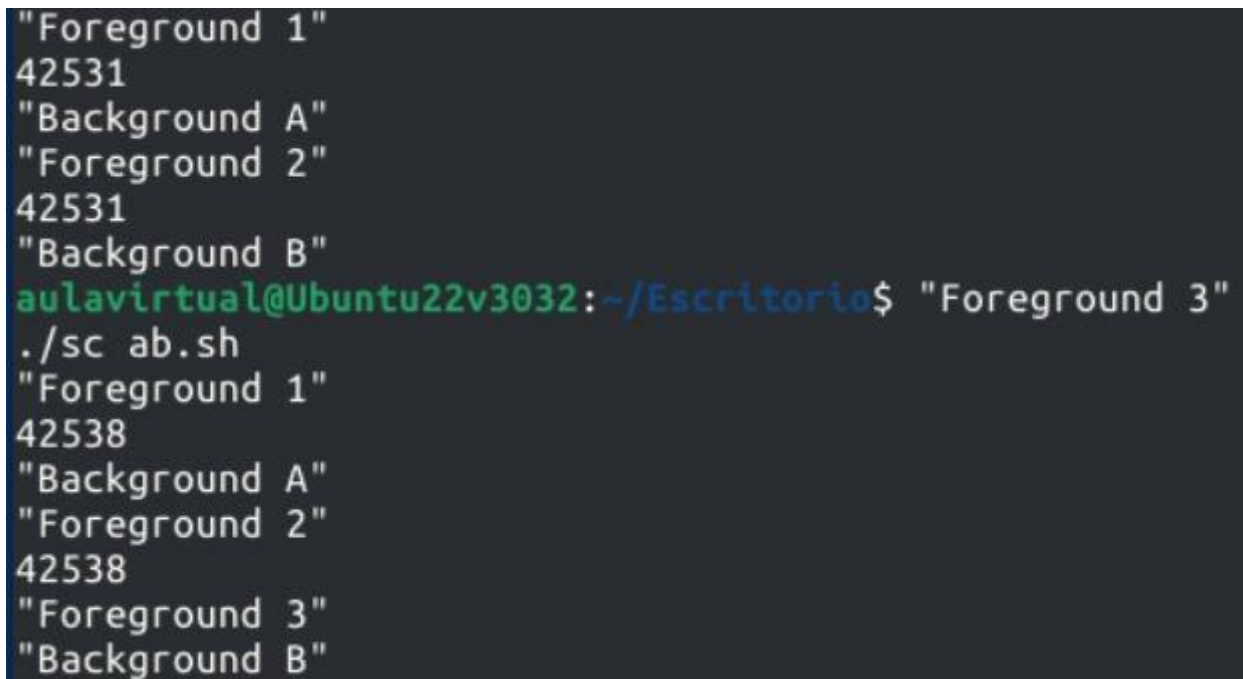
```
echo "Background A" &
```

```
echo "Foreground 2"
```

```
sleep 1
```

```
echo "Background B" &
```

```
echo "Foreground 3"
```

Output:

```
"Foreground 1"
42531
"Background A"
"Foreground 2"
42531
"Background B"
aulavirtual@Ubuntu22v3032:~/Escritorio$ "Foreground 3"
./sc ab.sh
"Foreground 1"
42538
"Background A"
"Foreground 2"
42538
"Foreground 3"
"Background B"
```

As seen above, in the second output foreground 3 is printed before background B due to them running simultaneously.

Test case 2:

The purpose of this test case is to see how the interpreter deals with non-existing files

Script de SSOO

cat < non_existent_file.txt

sort non_existent_file.txt > sorted_output.txt

```
aulavirtual@Ubuntu22v3032:~/Escritorio$ ./sc ab.sh
Input redirection failed: No such file or directory
sort: no se puede leer: non_existent_file.txt: No existe el archivo o el directorio
```

Test case 3:

This test case checks if an error is returned when ## Script de SSOO is not the first line

ls

cat foo.txt | grep a | grep 1

sort < foo.txt > sort.out

```
aulavirtual@Ubuntu22v3032:~/Escritorio$ ./sc ab.sh
Invalid script format: Success
```

Test case 4:

The goal of this test case is to verify that an error is returned, and the program terminates when an empty line is detected.

Script de SSOO

ls

cat foo.txt | grep a | grep 1

sort < foo.txt > sort.out

```

aulavirtual@Ubuntu22v3032:~/Escritorio$ gcc scripter.c -o sc
aulavirtual@Ubuntu22v3032:~/Escritorio$ ./sc ab.sh
ab.sh          mygrep1-medium-scripter.out  sort-medium-scripter.out
a.txt          mygrep.c                sort.out
error1-medium-scripter.err  sc                Tester
foo.txt        scripter.c
Makefile        sorted_output.txt
10a
Empty line detected: Success

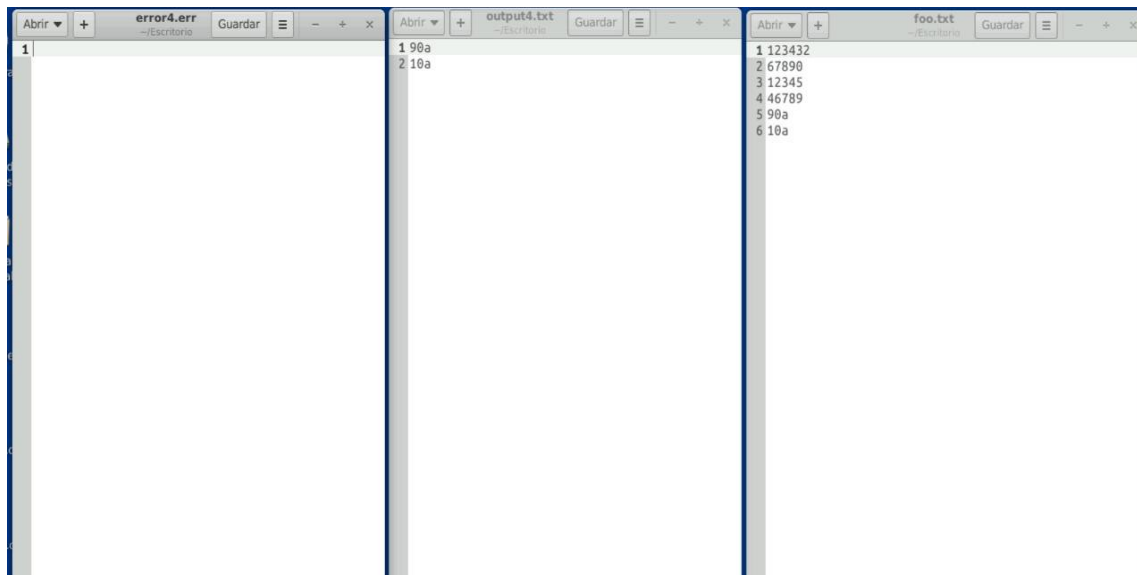
```

Test case 5:

This test case solidifies the working of the three redirections.

Script de SSOO

`grep "pattern" < input.txt > output4.txt 2> error4.log &`



Mygrep file test:

This test cases aims to verify that commands with & run in the background.

Script 1

echo "Hello, this is a test file.

The quick brown fox jumps over the lazy dog.

The word \"test\" appears multiple times in this file.

End of the file." > test_file_1.txt

Script 2

echo "This is another test file.

It has some content but does not contain the search term.

Just a few more lines of text here.

Nothing to find here." > test_file_2.txt

Script 3

echo "The only line with a matching search term: apple." > test_file_3.txt

```
aulavirtual@Linuxv3034:~/Música$ echo "Hello, this is a test file.
> The quick brown fox jumps over the lazy dog.
> The word \"test\" appears multiple times in this file.
> End of the file." > test_file_1.txt
```

```
aulavirtual@Linuxv3034:~/Música$ echo "This is another test file.
> It has some content but does not contain the search term.
> Just a few more lines of text here.
> Nothing to find here." > test_file_2.txt
aulavirtual@Linuxv3034:~/Música$
aulavirtual@Linuxv3034:~/Música$ echo "The only line with a matching search term
: apple." > test file 3.txt
```

Test case 1:

Searching for the word **test** in test_file_1.txt

output:

```
End of the file." > test_file_1.txt
aulavirtual@Linuxv3034:~/Música$ ./mygrep test_file_1.txt test
1: Hello, this is a test file.
3: The word "test" appears multiple times in this file.
aulavirtual@Linuxv3034:~/Música$ █
```


Test case 2:

Searching for the word **banana** in test_file_2.txt

Output:

```
aulavirtual@Linuxv3034:~/Música$ ./mygrep test_file_2.txt banana
aulavirtual@Linuxv3034:~/Música$ ./mygrep test_file_3.txt apple
```

We can see that the output is no output since the word banana is not in test_file_2.txt

Test case 3:

Searching for the term **apple** in test_file_3.txt

Output:

```
aulavirtual@Linuxv3034:~/Música$ ./mygrep test_file_3.txt apple
1: The only line with a matching search term: apple.
```

As we can see test_file_3.txt was only one line that has the word apple, so the only line was displayed

Test case 4:

Searching for the term **apple** in empty_file.txt

Output:

```
aulavirtual@Linuxv3034:~/Música$ touch empty_file.txt
aulavirtual@Linuxv3034:~/Música$ ./mygrep empty_file.txt apple
aulavirtual@Linuxv3034:~/Música$ ./mygrep
```

There is no output since the file is empty and contains no lines to search through

Test case 5:

Running the program with incomplete-incorrect arguments

Output:

```
aulavirtual@Linuxv3034:~/Música$ ./mygrep
Usage: ./mygrep <file_path> <search_term>
```

The program prints the correct form of arguments that should be passed

Test case 6:

Searching for a non-existent file

```
aulavirtual@Linuxv3034:~/Música$ ./mygrep non_existent_file.txt test  
Unable to open file: No such file or directory
```

Conclusion:

Problems encountered:

1- Problem:

We had issues reading the file line by line because the standard `read()` function reads chunks of data rather than structured lines. This made it difficult to process and search for specific terms in a text file.

Solution:

We implemented a custom function `read_line()` to read one line at a time from the file, ensuring we can process and analyze each line separately

2- Problem

The given function `procesar_redirecciones` was originally designed to handle redirection symbols (`<`, `>`, `>>`) in command-line arguments, but it could only handle one redirection per line. When multiple redirections were present, the function did not correctly update the argument indices, resulting in unexpected behavior or incorrect redirection.

Solution

To fix this issue and support multiple redirections in one line, we added `i++` after each redirection processing block. This ensures that the argument index correctly moves past the processed redirection symbol and its corresponding file name. Without this increment, the function would repeatedly process the same arguments, causing errors.

3- Problem: The program have experienced a buffer overflow when a single line exceeded the buffer size, leading to segmentation faults or corrupted output.

Solution: The line length is limited by checking whether `line_index` has reached `BUFFER_SIZE - 1` before adding more characters. This prevents buffer overflows and keeps the program stable.

4- Problem: Lines that do not end with a newline character (`\n`) are not properly null-terminated, causing incorrect or garbled output.

Solution: The code explicitly null-terminates each line with `line[line_index] = '\0';`, ensuring proper string handling when using `strstr()` or printing the line.