Operating systems: Laboratory practice 2 Programming a shell script interpreter

Héctor Molina Garde.
Group 88.
Computer Science Engineering Degree

Nicolás Maire Bravo.
Group 88.
Computer Science Engineering Degree

Guillermo González Avilés.
Group 88.
Computer Science Engineering Degree

Code description:

1. **‘*scripter.c*’ File**:

Our ‘scripter.c’. code implementation uses mainly two functions: ‘read_script()’ and ‘execute_command()’, as well as the code implemented in ‘main()’ and some slight modifications in other functions.

a. Miscellaneous Changes.

Initially, a global variable for the maximum number of lines is created so a fixed number of lines in the dynamic creation of the parsed text matrix is done, the variable increases in size every time a reallocation is performed.

Aditionally, an empty line checker at the beginning of the ‘procesar_linea()’ function is implemented, so the interpreter can perform all the previous commands before the raising error of the empty line. Also a variable to store the result of the write file descriptor from the previous command execution is declared, it is initialized to ‘-1’ because there was not a pipe in the last execution, i. e. it is the first command in the pipeline or it is unique. This variable is modified in the ‘execute_command()’ function, a more thorough description will be provided at that point.

Also, as suggested in the FAQ, a brief function for removing the quotes from the first and last character from the string arguments of the commands array.

b. *read_script*()

The function receives as parameters: the path to the script, the dynamic matrix pointer to store the parsed text and the pointer to an integer variable to count the number of lines.

Initially, the file is opened by the corresponding system call and checks if it was opened correctly. Then, three local variables are declared: the index of the parsed line, a variable to store the characters read and other to save the output of the read() system call. A ‘while’ loop is initialized until it reads an error or an end of file. Inside the loop, the function checks whether the character read was a new line (\n), then the function ensures memory is correctly allocated for storing the line. the current line is terminated with a null character (\0), the line_count is incremented, and line_index is reset. If the number of lines exceeds the allocated memory, the function reallocates more memory to store additional lines.

In the case that a regular character is encountered, the line continues and it is parsed. The function ensures memory allocation for the current line. The character is stored in the line array, and the line index is incremented. If the line length exceeds the predefined limit, an error is displayed.

After exiting the loop, the function checks for errors during reading. If an error occurred, it prints an error message and returns -1. The file is closed using the close() system call, and an error message is printed if the operation fails. If the last line did not end with a newline character, it is properly terminated and counted. Finally, the function returns 0 to indicate a successful execution.

c. *execute_command*()

This function ‘execute_command()’ is responsible for executing a command in a pipeline while handling input/output redirections and process creation. It takes three parameters: an integer with the total number of commands in the pipeline, an integer indicating the index of the current command in the pipeline and a pointer to an integer that stores the file descriptor of the last pipe’s read end, which is the previously declared and commented variable. Infinite pipes have been implemented in the script.

The function starts by declaring local variables: the array to hold the pipe’s file descriptors, a variable to store the process ID after the ‘fork()’ system call. Also, a pair of variables intended as file descriptors (‘input_fd’ and ‘output_fd’) are initialized to standard input and standard output .

If there are multiple commands (num_commands > 1) and the current command is not the last one, a new pipe is created using ‘pipe()’. Then, a child process is created through the ’fork()’ system call. So, if ‘fork()’ fails, an error is displayed and the function exits.

Hence, if the ‘PID’ equals zero, the process acts as the child process. If the first command has an input redirection, the input file is opened, and its file descriptor is assigned to ‘input_fd’. If the command is not the first in the pipeline and an input redirection is attempted, an error message is displayed. If the command is in a pipeline (‘prev_pipe_fd’ is valid), it reads from the previous pipe into ‘input_fd’. It then checks if ‘input_fd’ is different from the standard input, if it is, the ‘dup2()’ system call is used to redirect ‘input_fd’ to standard input, and the original file descriptor is closed.

In the output redirection of the child process checks that if the last command in the pipeline has an output redirection, the output file is opened, and its file descriptor is assigned to ‘output_fd’. Additionally, if the command is not the last in the pipeline and an output redirection to a file is attempted, an error message is displayed. If the command is in a pipeline and not the last one, the write end of the pipe is used as the ‘output_fd’. Then, if ‘output_fd’ is not the standard output, it is redirected by the use of ‘dup2()’ and the original file descriptor is closed.

The error redirection is handled if it is specified in any command of the pipeline.Thus, it is opened, and its file descriptor is redirected to standard error using `dup2()`. Furthermore, to avoid error if the command is in a pipeline and not the last one, the read end of the newly created pipe is closed. Finally in the child process, the command is executed using ‘execvp()’. If execution fails, an error message is displayed, and the child process exits.

Then, in the parent process, if there was a previous pipe, its read end is closed. If the command is part of a pipeline and not the last one, the write end of the new pipe is closed, and the read end is stored in ‘prev_pipe_fd’ for the next command in the pipeline. In addition, if the command is not running in the background, ‘waitpid()’ is used to wait for the child process to finish execution. Otherwise, the background process ID is printed.

The error handling and process management is taken into account in: the function in process creation, file handling, and system calls like ‘pipe()’, ‘dup2()’, and ‘execvp()’. Thus, it ensures proper redirection of input, output, and error streams while handling multiple commands in a pipeline. It manages background execution by waiting for processes only if they are not running in the background.

d. *main*()

The ‘main()’ function is responsible for controlling the flow of the program: reading a script file, validating its format, and processing its lines using the ‘procesar_linea()’ function. It dynamically allocates memory for storing the script lines and ensures proper memory management throughout execution.

The execution of the function validates the usage of it with exactly the use of two arguments, the name of the executable and the script file path, whenever an incorrect usage is detected, it prints a message and exits. A matrix ‘lines’ is allocated to store the script parsed lines, with an initial capacity of ‘max_lines’. Each entry in the array is allocated ‘max_line’ characters to hold individual lines. If any memory allocation fails, an error message is displayed, and the function exits.

Thus, the function calls ‘read_script()’ to read the script file into ‘lines’ and count the number of lines (‘line_count’). If `read_script()` fails, allocated memory is freed before exiting. The function then validates whether the first line of the script matches "## Script de SSOO", if it doesn’t, an error is displayed, and the function exits. During the processing of lines, each line (except the first one) is processed using ‘procesar_linea()’. After all the executions, the memory is deallocated by using ‘free()’ in each line of the matrix to avoid memory leaks.

2. ‘*mygrep.c*’ File:

a. *main*()

The ‘mygrep’ program is responsible for reading a file line by line, searching for a given string, and printing lines that contain the specified string. It takes command-line arguments specifying the file path and the search string or if just one argument is given if the redirection of input is given. The function checks if two arguments are provided (the file path and the string to search). If incorrect usage is detected, an error is raised. The file is opened in read/write mode using ‘open()’. If the file cannot be opened, an error is raised. Nevertheless, if the case of a redirection is given, the argument to be searched will just be the standard input, as in the basic grep command.

A buffer to store the readed line is allocated using ‘calloc()’ to set all the values of the buffer array to zero. If memory allocation fails, an error is raised. Thus, local variables are created to store the index of the line read, the character read by the read() system call and a flag to check if the string is in the file.

The function reads the file character by character using ‘read()’. Whenever a newline is encountered: it is checked with the function ‘strstr()’ that the string inputted is a substring of the buffer, so the line contains the wanted line. Else the character read is appended to the buffer array of the line and the index incremented. However if the line is big enough a reallocation is done.

After the file is read, it is checked with the function ‘strstr()’ if the line contains the wanted line, as the final line could not have a new line character (‘\n’) so the comprobation is needed. If ‘read()’ fails, an error message is displayed, and the file is closed. If ‘close()’ fails, an error is displayed. If the search string was not found in the entire file due to the found in

file flag being deactivated, a message is displayed. Finally, the allocated buffer is freed before the function returns. Furthermore, all errors in functions are considered and handled and the dynamic memory is allocated and deallocated to avoid memory leaks.

Test cases:

As a starting point, the programs pass the tests provided by the teachers in tester.sh. The environment of execution is a virtual machine of Ubuntu x64 in VMWare Workstation, in the Bash`s terminal. The programs were tested in the Picasso LabINF to check the correctness of the code, the repository where the code resided was cloned and the first iteration of the tester.sh used. However, due to the slowness of it and the geolocational constraints, the code was decided to be tested on another machine.

Note: All the planned input rows will just have the inputted arguments in mygrep.c case, and the testing commands in the scripter.c or the conditions of executions (file exits, no permissions, etc.) to avoid redundancy, such as the inclusion of '## Script de SSOO' in the testing commands beyond the tests that require the line management.
Abbreviations: (com: command/s), (redir: redirection), (F: file), (&: background), (exec: execution), (str: string), (perm: permissions), (incorr: incorrect), (argm: argument)

'*scripter.c*' Tests				
Description	Planned input	Expected output.	Terminal input/output	Status
More than 2 arguments	./scripter test1.txt test2.txt	Usage error	$ ./scripter test1.txt test2.txt Incorrect number of arguments: Invalid argument	✅
Script file without read perm	chmod -r test.txt ./scripter test.txt	Error opening file	$ chmod -r test.txt $ ./scripter test.txt Error opening input script: Permission denied	✅
Script file without write perm	chmod -w test.txt ./scripter test.txt	Run correctly the test as the script is not written	$ ./scripter test.txt pre kill	✅
scripter without exec perm	chmod -x test.txt ./scripter test.txt	Run correctly the test	$ ./scripter test.txt pre kill	✅
script file is a directory	./scripter ./dir	:is a directory	$ ./scripter ./dir Error opening input script: No such file or directory	✅
script file is empty	input file is empty	The script does not start with: "## Script de SSOO""	$ ./scripter test.txt The script does not start with: ## Script de SSOO: Exec format error	✅
F with incorr 1st line	without "## Script de SSOO"	The script does not start with: "## Script de SSO\"	$ ./scripter test.txt The script does not start with: ## Script de SSOO: Exec format error	✅

F with just 1st line	“## Script de SSOO”	Nothing	$ ./scripter test.txt	✅
empty line in 2nd line	“## Script de SSOO ”	Error: empty line in the script	$ ./scripter test.txt Empty line in the script: Exec format error	✅
Empty line in 3th line after com. execution	“## Script de SSOO pwd ls wc test.txt ”	output pqd output ls Error: empty line in the script	$ ./scripter test.txt /home/hector/Oper_sys_lab2/src autores.txt Makefile mygrep mygrep.c mygrep.o scripter scripter.c scripter.o test.txt txt_files zipper.sh total 116 Empty line in the script: Exec format error	✅
Script file with mixed ‘\n’ and ‘\r’	pwd wc test.txt	Normal output of pwd and ls	$ ./scripter test.txt /home/hector/Oper_sys_lab2/src 2 7 45 test.txt	✅
F with +10 lines to force realloc	25 ‘pwd’ commands	Normal output of 25 times pwd	Normal output of 25 times pwd	✅
F with an extremely big size	14000 com in the script	Are executed but slowly	14000 pwd executions with success in the allocation	✅
Incorrect com.	invented command: cacatua	Command execution failed: No such file or directory	$ ./scripter test.txt Command execution failed: No such file or directory	✅
Com. with incorr argm.	ls -%	ls: opción incorrecta -- «%» Pruebe 'ls --help' para más información.	$ ./scripter test.txt ls: opción incorrecta -- «%»	✅
exec absolute path com.	/bin/ls	Normal output of ls	$ ./scripter test.txt autores.txt Makefile mygrep mygrep.c mygrep.o scripter scripter.c scripter.o test.txt txt_files zipper.sh	✅
exec ‘.’ path	pwd | ./grep home	the result of home	$ ./scripter test.txt /home/hector/Oper_sys_lab2/src	✅
exec ‘..’ path	../test/tester.sh zipfile.zip 1	executed with an error due to the zip name	$ ./scripter test.txt [ENG] zipfile.zip does not exist. [ESP] zipfile.zip no existe.	✅

Input redir	wc < in1.txt	A correct output from wc	$ chmod 000 in1.txt
			$ chmod +r in1.txt
			$ ./scripter test.txt
			5 28 151
				✅

Input redir with only read perm				
Output redir	ls > out1.txt	Correct result in the redirected file	$ ./scripter test.txt Correct result in the redirected file	✅
Output redir with only write perm				
Output redir to several F	ls > out1.txt > out2.txt	ls: opción incorrecta -- «> out1.txt»	$ ./scripter test.txt Saved in 1st redir. file, out2 ignored	✅
Error	ls -% !> out1.txt	ls: opción incorrecta --	$ ./scripter test.txt	✅

redirection		«%» in the redirected file out1.txt	ls: opción incorrecta -- «%» Pruebe 'ls --help' para más información.	
Input redir without read perm	wc < in1.txt	Error opening	$ chmod -r in1.txt $ ./scripter test.txt Error opening input file for read: Permission denied	✅

Output redir without write perm	ls > out1.txt	Error creating file	$ chmod -w out1.txt
			$ ls -l | grep out1.txt
			-r--r----- 1 hector hector 123 abr 2 00:24 out1.txt
			$ ./scripter test.txt
			Error creating output file: Permission denied	

Error redir without write perm	ls -% !> out1.txt	Error creating file	$ ls -l | grep out1.txt -r--r----- 1 hector hector 75 abr 2 00:32 out1.txt $ ./scripter test.txt Error creating error file: Permission denied	
Exec in & Basic pipe Pipe with 3 com. Pipe with +5 com. Kill the shell in the script	echo pre kill ps aux | grep scripter | grep -v grep | awk {print$2} | xargs kill -SIGINT & sleep 3 echo post kill	“pre kill “	$ ./scripter ./txt_files/test.txt pre kill	✅
Pipe with +10 commands	echo test | cat | cat | cat | cat | cat | cat | cat | cat | cat	test	$ ./scripter test.txt test	✅
Pipe with mixed com.	cat < input.txt | grep error > output.txt	error line 1 line 3 with an error too in out1	error line 1 line 2 line 3 with an error too —-- $ ./scripter test.txt —-- error line 1 line 3 with an error too	✅
Pipeline where a com. fails midway	cat file.txt | nonexistentcommand | wc -l	Command execution failed	$ ./scripter test.txt Command execution failed: No such file or directory 0	✅
Pipe with +2 com. with ‘>’ in mid. com. Pipe with +2 com. with ‘<’ in mid. com.	echo test | cat < input1.txt | cat | cat | cat	Error redirection	$ ./scripter test.txt Commands between pipes cannot have redirections: Invalid argument	✅

Pipe with +2 com. with '!<' in mid. com.	echo test | cato !> file..txt | cat | cat | cat | cat	Command execution failed.	$ ./scripter test.txt Command execution failed: No such file or directory	✅
'*mygrep.c*' Tests				
Description	Planned input	Expected output.	Terminal input/output	Status
More than 3 arguments	./mygrep file.txt word word2 word3	"Usage: mygrep <ruta_fichero> <cadena_busqueda>	$ ./mygrep file.txt word word2 word3 Usage: ./mygrep <ruta_fichero> <cadena_busqueda>: Invalid argument	✅
Force buffer realloc	Input file with a line bigger than 1024 chars.	Correct output	$ ./mygrep inp.txt a aaaaaaaaaaaaaaaaaaaaaaaaaaaaa…aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa	✅
Force multiple buffer realloc	Input file with a line bigger than 2*1024 chars.	Correct output	Same as before but with six thousand char in a line	✅
Basic search	./mygrep file.txt file	This line is inside a file :)	$ ./mygrep file.txt file This line is inside a file :)	✅
Several str in same line	here here here bla text blabla bla bla bla	here here here bla text	$ ./mygrep file.txt here here here here bla text	✅
only one occurrence of str	nothing here just one test here nothing here	just one test here	$ ./mygrep file.txt test just one test here	✅
Only one str in F	here	here	$ ./mygrep file.txt here here	✅
Str in every line	test in line one test in line two test in line three	test in line one test in line two test in line three	$ ./mygrep file.txt test test in line one test in line two test in line three	✅
Search special char	?world! hello?world! special*characters +here	hello?world!	./mygrep file.txt hello?world! hello?world!	✅
non-ascii character	"é" café münchen naïve façade	café münchen	$ ./mygrep file.txt é café münchen	✅
Check intersection of strings	"cbcc" acbccbcc	abcbcc	$ ./mygrep file.txt acbccbcc acbccbcc	✅
Str at start and end of line	"test" test is at the beginning ends with test no match here	test is at the beginning ends with test	$ ./mygrep file.txt test test is at the beginning ends with test	✅

Search in an empty F	./mygrep empty_file.txt search_string	str not found	$ ./mygrep file.txt test test not found	✅
File not existing	./mygrep non_existent.txt search_string	Error opening input script	$ ./mygrep non_existing_file.txt test Error opening input script: No such file or directory	✅
Input file with no permission	chmod 0000 file.txt ./mygrep file.txt search_string	Error opening input script	chmod 000 file.txt $ ls -l | grep file.txt ---------- 1 hector hector 0 abr 2 01:10 file.txt $ ./mygrep file.txt test Error opening input script: Permission denied	✅
Combining '*mygrep.c*' and '*scripter.c*'. mygrep being a program inside the script read by scripter				
Description	Planned input	Expected output.	Terminal input/output	Status
mygrep in a pipe as STD output	./mygrep ./scripter.c std | sort	All appearances of std sorted	$ ./scripter test.txt } // Redirect stdin to fd #define max_redirections 3 // stdin, stdout, stderr #include <stdio.h> #include <stdlib.h> #include <unistd.h>	✅
mygrep redir output	./mygrep scripter.c std > output.txt	std this line contain std	$ ./scripter test.txt ————— #include <stdio.h> #include <stdlib.h> #include <unistd.h> #define max_redirections 3 // stdin, stdout, stderr } // Redirect stdin to fd	✅
mygrep redir error	./mygrep test.txt !> output.txt	Usage error in the redirected file	Usage: mygrep <ruta_fichero> <cadena_busqueda>: Invalid argument	✅

Conclusions:

During the development of the minishell, several challenges made the implementation process difficult. One of the most complex aspects was debugging, as handling multiple processes, redirections, and system calls required careful testing and analysis. Unlike debugging a regular program, where you can simply print values or use a debugger, working with a shell means dealing with processes running in parallel, which makes it harder to track errors. Sometimes, bugs appeared only under specific conditions, making them difficult to reproduce and fix. That increases the time developing the execution part of the shell

Another particularly difficult feature to implement was handling unlimited pipes. Any small mistake in this process could lead to unexpected behavior, such as broken pipes, execution, or commands getting stuck indefinitely.

Another difficulty was the lack of clarity in some parts of the project statement, leading to uncertainty about the expected functionality. This meant that extra research and discussions were needed to fully understand what had to be implemented.