University Carlos III of Madrid

BSc in Applied Mathematics and Computing

Lab 2: Programming a shell script interpreter

Member 1: Sabela Rubert Docampo
100523102
100523102@alumnos.uc3m.es

Member 2: Roberto Hogas Goras
100523139
100523139@alumnos.uc3m.es

Member 3: Alejandro Tarbay Guadalupe
100523065
100523065@alumnos.uc3m.es

Academic Year 2024/25 Group 121

With love, made in LaTeX for Catherine, our Operating Systems' lab teacher.

Click here to see our Overleaf code.

Contents

1 Description of the codes

1.1 scripter.c

In this first code we were asked to program a C program that reads and executes a series of commands defined in a file. Each line of the file can contain one or more commands, which may be connected by pipes and redirect input and output to specific files. It also allows executing commands in the background (without waiting for them to finish).

To make the code, in Aula Global we downloaded a folder called "**Support code**" which contained the files we needed to compile the codes, as well as the start of `scripter.c` and `mygrep.c`.

We had some constraints as constant variables to help us delimit the extension of the inputs:

- const int maxline = 1024;
- const int maxcommands = 10;
- #define maxredirections 3
- #define maxargs 15
- #define READ 0
- #define WRITE 1

The program starts by checking if the correct number of arguments has been provided. If the wrong number of arguments is given, it prints an error message explaining how to enter the arguments and then stops running.

Then, the program tries to open the file specified by the user. If it cannot open the file, it prints an error message and exits. Once the file is opened, the program allocates a `max_line`. Then, it begins reading the file one character at a time with a buffer called `buffer`, that stores all characters from the provided script file in an array called `line`, of size `max_line`..

The program processes these lines one by one, taking into account that no line can be superior to `max_line` and that every time it encounters a newline character (meaning the end of a line), it is added the character `\0`. By doing this, we do not need to reset `line`, as the program only reads until the first `\0`.

Also, the program checks if there are any special symbols or characters that indicate redirection or background execution. **Redirection** is the process of using as the input the content of a file instead of the one the user gives, or sending the output to a file instead of printing it on the screen. The program checks for three types of redirection:

- '`<`' for input redirection
- '`>`' for output redirection
- '`!>`' for error redirection

If it finds any of these, it stores the respective file names in an array and removes the redirection symbols from the command to allow proper execution.

The program also checks if the command should run in the background by looking for an '&' symbol at the end of the command. If the symbol is found, it sets a flag to indicate that the process should run in the background.

After processing the redirection and background flags, the program moves on to handle pipes. **Pipes** are used to connect multiple commands, where the output of one command is passed as input to the next. If there are some commands in the line separated by the '|' symbol, the program creates pipes to allow the commands to communicate between each other. Each command in the **pipeline** (we call pipeline the method of connecting multiple processes so that the output of one process becomes the input of another) is then executed in a separate child process created with the 'fork()' system call. The parent process creates a new child process for each command in the pipeline. In each child process, the program sets up the pipes and redirects the input and output to where they should be redirected. For the first command, the program writes only to the next pipe and does not read from any pipe, while the last command reads only from the previous pipe and does not write to any pipe. For commands in the middle, the program ensures that they both read from the previous pipe and write to the next pipe.

If any of the commands include redirection, the program uses the 'dup2()' function to redirect the standard input or output to the specified file. This allows the program to handle input and output redirection properly. For example, if a command has an input redirection ('<'), the program opens the specified file and redirects the input to come from that file instead of the keyboard. Similarly, if there is output redirection ('>'), the program opens the specified file and redirects the output to that file instead of printing it to the screen. The same happens with error redirection ('!>'), but for error messages instead of normal output. All of this is done with the help of a function called: `procesar_linea`.

After setting up all the pipes and redirections in the child process, the program uses `execvp()` to execute the command. If the command fails to execute, it prints an error message.

Meanwhile, in the parent process, after forking the child process, it waits for the child process to finish unless the command is running in the background. If the command is in the **background**, the parent prints the child's process ID and continues to the next command without waiting for the child to finish. The program ensures that no **zombie processes** are left behind by waiting for all processes that don't run in the background to terminate before exiting.

Lastly, we must remember that the program also has **error handling**. For example, if the first line of the script is not the expected `## Script de SSOO`, the program will print an error and exit. If the script contains any empty lines, it will also print an error and exit. Additionally, if there are any issues opening a file for redirection, the program will print an error message and exit.

1.2 mygrep.c

In this second code we were asked to program a C program very similar to the already existing 'grep' that comes from POSIX, used to search for a specific string within a file and print the lines that contain that string. The program starts by checking if the correct number of arguments has been provided. If the wrong number of arguments is given, it prints an error message explaining how to enter the arguments and then stops running.

Next, the program tries to open the file specified by the user. If it cannot open the file, it prints an error message and exits. Once the file is opened, the program allocates a 1024-byte buffer in memory to store the data as it reads it from the file. If it cannot allocate enough memory, it shows an error message and stops.

The program then begins reading the file one character at a time with a pointer called `c`. If the buffer is not large enough to store the characters being read, the program increases the buffer size to make sure it can continue reading the file without problems. The buffer size is doubled each time it's increased, which is more efficient than increasing it little by little because it reduces the number of times the program has to rearrange the memory.

Every time it encounters a newline character (meaning the end of a line), the program checks if the line contains the string we are searching for. This is done by not adding a newline, but instead adding the character `\0`. By doing this, we do not need to reset the buffer, as the program only reads until the first `\0`. If it finds the string in the line, it prints the line. If not, it moves on to the next line. After reading the whole file, it checks if the last line (which might not end with a newline character) contains the target string and prints it if necessary.

The program is also set up to handle errors. If any problems occur while reading the file or allocating memory, it shows an error message and stops running. If no matches are found, it informs the user that the string wasn't found.

Finally, the program closes the file and frees the memory it allocated to read the file. This way, the program efficiently handles memory while processing large files and provides a basic way to search for strings within files.

2 Libraries used in both scripter.c and mygrep.c

In `scripter.c` and `mygrep.c`, there were included the most common libraries used by any program that needs to open, close and edit files. These are:

- **#include <stdio.h>**: used to print, read and display error messages with `pererror`
- **#include <stdlib.h>**: includes functions such as `malloc` and `free` (to manage memory) and `exit` (to terminate the program)
- **#include <fcntl.h>**: includes the flags we need to create files with the `open` function
- **#include <sys/stat.h>**: library used for process control, specifically to manage child processes. It provides functions like `wait()` and `waitpid()` to allow a parent process to wait for its child processes to terminate.
- **#include <unistd.h>**: includes POSIX system calls, such as `close`, `open` and `read`
- **#include <errno.h>**: defines the global variable `errno` to store system error codes and print them with `perror`
- **#include <string.h>**: includes functions for handling strings, such as `strlen` (length of a string) or `strcmp` (compares two strings)
- **#include <signal.h>**: this library provides functions and macros for handling signals, which are asynchronous events that notify a process that something has happened. Signals can be generated by the user, the system, or a program itself, and they are used for various purposes like process control, error handling, and inter-process communication.
- **#include <sys/types.h>**: includes several data types used when working with system files and other processes that engage in low-level operations.
- **#include <sys/wait.h>**: includes unctions related to process termination and status information, such as `wait` and `waitpid`

3 Test cases

3.1 scripter.c

The test cases provided by the teacher have been tried and they worked perfectly (100%). We did not include them in this subsection because we were not the ones to come up with them.

Data to enter	**test_case_number.sh**	**Description of the test**	**Expected result**	**Obtained result**
./scripter InputScripts/scriptertest1.sh	## RANDOM LINE ls echo "ssoo,os"	The first line is different from ## Script de SSOO	The first line is not "## Script de SSOO":	The first line is not "## Script de SSOO":
./scripter InputScripts/scriptertest1.sh	ls echo "ssoo,os"	The first line is not ## Script de SSOO	The first line is not "## Script de SSOO":	The first line is not "## Script de SSOO":
./scripter InputScripts/scriptertest1.sh	## Script de SSOO sleep 5 echo "Background process started"	Ensures that the process runs in the background and does not block execution	"Background process started".	"Background process started"
./scripter InputScripts/scriptertest1.sh	## Script de SSOO ls nonexistentfile !> error˙log.txt cat error˙log.txt	Ensures that errors are redirected correctly using (!>)	creation of 'error˙log.txt' with content: ls: cannot access 'nonexistentfile': No such file or directory	creation of 'error˙log.txt' with content: ls: cannot access 'nonexistentfile': No such file or directory
./scripter InputScripts/scriptertest1.sh	## Script de SSOO cat < nonexistent_ input.txt	Ensures that input redirection (<) fails when the specified file does not exist	In console: Error opening input file: No such file or directory	In console: Error opening input file: No such file or directory
./scripter InputScripts/scriptertest1.sh	## Script de SSOO	Ensures that an empty script file does not cause crashes	No output	No output
./scripter	## Script de SSOO echo "Hola Mundo" > salida.txt cat salida.txt	Ensures that output redirection (>) correctly writes to a file	Creation of 'salida.txt' file with content: "Hola Mundo"	Creation of 'salida.txt' file with content: "Hola Mundo"
./scripter InputScripts/scriptertest1.sh	## Script de SSOO echo "computeeeeeeeer" | head -c 20 | head -c 10 | head -c 5	Checking multiple pipes	compu	compu
./scripter InputScripts/scriptertest1.sh	## Script de SSOO echo "computeeeeeeeer" | head -c 20 | head -c 10 | head -c 5 &	Checking multiple pipes in background	207573compu	207573compu

Table 1: Test cases for the scripter

3.2 mygrep.c

Data to enter	File.txt	Description of the test	Expected result	Obtained result
./mygrep file.txt "test"	Hi world This is a test To see if the search is successful	A normal search case, to see if the main function works.	This is a test	This is a test
./mygrep file.txt "looking"	Line one Line two This is the line we are looking for	A normal search but in the last line.	This is the line we are looking for	This is the line we are looking for
./mygrep file.txt "sistemas"	Hello Catherine Operating Systems	String not found.	"sistemas" not found.	"sistemas" not found.
./mygrep file.txt "test"	We're testing our mygrep function This is the test	The string is part of a word, and it appears twice.	We're testing our mygrep function This is the test	We're testing our mygrep function This is the test
./mygrep file.txt "empty"		The file file.txt is empty.	"empty" not found.	"empty" not found.
./mygrep hello.txt "empty"		The file hello.txt does not exist.	Error opening the file: No such file or directory	Error opening the file: No such file or directory
./mygrep file.txt		Missing arguments	Error: incorrect number of arguments. Usage: ./mygrep file string: Invalid argument	Error: incorrect number of arguments. Usage: ./mygrep file string: Invalid argument
./mygrep file.txt "test" extra		Extra arguments	Error: incorrect number of arguments. Usage: ./mygrep file string: Invalid argument	Error: incorrect number of arguments. Usage: ./mygrep file string: Invalid argument

Table 2: Test cases for the function mygrep

4 Conclusion

This assignment helped us understand how the pipes work and developed our knowledge of C. We had several problems while running the tests, as it was hard for us to know where the errors were. Furthermore, as it was the first time for us working with the background concept, we especially struggled to find an implementation for it. The time spent on this project was about 40 hours.