Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Laboratory 2. Programming a shell script interpreter

Bachelor's Degree in Computer Science & Engineering

Bachelor's Degree in Applied Mathematics & Computing

Dual Bachelor's in Computer Science & Engineering & Business Administration

Academic Year 2024/2025

1. Laboratory statement

This laboratory allows the student to become familiar with the process management services provided by POSIX. It is also intended to familiarise the student with the inner workings of a UNIX/Linux shell. In short, a shell allows the user to communicate with the operating system kernel by executing commands or commands, either simple or chained.

For heavy process management, the POSIX system calls such as fork, wait, exit will be used. For inter-process communication, the system calls pipe, dup, close, signal shall also be used.

The student must design and code, in C language and on the Linux operating system, a program that reads a file that includes Linux commands and executes them. The file will have one command per line and the entire command on the line must be executed before moving on to the next line. The program must strictly follow the specifications and requirements contained in this document.

1.1. Laboratory description

The program to be developed is called "Scripter", which must read line by line a command file received as input argument, and by using a command interpreter it must execute the commands as if it were a native terminal.

An example of the content of an input file is shown below:

```
## Script de SSOO
ls -l
cat fich | grep word
cat fich > fich2 &
ls -l | sort < file
```

The first line is mandatory, so the program must check that it exists to know if the file is a script. If it does not exist, it must return an error using the **perror** function and **return -1**.

Subsequently, the developed program must be able to execute, one by one, the commands listed in the file. Empty lines cannot be included, so this format must be checked for correctness. If an empty line is found, the program must terminate and display an error message using the **perror** function and return -1

The program must execute all commands, even if they fail (the failure of a command does not imply a failure of the interpreter). The interpreter must terminate:

1. When all the commands in the file have been completed.

2. When an error has occurred in the system calls used in the code (fork, exec, pipe, dup, etc.)
3. When the first line of the input file is different from "## `Script de SSOO`".
4. An empty line has been found.

1.2 Command processing

For the development of this lab, it is necessary for the students to process a command file passed as an argument to the program. **They must read character by character until they find a new line and insert a '\0' at the end of the string.** Once this step is implemented, each line can be passed as a parameter to the function ***procesar_linea (****included in the initial code; scripter.c)*, which is responsible for providing everything necessary for the student to proceed with the management of the processes.

Variables to be used by students included in the initial code:

- **char * argvv[]:** structure equivalent to the "argv" that stores the command line when executing a program. In this case, this structure will store the line corresponding to a command with its arguments.
- **char * filev[3]:** array that stores if there is a redirection from/to a file. Each position corresponds to:
    - filev[0]: input redirection (file descriptor 0).
    - filev[1]: standard output redirection (file descriptor 1).
    - filev[2]: error output redirection (file descriptor 2).

  If there is a redirection, the content of the position shall be the pointer to the filename. Otherwise, the value of the position shall be *NULL*.
- **int background:** the value is 1 if the command has to run in the background. Otherwise, it is 0.

The initial code (*scripter.c*) includes, within the *procesar_linea* function, a code that displays the contents of argvv, background and redirections. It is recommended to evaluate different cases once the command file has been read and understood. Subsequently, it is recommended that this code be modified to add process execution functionality.

Considerations:

1. A line can be a command with or without arguments (e.g. "ls" vs "ls -l" vs "ls -l directory").

2. A line can be a series (pipe) of chained commands, each with its respective flags and arguments (ls -l | wc).
3. In addition, each command line may have a series of standardised characters at the end indicating redirection and background. The syntax used is as follows:
    a. " > ": output redirection.
    b. " < ": input redirection.
    c. " **!**> ": error redirection.
    d. " **&** ": indicates that the command will be executed in the background.

Therefore, **a command is understood as** a sequence of texts separated by spaces. The first text specifies the name of the command to be executed. The remaining ones are the arguments of the command invoked.

The command name is passed as argument 0 (*man execvp*). Each command is executed as a direct child process of the interpreter (*man 2 fork*). The value of a command is its termination status (*man 2 wait*). If the execution fails, the error is reported by the standard error output via the **perror** function.

A sequence of two or more commands is a series of commands separated by the " | " character (*man 2 pipe*). The output of the first command is automatically passed as input to the next command. The main program waits for the last command to finish before accepting a new instruction. The output of the entire sequence will be that of the last command executed.

A **redirect** indicates that the input or output of a command should be taken from/to a file instead of the terminal. The input or output of a command or script can be redirected by adding the following notation after it:

< file Use a **file** as standard input by opening it for reading (*man 2 open*).

> file Use a **file** as standard output. If the file does not exist it is created, if it does exist it is truncated (*man 2 open / man creat*).

!> file Use a **file** as standard error output. If the file does not exist it is created, if it does exist it is truncated (*man 2 open / man creat*).

In case of any error during redirections, it is reported by the standard error output (**perror**), execution of that line is suspended and the next line is continued.

A **command or sequence ending in " & "** implies asynchronous execution of the command, i.e. it is executed in the background, whereby the main program continues the execution of the rest of the commands. Considerations should be given to:

1. The parent process must print via standard output the *pid* of the child process (See Annexes).

2. The ending of the child processes must be managed correctly, as failure to do so will result in a penalty in the final grade.

Note: To become familiar with the structures in which commands, redirections, and background are returned, we recommend that the student compile and run the provided program to test different commands and scripts in the input file. The aim is to understand how to access each command clearly and ensure they are decoded correctly. However, to achieve this, the student must implement line-by-line reading of the input file, and make use of the provided function: **int procesar_linea(char *línea);**

1.3 Command interpreter development

To develop the command interpreter, the student is recommended to follow a series of steps to build the program incrementally. In each step, a new functionality will be added on top of the previous one.

1. Reading the input file character by character, recognising the command lines and adding '\0' as a delimiter.
2. Execution of simple commands such as ls, who, etc.
3. Execution of simple commands in the background (&).
4. Execution of scripts connected by pipes (|). The number of commands in a sequence will be limited to 3, for example: ls -l | sort | wc. If students choose to implement an unlimited sequence of pipes**, this will be considered for an additional score** (maximum score 10).
5. Execution of simple commands, scripts with input, output, error (<, >, !>), and background (&) redirects.
6. Execution of an external command (*mygrep*).

1.4 Development of an external command: mygrep

In addition to the code for the *Scripter* program, the student must implement a new command called *mygrep*. This command will search for a text string in a text file and display on standard output those lines in which it appears. If the file does not contain the searched string, the program must display the message: " %s not found.\n ", indicating the searched string between quotes. Finally, if any error occurs, the program must display a message on standard error output using the **perror** function and return -1.

The program must have the following interface:

mygrep *<file_path> <string_to_search>*

This program will be executed from the Scripter's input command file. So when the *mygrep* command is detected, it must be executed in a child process (like any other command), but it must indicate the path where the binary is located. For more information, see *man 3 exec.*

To execute this external program, you may use whatever necessary functions. The only requirement is that these functions have to be included in C standard libraries. It's not allowed to link new dynamic or static libraries.

Once the program is implemented, a compilation line must be added to the Makefile so that when "make", "make mygrep" and "make clean" are executed, the program is automatically compiled or the generated files are deleted.

Note: Note that the standard output *(file descriptor = 1)* must be used to present the on-screen output of the internal commands, while **all error messages** of the generated codes shall be displayed by the standard error output (file descriptor = 2) via the **perror** function **(**see documentation in man 3 perror**)**. In addition, **no debugging messages are allowed to appear on the standard output** (nothing but the outputs of the running commands must appear on the screen).

2. Initial Source Code

We provide you the *p2_scripter.zip* file with some initial code. To extract its contents, you should run the following command:

unzip p2_scripter.zip

By extracting its content, the directory *p2_scripter/* is created, where the laboratory should be developed. This directory contains the following files:

- **Makefile**
  Source file for the make tool, which achieves automatic recompilation only of modified source files. **It must be modified to add the compilation of the requested external command mygrep**.
- **scripter.c**
  C source file. **This file is the one to be modified for the development of the command processor.**
- **mygrep.c**

  C source file for the development of the external command.

3. Compilation and implementation

To compile and run the code of this lab, it is necessary to use the "make" command. The initial rules are already in the provided Makefile. However, the student must **modify** the **"Makefile"** file to add the "mygrep" program compilation rule.

Example of compilation:

$ make

$ make scripter

$ make mygrep → must be added in the Makefile by the student

make clean → must be modified in the Makefile by the student

Example of implementation:

$ ./scripter *<script_file_with_commands>.*

4. Submission

4.1 Submission Deadline

The laboratory's delivery deadline in AULA GLOBAL is **April 3rd, 2025 at 23:55h.**

4.2 Submission Procedure

In AULA GLOBAL, **one submission link** will be enabled **for the code** and another **TURNITIN** link **for the report**. A **single** member of the group must submit it.

4.3 Files to be submitted

A compressed file in ZIP format must be submitted with the name:

ssoo_p2_NIA1_NIA2_NIA3.zip

With the NIAs of the members of the group. In case of doing the laboratory alone, the format will be: **ssoo_p2_NIA1.zip.** The file must contain:

- **scripter.c**: Main file of the command interpreter.
- **autores.txt**: Text file in CSV format, with one author per line. The format is: NIA, Surname, First name
- **mygrep.c**: Main file of the requested external command.
- **Makefile**: Makefile modified by the student to compile the external program *mygrep*.

Important: deletion/modification of "*CC*", "*FLAGS*", "*CFLAGS*" variables is not allowed in the Makefile.

The report file must be delivered to the TURNITIN link with the name:

ssoo_p2_NIA1_NIA2_NIA3.pdf

The report shall contain at least the following sections:

- **Cover page:** with authors' full names, NIAs, and e-mail addresses.
- **Index**
- **Description of the code:** detailing the main functions implemented. Do NOT include source code from the lab in this section. Any code will be ignored.
- **Battery of tests** used and results obtained for both programs (Scripter and mygrep). Higher scores will be given to advanced tests, extreme cases, and to those tests that guarantee the correct functioning of the laboratory in all cases. It must be considered:
  - If a program compiles correctly and without *warnings*, it is not guaranteed that the program works correctly.
  - Avoid duplicate tests that assess the same program streams. The score for this section is not measured based on the number of tests but by covering a wide range of scenarios. Few tests assessing different cases are better than many considering the same case.
- **Conclusions** describe the problems encountered, how they have been solved, and opinions.

The following aspects relating to the **presentation** of the laboratory will also be evaluated:

- The report should have page numbers on all pages (except the title page).
- The text of the report must be justified.

The report has a page limit of 15 pages (cover page and table of contents included). You must obtain an approbatory score on the report to get an approbatory score on the lab, so the quality of the report must not be neglected.

NOTE: Only the latest delivered version of the code will be evaluated.

THE REPORT CAN ONLY BE SUBMITTED ONCE VIA TURNITIN.

Rules

1) **Programmes that do not compile or do not satisfy the functionality and requirements will receive a score of 0.**
2) Programs that use library functions **(fopen, fread, fwrite, etc.)** or similar, instead of system calls, will receive a score of 0.
3) Special attention will be paid to detect copied functionalities between two laboratories. In case of finding common implementations, the students involved (copied and copiers) will lose the grades obtained for continuous assessment.
4) Programs must compile without warnings.
5) The programs must run under a Linux system; the laboratory is not allowed to be carried out on Windows systems. In addition, to ensure the correct functioning of the laboratory, you must check its compilation and execution in the university's computer laboratories (on the server **guernika.lab.inf.uc3m.es**). If the code submitted does not compile or does not work on these platforms, the implementation will not be considered correct.
6) A programme that is not commented on will receive a very low grade.
7) The laboratory will be delivered through Aula Global, as detailed in this document's "**Submission**" section. Submissions via email are not allowed without prior authorisation.
8) Students must follow the guidelines on the input and output formats specified for each program.
9) Error checking must be performed for each program.

Submitted programs that do not follow these rules will not be considered approved.

Annexes

1.2. man function

man is the pager for the system manual, i.e. it allows you to search for information about a program, a utility or a function. See the following example:

man 2 fork

The pages used as arguments when running *man* are usually names of programs, utilities or functions. Normally, the search is carried out in all available manual sections in a predetermined order, and only the first page found is presented, even if that page is found in multiple sections.

To exit the displayed page, simply press the 'q' key.

A manual page has several parts. These are labelled NAME, SYNOPSIS, DESCRIPTION, OPTIONS, FILES, SEE ALSO, BUGS, and AUTHOR. The SYNOPSIS tag contains the libraries (identified by the *#include* directive) that must be included in the user's C program in order to make use of the corresponding functions.

The most common ways of using *man* are as follows:

- **man item section:** Displays the item page available in the manual section.
- **man -a element:** This displays, sequentially, all the pages of elements available in the manual. Between pages, you can choose to jump to the next page or exit the pager completely.
- **man -k keyword:** Searches for the keyword among the short descriptions and man pages and displays all matching ones.

1.3 Background and foreground mode

When a sequence of commands is executed in the background, the *pid* that is printed is that of the process executing the last command in the sequence.

When a simple command is executed in the background, the *pid* that is printed is that of the process executing this command.

With the background operation, it is possible for the main process to display the prompt intermingled with the output of the child process. This is correct.

After executing a command in the foreground, the interpreter cannot have zombie processes from previous commands executed in background.

1.4 External mandates

External commands (mygrep) are executed like any other program, running on top of a process. Therefore:

- They are part of sequences of mandates
- Have file redirections
- They run in the background

2. Bibliography

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada. Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (man function)