Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Lab 2. Programming a shell (Minishell)

Bachelor's Degree in Computer Science & Engineering
Bachelor's Degree in Applied Mathematics & Computing
Dual Bachelor in Computer Science & Engineering & Business Administration

Year 2023/2024

Contents

1 Lab Statement

This lab allows the student to familiarize with the services for process management that are provided by POSIX. Moreover, one of the objectives is to understand how a Shell works in UNIX/Linux. In summary, a shell allows the user to communicate with the kernel of the Operating System using simple or chained commands.

For the management of processes, you will use the POSIX system calls such as `fork, wait, exit`. For process communication `pipe, dup, close and signal` system calls.

The student must design and implement, in C language and over the UNIX/Linux Operating System, a program that acts like a *shell*. The program must follow strictly the specifications and requirements that are inside this document.

1.1 Lab description

The *minishell* uses the standard input (*file descriptor = 0*) to read the commands that will be interpreted and executed. It uses the standard output (*file descriptor = 1*) to present the result of the commands on the screen. And it uses the standard error (*file descriptor = 2*) to notify the errors that have happened. **If an error occurs in any system call, `perror` is used to notify it.**

1.1.1 Parser provided

For the development of this lab a **parser** is given to the student. This parser can read the commands introduced by the user. The student should **only** work to create a command interpreter. The syntax used by the parser is the following:

A space Is a space or a tab character.

A separator Is a character with a special meaning (—, <, >, &), a new line or the end of file (CTRL-C).

A string Is any sequence of characters delimited by a space or a separator.

A command Is a sequence of strings separated by spaces.

- The first string is the name of the command to be executed. The remaining strings are the arguments of the commands. For instance, in the command `ls -l`, **ls** is the command and –**l** is the argument.
- The name of the command is to be passed as the argument 0 to the **execvp** command (`man execvp`).
- **Each command must execute as an immediate child of the *minishell*, spawned by fork command** (`man 2 fork`).

- The value of a command is its termination status (`man 2 wait`), returned by exit function from the child and received by wait function in the father.
- If the execution fails, the error must be notified by the shell to the user through the standard error.

A command sequence Is a list of commands separated by |.

- The standard output of each command is connected through an unnamed pipe (`man 2 pipe`) to the standard input of the following command.
- The *minishell* typically waits for the termination of a sequence of commands before requesting the next input line
- The value of a sequence is the value returned by the last command in the sequence.

Redirection The input or the output of a command sequence can be redirected by the following syntax added at the end of the sequence:

- $< file \rightarrow$ Use *file* as the standard input after opening it for reading (`man 2 open`).
- $> file \rightarrow$ Use *file* as the standard output. If the file does not exist, it is created. If the file exists, it is truncated (`man 2 open / man creat`).
- $! > file \rightarrow$ Use *file* as the standard error. If the file does not exist, it is created. If the file exists, it is truncated (`man 2 open / man creat`).

In case of a error, the execution of the line must be suspended, and the user should be notified using the standard error.

Background (&) A command or a sequence of commands finishing in & must execute in background, i.e., the *minishell* is **not blocked waiting for its completion**. The *minishell* must execute the command without waiting and **print on the screen the identifier of the child process (*pid*)** in the following format, where `%d` denotes the *pid* of the process for which it is not wait **(See Appendix)**:

`[%d]\n`

The prompt Is a message indicating that the shell is ready to accept commands from the user. By default is:

`MSH>>`

1.1.2 Command line parsing

In order to obtain the parsed command line introduced by the user **you must use the function `read_command`**:

```
int read_command(char ***argvv, char **filev, int *bg);
```

The function returns 0 if the user types CTRL-C (EOF) and -1 in case of error. **If successful, the function returns the number of commands (*)**. For example:

- For `ls -l` → returns 1.
- For `ls -l | sort` → returns 2.

(*) **The input format must be respected** for the correct functioning of the parser, separating each term by a space:

```
[<command><args>][|<command><args>]* [<input_file>[>
output_file][!>outer_file][&]
```

The argument ***argvv*** contains the commands entered by the user. The argument ***filev*** contains the files employed in redirections, if any:

- **filev[0]** contains the file name to be used in standard input redirection. If it does not exist, it contains a string with a zero ("0").
- **filev[1]** contains the file name to be used in standard output redirection. If it does not exist, it contains a string with a zero ("0").
- **filev[2]** contains the file name to be used in standard error redirection. If it does not exist, it contains a string with a zero ("0").

The argument **in_background** is 1 if the command or command sequence are to be executed in background, its value is 0 otherwise.

EXAMPLE: : If the user enters `ls -l | sort < file` the structure of the arguments of `read_command` is shown in the following figure:

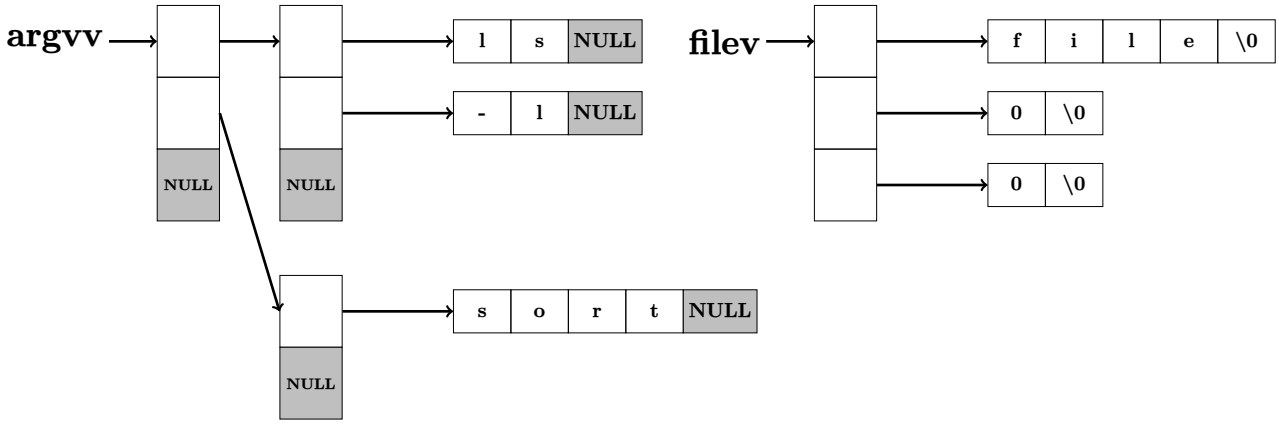

Fig. 1: Data structure used by the parser.

In the file msh.c **(file that must be filled in by the student with the *minishell* code)** the function `read_command` is invoked, and the next loop executed:

```
if (command_counter > 0) {
    if (command_counter > MAX_COMMANDS){
        printf("Error: Maximum number of commands is %d \n'', MAX_COMMANDS);
    }
    else {
        // Print command
        print_command(argvv, filev, in_background);
    }
}
```

In this code appears the function **`print_command()`**, which is in charge of printing the stored information captured from the command line. Its code is:

```
for (int i = 0; i < num_commands; i++){
    for (int j = 0; argvv[i][j] != NULL; j++){
        printf("%s\n'', argvv[i][j]);
    }
}

printf("Redir IN:  %s\n'', filev[0]);
printf("Redir OUT: %s\n'', filev[1]);
printf("Redir ERR: %s\n'', filev[2]);

if (in_background == 0)
    printf("No Bg\n'');
else
    printf("Bg\n'');
```

It is recommended that the students familiarize themselves with the execution of the provided code, before starting to modify it. This can be done by entering different commands and command sequences and understanding how they are internally handled by the code.

1.1.3 *minishell* development

To develop the *minishell* we recommend following the next steps, so that it is implemented incrementally. Each step will add a new functionality.

1. Execution of simple commands such as `ls -l`, `who`, etc.
2. Execution of simple commands in background (see Appendix to learn about commands n foreground and background to see more details about the requirements of commands in background).
3. Execution of sequences of commands connected through pipes. The number of commands is limited to 3 (*), e.g. `ls −l | sort | wc`.

   (*) The implementation of a version that accepts an arbitrary number of commands will be considered for additional marks.
4. Execution of simple commands and sequence of commands with redirections (input, output and error) and in background.
5. Execution of internal commands.

   An internal command is a command, which maps directly to a system call or a command internally implemented inside the shell. **It must be implemented and executed inside the *minishell* (in the parent process) without redirections and without background**. If it finds any error (not enough arguments or other type of error), a notification will appear and returns a non-zero value. The internal commands to be implemented are:

Internal command: mycalc

It works as a simple calculator in the command line. It takes a simple equation, following the format **operand_1 operator operand_2**, where operand is an integer number and operator can be sum (add), multiplication (mul), or division (div), in which the integer quotient and the remainder of the division must be calculated.

For the sum operation, the values will be stored in an **environment variable** called "Acc". This variable will start with the value 0, and later the results of the sums will be added, but **not the results from the multiplication and division operation**.

If the operation is successful, the command will show **using the standard error output**, the result of solving the computation preceded by the label [OK]. In the case of the sum operation, the accumulated value must be also shown, and in the case of the division the remainder.

For the add operation, the results will be shown with the following message:

```
[OK] <Operand_1> + <Operand_2> = <Result>; Acc <Acc_Value>
```

For multiplication, the result will be displayed with the following message:

```
[OK] <Operand_1> * <Operand_2> = <Result>
```

For the division operation, the results will be shown with the following message:

`[OK] <Operand_1> / <Operand_2> = <Quotient>; Remainder <Remainder>`

If the operator does not correspond with the ones previously described, or not all the terms of the equation were introduced, the following message will be shown **in the standard output**:

`[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>`

Here we show some examples of the usage of this command:

```
msh>> mycalc 3 add −8
[OK] 3 + −8 = −5; Acc −5
msh>> mycalc 5 add 13
[OK] 5 + 13 = 18; Acc 13
msh>> mycalc 4 mul 8
[OK] 4 * 8 = 32
msh>> mycalc 10 div 7
[OK] 10 / 7 = 1; Remainder 3
msh>> mycalc 10 / 7
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
msh>> mycalc 8 mas
[ERROR] The structure of the command is mycalc <operand_1> <add/mul/div> <operand_2>
```

Internal command: myhistory

Utility to retrieve the last 20 commands or sequences introduced in the current execution of the *minishell*. This internal command shall conduct the following actions:

- If it is executed without arguments, it will show by the **standard error output** a list with the last 20 commands introduced by the user in the current *minishell* using the following format:

  `<N> <command>`

- If a number between 0 and 19 (inclusive) is passed as an argument, the associated command from the history list will be executed instead of requesting a new command from the parser and will show the message in the **standard error output**:

  `Running command <N>`

  If the command number does not exist or is out of range, the following message will be shown by the **standard output**:

  `ERROR: Command not found`

To implement this parameter, the **20 last commands** entered during the current execution of the *minishell* must be stored. To facilitate the implementation of this internal command, the variable `run_history`, the structure `command` and two auxiliar functions are provided:

- `run_history`: indicates whether the next command to be executed comes from the internal command `myhistory`.
- `store_command` copies a command in the format used by the parser to the provided structure.
- `free_command` releases the resources used by said structure.

Below is an example of use to store a command and free the memory later.

```
// Declaration of command structure
struct  command cmd;
// Store the command received by the parser in argvv, filev y in_background in the structure
store_command(argvv, filev, in_background, &cmd);
// Free resources used by the command
free_command(&cmd);
```

An example of use is shown below:

```
msh>> myhistory
0 ls
1 ls | grep a
2 ls | grep b &
3 ls | grep c > out.txt &
msh>> myhistory 0
Running command 0
file.txt file2.txt
msh>> myhistory 27
ERROR: Command not found
```

Important

The internal commands (**mycalc** y **myhistory**) are executed in the *minishell* process and therefore:

- **They are not part of the command sequences.**
- **They do not have file redirections.**
- **They are not executed in background.**

1.2 Initial Code

To facilitate the realization of this lab you have the file `p2_minishell_2024.zip` which contains supporting code. To extract the content, you can execute the following:

```
unzip p2_minishell_2024.zip
```

To extract its content, the directory p2_minishell_2024/ is created, where you must develop the lab. Inside that directory the next files are included:

- **Makefile**
  It must NOT be modified. Input file for the `make` tool. It serves to recompile automatically only the source code that is modified.
- **libparser.so**
  It must NOT be modified. Shared library with the parser functions. It allows to recognize and parser the input commands.
- **checker_os_p2.sh**
  It must NOT be modified. Shell script that carries out an auto-correction of the student's code. It shows, one by one, the test instructions to be carried out, as well as the expected result and the result obtained by the student's programme. At the end, a tentative score is given for the code of practice (excluding manual review and memory). To run the script the students must change the permissions:

  ```
  chmod +x checker_os_p2.sh
  ```

  And run it:

  ```
  ./checker_os_p2.sh <zip_file>
  ```

- **msh.c**
  This file must be modified to do the lab. C source file which shows an example on how to use the parser. It is recommended that you study the function `read_command` to understand the lab. The current version simply implements an echo (print) of the commands given to msh as arguments, which are syntactically correct. **This functionality must be removed and substitutes by the lines of code that implement the lab.**
- **authors.txt**
  This file must be modified. `txt` file where to include the authors of the practice.

1.3 Compilation and shared library linking

In order to compile and run the code, **it's mandatory to link the shared library libparser.so**, which is in charge of providing to the programmer (students) a useful function that extracts the list of commands given as input to the *minishell*. These commands are saved in a global structure that can be manipulated by the programmer.

1. Create a directory in your preferred folder and remember the *path* and unzip the initial code.
2. Compile the initial code with the provided library.
   - Execute `make` and evaluate the behavior of the compilation.
3. Run the *minishell*:
   (a) If there is an error because the library is not found, the compiler must be told where to search. For example:

   ```
   export LD_LIBRARY_PATH=/home/username/path:$LD_LIBRARY_PATH
   ```

   (b) Run the *minishell* again.

2 Assignment submission

2.1 Deadline and method

The deadline for the delivery of this programming assignment in AULA GLOBAL will be **April 12th 2024 (until 23:55h)**

2.2 Submission

The submission must be done using Aula Global using the links available in the first assignment section and **by a single member of the group**. **The submission must be done separately for the code and report**. The report will be submitted through the **TURNITIN** tool.

2.3 Files to be submitted

You must submit the code in a zip compressed file with name:

os_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip

Where A...A, B...B, and C...C are the student identification numbers of the group. A maximum of 3 persons is allowed per group, if the assignment has a single author, the file must be named **os_p2_AAAAAAA.zip**. **The zip file will be delivered in the deliverer corresponding to the code of the practice.** The file to be submitted must contain:

- **msh.c**
- **authors.txt: csv file with one author per line with the following format: NIA, Surname, Name**

The report must be submitted in a PDF file. Notice that only PDF files will be reviewed and marked. The file must be named:

os_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.pdf

The report must contain at minimum:

- **Cover** with the authors (including the complete name, NIA, and email address).
- **Table of contents**
- **Description of the code** detailing the main functions it is composed of. Do not include any source code in the report.
- **Tests cases** used and the obtained results. All test cases must be accompanied by a description with the motivation behind the tests. In this respect, there are three clarifications to take into account:

  1. If a program compiles correctly and without warnings is not a guarantee that it will work correctly.

2. Avoid duplicating tests that assess the same program flows. The score in this section is not measured by the number of tests, but by the degree of test coverage. Few tests that evaluate different cases are better than many tests that always evaluate the same case.

- **Conclusions**, describe the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the completion of this assignment.

Additionally, marks will be given attending to the quality of the **report**. Consider that a minimum report:

- Must contain a title page with the name of the authors and their student identification numbers.
- Must contain a table of contents.
- Every page except the title page must be numbered.
- Text must be justified

The PDF file must be submitted using the TURNITIN link. The length of the report should not exceed 15 pages (including cover page and table of contents). Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

NOTE: It is possible to submit the lab code as many times as you wish within the deadline, being the last submission the final version. **THE LAB REPORT CAN ONLY BE SUBMITTED ONCE ON TURNITIN.**

3 Rules

1. Programs that do not compile or do not satisfy the requirements will receive a mark of **zero**.
2. **Programs where redirects are implemented in the master thread will receive a severe penalty.**
3. **Programs that execute commands or sequences of commands with processes that are not children of the minishell (child, grandchild, great-grandchild hierarchy) will receive a low grade. It is also not allowed to use statements or functions such as `goto`.**
4. Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be filled.
5. All programs should compile without reporting any **warnings**.
6. The programs must run under a Linux system, it is not allowed to perform the practice for Windows systems. In addition, to ensure the correct functioning of the practice, its compilation and execution must be checked in a virtual machine with Ubuntu Linux or in the Virtual Aulas provided by the Computer Lab of the university. If the submitted code does not compile or does not work on these platforms the implementation will not be considered correct.
7. Programs without comments will receive a **very low grade**.
8. The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.
9. It is mandatory to follow the input and output formats indicated in each program implemented. In case this is not fulfilled there will be a penalization to the mark obtained.
10. It is mandatory to implement error handling methods in each of the programs.

Programmes submitted that do not follow these rules will not be considered approved.

4 Appendix

4.1 Manual (man command)

man is a command that formats and displays the online manual pages of the different commands, libraries and functions of the operating system. If a section is specified, man only shows information about name in that section. Syntax:

```
man [section] open
```

A man page includes the synopsis, the description, the return values, example usage, bug information, etc. about a name. The utilization of man is recommended for the realization of all lab assignments. **To exit a man page, press q**.

The most common ways of using man are:

1. **man section element**: It presents the element page available in the section of the manual.
2. **man –a element**: It presents, sequentially, all the element pages available in the manual. Between page and page you can decide whether to jump to the next or get out of the pager completely.
3. **man –k keyword**: It searches the keyword in the brief descriptions and manual pages and present the ones that coincide.

4.2 Background and Foreground mode

When a command sequence is executed in background, **the *pid* printed is the one from the process that executes the last command of the sequence.**

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

With the background operation, is possible that the minishell process shows the prompt mixed with the output of the process child. **This is a correct behavior**.

After executing a command in foreground, the minishell can not have zombie processes of previous commands executed in background.

5 Bibliography

- C Programming Language (2nd Edition).Brian W. Kernighan , Dennis M. Ritchie.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Operating System Concepts 8^{th} Edition. Abraham Silberschatz, Yale University, ISBN: 978-0-470-23399-3.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (`man function`)