

Computer Architecture and Technology Area

Universidad Carlos III de Madrid



OPERATING SYSTEMS



Lab 1. Programming a command prompt: minishell

**BACHELOR'S DEGREE IN COMPUTER SCIENCE AND
ENGINEERING**



Table of contents

1. Lab Statement	2
1.1. Lab description	2
1.1.1. Provided parser	2
1.1.2. Command line parsing	4
1.1.3. Development.....	6
a) Internal command: globalusage	6
a) Internal command: averageusage.....	7
1.2. Support code	8
2. Assignment submission	9
2.1. Deadline and method	9
2.2. Files to be submitted	9
Appendix	10
3.1. Manual (man command).	10
4.2 Background and foreground mode	10
Bibliography	11

 <p>Universidad Carlos III de Madrid</p>	<p>Degree in Computer Engineering Operating Systems</p> <p>Lab 1- Minishell</p>	
---	---	---

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 environments. In summary, a shell permits 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 systems 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 command lines that later interprets and execute. 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. Provided parser

For the development of this lab a ‘parser’ is given to the student. This parser is capable of reading the commands introduced by the user. The student should only work to create a command interpreter. The sintaxis used by the parser is the following:

A space (blank character) is a space or a tab.

A separator is a character with a special meaning (| , < , > , &), a new line or the end of file (CTRL-D).

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 a immediate child of the minishell spawned by fork command. The value of a command is its termination status, 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 error of each command is connected through an unnamed pipe to the standard input of the following command. A shell 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** → Use *file* as the standard input after opening it for reading (*man 2 open*).
- **>file** → Use *file* as the standard output. If the file does not exist it is created. If the file exists it is truncated.
- **>&file** → Use *file* as the standard output. If the file does not exist it is created. If the file exists it is truncated.

In case of a redirection error, the execution of the line must be suspended and the user should be notified by 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 in the following format:

"[%d] \n"

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

"msh>"



1.1.2. Command line parsing.

In order to obtain the parsed command line introduced by the user you can use the function *obtain_order*:

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

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

- For *ls -l* returns 2
- For *ls -l | sort* returns 3

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 and NULL if there is no such redirection.
- **filev[1]** contains the file name to be used in standard output redirection and NULL if there is no such redirection.
- **filev[2]** contains the file name to be used in standard error redirection and NULL if there is no such redirection.

The argument *bg* is 1 if the command or command sequence are to be executed in background.

Example: If the user enters *ls -l | sort < fichero* the structure of the arguments of *obtain_order* is shown in the following figure:

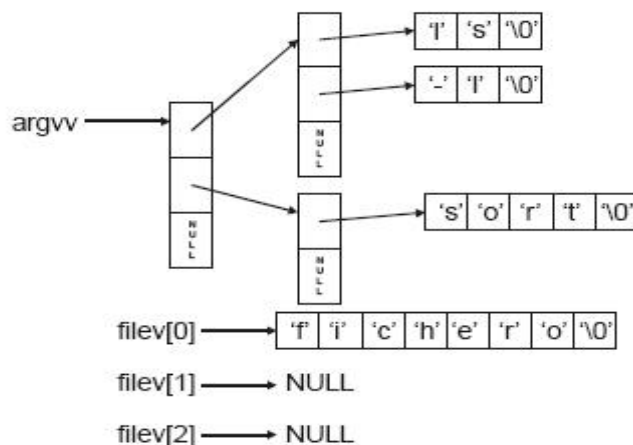


Figura 1: Data structure used by the *parser*.

In the file *msh.c* (file that must be completed by the student with the minishell code) the function *obtain_order* is invoked and the next loop executed:



```
for (command_counter = 0; command_counter < num_commands;
command_counter++)
{
    for (args_counter = 0; (argvv[command_counter][args_counter] !=
NULL); args_counter++)
    {
        printf("%s ", argvv[command_counter][args_counter]);
    }
    printf("\n");
}
if (filev[0] != NULL) printf("<%s\n", filev[0]); // IN
if (filev[1] != NULL) printf(">%s\n", filev[1]); // OUT
if (filev[2] != NULL) printf(">& %s\n", filev[2]); // ERR
if (bg) printf("&\n");
```

It is recommendable 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. Development

It is recommended to approach the development of the assignment in an incremental way (step by step). The different steps are:

- Execution of simple commands such as *ls -l*, *who*, etc.
- Execution of simple commands in background.
- Execution of simple commands with redirection (input, output and error).
- 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 (bigger than 3) will be considered for extra marks.
- Execution of simple commands and sequence of commands with redirections (input, output and error), in background (see Appendix to learn about commands *n* foreground and background to see more details about the requirements of commands in background).
- 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). If it finds any error, a notification will appear (using standard error).

In this assignment, two different commands (based on the minishell *usage level*) have to be implemented. The usage level is defined as the number of commands that are executed. This value can be obtained from `obtain_order` function. The minishell has to create a file named “usage.log” with the related usage level. Every time the minishell is closed (and only at this time) the number of commands executed in the session (without counting the internal commands) are written on the file. Using this information, you have to implement the following internal commands:

a) Internal command: **globalusage**

The code has to read the usage.log file and count the overall number of commands executed since the first use of the minishell. Then, the following message has to be displayed:

MINISHELL USAGE: <n> COMMANDS; CURRENT SESSION: <A> COMMANDS

Where N is the overall number of commands and A is the current number of commands (related to the current minishell session).



a) Internal command: **averageusage**

The code has to read the usage.log file and show the average number of commands executed since the first minishell execution. In order to do that, the code has to read all the file records, to sum them and obtain the average value. Then, this value is displayed with the percentage utilization of the current session (that is obtained as $\text{current_usage}/\text{average_usage} * 100$). This information is displayed in the following format:

AVERAGE USAGE: <M> COMMANDS; CURRENT USAGE: <P>%

Where M is the average usage (without counting the current session) and P is the percentage utilization of the current session. For example:

```
msh> globalusage
```

```
MINISHELL USAGE: 123 COMMANDS; CURRENT SESSION: 13 COMMANDS
```

```
msh> averageusage
```

```
AVERAGE USAGE: 20 COMMANDS; CURRENT USAGE: 50%
```




1.2. *Support code*

To facilitate the realization of this lab you have the file *minishell.zip* which contains the support code. To extract the content you can execute the following:

Unzip minishell.zip

To extract its content, the directory *minishell/* is created, where you have to develop the lab. Inside that directory the next files are included:

Makefile

File for the tool a make. **It must NOT be modified.** It serves to recompile automatically only the source code that is modified.

y.c

C source file. **It must NOT be modified.** It defines basic functions to use the tool lex without using the library l.

scanner.l

Source file for the lex tool. **It must NOT be modified.** It allows you to generate automatically C code that implements a lexicographic analyzer (scanner) that recognizes the token TXT, considering the possible separators (nt j < > & nn).

parser.y

Source file for the yacc tool. **It must NOT be modified.** It generates automatically C code that implements a grammatical analyzer (parser) that recognizes correct sentences of the input grammar of the minishell.

msh.c

C source file which shows how to use the parser. **This file must be modified to complete the assignment.** It is recommended that you study the function *obtain_order* to understand the lab. The current version simply implements an echo of the types lines that are syntactically correct. This functionality must be removed and substitutes by the lines of code that implement the lab.

NOTE 1: For the compilation of the lab code is necessary to have installed the packages correspondent to **Yacc** and **Lex**. In case of implementing the code outside the lab classrooms in personal computers you have to have the lexi and sintactic analyzer Yacc and Lex. In the case of Ubuntu / Debian systems you can install the packages 'byacc' and 'flex' in the following way.

```
sudo apt-get install byacc flex
```

In case of having another Operating System, you must search for the equivalent package for each distribution.



2. Assignment submission

2.1. *Deadline and method*

4 weeks are recommended for this lab.

2.2. *Files to be submitted*

The student must submit the code in a zip compressed file with name `ssoo_p2_AAAAAAAAAA_BBBBBBBBBB.zip` where A...A and B...B are the student identification numbers of the group. A maximum of 2 persons is allowed per group, but it can be done individually. The file to be submitted must contain:

- `msh.c`
- `Makefile`
- `parser.y`
- `scanner.l`
- `y.c`

The report must be submitted in a PDF file. A minimum report must contain:

- **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.
 - Avoid duplicated tests that target the same code paths with equivalent input parameters.
 - Passing a single test does not guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, not the number of tests per program.
 - Compiling without warnings does not guarantee that the program fulfills the requirements.
- **Conclusions**, describing the main problems found and how they have been solved. Additionally, you can include any personal conclusions from the realization of this assignment.

Do not neglect the quality of the report as it is a significant part of the grade of each assignment.



Appendix

3.1. *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] name
```

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 presents the ones that coincide.

4.2 *Background and foreground mode*

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

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.

With the background operation, it 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 cannot have zombie processes of previous commands executed in background.



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 8th Edition.** Abraham Silberschatz, Yale University, ISBN: 978-0-470-23399-3.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (man function)