Computer Architecture and Technology Area

Universidad Carlos III de Madrid

OPERATING SYSTEMS

Programming Assignment 2. Minishell

BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

Year 2018/2019

Table of 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 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. Programming assignment 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. Given 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 syntax used by the parser is the following:

A space 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 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 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 (*man 2 open/ man creat*).
- **>& *file*** → 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*).

In case of a redirection 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 in the following format:

[%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 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.

<u>EXAMPLE:</u> If the user enters:

ls -l | *sort* < *file*

the structure of the arguments of obtain_order is shown in the following figure:

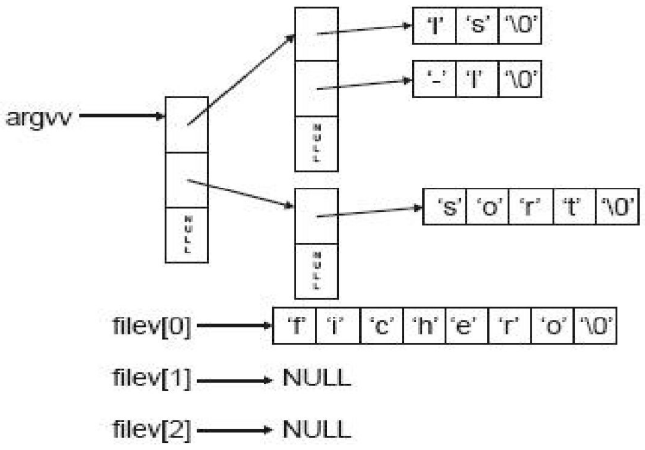

Figure 1: Data structure used by the *parser*.

<u>**In the file msh.c (file that must be filled in by the student with the minishell code)**</u> 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 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. Development

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

- 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 will be considered for additional 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 (not enough arguments or other type of error), a notification will appear (using standard error output). The internal commands to be implemented are:

a) Internal command: **mytime**

This internal command must measure the time it takes a process to execute its function. For example, if the command "sleep 10" is executed in the linux terminal, the process will sleep at least, the indicated time. The purpose of this internal command is to measure that time.

For this, it is recommended to study the *sys/time.h* library and the *gettimeofday()* function. The time obtained must be shown with the message "*Time spent: %f secs.\n*". In case of error in using this command, the following message should be printed: "*Usage: mytime <command <args >>".*

Keep in mind that this internal command should measure the time that takes a process to run, so you must execute a *fork()* and an *exec()*, and measure the time once the child process has performed its function.

An example of execution of this command is the following:

```
msh> mytime sleep 5
Time spent: 5.001719 secs.
msh>
```

b) Internal command: **mypwd**

When this internal command is executed, the path in which the user is at the time of execution, must be printed on the standard output. For the realization of this section it is recommended to review the use of the *getcwd()* function.

The message that should appear on the screen is the following: "*Current dir: <cwd>*" followed by a line break. In case of usage/execution error, the message "Mypwd error" will be displayed by the error output.

An example of the execution of this command is the following:

```
msh> mypwd
Current dir: /home/user/Desktop/minishell
msh>
```

1.2. Support code

To facilitate the realization of this lab you have the file *p2_minishell_2017.tgz* which contains the support code. To extract the content you can execute the following:

tar zxvf p2_minishell_2017.tgz

To extract its content, the directory *ssoo_p2_msh/* 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 do the programming assignment**. It is recommended that you study the function *obtain_order* to understand the assignment. The current version simply implements an echo of the types lines that are syntactically correct. This functionality must be removed and substituted by the lines of code that implement the programming assignment.

NOTE 1: For the compilation of the assignment code it 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 syntax 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*

The deadline for the delivery of this programming assignment in AULA GLOBAL will be **Sunday 31th March 2019 (until 23:55h).**

2.2. *Submission*

The submission must be done using Aula Global using the links available in the first assignment section. The submission must be done separately for the code and using **TURNITIN** for the report.

2.3. *Files to be submitted*

You must submit the code in a zip compressed file with name ssoo_p2_AAAAA_BBBBB_CCCCC.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 ssoo_p2_AAAAAAAAA.zip. The file to be submitted must contain the following file:

- msh.c

The report must be submitted in a PDF file through TURNITIN. Notice that only PDF files will be reviewed and marked. The file must be named **ssoo_p2_AAAAAAAAA_BBBBBBBBB.pdf**. A minimum report must contain:

- **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.
    - o Avoid duplicated tests that target the same code paths with equivalent input parameters.
    - o Passing a single test does guarantee the maximum marks. This section will be marked according to the level of test coverage of each program, nor the

number of tests per program.

  - Compiling without warnings does not guarantee that the program fulfills the requirements.

Screenshots are not allowed.

- **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.

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 an index.
- Every page except the title page must be numbered.
- Text must be justified.

<u>The PDF file must be submitted using the TURNITIN link.</u> Do not neglect the quality of the report as it is a significant part of the grade of each assignment.

The report must be no larger than 8 pages (including cover and table of contents). It is essential to pass the report in order to pass the programming assignment so do not disregard the quality of your report.

<u>NOTE:</u> Only the last version of the submitted files will be reviewed.

3. Rules

1) **Programs that do not compile or do not satisfy the requirements will receive a mark of zero.**
2) **All programs must compile without reporting any warnings.**
3) **Programs without comments will receive a grade of 0.**
4) **The assignment must be submitted using the available links in Aula Global. Submitting the assignments by mail is not allowed without prior authorization.**
5) **The programs implemented must work in the computers of the informatics lab in the university or the Guernika (guernika.lab.inf.uc3m.es) platform. It is the student responsibility to be sure that the delivered code works correctly in those places.**
6) **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.**
7) **It is mandatory to implement error handling methods in each of the programs.**
8) **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.**

Failing to follow these rules will be translated into zero marks in the affected programs.

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] 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 present 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, 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.

4.3 Internal commands

The internal commands (mytime and mypwd) 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.

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)