**Operating Systems**

**PROJECT 2**



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

**Year 2018/2019**

David Rico Menéndez 100384036

Javier Ruiz Plaza 100383436

# INDEX

[**INDEX**](#_xl6ocvwjtf32) **1**

[**Code Description**](#_uj05wnkzcbjo) **2**

[MSH.C](#_o6fvr4nyus16) 2

[Libraries](#_t4f3w8lb9rvz) 2

[Obtain order](#_5hiilk8960nh) 2

[mytime](#_8ciu3ils7rhi) 2

[mypwd](#_rmsco3v9fzqz) 2

[Simple/multiple commands](#_8exjt2muct1c) 3

[**Testing**](#_9aea6djl1jb1) **4**

[MSH](#_kh3qf2taqzrn) 4

[**Conclusion**](#_dg30lbin5jyw) **5**

[Problems encountered](#_xb516we5l03a) 5

[Personal conclusions](#_87xboqrla3rq) 5

[David:](#_cd8mxxn01lel) 5

# 

# Code Description

## MSH.C

### Libraries

We have ended up using a lot of different libraries, used to access some system commands as sys/types, sys/wait or sys/time. Also we needed others more specific, in order to, for example, compare strings in an easy way. (string.h).

### Obtain order

Function that allows the program to read the command lines in the shell, sets the number of commands, stores the commands and its arguments in the variable “argvv” for the shell to process them, tells if any redirection has been specified in the line in the “filev” variable and if the line is to be executed in the background in the variable “bg”.

### mytime

Firstly checks if the command has been inputted correctly in case it has not, printf error message in standard error.

After defining two structures in which we are going to store the start and end values. The first step we decided to do in order to execute the command that we want to measure is “delete” the mytime statement from the command array. To do so, we are going to shift everything one position left using a for loop. Then, we store the first time value before execution, and then fork. The son will execute the command, and father will wait to it to finish, and then get the time again in order to get the time spent executing the child command.

Finally do some maths in order to obtain the time spent in the execution of the command.

The time has a precision of microseconds.

### mypwd

First check if the input is correct after that, we get the directory using getcwd() and then print it in the standard output.

In order to save the directory we use a char array of length PATH\_MAX defined in the library linux/limits.h, even though the directories can has more character than the defined in the constant,getcwd() only reads up to 4096 chars.

### Simple/multiple commands

If none of the internal commands were inputted this part of the code is run.

Firstly we create two array one for storing the pid and other for the pipes. Now in a for loop which will be executed as many time as there are commands in the input line.

At the start of the loop we create a pipe unless it’s the last iteration of the for loop, this means in simple commands no pipe will be create and with multiple commands the last command will not need to create a new pipe. After this a fork is executed, in the child process first we check what kind of command is, a simple one or in case of multiple commands if it's the first, last or neither. In simple commands we skip to the process of connecting pipes, if it’s the first one of a sequence we redirect to output to the pipe, if it's the last one redirect the input and if it's in the middle redirect both input and output, also unnecessary pipes will be closed in the process. Then we check redirections of the input, output or error. Redirections of the input only affects the simple or first command, output also simple commands and last commands, the error redirection will be applied to every command. Finally the child will execute the command, in case of error it will exit with value (-1). In the father process while this is executing it will be closing the used pipes.

Once we exit the for loop if the line is going to be executed in foreground we will need to wait for the children to finish , to do that in a for loop we will wait for each of the child processes created with the fork in the previous loop. In case the command is to be executed in the background we will skip the wait loop.

And finally once the execution is finished before going back to the shell the program to wait for the next line the program will check if any zombies in the background exist. In case there are the parent will wait for them freeing resources.

# Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Test description** | **Input** | **Expected Result** | **Obtained result** |
| 1 | Test for a simple command | ls -l | .Output should b e the same as if executed in the linux shell | OK |
| 2 | Test simple command with redirection | ls -l > indice.txt | Output should be in indice.txt | OK |
| 3 | Test sequence of commands | ls -l | sort -n | wc -l | Output should be the same as if executed in the linux shell | OK |
| 4 | Test sequence of commands with redirection | cat indice.txt | head -n 4 > out.txt | Output should be in out.txt | OK |
| 5 | Test sequence of commands with multiple redirections | sort -n | head -n 1 <file.txt >file.text >&file.txt | Input is taken from file.txt and then output is written back in file.txt | OK |
| 6 | Test internal command mypwd | mypwd | Output should be current directory | OK |
| 7 | Test error internal command mypwd | mypwd ls | Output should display the error message | OK |
| 8 | Test internal command mytime with no arguments | mytime | Output is the time of the parent forking and calculate the time. | OK |
| 9 | Test internal command mytime | mytime ls -l | Output is the time spent executing ls -l | OK |
| 10 | Test error message internal command mytime | mytime ls -l | ls -l | Output should be error message as the mytime command should have not redirections | OK |
| 11 | Background testing.  Sort sleep in bg follow by large sleep in bg then execute command in foreground | sleep 1 &  sleep 100&  ls -l | Output displays correctly the pid when a process is run in the bg and after the ls executes the child executing the first sleep is freed by the parent | OK |
| 12 | Sequence of commands in which the first one does not give output and the second requires input | sleep 4 | head | Output should be nothing and the shell should reappear after sleeps ends | OK |
| 13 | Sequence of commands with various sleep, checks if the programs run concurrently | sleep 2 | sleep 3 | sleep 6 | Output should be nothing the shell should reappear after 6 seconds | OK |
| 14 | Sequence of commands where one input is wrong | lss | sort -n | Output should display the error related to lss and sort do nothing | OK |

# Conclusion

## Problems encountered

Connecting the children with pipes, first we had a similar version of the current program in order to set the pipes, but at the start of the execution after reading the line we would create all the pipes and then assign the pipes. This caused various problems due that we had to close much more pipes in each iteration, for example that even when we redirect the input to a pipe the command will still request a input from the keyboard.

This problem was solved by instead of creating all pipes at once, the program will be creating them as it's necessary making easier the clean up of the pipes.

Waiting children executed in the background, at the start to try to solve this problem we would had an array were we would store all the processes executed in the background and then when the user executes a line in the foreground we will send a signal to every process in the array in order to terminate them, this caused that some child would be terminated even if their execution has not been completed and most important break the rule of not sending kill signals to the children.

This problem was solved by completely removing the kill stuff and putting at the end of each execution waitpid with the option WNOHANG which will check if any child is terminated and need to be freed by the parent, returns immediately if no child is there to be terminated.

## Personal conclusions

### David:

Personally, during this project I found myself having some issues in understanding how the threading will act in some specific cases. Luckily, my colleague had been able to explain me how to solve and that, plus the help of the teacher during the last revision, made me finally understand the main points, having now a clear and wiser vision of our program. In the other side, I found pipes easier than I expected.