PRACTICE 2

Sheila Gallardo Redondo
NIA: 100472913
Mail: 100472913@alumnos.uc3m.es
Group: 121

Alexandra Perruchot-Triboulet Rodríguez
NIA: 100453024
Mail: 100453024@alumnos.uc3m.es
Group: 121

Contents

Description of the Code

Introduction

We start the code checking what type of command has been introduced in the mini shell. To do that, we extract the first command with its parameters (from the first row of the *argvv* matrix) and insert it into the *argv_execvp* vector with the function *getCompleteCommand*. If the command isn't any of the internal commands *mycalc* nor *mytime*, then it means that it is another internal command, and, in this case, we can use pipes, redirections and background. To separate these three cases, we have used an if/else structure comparing the first element of *argv_execvp* (the name of the given command) with the possible internal command's names.

mycalc internal command

For this command, we start checking if the syntax is correct: (*mycalc* <int><operation><int>). To do that, first we check that we are given three arguments. In case we are given less/more arguments than the expected, we write an error in the *standard output file* with the *write_err* function (defined by us outside the main) that works with the *write* syscall.

Then, if everything was fine, we convert the first and the third parameters into integers (with the *atoi* function) and store them in *operator1* and *operator2* respectively. If the given parameters are not integers, the *atoi* function returns 0, so, in order to check that the first and the third parameters are truly integers, we consider the case where we are given a "0" as an argument. If it is not the case, we write an error message into the *standard output file*.

We check that the operator is one of the three valid operators (add, mul or div), if not, we write again an error message in the *standard output file*. We then store the second parameter in the *operation* variable (set with a pointer) and create a buffer to store the formatted string (the conversion of the output into a string) to write it later in the *standard error file* and so it can be displayed in the terminal.

Now, we can start checking which of the three operations is stored in the *operation* variable. In the case that it is the *add* operation, we store the addition´s result of the two given integers in the *result* variable, we get the value of *Acc*, the environment variable (defined previously by us before the while loop of the mini shell starts), that, initially is set to "0", convert it into an integer (using atoi) and store it in *acc.* We add the result of the addition to the one stored in *acc* and store it again in *acc*, then, we convert the value of *acc* into a string (with the *snprintf* function and the *buffer*) and store it again in the environmental variable

Acc. Finally, we write the *result* and the *acc* values in the *standard error file* with the given format: [OK] <*operand1*> + <*operand2*> = <*result*>; *Acc* <*acc*>

Moreover, in the case that it is the *mul* operation, we store the multiplication of the two given integers in the *result* variable and write it in the *standard error file* with the given format: [OK] <*operand1*> * <*operand2*> = <*result*>

And in the last case, where it is the div operation, we store the integer division of the two given integers in the *result* variable and the remainder of that division in the *remainder* variable. Then, we write it in the *standard error file* with the given format: [OK] <*operand1*> / <*operand2*> = <*quotient*>; Remainder <*remainder*>

We do this checking for errors and writing them in the *standard output file.*

mytime internal command

When the command given is *mytime*, we start checking the syntax, like we did in the previous case. Here, there are no parameters, so we only must check that the first parameter is *NULL*, if not then we write an error message in the *standard output file* (with the *write_err* function explained before). Then, we create another buffer (*buffer1*) to transform the output into a string, so that we can write it later in the *standard error file*. We then define a variable (*time_shell*) to store the total time in seconds (given by the thread variable *mytime* in milliseconds and dividing it by 1000 to convert it in seconds) and another one to store the final number of hours, minutes, and seconds. To obtain the hours, we do an integer division between *time_shell* and 360, then, we perform an integer division between the remainder of the previous division and 60 to obtain the minutes, and, finally, to obtain the seconds, we take the reminder of the last operation. Then, we convert the result into a string with the given format (HH:MM:SS) using the function *snprintf* , which stores the formatted string in *buffer1* to be able to write it in the *standard error file* with our *write_err* function.

General internal commands

In the last case, we are given the rest of commands. Here, we can have a sequence of commands and, to manage this, we need pipes. For that, we have declared the array *fd_pipe* for the file descriptors associated with the pipe and we have created the pipe. Also, we have declared the variable *previous_output* (stores the output of the previous command) to help us later with the pipe. We can have a pipe only if the command is not the last one (then if there is only one command, we don't need to create any pipe).

Then, we use fork to create a new process (the child process) to execute the given command. Inside the child process, we start checking for redirections. The input redirection (<), only affects the first command, so we do an *if-else* to check if we are in the first command, and, if it´s the case, we check if there is any redirection for the input, we do that by looking if there is something in *filev[0]*. If there is, we close the *standard input file* and open on its position the file given by *filev[0]*. In the case where the command is not the first one, we must take as its input the output of the previous command. To do that, we close the *standard input file* and open the file stored in *previous_output* (where we are going to store later the output of the command).

If there is an output redirection (>), it only affects the last command, so we do an *if-else* to check if we are in the last command, and, if it´s the case, we check if there is any redirection for the output, we do that by looking if there is something in *filev[1]*. If there is, we close the *standard output file* and open on its position the file given by *filev[1]*. In the case that the command is not the last one, we must take as its output the output of the pipe (*fd_pipe[1]*). To do that, we close the *standard output file* and duplicate the output file descriptor of the pipe in that position.

Finally, if there is an error redirection (checking if there is something in *filev[2]*) we close the *standard error file* and open on its position the file given by *filev[2]*. Now, with all redirections done, we can execute the given command with the *execvp* syscall.

In the parent process we take care of the background, if there is no background the parent process waits until the child process has finished. On the other hand, if there is background, the parent process doesn´t need to wait and it prints on the screen the pid of the process in background.

The parent process closes the files we are not going to use anymore (*fd_pipe[1]* and *previous_output*) and assigns next command´s input with the previous command´s output (*previous_output = fd_pipe[0]*).

We repeat this process with every command checking for errors everywhere.

Test Cases

mycalc internal command

Tests	Expected Output	Actual Output
mycalc 12 add -8 mycalc 29 add 5	[OK] 12 + -8 = 4; Acc 4 [OK] 29 + 5 = 34; Acc 38	[OK] 12 + -8 = 4; Acc 4 [OK] 29 + 5 = 34; Acc 38
mycalc 7 mul -2	[OK] 7 * -2 = -14	[OK] 7 * -2 = -14
mycalc 44 div 3	[OK] 44 / 3 = 14; Remainder 2	[OK] 44 / 3 = 14; Remainder 2
mycalc 5 add two	[ERROR] The structure of the command is mycalc < operand_1 > <add / mul / div > < operand_2 >	[ERROR] The structure of the command is mycalc < operand_1 > <add / mul / div > < operand_2 >
mycalc 21 * 9	[ERROR] The structure of the command is mycalc < operand_1 > <add / mul / div > < operand_2 >	[ERROR] The structure of the command is mycalc < operand_1 > <add / mul / div > < operand_2 >

mytime internal command

Tests	Expected Output	Actual Output
mytime	00:01:52 (time the minishell has been executing)	00:01:52
mytime hello	[ERROR] The structure of the command is mytime	[ERROR] The structure of the command is mytime

General internal commands

Tests	Expected Output	Actual Output
cat out.txt (read content of a file)	hello	hello
ls | sort | wc (the output of ls is passed as input to sort and the output of this command is passed as input to wc which will count the number of words, lines and bytes of the directory)	18 18 293	18 18 293

ls | **cat** > out.txt < in.txt (in.txt has the name of a directory (p2_minishell_2023) and it´s passed as input to the ls command. The ls command lists the corresponding content of that directory and this is stored in the out.txt file)	Nothing is displayed on the screen. But the content of out.txt: Makefile Makefile:Zone.Identifier checker_os_p2.sh checker_os_p2.sh:Zone.Identifier error.txt in.txt libparser.so libparser.so:Zone.Identifier msh msh.c msh.c:Zone.Identifier msh.o msh.save os_p2_100472913_100453024.zip out.txt	Nothing is displayed on the screen. But the content of out.txt: Makefile Makefile:Zone.Identifier checker_os_p2.sh checker_os_p2.sh:Zone.Identifier error.txt in.txt libparser.so libparser.so:Zone.Identifier msh msh.c msh.c:Zone.Identifier msh.o msh.save os_p2_100472913_100453024.zip out.txt
more | **wc** < out.txt & (the command "more" receives as input the out.txt file and it will return the content of the file, this will be passed to the command "wc" as input and the output will be the number of words, bytes and lines of out.txt which will be displayed on the screen. Executed in background)	MSH>> 15 15 228	MSH>> 15 15 228
wc hee.txt !> error.txt (the hee.txt does not exist, then, it will give an error. Instead of displaying the error in the screen, it will be redirected to the error.txt file)	Nothing is displayed on the screen but the content of error.txt is: wc: hee.txt: No such file or directory	Nothing is displayed on the screen but the content of error.txt is: wc: hee.txt: No such file or directory

Conclusions and Errors encountered

During this lab, we experienced some complications. The main complication for us was to understand what to do, we were a bit confused with the initial script given to us (msh.c) cause we didn´t know what each of the lines of the code was doing and how we could obtain the command entered by the user to work with it.
Once we started to write our code, we struggled with the internal command (*mytime*) cause we didn´t notice that there was a variable already defined where the time the minishell has been running was stored, in milliseconds, so we only had to divide by 1000 to obtain this time in seconds. Instead, we looked on the internet for a function to obtain this information, but finally it didn´t work as we thought.
Moreover, we also had complications with the background execution. We didn´t know if execution at the background implied that the parent had to wait, or instead could go on without waiting for the child. Finally, we understood that with background, the parent doesn´t need to wait for the child, but, of course, if there´s no background, the parent should wait for it.

Aquí ponemos un par de cosas más

(Conclusion)

To conclude this report, we think this lab helped us to improve our knowledge in several topics such as pipes or execution in background.