LAB ASSIGNMENT 2

-

Iván Miguélez García - 100383387

100383387@alumnos.uc3m.es

José Manuel Frías Salvachúa - 100383533

100383533@alumnos.uc3m.es

TABLE OF CONTENTS

1. Introduction
2. General structure of the code
3. Cases faced and how were solved
   1. Command sequences
      1. Executing single commands
      2. Executing a sequence of commands
         1. Sequence of three commands
         2. Sequence of n commands
   2. Redirections
      1. Executing single commands with redirections
      2. Executing command sequences with redirections
   3. Background
      1. Executing single commands in background
      2. Executing sequence of commands in background
      3. Executing single commands with redirections in background
      4. Executing command sequences with redirections in background
   4. Internal commands
      1. Mytime
      2. Mypwd
4. Tests
   1. Command sequences
   2. Redirections
   3. Background
   4. Internal commands
5. Conclusions and main problems found

**INTRODUCTION**

Using the knowledge we acquire during the classes, we completed this lab mainly with system calls for creating child processes from a parent process, executing commands or waiting for child processes to return something to the father when finishing, among other things.

**GENERAL STRUCTURE OF THE CODE**

The main function starts with a while loop that will run always. First, the parent process, which is the minishell will check for signals from any child that has dead without the parent waiting for it, and remove it from the PCB. Then the function obtain\_order() will give us the information of how many commands are entered into the shell. Before starting to check how to execute the commands, the code will check if the internal commands “mytime” or “mypwd” were used. If this is the case, the code for those commands will be executed. Otherwise, child processes will be created (one for each command). If the command or command sequence is executed in background, the parent process (the minishell) won´t wait for the process to finish. Instead of that, it will complete the while iteration, start the while again, and wait for an input command. If commands are not executed in background, the parent just wait for the children to finish. Redirections are checked when the last command of the input argument is about to be executed.

**CASES FACED AND HOW WERE SOLVED**

COMMAND SEQUENCES

* Executing single commands

In order to execute a single command, we just need the parent to create a child with the fork() function. Then, the child will execute the input command, while the parent waits for the child either to finish (successfully executing the command) or returning -1 (which is the value that the function execvp() returns when there was a problem during the execution of the system call).

* Executing a sequence of commands
  + Sequence of three commands

For executing a sequence of commands, we needed to create a pipe, so that the output of a command becomes the input of the next one. Here is were we found the first problem of the lab: we couldn’t create pipes between children because they existed in different periods of time. We finally decided to create a pipe between the children and the parent processes, where the parent behaves as a buffer, storing the output of the children in the piped file descriptor, that will be used later by the children when the parent forks. When the last command of the sequence is executed, the output of the child goes to the standard output (screen) instead of being stored in the parent. This worked fine for two commands, but no for three. The second problem had just appeared. We found out that in order to reuse a pipe for storing information in the parent for the second time, we first needed to close and reopen the pipe, something we couldn’t do because we would lose the output of the previous command if we did it. A second pipe was created to solve this problem, so the second command received the output of the first command through the first pipe, and the third command received the output of the second command through the second pipe. Doing this, and taking into account that the output of the third command should go to the standard output, as in the case for two commands, everything works fine.

* + Sequence of n commands

Based on the previous experience with command sequences, the first idea that we came up with to solve this problem was to use one pipe per command. However this didn’t seem like an efficient way to solve the problem. Then we realised that we didn’t need more than two pipes, because the code would never use two pipes at the same time. When the file descriptors of one pipe where being used, the other pipe was closed and reopened to be used later. As we needed to alternate the uses of pipes, we came to the conclusion that the best way to do it was to use one pipe for commands in an odd position of the input and another for the even ones. The if conditions just help us gather intermedial and final states of each of them.

REDIRECTIONS

* Executing single commands with redirections

\**/Explanation referring to the initial design before applying command sequences/\**

For applying redirection in single commands we use the same concepts as used in pipes. If, after applying the parser, we found that any of the spaces of the array filev[ ] wasn’t NULL, the user had entered a redirection with their command and we should apply it. If filev[0] wasn’t null, the user had entered an input redirection. This means we should close the standard Input of the command (in many commands this is the keyboard) and open the filev[0]. This way, the file descriptor 0 will change to the file entered in filev[0]. If the file, doesn’t exist, the opensystemcall will return an error value so the child process will exit with an error value -1 and won’t execute.

If filev[1] is not NULL, the user has entered an output redirection. In this case we will close the Standard Output (usually the screen) and open filev[1]. This time, to the open function we apply flags RDWR, O\_CREAT and O\_TRUNC. O\_CREAT will create the file if this one doesn’t previously exist, O\_TRUNC will delete what is written on the file leaving it blank, and O\_RDWR will take care of the permissions allowing to read and write in that file. First time designing the output redirection, we didn’t apply the O\_RDWR flag so a problem usually arose telling us there was an error while writing and “*bad file descriptor”*. This happened because the minishell was trying to write the execution of the command in a file that didn’t allowed it. Once this was applied, everything worked well. Same as before, if any problem arose while opening (or creating), we would exit the child with a -1 value indicating error.

If filev[2] wasn’t NULL, the user had entered an error redirection. Same as before, we closed the standard error and opened the filev[2]. We will act the same as explained in the previous paragraph, both in the terms of the Flags and possible errors.

* Executing command sequences with redirections

With command sequences, the theory explained before still remains: “Close the standard input/output/error of the child and open the corresponding filev[ ] as explained before. The problem was where do we apply this redirections.

Our code for preparing the child processes before executing is divided in 3 parts. First we take care of the input, then the output and last (if needed) the error. Redirection input should only be assigned to the first command of the sequence. The way the conditionals are established in the code leave us 2 possibilities. First we have initial child in command sequences. If filev[0] is not NULL we act as explained before and then change our control variable “*already”* to 1 (this will be explained later). If it isn’t NULL we keep everything as already is. Second we have the possibility of only having 1 command. With our command sequence programmed and in order to keep code compact, we think of this single command as an even last command(if only 1 command, this will always be it’s last command in it’s sequence and command counter will be 0). We should also apply a control variable for avoiding that this redirection is done at the end of all even command sequences. This is when the control variable “*already”* shows it’s importance. If we are in a command sequence of more than 1 command, the first case has been applied so the “*already”* variable will be equal to 1. This will avoid the last command entering the code section in charge of the redirection. If the variable value is 0, it means we haven’t apply the initial redirection so we must have only 1 command. This will allow the process to enter this part of the code and apply the redirection. Each time a command sequence it’s executed and the while loop is started, this control variable is restarted.

For the output redirection is much easier because we know it can only be apply in the last commands. If filev[1] is not NULL we simply look for the last commandsand apply redirection as explained before.

Error redirection is done over all the commands of the sequence so it’s outside any conditional barrier and apply to everyone before the execvp(). If a redirection is not needed,this code section will be ignored.

BACKGROUND

* Executing single commands in background

Executing a command in background it’s quite easy. We add a conditional in the parent process saying if variable “*bg*” equals to 1, we know user wants to execute the command in background so we don’t apply the wait system call (leaving child zombie after finishing it’s execution). We also print the number of the command that is being left in the background and it’s pid(this has been store in pid1 while doing the fork(). We will talk about the elimination of zombies in the section of command sequences.

* Executing sequence of commands in background

When talking about command sequences in background we must first apply what we have talked in the single command section: simply don’t apply the wait systecall. This will leave for sure zombie process. We know that when a child process ends, it sends a SIGCHILD signal to its parent. For eliminating the zombies we simply apply a handler for this signal. Our handler executes a wait system call so the process won’t be zombie anymore and its PCB position will be freed. We use waitpid with parameters -1, NULL and the WNOHANG. -1 says that the wait system call should be activated with any returning state its receives (we are not waiting for an specific pid. WNOHANG is a flag that avoids the block of this wait system call, just collect the returning status if there was any. The combination of both arguments will take care of the possible zombies. This waitpid could be put in the code and not be used in a handler but putting it there assures that it only activates when a process ends.

We can also see that because there is no blocking call for the background command sequences, we can execute multiple at the same time without interfearence

* Executing command sequences with redirections in background

This is simply a combination of the two previous points. Redirections and background don’t interfere with each other in our code so we simply act as stated before.

INTERNAL COMMANDS

Every internal command is executed by the parent process (the shell).

* Mytime

The way of executing mytime is typing another command after it, so that the shell returns how much time did it take to the computer to do that command. We first take an initial timestamp and we proceed to modify the argvv shifting every position to the previous position and eliminating the mytime. We then execute in the same fashion as before. This makes it possible to take the time of a ocmmand with various arguments. Once the child process has ended (and the parent process has received the state), we take another timestamp. Last, for calculating the time we simply substract the first one to the last one, thus finding how much time has passed, and we print it.

* Mypwd

The performance of this command is very simple: if it is received, the system call getcwd() will return the current working directory in a buffer, we print this buffer and, thanks to the if condition, we end and start the while loop again so the shell will wait for another input. If an error occurs while executing getcwd(), it will return NULL so we will have a perror telling us about this.

**TESTS**

* **Testing command sequences having an valid initial input redirection:**
  + **Pre:** No processes in the background, zombies or command executing
  + We want to test that the commands in the sequences use the pipes properly, passing their outputs to each other. We also want to test that the input value is taken only by the initial command and then process by the sequence.

*msh>sort|head -3| tail -2| grep a|wc<filetest*

* + **Post:** The information of the input is well process through the sequence and last command takes as input what was passed as output from previous command.
* **Testing background commands**
  + **Pre:**No processes in the background, zombies or commands executing.
  + We want to test that when process are send to the background are actually executed no matter how many they are or how much time they need. Also we want to assure that when when they are processes been done in the background, we can still be doing things in foreground. For this we will first introduce a command sequence that takes some time (sleep 5&), then another command in background to check that after the first one is finish, this command is also done. Last, we execute some commands in foreground to check that they can be executed while this is going on(we chose “ps” for controlling the different process that were in execution).

*msh>sleep 12&*

*msh>sleep 5&*

*msh>ls&*

*msh>ps*

*msh>ps*

* + **Post:**Both commands in background and foreground were executed leaving no zombies behind.
* **Testing output redirection in background** 
  + **Pre:**Long sleep process in the background, no zombies.
  + First of all we will like to test that while we have long executing process in the background, more processes can be executed as well. Next introduce a command with an output redirection. This will also help to check if the output redirections works well.

*msh>sleep 1|ls|wc>filetest&*

* + **Post:** Output of the command sequence was written on the designated redirection and no zombies were left behind
* **Testing correct functioning of mytime**
  + **Pre:**No processes in the background, zombies or commands executing.
  + Introduce a command as argument for our internal command. Test the well functioning of the execution of the argument command (by printing correct output), the recognition of the mytime command in the user input line and the well measuring of time. For this we can work with a sleep command. Expected time will be one near the value of the argument of the sleep.
    - Variant 1

*msh>mytime sleep 3*

* + - *Variant 2*

*msh>mytime ls*

* + **Post:** Programs works correctly printing on the screen the desire message as well as the expected time.
    - *Variant 1*

*Time spent: 3.00227 secs*

* + - *Variant 2*

*Output of ls + the time spend*

* **Testing wrong command in the middle of a sequence**
  + **Pre:**No processes in the background, zombies or commands executing.
  + Make sure that if a wrong command is introduced in the middle of a sequence the while loop restarts and doesn’t stay stuck in the process.
    - *msh>cat|hfslhfg|head<filestest*
  + **Post:** Msh prompt appears again without any problems and without letting anything back as zombie or something else
* **Testing invalid input redirection file**
  + **Pre:**No processes in the background, zombies or commands executing.
  + Test that the shell receives and error value, prompts it and restart the while loop
    - *msh>cat|sort<sofgjrrvgvgbsgrh*
  + **Post: “**Bad file descriptor” and “Error opening” printed, and while loop restarted

**CONCLUSIONS AND MAIN PROBLEMS FOUND**

Having a parent process in charge of creating child processes to perform part of the job in order to increase the efficiency of a program, is the main conclusion that we are coming up with.

By scaling the complexity of the problems that we were facing from the simplest ones to others that required more time to develop a working solution, we were unconsciously creating a structure, which would lead us to new solutions for our next problems that we hadn’t thought at the beginning.

We also thought we should give more importance to modularity, as it’s more difficult to debug a program in the linux shell than in a proper developing environment such as eclipse. Then, later, when every bug was solved and every problem had a solution, we try to optimize the code as much as possible.