PA 2 REPORT

I've used C++ for this assignment since I was familiar with stringstreams. I found it easier to work with while parsing the commands. Since I was working with stringstreams I didn't need to do extra work to escape special characters.

The program first opens up the commands.txt file. I created 3 vectors namely; commands (a parsed command vector) which are for the commands, backprocess which are for the background processes and lastly backthreads which are for the threads that I am going to use. For the commands I've created a struct called parsedCommand which has the following attributes: command (first string in a whole command), option, input, redirection(either < or >), background (either y or n depending on if the command has & symbol or not) and lastly textword (the word associated with the redirectioning). After declaring these vectors and the struct, the program starts reading the commands.txt line by line. Later I declare the default values for the parsedCommand struct. The program starts to read the line word by word. I first check if we are on the first string which would be the command value for the parsedCommand. After that, I check if the command was "wait". If not I then check if the word had a "-" at the beginning. This would indicate that it's an option for the command so I assign it. I check if the word is &. This would indicate that the command is a background processs. I check if the word is a redirection sign (<, >). If so I assign it to redirection variable of my struct and set txtword variable to true indicating that the word after this word is the textword of my struct. Later I assign the textword. If none of these options are matched I assign the word as an input. After making the assignments I push the command into my commands vector. After that since I'm done with file operations I close the file and open another file which is parse.txt.

Later I start to go over the commands that I've pushed, inside a loop. This will be the main loop of the program. First thing I do is to print the desired values from my struct into parse.txt.

After that the program checks if the command is wait. If the command is wait the program waits for all the background processes that are pushed to backprocess vector to finish and for all threads that have been pushed to backthreads to finish. (I will cover these vectors later).

If the command is not a background process and it doesn't have a redirection command the program does a fork call (fork()). If it didn't fail, the child process parses the command again in order to put the elements into a fixed array one by one. After parsing the command into a string it puts all the words of this string one by one into an array and execute it with execvp command. I tried to execute the commands the same way I would do in a C file therefore I had to convert the strings to C strings (c_str()). Meanwhile parent process waits for the child process to finish and then it checks if the program is on the last iteration which would mean that this is the last chance to wait for all the background processes and to join the threads.

If the command has a redirection symbol; I check what that symbol is. If it is a > symbol; the program makes a fork call. The child process closes the standard output and opens up the file that is specified inside the command meaning that the output of the following process will not be printed to the console but to this specified file. Then I parse the command again the same

way and execute it with execvp. The parent process checks if this command is a background process. If it is; it pushes the process into backprocess vector (this vector holds the background processes' ids). If it is not a background process parent waits for the child process to end. And I do the last command iteration check again. This last check is on every edge case therefore I won't mention it anymore.

If the command's redirection symbol is >; the program declares a file descriptor in order to do piping (int* fd = new int[2]). In the child process, I close the standard input and open up the specified file in the command. This would result in taking the specified input file as input. Therefore when execvp is executed, it will get the input from this file. I also use the dup2 command to put the output of this operation into the pipe's write end. I parse the command, put it in an array and execute it again. In the parent process, if the command is a background process, I add the current process to background processes while also creating a thread with the printThread function. Then I push it to backthreads vector. The printThread function takes the file descriptor as a parameter. It reads the pipe with file streams and prints it. It utilizes the global variable printLock which is a mutex variable. I used it for the input to not be interrupted. If it is not a background process, parent waits for the child process to end again and then create a thread. After creating the thread it is pushed to backthreads again.

If the command is none of the above conditions; the program makes a fork call again. The child makes a dup2 call, redirecting the output of the current process to the pipe (when the execvp command is called it will be in the pipe not in the terminal). After executing the command, we switch the parent again and parent checks if the current operation is a background operation. If so, it pushes the current process into backprocess vector, it creates a thread with printThread function again and pushes it to backthreads vector. If the process isn't a background process, parent waits for the child and creates a thread, then pushes it to backthreads.

Ural Sarp Sipahi 28093