Zeynep Türkmen - 29541

Programming Assignment 2

I wrote my program using C++.

Program Flow is as follows:

I first started with reading the commands.txt using file streams and then parsing it. I also initialized two vectors, first for keeping the child process IDs and the other for the thread identifiers.

I used **getline** to read each code line first, then I used **string streams** to read those lines word by word. Inside the **getline** I declared the char pointer array to store the arguments for the **execvp** and initialized it with 4 elements, all of them as null (I figured the **execvp** will take at most 3 arguments = the command, options, and the input. The 4th element is always **null** and declares the end of the array for **execvp.**)

After that I read the first word into command[0] as the first word is always the command to be executed. I then used another loop to read the rest of the words in that line, parsing them along the way checking all possibilities and assigning the variables for inputs, options, redirection, background.

To make sure execvp got the arguments correctly and interpreted dashed arguments as options, I first converted the argument strings (like "-a") into a **c_str**. This method helps convert the string into a const char* array and it also adds /0 to the end to simply denote the end. Converting it this way allowed the arguments to be **null terminated** and helped execvp interpret them as intended. I then used **strdup()** to duplicate that string, which helped to not modify the original argument, and allowed execvp to work correctly and modify its arguments.

After I printed out the parsed result into parse.txt. and used **flush()** to make sure everything was received.

Then, I added an if check that checks whether the command is **wait** or something to be executed.

IF **wait:**

> I looped through all the child processes in the processID vector, calling **waitpid** on all of them (I researched and found this function waitpid which does the same thing as **wait (NULL)** but it also takes the pid of the child and waits for that given child. In this case it helped me a lot better)
>
> Then I looped through all the **listener threads** calling **pthread_join** on them. Then of course I cleaned up the vectors for memory efficiency.

This way the program will make sure to wait for all the listener threads which print on the console AND it will also wait for the background processes that use files as their output thanks to **waitpid** (as I did not need threads for them).

IF it's a command to be executed:

I used 2 conditions = printing to console OR printing to a file. I used listener threads for printing to the console, but I didn't use threads to print to an output file as it was unnecessary.

IF printing to a file:

I simply forked the process.

Child => I changed the output file descriptor as the file I opened using dup2, then simply closing the unused descriptors and calling the execvp.

Parent => I checked whether it was a background process or not

No => I simply called waitpid on the child, so the parent program waits for the execvp command before looping through the rest of the commands.

Yes => I added it to the process id array so that I can call wait on it later on if necessary. After this the parent continues reading new lines from the commands

ELSE printing to the console:

NOTE => In my design, all the commands are executed under the getline function therefore I initialized **separate pipes for each command** which made things a lot simpler.

I initialized a new pipe first before forking.

Child => Checks whether it's a redirected input and changes the STDIN using dup2. Then it changes STDOUT to be the write end of the pipe again with dup2, closing the unused ones ofc. Then it calls execvp.

The parent => First it creates a listener thread.

If it's background => It pushes the PIDs and Thread identifiers into dedicated vectors, moves on to the get a new command.

Else => It waits for the child process to terminate and joins the related thread.

Listener thread function => Listener thread takes an argument which is the identifier integer for the read end pipe of that specific command, it then sets up a flag variable as true which is used to run the loop until we printed everything.

As of suggestions I used file streams. I used **FILE *** and opened the read end of the pipe with **fdopen** and iterated through the lines using **fgets**.

Inside while(flag) the threads first try to acquire the lock (the global mutex) then it opens the file with **fdopen**, checks if it can read one line from it with **fgets.**

IF it cannot read: Means the child didn't write into the pipe yet so it gives the mutex back and keeps looping.

IF yes: It goes into another while loop with **fgets** and prints the **TID** and the pipe contents to the console, syncs the output with **fsync**, sets the flag as false, **gives up the mutex** and terminates.

This way the thread will keep looping until it gets the mutex and is able to print everything to the console. Also, since all the print functions are within the mutex and a thread doesn't give it up until it prints everything or when pipe is empty, the results won't get garbled.

After reading all the lines. The child's keep executing the commands, the parents' listener threads still working on the background listening the results. And the parent (main) simply closes the files, clears the vectors. Then uses the same wait method I explained above to make sure every command was printed before terminating.