27760 Utku Alkan PA2 Report

Firstly, I created a struct and put all of the commands in commands.txt to a vector of that struct with parsing.

Then I created for loop for looking through every struct element which is parsed lines in commands.txt.

Then I forked the shell process and in the child, I handled the situation which is output redirectioning. It just writes the output to a file and there is no pipe.

After also again in child, I handled input redirectioning and no redirectioning parts by writing them into a pipe with the help of dup2. Since it is in "for loop" and "for loop" iterates through every line of the command.txt my code creates a pipe per command.

After pipe writing is finished in the parent I checked if it is a background job or not. If it is not we call wait and wait for its child process. After waiting we can read the pipe with dup2 again and put the pipe output into a buffer. By using that buffer, we create our thread. The buffer is given as argument. I will talk about printer function at the end so I am skipping it right now. After creating the thread I joined the thread right after since it is not a background job so it is finished.

If it is a background job every thing is the same except wait command and joining. Since we want to make this implementation multi-threading we cannot join right after. I created a threadArray for this implementation and put all the threads into this array after it is created. Note that, when I tested my code buffer was problematic since one thread was able to change another thread's buffer. That's why I created also a buffer array that will hold the thread buffers in a different element of the array. So overwrite issue is solved. Also, it is important to note that I am doing these for only no redirectioning or input redirectioning so when output redirectioning happens parent does nothing. The process is finished in the child.

Then we came to our multi-threading "for loop" in the main (shell process). We iterate through our threadArray and join them one by one. So we can be sure none of the threads is unfinished when the main ends. And, the most important thing is there can be multiple threads at the same time that are not joined yet. So background jobs can run in any order.

For my wait implementation, I actually did the same thing as the multi-threading "for loop" part. So, if the command is "wait", it joins all of the background threads in a for loop, and it is finished.

Lastly, I want to mention about my synchronization. When creating threads I called a function called printer. It writes the buffer of the thread between the IDs of the current thread. For this thread implementation to be correct, a context switch shouldn't happen in this printing operations. So I used mutex lock at the beginning and mutex unlock at the end. Since all of our threads are created in this way none of the threads can write their buffers or thread IDs intertwiningly. All printing processes became like an atomic instruction.