CS 307 PA 2 Report

1. Command Parsing

Command parsing is done first before executing commands. There is no specific reason for this but I wanted get the grade points for parsing even if program somehow fails. It first opens commands.txt and parse.txt and then reads file line by line and for each line reads word by word with string streams. Parse data is also stored in CliLine struct and vector. Algorithm works like this for each line:

1. If first in line > it is command name
2. Else if has "-" character it is option
3. Else if has ">" or "<" it is redirection.
4. Else if had redirection symbol in previous word it is redirection file name
5. Else if has "&" char it is background job
6. Else it is input text

After parsing a function writes result to parse.txt

2. Executing Commands

For each parsed command line:

First, we check if the command is wait. If it is wait, we wait by first waiting number of Background Jobs times for processes to finish using a vector of pid's. After I also did it inside a while loop because of possible background job synchronizing problems which I shouldn't have any because all pushbacks are inside parent but it good to be safe. Than we wait for all threads in the threads queue to finish. Then number of Background Jobs is set to 0. After we skip current line.

If not, we check for redirection Symbol. If command doesn't write to console (symbol is >) we just fork. In child we open output file and duplicate to std out file descriptor. Than run the command with execvp. In parent we check if line is background job. If background job, we skip to next line. Else we wait for it to finish with waitpid.

If line writes to console, we do piping. First, we dynamically allocate 2 int's space for pipe variable. Then we do pipe and create fork. In child unused end of the pipe is closed and other end of pipe is opened with file stream and duplicated to std out. Than if has input redirection we also open input file and duplicate it std in file descriptor. Than we run command as usual. Main process closes the unused end of the pipe creates a thread for pipe read. And then we check if line is background job. If background job, we skip to next line. Else we wait for it to finish with waitpid and join the thread so finished process output is also properly printed. For the background jobs I put thread in a vector so I can wait all to finish when wait command is called or program ends.

When all is we wait for all the finish in a similar way as when wait statement is called.

(I didn't need to escape '-'character because of the way I put the string. So, I didn't check for any other possible characters.)