PROGRAMMING ASSIGNMENT (PA) 2  
SYNCHRONIZING THE CLI SIMULATOR REPORT

CS307 OPERATING SYSTEMS

SPRING 2022-2023

DEADLINE: 26 APRIL 2023

REPORT BY

ZEYNEP KURTULUS

29045

C++ Program Implementation

Before giving detailed information about implementation of each task asked in this programming assignment, I want to clarify that a struct called **CodeLines** is being used so the parsed data information will be kept together and they can be accessed from a single variable of **CodeLines**. Also, I have used 3 different vectors naming **backgroundvec**, **threadvec** and **codevec** to keep track of the background jobs, threads and **CodeLines**.

1. Command Parsing

In the context of this programming assignment, we have different sets of information that we want to access such as commands, inputs, options, redirecting operator and background operator. We need to extract each of this information separately inside the code and to be able to do that we need to perform a parsing operation. The parsing operation is being conducted inside the function called parser. This is a void function that takes 5 inputs; **CodeLines** struct (passed as a reference parameter so that changes will be permanent in main), line string which indicates each line received from the commands.txt file, **wordholder** string which will contain each word using stringstream, two Boolean variables naming **getCommand** and **isFilenameNext**, **getCommand** is being used so that we can extract the command from the line, initially **getCommand** is true it will be false once we captured the very first word of the line, since we know that the command is the first word in each line we are able to assign it to **Code.command** inside the struct, **isFilenameNext** is used to capture the *file\_name* before the redirection symbol, initially **isFilenameNext** is false, inside the parser function once we detect “<” or “>” symbol, we first store it inside **Code.rdsymbol** then we make **isFilenameNext** true because we know that the *file\_name* comes right after the redirection symbol. Once **isFilenameNext** turns true we are certain that next word in line corresponds to *file\_name* so, we store it inside **Code.newfilename**. Moving on, once we detect “-“ symbol we store it inside **Code.option** because we know that the option inside *commands.txt* file starts with a dash character. As indicated in the assignment description, we know that an optional & operator at the end enforces shell to run this command at the background so & operator indicates the presence of a background process, accordingly once we detect & character we make **Code.isbackjob** true indicating that we have captured a background job. At this point we have checked all possibility other than the word we captured being the input itself such as danger hw2.c, so in the last else part (line 69) we simply assign **wordholder** to be **Code.input.**In the main function after we open *parse.txt*, we use getline to get the contents of the *commands.txt* line by line, assign them to the line variable and call the parser function with it inside a while loop then we call the **promt()** function and add the **Code** variable to the corresponding vector, below you may see the details regarding **promt()** function.

1. Writing Results to *parse.txt*

After the parser function returns we call a function called promt. It takes two parameters types **CodeLines** and ofstream. It writes the contents inside the **CodeLines** in the order specified in the assignment document to the file called parse.txt. The if – else statements inside the function basically checks if **Code.isbackjob** is true or false. If it is true “y” is written to the file indicating it is a background job if it is false “n” is written to the file indicating it is not a background job.

1. Command Execution

After the parsing process and writing the contents into the *parsed.txt* the execution of the rest of the program starts. The code begins by looping through a vector of CodeLines called **codevec**, which contains a sequence of commands to execute. In each iteration of the loop, the current CodeLines object is stored in a variable called **codeholder**. If the current command is a "wait" command, the shell waits for all background processes and threads to terminate before continuing with the next command. It does this by looping through the **backgroundvec** vector of background process (note that we keep background processes inside the **backgroundvec**) IDs and waiting for each one to terminate with **waitpid().**It then loops through the **threadvec** vector of thread IDs and waits for each one to join with **pthread\_join().** Finally, it loops through all remaining child processes with **waitpid()** and waits for them to terminate with the **WNOHANG** flag. When the WNOHANG flag is used, **waitpid()** checks for the completion of a child process and returns immediately, even if the child process has not yet exited. If there are no child processes that have exited, **waitpid()** returns 0. At the end of the while loop we are sure that are background processes are finished (since we made the shell wait for them) so we set **the backgroundProcessNum** to 0. If the command is not "wait", we check if the redirection symbol is ">", indicating that the output should be redirected to a file. If the command contains an output redirectioning part shell process forks a new process that manipulates standard stream file handlers using **fork()**.Moving on, we check if we are inside the child process using pid, id so we create a character pointer array **args**, we will simply use this array to run command with **execvp()**call that is why inside the array we store the input and option variables. The child process then opens the file with the given name in write mode and redirects its stdout to the file descriptor of the open file using **dup2()**. It then assigns the last element of the array as null so that we can use **execvp()**to execute the command. If the redirection symbol is not ">", indicating that the command does not have output redirecting part then, shell process must create a pipe (channel) for this command that will enable the communication between the new command process and the shell process before forking the new process, so we first create a pipe using the **pipe()** system call, then fork a new child process using **fork().** The child process closes the read end of the pipe, opens a file to perform write operations, and redirects its stdout to the file descriptor of the opened file using **dup2().** It then creates an array of arguments called **args2** for the shell command and uses **execvp()** to execute the command. If we are not in the child process (i.e the pid is different than 0 we are in the parent process), we closes the write end of the pipe and create a new thread using pthread\_create() to listen for input from the pipe. Here we use a function called **threadListener.** It is responsible for listening to the output of the child process so that we will not experience any concurrency problem and lines printed by a command won’t be interpreted. Here is a breakdown of what the **threadListener** function does; first we simply acquire the mutex object that is shared among threads of the shell process with the means of concurrency control, The MUTEX variable is a mutex used for synchronization between threads. The program uses the mutex to control access to shared resources, such as the console output. The mutex is locked at the start of the **threadListener** function to prevent multiple threads from writing to the console simultaneously. It is released at the end of the function, allowing other threads to acquire the lock and write to the console. Then we initialize a buflen variable which will store the output of the command, then.We print the thread's ID to the console using **pthread\_self()** and opens a **FILE** object using the file descriptor, we read the output from the command through the **FILE** object using the **fread()** function. The while loop continues to read data from the file descriptor as long as **nread** is greater than 0, which means that data was successfully read into the **buffer**. Inside the loop, the null-termination of **buffer** is ensured by setting the character after the last read character to '\0', which allows the contents of **buffer** to be printed to the console using **printf**. Finally, **fflush(stdout)** is called to ensure that any buffered data is immediately printed to the console. Lastly, we close the file stream and release the mutex so that other threads can have it. If the command is not a background job, it waits for the child process to terminate using **waitpid()** and then joins the thread using **pthread\_join().**Finally, the code waits for all the background jobs to finish executing before continuing with the execution of the shell. The variable **backjobs** is the number of background jobs that are currently running.

The first while loop waits for **backjobs** number of child processes to finish using **waitpid().**The loop waits for child processes using **wait()** until there are no more child processes left to wait for. After all child processes have finished executing, the for loop joins all the thread IDs in **threadvec. pthread\_join()** blocks the calling thread until the specified thread terminates.