Programming Assignment (PA) – 2

(Synchronizing the CLI Simulator)

CS307 - Operating Systems

*22 April 2023*

Solution Report

Ataollah Hosseinzadeh Fard

ID: 28610

**Implementation Explanation**

It is needed to implement a program that will read “commands.txt” and will execute each line as a terminal. Each line will have following format:

*command* [*input*] [*option*] [> | < *file\_name*] [&]

Other than ‘command’ all others are optional, which means they may or may not be appear, the only constraints are that if there is an input, it will be a single word string, which means it cannot contain any blank spaces. And other constraint is that if there is an option it will be a single letter flag for example ‘-a’, ‘-l’, etc.

My solution read “commands.txt” and while reading each line, it will directly parse that line and will add corresponding data to “parse.txt” and then will start executing command. If command is “wait” it will implement a “wait” that will wait for all other processes and in case if any thread has not finished will join those threads as well. But if command is not “wait”, then it will first check type of direction, if it is not ‘>’ it will create a pipeline using a unique file descriptor, I will explain this uniqueness of file descriptors at the end. then it will create a new child process called command using fork(). inside child process will do the needed redirections for each redirection type. At last, it will execute the command with variables obtained from parsing. Inside parent process, it should wait for child to finish if that command is a background job otherwise it will not wait. At last console\_output() functions are assigned as a task to a unique thread, I will explain uniqueness of threads at the end. Before terminating program, a mechanism same as implementation of “wait” command is applied to make sure every process is finished and safe to terminate program.

In case of concurrency, I used 3 mutexes, one for editing the global variable of “thread\_count”, one for initializing buffer inside each thread, and one for printing the data of buffer to console. However, these 3 mutexes are only half side of solving concurrency issues. since if redirection is ‘<’ or ‘-‘ it should push output of execvp() to buffer of corresponding thread, and if I had used same file descriptor to read and write to an Anon file, it was going to accessed multiple times and I was going to get each an empty buffer or a mixed up buffer, which is not what I want to have, so as solution I created a 2D integer array which has same number of rows as size of threads array, but each row has 2 columns, 0th columns is read end and 1st column is write end. This will bring uniqueness to file descriptor because each thread will have corresponding file descriptor to read and write from. However, a small edge case is still there to solve, if number of active threads reach to maximum size of threads array, it will cause a crash, so in order to solve this issue, added a fail safe part after assigning a task to an empty thread, if the maximum threads have reached program will wait for all other process to finish, then it will wait for all other thread to finish and join and also it will reset all file descriptor to NULL, so this means that if program has maximum number of 10 threads and there are more than 10 lines in “commands.txt” that will require a new thread, as soon as threads array is full program will pause for a small time for all ongoing tasks to be done. This mechanism is also added to the case when command is “wait”, this means that fail safe will work in 2 cases, either when command is “wait” or when threads array is full. I think thanks to this this uniqueness of file descriptors, there will be no concurrency problems occurred.

Following are test run of given sample run (**changed hw2.c to cli.c in input1.txt**):

Text

Description automatically generated

**Code Implementation**

All needed libraries are included, needed mutexes are initialized as global variable also a helper global integer variable is initialized. Also, 2 macros are defined to make program more portable and easier to modify. The macro “thr\_size” will be maximum number of threads to be used at same time, and the macro “buff\_size” is the maximum buffer size of printing output of each thread.

Text

Description automatically generated

console\_output() is the function that will be assigned to each thread, it will get a unique read-end of file descriptor that belongs to same thread. It first creates a buffer with size of “buff\_size” macro, then locks “buff\_lock” mutex so that we can make sure buff is filled with null and no other thread is accessing and editing buff at same time. then I unlock “buff\_lock” and create a fd which is copy of read-end of file descriptor given as argument, and until there is anything to read from read-end, tries to red from fd. If there is anything in to read, it copies data to buff. At last, to print the content of butt locks the “cout\_lock” mutex and prints the buff with its thread id and then unlocks the mutex. “cout\_lock” is used to make sure that no other thread is printing buff to console.

Text

Description automatically generated

In main function first things to do is to create a threads array with size of “thr\_size” macro and creating a 2D integer array with row count of “thr\_size” macro and column count of 2, this will be used as file descriptor in each thread, which means each thread has its unique file descriptor.

Text

Description automatically generated

Then “commands.txt” is opened, if it does not exist then program stops, and using a while loop, in each iteration, each line of txt is written to line variable.

Text

Description automatically generated

Inside loop, first all the variables that correspond to command, input, option, redirection type, redirection file, and background job are created. Then parsing starts, it iterates over line char by char and determines which word should be places in which variable. There were some special cases that needed to be handles, double quote (“) and dash (-) characters when they were inside input variable, it was causing crash while executing execvp, therefore while parsing program removes any double quote and dash character from input variable.

Text

Description automatically generated

After parsing variables, the string array that is going to be passed to execvp should be created according to the number of existing arguments, which means number of arguments can be vary from 1 (only command is available) to 3 (all command, input, and option are available).

Text

Description automatically generated

Since parsing is done, now program can add the parsed data to “parse.txt”.

A screenshot of a computer

Description automatically generated with medium confidence

Now main objective begins, we start to execute command, but first program must check if command is “wait” or not, and if it is “wait”, then it should implement the “wait” command. First waits for all other processes to finish. Then, inside a mutex lock block of “count\_lock” it waits for all other threads to finish and join, then sets thread count to 0 and resets all file descriptors to NULL.

Text

Description automatically generated

But if command is not “wait” then it will first check the type of redirection and if type is not ‘>’ then will create unique pipeline of corresponding thread.

Text

Description automatically generated

Then it creates a new process called command, and if program is inside child process program first redirects the necessary input-ends and output-ends and then passes arguments to execvp().

Text

Description automatically generated

If program is inside parent process, it first checks whether this task is a background job or not. If it is not a background job, then parent process waits for child process to finish, otherwise it will not wait. Then if redirection type is not ‘>’, it will create thread and assign console\_output() to that thread and passes its own file descriptor’s read-end as reference. Then to be safe the thread count is incremented inside mutex block of “count\_lock”, also the fail safe of file descriptor uniqueness is implemented inside this block.

Text

Description automatically generated

Before terminating the program, it should wait for all processes to finish, also it needs to wait for all threads to finish and join. And to be safe, closes the “commands.txt” streamer. At last, program terminates.

Text

Description automatically generated