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:

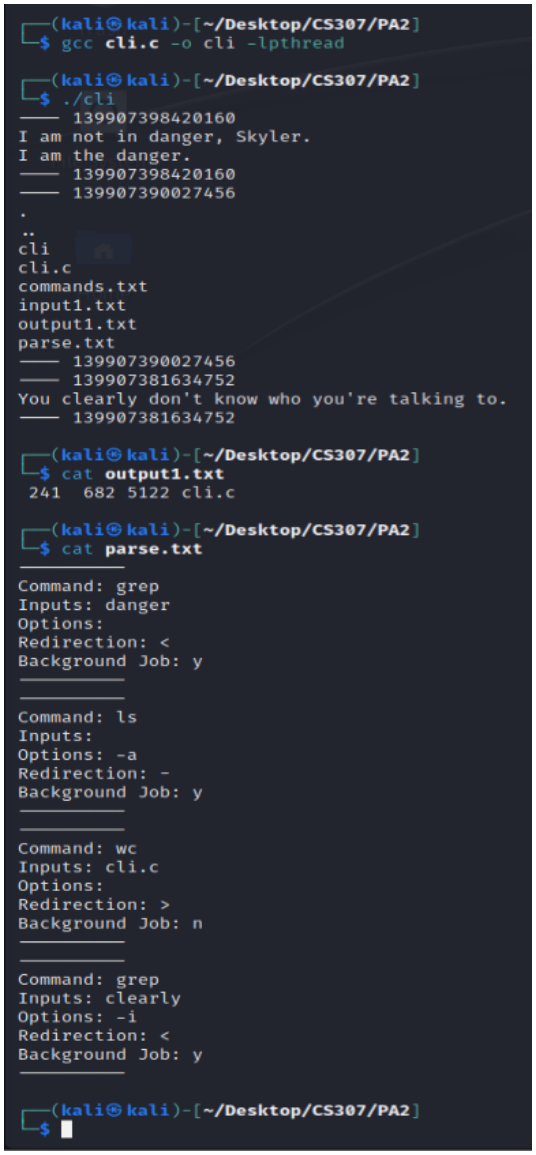
command [*input*] [*option*] [> | < *file_name*] [&]

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, 0^{th} columns is read end and 1^{st} 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**):



Code Implementation

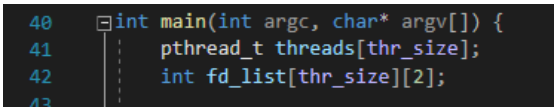
All needed libraries are included, needed mutexes are initialized as global variable also a helper global integer variable is initialized. Also, 2 macros are defined to make program more portable and easier to modify. The macro “thr_size” will be maximum number of threads to be used at same time, and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

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

#define thr_size 100
#define buff_size 5000

pthread_mutex_t count_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t buff_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t cout_lock = PTHREAD_MUTEX_INITIALIZER;

int thread_count = 0;
```

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.

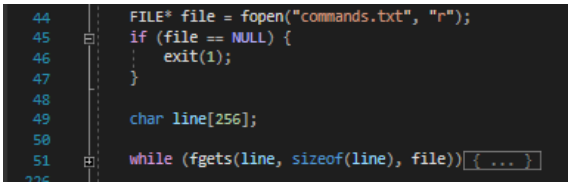
```
17
18  void* console_output(void* args) {
19      char buff[buff_size];
20
21      pthread_mutex_lock(&buff_lock);
22      for (int i = 0; i < buff_size; i++) {
23          buff[i] = '\0';
24      }
25      pthread_mutex_unlock(&buff_lock);
26
27      int fd = *(int*)args;
28      while (read(fd, buff, buff_size) == 0);
29
30      pthread_mutex_lock(&buff_lock);
31      printf("---- %ld\n", pthread_self());
32      printf("%s", buff);
33      printf("---- %ld\n", pthread_self());
34      fflush(stdout);
35      pthread_mutex_unlock(&buff_lock);
36
37      return NULL;
38  }
```

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.



```
40  int main(int argc, char* argv[]) {
41      pthread_t threads[thr_size];
42      int fd_list[thr_size][2];
```

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.



```
44      FILE* file = fopen("commands.txt", "r");
45      if (file == NULL) {
46          exit(1);
47      }
48
49      char line[256];
50
51      while (fgets(line, sizeof(line), file)) { ... }
```

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.

```c
        char cmd[10] = "";
        char in[50] = "";
        char opt[3] = "";
        char rd[2] = "";
        char rd_file[256] = "";
        char bg_j[2] = "n";

        // beginning of parsing
        char* itr = line;
        int var_idx = 0;
        int need2switch = 0;
        int w_count = 1;

        while (*itr != '\0' && *itr != '\n' && *itr != '&') {
            if (need2switch == 1) {
                if (var_idx == 0 || var_idx == 1 || var_idx == 2) {
                    if (*itr == '-') {
                        var_idx = 2;
                    }
                    else if (*itr == '>' || *itr == '<') {
                        var_idx = 3;
                    }
                    else {
                        var_idx = 1;
                    }
                }
                else if (var_idx == 3) {
                    var_idx++;
                }
                need2switch = 0;
            }
            if (!strcmp(cmd, "wait")) {
                break;
            }
            if (*itr != ' ') {
                switch (var_idx) {
                case 0:
                    strncat(cmd, itr, 1);
                    break;
                case 1:
                    /* to handle special characters of(") and (-), current character
                       is one of those, does not add to input string.*/
                    if (*itr != '"' && *itr != '-') {
                        strncat(in, itr, 1);
                    }
                    break;
                case 2:
                    strncat(opt, itr, 1);
                    break;
                case 3:
                    strncat(rd, itr, 1);
                    break;
                case 4:
                    strncat(rd_file, itr, 1);
                    break;
                }
            }
            else {
                need2switch = 1;
                w_count++;
            }
            itr++;
        }
        if (*itr == '&') {
            bg_j[0] = 'y';
        }
        if (strlen(rd) == 0) {
            rd[0] = '-';
        }
```

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).

```c
        int idx = 0;
        char* args[4];
        args[idx++] = cmd;
        if (strlen(in) > 0) {
            args[idx++] = in;
        }
        if (strlen(opt) > 0) {
            args[idx++] = opt;
        }
        args[idx] = NULL;
        // end of parsing
```

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

```
134         FILE* parse_file = fopen("parse.txt", "a");
135         if (parse_file == NULL) {
136             exit(1);
137         }
138
139         fprintf(parse_file, "----------\n");
140         fprintf(parse_file, "Command: %s\n", cmd);
141         fprintf(parse_file, "Inputs: %s\n", in);
142         fprintf(parse_file, "Options: %s\n", opt);
143         fprintf(parse_file, "Redirection: %s\n", rd);
144         fprintf(parse_file, "Background Job: %s\n", bg_j);
145         fprintf(parse_file, "----------\n");
146
147         fclose(parse_file);
```

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.

```
149     if (!strcmp(cmd, "wait")) {
150         fflush(stdout);
151
152         int waiting = wait(NULL);
153         while (waiting > 0) {
154             waiting = wait(NULL);
155         }
156
157         /* If there is any background jobs, to make sure they also
158            are done, program have to wait for all threads to join. */
159         pthread_mutex_lock(&count_lock);
160         for (int i = 0; i < thread_count; i++) {
161             pthread_join(threads[i], NULL);
162         }
163         thread_count = 0;
164         memset(fd_list, 0, sizeof(fd_list));
165         pthread_mutex_unlock(&count_lock);
166     }
167     else { ... }
224 }
```

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.

```
149         if (!strcmp(cmd, "wait")) { ... }
167         else {
168             if (rd[0] != '>') {
169                 pipe(fd_list[thread_count]);
170             }
171
```

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().

```
172         int command = fork();
173         if (command < 0) {
174             exit(1);
175         }
176         else if (command == 0) {
177             if (rd[0] == '<') {
178                 close(STDIN_FILENO);
179
180                 int fileNo = open(rd_file, O_RDONLY);
181
182                 dup2(fd_list[thread_count][1], STDOUT_FILENO);
183             }
184             else if (rd[0] == '>') {
185                 close(STDOUT_FILENO);
186
187                 open(rd_file, O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);
188             }
189             else {
190                 dup2(fd_list[thread_count][1], STDOUT_FILENO);
191
192                 fflush(stdout);
193             }
194
195             execvp(args[0], args);
196         }
197         else { ... }
223     }
```

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.

```
197         else {
198             if (bg_j == "n") {
199                 waitpid(command, NULL, 0);
200             }
201
202             if (rd[0] != '>') {
203                 pthread_create(&threads[thread_count], NULL, console_output, &fd_list[thread_count][0]);
204
205
206                 pthread_mutex_lock(&count_lock);
207                 thread_count++;
208
209                 /* If the thread list is fully filled, then program wait for all threads to join and
210                    will reset fd_list, if this is not done, next iteration thread_count will be out
211                    of range of threads list and this will cause a crash in program.*/
212                 if (thread_count == thr_size) {
213                     for (int i = 0; i < thread_count; i++) {
214                         pthread_join(threads[i], NULL);
215                     }
216                     thread_count = 0;
217                     memset(fd_list, 0, sizeof(fd_list));
218                 }
219
220                 pthread_mutex_unlock(&count_lock);
221             }
222         }
```

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.

```
226     int waiting = wait(NULL);
227     while (waiting > 0) {
228         waiting = wait(NULL);
229     }
230
231     /* If there is any background jobs, to make sure they also
232        are done, program have to wait for all threads to join. */
233     for (int i = 0; i < thread_count; i++) {
234         pthread_join(threads[i], NULL);
235     }
236
237     fclose(file);
238
239     return 0;
240 }
241
```