CSE 3033 Project #2 Report

Implementing a Basic Shell with C

Student 1: Zehra Zeynep Pehlivan - 150116504
Student 2: Hümeyra Ceyda Polat – 150117065

Our objective in this project is to implement a basic shell with C. This shell has the functionalities:

1. Execute every possible executables stored in variable PATH for both foreground and background executions.

2. Execute some built-in commands listed below:

• history – see the list of and execute the last 10 commands.
• ^Z signal – stop currently running foreground process; if no process running on foreground, ignore the signal.
• path – path is a command not only to show the current pathname list, but also to append or remove several pathnames.

- path (without arguments) - displays the pathnames currently set.
- path + /foo/bar - appends the pathname to the "path" variable.
- path - /foo/bar - removes the pathname from the "path" variable.

• fg %num – Move the background process with process id num to the foreground.
• exit: terminate shell program.

3. Execute and handle commands with redirect operations that are listed below
• myprog [args] > file.out – write output of the program with given arguments to the given file with
truncation mode.
• myprog [args] >> file.out – write output of the program with given arguments to the given file with append mode.
• myprog [args] < file.in – use given file as the standard input for program with given arguments.
• myprog [args] 2> file.out – set given file the standard error output for program with given arguments.
• myprog [args] 2> file.out – set in_file as the standard input of program with given arguments, write output of the program to the file.out.

PART A:

In this part of project we used fork() in order to create a child process to execute given arguments. If user gives the process as a foreground process parent process waits until the child process terminates.

```
void execute(char* args[] ,int background, bool doFork, pid_t child_pid){

    if(doFork)
        child_pid = fork();

    // Fork error
    if(child_pid == -1) {
        fprintf(stderr, format: "Failed to fork\n");
        return;
    }

    else if(child_pid > 0){  // Parent process
        if(background == 0) {  //Foreground process
            //Update foreground process variables
            ANY_FOREGROUND_PROCESS = true;
            CURRENT_FOREGROUND_PROCESS = child_pid;
            wait(&child_pid);//wait for child process to end
            //No foreground process left
            ANY_FOREGROUND_PROCESS = false;
        }
        else {  //Background process
            printf( format: "Process is a background process.\n");
            addBackgroundProcess(child_pid, args);
            printQueue(); //Print current background processes
        }
        background = 0;

    else if (child_pid == 0){  //CHILD PROCESS
        //If the process created successfully
        if(background == 0)
            printf( format: "Process is a foreground process.");

        execute_path(args);
```

If child process created successfully, we called our execute_path() function with given arguments. In this function first we get value of PATH variable, then split the result from “:”. We appended “/“ and first argument to each path variable, then checked if “concatenated string” named file accessible in executable mode. If an accessible(executable) file found in this block, execute given parameter with “execv” system call function.

```c
//Separate each path from ":"
char * path_token = strtok(path_array, delim: ":");
char buffer[MAX_LINE]; // array for add / and args[0] to the path token

// loop through the string to extract all other tokens
 i = 0;
while( path_token != NULL ) {
    strcat(buffer, path_token);
    strcat(buffer, src: "/");
    strcat(buffer, src: args[0]);
    char * temp = " ";
    printf( format: "%c", temp);

    if(execv(buffer, args) == -1){
        path_exist = 0 ;
    }

    clearBuffer(buffer);

    if(path_exist == 0 && numberofPath == i-1)
        fprintf(stderr, format: "\nInvalid command! Please try again.");

    i++;
    path_token = strtok( s: NULL, delim: ":");
```

If user gives the process as a background process we added process to our background process queue. Then program will take another input immediately without complete the background process.

```c
typedef struct background_process_queue{
    pid_t p_id;
    char command[MAX_LINE];
    int id;
    struct background_process_queue *next;
} background_process;

background_process *head = NULL;
background_process *curr = NULL;
background_process *previous = NULL;
```

```c
}
else {  //Background process
    printf( format: "Process is a background process.\n");
    addBackgroundProcess(child_pid, args);
    printQueue(); //Print current background processes
}
```

```c
void addBackgroundProcess(int data, char* args[]){


    if(head == NULL) {
        head = (background_process *) malloc(sizeof(background_process));
        head->p_id = data;
        generateCommandLine(args, head->command);
        head->id = backgroundProcessNumber;
        head->next = (background_process *) malloc(sizeof(background_process));
        previous = curr;
        curr= head->next;
    }
    else{
        previous = curr;
        curr->p_id = data;
        generateCommandLine(args, curr->command);
        curr->id=backgroundProcessNumber;
        curr->next = (background_process *) malloc(sizeof(background_process));
        curr=curr->next;

    }
    backgroundProcessNumber++;

}
```

Example output:

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o
myshell:
ls -l
ls -l
total 64
drwxrwxr-x 3 ceyda ceyda  4096 Ara 10 09:57 cmake-build-debug
-rw-rw-r-- 1 ceyda ceyda   117 Ara  7 13:34 CMakeLists.txt
-rw-rw-r-- 1 ceyda ceyda   588 Ara 10 02:19 Eksikler
-rw-rw-r-- 1 ceyda ceyda 25664 Ara 10 09:49 main.c
-rwxr-xr-x 1 ceyda ceyda 23120 Ara 10 09:58 main.o
myshell:
pwd
pwd
/home/ceyda/CLionProjects/untitled1
myshell:
pwd &
pwd &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  28773              pwd &
ls
myshell:
cmake-build-debug  CMakeLists.txt  Eksikler  main.c  main.o
```

PART B:

- ## History:

  For this part, we used setupHist_Fg function to generate a new args double array and background variable to forked and execute to command. We have double history array which can holds 10 previous commands.

```
323
324 void runningHistory(char * histBuffer , int length) {
325
326     int background = 0; /* equals 1 if a command is followed by '&' */
327     char *histArgs[MAX_LINE/2 + 1]; /*command line arguments */
328
329     numberofHistory++;
330     if(controlOfHistoryNum )  //when hist num will be greater than 10, hist array will be shifted one
331         shiftHistArray();
332
333     copyBufferToHist(histBuffer,  length: findCommandLength(histBuffer)+1);
334     histBufferLengths[numberofHistory-1]= length;
335     setupHist_Fg(histBuffer, histArgs, &background, length, controlVariableForFG);
336     execute(histArgs ,background , doFork: 1, child_pid: 0);
337 }
```

```
myshell: ls -l
toplam 88
-rw-rw-r-- 1 zeynoi zeynoi 22723 Ara 10 21:02 CMakeCache.txt
drwxrwxr-x 5 zeynoi zeynoi  4096 Ara 10 22:39 CMakeFiles
-rw-rw-r-- 1 zeynoi zeynoi  1541 Ara 10 21:02 cmake_install.cmake
-rw-rw-r-- 1 zeynoi zeynoi  4977 Ara 10 21:02 Makefile
-rwxrwxr-x 1 zeynoi zeynoi 38488 Ara 10 22:36 untitled1
-rw-rw-r-- 1 zeynoi zeynoi  5343 Ara 10 21:02 untitled1.cbp

myshell: history -i 1
toplam 88
-rw-rw-r-- 1 zeynoi zeynoi 22723 Ara 10 21:02 CMakeCache.txt
drwxrwxr-x 5 zeynoi zeynoi  4096 Ara 10 22:39 CMakeFiles
-rw-rw-r-- 1 zeynoi zeynoi  1541 Ara 10 21:02 cmake_install.cmake
-rw-rw-r-- 1 zeynoi zeynoi  4977 Ara 10 21:02 Makefile
-rwxrwxr-x 1 zeynoi zeynoi 38488 Ara 10 22:36 untitled1
-rw-rw-r-- 1 zeynoi zeynoi  5343 Ara 10 21:02 untitled1.cbp

myshell: history
   1  ls -l
   2  ls -l

myshell: ls &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  11560      ls &
```

```
history -i 1
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  11560      ls &
[1]  11561      ls &
```

When number of history reached to 10, history array will be shifted for any command unless command is a “history”.

```
void setupHist_Fg(char *histBuffer, char *histArgs[], int *background, int length,
void shiftHistArray();
void runningHistory(char * histBuffer, int length);
```

- ^Z

For this part, first we initialize the signal SIGTSTP, then assign to it its handler. This signal should stop process that is running on foreground. If there is no foreground process, this signal should be ignored.

```
/* Initialize signal */
struct sigaction act;
act.sa_handler = controlZ;
act.sa_flags = SA_RESTART;

// Set up signal handler for ^Z signal
if ((sigemptyset(&act.sa_mask) == -1) || (sigaction(SIGTSTP, &act, oact: NULL) == -1)) {
    fprintf(stderr, format: "Failed to set SIGTSTP handler\n");
    return 1;
}
```

```
// Handler for ^Z signal, if this signal received, that means user wants to stop foreground process and its descendants
static void controlZ() {
    int status;
    // If there is a foreground process
    if(ANY_FOREGROUND_PROCESS) {
        // Check if this process still running
        kill(CURRENT_FOREGROUND_PROCESS, sig: 0);
        // If not running, errno will set to ESRCH
        if(errno == ESRCH) {
            // Inform user
            fprintf(stderr, format: "\nprocess %d not found\n", CURRENT_FOREGROUND_PROCESS);
            // Set any foreground process to false
            ANY_FOREGROUND_PROCESS = false;
            printf( format: "myshell: ");
            fflush(stdout);
        }
            // If foreground process is still running
        else {
            // Send a kill signal to it
            kill(CURRENT_FOREGROUND_PROCESS, SIGKILL);
            // Then wait for its group to terminate with option WNOHANG
            waitpid(-CURRENT_FOREGROUND_PROCESS, &status, WNOHANG);
            printf( format: "\n");
            // If code reaches here, there is no foreground process remaining
            ANY_FOREGROUND_PROCESS = false;
        }
    }
        // If there is no background process, ignore the signal
    else {
        printf( format: "\nmyshell: ");
        fflush(stdout);
```

We have some variables that changes according to the current working process on foreground and we use these variables as flags of the signal handler. If there is a process running on foreground, check if it’s still running on foreground with kill(pid, 0). If errno is set to ESRCH, that means the process is not running anymore. If process is running, call kill

function with the signal SIGTERM, then wait for its descendants' terminations. If there is no process running on foreground, ignore signal, prompt myshell message.

Example output:

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o
myshell:
ls -l
ls -l
total 72
drwxrwxr-x 3 ceyda ceyda  4096 Ara 10 10:33 cmake-build-debug
-rw-rw-r-- 1 ceyda ceyda   117 Ara  7 13:34 CMakeLists.txt
-rw-rw-r-- 1 ceyda ceyda   588 Ara 10 02:19 Eksikler
-rw-rw-r-- 1 ceyda ceyda 29797 Ara 10 10:33 main.c
-rwxr-xr-x 1 ceyda ceyda 27640 Ara 10 10:44 main.o
myshell:
^Z
myshell:
```

- Path:

In our path function first we generated pathname queue with data structure 'linked list' using value of PATH variable after split the result from ":".

```c
typedef struct pathname_list{
    char pathname[MAX_LINE];
    struct pathnamelist *next;
}pathname_list;


pathname_list *pathname_head = NULL;
pathname_list *current_path = NULL;

void path(char* args[]){
    //Get the ENV PATH'sargs
    char *path_array = getenv( name: "PATH");
    //Separate each path from ":"
    char * path_token = strtok(path_array,  delim: ":");

    if(pathgeneratedControl == 0){
        // add all paths to the pathname list
        while (path_token != NULL) {
            addPath(path_token);
            path_token = strtok( s: NULL,  delim: ":");
        }
        pathgeneratedControl = 1;
    }

    int argsLen = 0;

    while(args[argsLen])
        argsLen++;

    if(argsLen == 1) {   //if user give just "path" as an argument print it separately
        //if user give just "path" as an argument print it separately
    }
    else {
        if (!strcmp(args[1], "+")) {  //if user wants to add path
            addPath( path_name: args[2]);
        }else if (!strcmp(args[1], "-")) {  //if user wants to remove path
            removePath( path_name: args[2]);
        }
    }
```

- If user gives input as a “path” we printed our queue to the screen.

```
//print list
pathname_list *temp = pathname_head;
int i = 0;
while(i < numberofPath ){
    printf( format: "[%d]  \t%s\n", i, temp->pathname);
    temp=temp->next;
    i++;
}
```

Example output:

```
path
[0]     /home/ceyda/.local/bin
[1]     /usr/local/sbin
[2]     /usr/local/bin
[3]     /usr/sbin
[4]     /usr/bin
[5]     /sbin
[6]     /bin
[7]     /usr/games
[8]     /usr/local/games
[9]     /snap/bin
myshell:
```

- If user gives input as a “path + /directory_name” we added given directory to the queue by using our addPath() function. In this function we added given argument to the end of queue.

```
void path(char *args[]);
void addPath(char* path_name);
void removePath(char* path_name);
```

Example output:

```
path + /foo/bar
[0]     /home/ceyda/.local/bin
[1]     /usr/local/sbin
[2]     /usr/local/bin
[3]     /usr/sbin
[4]     /usr/bin
[5]     /sbin
[6]     /bin
[7]     /usr/games
[8]     /usr/local/games
[9]     /snap/bin
[10]    /foo/bar
myshell:
```

- If user gives input as a “path - /directory_name” we removed given directory from the queue by using our removePath() function. In this function we removed given argument from the reliable place in queue.

Example output:

```
path - /bin
[0]     /home/ceyda/.local/bin
[1]     /usr/local/sbin
[2]     /usr/local/bin
[3]     /usr/sbin
[4]     /usr/bin
[5]     /sbin
[6]     /usr/games
[7]     /usr/local/games
[8]     /snap/bin
[9]     /foo/bar
myshell:
```

- Fg % num:

We store all background processes in a queue. If user enters fg command, we dequeue this process from queue and execute.

If queue is empty, inform user that there is no background process.

```c
void fg(int id_num) { /*command line arguments */

    background_process* temp = head;

    char FgBuffer [MAX_LINE];

    int background = 0; /* equals 1 if a command is followed by '&' */
    char *FgArgs[MAX_LINE/2 + 1]; /*command line arguments */

    bool fgIdValid = false;
    int cnt = backgroundProcessNumber;
    while(cnt--)  //get command of process which has fg command pid
    {
        int id = id_num;
        if(temp->p_id == id){
            strcpy(FgBuffer, temp->command);
            fgIdValid = true;
            break;
        }
        temp = temp->next;
    }

    if(fgIdValid){

        int length = findCommandLength(FgBuffer) ;
        controlVariableForFG = 1;
        setupHist_Fg(FgBuffer, FgArgs, &background,  length: length-1, controlVariableForFG);
        controlVariableForFG = 0;
        removeProcess(id_num);
        execute(FgArgs , background, doFork: 1, child_pid: 0);
        printQueue();

    } else
        fprintf(stderr, format: "There is no such background process which has this id\n");

}
```

If queue is not empty, we travel in queue for find the process and if this id is valid, we generate the args double pointer char array and execute command.

```
myshell: ls &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  12492      ls -l &
[1]  12497      ls &
myshell: ls -l &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  12492      ls -l &
myshell: ls: '&''e erişilemedi: Böyle bir dosya ya da dizin yok
fg 12497

CURRENT BACKGROUND PROCESSES:
[0]  12492      ls -l &

myshell: CMakeCache.txt cmake_install.cmake  untitled1
CMakeFiles  Makefile         untitled1.cbp
fg 12492
No background processes found

myshell: toplam 88
-rw-rw-r-- 1 zeynoi zeynoi 22723 Ara 10 21:02 CMakeCache.txt
drwxrwxr-x 5 zeynoi zeynoi  4096 Ara 10 23:18 CMakeFiles
-rw-rw-r-- 1 zeynoi zeynoi  1541 Ara 10 21:02 cmake_install.cmake
-rw-rw-r-- 1 zeynoi zeynoi  4977 Ara 10 21:02 Makefile
-rwxrwxr-x 1 zeynoi zeynoi 38512 Ara 10 23:18 untitled1
-rw-rw-r-- 1 zeynoi zeynoi  5343 Ara 10 21:02 untitled1.cbp
```

- Exit:

If user called exit and there is no background processes running, shell should end its session. If there are some background processes running, shell should warn user.

```c
int exitt(char *args[] ){     // If user entered exit
    // Check background processes
    if(head != NULL) {
        // inform user
        fprintf(stderr, format: "\nThere are some processes that are running background\n");
        printQueue();
        fflush(stdout);

        fprintf(stderr, format: "Do you want to terminate all of them? [y/n] \n");
        char answer = getchar();
        if (answer == 'y') {

            head = NULL;
            previous = NULL;
            printf( format: "Session ends\nBye!\n");
            exit( status: 0);
        } else
            printf( format: "Program will not terminated. There is still background processes.\n");
    }
        // If no background processes running currently, exit program
    else {
        printf( format: "Session ends\nBye!\n");
        exit( status: 0);
    }
```

If background process queue is not empty, shell should warn user and print all current background processes, then wait for new input.

Example:

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o

myshell: ls &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  30250              ls &

myshell: ls: cannot access '&': No such file or directory
exit

There are some processes that are running background

CURRENT BACKGROUND PROCESSES:
[0]  30250              ls &
Do you want to terminate all of them? [y/n]
n
Program will not terminated. There is still background processes.

myshell:
```

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o

myshell: ls &
Process is a background process.

CURRENT BACKGROUND PROCESSES:
[0]  30397              ls &

myshell: ls: cannot access '&': No such file or directory
exit

There are some processes that are running background

CURRENT BACKGROUND PROCESSES:
[0]  30397              ls &
Do you want to terminate all of them? [y/n]
y
Session ends
Bye!
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$
```

PART C

For this part, first we checked if arguments contains any redirection operators or not. If contains, we created a child process and executed the arguments after setting redirections.

```
int TRUNC_OUT = 0;  // For truncation
int APPEND_OUT = 0;   // For append
int ERR_OUT = 0;     // For printing err
int IN_MODE = 0;    //For input
```

```
int ioRedirection(char *args[]){

    int i = 0;
    while(args[i]){
        if (!strcmp(args[i], ">")){
            TRUNC_OUT = 1;
            break;
        }
        else if (!strcmp(args[i], ">>")){
            APPEND_OUT = 1;
            break;
        }
        else if (!strcmp(args[i], "<")){
            IN_MODE = 1;
            break;
        }
        else if (!strcmp(args[i], "2>")) {
            ERR_OUT = 1;
            break;
        }
        i++;
    }
```

- ## myprog [args] > file.out:

With this redirection mode, we need to redirect output of the program to given file. If this file doesn’t exist, create the file. If file exists, truncate it.

```c
else if (TRUNC_OUT == 1) { // Write the standard output of program to the file
    pid_t pid;
    char *path = "/bin/"; //Start Path
    pid = fork();

    // Create file if needed
    if (pid == 0) {
        int openFile;
        openFile = open( file: args[arg_count - 1],  oflag: CREATE_FLAGS, CREATE_MODE);
        // Check if file opened
        if (openFile == -1) {
            fprintf(stderr,  format: "Failed to open file\n");
            return 1;
        }
        //Check if redirection is finished
        if (dup2(openFile, STDOUT_FILENO) == -1) {
            fprintf(stderr,  format: "Failed to redirect standard output\n");
            return 1;
        }
        //Check if the file is gonna close
        if (close(openFile) == -1) {
            fprintf(stderr,  format: "Failed to close the file\n");
            return 1;
        }
        char pathForExec[50];
        //create a path for the redirection
        //copy path
        strcpy(pathForExec, path);
        //add args for different commands
        strcat(pathForExec,  src: args[0]);
        args[arg_count - 1] = NULL;
```

```c
        args[arg_count - 2] = NULL; //The last two arrays will be NULL
        //Print the errors in stderr like requested
        if (execv(pathForExec, args) == -1) {
            fprintf(stderr,  format: "Child process couldn't run \n");
        }
    } else if (pid > 0) {
        wait(&pid);
    }
    TRUNC_OUT = 0;
}
```

Example:

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o

myshell: ls -l > file.out

myshell:
```

- myprog [args] >> file.out

With this redirection mode, we need to redirect output of the program to given file. If this file doesn't exist, create the file. If file exists, append to it.

```
else if (APPEND_OUT == 1) {
    char* path="/bin/"; //Start Path
    pid_t pid= fork();

    if(pid==0){
        int openFile;
        // create new folder
        openFile = open( file: args[arg_count-1], oflag: CREATE_APPENDFLAGS,CREATE_MODE);
        if(openFile == -1){
            fprintf(stderr,  format: "Failed to open file\n");
            return 1;
        }
        if(dup2(openFile,STDOUT_FILENO) == -1){
            fprintf(stderr,  format: "Failed to redirect standard output\n");
            return 1;
        }

        if(close(openFile) == -1){
            fprintf(stderr,  format: "Failed to close the file\n");
            return 1;
        }

        char pathForExec[50];

        strcpy(pathForExec, path);
        strcat(pathForExec,  src: args[0]);

        if (execv(pathForExec,args) == -1){
            fprintf(stderr,  format: "Child does not work \n");
        }

    }
    else if(pid>0){
        wait(&pid);
    }
    APPEND_OUT = 0;
}
```

Example:

```
ceyda@ceyda-X541UJ:~/CLionProjects/untitled1$ ./main.o

myshell: ls >> file.out

myshell:
```

- myprog [args] < file.in

With this redirection mode, we need to make standard input given file for the program. If this file doesn't exist, print error.

```
else if (IN_MODE == 1) {
    pid_t pid = fork();
    if (pid == 0) {
        int fd_in;
        // open in read only mode, if opening fails...
        if ((fd_in = open( file: args[arg_count - 1], O_RDONLY)) < 0) {
            fprintf(stderr,  format: "Couldn't open input file\n");
            exit( status: -1);
        }
        // duplicate standard input file
        dup2(fd_in, STDIN_FILENO);

    } else if (pid > 0) {
        wait(pid);
    }
    IN_MODE = 0;
}
```

```
myshell: myprog < file.in

myshell: cmake-build-debug  CMakeLists.txt  Eksikler  file.in  file.out main.c m
ain.o
```

- myprog [args] 2> file.out

With this redirection mode, we need to redirect error output of the program to given file. If this file doesn't exist, create the file.

```c
else if (ERR_OUT == 1) {
    pid_t pid = fork();
    if (pid == 0) {
        int fd_err;
        // open file with write permission, create if doesn't exists, truncate if exists...
        // with desired read, write, execute permissions (0644 in our program)
        if ((fd_err = open( file: args[arg_count - 1],  oflag: O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
            fprintf(stderr,  format: "Couldn't open output file\n");
            exit( status: -1);
        }
        // redirect standard error file to the given file
        dup2(fd_err, STDERR_FILENO);
        close(fd_err);                    // close file
    } else if (pid > 0) {
        wait(&pid);
    }
    ERR_OUT = 0;
}
```

- myprog [args] < file.in > file.out

With this redirection mode, we need to make standard input file.in for the program and file.out standard output of the program. If file.in doesn't exist, print error message. If file.out doesn't exist, create it. If it exists, truncate it.

Example:

```
myshell: myprog < file.in > file.out

myshell:
```

```c
if (IN_MODE == 1 && TRUNC_OUT == 1) {   // This condition is for opening output file of the command "x < y > z"
    pid_t pid = fork();
    if (pid == 0) {
        int inputFile;
        int outputFile;
        char *path = "/bin/";
        //input file open
        inputFile = open( file: args[arg_count - 3], O_RDWR, CREATE_MODE);
        //new file
        outputFile = open( file: args[arg_count - 1],  oflag: CREATE_FLAGS, CREATE_MODE);

        if (inputFile == -1) {
            perror( s: "Failed to open file");
            return 1;
        }

        if (dup2(inputFile, STDIN_FILENO) == -1) {
            perror( s: "Failed to redirect standart output");
            return 1;
        }

        if (close(inputFile) == -1) {
            perror( s: "Failed to close the file");
            return 1;
        }

        if (outputFile == -1) {
            perror( s: "Failed to open file");
            return 1;
        }
```

```c
        if (dup2(outputFile, STDOUT_FILENO) == -1) {
            perror( s: "Failed to redirect standart output");
            return 1;
        }

        if (close(outputFile) == -1) {
            perror( s: "Failed to close the file");
            return 1;
        }


        char kp[50];
        //creating path
        strcpy(kp, path);
        strcat(kp,  src: args[0]);
        args[arg_count - 4] = NULL;

        if (execv(kp, args) == -1) {
            //fprintf(stderr, "Child does not work \n");
        }
    } else if (pid > 0) {
        wait(&pid);
    }
    IN_MODE = 0;
    TRUNC_OUT = 0;
}
```