CSE 3033 Project #2 Report

Implementing a Basic Shell with C

Student 1: Mert Kelkit - 150115013

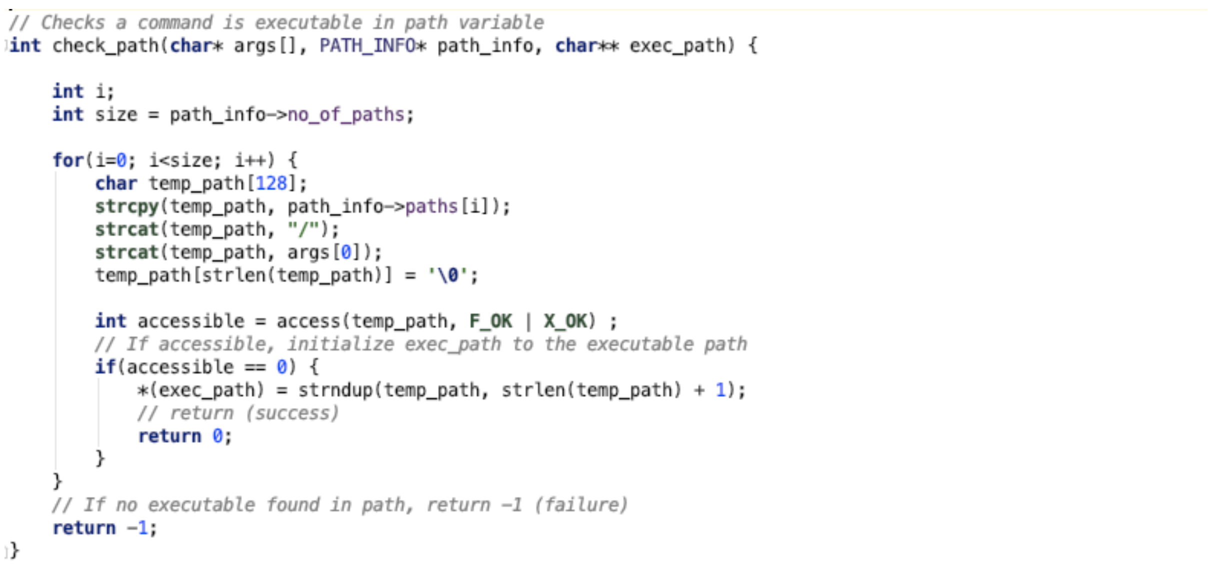
Student 2: Furkan Nakıp - 150115032

In this project, our objective is to implement a basic shell with C. Our 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:
    - alias/unalias: set alias/unalias for a command.
    - ^Z signal: stop currently running foreground process; if no process running on foreground, ignore the signal.
    - clr: clear terminal screen.
    - fg: move all background processes to foreground.
    - exit: terminate shell program.
3. Execute and handle commands with redirect operations that are listed below
    - prog args > file: write output of the program with given arguments to the given file with truncation mode.
    - prog args >> file: write output of the program with given arguments to the given file with append mode.
    - prog args < file: use given file as the standard input for program with given arguments.
    - prog args 2> file: set given file the standard error output for program with given arguments.
    - prog args < in_file > out_file: set in_file as the standard input of program with given arguments, write output of the program to the out_file.

Part 1:

For this part of project, first we get value of PATH variable, then split the result from ":" because it's stored like this: "/usr/bin:/bin:/usr/local/bin".

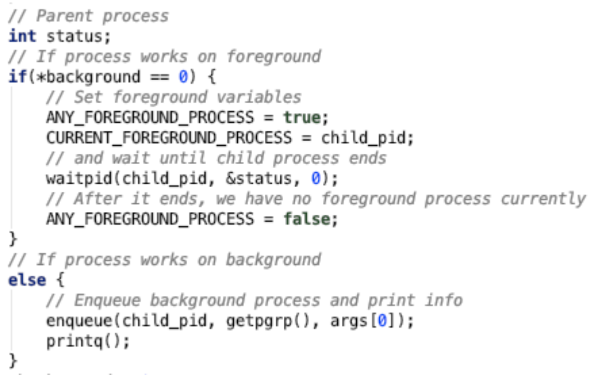


```
// Checks a command is executable in path variable
int check_path(char* args[], PATH_INFO* path_info, char** exec_path) {

    int i;
    int size = path_info->no_of_paths;

    for(i=0; i<size; i++) {
        char temp_path[128];
        strcpy(temp_path, path_info->paths[i]);
        strcat(temp_path, "/");
        strcat(temp_path, args[0]);
        temp_path[strlen(temp_path)] = '\0';

        int accessible = access(temp_path, F_OK | X_OK) ;
        // If accessible, initialize exec_path to the executable path
        if(accessible == 0) {
            *(exec_path) = strndup(temp_path, strlen(temp_path) + 1);
            // return (success)
            return 0;
        }
    }
    // If no executable found in path, return -1 (failure)
    return -1;
}
```

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, initialize executable_path variable to the path of the accessible file. Return 0 as success, -1 as failure. If returns -1, shell will print "command not found". Then waits for new input.

After getting executable path, we send our arguments to the function we created called "exec_command". In this function, we used fork() in order to create a child process to execute given arguments. If child process created successfully, we called execl function with our arguments - 32 arguments were given to the function because we assumed that there might be 32 arguments at max - and executable path. execl function stops executing arguments when it sees a NULL as an argument, that's why we passed 32 arguments to it.

We have a background variable that can have values 0 and 1. 0 means created process will run on foreground, that means when it's running, our shell won't be getting new inputs. Parent process must wait end of child process. If background is 1, that means child process will run in background so we can give new inputs to our shell while child process is running.

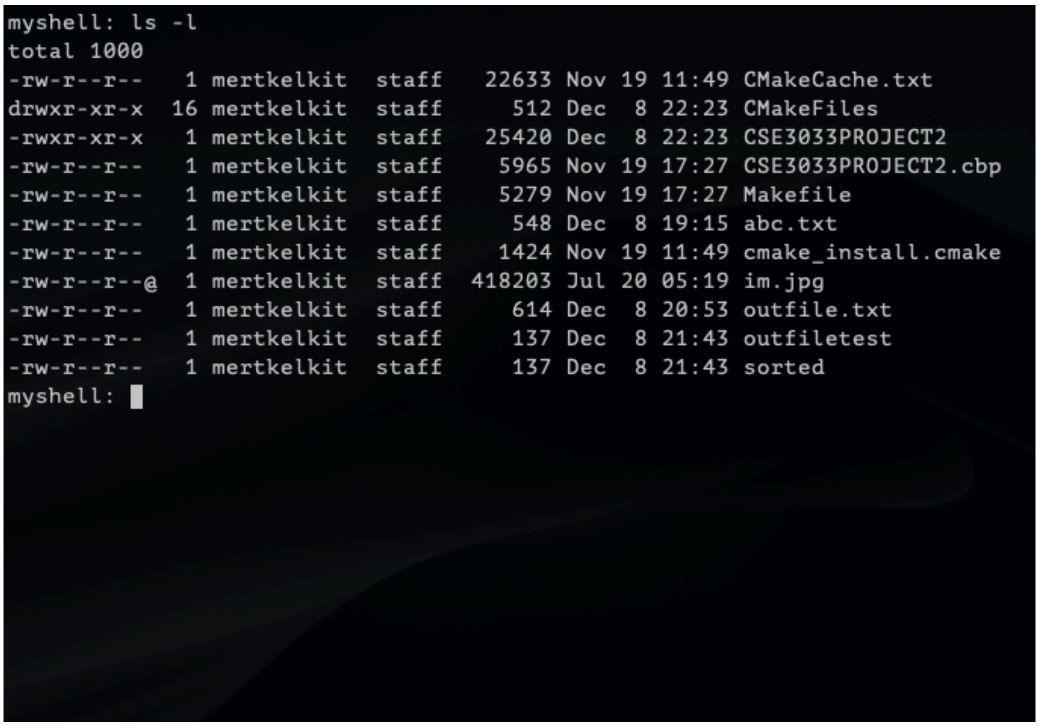


```
// Parent process
int status;
// If process works on foreground
if(*background == 0) {
    // Set foreground variables
    ANY_FOREGROUND_PROCESS = true;
    CURRENT_FOREGROUND_PROCESS = child_pid;
    // and wait until child process ends
    waitpid(child_pid, &status, 0);
    // After it ends, we have no foreground process currently
    ANY_FOREGROUND_PROCESS = false;
}
// If process works on background
else {
    // Enqueue background process and print info
    enqueue(child_pid, getpgrp(), args[0]);
    printq();
}
```

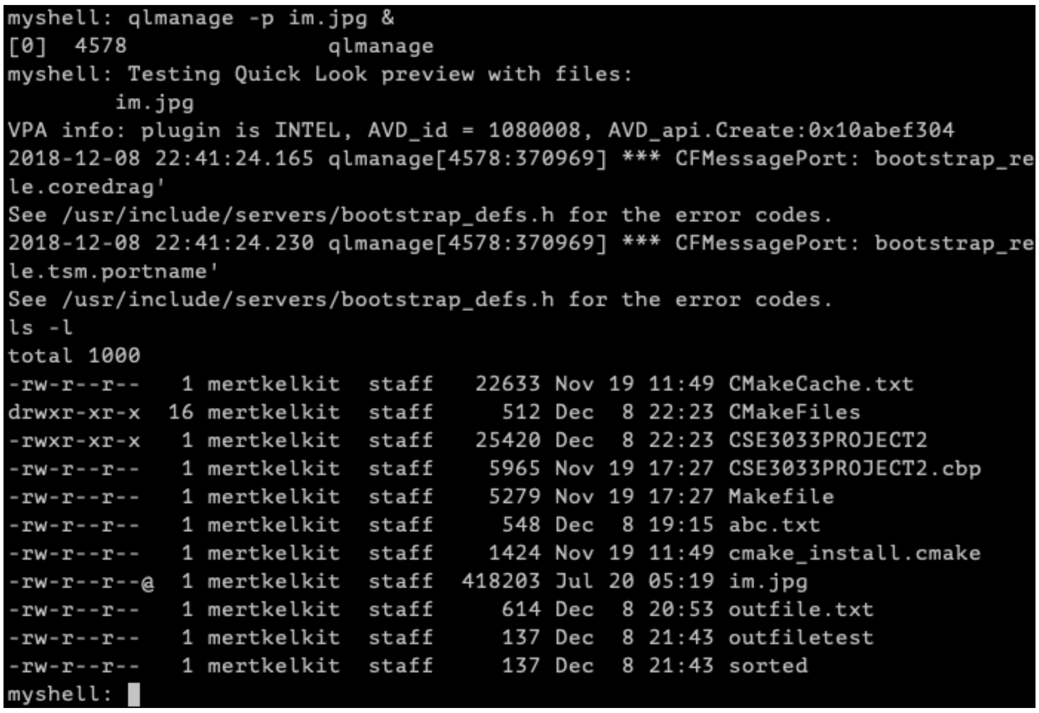
If background is 0, waitpid(child_pid, &status, 0) ensures that parent process will not continue until child process terminates.

If background is 1, child process will be enqueued to the background process queue which will be discussed later.

Example outputs for Part 1:



```
myshell: ls -l
total 1000
-rw-r--r--   1 mertkelkit  staff   22633 Nov 19 11:49 CMakeCache.txt
drwxr-xr-x  16 mertkelkit  staff     512 Dec  8 22:23 CMakeFiles
-rwxr-xr-x   1 mertkelkit  staff   25420 Dec  8 22:23 CSE3033PROJECT2
-rw-r--r--   1 mertkelkit  staff    5965 Nov 19 17:27 CSE3033PROJECT2.cbp
-rw-r--r--   1 mertkelkit  staff    5279 Nov 19 17:27 Makefile
-rw-r--r--   1 mertkelkit  staff     548 Dec  8 19:15 abc.txt
-rw-r--r--   1 mertkelkit  staff    1424 Nov 19 11:49 cmake_install.cmake
-rw-r--r--@  1 mertkelkit  staff  418203 Jul 20 05:19 im.jpg
-rw-r--r--   1 mertkelkit  staff     614 Dec  8 20:53 outfile.txt
-rw-r--r--   1 mertkelkit  staff     137 Dec  8 21:43 outfiletest
-rw-r--r--   1 mertkelkit  staff     137 Dec  8 21:43 sorted
myshell: 
```



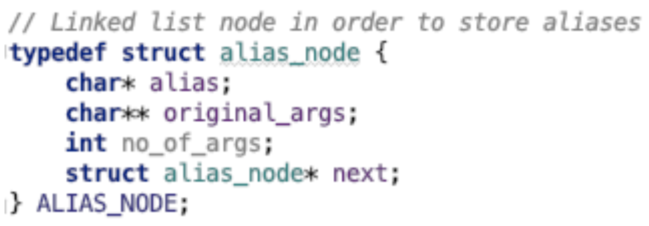
```
myshell: qlmanage -p im.jpg &
[0]  4578               qlmanage
myshell: Testing Quick Look preview with files:
        im.jpg
VPA info: plugin is INTEL, AVD_id = 1080008, AVD_api.Create:0x10abef304
2018-12-08 22:41:24.165 qlmanage[4578:370969] *** CFMessagePort: bootstrap_re
le.coredrag'
See /usr/include/servers/bootstrap_defs.h for the error codes.
2018-12-08 22:41:24.230 qlmanage[4578:370969] *** CFMessagePort: bootstrap_re
le.tsm.portname'
See /usr/include/servers/bootstrap_defs.h for the error codes.
ls -l
total 1000
-rw-r--r--   1 mertkelkit  staff   22633 Nov 19 11:49 CMakeCache.txt
drwxr-xr-x  16 mertkelkit  staff     512 Dec  8 22:23 CMakeFiles
-rwxr-xr-x   1 mertkelkit  staff   25420 Dec  8 22:23 CSE3033PROJECT2
-rw-r--r--   1 mertkelkit  staff    5965 Nov 19 17:27 CSE3033PROJECT2.cbp
-rw-r--r--   1 mertkelkit  staff    5279 Nov 19 17:27 Makefile
-rw-r--r--   1 mertkelkit  staff     548 Dec  8 19:15 abc.txt
-rw-r--r--   1 mertkelkit  staff    1424 Nov 19 11:49 cmake_install.cmake
-rw-r--r--@  1 mertkelkit  staff  418203 Jul 20 05:19 im.jpg
-rw-r--r--   1 mertkelkit  staff     614 Dec  8 20:53 outfile.txt
-rw-r--r--   1 mertkelkit  staff     137 Dec  8 21:43 outfiletest
-rw-r--r--   1 mertkelkit  staff     137 Dec  8 21:43 sorted
myshell: 
```

Here we called "qlmanage -p im.jpg &" command which blocks the shell normally. Adding & to the end of the command will make it run on background. After running it on background, as we can see we called new command "ls -l" that ran successfully.

Part 2:

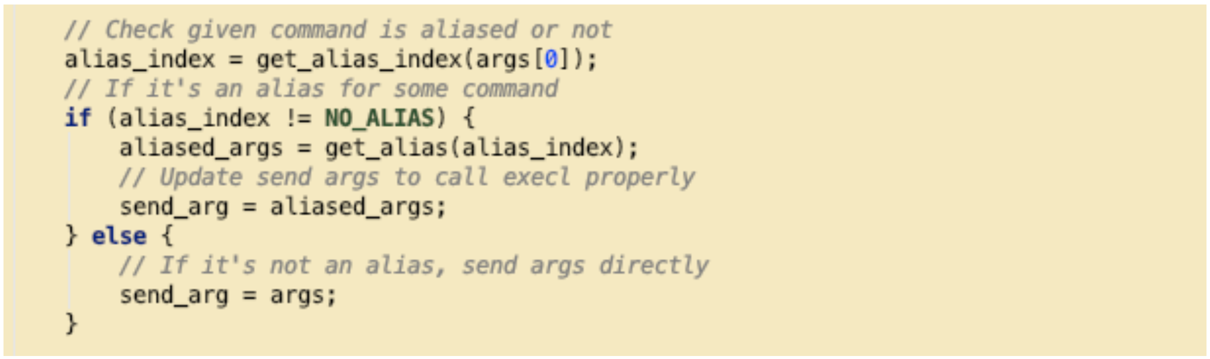
~ alias/unalias:

For these commands, we planned to store aliases and their corresponding arguments in a linked list. Here is our struct:



```
// Linked list node in order to store aliases
typedef struct alias_node {
    char* alias;
    char** original_args;
    int no_of_args;
    struct alias_node* next;
} ALIAS_NODE;
```

We declared root of the linked list globally and set it to NULL to make it reachable for responsible functions.



```
    // Check given command is aliased or not
    alias_index = get_alias_index(args[0]);
    // If it's an alias for some command
    if (alias_index != NO_ALIAS) {
        aliased_args = get_alias(alias_index);
        // Update send args to call execl properly
        send_arg = aliased_args;
    } else {
        // If it's not an alias, send args directly
        send_arg = args;
    }
```

After a command arrives to our program, first we check this command is in alias linked list or not. If it's in our alias list, we will set send_args to the corresponding arguments stored in alias list. If it's not in the list, we will set send_args to args. send_args is used to call execl function properly.

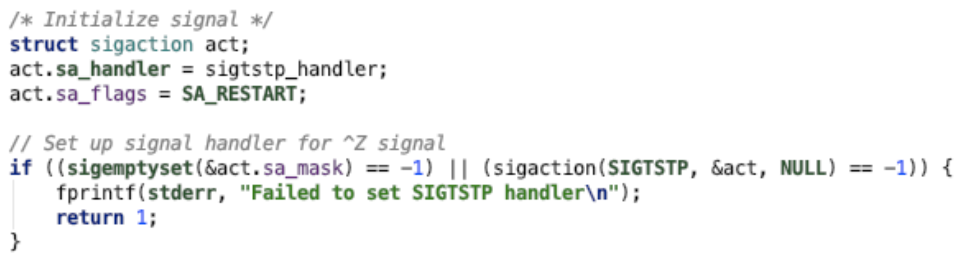
If user wants to use alias command to list all aliases (alias -l), our shell calls function "print_alias". This function iterates through linked list and print all nodes' informations to the screen.

If user wants to use alias command to register a new alias (alias "ls -l" list), our shell calls function "store_alias". This function parses given arguments as "original_args" and "alias", then append a new node to our linked list. We assumed that all commands that wanted to be aliased must given between 2 quotation marks. If no aliases were given or given alias is already in the list, our shell prints corresponding error messages.

If user wants to use unaries command to remove an alias(unalias list), our shell calls function "remove_alias". This function searches for given alias in the linked list, then removes the found node. If no nodes are found, shell prints an error message.

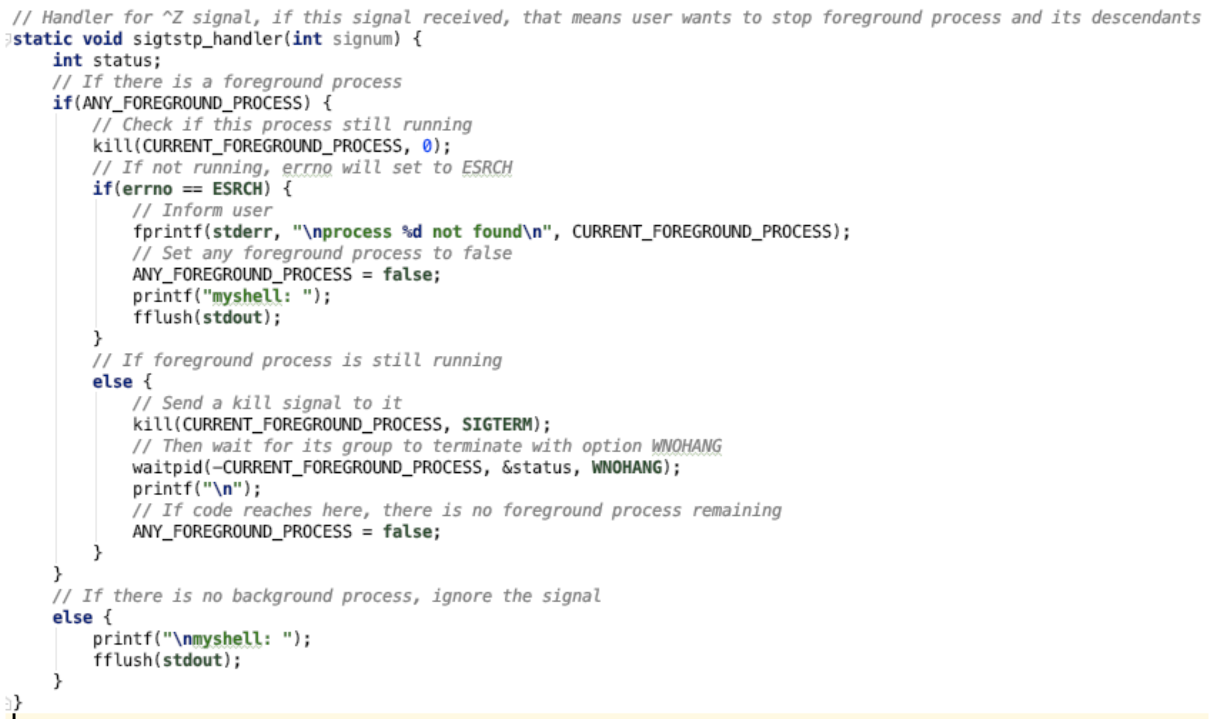
~ ^Z signal:

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



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

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

Here is the handler function:



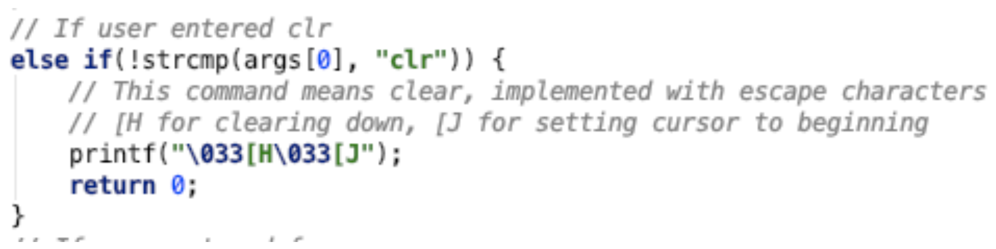
```
// Handler for ^Z signal, if this signal received, that means user wants to stop foreground process and its descendants
static void sigtstp_handler(int signum) {
    int status;
    // If there is a foreground process
    if(ANY_FOREGROUND_PROCESS) {
        // Check if this process still running
        kill(CURRENT_FOREGROUND_PROCESS, 0);
        // If not running, errno will set to ESRCH
        if(errno == ESRCH) {
            // Inform user
            fprintf(stderr, "\nprocess %d not found\n", CURRENT_FOREGROUND_PROCESS);
            // Set any foreground process to false
            ANY_FOREGROUND_PROCESS = false;
            printf("myshell: ");
            fflush(stdout);
        }
        // If foreground process is still running
        else {
            // Send a kill signal to it
            kill(CURRENT_FOREGROUND_PROCESS, SIGTERM);
            // Then wait for its group to terminate with option WNOHANG
            waitpid(-CURRENT_FOREGROUND_PROCESS, &status, WNOHANG);
            printf("\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("\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.

~ clr:

If user enters clr, we printed to the screen this:

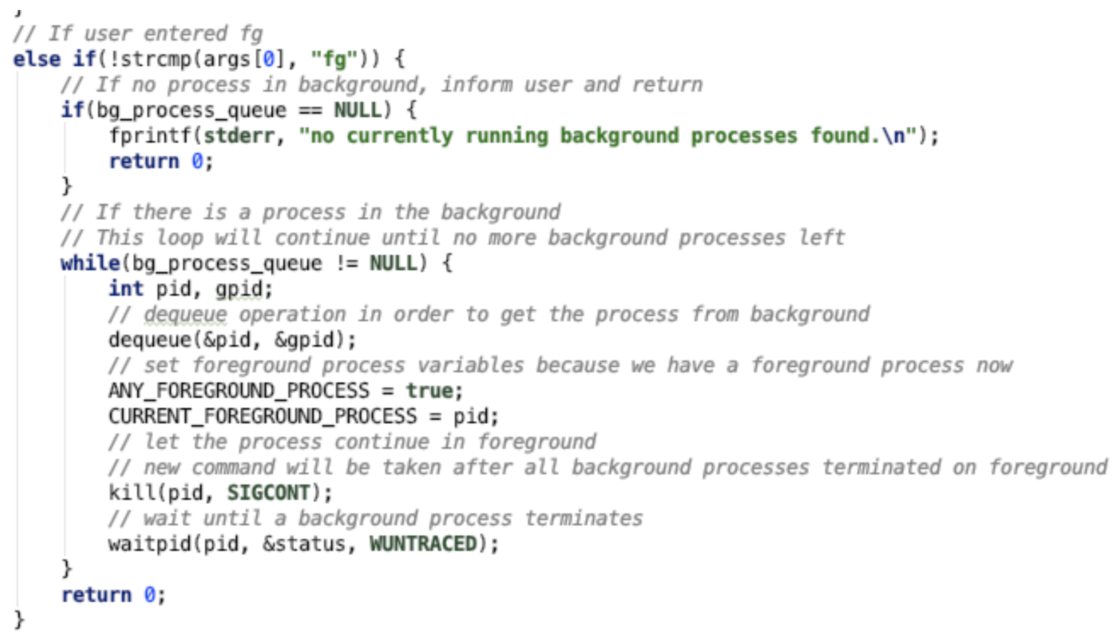


```
// If user entered clr
else if(!strcmp(args[0], "clr")) {
    // This command means clear, implemented with escape characters
    // [H for clearing down, [J for setting cursor to beginning
    printf("\033[H\033[J");
    return 0;
}
```

We tried system("clear"); that worked in Mac OS X machine, but it didn't work on an Ubuntu machine. So we printed "\033[H\033[J", \033 means octal ESC character, [H for clearing down side of the screen, J for setting cursor to the beginning.

~ fg:

As we said, we store all background processes in a queue. If user enters fg command, we dequeue processes one by one, then wait for them to terminate, one by one again.



```
// If user entered fg
else if(!strcmp(args[0], "fg")) {
    // If no process in background, inform user and return
    if(bg_process_queue == NULL) {
        fprintf(stderr, "no currently running background processes found.\n");
        return 0;
    }
    // If there is a process in the background
    // This loop will continue until no more background processes left
    while(bg_process_queue != NULL) {
        int pid, gpid;
        // dequeue operation in order to get the process from background
        dequeue(&pid, &gpid);
        // set foreground process variables because we have a foreground process now
        ANY_FOREGROUND_PROCESS = true;
        CURRENT_FOREGROUND_PROCESS = pid;
        // let the process continue in foreground
        // new command will be taken after all background processes terminated on foreground
        kill(pid, SIGCONT);
        // wait until a background process terminates
        waitpid(pid, &status, WUNTRACED);
    }
    return 0;
}
```

If queue is empty, inform user that there is no background process. If queue is not empty, get head of the queue, send continue signal to dequeued process and wait for its termination. This process will be repeated until background process queue is emptied.

~ 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 check_is_builtin(char* args[], int arg_count) {
    int status;
    // If user entered exit
    if(!strcmp(args[0], "exit")) {
        // Check background processes
        if(bg_process_queue != NULL) {
            // inform user
            fprintf(stderr, "there are some processes that are running background\n");
            fprintf(stderr, "type 'fg' and ^Z to terminate these processes one by one\n");
            printq();
            // return to get new commands
            return 0;
        }
        // If no background processes running currently, exit program
        else {
            printf("session ends\nbye!\n");
            exit(0);
        }
    }
```

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

Part 3:

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

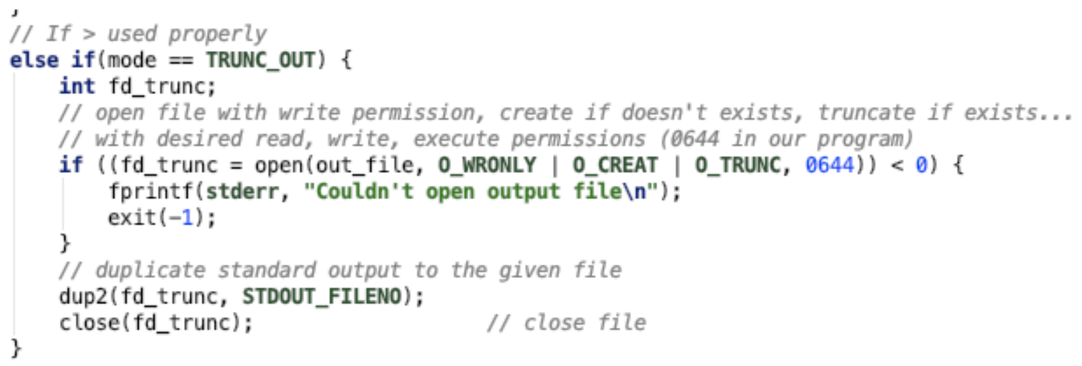
For setting redirections, we parsed given arguments and checked syntax errors. If there is no error, mode of the redirection should be one of these:

```c
/* File opening modes for I/O redirection */
#define TRUNC_OUT 2      // For truncation
#define APPEND_OUT 3     // For append
#define ERR_OUT 4        // For printing err
#define IN_MODE 5        // For using file as input
```

According the mode of redirection, we wrote/red given files. After that, we called exec function to execute given arguments with given standard input / output / error files.

~ prog args > file:

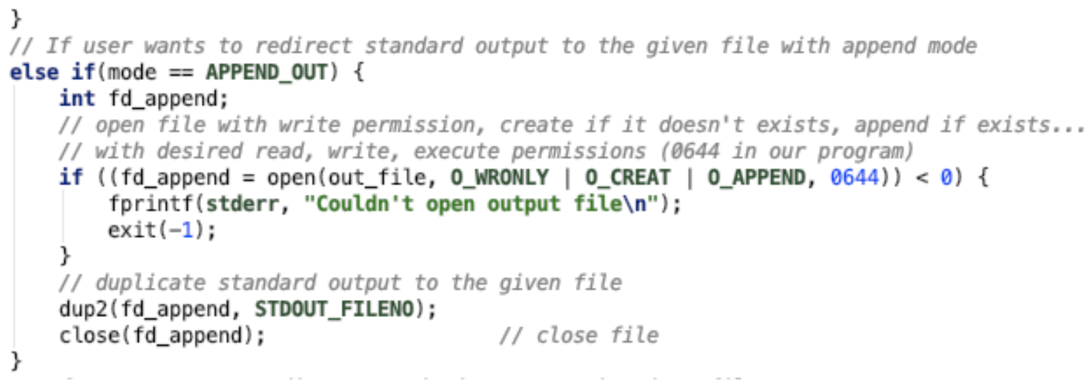
With this redirection mode, we need to redirect output of the program to given file. If this file doesn't exist, create the file. If file exists, truncate it.



```
// If > used properly
else if(mode == TRUNC_OUT) {
    int fd_trunc;
    // 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_trunc = open(out_file, O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
        fprintf(stderr, "Couldn't open output file\n");
        exit(-1);
    }
    // duplicate standard output to the given file
    dup2(fd_trunc, STDOUT_FILENO);
    close(fd_trunc);                  // close file
}
```

~ prog args >> file:

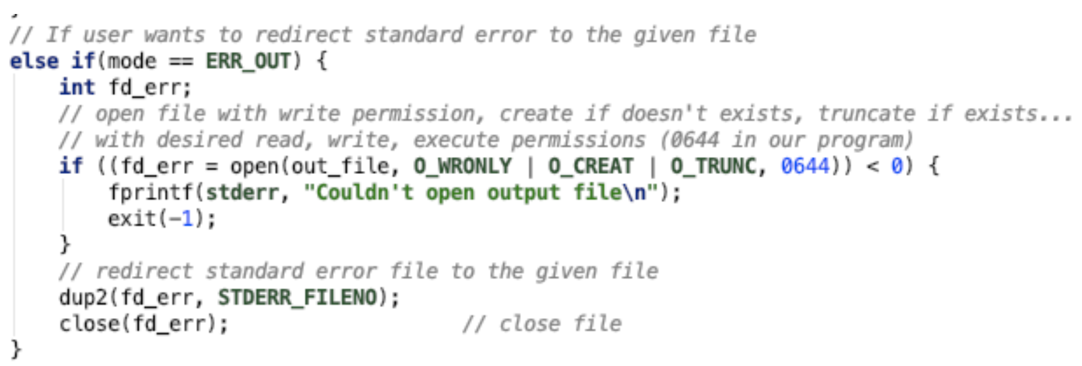
With this redirection mode, we need to redirect output of the program to given file. If this file doesn't exist, create the file. If file exists, append to it.



```
}
// If user wants to redirect standard output to the given file with append mode
else if(mode == APPEND_OUT) {
    int fd_append;
    // open file with write permission, create if it doesn't exists, append if exists...
    // with desired read, write, execute permissions (0644 in our program)
    if ((fd_append = open(out_file, O_WRONLY | O_CREAT | O_APPEND, 0644)) < 0) {
        fprintf(stderr, "Couldn't open output file\n");
        exit(-1);
    }
    // duplicate standard output to the given file
    dup2(fd_append, STDOUT_FILENO);
    close(fd_append);                 // close file
}
```

~ prog args 2> file:

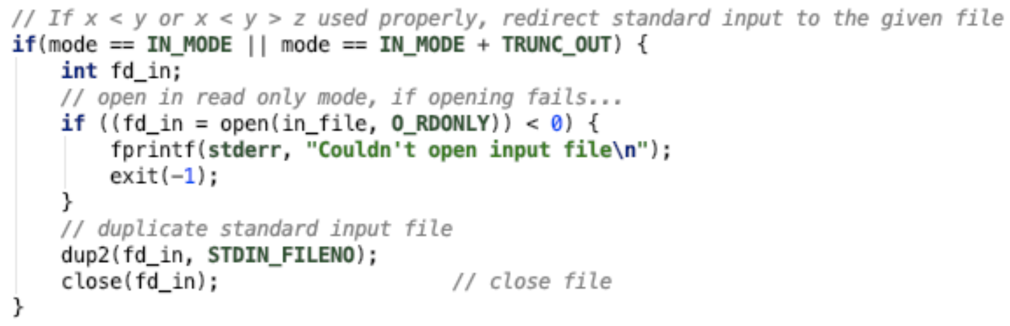
With this redirection mode, we need to redirect error output of the program to given file. If this file doesn't exist, create the file. If file exists, truncate it (our assumption, project document didn't specified it).



```
// If user wants to redirect standard error to the given file
else if(mode == ERR_OUT) {
    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(out_file, O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
        fprintf(stderr, "Couldn't open output file\n");
        exit(-1);
    }
    // redirect standard error file to the given file
    dup2(fd_err, STDERR_FILENO);
    close(fd_err);                    // close file
}
```

~ prog args < file:

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

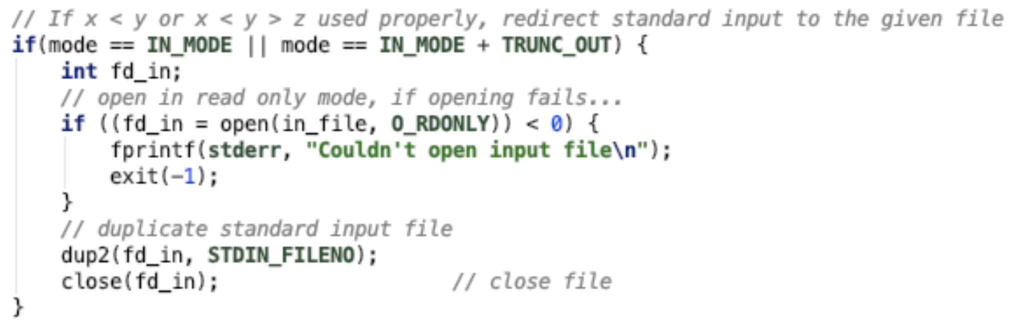


```
// If x < y or x < y > z used properly, redirect standard input to the given file
if(mode == IN_MODE || mode == IN_MODE + TRUNC_OUT) {
    int fd_in;
    // open in read only mode, if opening fails...
    if ((fd_in = open(in_file, O_RDONLY)) < 0) {
        fprintf(stderr, "Couldn't open input file\n");
        exit(-1);
    }
    // duplicate standard input file
    dup2(fd_in, STDIN_FILENO);
    close(fd_in);                   // close file
}
```

There is an OR operation because both < and < > redirect operations require opening an input file. We handled < > this case's input file redirection here.

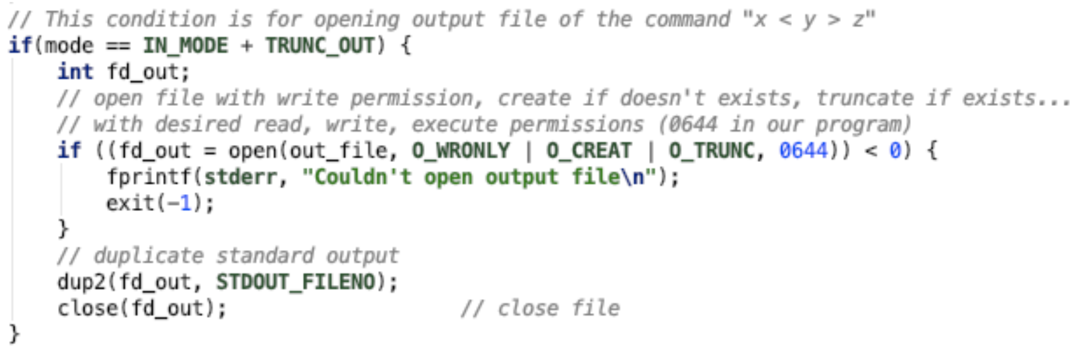
~ prog args < in_file > out_file:

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



```
// If x < y or x < y > z used properly, redirect standard input to the given file
if(mode == IN_MODE || mode == IN_MODE + TRUNC_OUT) {
    int fd_in;
    // open in read only mode, if opening fails...
    if ((fd_in = open(in_file, O_RDONLY)) < 0) {
        fprintf(stderr, "Couldn't open input file\n");
        exit(-1);
    }
    // duplicate standard input file
    dup2(fd_in, STDIN_FILENO);
    close(fd_in);                   // close file
}
```

in_file redirected here,



```
// This condition is for opening output file of the command "x < y > z"
if(mode == IN_MODE + TRUNC_OUT) {
    int fd_out;
    // 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_out = open(out_file, O_WRONLY | O_CREAT | O_TRUNC, 0644)) < 0) {
        fprintf(stderr, "Couldn't open output file\n");
        exit(-1);
    }
    // duplicate standard output
    dup2(fd_out, STDOUT_FILENO);
    close(fd_out);                  // close file
}
```

out_file redirected here.

After handling redirections, we used the same manner as in function exec_command. Given arguments taken until a redirect operator, these arguments checked if it's an alias or not or builtin or not. If it's an alias or a builtin command, arguments and executable path are passed to the execl function in a child process. If command not found, an error message is printed.

Also redirection has same manner with exec_command with background / foreground processes.

If it's required to be a foreground process, parent process will set foreground process variables and wait until child process terminates.
If it's a background process, it will be enqueued to the background process queue, then shell continues getting input from user.

References:

[1] stackoverflow.com

[2] linux.die.net

[3] docs.oracle.com

[4] geeksforgeeks.com

[5] unix.com

[6] http://akademik.marmara.edu.tr/zuhal.altuntas