Tunaberk Almaci
Aybars Inci

# COMP 304 Project 1 Report

**Part1**

We implemented a “path_finder” function  to get the system call’s absolute path. In order to do that, initially we assigned all the paths to a string by getenv(1) function. Then we sliced every path inside that string with strsep. After that, we used snprintf to assign  that absolute path to a string. In the end we used the “path_finder” method in the “process_command”. We get the wanted commands( command that entered by user to shell) path in process command and we use the path in “execv()” functions first parameter, in the second parameter we use an array that contains all arguments with null string in the end.

**Part 2**

In this part, we implemented the “shortdir” command. To satisfy the specified requirements, we used different strategies. Since this command is expected to run on different shell sessions, we defined a global text file called “shortdirs.txt” under “tmp” folder. We chose this folder because this folder exists in every UNIX based operating systems. Depending on the option provided to shortdir command, we handled the caes accordingly. For “set” option, we take the current working directory with getcwd() command. We store the path to a variable to use later. Firstly, we open the file “shortdirs.txt” by “a+” option. By providing this option to fopen() function, we append the file if it exists or we create it from scratch if it does not. Then, by checking every line, we look for an existing association for the current directory. The format of the associations is “name -> directory”. Since a directory can have more than one association but not the other way around, if there is an association with the given name, we mark the line of that association. Then, we create a temporary file to copy the previous associations to the newly created temporary file. After we change the aforementioned line, we close the files and rename the new file to the previous file’s name. If no association with the given name exists, we just simply append a new association. If the provided option is “jump”, we fetch the name of the searched association, open the “shortdirs.txt” file with “r” option, and look at each line to find the required name. If the name exists, we save the path associated with that name to a variable and then invoke chdir() function with that variable. If no such association exists, we simply prompt the user about it. If the option is del, we fetch the name of the association, open the aforementioned file, look for an association. If such an association exists, we store the line of the association into a variable, create a temporary file called “temp_shortdirs.txt”, copy every line other than the line containing the association to the newly created file, and finally rename the new file with the previous file’s name. If no such association exists, we prompt the user about it. If the provided option is “clear”, we create a temporary file with the “w” option. The reason for this is that the “w” option creates an empty file. By renaming this file to “shortdirs.txt”, we simply clear the contents of the “shortdirs.txt” file. Finally, if the option is “list”, we open the “shortdirs.txt” file with the “r” option, loop over the lines and simply print them.

Tunaberk Almaci
Aybars Inci

**Part 3**
We implemented this part by starting to define the delimiters that could exist inside the text file. There are over 10 escape sequences in C, but we decided to use , . : ; tab newline carriage vertical tab and feed page break. After we decided on this, we fetch the word, color and the file path from the user. Then, we open the file with the "r" mode and create a while loop to look into every line. If no such file exists, we prompt the user and exit the execution. Otherwise, we store every word into a variable, and if the variable is equal to the specified word, we simply flag that line. If the line is flagged, we then loop through the words in that line, print every word with the exception that the specified words will be colored.

**Part 4**
In this part, we started by creating a global file called "sch_jobs.txt" inside the "tmp" folder. We then fetch the time and music file path from the user. Since the user specifies the time in "HH.MM" format, we tokenize the hour and minute part separately. We then use the function created in the first part to find the path of the "crontab" and "rhytmbox-client" commands. After storing those paths in different variables, we open the "sch_jobs.txt" file with the "w+" mode. The reason for this is that we want to overwrite an existing scheduled job but we also want to read the scheduled jobs later. After opening the file, we write the full command that is required to run the crontab command properly. This corresponds to:
%minute %hour * * * XDG_RUNTIME_DIR=/run/user/$(id -u) %rythmbox_path
--play-uri=%music file
After the full command is written into the aforementioned file, we close the file pointer and remove it. Then, we define the argos required to run the crontab command with execv() function properly. These arguments are "crontab" "/tmp/sch_jobs.txt" and NULL respectively. We then fork the parent process so that the code would not get out of the shell. After forking the parent process, the child process invokes execv() function with the path to the crontab executable and the arguments array we created as the parameters of the function. Meanwhile, the parent process waits for the child process to terminate its execution.

**Part 5**
In this part, we fetch the option and the paths of the files from the user. We open both of the files with the "r" mode. If both of the file pointers are null, we prompt the user with a message indicating that none of the files exists. If only one of the files does not exist, we prompt the user with an appropriate message and terminate the execution of the program. Similarly, if both of the files exists but even one of them is not a txt file, we prompt the user and terminate the execution of the program. If the provided option is "-a", we loop through the lines and check if both of the lines are identical. If they differ, we print the corresponding lines and increment the count variable by 1. At the end, we check if both of the files have reached to their ends. If even

one of them has not finished yet, we inform the user about this with an appropriate message. But if both of them finished, we inform the user about number of lines they differ. For the second option; namely “-b”, we accept any type of file. We then check the contents of the files byte by byte and compare them. If they differ, we increment the count variable by one. At the end, if one of the files is longer than the other, we keep adding the bytes to the count variable. We then display the number of bytes that the files differ in the terminal.

**Part 6**

In this part we wanted to create an entertainer command. This commands name is “iambored”. This command contains 3 mini games: Magic - 8 Ball, Tic Tac Toe, Guess my height. When you call this command you encounter these 3 options. You can choose an option by entering 1, 2 or 3. In Magic - 8 Ball mini game, the user asks his/hers question to shell and shell answers to the user. We implemented this game in a while loop. In this while loop we ask “fgets” and “sscanf” functions. However, the answer does not depend on the user's question. We have 20 different answers and answers are chosen randomly by random function  “rand()”. After the answer is shown we set 2 second delay with the “sleep” function before showing the options table again. Our second mini game is Tic Tac Toe. in this game, the user is playing against our Tic Tac Toe bot. User plays his moves by entering coordinates of the Tic Tac Toe table. These coordinates are shown below.

| | | |
|---|---|---|
| 11 | 12 | 13 |
| 21 | 22 | 23 |
| 31 | 32 | 33 |

User must enter exact same coordinates as shown in the table. In this game, initially, we created a 3x3 char array to implement the table. After creating, we assigned all the array elements to ‘ ‘. then we implemented a while loop to maintain the game. In this while loop we ask user to enter his coordinates of his move. We check availability of the coordinates by “valid_input” function. If the entered coordinates is not available, then we ask user again to enter valid coordinates. After user enters valid coordinates, we assign ‘X’ character to user specified elements of the array. In order to do that we use ‘user_move’ function. After user plays his turn, tic tac toe bot plays its turn. Tic tac toe bot checks if there is a user win condition first. If there is a win condition, tic tac toe bot prevents the win condition by choosing the special coordinates. In order to do this action we implemented “ai_move” function. If there are no win condition tic tac toe bot choose a coordinate randomly by assigning random coordinates via using rand() function. When tic tac toe bot chooses his coordinates ‘O’ character is assigned to char array. After every move we check the state of the game by ‘win_condition’ and ‘check_draw’ function. “win_condition” funtion checks the array and determine who won the game. “check_draw” checks the array and determines whether there is a draw or not. We used “vis_table” function to display the tic tac toe board in the shell after every move. In “Guess my height” game we ask user to guess Andy’s height continuously. We specified the height of Andy by using rand() function and it can be between 50 and 200 cm. Then we created a while loop to maintain the game until user finds the correct height. If user guesses higher value, shell tells user to go lower. If user guesses lower value, shell tells user to go higher.

| | | |
|---|---|---|
| X | | |
| O | X | |
| | X | O |

**This is an example of win condition of the user.**