Efe Yılmaz

Doruk Özer

COMP304

04.04.2021

Project 1

We are compiling the shell using gcc.

Part 1: Working

We are tokenizing the PATH to get the possible locations. From here, we are using strcat() to create a new address with the command we’re trying to execute, which is the first element in the argument array. Using the access command, we are checking if the command exists within that address and if it exists, we are executing the prompt within that location using execv().

Part 2: Working

We are creating a .txt file which contains the aliases for the directories and the exact path for these directories. From here, we are executing all of the required commands by accessing the .txt file and reading or modifying it. Using .txt also allows us to store these short directory names between sessions, or even when the computer is rebooted.

We are storing the shortdir information in the form of SHORT\_NAME>EXACT\_PATH. We are creating these with the “set” command, with each new line representing another short name and exact pair combination. This format allows us to tokenize each line with > to separate the names and paths, which allows us to easily search for the requested path in “jump” and “del” commands. Our “del” function creates a new .txt file with the deleted association removed. The old .txt file is then deleted, and the new .txt file is renamed to the old one.

Since we are using .txt files to store name-directory pairs, our “clear” function simply deletes the existing .txt file and creates a new one. In a similar vein, our “list” function simply reads the lines from the .txt file and prints them to the console.

Part 3: Working

We have two specific requirements for this part to function correctly: firstly, we assume that the searched word is written in letters and not symbols, and secondly, that the searched text file is written in a format which puts spaces between the punctuation marks and other sentence elements. For example, the sentence:

I, Alex, love eating food (especially burgers).

Should be formatted as such in the .txt file:

I , Alex , love eating food ( especially burgers ) .

We are checking for the validity of color and filename inputs initially, and returning an error message if either of these values are invalid. For color, we are converting the char\* to char, which allows us to use a switch case to easily modify the color value within the system. Using fgets and a secondary buffer array, we divide the lines. From here, we are tokenizing each line of the file using the space character. Since the tokenizer modifies the line itself, we are using the buffer array to store the modified sentence. We are also tracking whether a word exists in a sentence or not via a boolean.. If a searched word is found, we are using strcat to create a new string which carries the required input for changing the word’s color. Words that do not match are added to the buffer array without modification. If the searched word was found, the buffer array is printed, and at the end of the loop, the buffer array is cleared for the next line.

Part 4: Working

We are using a .txt file to store the information about the alarm. This information contains time, which is tokenized into hours and minutes by the “.” character. We are then adding this information to the new .txt file in reverse order: first minutes, then hours, then the address of the music file and lastly, the address of the .txt file and any additional arguments. We are creating another process to execute crontab, which reads the information from the .txt file and sets the alarm accordingly.

Part 5: Working

After handling the .txt access for both files this function has two modes, so we will be explaining them separately.

In the case of string comparison, we are reading every line and using strcmp() to check if they are equivalent. We keep track of mismatches using integer values, and in the event of a mismatch, we are printing those lines. In the event that one of the files ends prior to the other, we are printing every remaining line from the latter file. Lastly, we are printing the total amount of mismatched lines. If all lines are equivalent, we print the identical message.

In the case of bitwise comparison, we are moving through the file character by character. In the event that two characters are different, we are incrementing a difference value, which is then checked for it’s original value in the end. If it is not it’s original value, we print the value as the number of differences, otherwise we state that there are no differences.