COMP 304 - Operating Systems
Project 1: Hshell

Spring 2024

Due: 12.05.2024

Mete Erdoğan, 69666
Şebnem Demirtaş, 76813

Table of Contents

Note: We implemented all the requested parts of the project as explained in this report. Also note that gcc-12 is required to run the kernel module for part-4 (**installation: sudo apt install gcc-12**).

Part I - Execv and Background Programs

Running Linux programs necessitates executing the execv command with the program's full path and its arguments. In our implementation, to locate the path for a specific program, all directories listed under PATH are searched, and the resulting paths where the program is found are combined for use in the execv command. This is implemented in the **"search_and_run_command"** method in our implementation as below:

```
void search_and_run_command(struct command_t *command, int issudo){
        //PART 1
        int exist_checker=0;
        //getting path
        char *path = getenv("PATH");
        char checkedPath[4096];
        char* pTokens = strtok(path, ":");

        while(pTokens!=NULL){
                //building path to command
                snprintf(checkedPath, sizeof(checkedPath), "%s/%s", pTokens, command->name);
       //check if there is an accesible file
        if (access(checkedPath, X_OK) == 0) {
            exist_checker = 1;
            break;
        }
        pTokens = strtok(NULL, ":");
        }
        if (exist_checker) {
                if(issudo == 1) execv("/usr/bin/sudo", command->args);
                else execv(checkedPath, command->args);
    } else {
        printf("-%s: %s: command not found\n", sysname, command->name);
    }
}
```

Figure 1: search_and_run_command method implementation.

Furthermore, we can implement the capability of running background processes with the following structure inside the **"process_command"** method. This way, the parent will not wait for the child process if the background is activated with the ampersand symbol "&":

```
if (command->background == false) {
        waitpid(pid, &status, 0);
}
```

Figure 2: Code snippet enables the ability to run in the background.

Also, although the **"cd"** and **"exit"** commands were requested to be implemented in the project description, these commands were already implemented in the skeleton code given with the project.

Part II - Program Piping

In our **“process_command”** function, we call the **“pipe_function”** if the command is followed by other commands that are denoted by the “|” symbol.

Since process_command is also called from pipe_function, if there are multiple pipes, it calls **“pipe_function”** again, which is recursive.

Relevant part of process_command is as follows:

```
if(command -> next != NULL){
        return pipe_function(command); //indirect recursion inside process_command
}
```

Figure 3: Code snippet from process_command method that performs indirect recursive calls

The pipe_function creates children for the left and right parts of the pipe. The important part is to direct the output of the method on the left as the input of the method on the right. If there are multiple pipes, this method will be called again and the same things will be done for the other pipe.

The parent waits until the children are finished before returning “success”.

```
// Function to perform program piping for Part-2
int pipe_function(struct command_t *command){
        //Create a pipe
    int fd[2];
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return EXIT;
        }

        //First child
        pid_t left = fork();
        if(left == 0){
        close(1);
        //Close read end
        close(fd[0]);
        // Redirect stdout to the write end of the pipe:
        dup2(fd[1],1);
        //Close write end
                close(fd[1]);
                search_and_run_command(command,0); //running command
        }

        //Second child
        pid_t right = fork();
        if(right == 0){
        close(0);
        close(fd[1]);
        // Redirect stdin to the read end of the pipe
        dup2(fd[0],0);
                close(fd[0]);
                process_command(command->next); //Possible recursive
                exit(0);
        }
        //Parent process
        close(fd[0]);
        close(fd[1]);
        //Wait children to finish:
        waitpid(left,NULL,0);
        waitpid(right,NULL,0);
        return SUCCESS;
}
```

Figure 4: “pipe_function” implementation.

Part III

1. Auto-complete

For the auto-complete functionality, the prompt function is modified, more specifically, the place that handles the tab key press is modified. The modification is as below:

```
if (c == 9) {
        if (index == 0) continue;
        char *buf_cpy = strdup(buf);
        fname = (char *)malloc(4096 * sizeof(char));
        struct autocomplete_struct *match;
        int command_or_filename = check_command_or_filename(buf_cpy, fname); //update buffer and check
        if (command_or_filename) { // complete filename case
                match = directory_complete(fname); //find all matching files in the directory
                if (match->count == 1) {
                        for (int i = strlen(fname); i < (int) strlen(match->matches[0]); i++) {
                                putchar(match->matches[0][i]);
                                buf[index++] = match->matches[0][i];
                        }
                        c = ' ';
                }else if (match->count > 1) {
                        printf("\n");
                        for (int i = 0; i < match->count; i++) {
                                if(strstr(buf,"cd")==NULL) printf("%s ", match->matches[i]);
                                else if (match->matches[i][strlen(match->matches[i])-1]=='/') printf("%s ", match->matches[i]);
                        }
                }
                printf("\n");
                show_prompt();
                printf("%s", buf);
        }else { // complete command case
                char *command = *buf_cpy ? strdup(buf_cpy) : NULL;
                command = fname;
                match = command_complete(command); //find all matching commands
                if (match->count == 1) {
                        for (int i = strlen(command); i < (int) strlen(match->matches[0]); i++) {
                                putchar(match->matches[0][i]);
                                buf[index++] = match->matches[0][i];
                        }
                        c = ' ';
                }else if (match->count > 1) {
                        printf("\nAvailable commands: \n");
                        for (int i = 0; i < match->count; i++) printf(" - %s \n", match->matches[i]);
                        printf("\n");
                        show_prompt();
                        printf("%s", buf);
                }
        }
        free_autocomplete_struct(match); //free match struct
        free(fname); //free fname
        if (c == 9) {
                continue;
        }
}
```

Figure 5: Code snippet showing the modifications inside "prompt" function for handling tab key press.

Here, we first use the **"check_command_or_filename"** function to determine whether we should complete the initial string to a command or a filename and update the buffers accordingly. Then, we use a struct as below to save our matches that our found as explained in the following sections:

```
struct autocomplete_struct {
        char **matches; // matchings
        int count; // matching count
};
```

Figure 6: autocomplete_struct.

a. Completing Filenames:

If the **"check_command_or_filename"** function determines whether an initial command is already given such as we are trying to use the "cat" function on a file that starts with the letter "a", then we will list all the files with the initial letter "a" in the current directory. If there is only one file with the initial "a", then the function will complete the buffer rather than listing all of the files.

Here, the **"directory_complete"** method is used to find all the matching filenames in the current directory.

- If there are multiple matching files:

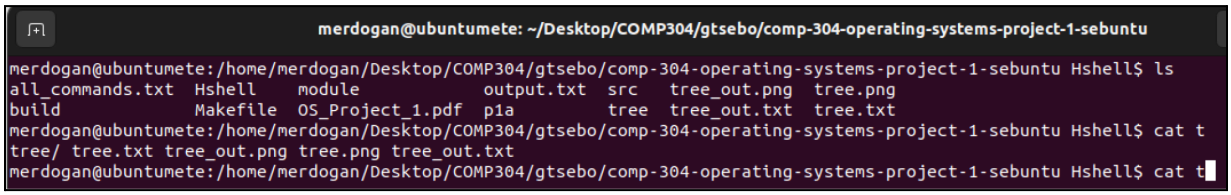

Figure 7: Auto-complete example for completing filenames with the initial "t".

- If no initial letters given after a command, list all files:

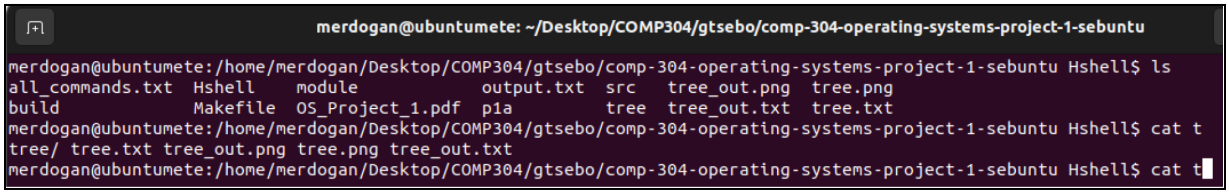

Figure 8: Auto-complete example for completing filenames that list all files.

- If we write cd and hit tab, we list only the subdirectories:

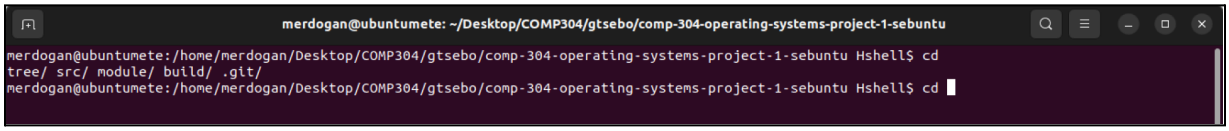

Figure 9: Auto-complete example for completing filenames after "cd" command.

b. Completing Commands:

If there is only the given initials but not a prior command, pressing the tab will complete the command. For this case, when the shell is first compiled, in the main function we use the **"save_available_commands"** to search all the commands in all directories in the PATH directory, and save them to a text file called "all_commands.txt" together with the custom commands psvis, regression, textify and hdiff. Here, there may be some duplicate commands coming from the PATH directory, so that we remove all of them before the save operation.

Then, if the required tab press occurs, we search all the matching commands from this text file using the **"command_complete"** method. If there is only one match, we directly update the buffer with the matching command, or list all the available commands if there are multiple of them as below:

Figure 10: Auto-complete example for completing commands that start with the initials “ps”.

2. Hdiff

In this part, we are asked to implement a method that compares files, either line-by-line or byte-by-byte.

The command gets the following arguments:

hdiff [<mode_flag>] <file_name1> <file_name2>

If the mode flag is -a, or not given, we compare line by line. For this part, we got the line using the getline() function and compared them. In the end, we did not forget to add the lines from the longer file as different lines also.

If the mode flag is -b, bit by bit comparison is made. For that, we read the files bit by bit with a while loop and increment the difference count as we encounter them. We use a buffer memory for the files, in the end we do not forget to free them.

Sample outputs can be seen:

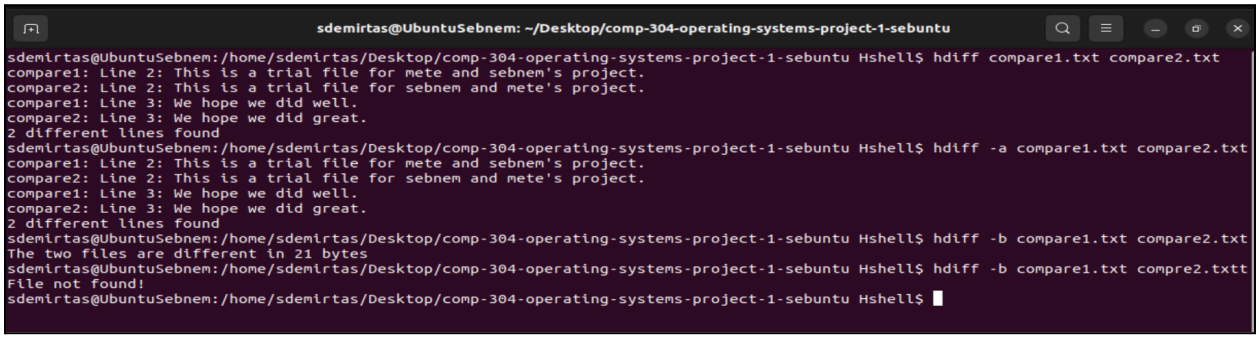

Figure 11: Sample outputs for hdiff function.

3. Custom Commands

a. regression

We have written a command that would take the coordinates(x,y) of the points in a 2D plane, and perform either linear or polynomial regression and plot it.

regression <file_name> [<polynomial_indicator>] [<degree>]
Description of the parameters:

- **file_name**: The name of the text file containing the data.
- **polynomial_indicator**: Optional. If -p is written, then polynomial regression is performed. If not indicated, then linear regression is performed.
- **degree**: If type is indicated as polynomial, then the degree of the polynomial regression should also be entered as the third argument.

(If degree is given as 1, again, linear regression is performed. For simplicity and ease of use, we have implemented linear regression as a case where no extra argument is needed.)

The plots are created through the "gnuplot" library.

Linear Regression

If linear regression is chosen, intercept and the slope is calculated according to the points given, and the plot is drawn.

Let the input file be input.txt containing the following:

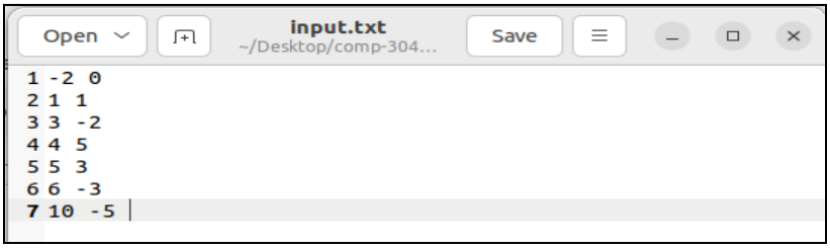

Figure 12: input.txt file containing x and y coordinate data for regression.

Command is used for linear regression as follows: regression input.txt

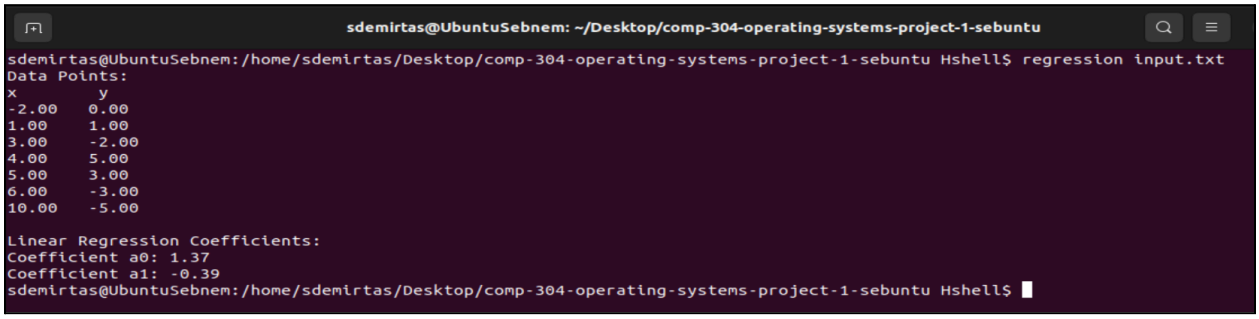

Figure 13: Sample output for linear regression using regression command. In the command prompt, the coefficients (y-intercept and the slope) are shown.

The graph is saved as “plot.png”. Data points and the fitted line can be seen:

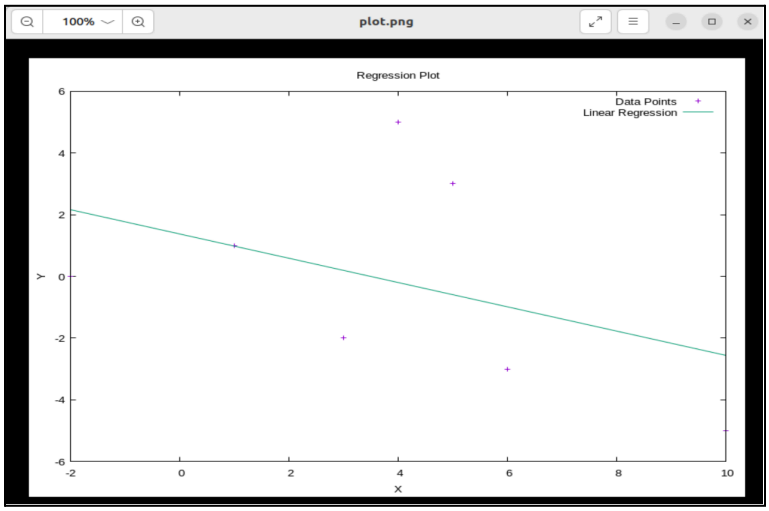

Figure 14: Example output graph generated by the regression command.

Polynomial Regression

Polynomial regression is indicated with the -p parameter. If we use the same input file following outputs will be obtained:

Polynomial fitting with degree = 2

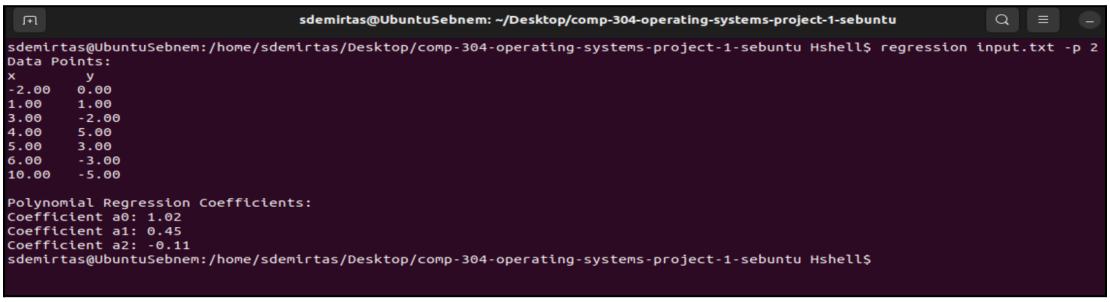

Figure 15: Output of polynomial regression with degree 2 using the regression command.

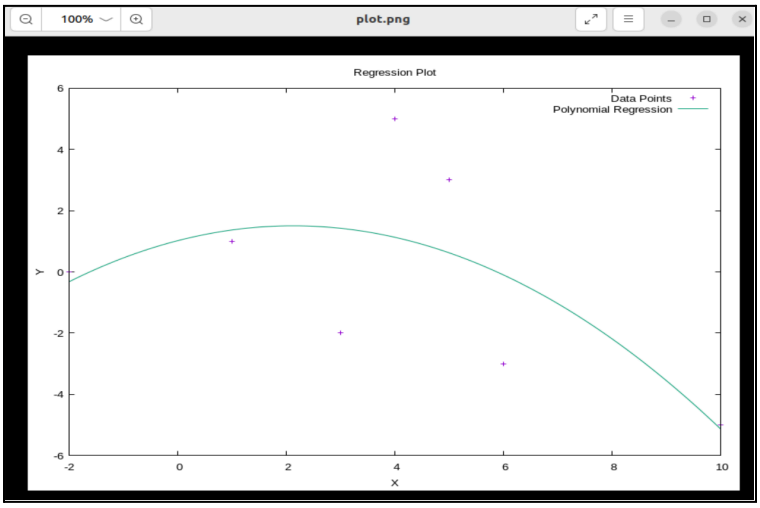

Figure 16: Example output graph for second degree polynomial regression generated by the regression command.

Polynomial fitting with degree = 4:

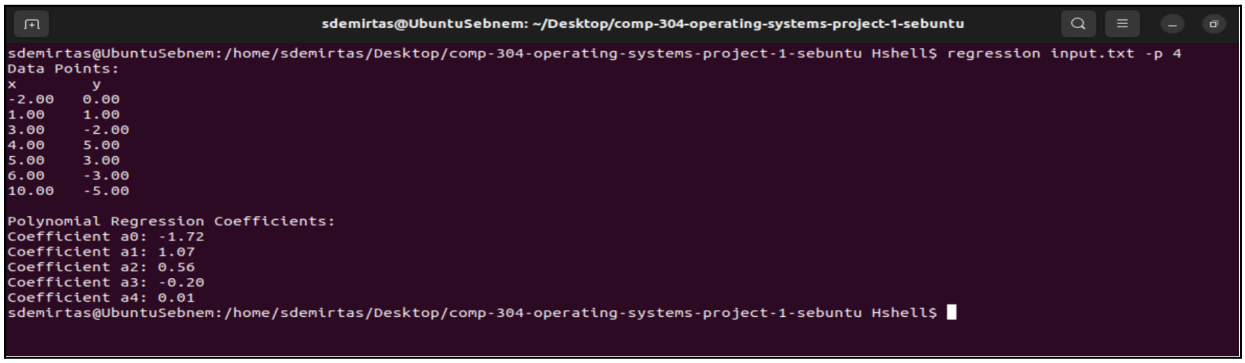

Figure 17: Output of polynomial regression with degree 4 using the regression command.

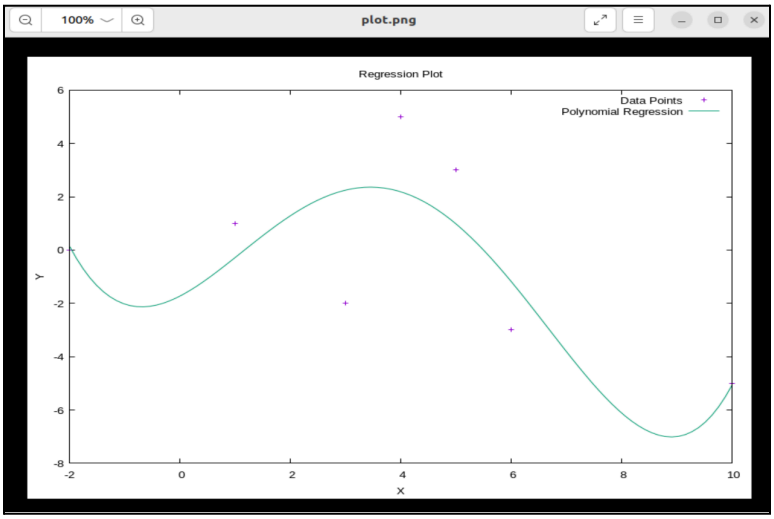

Figure 18: Example output graph for 4th degree polynomial regression generated by the regression command.

Polynomial fitting with degree = 6

```
sdemirtas@UbuntuSebnem: ~/Desktop/comp-304-operating-systems-project-1-sebuntu
sdemirtas@UbuntuSebnem:/home/sdemirtas/Desktop/comp-304-operating-systems-project-1-sebuntu Hshell$ regression input.txt -p 6
Data Points:
x         y
-2.00    0.00
1.00     1.00
3.00     -2.00
4.00     5.00
5.00     3.00
6.00     -3.00
10.00    -5.00

Polynomial Regression Coefficients:
Coefficient a0: 36.18
Coefficient a1: -43.34
Coefficient a2: 1.93
Coefficient a3: 8.86
Coefficient a4: -2.97
Coefficient a5: 0.35
Coefficient a6: -0.01
sdemirtas@UbuntuSebnem:/home/sdemirtas/Desktop/comp-304-operating-systems-project-1-sebuntu Hshell$
```

Figure 19: Output of polynomial regression with degree 6 using the regression command.

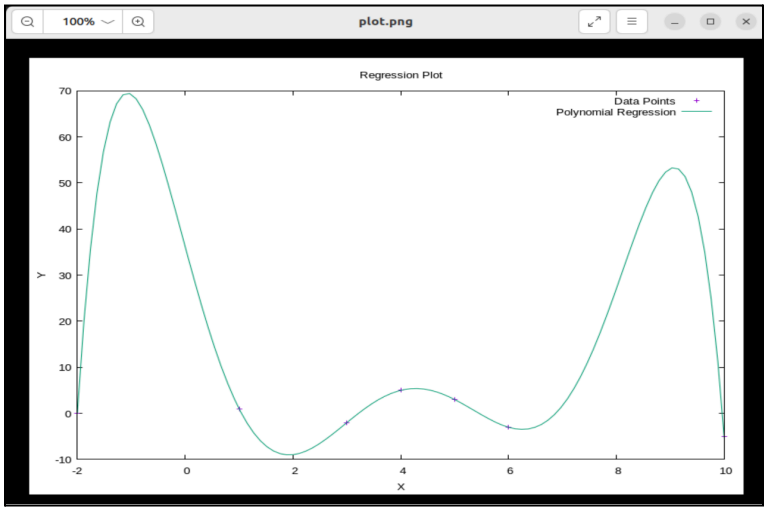

Figure 20: Example output graph for 4th degree polynomial regression generated by the regression command.

b. textify

We have written a command that would either share the asked statistic of a text file or make necessary modification (changing words) and write to a new text file.

textify <filename> <mode> [additional arguments]

Example usages are:

- textify textinput.txt -count_letters
- textify textinput.txt -count_words
- textify textinput.txt -count_specific_word search_word
- textify textinput.txt -change_words old_word new_word

-count_letters

```
if (strcmp(mode, "-count_letters")==0) {
    int count=0;
    int c;
    while ((c=fgetc(file)) != EOF) {
        if (isalpha(c)||isdigit(c)) count++;
    }
    printf("Number of letters in %s: %d\n", filename, count);
    fclose(file);
}
```

Figure 21: Code snippet showing the implementation of count_letters mode of "textify" method.

In this mode, with a loop until the end of file, each character is checked if they are a letter or digit. Then, number of letters (digits are also included) is shown in the command prompt.

-count_words

```
else if (strcmp(mode, "-count_words") == 0) {
    int count=0;
    char word[100]; //max word length given as 100
    while (fscanf(file, "%s", word)==1) {
        count++;
    }
    fclose(file);
    printf("Number of words in %s: %d\n", filename, count);
}
```

Figure 22: Code snippet showing the implementation of count_words mode of "textify" method.

In this mode, the number of words is calculated, again through a loop. Maximum length for a word is given as 100. fscanf, reads the file until encountering a white space character (%s indicates that), and increments count.

-count_specific_word search_word

```
else if (strcmp(mode, "-count_specific_word") == 0) {
    if (command->arg_count < 4) {
        printf("Do not forget to enter the word to look for as the third argument!\n");
        return;
    }

    const char *searched_word = command->args[3];
    int count=0;
    char word[100];
    while (fscanf(file, "%s", word)==1) {
        if (strcmp(word, searched_word)==0) count++;
    }
    fclose(file);
    printf("Number of occurrences of '%s' in %s: %d\n", searched_word, filename, count);
}
```

Figure 23: Code snippet showing the implementation of count_specific_word mode of "textify" method.

In this mode, the searched word should also be given as the argument. In the given file, the given word is searched and the number of occurrences is counted. Each word is compared with the strcmp function.

-change_words old_word new_word

```
else if (strcmp(mode, "-change_words") == 0) {
    if (command->arg_count < 5) {
        printf("Do not forget to write the word that will be changed as the 3th, word to change to 4th argument\n");
        return;
    }

    const char *old_word = command->args[3];
    const char *new_word = command->args[4];

    //To name the second file properly.
    const char *dot_position = strrchr(filename, '.');
    if (!dot_position) {
        fprintf(stderr, "Error: Invalid filename\n");
        fclose(file);
        return;
    }
    size_t filename_length = dot_position - filename;
    char updated_filename[filename_length + strlen("-updated") + strlen(".txt") + 1];
    strncpy(updated_filename, filename, filename_length); // Copy the filename without extension
    updated_filename[filename_length] = '\0'; // Null-terminate the string
    strcat(updated_filename, "-updated.txt"); // Append "-updated.txt"

    //Opening new file:
    FILE *updated_file = fopen(updated_filename, "w");
    if (!updated_file) {
        fprintf(stderr, "Error: Failed to create updated file\n");
        fclose(file);
        return;
    }

    char word[100];
    while (fscanf(file, "%s", word) == 1) {
        if (strcmp(word, old_word) == 0) {
            fprintf(updated_file, "%s ", new_word);
        }
        else {
            fprintf(updated_file, "%s ", word);
        }

        int next_char = fgetc(file);
        ungetc(next_char, file); // Put back the character for further processing
        if (next_char == '\n' || next_char == EOF) {
            fprintf(updated_file, "\n"); // Add a newline character if it is
        }
    }
    fclose(file);
    fclose(updated_file);
    printf("Occurrences of '%s' in %s changed to '%s' in %s\n", old_word, filename,  new_word,updated_filename);
    return;
}
```

Figure 24: Code snippet showing the implementation of change_words mode of "textify" method.

This mode takes 2 additional parameters, other than the txt file's name. These are the old_word and new_word. The function changes the ol_word with the new_word and puts in a new file called the "...-updated.txt"

First part of the function is to create a new file with the modified name.

Second part is where the old_word occurrences are detected. Until occurrence, words are written as they are to the new file. When detected, a new_word instead of the old_word is written.

Line ends are checked to ensure that the modified file's rows follow the rows of input's rows.

Part IV - Psvis Command

To execute the psvis operation, a Kernel module named psvis was initially created. This module traverses, collects, and outputs all processes connected to the process with the specified PID, using a depth first search (DFS) algorithm and by utilizing the task struct. It prints each child process with its PID and creation time based on its depth in the hierarchy to maintain the parent-child relationship. The printing of the processes has the order that is naturally calculated by the DFS algorithm. Then, we can inquire about the parent-child relationships based on this ordering. In our Hshell implementation, the psvis command activates the psvis.ko module by passing the PID through the insmod command for loading, and then removes the module using the rmmod command after it is completed.

Also note that gcc-12 is required to run the psvis command. (**installation: sudo apt install gcc-12**).
The command should be run with the following structure:

- psvis <PID> <filename>

Then the program will create “filename.txt” that depicts the tree in a text file, also “filename.png” that depicts the tree as an image using the gnuplot library.

For example, we use the tree generation function from the problem 1-A from the first assignment to generate a tree with the intended depth. If we generate a tree with depth four, a tree with the following structure will be generated:

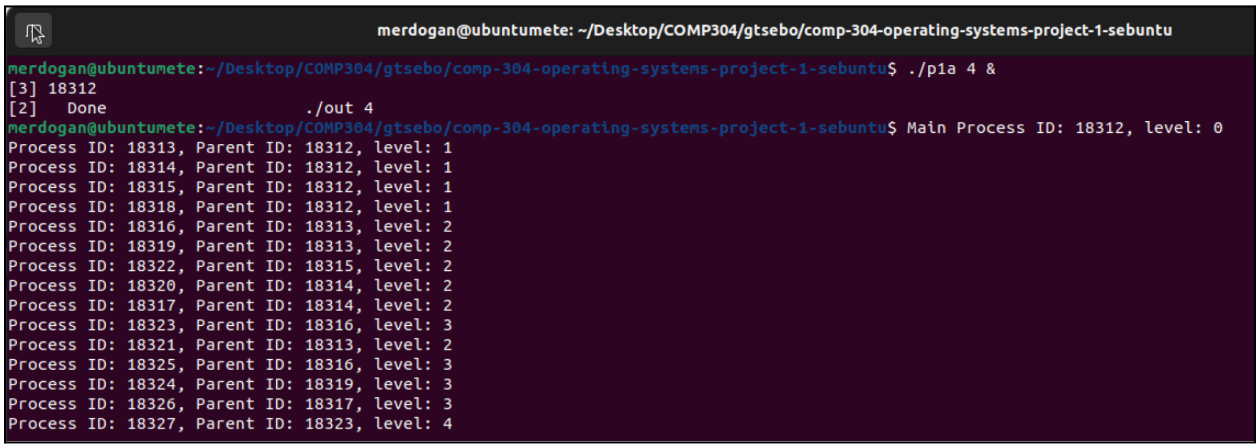

Figure 25: Running the process tree creating function from the first assignment problem 1-a, to test the psvis function.

Then if we run the psvis command as follows:

```
merdogan@ubuntumete:~/Desktop/COMP304/gtsebo/comp-304-operating-systems-project-1-sebuntu$ ./Hshell
merdogan@ubuntumete:/home/merdogan/Desktop/COMP304/gtsebo/comp-304-operating-systems-project-1-sebuntu Hshell$ psvis 18312 tree_out
[Paz May 12 14:56:56 2024] Loading the psvis module to the kernel.

[Paz May 12 14:56:56 2024] depth: 0, PID: 18312, Creation Time: 28100472662601 ns
[Paz May 12 14:56:56 2024] depth: 1, -PID: 18313, Creation Time: 28100474981275 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18316, Creation Time: 28100475255235 ns
[Paz May 12 14:56:56 2024] depth: 3, ---PID: 18323, Creation Time: 28100475568319 ns
[Paz May 12 14:56:56 2024] depth: 4, ----PID: 18327, Creation Time: 28100475758569 ns
[Paz May 12 14:56:56 2024] depth: 3, ---PID: 18325, Creation Time: 28100475675736 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18319, Creation Time: 28100475372068 ns
[Paz May 12 14:56:56 2024] depth: 3, ---PID: 18324, Creation Time: 28100475654569 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18321, Creation Time: 28100475475527 ns
[Paz May 12 14:56:56 2024] depth: 1, -PID: 18314, Creation Time: 28100475090234 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18317, Creation Time: 28100475336485 ns
[Paz May 12 14:56:56 2024] depth: 3, ---PID: 18326, Creation Time: 28100475727819 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18320, Creation Time: 28100475463860 ns
[Paz May 12 14:56:56 2024] depth: 1, -PID: 18315, Creation Time: 28100475229234 ns
[Paz May 12 14:56:56 2024] depth: 2, --PID: 18322, Creation Time: 28100475479610 ns
[Paz May 12 14:56:56 2024] depth: 1, -PID: 18318, Creation Time: 28100475335526 ns

[Paz May 12 14:56:56 2024] Removing the psvis module from the kernel.
merdogan@ubuntumete:/home/merdogan/Desktop/COMP304/gtsebo/comp-304-operating-systems-project-1-sebuntu Hshell$
```

Figure 26: Running the psvis command from the terminal and its output.

We also have the following text file result:

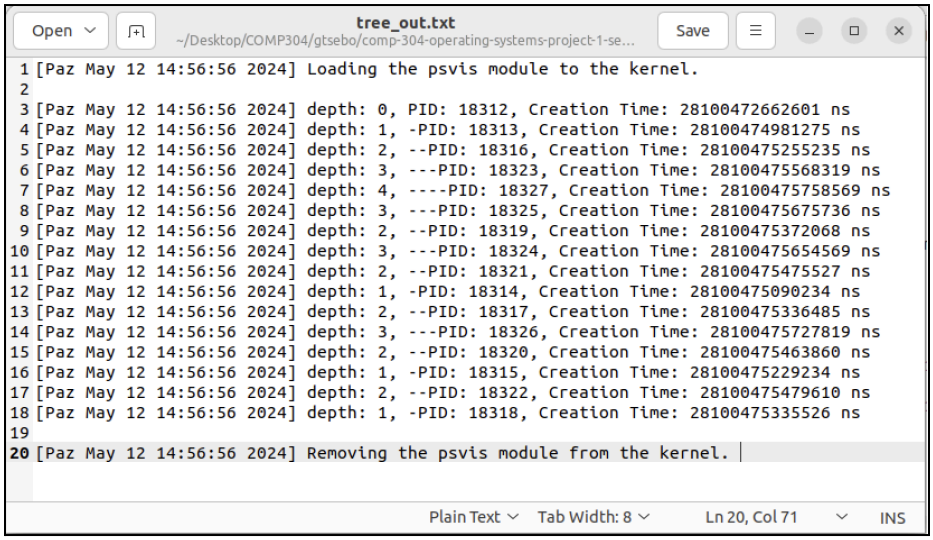

Figure 27: The process tree of a given root process. This is the text output of the psvis command, after running the "sudo dmesg -c -H" command. Then the output of the dmesg command is dumped into this text file using the program piping with the "tee" command.

Here we can see the increased depth of the nodes with the dashes ("-") and the process creation times in nanoseconds. Then, we have the "tree_plot.c" code that reads this text file and converts it into a tree data structure with the following struct:

```c
typedef struct {
        int pid;
        long long int creation_time;
        int depth;
        int children[MAX_DEPTH];
        int num_children;
} Node;
```

Figure 28: The struct for representing trees.

Then, using the gnuplot library, we print the tree as an image where each node is denoted with its PID and its creation time relative to the root process. Here, the root process is plotted with a red circle. Also the eldest childs of a parent is denoted with the green circle.

The code for reading the tree from the text file and generating the tree plot using the gnuplot library is implemented in the **tree_plot.c** file under the **tree** directory, which is imported by our main shell code.

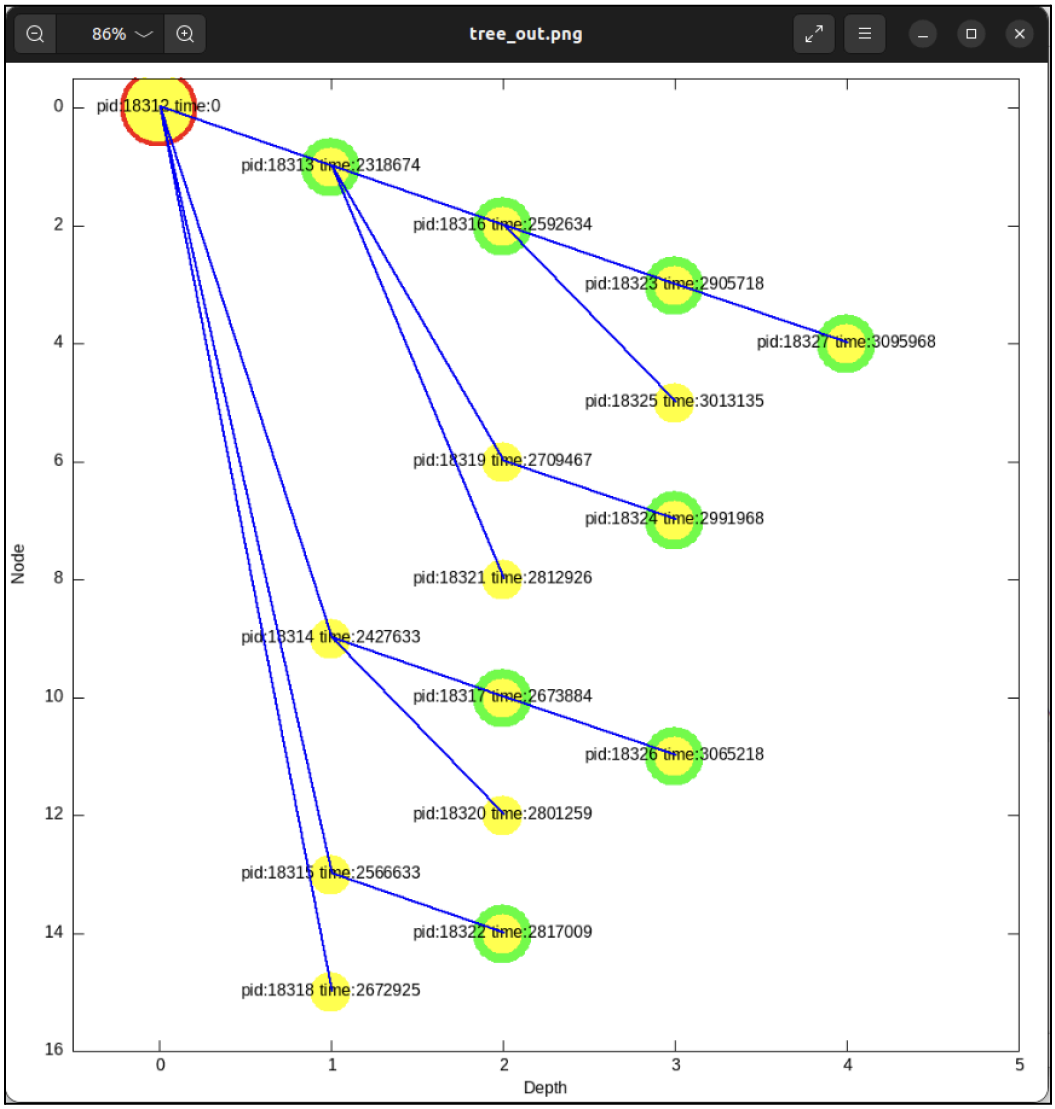

Figure 29: The process tree of a given root process. Each node represents a process and the connections between nodes denote the parent-child relationship. The eldest child of each parent has a green circle. The y-axis represents the nodes in the text printing order, and the x-axis denotes the increasing depth of the tree.

References

[1]. *Gnuplot*. gnuplot homepage. (n.d.). http://www.gnuplot.info/