CSE 344

Homework #1

200104004074

Bedirhan Ömer Aksoy

1) Introduction

Program compiles with “make” command, runs with “make run” command, and cleans the object and executable files with “make clean” command. Program takes inputs from the user in a while loop in the program and terminates with the “exit” command.

Program implements fork() in executeCommand() function, so that every command runs in a child process. (except the changeDir() function, because the main process’s working directory should be change for that.)

Program do not uses <stdio.h> library, instead of using printf() and fopen() for i/o and file operations, program uses read(), write(), open() and close() functions from <unistd.h> library.

Program initializes the log.txt files path at the beginning of the program, so that, if the working directory changes, the log file’s path stays the same without depending on the current working directory location.

Program tokenizes string with whitespaces by checking the string char by char and if tokenizer encounters a quote sign (“), it means it’s a text to append a file, it takes the rest of the input as the append string until encounter another quote sign to close the quote (for example: appendToFile example.txt “this is a test input”, token[0]: appendToFile, token[1]: example.txt, token[2]: this is a test input).

Program writes log events after every operation with using one function (writeLogEvent()) to reduce the same log write blocks with appending timestamp at the beginning of every log statement with using getTimestamp() function. Program writes output to console with writeStdout() function to not to use write() statement directly inside of the functions.

Program implements changeDir() function to change the working directory of the program. Program writes log events to the main.c files existing directory even the working directory changes.

2) Code Explanation

2.1) is_space(char c), checks whether the current character is a whitespace or not for tokenizer.

```
int is_space(char c) {
    return (c == ' ' || c == '\t' || c == '\n' || c == '\r');
}
```

2.2) initializeLogFilePath(), initializes the log file's path when the program starts and called in main() function once when program starts, so the log file's path will be the same even the working directory changes.

```
void initializeLogFilePath() {
    if (getcwd(logFilePath, sizeof(logFilePath)) != NULL) {
        strcat(logFilePath, "/log.txt");  // Append /log.txt to current directory
    } else {
        write(2, "Error getting current directory\n", 32);  // Print error to stderr
    }
}
```

2.3) tokenizeString(const char* str, int* tokenCount), takes user input as string and tokenCount integer as pointer to use the count later, function checks input char by char and skips the leading whitespaces, takes the chars as a token until encountering a new whitespace or a quote sign. If a quote sign encounters (“), the program takes the rest of the string as a single token without checking for whitespaces until the quoting sign encounters to close the quote (“). Then returns the tokens in the tokens array and the count of the tokens are assigned to the tokenCount pointer.

```
char** tokenizeString(const char* str, int* tokenCount) {
    int length = strlen(str);
    int maxTokens = 10;                                     // Initial size for the token array
    int tokenIndex = 0;
    char** tokens = (char**)malloc(maxTokens * sizeof(char*));

    int i = 0;
    while (i < length) {
        while (i < length && is_space(str[i])) {            // Skip leading spaces
            i++;
        }
        if (i >= length) {                                  // If we reach the end of the string, break
            break;
        }
        if (str[i] == '"') {                                // If it's a quote, start a quoted token
            i++;                                            // Skip the opening quote
            int start = i;
            while (i < length && str[i] != '"') {           // Find the closing quote
                i++;
            }
            if (i < length) {                               // If we found the closing quote
                int tokenLength = i - start;
                tokens[tokenIndex] = (char*)malloc((tokenLength + 1) * sizeof(char));
                strncpy(tokens[tokenIndex], str + start, tokenLength);
                tokens[tokenIndex][tokenLength] = '\0';     // Null-terminate the string
                tokenIndex++;
                i++;                                        // Skip the closing quote
            }
        }
        else {                                              // If it's not a quote, start a regular token
            int start = i;
            while (i < length && !is_space(str[i]) && str[i] != '"') {
                i++;
            }
            int tokenLength = i - start;
            if (tokenIndex >= maxTokens) {
                maxTokens *= 2;                             // Double the size of the token array
                tokens = (char**)realloc(tokens, maxTokens * sizeof(char*));
            }
            tokens[tokenIndex] = (char*)malloc((tokenLength + 1) * sizeof(char));
            strncpy(tokens[tokenIndex], str + start, tokenLength);
            tokens[tokenIndex][tokenLength] = '\0';         // Null-terminate the string
            tokenIndex++;
        }
    }
    *tokenCount = tokenIndex;
    return tokens;
}
```

2.4) getTimeStamp(char* buffer, size_t buffer_size), creates a timestamp for log events in the given format in the homework document.

```
79
80  void getTimestamp(char* buffer, size_t buffer_size) {
81      // Get the current time
82      time_t now = time(NULL);
83
84      // Check if time retrieval was successful
85      if (now == (time_t)(-1)) {
86          write(2, "Error getting time\n", 19);  // Print error to stderr
87          return;
88      }
89
90      struct tm *tm_info;
91      tm_info = localtime(&now);  // Convert time_t to tm struct
92
93      // Format the time as [YYYY-MM-DD HH:MM:SS]
94      strftime(buffer, buffer_size, "[%Y-%m-%d %H:%M:%S]", tm_info);
95  }
96
```

2.5) writeLogEvent(const char* message), resets the globalLogBuffer to clean, gets timestamp with calling getTimestamp() function, writes timestamp to log buffer, appends a whitespace after time stamp “ “, appends message that sent from the caller function to write to log file, then writes the log buffer to log file.

```
97  void writeLogEvent(const char *message) {
98      memset(globalLogBuffer, 0, BUFFER_SIZE);
99
100     char timestamp[32];
101     getTimestamp(timestamp, sizeof(timestamp));
102
103     // Copy timestamp into the globalOutputBuffer
104     size_t timestamp_len = strlen(timestamp);
105     if (timestamp_len < BUFFER_SIZE) {
106         strncpy(globalLogBuffer, timestamp, timestamp_len);
107     }
108
109     // Append a space after the timestamp
110     size_t buffer_len = strlen(globalLogBuffer);
111     if (buffer_len + 1 < BUFFER_SIZE) {
112         strcat(globalLogBuffer, " ");  // Add space
113     }
114
115     // Append the log message to the buffer
116     size_t message_len = strlen(message);
117     if (buffer_len + message_len + 1 < BUFFER_SIZE) {
118         strncat(globalLogBuffer, message, message_len);
119     }
120
121     // Append newline
122     buffer_len = strlen(globalLogBuffer);
123     if (buffer_len + 1 < BUFFER_SIZE) {
124         strcat(globalLogBuffer, "\n");
125     }
126
127     // Open the log file at the dynamically determined location
128     int fd = open(logFilePath, O_WRONLY | O_CREAT | O_APPEND, 0644);
129     if (fd < 0) return;  // Fail silently if unable to open log file
130
131     write(fd, globalLogBuffer, strlen(globalLogBuffer));
132     close(fd);
133 }
```

2.6) createFile(char* filename), function checks if the file exists and if it exists prints “file ‘filename’ already exists” message, if not, creates file and prints “file ‘filename’ created successfully” message. Clears the output buffer before and after the function to prevent overwrite cases.

```
138  void createFile(char* filename) {
139      // Clear global buffer before use
140      memset(globalOutputBuffer, 0, BUFFER_SIZE);
141
142      if (access(filename, F_OK) == 0) {
143          strcpy(globalOutputBuffer, "File ");
144          strcat(globalOutputBuffer, filename);
145          strcat(globalOutputBuffer, " already exists.\n");
146      }
147      else {
148          int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0644);
149          if (fd != -1) {
150              close(fd);
151              strcpy(globalOutputBuffer, "File ");
152              strcat(globalOutputBuffer, filename);
153              strcat(globalOutputBuffer, " created successfully.\n");
154          }
155          else {
156              strcpy(globalOutputBuffer, "Failed to create file ");
157              strcat(globalOutputBuffer, filename);
158              strcat(globalOutputBuffer, ".\n");
159          }
160      }
161
162      writeStdout(1, globalOutputBuffer);
163      writeLogEvent(globalOutputBuffer);
164
165      memset(globalOutputBuffer, 0, BUFFER_SIZE);
166  }
167
```

2.7) createDir(char* directoryName), function checks if the directory exists and if it exists, prints a warning message, if not, creates a new directory and prints success message and if directory could not be created, prints related error message. Clears output buffer at the beginning and the end of the function.

```
168  void createDir(char* directoryName) {
169      struct stat stat_buf;
170
171      // Clear global buffer before use
172      memset(globalOutputBuffer, 0, BUFFER_SIZE);
173
174      // Check if the directory exists and is a directory
175      if (stat(directoryName, &stat_buf) == 0 && S_ISDIR(stat_buf.st_mode)) {
176          strcpy(globalOutputBuffer, "Directory ");
177          strcat(globalOutputBuffer, directoryName);
178          strcat(globalOutputBuffer, " already exists.\n");
179      }
180      else {
181          if (mkdir(directoryName, 0777) == 0) {
182              strcpy(globalOutputBuffer, "Directory ");
183              strcat(globalOutputBuffer, directoryName);
184              strcat(globalOutputBuffer, " created successfully.\n");
185          }
186          else {
187              strcpy(globalOutputBuffer, "Error creating directory ");
188              strcat(globalOutputBuffer, directoryName);
189              strcat(globalOutputBuffer, ".\n");
190          }
191      }
192
193      writeStdout(1, globalOutputBuffer);
194      writeLogEvent(globalOutputBuffer);
195
196      memset(globalOutputBuffer, 0, BUFFER_SIZE);
197  }
```

2.8) listDir(char* directoryName), clears output buffer, checks if the directory exists, if exists creates a dirent struct to make operations on the directory object, reads directory containings, skips the “.” and “..” file names to list only the visible files and directories in current directory, closes directory and writes to log and stdout, and clears the output buffer.

```
199  void listDir(char* directoryName) {
200      DIR *dir = opendir(directoryName);
201
202      // Clear global buffer before use
203      memset(globalOutputBuffer, 0, BUFFER_SIZE);
204
205      if (dir == NULL) {
206          strcpy(globalOutputBuffer, "Directory ");
207          strcat(globalOutputBuffer, directoryName);
208          strcat(globalOutputBuffer, " not found.\n");
209      }
210      else {
211          struct dirent *entry;
212
213          // Start listing the files
214          strcpy(globalOutputBuffer, "Files in directory '");
215          strcat(globalOutputBuffer, directoryName);
216          strcat(globalOutputBuffer, "':\n");
217
218          while ((entry = readdir(dir)) != NULL) {
219              // Skip the "." and ".." entries
220              if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
221                  strcat(globalOutputBuffer, entry->d_name);
222                  strcat(globalOutputBuffer, "\n");
223              }
224          }
225
226          // Close the directory
227          closedir(dir);
228      }
229
230      writeStdout(1, globalOutputBuffer);
231      writeLogEvent(globalOutputBuffer);
232
233      memset(globalOutputBuffer, 0, BUFFER_SIZE);
234  }
235
```

2.9) hasExtension(const char* filename, const char* extension), takes filename and extension strings as input, compares the last strlen(extension) chars with the extension string to check whether the filename has the provided extension. If filename's extension is same with the provided extension, returns 1, if not, returns 0.

```
236
237 int has_extension(const char *filename, const char *extension) {
238     size_t len_filename = strlen(filename);
239     size_t len_extension = strlen(extension);
240
241     if (len_filename >= len_extension && strcmp(filename + (len_filename - len_extension), extension) == 0) {
242         return 1;           // Match found
243     }
244     return 0;
245 }
246
```

2.10) listFilesByExtension(char* directoryName, char* extension), clears output buffer, checks if the directory exists, if exists creates a dirent struct to make operations on the directory object, reads directory containings, skips the "." and ".." file names to list only the visible files and directories in current directory, checks if the current readed file has the same extension with using has_extension() function, if it has, writes that file to output buffer and adds a new line "\n" to end of it, and continues until the end of the directory, then closes directory and writes to log and stdout, and clears the output buffer.

```c
void listFilesByExtension(char* directoryName, char* extension) {
    DIR *dir = opendir(directoryName);
    int isFound = 0;

    // Clear global buffer before use
    memset(globalOutputBuffer, 0, BUFFER_SIZE);

    if (dir == NULL) {
        strcpy(globalOutputBuffer, "Directory ");
        strcat(globalOutputBuffer, directoryName);
        strcat(globalOutputBuffer, " not found.\n");
    }
    else {
        struct dirent *entry;

        // Start listing the files
        strcpy(globalOutputBuffer, "Files in directory '");
        strcat(globalOutputBuffer, directoryName);
        strcat(globalOutputBuffer, "' with extension '");
        strcat(globalOutputBuffer, extension);
        strcat(globalOutputBuffer, "':\n");

        while ((entry = readdir(dir)) != NULL) {
            // Skip the "." and ".." entries
            if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
                if (has_extension(entry->d_name, extension)) {
                    strcat(globalOutputBuffer, entry->d_name);
                    strcat(globalOutputBuffer, "\n");
                    isFound = 1;
                }
            }
        }

        if (!isFound) {
            strcpy(globalOutputBuffer, "No files with extension '");
            strcat(globalOutputBuffer, extension);
            strcat(globalOutputBuffer, "' found in ");
            strcat(globalOutputBuffer, directoryName);
            strcat(globalOutputBuffer, ".\n");
        }

        // Close the directory
        closedir(dir);
    }

    writeStdout(1, globalOutputBuffer);
    writeLogEvent(globalOutputBuffer);

    memset(globalOutputBuffer, 0, BUFFER_SIZE);
}
```

2.11) changeDirectory(char* newDirectoryname), function first checks if the newDirectoryname exists, if exists, changes current working directory with using chdir() function and prints success message, if not exists or could not changes the directory, prints the related error message.

```
298  void changeDirectory(char* newDirectoryName) {
299      char currentWorkingDirectory[100];
300
301      // Clear global buffer before use
302      memset(globalOutputBuffer, 0, BUFFER_SIZE);
303
304      if (getcwd(currentWorkingDirectory, sizeof(currentWorkingDirectory)) != NULL) {
305          if (chdir(newDirectoryName) == 0) {
306              strcpy(globalOutputBuffer, "Successfully changed directory to: ");
307              strcat(globalOutputBuffer, newDirectoryName);
308              strcat(globalOutputBuffer, "\n");
309          }
310          else {
311              strcpy(globalOutputBuffer, "Couldn't change the directory to ");
312              strcat(globalOutputBuffer, newDirectoryName);
313              strcat(globalOutputBuffer, ".\n");
314          }
315      }
316      else {
317          strcpy(globalOutputBuffer, "Couldn't get current working directory or change the directory to ");
318          strcat(globalOutputBuffer, newDirectoryName);
319          strcat(globalOutputBuffer, ".\n");
320      }
321
322      writeStdout(1, globalOutputBuffer);
323      writeLogEvent(globalOutputBuffer);
324
325      memset(globalOutputBuffer, 0, BUFFER_SIZE);
326  }
327
```

2.12) readFile(char* filename), function clears output buffer before using, opens file, reads file with read function into buffer, copied to output buffer, and after reading, prints the file containings, if file not exists or could not be opened, prints error message, clears the output buffer at the end.

```
328  void readFile(char* filename) {
329      int fd;
330      char buffer[BUFFER_SIZE];
331      ssize_t bytesRead;
332
333      // Clear global buffer before use
334      memset(globalOutputBuffer, 0, BUFFER_SIZE);
335
336      fd = open(filename, O_RDONLY);
337
338      if (fd == -1) {
339          strcpy(globalOutputBuffer, "Error opening file ");
340          strcat(globalOutputBuffer, filename);
341          strcat(globalOutputBuffer, "\n");
342      }
343      else {
344          // Read from the file and store in globalOutputBuffer
345          while ((bytesRead = read(fd, buffer, BUFFER_SIZE)) > 0) {
346              strncat(globalOutputBuffer, buffer, bytesRead);
347          }
348
349          strcat(globalOutputBuffer, "\n");
350          // Close the file
351          close(fd);
352      }
353
354      writeStdout(1, globalOutputBuffer);
355      writeLogEvent(globalOutputBuffer);
356
357      memset(globalOutputBuffer, 0, BUFFER_SIZE);
358  }
359
```

2.13) appendToFile(char* filename, char* dataToAppend), function opens the file, clears the output buffer, if file could not be opened, prints message, else, locks the file with flock() function with LOCK_EX flag to block other operations to reach the file, writes the data to append to file, writes success message to output and log file, clears the buffer.

```
360  void appendToFile(char* filename, char* dataToAppend) {
361      int fd = open(filename, O_WRONLY | O_APPEND | O_CREAT, 0644);
362
363      // Clear global buffer before use
364      memset(globalOutputBuffer, 0, BUFFER_SIZE);
365
366      if (fd == -1) {
367          strcpy(globalOutputBuffer, "Cannot write to ");
368          strcat(globalOutputBuffer, filename);
369          strcat(globalOutputBuffer, ". File is locked or read-only.\n");
370      }
371      else {
372          // Lock the file (Exclusive Lock - blocks others)
373          if (flock(fd, LOCK_EX) == -1) {
374              strcpy(globalOutputBuffer, "Error locking file ");
375              strcat(globalOutputBuffer, filename);
376              strcat(globalOutputBuffer, "\n");
377              close(fd);
378          }
379          else {
380              // Write data to file
381              write(fd, dataToAppend, strlen(dataToAppend));
382
383              // Unlock the file
384              flock(fd, LOCK_UN);
385
386              strcpy(globalOutputBuffer, "Data successfully appended to ");
387              strcat(globalOutputBuffer, filename);
388              strcat(globalOutputBuffer, ".\n");
389
390              close(fd);
391          }
392      }
393
394      writeStdout(1, globalOutputBuffer);
395      writeLogEvent(globalOutputBuffer);
396      memset(globalOutputBuffer, 0, BUFFER_SIZE);
397  }
```

2.14) deleteFile(char* filename), function clears the output buffer, checks if file exists, if exists, unlinks the filename for delete operation as the course assistant suggests us to use, if unlink() successful, or not successful, or the file does not exists, prints related message to stdout with using output buffer, and logs the result message, and clears the output buffer again.

```
399  void deleteFile(char* filename) {
400      // Clear global buffer before use
401      memset(globalOutputBuffer, 0, BUFFER_SIZE);
402
403      if (access(filename, F_OK) == 0) {
404          if (unlink(filename) == 0) {
405              strcpy(globalOutputBuffer, "File ");
406              strcat(globalOutputBuffer, filename);
407              strcat(globalOutputBuffer, " deleted successfully.\n");
408          }
409          else {
410              strcpy(globalOutputBuffer, "Failed to delete file ");
411              strcat(globalOutputBuffer, filename);
412              strcat(globalOutputBuffer, "...\n");
413          }
414      }
415      else {
416          strcpy(globalOutputBuffer, "File ");
417          strcat(globalOutputBuffer, filename);
418          strcat(globalOutputBuffer, " not found.\n");
419      }
420
421      writeStdout(1, globalOutputBuffer);
422      writeLogEvent(globalOutputBuffer);
423
424      memset(globalOutputBuffer, 0, BUFFER_SIZE);
425  }
426
```

2.15) deleteDir, function checks if the directory exists, if exists, removes directory with rmdir() function, if this function cannot remove directory, it means directory is not empty, if its empty, directory removes, related output message will be written to the output buffer, output buffer will be printed to stdout and log file, then function clears the output buffer at the end.

```
427  void deleteDir(char* directoryName){
428      struct stat stat_buf;
429
430      // Clear global buffer before use
431      memset(globalOutputBuffer, 0, BUFFER_SIZE);
432
433      // check if the directory exists
434      if (stat(directoryName, &stat_buf) == 0 && S_ISDIR(stat_buf.st_mode)) {
435          // if directory exists, try to remove the directory
436          if (rmdir(directoryName) == 0) {
437              strcpy(globalOutputBuffer, "Directory ");
438              strcat(globalOutputBuffer, directoryName);
439              strcat(globalOutputBuffer, " deleted successfully.\n\n");
440          }
441          // if rmdir fails, it means that directory is not empty
442          else{
443              strcpy(globalOutputBuffer, "Directory ");
444              strcat(globalOutputBuffer, directoryName);
445              strcat(globalOutputBuffer, " is not empty.\n");
446          }
447      }
448      else{
449          strcpy(globalOutputBuffer, "No such directory called ");
450          strcat(globalOutputBuffer, directoryName);
451          strcat(globalOutputBuffer, " has been found...\n\n");
452      }
453
454      writeStdout(1, globalOutputBuffer);
455      writeLogEvent(globalOutputBuffer);
456
457      memset(globalOutputBuffer, 0, BUFFER_SIZE);
458  }
```

2.16) executeCommand(char** commandTokens, int tokenCount), program implements fork() in this function with checking the command token's type for executing operations from user in a child process, function first checks if the command is change directory command before forking the program, because changing directory must be executed in the parent process, not in child process to keep program in the changed directory, if the command is not changeDir command, function forks the program, checks the command type, calls the appropriate function to execute operation that user demands, after executing the related function, child process exits the program, parent process waits for the child process to exit, after waiting, free's the command tokens and the command token pointers itself.

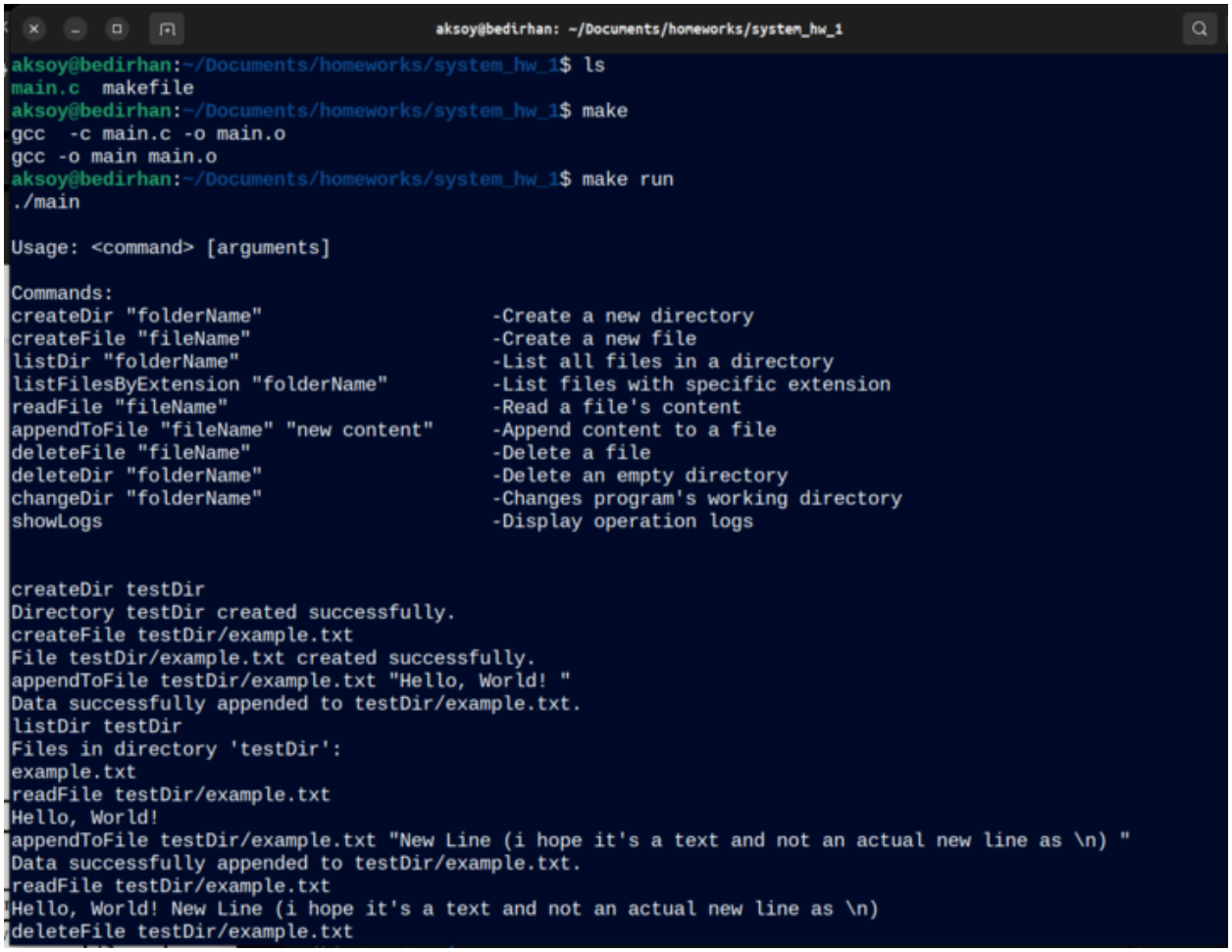
```c
void executeCommand(char** commandTokens, int tokenCount){
    if(strcmp(commandTokens[0], "changeDir") == 0){
        changeDirectory(commandTokens[1]);
    }
    else{
        pid_t pid = fork();

        if (pid<0){
            memset(globalOutputBuffer, 0, BUFFER_SIZE);
            strcpy(globalOutputBuffer, "Fork failed...\n");
            writeStdout(1,globalOutputBuffer);
            memset(globalOutputBuffer, 0, BUFFER_SIZE);
            exit(0);
        }
        else if(pid == 0){
            // child process

            if(strcmp(commandTokens[0], "createFile") == 0){
                createFile(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "createDir") == 0){
                createDir(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "listDir") == 0){
                listDir(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "listFilesByExtension") == 0){
                listFilesByExtension(commandTokens[1], commandTokens[2]);
            }
            else if(strcmp(commandTokens[0], "readFile") == 0){
                readFile(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "appendToFile") == 0){
                appendToFile(commandTokens[1], commandTokens[2]);
            }
            else if(strcmp(commandTokens[0], "deleteFile") == 0){
                deleteFile(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "deleteDir") == 0){
                deleteDir(commandTokens[1]);
            }
            else if(strcmp(commandTokens[0], "showLogs") == 0){
                readFile(logFilePath);
            }
            exit(0);
        }
        else{                                                   // parent process
            waitpid(pid, NULL, 0);                              // wait for child to complete
        }

        // free the each command token char pointers that user entered
        for(int i=0; i<tokenCount; i++){
            free(commandTokens[i]);
        }
        // free the char pointer pointer
        free(commandTokens);
    }
}
```

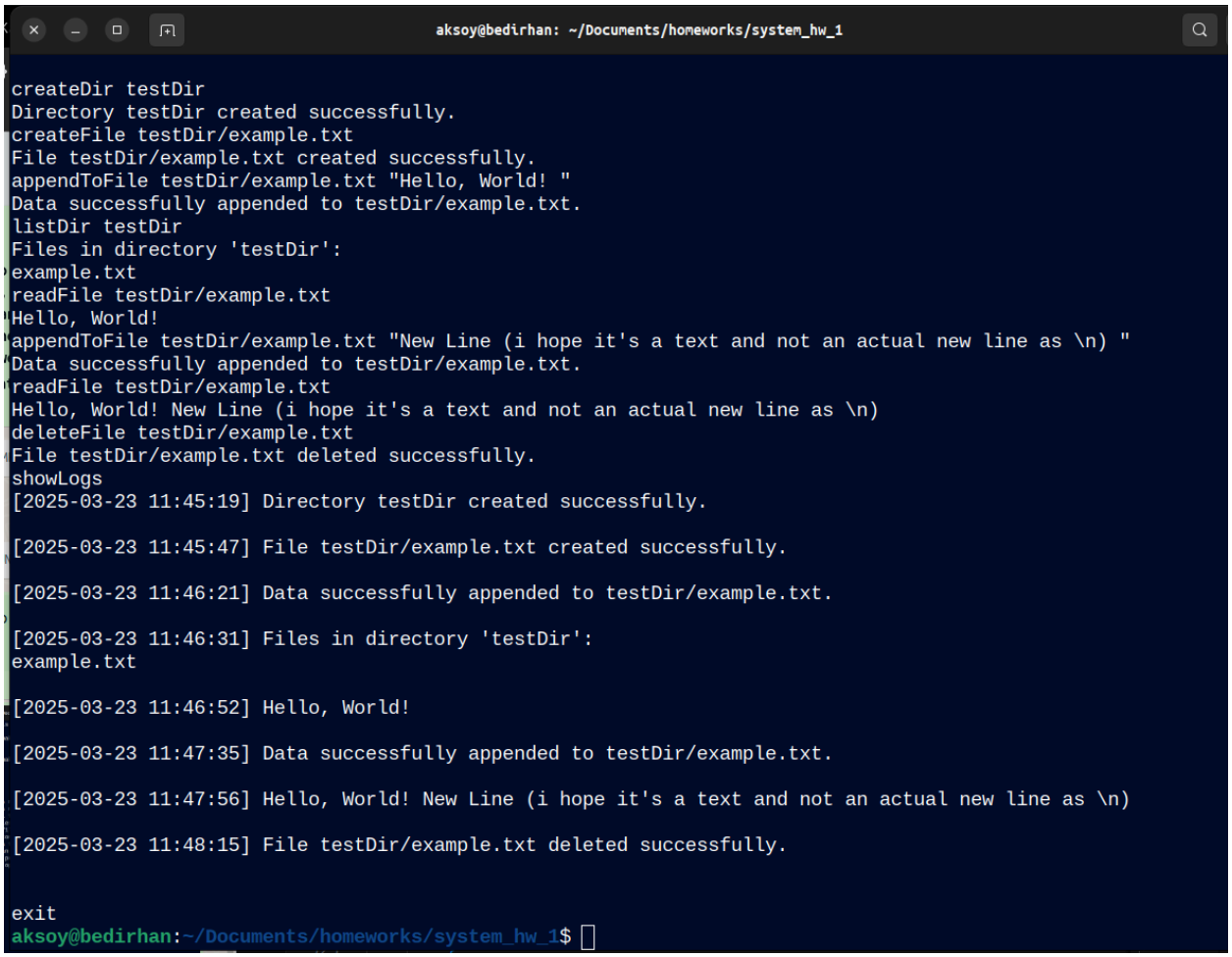
2.17) main(), main function declares input array, timestamp array, inputTokens double pointer and input token count integer, prints program manual and enters an infinite loop to take inputs from user until user enters “exit” command.

```c
int main(){

    char input[BUFFER_SIZE];                                        // The user input command
    ssize_t bytesRead;
    char timestamp[TIMESTAMP_SIZE];                                 // Buffer for the timestamp string
    char** inputTokens;                                             // Tokens of the user input
    int inputTokenCount = 0;                                        // The count of the tokens to operate with its indexes

    writeStdout(1, "\nUsage: <command> [arguments]\n\n");
    writeStdout(1, "Commands:\n");
    writeStdout(1, "createDir \"folderName\"                    -Create a new directory\n");
    writeStdout(1, "createFile \"fileName\"                     -Create a new file\n");
    writeStdout(1, "listDir \"folderName\"                      -List all files in a directory\n");
    writeStdout(1, "listFilesByExtension \"folderName\"         -List files with specific extension\n");
    writeStdout(1, "readFile \"fileName\"                       -Read a file's content\n");
    writeStdout(1, "appendToFile \"fileName\" \"new content\"     -Append content to a file\n");
    writeStdout(1, "deleteFile \"fileName\"                     -Delete a file\n");
    writeStdout(1, "deleteDir \"folderName\"                    -Delete an empty directory\n");
    writeStdout(1, "changeDir \"folderName\"                    -Changes program's working directory\n");
    writeStdout(1, "showLogs                                  -Display operation logs\n\n\n");

    while(1){
        initializeLogFilePath();

        memset(input, 0, BUFFER_SIZE);
        bytesRead = read(0, input, BUFFER_SIZE - 1);  // 0 = stdin
        input[bytesRead] = '\0';  // Null-terminate
        inputTokens = tokenizeString(input, &inputTokenCount);              // Tokenize the user input
        if(strcmp(inputTokens[0],"exit") == 0) return 0;
        executeCommand(inputTokens, inputTokenCount);
    }

    return 0;
}
```

3) Screenshots

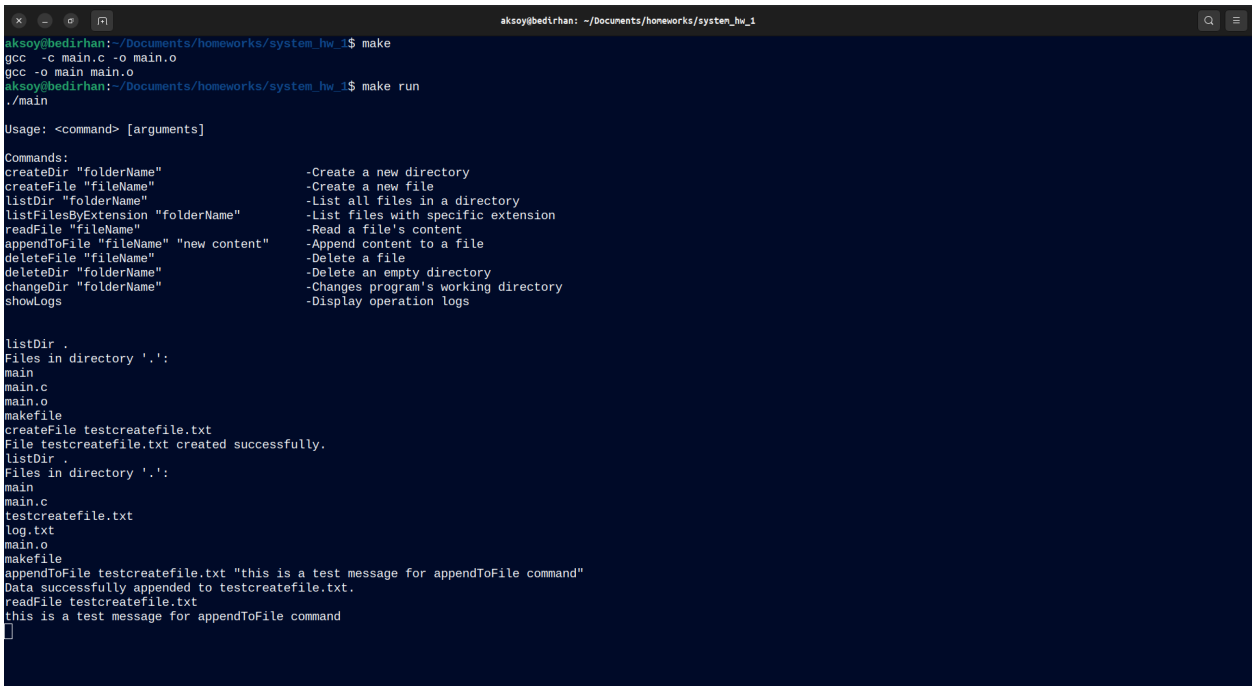
The test scenario given in pdf:



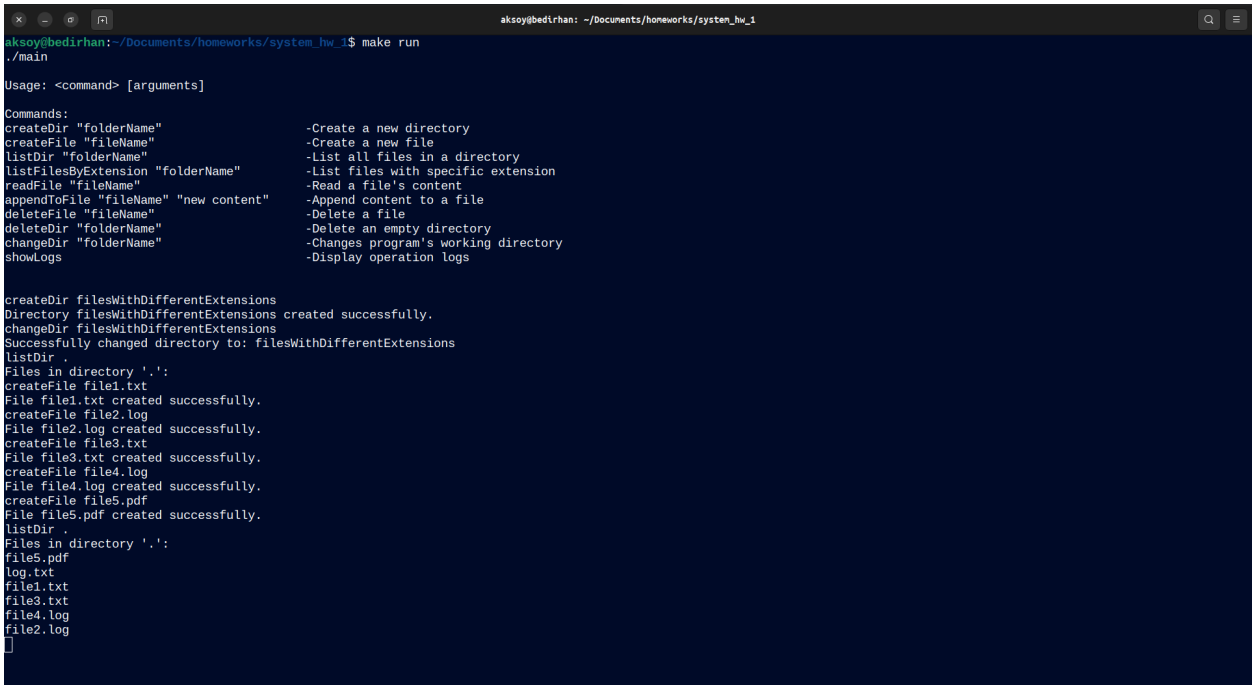

aksoy@bedirhan: ~/Documents/homeworks/system_hw_1
createDir testDir
Directory testDir created successfully.
createFile testDir/example.txt
File testDir/example.txt created successfully.
appendToFile testDir/example.txt "Hello, World! "
Data successfully appended to testDir/example.txt.
listDir testDir
Files in directory 'testDir':
example.txt
readFile testDir/example.txt
Hello, World!
appendToFile testDir/example.txt "New Line (i hope it's a text and not an actual new line as \n) "
Data successfully appended to testDir/example.txt.
readFile testDir/example.txt
Hello, World! New Line (i hope it's a text and not an actual new line as \n)
deleteFile testDir/example.txt
File testDir/example.txt deleted successfully.
showLogs
[2025-03-23 11:45:19] Directory testDir created successfully.
[2025-03-23 11:45:47] File testDir/example.txt created successfully.
[2025-03-23 11:46:21] Data successfully appended to testDir/example.txt.
[2025-03-23 11:46:31] Files in directory 'testDir':
example.txt
[2025-03-23 11:46:52] Hello, World!
[2025-03-23 11:47:35] Data successfully appended to testDir/example.txt.
[2025-03-23 11:47:56] Hello, World! New Line (i hope it's a text and not an actual new line as \n)
[2025-03-23 11:48:15] File testDir/example.txt deleted successfully.
exit
aksoy@bedirhan:~/Documents/homeworks/system_hw_1$

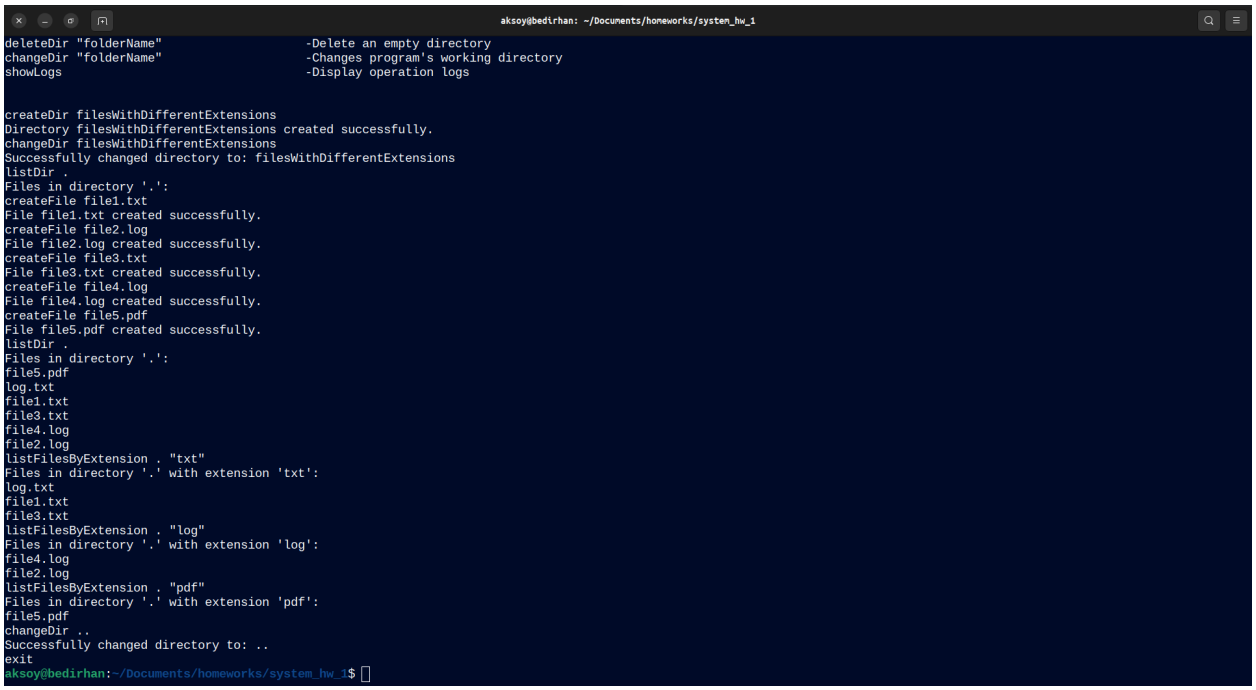
Test cases for each function:

listDir, createFile, appendToFile, readFile

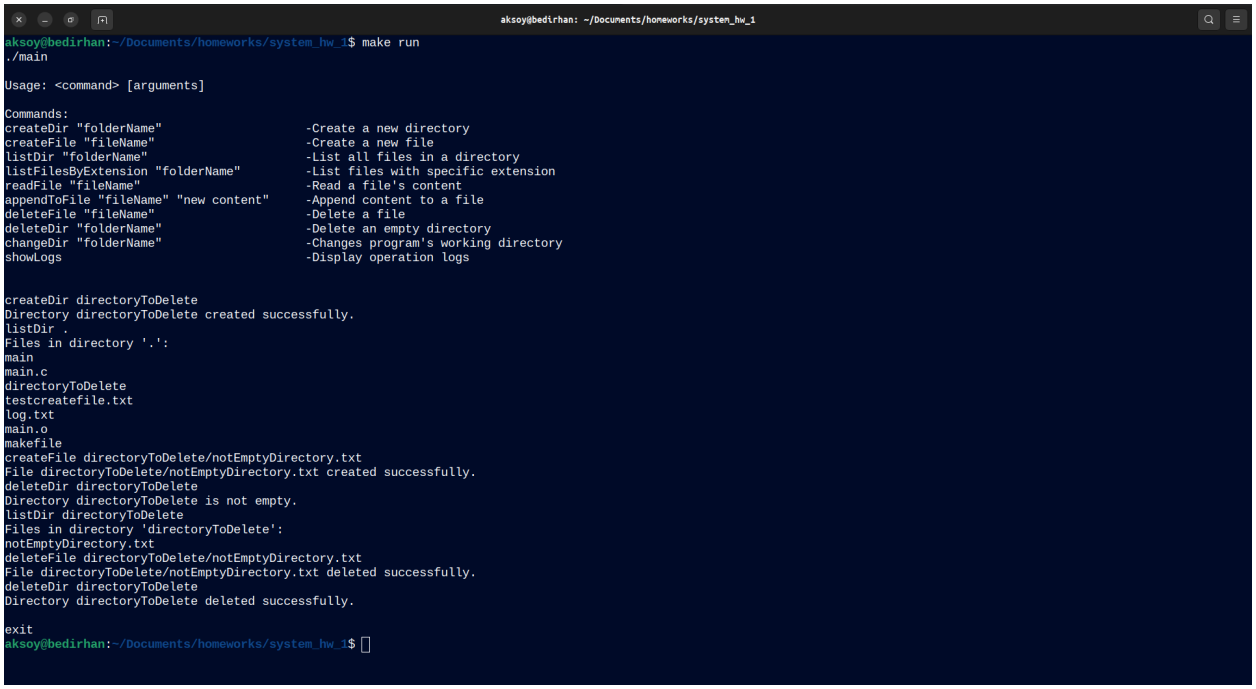


listFilesByExtension, changeDir

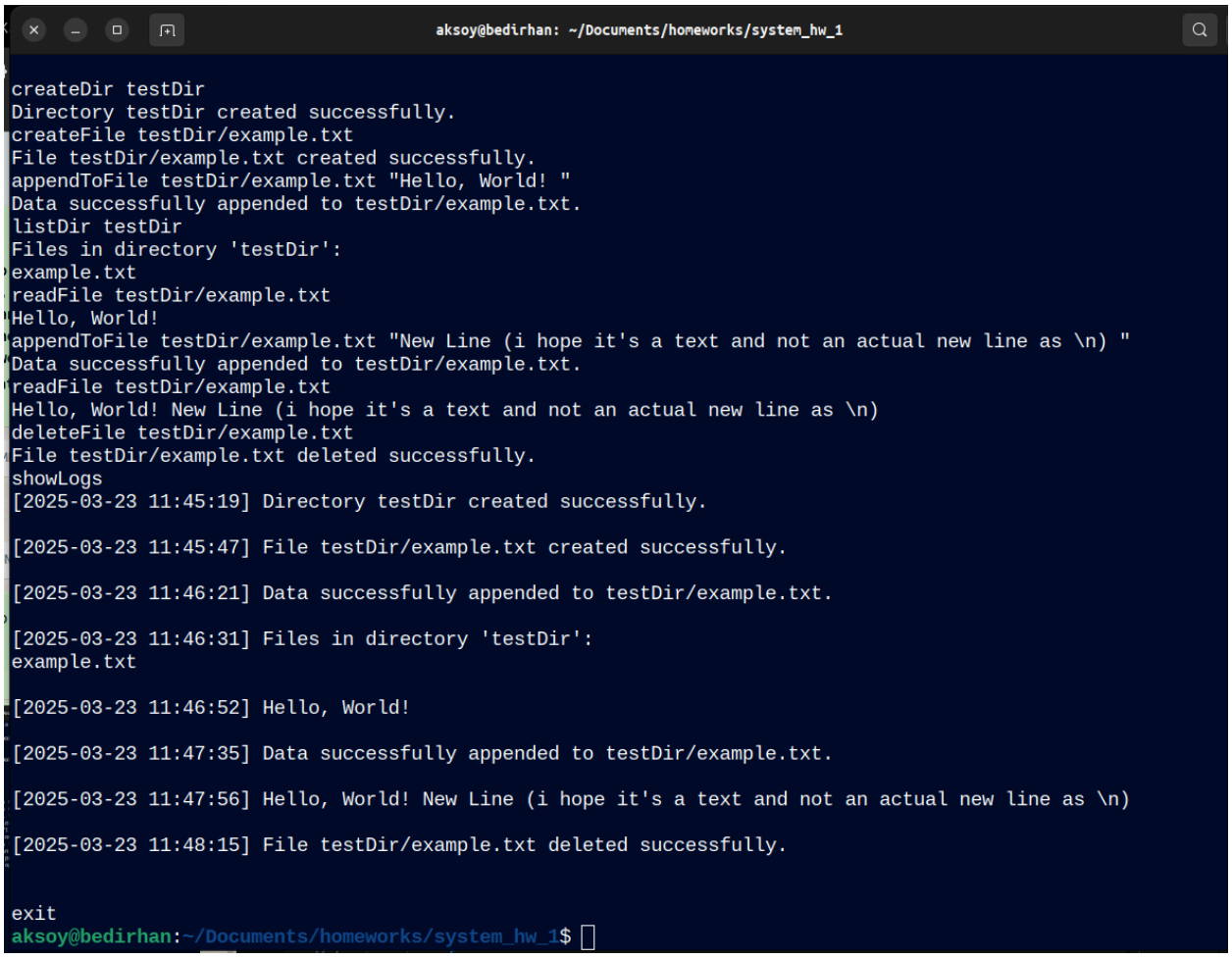




createDir, deleteDir, deleteFile, listDir



showLogs:



4) Conclusion

The only challenge was the file creation inside a folder, I just cannot understand the requirement from the pdf about how to implement it because it was not explicitly explained, then I have implemented changeDir function to solve this problem.