```c
// Name - Manjinder Singh
// Section - 4
// Course Code and Subject Name - COMP8567-4-R-2023S|Advanced Systems Programming
// Student ID - 110097177

/*
References -
1. Class Code Files shared by DOC during lecture.
2. Illustrations shared on Brightspace for Assignment 1
3. Assignement PDF for Problem Statement.
4. NFTW Library Documentation. (https://linux.die.net/man/3/nftw)
*/

/* Steps to execute
First we will execute on Terminal: gcc -o ncpmvdir ncpmvdir.c
Then we executed the below illustartions one by one:

./ncpmvdir ~sample1 ~/sample2 -cp
./ncpmvdir ~sample1 ~/sample2 -mv

./ncpmvdir ~sample1 ~/sample2 -cp pdf
./ncpmvdir ~sample1 ~/sample2 -mv java

./ncpmvdir ~sample1 ~/sample4/sample5 -cp
./ncpmvdir ~sample1 ~/sample4/sample5 -mv

./ncpmvdir ~sample1 ~/sample4/sample5 -cp pdf
./ncpmvdir ~sample1 ~/sample4/sample5 -mv java
*/

// Creating symbolic constants or macros.
#define MAXIMUM_EXTENSIONS 6 // Maximum 6 extensions are allowed.
#define MAXIMUM_DEPTH 64 // Maximum depth allowed is 64 for this program execution for directory traversal.
#define MAXIMUM_PATH_LENGTH 512 // Maxmimum path length is restricted to 512
#define DEBUG_MODE_ENABLED 1 //  It indicates about the debug mode. Its value can be either 0(Inactive) or 1(active).
#define _XOPEN_SOURCE 1
#define _XOPEN_SOURCE_EXTENDED 1

// Including external header files.
#include <ftw.h> // Header file defined for nftw function which will be used during the execution of this program for traversing directory trees.
#include <sys/stat.h> // System specific header file which works with file and fiel system information.
#include <unistd.h> // Provides access to many operating system functionalities.
#include <libgen.h> // Provides functionality for manipulating file path.
#include <stdio.h> // Provides functions for Input and Output operations.
#include <stdlib.h> // Provides function for memory allocation and process control.
#include <string.h> // Provides functions for string(chr, cmp, len, cpy, cat) manipulation.


char * source_directory_path, * destination_directory_path, * home_directory; // Declaring 3 character pointers.
char resolved_source_directory_path[MAXIMUM_PATH_LENGTH], resolved_destination_directory_path[MAXIMUM_PATH_LENGTH], output_directory[2 * MAXIMUM_PATH_LENGTH], temporary_path[MAXIMUM_PATH_LENGTH]; // Declaring arrays for storage space to store sequence of characters.
int debug_mode = DEBUG_MODE_ENABLED; // Refering to symbolic constant declared on top of the program execution.

int mode = 0, number_of_extensions = 0; // Defining 2 integer variables. Mode value 0 defines inactive or default state and declaring number of extensions to zero. Latter variable will keep a check on the count of extensions during the program execution.
char * extensions[MAXIMUM_EXTENSIONS]; //Declaring an array of character pointers. It will be used to process files with specific extensions or perform operations based on different file types.

```

```
56  // Function created to log messages in log_file.txt(placed in the same directory as this
    file.)
57  void log_message(const char* logged_message)
58      {
59      FILE* log_file = fopen("log_file.txt", "a"); // It will open the log file in append
        mode.
60      if (log_file != NULL)
61      {
62          fprintf(log_file, "%s\n\n", logged_message); // It will write the message to the
            log file in the end.
63          fclose(log_file); // fclose() function will close the log file.
64      }
65      }
66
67  // Callback function for nftw to copy/move directories.
68  // Callback function for nftw which performs file treewalk.
69  int remove_directory_callback(const char * filepath,
70     const struct stat * file_stat, int flag_type,
71        struct FTW * ftwbuf)
72        {
73     // Delete the entire source file/directory
74     if (debug_mode) {
75        if (flag_type == FTW_F) // Executing if it is a regular file.
76        {
77           printf("Removing a file present at this filepath: %s\n", filepath);
78        } else if (flag_type == FTW_DP) // Executing if it is a directory.
79        {
80           printf("Removing a directory present at this filepath: %s\n", filepath);
81        } else
82           printf("Removing from the filepath: %s\n", filepath);
83     }
84     remove(filepath); // For deleting the file or dir
85     return 0;
86  }
87
88  // This function is resolving relative path by converting to absolute path.
89  char * resolve_relative_path(char * path)
90  {
91     char temporary[MAXIMUM_PATH_LENGTH]; // Temporary variable to store the path.
92     char current_working_directory[MAXIMUM_PATH_LENGTH]; // for storing path of current
       working directory.
93     char * pointer = NULL;
94     size_t length;
95
96     // Copy the path to temporary variable.
97     snprintf(temporary, sizeof(temporary), "%s", path); //Copying input path using
       snprintf. Also, used for safely formatting a string into a fixed-sized character
       array.
98     length = strlen(temporary); //strlen function of string library calculates the length
       of path saved in temporary variable.
99
100    if (strncmp(temporary, "./", 2) == 0) // For relative path then converting to
       absolute path and then updating the new path.
101    {
102       getcwd(current_working_directory, sizeof(current_working_directory));
103       sprintf(temporary_path, "%s%s", current_working_directory, temporary + 1);
104    } else // If already a absolute path
105       strcpy(temporary_path, temporary);
106
107    // Replacing last part of path from "/" to "\0"  if "/" is found.
108    length = strlen(temporary_path);
109    if (temporary_path[length - 1] == '/')
110        temporary_path[length - 1] = '\0';
111    return temporary_path;
112
113 }
114
115 // This function is recursively creating a directory folders/structure.
116 int directory_creation(const char * path) {
```

```
117      // Temporary variable to store the path
118      char temp[1024];
119      char * p = NULL;
120      size_t len;
121
122      // Using snprintf to Copy the contents of the path argument into the temp array.
123      snprintf(temp, sizeof(temp), "%s", path);
124      len = strlen(temp);
125
126      // Replacing the trailing '/' if any with 0 to remove trailing slash from the path.
127      if (temp[len - 1] == '/')
128         temp[len - 1] = 0;
129
130      // Iterating through the each character of string to replace '/' with null character
         '\0'
131      for (p = temp + 1;* p; p++)
132         if ( * p == '/') {
133            * p = 0; // Replace the '/' with null character
134            // Create the directory with read, write, and execute permissions for the owner
135            if (debug_mode) {
136            log_message("Directory is created.");
137               printf("Creating directory and directory path is: %s\n", temp);
138            }
139            mkdir(temp, S_IRWXU); //Creating a directory using mkdir function using READ,
               Write and Execute permission.
140            * p = '/'; // Restoring the null character to its original value.
141         }
142
143      mkdir(temp, S_IRWXU); // For creating the last directory in the temp string, which
         represents the final directory in the path.
144
145      // Return success i.e. 0
146      return 0;
147   }
148
149   // This function is Verifying if a file has a particular extension to allow for
      operation(copy/move)
150   int check_extensions(const char * file_path)
151       {
152       // The strrchr function is utilized in order to find the last occurrence of the
          character '.' in the file path.
153      const char * extension = strrchr(file_path, '.');
154
155      // In case no extension is found, 0 is returned.
156      if (!extension) return 0;
157
158      // Incrementing the extension pointer to skip the dot(.).
159      extension++;
160
161      // Iterating through extensions
162      for (int i = 0; i < number_of_extensions; i++)
163      {
164         if (strcmp(extension, extensions[i]) == 0)
165         {
166            // 1 is returned if Extension is found in the list.
167            return 1;
168         }
169      }
170
171      // 0 is returned if Extension is not found in the list.
172      return 0;
173   }
174
175   int copy_move_directory_callback(const char * filepath,
176      const struct stat * file_stat, int flag_type, struct FTW * ftwbuf)
177      {
178      int creating_directory = 0;
179
180      // Constructing the destination path
```

```
181      char destination_path[4096];
182      strcpy(destination_path, output_directory);
183      strcat(destination_path, filepath + strlen(resolved_source_directory_path));
184
185      // Leaving aside the files/folders that are not specified in the extension list.
186      if (number_of_extensions > 0 && flag_type == FTW_F && check_extensions(filepath))
187      {
188         if (debug_mode) // It skips the files that are not in the extension list if
         debugging mode is enabled, it will print a message.
189         {
190            printf("Skipping this filepath: %s\n", filepath);
191         }
192      }
193
194      // 1. Creating a directory if flag type is FTW_D,
195      // 2. It will print few messages if DEBUG_MODE_ENABLED is set to 1.
196      // 3. Then it will create a correponding directory in the destination path with
      appropriate permissions mentioned in the below logic.
197      else if (flag_type == FTW_D)
198      {
199         if (debug_mode) {
200            printf("Copying filepath for the directory is: %s\n", filepath);
201            printf("Destination path to which directory is copied: %s\n", destination_path);
202         }
203         //Creating the directory with Read, Write, and Execute Permission for the owner.
204         mkdir(destination_path, S_IRWXU);
205      }
206
207      // 1. Copying or moving a file if flag type is FTW_F(signifies regular file).
208      // 2. It will print few messages if DEBUG_MODE_ENABLED is set to 1.
209      // 3. It will copy the file by opening(using fopen) the source and destination files,
      reading the content of the source file character by character, and then proceeding
      with writing it to the destination file.
210      // 4. It then closes both files using fclose.
211      else if (flag_type == FTW_F)
212      {
213         if (debug_mode)
214         {
215            printf("Filepath Copied: %s\n", filepath);
216            printf("Destination of Filepath: %s\n", destination_path);
217         }
218         FILE * source_file = fopen(filepath, "r");
219         FILE * destination_file = fopen(destination_path, "w");
220         int c;
221         while ((c = fgetc(source_file)) != EOF) // Iterating until reaches end of file.
222         {
223            fputc(c, destination_file);
224         }
225         fclose(source_file);
226         fclose(destination_file);
227      }
228      return 0;
229   }
230
231
232   // MAIN FUNCTION
233   int main(int argc, char * argv[]) {
234      umask(0); // File mode creation mask is set to zero which will allow all permissions
      to be set on newly created files as well as directories.
235      char * base;
236      int source_directory_path_length;
237
238      // Verifying if the correct number of arguments are passed or not.
239      if (argc < 4)
240      {
241      log_message("ERROR: Wrong command syntax used.");
242         printf("Synopsis to use: directory_copy_move [source_directory]
         [destination_directory] [options] <extension list>\n");
243         return 1;
```

```
244     }
245     else if (argc > MAXIMUM_EXTENSIONS + 4)
246     {
247        log_message("Error: Maximum extensions limit exceeded.\n");
248        printf("Error: Maximum %d extensions are permitted.\n", MAXIMUM_EXTENSIONS);
249        return 1;
250     }
251
252     // Verifying if the specified options are either -cp or -mv, then setting mode
       accordingly.
253     if (strcmp(argv[3], "-cp") == 0)
254     {
255        mode = 0;
256        log_message("Copy Mode : Mode 0 is set for Copy(mv) Operation.");
257     }
258     else if (strcmp(argv[3], "-mv") == 0)
259     {
260        mode = 1;
261        log_message("Move Mode : Mode 1 is set for Move(mv) Operation.");
262     } else
263     {
264        log_message("Error: You have used Invalid option. You can either use -cp(for copy)
           or -mv(for moving file or directory).\n");
265        printf("Error: You have used Invalid option. You can either use -cp(for copy) or
           -mv(for moving file or directory).\n");
266        return 1;
267     }
268
269     // Verifying if the extension list is supplied by the user in the terminal command
       and adding it to the extensions array for future use for performing operations.
270     for (int i = 4; i < argc; i++)
271     {
272        extensions[number_of_extensions++] = argv[i];
273     }
274
275     // Getting the path of the home directory.
276      home_directory = getenv("HOME");
277
278      // Reading the source and destination directories from the input arguments supplied
        by user.
279      source_directory_path = argv[1];
280      destination_directory_path = argv[2];
281
282      // Resolving the relative paths using the "resolve_relative_path" function for the
        source and destination directories.
283      sprintf(resolved_source_directory_path, "%s", resolve_relative_path(
        source_directory_path));
284      sprintf(resolved_destination_directory_path, "%s", resolve_relative_path(
        destination_directory_path));
285
286      // Getting the path length using "strlen" function of the resolved source directory.
287      source_directory_path_length = strlen(resolved_source_directory_path);
288
289      // Verifying if the source directory exists or not.
290      struct stat source_stat;
291      if (stat(resolved_source_directory_path, &source_stat) == -1)
292      {
293          log_message("Error: Source directory is not available.");
294          perror("Error: Source directory is not available.");
295          return 1;
296      }
297
298      // Verifying if the source and destination directories are the equivalent or if the
        destination is inside the source directory. - This condition prevents infinite loop
        and is not permissible.
299      if (strncmp(resolved_source_directory_path, resolved_destination_directory_path,
        source_directory_path_length) == 0)
300      {
301          log_message("Error: Source and destination directories are either in the same
```

```
            path or the destination is inside the source directory.\n");
302         log_message("This is not allowed as it will lead to an infinite loop.\n");
303         printf("Check for Error: Source and destination directories are either in the
            same path or the destination is inside the source directory.");
304         return 1;
305     }
306
307     //Verifying for Destination Directory
308     base = basename(resolved_source_directory_path);
309     sprintf(output_directory, "%s/%s", resolved_destination_directory_path, base);
310
311     // Verifying if both the source and destination directories belongs to the home
        directory hierarchy.
312     if (strncmp(resolved_source_directory_path, home_directory, strlen(home_directory))
        || strncmp(resolved_destination_directory_path, home_directory, strlen(home_directory
        )))
313     {
314         log_message("Error: Both source and destination directories must be within the
            home directory hierarchy.\n");
315         printf("Check for Error: Both source and destination directories must be within
            the home directory hierarchy.\n");
316         return 1;
317     }
318
319     // Creating output directory if it does not exist using creating_directory function.
320     directory_creation(resolved_destination_directory_path);
321
322     // Copy or move the directory using C library nftw(New File Tree Walk).
323
324     // If the nftw function returns a non-zero value which will eventually means that
        there is a failure in traversing the directory tree. In this case, an error message
        is logged into log file.
325     if (nftw(resolved_source_directory_path, copy_move_directory_callback, MAXIMUM_DEPTH,
         FTW_PHYS) != 0)
326     {
327         log_message("Check for Error: Error is related to nftw failure, please check.");
328         return 1;
329     }
330
331     //If the mode is 1 that means the program is in move(mv) mode, and tries to remove
        the source directory using the nftw function. If it displays , it wil log an error
        message to the log file and return 1.
332     if (mode == 1 && nftw(resolved_source_directory_path, remove_directory_callback,
        MAXIMUM_DEPTH, FTW_DEPTH | FTW_PHYS) != 0)
333     {
334         printf("Check for nftw and logic for move mode.");
335         log_message("Check for Error: Error related to the failure of nftw.");
336         return 1;
337     }
338     return 0;
339     }
340
341 /*
342 HANDLED ERROR CONDITIONS IN THE ABOVE PROGRAM:
343
344 1. Wrong command syntax
345 2. Maximum extensions limit exceeded
346 3. Invalid option(other than cp/mv)
347 4. Source directory not available
348 5. Source and destination directories lies in the same path or destination inside source
    directory(it will cause infinite loop.)
349 6. Source and destination directories not within the home directory hierarchy(source and
    destination lies in HOME directory)
350 7. nftw failure for directory traversal
351
352 Above mentioned error conditons are checked at different stages of the program execution
    and then appropriate messages are logged in a file and also, printed on terminal.
353
354 */
```