Myshell

1.0

Generated by Doxygen 1.9.5

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 file Struct Reference

Data Fields

- char * name
- struct file * pun
- struct stat sb

3.1.1 Detailed Description

Definition at line 16 of file ls.c.

3.1.2 Field Documentation

3.1.2.1 name

```
char* name
```

File name

Definition at line 17 of file ls.c.

3.1.2.2 pun

```
struct file* pun
```

Pointer to file i-node

Definition at line 18 of file ls.c.

3.1.2.3 sb

```
struct stat sb
```

File's i-node

Definition at line 19 of file ls.c.

The documentation for this struct was generated from the following file:

- src/ls.c

3.2 option Struct Reference

Data Fields

- int l_info
- int a_hidden
- int desired
- int help

3.2.1 Detailed Description

Definition at line 22 of file ls.c.

3.2.2 Field Documentation

3.2.2.1 a_hidden

```
int a_hidden
```

Flag for printing all the files (hidden included)

Definition at line 24 of file ls.c.

3.2.2.2 desired

```
int desired
```

Flag for desired output: 0 files, 1 directories, 2 all

Definition at line 25 of file ls.c.

3.2.2.3 help

```
int help
```

Flag for printing help

Definition at line 26 of file ls.c.

3.2.2.4 l_info

```
int l_info
```

Flag for printing the files' i-nodes

Definition at line 23 of file ls.c.

The documentation for this struct was generated from the following file:

- src/ls.c

Chapter 4

File Documentation

4.1 src/history.c File Reference

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
```

Macros

- #define FILE_NAME ".history"

Functions

- int main (int argc, char ∗argv[ ])

  This program manages the history of the command successfully executed by the user. The history of myshell is kept inside a hidden filled ".history", stored in the original folder where the process myshell is launched. Depending on the value of "command_counter", this program can write in append on the file ".history" (command_counter >= 0) or it can read from it and print its content on stdout (command_counter = -1). It takes as inputs the total number of the executed commands, the name of the last command and the path for the storage of the file ".history".

Variables

- FILE ∗ history = NULL
- const int MAX_LINE = 1024

4.1.1 Macro Definition Documentation

4.1.1.1 FILE_NAME

```
#define FILE_NAME ".history"
```

Definition at line 7 of file history.c.

4.1.2 Function Documentation

4.1.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

This program manages the history of the command successfully executed by the user. The history of myshell is kept inside a hidden filled ".history", stored in the original folder where the process myshell is launched. Depending on the value of "command_counter", this program can write in append on the file ".history" (command_counter >= 0) or it can read from it and print its content on stdout (command_counter = -1). It takes as inputs the total number of the executed commands, the name of the last command and the path for the storage of the file ".history".

Parameters

in	*argv[1]*	equivalent to the command counter. If set to -1, the file ".history" will be open on read.
in	*argv[2]*	contains the absolute storage path, where the file ".history" will be saved
in	*argv[3]*	if argv[1] >= 0, it holds the name of the last successfully executed command.

Returns

Returns 0 on success, a non-zero integer on error.

Definition at line 26 of file history.c.

4.1.3 Variable Documentation

4.1.3.1 history

```
FILE* history = NULL
```

Definition at line 9 of file history.c.

4.1.3.2 MAX_LINE

```
const int MAX_LINE = 1024
```

Definition at line 10 of file history.c.

4.2 history.c

Go to the documentation of this file.

```
00001 #include <unistd.h>
00002 #include <stdlib.h>
00003 #include <stdio.h>
00004 #include <errno.h>
00005 #include <string.h>
00006
00007 #define FILE_NAME ".history"
00008
00009 FILE* history = NULL;
00010 const int MAX_LINE = 1024;
00011
00026 int main(int argc, char* argv[]) {
00027
00028     int command_counter = atoi(argv[1]);
00029
00030     if (argc > 6) {
00031         fprintf(stderr, "Unexpected arguments\n");
00032         exit(EXIT_FAILURE);
00033     }
00034
00035     /* Redirects the error output stream to the specified file in the arguments (if present) */
00036     FILE* error_stream = stderr;
00037
00038     for (int i = 0; i < argc; i++) {
00039         if (strcmp(argv[i], "2>") == 0) {
00040             if (argv[i + 1] != NULL) {
00041                 if ((error_stream = fopen(argv[i + 1], "a")) == NULL) {
00042                     fprintf(stderr,"Error while opening the error output stream\n");
00043                     exit(EXIT_FAILURE);
00044                 }
00045                 /* Set the error stream to unbeffered */
00046                 if (setvbuf(error_stream, NULL, _IONBF, 0) != 0) {
00047                     fprintf(stderr, "Error while setting the error stream as unbuffered:  %s",
      strerror(errno));
00048                     exit(EXIT_FAILURE);
00049                 }
00050             } else {
00051                 fprintf(stderr, "Wrong arguments!  Did you mean \"2> error_output_file\"?\n");
00052                 exit(EXIT_FAILURE);
00053             }
00054         }
00055     }
00056
00057     /* Updates number of arguments (excluding redirection arguments) */
00058     if (error_stream != stderr) {
00059         argc = argc - 2;
00060     }
00061
00062     /* Checks arguments to understand in which mode the program should run */
00063     if ((argc < 4 && (command_counter >= 0)) || (argc < 3 && (command_counter == -1))) {
00064         fprintf(error_stream, "Too few arguments\n");
00065         exit(EXIT_FAILURE);
00066     }
00067
00068     if (command_counter < -1) {
00069         fprintf(error_stream, "Wrong arguments\n");
00070         exit(EXIT_FAILURE);
00071     }
00072
00073     /* Get the storage path for ".history" */
00074     char* path = (char*)malloc(strlen(argv[2]) + strlen(FILE_NAME) + 2);
00075     strcpy(path, argv[2]);
00076     strcat(path, "/");
00077     strcat(path, FILE_NAME);
00078
00079     if (command_counter >= 0) {
00080
00081         /* Open the file stream in write mode */
00082         if (command_counter == 0) {     /* New session of myshell:  truncate the file */
```

```
00083             if ((history = fopen(path, "w")) == NULL) {
00084                 fprintf(error_stream, "Error while opening file \".history\":  %s\n",
    strerror(errno));
00085                 exit(EXIT_FAILURE);
00086             }
00087         } else {                        /* Not a new session:  append to the file */
00088             if ((history = fopen(path, "a")) == NULL) {
00089                 fprintf(error_stream, "Error while opening file \".history\":  %s\n",
    strerror(errno));
00090                 exit(EXIT_FAILURE);
00091             }
00092         }
00093
00094         /* Print the line to ".history" */
00095         if (fprintf(history, "%s %s\n", argv[1], argv[3]) <= 0) {
00096             fprintf(error_stream, "Error while writing\n");
00097             exit(EXIT_FAILURE);
00098         }
00099
00100     } else if (command_counter == -1) {
00101
00102         /* Open the file stream in read mode */
00103         if ((history = fopen(path, "r")) == NULL) {
00104             fprintf(error_stream, "Error while opening file \".history\":  %s\n", strerror(errno));
00105             exit(EXIT_FAILURE);
00106         }
00107
00108         /* Read each line of ".history" */
00109         char buff[MAX_LINE];
00110         while (fgets(buff, MAX_LINE, history) != NULL) {
00111
00112             //Print it on stdout
00113             if (fputs(buff, stdout) == EOF) {
00114                 fprintf(error_stream, "Error while printing to stdout\n");
00115                 exit(EXIT_FAILURE);
00116             }
00117         }
00118     }
00119
00120     /* Close the stream */
00121     if (fclose(history) == EOF) {
00122         fprintf(stderr, "Error while closing the stream\n");
00123         exit(EXIT_FAILURE);
00124     }
00125
00126     /* Closes the error stream */
00127     fclose(error_stream);
00128
00129     free(path);
00130     return EXIT_SUCCESS;
00131 }
```

4.3 src/ls.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
```

Data Structures

- struct file
- struct option

Macros

- #define MAX_BUFF 256
- #define FILES 0
- #define DIRS 1
- #define ALL 2

Functions

- void add_to_list (struct file **last, const struct stat *sb, const char *name)

  This function appends on top the passed entry of the directory to the specified list.
- void free_list (struct file **files)

  This function deallocates the reserved space in the heap memory used for the lists.
- void print_error (const char *)
- void print_list (struct file *list, const char *title, struct option *opt)

  This function prints the final output of the ls command, based on the selected options.
- void print_help ()

  This function prints the help section for the ls command.
- struct option options (int argc, char **argv)

  This function parses the argument string and select the desired options.
- int main (int argc, char *argv[ ])

  This function emulates the ls command from the Unix shell.
- void print_error (const char str[ ])

  *This function prints an error message on error_stream, with format: "Error while *your string* [code *errno*, *strerror(errno)*]".*

Variables

- char * ddir = NULL
- FILE * error_stream = NULL

4.3.1 Macro Definition Documentation

4.3.1.1 ALL

```
#define ALL 2
```

Definition at line 14 of file ls.c.

4.3.1.2 DIRS

```
#define DIRS 1
```

Definition at line 13 of file ls.c.

4.3.1.3 FILES

```
#define FILES 0
```

Definition at line 12 of file ls.c.

4.3.1.4 MAX_BUFF

```
#define MAX_BUFF 256
```

Definition at line 11 of file ls.c.

4.3.2 Function Documentation

4.3.2.1 add_to_list()

```
void add_to_list (
            struct file ** last,
            const struct stat * sb,
            const char * name )
```

This function appends on top the passed entry of the directory to the specified list.

Parameters

in	*last*	Pointer to the address of the last element of the passed list.
in	*sb*	Pointer to the file's i-node
in	*name*	String of the file's name

Definition at line 207 of file ls.c.

4.3.2.2 free_list()

```
void free_list (
            struct file ** files )
```

This function deallocates the reserved space in the heap memory used for the lists.

Parameters

in	*files*	Pointer to the address of the last element of the list, which has to be freed.

Definition at line 288 of file ls.c.

4.3.2.3 main()

```
int main (
            int argc,
            char * argv[] )
```

This function emulates the ls command from the Unix shell.

Note

The options must be preceded by a dash '-'.

Parameters

in	*argv[1]*	Desired path on which the ls will be executed
in	*argv[2]*	Options (Try "ls -h" for more informations).

Returns

Returns 0 on success, 1 on error.

Definition at line 49 of file ls.c.

4.3.2.4 options()

```
struct option options (
            int argc,
            char ** argv )
```

This function parses the argument string and select the desired options.

Returns

Returns an option structure containing the flags relative to the desired options.

Definition at line 162 of file ls.c.

4.3.2.5 print_error() [1/2]

```
void print_error (
            const char *  )
```

4.3.2.6 print_error() [2/2]

```
void print_error (
            const char str[] )
```

This function prints an error message on error_stream, with format: "Error while ∗your string∗ [code ∗errno∗, ∗strerror(errno)∗]".

Parameters

in	*str*	Your costant string, which will be pasted inside the error message.

Definition at line 325 of file ls.c.

4.3.2.7 print_help()

```
void print_help ( )
```

This function prints the help section for the ls command.

Definition at line 306 of file ls.c.

4.3.2.8 print_list()

```
void print_list (
            struct file * list,
            const char * title,
            struct option * opt )
```

This function prints the final output of the ls command, based on the selected options.

Parameters

in	*list*	List of files inside the desired directory (pointer to the last element of the list).
in	*title*	First line of the output.
in	*opt*	Selected option flag structure.

Definition at line 238 of file ls.c.

4.3.3 Variable Documentation

4.3.3.1 ddir

```
char* ddir = NULL
```

Definition at line 29 of file ls.c.

4.3.3.2 error_stream

```
FILE* error_stream = NULL
```

Definition at line 39 of file ls.c.

4.4 ls.c

Go to the documentation of this file.

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include <dirent.h>
00005 #include <errno.h>
00006 #include <unistd.h>
00007 #include <sys/stat.h>
00008 #include <pwd.h>
00009 #include <grp.h>
00010
00011 #define MAX_BUFF 256
00012 #define FILES 0
00013 #define DIRS 1
00014 #define ALL 2
00015
00016 struct file {
00017     char* name;
00018     struct file* pun;
00019     struct stat sb;
00020 };
00021
00022 struct option {
00023     int l_info;
00024     int a_hidden;
00025     int desired;
00026     int help;
00027 };
00028
00029 char* ddir = NULL; /* desired directory */
00030
00031 //void prepare_command(int, char**);
00032 void add_to_list(struct file**, const struct stat*, const char* name);
00033 void free_list(struct file**);
00034 void print_error (const char*);
00035 void print_list(struct file*, const char*, struct option*);
00036 void print_help();
00037 struct option options(int, char**);
00038
00039 FILE* error_stream = NULL;    /* Error output stream */
00040
00048 int
00049 main (int argc, char* argv[]) {
00050
00051     error_stream = stderr;
00052
00053     /* Cheks the number of arguments passed */
00054     if (argc > 9) {   /* Maximum number of arguments, considering the path and all the 5 options
    selected */
00055         fprintf(error_stream, "Unexpected arguments!\n");
00056         exit(EXIT_FAILURE);
00057     }
00058
00059     /* Redirects the error output stream to the specified file in the arguments (if present) */
00060     for (int i = 0; i < argc; i++) {
00061         if (strcmp(argv[i], "2>") == 0) {
```

```
00062             if (argv[i + 1] != NULL) {
00063                 if ((error_stream = fopen(argv[i + 1], "a")) == NULL) {
00064                     fprintf(error_stream,"Error while opening the error output stream\n");
00065                     exit(EXIT_FAILURE);
00066                 }
00067                 /* Set the error stream to unbeffered */
00068                 if (setvbuf(error_stream, NULL, _IONBF, 0) != 0) {
00069                     fprintf(error_stream, "Error while setting the error stream as unbuffered:  %s",
      strerror(errno));
00070                     exit(EXIT_FAILURE);
00071                 }
00072             } else {
00073                 fprintf(error_stream, "Wrong arguments!  Did you mean \"2> error_output_file\"?\n");
00074                 exit(EXIT_FAILURE);
00075             }
00076         }
00077     }
00078
00079     /* Gets options */
00080     struct option opt = options(argc, argv);
00081
00082     /* Prints the help section and terminates the process */
00083     if (opt.help) {
00084         print_help();
00085         return EXIT_SUCCESS;
00086     }
00087
00088     /* Updates number of argumetns (excluding redirection arguments) */
00089     if (error_stream != stderr) {
00090         argc = argc - 2;
00091     }
00092
00093     /* Sets "path_arg_num" to the index of the argument containing the path on which the ls will be
      executed */
00094     int path_arg_num = 1;   /* Index of the path argument.  Starts from 1 (excludes the program name)
      */
00095
00096     /* Scans the arguments, until it either reaches the end or it finds a "non-option" argument */
00097     while ((path_arg_num < argc) && (strchr(argv[path_arg_num], '-') != NULL)) {
00098         path_arg_num++;
00099     }
00100
00101     /* Open the specified path (if present); otherwise it opens the current directory*/
00102     if ((path_arg_num < argc) && (argv[path_arg_num] != NULL)) {
00103         if (chdir(argv[path_arg_num]) < 0) {
00104             fprintf(error_stream, "ls:  cannot access '%s':  No such file or directory\n",
      argv[path_arg_num]);
00105             return EXIT_FAILURE;
00106         }
00107     }
00108
00109     DIR* dp;
00110     int fd;
00111
00112     /* Open destination directory */
00113     dp = opendir(getcwd(NULL, 0));
00114
00115     if (dp == NULL) {
00116         print_error("opening directory");
00117         return EXIT_FAILURE;
00118     }
00119
00120     struct file* files = NULL;
00121     struct file* dirs = NULL;
00122
00123     struct dirent* dirp;
00124
00125     /* Scan each entry of the directory */
00126     while ((dirp = readdir(dp)) != NULL) {
00127
00128         struct stat sb;
00129
00130         if (lstat(dirp->d_name, &sb) < 0){
00131             print_error("opening i-node");
00132             return EXIT_FAILURE;
00133         }
00134
00135         /* Sort files and directories in the right list */
00136         if (S_ISDIR(sb.st_mode) > 0) {
00137             add_to_list (&dirs, &sb, dirp->d_name);
00138         } else {
00139             add_to_list (&files, &sb, dirp->d_name);
00140         }
00141     }
00142     closedir(dp);
00143
00144     /*print the lists*/
```

```
00145     if (opt.desired != FILES)
00146         print_list(dirs, "--> Directories:", &opt);
00147     if (opt.desired != DIRS)
00148         print_list(files, "--> Files:", &opt);
00149 
00150     /*free heap*/
00151     free_list(&files);
00152     free_list(&dirs);
00153 
00154     return EXIT_SUCCESS;
00155 }
00156 
00161 struct option
00162 options(int argc, char** argv) {
00163 
00164     struct option opt;
00165     opt.a_hidden = 0;
00166     opt.l_info = 0;
00167     opt.desired = ALL;
00168     opt.help = 0;
00169 
00170     /* Parses the arguments array to find the set options.  It automatically discards the */
00171     /* arguments not written with the format "-" + option */
00172     char ch;
00173     while ((ch = getopt (argc, argv, "ladfh")) != -1) {
00174 
00175         switch (ch) {
00176             case 'l':
00177                 opt.l_info = 1;     /*include info*/
00178                 break;
00179             case 'a':
00180                 opt.a_hidden = 1;   /*show hidden files*/
00181                 break;
00182             case 'd':
00183                 opt.desired = DIRS; /*show directories only*/
00184                 break;
00185             case 'f':
00186                 opt.desired = FILES; /*show files only*/
00187                 break;
00188             case 'h':
00189                 opt.help = 1; /*help*/
00190                 break;
00191             default:
00192                 fprintf(error_stream, "Try 'ls -h' for more information\n");
00193                 exit(EXIT_FAILURE);
00194         }
00195     }
00196 
00197     return opt;
00198 }
00199 
00206 void
00207 add_to_list (struct file** last, const struct stat* sb, const char* name) {
00208 
00209     struct file* new_elem = malloc (sizeof (struct file));
00210 
00211     if (new_elem == NULL) {
00212         print_error ("allocating");
00213         exit(EXIT_FAILURE);
00214     }
00215 
00216     new_elem->name = calloc (strlen (name) + 1, sizeof(char));
00217     if (new_elem->name == NULL) {
00218         print_error ("allocating");
00219         exit(EXIT_FAILURE);
00220     }
00221     strcpy (new_elem->name, name);
00222 
00223     /*adding the new element at the beggining of the list*/
00224     new_elem->sb = *sb;
00225     new_elem->pun = *last;
00226     *last = new_elem;
00227 
00228     return;
00229 }
00230 
00237 void
00238 print_list (struct file* list, const char* title, struct option* opt) {
00239 
00240     printf("%s\n", title);
00241 
00242     for (struct file* pivot = list; pivot != NULL; pivot = pivot->pun) {
00243 
00244         if (!(opt->a_hidden) && *(pivot->name) == '.')  /*continue if it is an hidden file*/
00245             continue;
00246         if (opt->l_info) { /*show options*/
00247 
```

```
00248             fprintf(stdout, "- ");
00249
00250             fprintf(stdout, "%c", !S_ISREG(pivot->sb.st_mode) ?  's' :  '-');
00251
00252             /*permissions*/
00253             fprintf(stdout, "%c", (pivot->sb.st_mode & S_IRUSR) ? 'r' :  '-');
00254                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IWUSR) ? 'w' :  '-');
00255                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IXUSR) ? 'x' :  '-');
00256                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IRGRP) ? 'r' :  '-');
00257                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IWGRP) ? 'w' :  '-');
00258                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IXGRP) ? 'x' :  '-');
00259                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IROTH) ? 'r' :  '-');
00260                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IWOTH) ? 'w' :  '-');
00261                         fprintf(stdout, "%c", (pivot->sb.st_mode & S_IXOTH) ? 'x' :  '-');
00262
00263             /*hard links*/
00264             fprintf(stdout, "\t%d", pivot->sb.st_nlink);
00265
00266             /*owners*/
00267             struct passwd *pw = getpwuid(pivot->sb.st_uid);
00268             struct group *gr = getgrgid(pivot->sb.st_gid);
00269             fprintf(stdout, "\t%s\t%s", pw->pw_name, gr->gr_name);
00270
00271             /*size*/
00272             fprintf(stdout, "\t%d", pivot->sb.st_size);
00273
00274             /*name*/
00275             fprintf(stdout, "\t%s\n", pivot->name);
00276
00277         } else /*show only the file name*/
00278             printf("\t- %s\n", pivot->name);
00279     }
00280     return;
00281 }
00282
00287 void
00288 free_list (struct file** files) {
00289
00290     if (*files == NULL)
00291         return;
00292
00293     for (struct file* pivot = *files; *files != NULL; *files = pivot) {
00294         pivot = (*files)->pun;
00295         free((*files)->name);
00296         free(*files);
00297     }
00298
00299     return;
00300 }
00301
00305 void
00306 print_help() {
00307     const char help[] =
00308         "\tList information about the FILEs (the current directory by default).\n\
00309 \t-a\tShow hidden files.\n\
00310 \t-l\tShow info.\n\
00311 \t-h\tPrint help page.\n\
00312 \t-f\tShow files only.\n\
00313 \t-d\tShow directories only.\n\
00314 \n";
00315
00316     fprintf(stdout, "%s", help);
00317 }
00318
00324 void
00325 print_error (const char str[]) {
00326     fprintf(error_stream, "Error while %s [code %d, %s] \n", str, errno, strerror(errno));
00327     return;
00328 }
```

4.5 src/myshell.c File Reference

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
```

```
#include <limits.h>
#include <sys/wait.h>
#include <math.h>
```

Macros

- #define default_name "guest"
- #define PROMPT "@myshell# "
- #define filename "/mainv2"
- #define MAX_BUFF 256
- #define MAX_ARG 20
- #define H_READ -1

Functions

- void sigint_handler (int signo)

  This functions handles the CTRL+C interrupt (SIGINT).
- void setup (int, char **)
- void loop ()

  This function repeats, until termination of the process, the tasks of the shell: reads the command, executes it and shows again the prompr, ready for the next command, eventually displaying an error message.
- void update_path (char *argv0)

  This function updates the local variable PATH of the process to the absolute path of the bin folder, where all the executable files can be found.
- void exit_command (void)

  This functions handles the exit command (greets the user).
- void print_error (const char *)
- void clear_args (char **args)

  This function clears the dynamic array of strings, made to hold the arguments.
- char * read_command (char *stdin_buff)

  This function reads one line from stdin and stores it in a string.
- int parse_command (char *string, char **args_vect)

  This function parses the string read on stdin and collects the name of the command and the arguments that the user wants to execute. In addition, it prepares the arguments' array, which will be sent to the child process.
- int prepare_history (const int, const char *, const char *, char **)
- int run_command (char **)
- int cd (char **)
- int main (int argc, char *argv[ ])

  Main function: after setting up the new PATH, it calls the setup() and loop() functions.
- void setup (int argc, char *argv[ ])

  This function checks if the right number of arguments was passed by the user, tries to register the exit and SIGINT handlers and sets up the prompt string with the right username. Moreover, it handles the eventual redirection of the error output stream.
- int prepare_history (const int command_counter, const char *storage_path, const char *last_command, char *args[ ])

  This function prepares the array of arguments for the history command.
- int run_command (char *args[ ])

  This function forks the current process and executes the command requested by the user in the child process. Sets the parent process on wait for the child process.
- int cd (char *args[ ])

  This function changes the current directory to the one passed in args.
- void print_error (const char str[ ])

  *This function prints an error message on stderr, with format: "Error while *your string* [code *errno*, *strerror(errno)*]".*

Variables

- char * username = NULL
- char * prompt = NULL
- char * bin_path = NULL
- char * err_file_path = NULL

4.5.1 Macro Definition Documentation

4.5.1.1 default_name

```
#define default_name "guest"
```

Definition at line 11 of file myshell.c.

4.5.1.2 filename

```
#define filename "/mainv2"
```

Definition at line 13 of file myshell.c.

4.5.1.3 H_READ

```
#define H_READ -1
```

Definition at line 16 of file myshell.c.

4.5.1.4 MAX_ARG

```
#define MAX_ARG 20
```

Definition at line 15 of file myshell.c.

4.5.1.5 MAX_BUFF

```
#define MAX_BUFF 256
```

Definition at line 14 of file myshell.c.

4.5.1.6 PROMPT

```
#define PROMPT "@myshell# "
```

Definition at line 12 of file myshell.c.

4.5.2 Function Documentation

4.5.2.1 cd() [1/2]

```
int cd (
            char **  )
```

4.5.2.2 cd() [2/2]

```
int cd (
            char * args[] )
```

This function changes the current directory to the one passed in args.

Parameters

in	*args[1]*	If present, contains the new path.

Returns

Returns 0 on success, 1 on error.

Definition at line 443 of file myshell.c.

4.5.2.3 clear_args()

```
void clear_args (
            char ** args )
```

This function clears the dynamic array of strings, made to hold the arguments.

Parameters

in	*args*	pointer to the array of strings that has to be cleared.

Definition at line 473 of file myshell.c.

4.5.2.4 exit_command()

```
void exit_command (
            void  )
```

This functions handles the exit command (greets the user).

Definition at line 510 of file myshell.c.

4.5.2.5 loop()

```
void loop ( )
```

This function repeats, until termination of the process, the tasks of the shell: reads the command, executes it and shows again the prompr, ready for the next command, eventually displaying an error message.

Definition at line 138 of file myshell.c.

4.5.2.6 main()

```
int main (
            int argc,
            char * argv[] )
```

Main function: after setting up the new PATH, it calls the setup() and loop() functions.

Parameters

in	*argc*	Number of arguments passed by the user through the main shell (2 maximum)
in	*argv*	Array of arguments. argv[0] contains the path to this file as passed by the user; argv[1] (optional) contains the username set by the user ("guest" as default).

Returns

Returns 0 in case of success, a non-zero integer oterwise.

Definition at line 47 of file myshell.c.

4.5.2.7 parse_command()

```
int parse_command (
            char * string,
            char ** args_vect )
```

This function parses the string read on stdin and collects the name of the command and the arguments that the user wants to execute. In addition, it prepares the arguments' array, which will be sent to the child process.

Parameters

in	*string*	pointer to the command string read on stdin
in	*args*	pointer to the array of strings, which will be sent to the child process.

Returns

Returns 0 on success, EOF on error.

Definition at line 280 of file myshell.c.

4.5.2.8 prepare_history() [1/2]

```
int prepare_history (
            const int command_counter,
            const char * storage_path,
            const char * last_command,
            char * args[] )
```

This function prepares the array of arguments for the history command.

Note

args[1] := command_counter args[2] := storage_path args[3] := last_command If command_counter is equal to -1, then the file ".history" will be opened on read, therefore there is no need of the string "last_command" (args[3] is set to NULL).

It clears the arguments' array.

Parameters

in	*command_counter*	counter of the successfully executed commands.
in	*storage_path*	absolute path to the storage folder, where the file ".history" is located.
in	*last_command*	a string containing the name of the last successfully executed command.
in	*args*	pointer to the array of arguments.

Returns

Returns 0 on success, EOF on error.

Definition at line 337 of file myshell.c.

4.5.2.9 prepare_history() [2/2]

```
int prepare_history (
            const int ,
            const char * ,
            const char * ,
            char **  )
```

4.5.2.10 print_error() [1/2]

```
void print_error (
            const char *  )
```

4.5.2.11 print_error() [2/2]

```
void print_error (
            const char str[] )
```

This function prints an error message on stderr, with format: "Error while ∗your string∗ [code ∗errno∗, ∗strerror(errno)∗]".

Parameters

in	*str*	Your costant string, which will be pasted inside the error message.

Definition at line 488 of file myshell.c.

4.5.2.12 read_command()

```
char * read_command (
            char * stdin_buff )
```

This function reads one line from stdin and stores it in a string.

Note

The pointer to the string sent by the user will be modified.

Parameters

in	*stdin_buff*	pointer to the location in memory, where the read string will be written.

Returns

Returns the pointer to the read string, NULL on error.

Definition at line 243 of file myshell.c.

4.5.2.13 run_command() [1/2]

```
int run_command (
            char **  )
```

4.5.2.14 run_command() [2/2]

```
int run_command (
            char * args[] )
```

This function forks the current process and executes the command requested by the user in the child process. Sets the parent process on wait for the child process.

Parameters

in	*args*	arguments' array for the called command.

Returns

Returns 0 if the child process executes and terminates correctly, a non-zero integer otherwise

Definition at line 407 of file myshell.c.

4.5.2.15 setup() [1/2]

```
void setup (
            int argc,
            char * argv[] )
```

This function checks if the right number of arguments was passed by the user, tries to register the exit and SIGINT handlers and sets up the prompt string with the right username. Moreover, it handles the eventual redirection of the error output stream.

Parameters

in	*argc*	number of arguments passed to main()
in	*argv*	array of arguments passed to main()

Definition at line 94 of file myshell.c.

4.5.2.16 setup() [2/2]

```
void setup (
            int ,
            char **  )
```

4.5.2.17 sigint_handler()

```
void sigint_handler (
            int signo )
```

This functions handles the CTRL+C interrupt (SIGINT).

Parameters

in	*signo*	ID numbe for SIGINT.

Definition at line 498 of file myshell.c.

4.5.2.18 update_path()

```
void update_path (
            char * argv0 )
```

This function updates the local variable PATH of the process to the absolute path of the bin folder, where all the executable files can be found.

Parameters

in	*argv0*	Corrsponds to the argv[0] string sent to the main; therefore, it contains the program path sent by the user as argument (It can be relative or absolute compared to the actual location of the main shell when myshell is launched).

Definition at line 68 of file myshell.c.

4.5.3 Variable Documentation

4.5.3.1 bin_path

```
char* bin_path = NULL
```

Definition at line 35 of file myshell.c.

4.5.3.2 err_file_path

```
char* err_file_path = NULL
```

Definition at line 36 of file myshell.c.

4.5.3.3 prompt

```
char* prompt = NULL
```

Definition at line 34 of file myshell.c.

4.5.3.4 username

```
char* username = NULL
```

Definition at line 33 of file myshell.c.

4.6 myshell.c

Go to the documentation of this file.

```
00001 #include <unistd.h>
00002 #include <stdlib.h>
00003 #include <stdio.h>
00004 #include <errno.h>
00005 #include <string.h>
00006 #include <signal.h>
00007 #include <limits.h>
00008 #include <sys/wait.h>
00009 #include <math.h>
00010
00011 #define default_name "guest"
00012 #define PROMPT "@myshell# "
00013 #define filename "/mainv2"
00014 #define MAX_BUFF 256
00015 #define MAX_ARG  20
00016 #define H_READ   -1
00017
00018 void sigint_handler(int);
00019 void setup(int, char**);
00020 void loop();
00021 void update_path(char*);
00022 void exit_command(void);
00023 void print_error(const char*);
00024 void clear_args(char** args);
00025
00026 char*  read_command(char*);
00027
00028 int parse_command(char*, char**);
00029 int prepare_history(const int, const char*, const char*, char**);
00030 int run_command(char**);
00031 int cd(char**);
00032
00033 char* username = NULL;
00034 char* prompt = NULL;
00035 char* bin_path = NULL;
00036 char* err_file_path = NULL;
00037
00038
00046 int
00047 main(int argc, char* argv[]) {
00048     /* Set stderr to default stderr */
00049     stderr = stderr;
00050
00051     update_path(argv[0]);
00052
00053     /* Setup function */
00054     setup(argc, argv);
00055
00056     /* Loop function */
00057     loop();
00058 }
00059
00067 void
00068 update_path(char* argv0) {
00069     char* last_occurence = NULL;
00070     if ((last_occurence = strrchr(argv0, '/')) == NULL) {
00071         print_error("updating the PATH variable");
00072     }
00073
00074     int rel_path_length = (int)(last_occurence - argv0);
00075     char* relative_path = (char*)malloc(rel_path_length);
00076     strncat(relative_path, argv0, rel_path_length);
00077
00078     /* Gets absolute bin path, given its relative path */
00079     bin_path = realpath(relative_path, bin_path);
00080     setenv("PATH", bin_path, 1);
00081
00082     free(relative_path);
00083     return;
00084 }
00085
00093 void
00094 setup(int argc, char* argv[]) {
00095     FILE* stderr = stderr;
00096
00097     /* Check if there's an exceeding number of arguments */
00098     if (argc > 4) {
00099         fprintf(stderr, "Unexpected arguments!\n");
00100         exit(EXIT_FAILURE);
00101     }
00102
00103     /* Tries to add an exit handler */
```

```
00104     if (atexit(exit_command) != 0) {
00105         fprintf(stderr, "Cannot register exit handler.\n");
00106         exit(EXIT_FAILURE);
00107     }
00108 
00109     /* Tries to add a SIGINT interrupt handler */
00110     if (signal(SIGINT, sigint_handler) == SIG_ERR) {
00111         fprintf(stderr, "Error!  Couldn't register SIGINT handler:  %s\n", strerror(errno));
00112         exit(EXIT_FAILURE);
00113     }
00114 
00115     /* Chooses username */
00116     if (argc == 1 && (stderr == stderr)) {
00117         username = (char*)malloc(strlen(default_name)+1);
00118         strcpy(username, default_name);
00119     } else {
00120         username = (char*)malloc(strlen(argv[1])+1);
00121         strcpy(username, argv[1]);
00122     }
00123 
00124     /* Initializes prompt string */
00125     prompt = (char*)malloc(strlen(username)+1);
00126     strcpy(prompt, username);
00127     prompt = strcat(prompt, PROMPT);
00128 
00129     return;
00130 }
00131 
00137 void
00138 loop() {
00139     char*  last_command = NULL;     /* Name of the last successfully executed command */
00140     char*  stdin_buff = NULL;       /* Stdin buffer */
00141     char*  args[MAX_ARG] = {NULL};  /* Array of strings, containing the arguments to pass */
00142     int    exit_status = 0;
00143     int    command_counter = 0;     /* Successfully executed command counter */
00144 
00145 
00146     /* Starts loop */
00147     while (1) {
00148 
00149         /* Print the prompt */
00150         printf("%s", prompt);
00151 
00152         /* Read the command */
00153         stdin_buff = read_command(stdin_buff);
00154 
00155         if (stdin_buff == NULL) {   /* Nothing was read:  error */
00156             print_error("reading");
00157             exit(EXIT_FAILURE);
00158         } else if (strcmp(stdin_buff, "\n") == 0) { /* Read \n:  prompt again */
00159             continue;
00160         }
00161 
00162         /* Parse the command */
00163         if (parse_command(stdin_buff, args) == EOF) {
00164             print_error("parsing the command");
00165             continue;
00166         }
00167 
00168         /* Selects command */
00169         if (strcmp (args[0], "cd") == 0) {
00170             exit_status = cd(args);
00171         }
00172         else if (strcmp(args[0], "ls") == 0 || strcmp(args[0], "pwd") == 0) {
00173             exit_status = run_command(args);
00174         }
00175         else if (strcmp(args[0], "history") == 0) {
00176             if (prepare_history(H_READ, bin_path, last_command, args) == EOF) {
00177                 print_error("preparing history");
00178                 continue;
00179             }
00180 
00181             exit_status = run_command(args);
00182 
00183         } else if (strcmp (args[0], "exit") == 0) {
00184             /* Add exit command to .history file */
00185             if (prepare_history(command_counter, bin_path, "exit", args) == EOF) {
00186                 print_error("preparing history");
00187                 /* No continue instruction is used, let the process exit anyways */
00188             }
00189 
00190             /* Run "history" and check its exit status */
00191             if (run_command(args) == EXIT_FAILURE) {
00192                 print_error("executing history");
00193                 /* No continue instruction is used, let the process exit anyways */
00194             }
00195 
```

```
00196             /* Frees all the dynamically allocated memory */
00197             /* String username is not freed, the exit handler function needs it */
00198             clear_args(args);
00199             free(last_command);
00200             free(stdin_buff);
00201             free(prompt);
00202             free(bin_path);
00203 
00204             /* Exit through exit handler */
00205             exit(EXIT_SUCCESS);
00206         }
00207         else {
00208             exit_status = EXIT_FAILURE;
00209             printf("bash:  %s:  command not found\n", args[0]);
00210             continue;
00211         }
00212 
00213         /* Check the exit status of the last command.  Add it to history if it succeeded.  */
00214         if (exit_status == EXIT_SUCCESS) {
00215             last_command = (char*)realloc(last_command, strlen(args[0]) + 1);
00216             strcpy(last_command, args[0]);
00217 
00218             if (prepare_history(command_counter, bin_path, last_command, args) == EOF) {
00219                 print_error("preparing history");
00220                 continue;
00221             }
00222 
00223             /* Run "history" and check its exit status */
00224             if (run_command(args) == EXIT_FAILURE) {
00225                 print_error("executing history");
00226             } else {
00227                 command_counter++;
00228             }
00229 
00230         }
00231 
00232         clear_args(args);
00233     }
00234 }
00235 
00242 char*
00243 read_command(char* stdin_buff) {
00244     char tempbuf[MAX_BUFF]; /* Temporary buffer */
00245     size_t inputlen = 0;
00246     size_t templen = 0;
00247 
00248     /* Store at most MAX_BUFF chars in tempbuf and append them in stdin_buff.  */
00249     /* If stdin stream has still content, repeat until new line.                */
00250     do {
00251         if (fgets(tempbuf, MAX_BUFF, stdin) == NULL) {
00252             print_error("reading from stdin");
00253             return NULL;
00254         }
00255 
00256         templen = strlen(tempbuf);
00257         stdin_buff = realloc(stdin_buff, inputlen + templen + 1);
00258 
00259         if (stdin_buff == NULL) {
00260             print_error("reading from stdin");
00261             return NULL;
00262         }
00263 
00264         strcpy(stdin_buff + inputlen, tempbuf);
00265         inputlen += templen;
00266     } while (templen == MAX_BUFF-1 && tempbuf[MAX_BUFF-2] != '\n');
00267 
00268     return stdin_buff;
00269 }
00270 
00279 int
00280 parse_command(char* string, char** args_vect) {
00281     int is_word = 0;
00282     int arg_num = 0;
00283     char buff[MAX_BUFF] = "";
00284 
00285     for (int i = 0; i < strlen(string); i++) {
00286         char c = string[i];
00287 
00288         /* Check if it's an alphanumerical character (i.e.  plausible text) */
00289 
00290         if (c != '\0' && c != ' ' && c != '\n') {
00291             is_word = 1;
00292             strncat(buff, &c, 1);   /*Concatenates chars until assembling the first isolated word in
    the user input */
00293         }
00294 
00295         if (c == ' ' || c == '\n') {
```

```
00296             if (is_word == 0) {
00297                 continue;
00298             } else {
00299                 /* Adds the word found to the arguments' array */
00300                 args_vect[arg_num] = (char*)malloc(strlen(buff) + 1);
00301 
00302                 if (args_vect[arg_num] == NULL) {
00303                     return EOF;
00304                 }
00305 
00306                 strcpy(args_vect[arg_num], buff);
00307                 arg_num++;
00308 
00309                 /* Clears the buffer made for containing each word */
00310                 buff[0] = '\0';
00311                 is_word = 0;
00312             }
00313         }
00314     }
00315 
00316     /* Adds the arguments' array terminator, i.e.  NULL pointer */
00317     args_vect[arg_num++] = NULL;
00318 
00319     return 0;
00320 }
00321 
00336 int
00337 prepare_history(const int command_counter, const char* storage_path, const char* last_command, char*
      args[]) {
00338     /* Clear the arguments' array */
00339     clear_args(args);
00340 
00341     /* Allocates enough characters, in order to contain the integer command_counter converted into a
      string */
00342     int size = 1;
00343     int _command_counter = command_counter;
00344 
00345     if (command_counter >= 0) {     /* Open ".history" file on write in append */
00346         /* Counts how many digits there are inside integer command_counter*/
00347         while ((_command_counter / 10) != 0) {
00348             size++;
00349             _command_counter = _command_counter / 10;
00350         }
00351 
00352         /* Prepare the first argument */
00353         args[1] = (char*)malloc((size_t)(size + 1));
00354         if (args[1] == NULL) {
00355             return EOF;
00356         }
00357 
00358         /* Prepare the third argument */
00359         args[3] = (char*)malloc(strlen(last_command) + 1);
00360         if (args[3] == NULL) {
00361             return EOF;
00362         }
00363         strcpy(args[3], last_command);
00364 
00365         /* Sets the terminator of the arguments' string */
00366         args[4] = NULL;
00367 
00368     } else if (command_counter == H_READ) {     /* Open ".history" file on read */
00369         /* Prepare the first argument */
00370         size = (int)strlen("-1") + 1;
00371         args[1] = (char*)malloc(strlen("-1") + 1);
00372         if (args[1] == NULL) {
00373             return EOF;
00374         }
00375 
00376         /* Sets the terminator of the arguments' string */
00377         args[3] = NULL;
00378     }
00379 
00380     /* Sets the argument containing the name of the command history */
00381     args[0] = (char*)malloc(strlen("history") + 1);
00382     if (args[0] == NULL) {
00383             return EOF;
00384         }
00385     strcpy (args[0], "history");
00386 
00387     /* Converts the integer command_counter to a string, by printing it on the very same string */
00388     /* snprintf() is used in order to avoid overflow errors */
00389     snprintf(args[1], (size_t)(size + 1), "%d", command_counter);
00390 
00391     /* Prepare the second argument for both cases */
00392     args[2] = (char*)malloc(strlen(storage_path) + 1);
00393     if (args[2] == NULL) {
00394             return EOF;
```

```
00395         }
00396     strcpy(args[2], storage_path);
00397
00398     return 0;
00399 }
00400
00407 int run_command(char* args[]) {
00408         /* start forking */
00409         int status;
00410         pid_t pid;
00411
00412         if ((pid = fork()) < 0) {
00413             print_error("forking");
00414             return EXIT_FAILURE;
00415         }
00416         else if (pid == 0) {
00417             /* child process */
00418             if (execvp(args[0], args) == EOF){
00419                 fprintf(stderr, "process:  %s\n", args[0]);
00420                 print_error("executing child process");
00421                 return EXIT_FAILURE;
00422             }
00423         }
00424         else {
00425             /* parent process */
00426             if (wait(&status) < 0) {
00427                 print_error("waiting");
00428                 return EXIT_FAILURE;
00429             }
00430             if (WIFEXITED(status) && WEXITSTATUS(status) == EXIT_SUCCESS)
00431                 return EXIT_SUCCESS;
00432             else
00433                 return EXIT_FAILURE;
00434         }
00435         //return EXIT_SUCCESS;
00436 }
00437
00443 int cd(char* args[]) {
00444     int arg_num = 0;
00445     while (args[arg_num] != NULL) {
00446         arg_num++;
00447     }
00448
00449     char* newdir;
00450
00451     if (arg_num > 2) {
00452         fprintf(stderr, "Unexpected arguments\n");
00453         return EXIT_FAILURE;
00454     }
00455
00456     if (arg_num == 2) {
00457         newdir = (char*)malloc(strlen(args[1]) + 1);
00458         strcpy(newdir, args[1]);
00459         if (chdir(newdir) < 0) {
00460             print_error("opening");
00461             return EXIT_FAILURE;
00462         }
00463     }
00464
00465     return EXIT_SUCCESS;
00466 }
00467
00472 void
00473 clear_args(char** args) {
00474     /* Frees each string in the array */
00475     for (int i = 0; i < MAX_ARG; i++) {
00476         free(args[i]);
00477         args[i] = NULL;
00478     }
00479     return;
00480 }
00481
00487 void
00488 print_error(const char str[]) {
00489     fprintf(stderr, "Error while %s [code %d, %s] \n", str, errno, strerror(errno));
00490     return;
00491 }
00492
00497 void
00498 sigint_handler(int signo) {
00499     /* Notifies the user about the right procedure in order to close Myshell */
00500     printf("\nTo close myshell, type \"exit\"\n");
00501     printf("%s", prompt);
00502     fflush(stdout);
00503     return;
00504 }
00505
```

```
00509 void
00510 exit_command(void) {
00511     printf("Goodbye, %s!\n", username);
00512 
00513     /* Deallocates the space reserved in the heap for the "username" string */
00514     free(username);
00515 }
```

4.7 src/pwd.c File Reference

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
```

Functions

- int main (int argc, char ∗argv[ ])

  This function gets the current path and prints it to stdout.

4.7.1 Function Documentation

4.7.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

This function gets the current path and prints it to stdout.

Returns

Returns 0 on success, 1 on error.

Definition at line 14 of file pwd.c.

4.8 pwd.c

Go to the documentation of this file.

```
00001 #include <stdlib.h>
00002 #include <stdio.h>
00003 #include <unistd.h>
00004 #include <errno.h>
00005 #include <string.h>
00006
00007 //Get current working path and print it to stdout:  returns 0 if it succeded, EOF = -1 otherwise
00008
00013 int
00014 main(int argc, char *argv[]) {
00015     /* Checks number of arguments (zero arguments expected) */
00016     if (argc > 3) {
00017         fprintf(stderr, "Unexpected arguments!\n");
00018         exit(EXIT_FAILURE);
00019     }
00020
00021     /* Redirects the error output stream to the specified file in the arguments (if present) */
00022     FILE* error_stream = stderr;
00023
00024     for (int i = 0; i < argc; i++) {
00025         if (strcmp(argv[i], "2>") == 0) {
00026             if (argv[i + 1] != NULL) {
00027                 if ((error_stream = fopen(argv[i + 1], "a")) == NULL) {
00028                     fprintf(stderr,"Error while opening the error output stream\n");
00029                     exit(EXIT_FAILURE);
00030                 }
00031                 /* Set the error stream to unbeffered */
00032                 if (setvbuf(error_stream, NULL, _IONBF, 0) != 0) {
00033                     fprintf(stderr, "Error while setting the error stream as unbuffered:  %s",
      strerror(errno));
00034                     exit(EXIT_FAILURE);
00035                 }
00036             } else {
00037                 fprintf(stderr, "Wrong arguments!  Did you mean \"2> error_output_file\"?\n");
00038                 exit(EXIT_FAILURE);
00039             }
00040         }
00041     }
00042
00043     char* current_path = NULL;
00044
00045     /* Gets current path and saves it in "current_path" string */
00046     if ((current_path = getcwd(NULL, 0)) == NULL) {
00047         fprintf(error_stream, "Error:  %s\n", strerror(errno));
00048         exit(EXIT_FAILURE);
00049     }
00050
00051     /* Prints "current_path" string */
00052     printf("%s\n", current_path);
00053     free(current_path);
00054
00055     /* Closes the error stream */
00056     fclose(error_stream);
00057
00058     return EXIT_SUCCESS;
00059 }
```

Index