Custom Shell Program

Mariam Ahmed Fathy 19016627

March 12, 2023

1 Introduction

This report presents a custom shell program written in C. The program provides a command-line interface where users can enter commands to be executed by the shell. The shell program supports basic commands, such as changing directories, list and echo.

2 Code Description

Video Link: https://youtu.be/8vgIPu$_g dS0$

The custom shell program is composed of several functions that are responsible for different aspects of the program. The `init()` function is responsible for clearing the terminal screen and displaying the current user, directory, and time.

The `argsFormater(bluechar line[])` function formats the input string by removing leading and trailing white space and checking for the presence of an ampersand () to indicate that a process should be run in the background. The function also splits the input string into individual arguments and stores them in an array.

The `scanner(bluechar line[])` function reads input from the console using the `fgets()` function and passes the input to the `argsFormater(bluechar line[])` function.

The `process(bluechar* args[],bluechar line[])` function processes the input by checking for the CD command and splitting the input string into individual arguments.

The `parser(bluechar* args[], bluechar line[])` function scans input from the user and calls the `scanner(bluechar line[])` and `process(bluechar* args[],bluechar line[])` functions to process the input.

The `execute(bluechar* args[])` function executes the command entered by the user. If the process is run in the background, the function sets the flag variable to 1 and executes the process using the `execvp()` function. If the process is run in the foreground, the function waits for the process to complete using the `waitpid()` function.

The `signalHandler(blueint signal)` function is called when a child process terminates and sets the flag variable to 0.

3 The Algorithm

The program is separated into small functions:

3.1 main()

The function starts by removing any old logs by calling the remove function with the name of the log file.

It then sets up a signal handler for child processes using the signal function with the SIGCHLD signal and the signalHandler function.

The fuction declares an array of char called line with a maximum size of MAXCHAR. This array will be used to store the user's command line input.

The program then calls the init function, which likely initializes any necessary variables or data structures.

The fuction enters a loop that will continue to read and execute commands from the user until the parser function returns false. The parser function is not shown in the code, but it likely parses the user's input into an array of strings, which are then stored in the args array.

For each iteration of the loop, the program calls the execute function with the args array as its argument. The execute function likely takes the array of strings and executes the appropriate command based on the user's

input.

Finally, it returns 0 to indicate successful completion.

3.2 init()

This init() function to be initializing the terminal by displaying some information to the user such as username, current directory, date, and time. Here are the steps it follows:
-Clear the terminal screen.
-Display a greeting message.
-Get the username using getenv("USER").
-Get the current working directory using getcwd() and display it.
-Display the current date and time using time() and localtime().
-Sleep for 2 seconds.
-Clear the terminal screen again.
Overall, this function provides some basic information to the user and gives the terminal a clean and organized appearance.

3.3 argsFormater(char line[])

The argsFormater function takes a character array line as input and formats it into individual arguments for command execution. Here is a breakdown of what it does:

1. It trims any leading whitespace characters in the input string line.
2. It trims any trailing whitespace characters in the input string line.
3. It checks for the presence of an ampersand () character at the end of the input string line. If found, it sets a global flag variable to 1 and removes the ampersand from the string.
4. It splits the input string line into individual arguments using strtok() function with delimiter of white space (" "). The individual arguments are stored in the global args array, with the last element of args set to NULL.

Note that MAXARGS is a macro defined at the beginning of the program and specifies the maximum number of arguments that can be parsed. The args array is also a global variable defined at the beginning of the program and stores the parsed command arguments.

3.4 scanner(char line[])

The scanner function reads a line of input from the console using the fgets function and passes the resulting string to the argsFormater function. The fgets function takes three arguments: a pointer to a character array (line), the maximum number of characters to read (MAXCHAR), and a pointer to the file from which to read the input (stdin in this case, which represents standard input, i.e., the console).

After reading the line of input, the argsFormater function is called to format the input and store the resulting arguments in the args array.

3.5 process(char* args[],char line[])

The process function takes the args array and the line string as input. The args array is expected to have been initialized and cleared before calling this function. The function first checks if the first argument is NULL or empty, and if so, it prints an error message and returns 1. Then it checks if the first argument is "cd", and if so, it handles the cd command by setting the path variable and changing the current directory using chdir. Finally, it loops through each character in the line string and sets the args array to each argument separated by spaces, while ignoring arguments enclosed in quotes.

If the process function encounters the exit command, it will not exit the program but will continue to execute the loop in the main function. This is because the exit command is checked within the while loop of the main function, and if the command is detected, the loop will exit and the program will terminate.

3.6 parser(char* args[], char line[])

The parser function is responsible for getting input from the user, processing it, and returning a flag indicating whether the program should continue running. Here's how it works:

It first prints the prompt to the user, consisting of their username followed by "¿¿ ". It then calls the scanner function, which reads input from the console and formats it into an array of arguments using the argsFormater function. It then calls the process function, which further processes the array of arguments to handle special commands like cd and to handle quoted arguments. Finally, it returns a flag indicating that the program should continue running. Overall, the parser function serves as the main entry point for getting user input and processing it, allowing the shell to interact with the user and execute commands.

3.7 signalHandler(int signal)

The signalHandler function is a callback function that is called when the parent process receives a SIGCHLD signal from one of its child processes.

When a child process terminates, the operating system sends a SIGCHLD signal to the parent process. The purpose of the signal is to notify the parent process that one of its child processes has terminated.

The signalHandler function opens a log file in append mode, writes a message indicating that a child process was terminated, and then closes the file. The function also prints a message to the console indicating that a child process was terminated.

3.8 execute(char* args[])

The execute() function uses fork() to create a new child process and then executes the command passed in through the args array using execvp(). If the flag variable is set to 1, it indicates that the command should run in the background, so the parent process does not wait for the child process to complete. Otherwise, the parent process waits for the child process to complete using waitpid().

If the execvp() function fails to execute the command, the child process prints an error message to the console.

4 Conclusion

The custom shell program provides a basic command-line interface that allows users to execute commands and run processes. The program demonstrates the use of several C libraries and functions, including `stdio.h`, `stdlib.h`, `string.h`, `unistd.h`, and `signal.h`. The program could be extended to include additional functionality, such as support for input/output redirection and pipes.

5 The source code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <sys/utsname.h>
#include <signal.h>
#include <fcntl.h>
#include <pwd.h>
#include <time.h>
#include <dirent.h>

#define MAX_CHAR 50
#define MAX_ARGS 10
#define MAX_HISTORY 10


#define clear() printf("\033[999;999H") // Clear the terminal and move the cursor

int flag = 0;                           // Flag of background process
char *log_file = "logFile.txt";
FILE * fptr;
char *args[MAX_ARGS] = {NULL};


void init();
void argsFormater(char line[]);
void scanner(char line[]);
int process(char* args[],char line[]);
int parser(char* args[], char line[]);
void signalHandler(int signal);
void execute(char* args[]);


/**********************main**********************/
int main() {

    //Remove old logs
    remove("logFile.txt");

    // Set up a signal handler for child processes *******
    signal(SIGCHLD,signalHandler);
    // char * args[MAX_ARGS];       //maximun length of command arguments
    char line[MAX_CHAR];            //maximum length of the command line
    init();

    // Read and excute commands from the user
    while(parser(args,line)) {
        execute(args);
    }
    return 0;
}
/*
```

Figure 1: Code Block 1

```
/*
/*********************init*********************/

void init() {
        // Clear the terminal screen.
    clear();
    printf("\n\n*****************"
        "************************");
    printf("\n\t    HELLO     ");
    printf("\n\n******************"
        "**********************");

    // Display username and current directory
    char* username = getenv("USER");
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("\n\n\nUSER is: @%s", username);
        printf("\nCurrent directory: %s", cwd);
    } else {
        printf("\n\n\nUSER is: @%s", username);
    }

    // Display date and time
    time_t t = time(NULL);
    struct tm *tm = localtime(&t);
    printf("\nDate and time: %s", asctime(tm));

    sleep(2);
    clear();
}


/******************argsFormater*******************/
// modify format of input
void argsFormater(char line[]) {

    int i = 0;
    int j = 0;
    // Trim leading white space
    while (isspace(line[i])) {
        i++;
    }
    // Trim trailing white space
    int len = strlen(line);
    while (isspace(line[len - 1])) {
        line[--len] = '\0';
    }
    // Check for presence of &
    while (line[j] != '\0') {
        if (line[j] == '&') {
            flag = 1;
            line[j] = '\0';
        }
        j++;
```

Figure 2: Code Block 2

```
        }
        j++;
    }
    // Split input into individual arguments
    char *token;
    int argIndex = 0;
    token = strtok(line, " \t");
    while (token != NULL && argIndex < MAX_ARGS - 1) {
      args[argIndex] = token;
        token = strtok(NULL, " \t");
        argIndex++;
    }
    args[argIndex] = NULL;
}


void scanner(char line[]) {
    // reads from console
    fgets(line, MAX_CHAR, stdin);
    argsFormater(line);
}

/*********************process*********************/
```

Figure 3: Code Block 3

```c
int process(char* args[],char line[]){
    int i = 0;
    args[i]=strtok(line," ");

    if(args[i]==NULL) {
        printf("Please Enter a COMMAND!\n");
        return 1;
    }

    // Flag variable to indicate whether we are inside a quoted argument
    int inside_quotes = 0;

    while(strcmp(line,"exit")==0 ){exit(0);}

    // Check for CD command
    if (strcmp(args[0], "cd") == 0) {
        char *path;
        if (args[1] == NULL || strcmp(args[1], "~") == 0) {
            // Go to home directory
            path = getenv("HOME");
        } else if (strcmp(args[1], "..") == 0) {
            // Go up one level in directory
            path = "..";
        } else if (args[1][0] == '/') {
            // Absolute path
            path = args[1];
        } else {
            // Relative path
            char cwd[1024];
            getcwd(cwd, sizeof(cwd));
            strcat(cwd, "/");
            strcat(cwd, args[1]);
            path = cwd;
        }
        if (chdir(path) != 0) {
            printf("Error: %s not found.\n", path);
        }
        return 0;
    }

    // Loop through each character in the input string
    for (int j = 0; j < strlen(line); j++) {
        // If we encounter a quotation mark, toggle the flag
        if (line[j] == '\"') {
            inside_quotes = !inside_quotes;
        }
        // If we encounter a space and we're not inside a quoted argument,
        // start a new argument
        else if (isspace(line[j]) && !inside_quotes) {
            args[++i] = strtok(NULL," ");
        }
    }

    return 1;
```

Figure 4: Code Block 4

```c
int process(char* args[],char line[]){
    int i = 0;
    args[i]=strtok(line," ");

    if(args[i]==NULL) {
        printf("Please Enter a COMMAND!\n");
        return 1;
    }

    // Flag variable to indicate whether we are inside a quoted argument
    int inside_quotes = 0;

    while(strcmp(line,"exit")==0 ){exit(0);}

    // Check for CD command
    if (strcmp(args[0], "cd") == 0) {
        char *path;
        if (args[1] == NULL || strcmp(args[1], "~") == 0) {
            // Go to home directory
            path = getenv("HOME");
        } else if (strcmp(args[1], "..") == 0) {
            // Go up one level in directory
            path = "..";
        } else if (args[1][0] == '/') {
            // Absolute path
            path = args[1];
        } else {
            // Relative path
            char cwd[1024];
            getcwd(cwd, sizeof(cwd));
            strcat(cwd, "/");
            strcat(cwd, args[1]);
            path = cwd;
        }
        if (chdir(path) != 0) {
            printf("Error: %s not found.\n", path);
        }
        return 0;
    }

    // Loop through each character in the input string
    for (int j = 0; j < strlen(line); j++) {
        // If we encounter a quotation mark, toggle the flag
        if (line[j] == '\"') {
            inside_quotes = !inside_quotes;
        }
        // If we encounter a space and we're not inside a quoted argument,
        // start a new argument
        else if (isspace(line[j]) && !inside_quotes) {
            args[++i] = strtok(NULL," ");
        }
    }

    return 1;
```

Figure 5: Code Block 5

```
****************************************
            HELLO

****************************************


USER is: @mariam
Current directory: /home/mariam/univ/sem8/os/19016627_mariamAhmed_lab1OS
Date and time: Sun Mar 12 08:27:11 2023

ariam>> ls
Debug  main  main.c  test
child terminated
mariam>> mkdir test
mkdir: cannot create directory ‘test’: File exists
child terminated
mariam>> ls
Debug  logFile.txt  main  main.c  test
child terminated
mariam>> ls -a -l -h
total 68K
drwxrwxr-x 5 mariam mariam 4.0K 08:27 12 مار .
drwxrwxr-x 5 mariam mariam 4.0K 08:20 12 مار ..
-rw-rw-r-- 1 mariam mariam  11K 18:46 5  مار .cproject
drwxrwxr-x 2 mariam mariam 4.0K 06:54 12 مار Debug
-rw-rw-r-- 1 mariam mariam   87 08:27 12 مار logFile.txt
-rwxrwxr-x 1 mariam mariam  18K 07:37 12 مار main
-rw-rw-r-- 1 mariam mariam 5.5K 07:22 12 مار main.c
-rw-rw-r-- 1 mariam mariam  758 18:46 5  مار .project
drwxrwxr-x 2 mariam mariam 4.0K 18:46 5  مار .settings
drwxrwxr-x 2 mariam mariam 4.0K 06:08 12 مار test
child terminated
mariam>> _
```

Figure 6: Basic commands

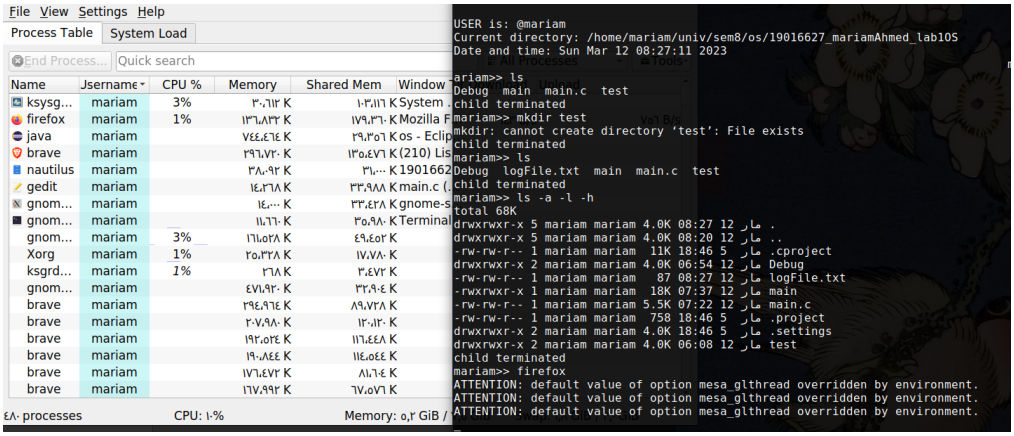

Figure 7: ksysduard with firefox is opened