SIMPLE SHELL PROGRAM

OS – LAB 1

Abdullah Mohamed AbdelHakim Gomaa
19015953

Code Summary:

This is a C program that implements a basic Unix shell. The main function registers a signal handler for SIGCHILD signal and sets the working environment to the current directory then it enters an infinite loop and takes input from the user using the parseInput() function. The program then accepts user input, parses it, and determines whether the command is a shell built-in command or an executable command using the inputType() function. If it is a shell built-in command, the program executes it. Otherwise, the program executes the command by calling execute_command() function which uses the execvp system call. The program continues the loop until the user enters the "exit" command to exit the shell. The code includes a section that creates a log file called "log.txt" to record when child processes are terminated.

Librares:

```
1  /************************************ Libraries Required ****************************************/
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<unistd.h>
5  #include<string.h>
6  #include<sys/wait.h>
7  #include<signal.h>
8  #include<errno.h>
```

Fig.1

Functions:

```
12  /************************************ Functions Protoypes ****************************************/
13  void parseInput();
14  void sig_handler();
15  int inputType();
16  void execute_shell_builtin(char* pr);
17  void execute_command();
18  void getValueFromKey(char* key, char** ret_val);
```

Fig.2

1) **parseInput():**

Step1:

- Taking input from the user and store every word in a separate element in array called args as shown in fig.3.

- The memset function is used to clear the arrays for future re-use of the arrays when the function is called again from the main program.

```
void parseInput()
{
    // Prepare variables for recieving next input
        memset(command, 0, sizeof(command));
        memset(p, 0, sizeof(p));
    // Initialize variables and set delimiter
        int i=0 , j=0;
        char buffer[512];
        char delim[] = " ";

    // Recieve input from the user and store it in buffer
        fgets(buffer, 512, stdin);

    // Remove new line feed at the end of input string
        buffer[strcspn(buffer, "\n")] = 0;

    // Seperate input string to seperate words and store every word in args array
        char *ptr = strtok(buffer, delim);
        while(ptr!=NULL)
        {
            args[i] = ptr;
            ptr = strtok(NULL, delim);
            i++;
        }
        args[i] = NULL;
```

Fig.3

Step2:

- Copy the command which is the first element in args array to cmd variable.
- If the command is “cd”, store the second element in args array to the path variable that clearly stores the path of the destination directory.
- Prepare the command array that is fed as input to the execvp function which will be illustrated later.
- Check for ‘&” and set the flag for background process.

```
// Store the command in cmd
    cmd = args[0];

// if the command is cd store path entered in path variable
    if(strcmp(cmd, "cd") == 0)
    {
        path = args[1];
    }

// Store the complete command entered in command array to be used in execvp
    i=0;
    while(args[i] != NULL)
    {
        command[i] = args[i];
        i++;
    }
    command[i] = NULL;

// Check for '&' for background process
    if(command[1]!= NULL )
    {
        if(command[1][0] == '&')
        {
            command[1] = NULL;
            // Set flag for background process
            flag =1;
        }
    }
```

Fig.4

Step3:

- Set cd_flag to 1 if the destination path is home directory.
- The chdir function requires a char array as an argument, so in this part the path received is stored in a character array p.

```
// Prepare the proper argument for cd command
    if(command[1] == NULL)
    {
        // Set cd_flag if required directory is home
        cd_flag = 1;
    }
    else if(command[1][0] == '~')
    {
        // Set cd_flag if required directory is home
        cd_flag = 1;
    }
    else
    {
        cd_flag = 0;
        // Prepare the path in suitable form for passing to chdir()
            path = command[1];
            for(int i=0; i<strlen(path); i++)
            {
                p[i] = path[i];
            }
    }
```

Fig.5

Step4:

- If there is an option or an argument to the command, it is going to be stored in character array called arr element by element including spaces which will help in future use of these options or arguments in other functions.
- Remove all quotes from arr.
- This was the last step in parsing input from the user and preparing data entered.

```
// Concatenate all arguments after the first word and store them in arr[] with space
    i=1; j=0;
    int k=0;
    while(args[i]!=NULL)
    {
        for(j=0; j<strlen(args[i]); j++)
        {
            arr[k] = args[i][j];
            k++;
        }
        arr[k++] = ' ';
        i++;
    }

// Remove all quotes from arr
    // Iterate over each character in the string
    for (i = 0, j = 0; arr[i] != '\0'; i++) {
        // If the current character is not a quote, copy it to the output string
        if (arr[i] != '"') {
            arr[j] = arr[i];
            j++;
        }
    }
    // Terminate the output string with a null character
    arr[j] = '\0';
```

Fig.6

2) inputType():

A helping function that checks if the entered command is shell built-in or if it is executable or error.

```
/****************************************** Input Type ******************************************/
int inputType()
{
    // Check for shell built-in or executable or error
        if( !strcmp(cmd,"cd") || !strcmp(cmd,"echo") || !strcmp(cmd,"export"))
            return shell_builtin;
        else
            return executable_or_error;
}
```

Fig.7

3) **getValueFromKey():**
Another helping function that facilitates getting the value of an environment defined variable from the variables array that is defined by the values entered by the user by export command.

```
void getValueFromKey(char* key, char** ret_val)
{
    // Initialize variables
        int index = 0;
        char *value = NULL;

    // Get index of value from variable name
    for(int i=0; i<30; i++)
    {
        if(strcmp(variables_names[i],key)==0)
        {
            index = i;
            break;
        }
    }

    // Get value
    value = variables_values[index];
    // store the value in the varibale passed by reference
    *ret_val = value;
}
```

Fig.8

4) **execute_shell_builtin(char* pr):**

Step1:

- Initialize the variables.

```
void execute_shell_builtin(char* pr)
{
    // Initialize variables
        char dir[256];
        int i =0, j=0, c=0;
        int index = 0;
        char temp[30];
        char* ptr = NULL;
        char * key = NULL;
        char * value = NULL;
```

Fig.9

Step2:

- If the command is cd, check for home directory flag (cd_flag), then change directory to the required path.

```
  /********************** cd ********************** */
if(strcmp(cmd, "cd") == 0)
{
    // Check cd flag for home directory
        if(cd_flag == 1)
        {
            // Get user home directory path
            path = getenv("HOME");
            for(int i=0; i<strlen(path); i++)
                p[i] = path[i];
        }

    // Change the directory to the required path
        chdir(p);

    getcwd(dir, 256);
}
```

Fig.10

Step3:

- Export command.
- There is an array called variable_names to store variable names entered by user and another called variable_values to store value of the variable such that by the same index both variable name and it's value could be accessed easily.

```
/********************* export ******************** */
else if ( strcmp(cmd, "export") == 0)
{
    // Get the variable name
        while(pr[i]!= '=')
        {
            temp[i] = pr[i];
            i++;
        }
    // Store variable name in the array variable_names
        strcpy(variables_names[variables_index],temp);
    // Clear temp array for getting value of the variable
        memset(temp, 0, sizeof(temp));
    // Increament the iterator for string entered
        i++;

    // Get value
        while (pr[i]!='\0')
        {
            temp[j++] = pr[i++];
        }

    // Store variable value in the array variable_values
        strcpy(variables_values[variables_index],temp);

    // Increament index for storing multiple variables
        variables_index++;

    // Clear temp array for future inputs
        memset(temp, 0, sizeof(temp));
}
```

Fig.11

Step4: echo command.

1. Check for $ in string then check if it is the first occurrence.
2. If it is the first occurrence get the value using the helping function getValueFromKey of the variable after $ and print the output.

```
// Check if input string contain a variable to be exported
    char * pos = strchr(pr, '$');
    if(pos!=NULL)
    {
        // Check if echo has no string but $ variable
        if(pr[0]=='$')
        {
            while(pr[c]!=0)
            {
                pr[c] = pr[c+1];
                c++;
            }
            // Get the options as string from the value stored in the variable
            getValueFromKey(pr, &value);
            printf("%s\n", value);
        }
```

Fig.12

3. If it is not first occurrence get variable name after $ to get required value.

```
else
{
    // Get variable name from string
        ptr = strtok(pr, "$");
        char* before = ptr;
        ptr = strtok(NULL, "$");
        char *after = ptr;

    // Remove space at the end of string after $
        if(after!=NULL)
            after[strcspn(after, " ")] = 0;

    // Check if variable is after a string to determine key
        if(after == NULL)
            key = before;
        else
            key = after;

    // Get index for the value corresponding to the required key
        for(int i=0; i<30; i++)
        {
            if(strcmp(variables_names[i],key)==0)
            {
                index = i;
                break;
            }
        }
    // Get the value of the key using the index
        value = variables_values[index];
```

Fig.13

4. Prepare output in suitable format for printing.

```
// Prepare suitable output format
    if(after != NULL)
        strcat(before, value);
    else
        before = value;

// Print output string after replacing variable with value
    printf("%s\n", before);
```

Fig.14

5. If the string does not contain $, print the output directly.

```
// Condition: String does not contain $
else
{
    // Print string entered by user
        printf("%s\n", pr);
}
```

Fig.15

5) execute_command():

step1:

- Initialize the variables.
- Create child process using fork.

```
void execute_command()
{
    // Initialize variables
        char* options = NULL;
        int wait_status =0;
        int w ;

    // Create child process
        __pid_t pid = fork();
```

Fig.16

Step2:

- Child Process
- Check if command is executable else raise an error.
- Check if command option start with $ to get value of the variable after $.
- Prepare command array to be passed to the execvp system call that executes the command.

```
if(pid==0)
{
// Child process created successfully
    // Check if command is executable
    if(inputType() == executable_or_error)
    {
        // Check if command option starts with $
            if((command[1] != NULL) && (command[1][0] == '$'))
            {
                // Remove $ sign to get variable name
                strncpy(command[1], command[1]+1, strlen(command[1])-1);
                command[1][strlen(command[1])-1] = '\0';

                // Get the options as string from the value stored in the variable
                getValueFromKey(command[1], &options);

                // Prepare command array by seperating options from the string
                    char * tok = strtok(options, " ");
                    int i =1;
                    while(tok != NULL)
                    {
                        command[i] = tok;
                        tok = strtok(NULL, " ");
                        i++;
                    }
            }
        // Execute the command
        execvp(command[0], command);

        // exevp failure errors
        perror("Error in execvp");
        exit(EXIT_FAILURE);
```

Fig.17

Step3:

- Parent or Error
- Check if error in child creation raise a fork error.
- In parent check for flag for background process and configure waitpid option to 0 to make the process run in background.
- For foreground process waitpid option should be WNOHANG to wait for child process to complete.

```
else if(pid == -1)
{
    // Child process creation failed
    perror("Error in Fork");
}
else
{
// Parent process
    if(flag == 0)
    {
        sleep(1);
        w = waitpid(-1, &wait_status, 0);
        // return;
    }
    else
    {
        // wait for child process to complete
        w = waitpid(-1, &wait_status, WNOHANG);
        if(w == -1)
        {
            // wait failure error
            perror("Error in waitpid");
            exit(EXIT_FAILURE);
        }
    }

}
```

Fig.18

6) sig_handler():

- Waits for child process to terminate to reap zombie processes.
- Create a log file which is appended by the statement “Child Terminated” every time a child process is terminated.

```
/*************************************** Signal Handler ***************************************/
void sig_handler()
{
    pid_t pid;
    int status;

    // Wait for any child process to terminate
        pid = waitpid(-1, &status, WNOHANG);

    // Enters the loop that continues as long as there are child processes to wait for
        while(pid>0)
        {
        // Create a log file if it is not created and open it
            FILE *f = fopen("log.txt", "a");
            if(f==NULL)
            {
                perror("Error in open log file");
            }
            else
            {
                // Add the string in a new line when a child process is terminated
                fprintf(f, "Child Terminated\n");
            }
            fclose(f);
        // update wait pid status
            pid = waitpid(-1, &status, WNOHANG);
        }

}
```

Fig.19

7) **main():**
   - Register a signal handler for signal SIGCHILD.
   - Check for exit command to break the super loop and ends the program.
   - Calls suitable function based on the command input type.

```
// Char array to store current directory
char currdir[256];
// Register a signal handler for SIGCHILD signal
signal(SIGCHLD, sig_handler);

// Loop continously until exit command
while (1)
{
    // Set working enviroment to current directory
        getcwd(currdir, 256);
        chdir(currdir);

    // Take input from user
        parseInput();

    // Check if command entered by user is exit to break the loop and end the program
        if(!strcmp(cmd, "exit"))
            break;

    // Check if command is shell built-in or exectuable or error
        if(inputType() == shell_builtin)
        {
            execute_shell_builtin(arr);

            // Clear arr for next input
            memset(arr, 0, sizeof(arr));
        }
        else if(inputType() == executable_or_error)
        {
            execute_command();
        }
}
```

Fig.20

Sample run:

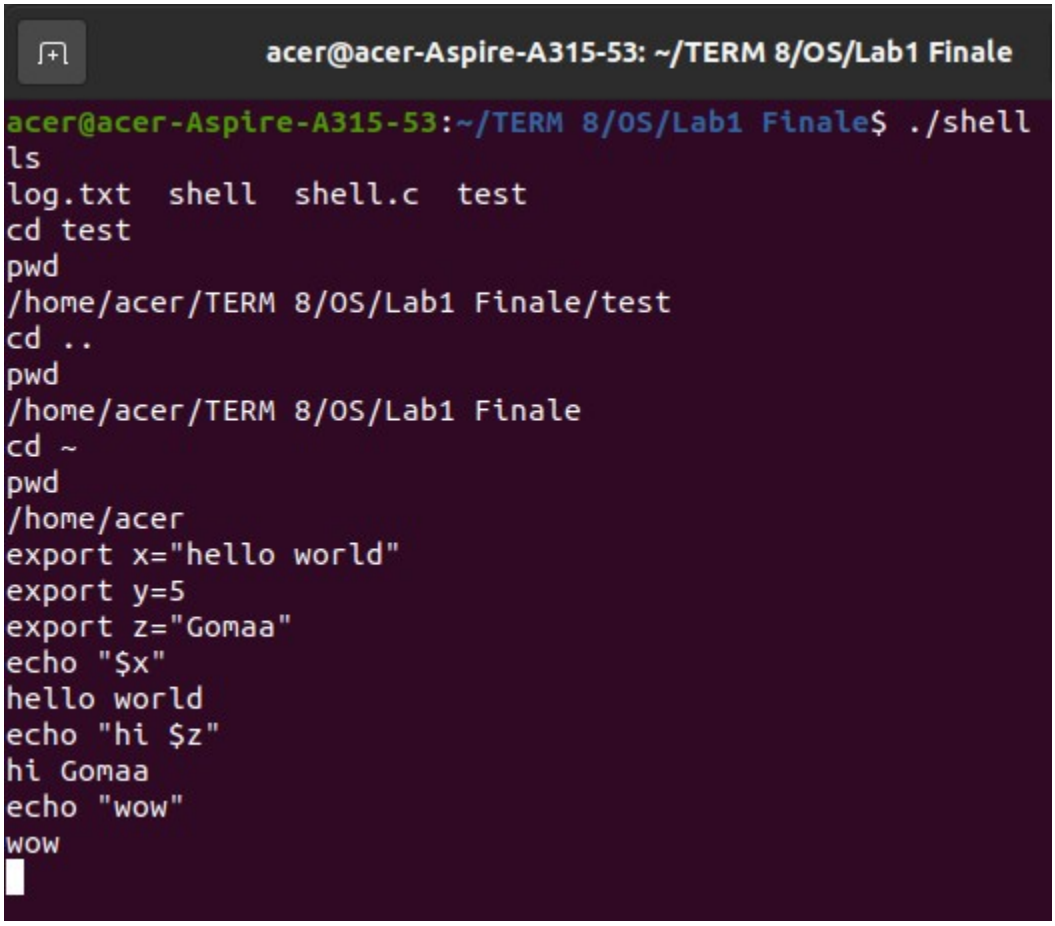

Fig.21