

Project Report: SimpleShell - A Basic Command Line Interpreter

Introduction:

- **Purpose:** The purpose of this project is to develop a simple command line interpreter (shell) in C that can execute basic commands, handle input/output redirection, and support command piping.
- **Objective:** To understand the fundamental concepts of process management, inter-process communication, and I/O redirection in Unix-like operating systems.

Background:

- A shell is a user interface for accessing an operating system's services. In Unix-like systems, the shell is a command-line interpreter that provides a user interface to the operating system's functions.
- The primary goal of this project is to replicate some of the basic functionalities of standard shells such as Bash.

Features:

- **Command Execution:** Executes standard commands available in the system.
- **Input Redirection:** Allows commands to read input from a file.
- **Output Redirection:** Allows commands to write output to a file.
- **Piping:** Supports the use of the pipe (|) to pass the output of one command as the input to another.

Design and Implementation:

System Overview:

- The shell reads a command from the user.
- It determines if the command involves redirection or piping.
- It then creates a child process to execute the command while the parent process waits for the child to complete.

Flowchart:

```
+-----+
|  Start  |
+-----+-----+
```

|
V

```
+-----+-----+
| Display Prompt |
+-----+-----+
```

|
V

```
+-----+-----+
| Read Command |
+-----+-----+
```

|
V

+-----+-----+
| Parse for Redirection |
| or Piping |
+-----+-----+
|
V

+-----+-----+
| Fork Process |
+-----+-----+
|
V

+-----+-----+
| Execute Command|
+-----+-----+
|
V

+-----+-----+
| Wait for Child |
+-----+-----+
|
V

+-----+-----+
| Loop to Start |
+-----+-----+

Code Explanation:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7
8  #define MAX_COMMAND_LENGTH 100 // Maximum length of a command
9
10 // Function to execute a command
11 void execute_command(char *command) {
12     char *args[MAX_COMMAND_LENGTH]; // Array to store command
arguments
13     int i = 0;
14
15     // Split the command into arguments using spaces
16     char *token = strtok(command, " ");
17     while (token != NULL) {
18         args[i++] = token; // Add each argument to the array
19         token = strtok(NULL, " ");
20     }
21     args[i] = NULL; // Mark the end of the arguments list
22
23     // Create a child process to run the command
24     pid_t pid = fork();
25
26     if (pid == 0) {
27         // Child process: Execute the command
28         execvp(args[0], args); // Run the command with arguments
29         perror("Error"); // If execvp fails, show an error
30         exit(EXIT_FAILURE); // Exit the child process with an error
```

```
31     }
32     else if (pid > 0) {
33         // Parent process: wait for the child to finish
34         wait(NULL);
35     }
36     else {
37         // If fork() fails, show an error
38         perror("Error");
39         exit(EXIT_FAILURE);
40     }
41 }
42
43 int main() {
44     while (1) {
45         // Keep the shell running forever
46         char command[MAX_COMMAND_LENGTH]; // Store the user's command
47
48         // Show the prompt
49         printf("SimpleShell> ");
50         fgets(command, sizeof(command), stdin); // Read the command
51         from the user
52
53         // Remove the newline character from the command
54         command[strcspn(command, "\n")] = '\0';
55
56         // Exit the shell if the user types "exit"
57         if (strcmp(command, "exit") == 0) {
58             break;
59         }
60
61         // Check for special characters in the command
62         char *output_redirect = strchr(command, '>'); // Check for
63         output redirection (>)
64
65         char *input_redirect = strchr(command, '<'); // Check for
66         input redirection (<)
```

```

63         char *pipe_operator = strchr(command, '|');    // Check for
piping (|)
64
65         if (output_redirect != NULL) {
66             // Handle output redirection
67             *output_redirect = '\0'; // Split the command at the '>'
68             char *output_file = strtok(output_redirect + 1, " "); //
Get the output file name
69             freopen(output_file, "w", stdout); // Redirect output to
the file
70             execute_command(command); // Execute the command
71             fclose(stdout); // Close the file
72         }
73         else if (input_redirect != NULL) {
74             // Handle input redirection
75             *input_redirect = '\0'; // Split the command at the '<'
76             char *input_file = strtok(input_redirect + 1, " "); // Get
the input file name
77             freopen(input_file, "r", stdin); // Redirect input from
the file
78             execute_command(command); // Execute the command
79             fclose(stdin); // Close the file
80         }
81         else if (pipe_operator != NULL) {
82             // Handle piping between two commands
83             *pipe_operator = '\0'; // Split the command at the '|'
84             char *first_command = command; // First part of the
command (before '|')
85             char *second_command = pipe_operator + 1; // Second part
of the command (after '|')
86
87             int pipefd[2]; // Create a pipe
88             pipe(pipefd);
89
90             pid_t pid = fork(); // Create a child process
91
92             if (pid == 0) {

```

```

93         // Child process: Run the first command
94         close(pipefd[0]); // Close the read end of the pipe
95         dup2(pipefd[1], STDOUT_FILENO); // Redirect output to
the pipe
96         close(pipefd[1]); // Close the write end
97         execute_command(first_command); // Execute the first
command
98         exit(EXIT_SUCCESS); // Exit the child process
99     }
100     else if (pid > 0) {
101         // Parent process: Run the second command
102         wait(NULL); // wait for the child process to finish
103         close(pipefd[1]); // Close the write end of the pipe
104         dup2(pipefd[0], STDIN_FILENO); // Redirect input from
the pipe
105         close(pipefd[0]); // Close the read end
106         execute_command(second_command); // Execute the second
command
107         fclose(stdin); // Close stdin
108     }
109     else {
110         // If fork() fails, show an error
111         perror("Error");
112         exit(EXIT_FAILURE);
113     }
114 }
115 else {
116     // If no special characters, just execute the command
117     execute_command(command);
118 }
119 }
120
121 return 0; // End of the program

```

Example Table:

Description	Input	Executes
List of files and Directories	<code>ls</code>	Shows all files and directories of the current path
Make New directory	<code>mkdir testdirectory</code>	Creates an empty directory named 'testdirectory'
Copy Paste	<code>cp text1.txt text2.txt</code>	Copies the content of 'text1.txt' to 'text2.txt'
Create empty file	<code>touch text1.txt</code>	Creates an empty file of .txt extension named 'text1'

Step-by-Step Explanation:

1. Include Necessary Libraries: These provide essential functions for input/output, string manipulation, process control, and inter-process communication.

2. Define Constants: `MAX_COMMAND_LENGTH` sets the maximum length for commands.

3. Function to Execute Commands:

- **Tokenize the Command:** Split the input command into arguments.
- **Fork a New Process:** Create a child process to execute the command.
- **Execute the Command:** Use `execvp` to run the command in the child process.
- **Parent Process:** Wait for the child process to finish.

4. Main Function Loop:

- **Display Prompt and Read Command:** Continuously prompt the user for a command and read the input.
- **Check for Exit Command:** Break the loop if the user types "exit".
- **Check for Redirection and Piping:** Use `strchr` to check for `>`, `<`, or `|` in the command.
- **Handle Output Redirection:** Redirect `stdout` to a file if `>` is found.
- **Handle Input Redirection:** Redirect `stdin` from a file if `<` is found.

- **Handle Piping:** Create a pipe and fork processes to handle the output of one command as the input to another.

- **Execute Normal Commands:** If no redirection or piping is found, execute the command normally.

5. Testing and Validation:

- **Normal Command Execution:** Commands like ``ls``, ``pwd``, ``echo`` were tested.

- **Input/Output Redirection:** Commands like ``cat < input.txt`` and ``echo "Hello" > output.txt`` were tested.

- **Piping:** Commands like ``ls | grep txt`` were tested to ensure piping works.

6. Limitations:

- The shell does not support background process execution (e.g., using ``&``).

- It has limited error handling and does not support complex shell features like scripting, command history, or advanced job control.

Conclusion:

- This project provided valuable insights into how basic shells work.

- The implementation covered fundamental concepts like process management, I/O redirection, and inter-process communication.

This project report outlines the development and functionality of a simple shell, providing a clear understanding of its features, design, implementation, and testing.