SIMPLE SHELL

The shell program allows users to execute commands, including support for piping between commands and background execution. It tracks command history, capturing the command, process ID, start/end time, and execution duration. Commands are run using `fork()` and `execvp()`, and pipes are managed to handle multiple command streams. The shell provides basic functionalities like viewing history and terminating the session.

---

1. Command History Structure and Globals

```
typedef struct CommandHistory {
    char command[MAX_COMMAND_LENGTH];
    pid_t pid;
    struct timeval start_time;
    struct timeval end_time;
} CommandHistory;

CommandHistory history[MAX_HISTORY];
int history_count = 0;
```

Explanation:

- `CommandHistory` is a structure used to store information about each executed command.
  - `command`: Stores the actual command string.
  - `pid`: Stores the process ID (PID) of the process that ran the command.
  - `start_time` and `end_time`: These `struct timeval` fields store the start and end times of the command execution.
- `history[]`: This is an array that holds the history of executed commands. The size is limited by `MAX_HISTORY`.
- `history_count`: Tracks the current number of stored commands.

---

2. Reading User Input

```
char* read_user_input() {
    char* input = malloc(MAX_COMMAND_LENGTH * sizeof(char));
    if (!input) {
        perror("Unable to allocate buffer");
        exit(1);
    }
    if (fgets(input, MAX_COMMAND_LENGTH, stdin) != NULL) {
        size_t length = strlen(input);
        if (input[length - 1] == '\\n') {
            input[length - 1] = '\\0';
        } else {
```

```
            perror("Command Length Exceeded than 1024 characters");
            exit(1);
        }
        return input;
    } else {
        return NULL;
    }
}
```

Explanation:

- `malloc()`: Allocates a buffer of size `MAX_COMMAND_LENGTH` to store the user input.
- `fgets()`: Reads the input from `stdin` into the allocated buffer. It ensures that at most `MAX_COMMAND_LENGTH - 1` characters are read, preventing buffer overflow.
- **Handling Newline**: If the last character in the buffer is a newline (i.e., user pressed `Enter`), it is replaced with a null terminator (`'\\0'`).
- **Error Handling**: If the input exceeds `MAX_COMMAND_LENGTH`, an error is triggered using `perror()` and the program exits.

---

3. Creating and Running a Process

```
int create_process_and_run(char* command, int background) {
    pid_t pid;
    int status;
    struct timeval start_time, end_time;
    char* args[MAX_ARGS];

    parse_command(command, args);

    gettimeofday(&start_time, NULL);
    pid = fork();

    if (pid == 0) {
        if (execvp(args[0], args) == -1) {
            perror("execvp failed");
        }
        exit(EXIT_FAILURE);
    } else if (pid < 0) {
        perror("fork failed");
        return -1;
    } else {
        if (!background) {
            waitpid(pid, &status, 0);
        } else {
            printf("[Running in background] PID: %d\\n", pid);
        }
```

```
        gettimeofday(&end_time, NULL);

        if (history_count < MAX_HISTORY) {
            strcpy(history[history_count].command, command);
            history[history_count].pid = pid;
            history[history_count].start_time = start_time;
            history[history_count].end_time = end_time;
            history_count++;
        }
    }
    return 1;
}
```

Explanation:

- `parse_command()`: Parses the command string into individual arguments, which are necessary for executing the command.
- `gettimeofday()`: Captures the time before and after execution, allowing the program to calculate how long the command takes to run.
- `fork()`: Creates a new process. In the child process (when `pid == 0`), the command is executed using `execvp()`.
- `execvp()`: This system call replaces the child process with a new program (the command to be run). If this fails, `perror()` prints the error.
- `waitpid()`: In the parent process, this function waits for the child process to terminate, unless the command is running in the background (denoted by the `background` flag).
- **Storing History**: After the process finishes, its details (command, PID, start time, end time) are saved to the history array.

---

4. Launching a Command

```
int launch(char* command) {
    int background = 0;

    int length = strlen(command);
    if (command[length - 1] == '&') {
        background = 1;
        command[length - 1] = '\\0';
    }

    return create_process_and_run(command, background);
}
```

Explanation:

- This function determines if the command should be run in the background by checking if the last character is an ampersand (`&`).

- If so, the `background` flag is set, and the `&` is removed from the command.
- Then, the command is passed to `create_process_and_run()` with the `background` flag indicating whether to wait for the command or not.

---

5. Command Parsing

```
void parse_command(char* input, char** args) {
    char* token = strtok(input, " ");
    int i = 0;
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " ");
    }
    args[i] = NULL;
}
```

Explanation:

- `strtok()`: This function splits the input command string into tokens (arguments) by spaces. It tokenizes the input into individual arguments and stores them in the `args` array.
- The last argument is set to `NULL`, which is required for `execvp()` to work.

---

6. Detecting Pipes

```
int has_pipe(char* input) {
    return strchr(input, '|') != NULL;
}
```

Explanation:

- `strchr()`: This function checks if the input command contains a pipe (`|`). If found, the function returns `1` indicating the presence of a pipe.

---

7. Executing Piped Commands

```
int execute_pipe_command(char* input) {
    char* commands[MAX_ARGS];
    char* args[MAX_ARGS];
    int pipe_fd[2], prev_pipe_fd[2];
    int num_commands = 0;
    pid_t pid;
    int status;

    char original_input[MAX_COMMAND_LENGTH];
    strcpy(original_input, input);

    char* token = strtok(input, "|");
```

```
    while (token != NULL) {
        commands[num_commands++] = token;
        token = strtok(NULL, "|");
    }

    for (int i = 0; i < num_commands; i++) {
        if (i < num_commands - 1 && pipe(pipe_fd) == -1) {
            perror("Pipe failed");
            return 0;
        }

        pid = fork();
        if (pid == 0) {
            parse_command(commands[i], args);

            if (i > 0) {
                dup2(prev_pipe_fd[0], STDIN_FILENO);
                close(prev_pipe_fd[0]);
                close(prev_pipe_fd[1]);
            }

            if (i < num_commands - 1) {
                close(pipe_fd[0]);
                dup2(pipe_fd[1], STDOUT_FILENO);
                close(pipe_fd[1]);
            }

            if (execvp(args[0], args) == -1) {
                perror("execvp failed");
                exit(EXIT_FAILURE);
            }
        } else if (pid < 0) {
            perror("Fork failed");
            return 0;
        }

        if (i > 0) {
            close(prev_pipe_fd[0]);
            close(prev_pipe_fd[1]);
        }

        if (i < num_commands - 1) {
            prev_pipe_fd[0] = pipe_fd[0];
            prev_pipe_fd[1] = pipe_fd[1];
        }
    }

    waitpid(pid, &status, 0);
```

```
    if (history_count < MAX_HISTORY) {
        strcpy(history[history_count].command, original_input);
        history[history_count].pid = pid;
        gettimeofday(&history[history_count].start_time, NULL);
        gettimeofday(&history[history_count].end_time, NULL);
        history_count++;
    }

    return 1;
}
```

Explanation:

- **Handling Multiple Commands**: This function handles piped commands by splitting the input string at each pipe symbol (`|`), using `strtok()`. The commands are stored in an array.
- **Pipe Creation**: For each command except the last one, a pipe is created using `pipe()`.
- **Process Creation**: Each command is executed in its own process created by `fork()`.
  - **Redirection**: `dup2()` is used to redirect input and output between commands via pipes. Input redirection from the previous pipe and output redirection to the current pipe are handled.
  - `execvp()` executes the command in the child process.
- **History**: After execution, the command and its details are stored in the history array.

---

8. Shell Loop

```
void shell_loop() {
    char* command;
    int status;

    do {
        printf("SimpleShell> ");
        command

 = read_user_input();

        if (strcmp(command, "history") == 0) {
            show_history();
        } else if (strcmp(command, "exit") == 0) {
            if (history_count < MAX_HISTORY) {
                strcpy(history[history_count].command, "exit");
                history[history_count].pid = getpid();
                gettimeofday(&history[history_count].start_time, NULL);
                gettimeofday(&history[history_count].end_time, NULL);
                history_count++;
            }
            show_execution_details();
```

```
            status = 0;
        } else if (has_pipe(command)) {
            status = execute_pipe_command(command);
        } else {
            status = launch(command);
        }

        free(command);
    } while (status);
}
```

Explanation:

- **Main Loop**: The shell runs in an infinite loop where it prompts the user for input (`SimpleShell>`).
- **Command Handling**: Depending on the input:
  - If the command is `"history"`, it displays the history of executed commands.
  - If the command is `"exit"`, it records the exit in the history and terminates the loop.
  - If the command contains a pipe (`|`), it executes the piped commands using `execute_pipe_command()`.
  - Otherwise, it launches a single command with `launch()`.
- **Memory Management**: The allocated command string is freed at the end of each iteration.

---

9. Displaying Command History

```
void show_history() {
    for (int i = 0; i < history_count; i++) {
        printf("%d: %s\\n", i + 1, history[i].command);
    }
}
```

Explanation:

- This function prints out the history of executed commands. Each command is prefixed with its index in the history.

---

10. Showing Execution Details

```
void show_execution_details() {
    printf("\\nExecution Details of Commands:\\n");
    for (int i = 0; i < history_count; i++) {
        struct timeval start_time = history[i].start_time;
        struct timeval end_time = history[i].end_time;

        double duration = (end_time.tv_sec - start_time.tv_sec) +
                          (end_time.tv_usec - start_time.tv_usec) / 1e6;
```

```
        printf("Command: %s\\n", history[i].command);
        printf("PID: %d\\n", history[i].pid);
        printf("Start Time: %ld.%06ld\\n", start_time.tv_sec, start_time.tv
_usec);
        printf("End Time: %ld.%06ld\\n", end_time.tv_sec, end_time.tv_use
c);
        printf("Duration: %.6f seconds\\n\\n", duration);
    }
}
```

Explanation:

- This function provides detailed execution statistics for each command stored in history.
- **Time Calculation**: The duration is calculated using the `tv_sec` (seconds) and `tv_usec` (microseconds) fields of the `start_time` and `end_time`.
- It then prints the command, its PID, start and end times, and execution duration.

---

11. Main Function

```
int main(int argc, char **argv) {
    shell_loop();
    return EXIT_SUCCESS;
}
```

Explanation:

- **Main Program Flow**: The program starts by calling `shell_loop()`, which manages the user interaction and command execution. When the user exits the shell, the program returns `EXIT_SUCCESS`, indicating successful termination.

LIMITATIONS OF SHELL

In the design of this shell program, certain commands will not be supported due to the following limitations:

1. **Shell Built-ins (e.g., `cd`, `export`, `source`)**:
    - These commands require modifications to the shell's own environment, such as changing directories or updating environment variables. Since `execvp()` spawns child processes, changes like `cd` do not persist in the parent shell.
2. **Commands Involving Job Control (e.g., `fg`, `bg`, `jobs`)**:
    - The shell does not implement full job control, which involves tracking multiple background jobs, suspending, and resuming processes. This would require more advanced process handling, including signals and process groups.
3. **Input Redirection and Output Redirection (e.g., `>` or `<`)**:

- The current implementation does not include support for input/output redirection, which would require parsing the command for redirection symbols and manipulating file descriptors accordingly.

4. **Complex Piping (e.g., `command1 | command2 | command3` with conditional execution)**:
   - While basic piping is supported, more complex command chains (like conditional execution `&&`, `||`) require advanced parsing and control flow mechanisms that are not currently implemented.
5. **Scripting Features (e.g., loops and conditionals in scripts)**:
   - This shell does not support scripting constructs like `if`, `for`, `while`, or `case` statements found in shell scripts. Implementing these would require designing a parser and interpreter for shell syntax, which is beyond the scope of this simple shell.
6. **Wildcard Expansion (e.g., `ls *.txt`)**:
   - The shell does not handle wildcard expansion (globbing) like the standard Bash shell. This would require integrating a pattern-matching mechanism, which is typically handled by the shell before passing arguments to commands.
7. **Subshell Execution (e.g., `command1 && command2`, `command1 || command2`)**:
   - The shell lacks support for complex command chaining with logical operators (`&&`, `||`), which would require enhanced parsing logic to handle conditional execution based on exit status. This would need more advanced control flow handling and error checking.
8. **File Descriptor Manipulation (e.g., `2>&1` or `>&file`)**:
   - This shell does not support advanced file descriptor manipulation, such as redirecting error output or merging standard output with error output. Implementing this would require significant handling of file descriptors beyond simple piping and redirection.
9. **Background Job Termination (`kill` or `disown`)**:
   - While the shell supports background execution with `&`, it does not have built-in functionality for managing or terminating background jobs using commands like `kill` or `disown`. This would require a robust job control system to track and manipulate background processes after they have been launched.
10. **Aliases (e.g., `alias ll='ls -l'`)**:
    - The shell does not support the creation or management of command aliases. This would require maintaining an alias table and expanding them during command parsing, which adds complexity to input processing.
11. **Environment Variable Manipulation (e.g., `export`, `set`, `unset`)**:
    - Although external commands can access environment variables, this shell does not support direct manipulation of the shell's own environment. Environment variables would need special handling to update the shell's environment space, which isn't feasible in a child process model.
12. **Inline Command Substitution (e.g., `$(command)` or backticks `` `command` ``)**:
    - The current shell lacks the ability to substitute the output of one command directly into another using inline command substitution. This would require parsing and executing nested commands, which significantly complicates command parsing and execution.

CODE FLOW

CONTRIBUTIONS

1. Priyanshu sharma /2023408
   + Code logic
   + Comment and explanation
2. Vansh tyagi / 2023582
   + Code logic
   + Documentation

GITHUB REPO

https://github.com/Priyanshu0phc/OS_66