Implementation of Shell:

1. **Header Files**: The code includes several standard C library header files such as `<stdio.h>`, `<unistd.h>`, `<string.h>`, `<stdlib.h>`, `<sys/wait.h>`, `<time.h>`, and `<signal.h>`. These headers are necessary to access various system functions and data types used throughout the code.

2. **History Storage**: The shell maintains a history of executed commands using a custom data structure called `struct Node`. This structure includes four fields:

- `command`: Stores the actual command string.
- `pid`: Represents the process ID (PID) of the executed command.
- `execution_time`: Records the time when the command was executed.
- `duration`: Indicates the duration of command execution.

This command history is stored in an array of `struct Node` called `history`, which can hold up to `MAX_HISTORY_SIZE` (256) entries.

3. **User Input**: The `read_user_input` function is responsible for reading user input from the command line. It does so by calling `fgets` to capture the user's input into a character array. Then, it dynamically allocates memory to store a copy of the input and returns a pointer to this copy. This ensures that the original input is not modified and can be freed later to prevent memory leaks.

4. **Command Parsing**: The `parse_input` function is responsible for parsing the user's input command. It counts the number of pipes ('|') in the command to determine if it involves piping. This function tokenizes the input command using `strtok` and stores the tokens in an array named `args`. If there are pipes in the command, it triggers the execution of piped commands; otherwise, it returns `0`, indicating no pipes were found.

5. **Piping**: If the command involves pipes, it is handled by two functions:

- `implement_pipe` for single pipes: This function creates two child processes. The first child's output is redirected to the second child's input using pipes, allowing data to flow from one command to the next.

- `execute_multi_pipes` for multiple pipes: This function handles commands with multiple pipes by creating multiple child processes connected via pipes to implement complex pipelines.

6. **Command Execution**: The `create_process_and_run` function is responsible for executing
non-piped commands. It forks a child process using `fork` and then uses `execvp` to replace the child process with the specified command. This function also records the executed command's history, including its start time, duration, and PID.

7. **Shell Loop**: The `display` function implements the main loop of the shell. It continuously prompts the user for input, reads the input using `read_user_input`, and then parses and executes the commands using the functions described above. If a command involves piping, the shell handles it gracefully. The loop continues until the user exits the shell by entering an appropriate command (e.g., 'exit').

8. **Signal Handling**: The code sets up a signal handler for the `SIGINT` signal (Ctrl+C) using the `sigaction` function. When the user interrupts the shell (e.g., by pressing Ctrl+C), the signal handler is triggered. It loads the command history, displays the history to the user, and then exits the shell gracefully.

9. **Loading and Saving History**: The `load_history` function loads command history from a file
named "history.txt." It reads the file line by line, extracts information about executed commands,
and populates the `history` array. The `save_history` function saves the current command history back to the "history.txt" file after each executed command.

10. **Main Function**: The `main` function serves as the entry point for the shell. It initializes the
`SIGINT` signal handler, loads the command history, and enters the main shell loop by calling the `display` function. The shell keeps running until the user decides to exit it.

Limitation:

● Our shell relies on the availability of executable files located within the 'bin' directory to run commands. This means that any command issued to the shell must have its corresponding executable file in the 'bin' folder for our shell to execute it successfully. For certain commands, such as 'cd' (change directory), there is no corresponding executable file present in the 'bin' directory. As a result, our shell is unable to execute these commands, as it lacks the necessary executable to do so.

● No Shell Scripting: There's no support for shell scripting or batch execution of commands. Our shell code is primarily designed to parse and execute simple command strings entered by the user. Shell scripts often contain more complex constructs like variables, loops, and conditionals. Executing a script would require more advanced parsing and handling of these constructs.

● Script File Execution: To execute a shell script, we need to read the script file from disk, parse its contents, and execute it line by line. Our code currently reads user input from the command line and doesn't have the logic to read and execute script files.