Assignment 2: Implementing a Simple Shell with Piping in C

- **Due:** 9th October

Objectives

- Learn how to use system calls like `fork()`, `execvp()`, `pipe()`, and `wait()`.
- Understand how to create processes and manage their execution in C.
- Explore inter-process communication using pipes.
- Implement basic shell functionality with support for built-in commands, piping, and error handling.

Requirements

You will implement a simple shell in C, capable of:

1. Reading and parsing user input.
2. Handling built-in commands such as cd and exit.
3. Executing external commands using `fork()` and `execvp()`.
4. Supporting inter-process communication through piping `(|)`.
5. Handling command failures and printing appropriate error messages.

Tasks

1. Set Up the Project

- Create a new file named `myshell.c`.
- Include the necessary headers: `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<unistd.h>`, `<sys/wait.h>`, and `<fcntl.h>`.

2. Implement the Main Shell Loop

- Implement an infinite loop that continuously prompts the user to enter commands.
- Use `fgets()` to read input from the user, storing it in a buffer.
- Parse the input to remove the newline character and handle empty commands.
- Add a feature that breaks the loop when the exit command is entered.

Question 1:
What is the purpose of `fgets()` in this shell, and why is it preferred over `scanf()`?

---

3. Command Parsing

- Use the `strtok()` function to split the input into command arguments, and store them in an array.
- Ensure the argument array is null-terminated for proper execution in `execvp()`.

Question 2:
Why is it necessary to null-terminate the arguments array before passing it to `execvp()`?

---

4. Execute Commands

- Implement the execution of commands using `fork()` and `execvp()`.
    - In the child process, use `execvp()` to execute the command.
    - In the parent process, wait for the child process to finish using wait().

Question 3:
Explain the difference between `fork()` and `execvp()` in process creation and execution.

---

5. Built-in Commands (cd and exit)

- Implement the cd command using `chdir()`.
    - If cd is followed by a directory, change to that directory.
    - If cd is called without arguments, print an error.
    - Print error messages if the directory cannot be changed.
- Ensure the shell exits when the exit command is entered.

Question 4:
Why does the cd command need to be handled within the parent process, rather than the child process?

---

6. Piping Between Commands

- Detect if the user input contains a pipe `(|)`.

- If a pipe is found, split the input into two commands.
- Set up a pipe using `pipe()`, and then fork two child processes:
  - The first process executes the first command and writes its output to the pipe.
  - The second process reads from the pipe and executes the second command.
- Ensure that the pipe's file descriptors are closed appropriately in both processes.

Question 5:
Explain how pipes enable communication between two processes. What happens to data written to the pipe if no process is reading from it?

---

7. Error Handling

- Implement proper error handling throughout the shell:
  - If `fork()` fails, print an error and return to the shell prompt.
  - If `execvp()` fails, print an error message informing the user that the command is not recognized.
  - If there is an issue with `chdir()`, display an appropriate message to the user.

Question 6:
Why is error handling critical in a shell, and what could happen if errors from `fork()` or `execvp()` are not properly handled?

---

8. Test the Shell

- Test your shell by running both simple and complex commands:
  - Basic commands like `ls`, `pwd`, and `date`.
  - Piped commands like `ls | grep .c` or `cat file.txt | wc -l`.
  - Built-in commands like `cd ..` and `exit`.

Question 7:
What is the expected output when running the command `cat file.txt | wc -l`? Explain how the pipe works between cat and `wc`.

Submission Guidelines

- Submit your `myshell.c` file along with a `README` files via **GitHub** containing:
    - Instructions on how to compile and run your shell including sample commands and their expected outputs as a `README_Run` file.
    - Answers to the posted questions as a `README_Answers` file.

Evaluation Criteria: (7pts)

- **Functionality (5 points):** Does the shell correctly handle basic commands, built-in commands, and piped commands?
- **Code Quality (1 points):** Is the code well-structured, commented, and follows naming conventions?
- **Error Handling (1 points):** Does the shell handle invalid commands, process creation errors, and execution failures gracefully?

Hints:

- **Forking and Waiting:** Remember that after you call fork(), the parent and child processes run in parallel. The parent should wait for the child to finish before prompting for another command.
- **Piping:** When working with pipes, you need to redirect the standard input/output of one process to communicate through the pipe.
- **Testing:** Start with simple commands and then move on to more complex cases involving pipes and multiple arguments.