OS Mini-Project

Deadline: 20th November 2025 - 11:59 PM

The objective of this assignment is to develop a custom interactive shell program in C.

Please be aware that plagiarism will be closely monitored for this assignment, so do not copy code from others!

Specification 1: Display Requirement [10 marks]

When you execute your code, a shell prompt of the following form must appear. You **must not** hard-code the username and the system name here.

Format: `<username@system_name:curr_dir>`

Example: `<Name@UBUNTU:~/test>`

The directory from which the shell is invoked will be the **home directory** of the shell and should be indicated by `~`. If the user changes the directory, the corresponding change must be reflected in the shell prompt. For example, if the home directory is `/home/user` and the current directory is `/home/user/test`, the prompt should show `~/test`.

Specification 2: Built-in Commands [30 marks]

Your shell must support special commands that are part of the shell itself. When these commands are used, they must run directly within your shell program.

Crucial Requirement: For this specification, you **cannot** use system calls that execute other programs (like the `exec` family) to implement these *built-in* commands. You must write the logic for them yourself.

ECHO (5 Marks)

The `echo` command displays the text given as arguments.

- You do not need to handle escape flags, quotes, or environment variables.
- You must handle tabs and spaces. If a user inserts multiple tabs or spaces between words, they should be treated as a **single space** in the output.

- **Example Input:** `echo all the best`
  Expected Output: `all the best`

PWD (5 Marks)

The `pwd` (Print Working Directory) command prints the full, absolute pathname of the current working directory.

- **Example Input:** `pwd`
  Expected Output: `/home/user/logs`

CD (10 Marks)

The `cd` (Change Directory) command allows the user to change the current working directory. You must implement the following cases:

- `cd` (with no arguments): Changes the directory to the shell's home directory.
- `cd ~`: Also changes the directory to the home directory.
- `cd ..`: Shifts one level up in the directory structure.
- `cd -`: Goes to the *previous* directory. This must also **print** the path of the directory you are moving to. If there is **no previous directory yet** (first use), print an appropriate error message like *"No previous directory."*
- `cd [directory_path]`: Changes to the specified directory.
- **Error Handling:** If `cd` has more than one argument, it is an error. You must print an appropriate error message.

HISTORY (10 Marks)

You must implement a `history` command that tracks commands.

- It must store the last **20** commands.
- When `history` is typed, it must display the last **10** commands entered.
- The history must be persistent; it should track commands across all sessions by saving them to a file named **inside the shell's home directory** (`~`)
- You must overwrite the oldest commands if more than 20 are entered.
- **DO NOT** store a command in history if it is exactly the same as the *immediately* preceding command.

Specification 3: Process Management & External Commands [50 marks]

This is the primary function of a shell. If the command entered by the user is **not** one of the built-in commands (cd, pwd, echo, history), your shell must execute it as an **external program**.

Foreground Process Execution (15 Marks)

Your shell must be able to run any external command (e.g., `ls -l`, `gcc`, `grep "foo" file.txt`).

- To do this, your shell program must create a **new, separate child process** to run the external command.
- Your main shell program (the **parent** process) must **wait** for this child process to finish running its command before you display the shell prompt again. Only one foreground command may run at a time.

Background Process Execution (10 Marks)

Your shell must support running commands in the background, which is indicated by a `&` at the end of the command.

- **Example:** `sleep 3 &`
- When a `&` is detected, your shell must create a new child process as normal.
- However, the parent process (your shell) must **NOT** wait for the child to finish. It should immediately return to the prompt and be ready for the user to type a new command.

I/O Redirection (15 Marks)

Your shell must support input (`<`) and output (`>`) redirection for external commands.
These may appear **individually or together** in the same command.

- **Example:** `grep "hello" < input.txt > output.txt`
- This means you must tell the new child process that its "standard input" is not the keyboard, but `input.txt`, and its "standard output" is not the screen, but `output.txt`.

- This involves manipulating the process's open file streams before the new command is run.

Signal Handling (10 Marks)

A robust shell must handle signals from the operating system.

- `Ctrl+C`: The shell (parent process) must not terminate when this signal is pressed. It should catch the signal, print a new prompt, and terminate **only** the current foreground child process if one is running. Background processes **must not** be affected.
- `Ctrl+D`: Pressing `Ctrl+D` on an empty line indicates "End of File" and should cause your shell to exit cleanly.
- **Background Process Cleanup:** When a background process finishes, it notifies its parent. Your shell must be able to receive this notification and properly "clean up" the finished process so it doesn't remain as a "zombie" process in the system.

Specification 4: Modularity & Makefile [10 marks]

Your code must be well-structured, modular, and easy to read.

- **Modularity:** You must split your code into logical, separate `.c` and `.h` files for different features (e.g., `cd.c`, `display.c`, `execute.c`, etc).
- **Makefile:** You must provide a `makefile` that has, at a minimum, an `all` target to compile your shell and a `clean` target to remove the executable.

Guidelines

- The assignment must ONLY be done in **C**.
- All C standard library functions are allowed unless explicitly banned. Third-party libraries are not allowed.
- If a command cannot be run or returns an error, it should be handled appropriately. Look at `"perror.h"` for routines to handle system call errors.
- If your code does not compile, no marks will be awarded.
- Segmentation faults at the time of grading will be penalized.

Submission Format

1. Upload a compressed file: `<Roll No>_OSMiniProject.tar.gz`
2. The directory structure must be:

```
<Roll-No>_OSMiniProject/
|-- README.md
|-- makefile
|-- ... (all other .c and .h files)
```
