OS Mini-Project Help Document

How to Get Started

This document provides conceptual hints to help you get started with the shell assignment. You should refer to `man` pages to find the specific C functions you need.

Part 1: The Shell Prompt (Display)

Your main goal here is to print a prompt like `<username@system_name:current_dir>`.

- **Username & System Name:** You'll need to find the right C functions to get this information. There are built-in functions to get the system's "host name" and to get information about the current user.
- **Current Directory:** The OS provides a function to get the **c**urrent **w**orking **d**irectory.
- **Making it Relative (`~`):** The `~` part is a feature you must implement. When your shell first starts, save the path of its home directory. Then, every time you update the prompt, check if the current directory path *starts with* that saved home directory path. If it does, you can replace that part of the string with `~`.

Part 2: Built-in Commands

For `cd`, `pwd`, `echo`, and `history`, you must implement the logic yourself.

- **Parsing Input:** Before running any command, you need to parse the user's input string. The `strtok()` function is very useful for splitting a string into "tokens" based on delimiters like spaces and tabs.
- **`pwd`:** After tokenizing, if the first token is `"pwd"`, you just need to call the C function that gets the current working directory and print it.
- **`echo`:** If the first token is `"echo"`, you need to loop through all *other* tokens and print them, with a single space in between.
- **`cd`:** If the first token is `"cd"`, check the *second* token. The OS provides a function to **ch**ange the **dir**ectory. You'll need to give it the correct path (like

`..`, the home directory, or another path). For `cd -`, you must remember what the previous directory was.

- `history`: This requires file I/O. You'll need to decide on a file (e.g., `history.txt`) to store commands. You'll need functions to open, read, and write to this file. Think about how to manage an array in your program to hold the 20 most recent commands.

Part 3: External Commands & Processes

This is for all commands that are *not* built-in.

- **Foreground:** Your shell needs to create a new, separate **process** to run the command. The operating system provides a way to "duplicate" your shell process (creating a parent-child relationship).
  - The **child** process's job is to *replace itself* with the new program (like `ls` or `grep`).
  - The **parent** process's job is to **wait** for the child to finish its work before continuing.
- **Background:** This is conceptually similar, but the **parent** process (your shell) does *not* wait for the child. It just lets the child run on its own and immediately returns to the prompt.
- **I/O Redirection:** Before the child process runs the new program, it can be configured. The OS allows you to change what "standard input" and "standard output" *mean* for that process. Instead of pointing to the terminal, you can make them point to files you open.
- **Signal Handling:** Your shell is a process, so it can receive signals.
  - You can tell the OS to "catch" certain signals (like `Ctrl+C`) and run a special function you define, instead of just quitting.
  - For background processes that finish, they send a specific "child" signal to the parent. Your shell needs to catch this signal to perform cleanup (this "cleanup" is also a form of waiting).

Some Useful Commands (Read man pages)

This is a list of commands and C-related topics to read about in the `man` pages. They will be very useful.

`gethostname` , `getpwuid` , `getcwd` , `chdir` , `strtok` , `fopen` , `fgets` , `fprintf` , `fork` , `execvp` , `waitpid` , `signal` , `sigaction` , `open` , `close` , `dup2` , `perror` , `SIGINT` , `SIGCHLD` , `WNOHANG`

Type: `man <command_name>` or `man 2 <command_name>` to learn about them.

ALL THE BEST WITH THE ASSIGNMENT!