Simple Shell Project

Project Overview

In this project, a simple shell program will be developed that can start and manage background processes. The shell will interact with the system's kernel using system calls, enabling it to execute commands, handle processes, and manage background tasks. This exercise aims to solidify the understanding of system programming concepts and how shells interact with operating systems through the system call interface.

In this report, I will outline the tasks completed during the development of the simple shell project. Initially, I forked the project with template files provided by the professor. I changed the files when I feel necessary to do it. The file `lab.h` contains the method definitions, while the methods are implemented and overridden in the corresponding `lab.c` file.

Before implementation

The provided command line prompts should be parsed before passing the cmd line to any functions. The **cmd_parse function is implemented in flowing way:

1. cmd_parse() function

   This function parses a given command line string into an array of strings (tokens) representing individual arguments. It first checks if the input string is valid, allocates memory for the arguments array, and tokenizes the input based on spaces, tabs, and newline characters. If any memory allocation fails, it frees the allocated memory and return **NULL**. Finally, it returns the array of arguments, with the last element set to NULL for termination.

2. The memory should be freed after utilization. Unless the the address sanitizer issue may be encountered. The cmd_free() should be implemented for handling the scenario.

3. Trims any leading or trailing whitespace from the input implementing the function **trim_white()**.

1. **Task 3 - Print Version**

   Requirements: When the shell is initiated with the command-line argument -v, it outputs the project's version information and subsequently terminates.

   Design and Implementation

   void parse_args(int argc, char **argv):
   This function processes command-line arguments passed to the shell using the getopt() function. It looks for the -v option. If the -v flag is found, it prints the shell version using predefined constants lab_VERSION_MAJOR and lab_VERSION_MINOR. If an unrecognized option is encountered, an error message is printed, and the program exits with a failure status (EXIT_FAILURE). The shell quits after showing the version number.

   Library Function: **getopt()**

   Result

2. **Task 4 - User Input**

   Requirements: The shell should be able to write the prompt by the user.

   Design and Implementation

   The C language has some build in function for handling this. Instead of using a function like **scanf** to get input from the user, we can take a more robust approach by leveraging the GNU **Readline** library. The GNU **Readline** library gives us control over the input line and adds useful features that let users edit the line, use the TAB key for filename completion, and navigate through command history with the arrow keys. To use the readline functions, we need to include the headers **readline/readline.h** and **readline/history.h.** I followed the documentation for the Readline.

   The implementation continuously reads input from the user using the readline() function until the input is NULL. For each line: line = readline(terminal.prompt): Prompts the user for input and stores it in line. line = trim_white(line): Trims any leading or trailing whitespace from the input. if (strlen(line) == 0): Checks if the trimmed line is empty (i.e., the user entered only whitespace). If the line is empty, it prints a message, frees the memory

allocated for the line, and skips the rest of the loop using continue. This structure ensures that the shell can handle user input effectively, even when the input is just whitespace.

Library Function: **readline()**

Result

3. Task 5 - Custom Prompt

Requirements: The name of the shell can be given as we line.

Design and Implementation
The shell uses a default prompt, but it checks for the environment variable MY_PROMPT. If this variable is set, the shell uses its value as the prompt. You can set the environment variable inline, like **MY_PROMPT="foo>" ./myprogram**, to quickly test your program with a custom prompt. Here, **MY_PROMPT** is an env variable.

Library Function: **getenv()**

Result

4. **Task 6 - Built in Commands**

Requirements: The shell should handle some built in unix commands like exit, change directory (cd), pwd, history, jobs, etc.

Design and Implementation

1. **Exit**

   The shell should quit when provide exit command instead of hardly quit using control c. It first checks if the user's input, stored in argv[0], matches the string "exit" by using the strcmp function. If the two strings are identical, meaning the user intends to exit the

shell, the program proceeds to clean up the shell's resources by calling the sh_destroy(sh) function. This function is likely responsible for releasing any memory or closing open files associated with the shell. Once the cleanup is complete, the program terminates by calling exit(0), where the argument 0 indicates that the program has successfully exited. This ensures that the shell shuts down gracefully when the user types "exit".

Library Function: Exit()

Result

2. **cd**

The implementation checks if the user input, stored in argv[0], is the command "cd", which is used to change directories in a shell. If the condition is true, it calls the change_dir(argv) function to attempt to change the current working directory based on the arguments provided. The change_dir function attempts to change the current working directory based on the provided arguments in the dir array. If no directory is specified (i.e., dir[1] is NULL), the function first tries to retrieve the user's home directory using the getenv function. If that fails, it obtains the home directory from the user's password entry via getpwuid(). If a directory is specified, the function uses that path instead. The function then calls chdir(path) to change the directory. If the change is successful, it prints a confirmation message and returns 0. If an error occurs, it prints an error message detailing the failure and returns -1.

Library Function: chdir(), getenv(), getuid(), getpwuid()

Result

3. **pwd**

If the user's input command argv[0] is "pwd", it fetches the current working directory. If this condition is met, it declares a character array cwd with a size defined by PATH_MAX to hold the directory path. The function then calls getcwd(cwd, sizeof(cwd)) to retrieve the current working directory. If the retrieval is successful, it prints the current directory path and returns true. If an error occurs during the call to getcwd, it prints an error message detailing the issue using strerror(errno) and returns false.

Library Function: getcwd()

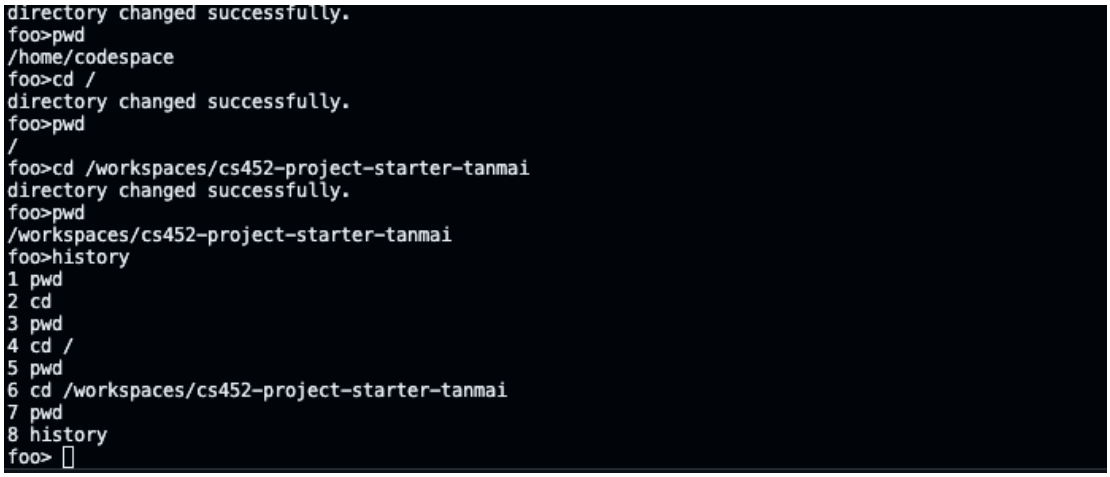
Result: Please check the previous screenshot.

4. **history**

The user can see the commands entered in a session using the history command. If the user provides **history** prompt, it retrieves the history entries by calling history_list(), which returns a pointer to an array of HIST_ENTRY structures. If the history list is successfully retrieved, the function iterates through the entries, printing each command along with its corresponding index (adjusted by history_base) to the console. If no history entries exist, it outputs a message indicating "No History." Finally, the function returns true to signal successful execution.

Library Functions: history_list()

Result



5. Task 7 - Create a Process

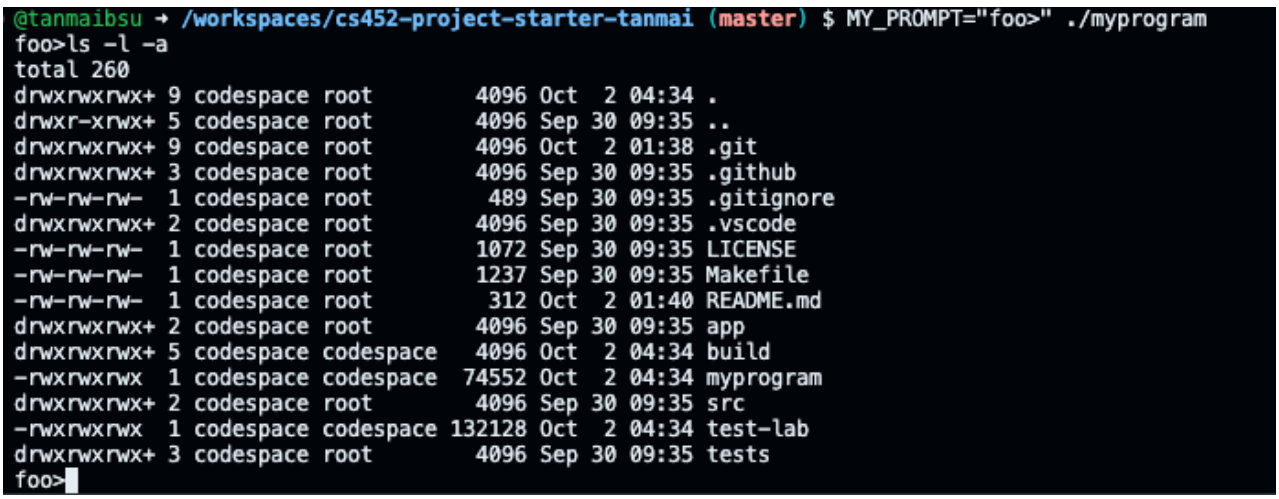
Requirements: The shell should create a process.

Design and Implementation

The shell creates a new process and waits for its completion, accepting one command per line with arguments, limited to at least ARG_MAX arguments. It uses the sysconf system call with _SC_ARG_MAX to determine the maximum number of arguments that the exec family of functions can handle. After parsing the entered line, the shell attempts to execute the command using execvp, which searches for the command via the PATH environment variable, simplifying the programming task by eliminating the need to locate the command manually. Assuming all command-line arguments are separated by spaces, the shell parses input accordingly. For example, the command ls -l -a would be parsed as an array of size 3: ls, -l, and -a. If the user enters ls "-l -a", it would parse to ls, "-l, and -a. Users can press the Enter key without typing any commands, or enter only spaces, and the shell will simply display another prompt without causing segmentation faults or memory leaks for empty commands.

Library Functions: execvp, fork(), waitpid()

Result



6. **Task 8 - Signals**

Requirements: The shell should be able to ignore specified signal.

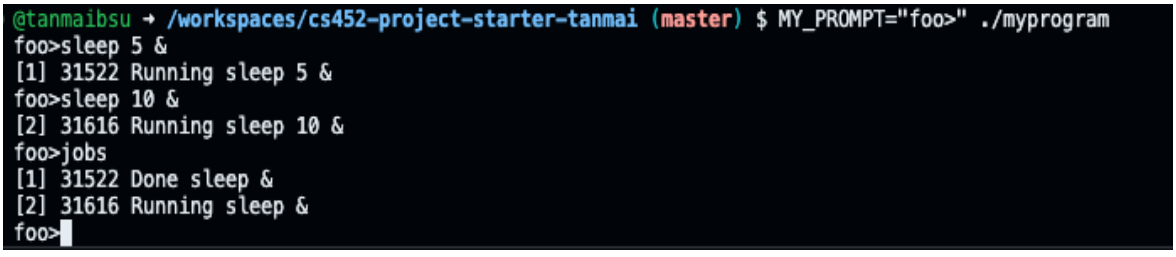
Design and Implementation

When no process is executing, the shell simply displays a new prompt and ignores any input on the current line. Use the tcgetpgrp and tcsetpgrp system calls to get and set the foreground process group of the controlling terminal, along with the signal system call to manage signals. While the glibc manual provides examples for a complete job control shell, you are not required to implement one with the same level of functionality. Use the documentation as a reference, but ensure you understand the code rather than just copying and pasting examples. You are free to incorporate code from the manual as long as you grasp its purpose and functionality.

7. **Task 9 - Background Processes**

Requirements: The shell should be able to run background process. Your shell can start a background process by placing an ampersand (&) at the end of the command line. For each background process initiated, the shell prints the job number, process ID (pid), and the full command provided by the user. After starting a background process, the shell promptly returns to the user with a new prompt, allowing for immediate input without waiting for the background process to finish. Users do not need to separate the & from the command with a space; both date & and date& are valid, and additional spaces after the ampersand are acceptable. The mini shell tracks running background processes, assigning and displaying a job number, the associated process ID, and the full command (including the ampersand).

Result

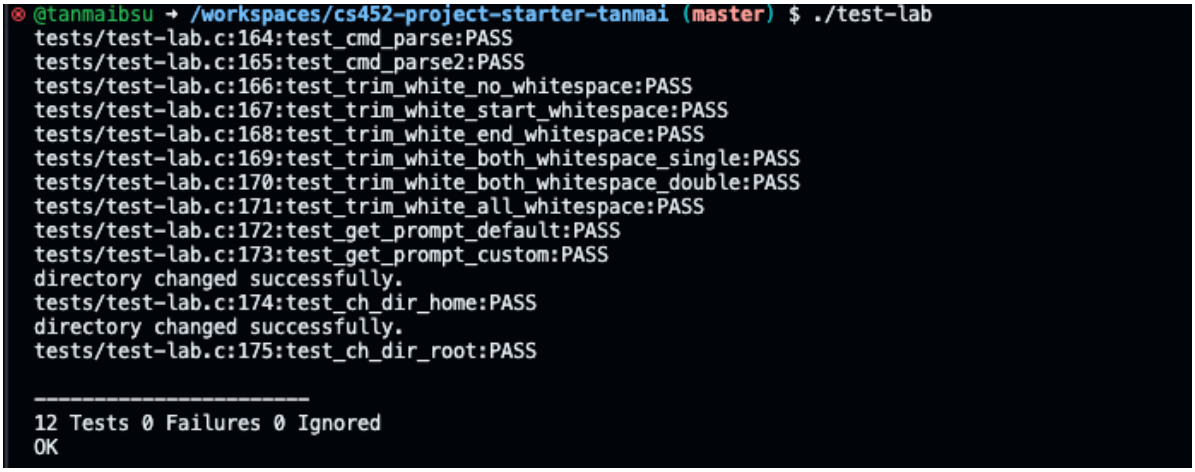


8. **Task 10 - Jobs command**

Requirements: We need to add a new built-in command jobs that prints out all the background commands that are running or are done but whose status has not yet been reported.

Result: See the previous screenshot.

Test Result



Known Issues

1. **AddressSanitizer:DEADLYSIGNAL: Please use control c when you face it (rare case.)**
2. **In test results, you may face leaksanitizer issue all tests have been passed.**