CSE 3033 - OPERATING SYSTEMS
Programming Assignment # 2 - Project Report

Umut Emre Önder - 150119018
Batuhan Baştürk - 150119035

Title: Implementation of a Simple Shell

Purpose: In this report, the design and implementation of a straightforward shell, a command-line interface for interacting with the operating system, are documented. A shell reads the user's input, analyzes it, and then runs the requested command. Several built-in commands for managing the shell environment are supported by the basic shell that was implemented in this project, in addition to the execution of external commands.

Scope: The scope of this project is limited to the implementation of the shell, including the handling of built-in commands and the execution of external commands.

Design: The shell was implemented in C, a general-purpose programming language that is widely used for system programming, including the development of operating systems and shells. The following functions and libraries were used in the implementation:

- **read() and write() from unistd.h**: The standard input/output is read from and written to using these functions, respectively. The write() function writes data from a buffer to a file descriptor, and the read() function reads data from a file descriptor into a buffer.
- **strtok() from string.h**: This function is used to parse the input command by separating it into distinct tokens using a specified delimiter. It returns a pointer to the next token in the string, and can be called repeatedly to extract all the tokens.

- **execvp() from unistd.h**: Using this function, external commands can be carried out. It replaces the current process with a new process that executes the command after receiving the command name and an array of arguments as input. If the execution is successful, the function does not return, and the new process takes over the execution of the program. If the execution fails, the function returns an error and the original process continues.
- **waitpid() from sys/wait.h**: This function is used to wait for the termination of a child process. It takes the PID of the child process and a pointer to a status variable as input, and returns the PID of the terminated child process when it terminates, or 0 if the child process is still running. The exit status of the child process can be obtained using the status variable.
- **signal() from signal.h**: This function is used to set a signal handler for a specified signal. A signal handler is a function that is called when the process receives a signal. In this project, the user pressing CTRL+Z initiates the SIGTSTP signal, which is handled by the signal() function.

Implementation: The shell was implemented as a single C file, with the following main functions:

- **setup()**: This function is used to parse the input command and separate it into distinct arguments. It takes the input buffer and an array of strings as input, and sets the elements of the args array to point to the beginning of null-terminated C-style strings. The background flag is also set by the function to specify whether or not the command should be executed in the background.
- **main()**: This is the main loop of the shell, which reads a command from the user, parses it, and executes it. The loop starts by displaying a prompt, then it reads a line of input from the user using the read() function. The input line is stored in the inputBuffer array, which is a character array of size MAX_LINE. If the input line is empty, the loop continues to the next iteration. The setup() function is used to parse the input and parse it into arguments if the input line is not empty.

The args array is an array of pointers to characters, with the last element set to NULL. The setup() function uses the strtok() function to extract the tokens from the input line, and stores the pointers to the beginning of each token in the args array. The execute() function receives

the args array after which it determines whether the command is internal or external and executes it accordingly.
The execute() function checks the first argument to determine the type of command and acts accordingly. If it is a built-in command, the function calls the corresponding function to handle it. If it is an external command, the function creates a child process using fork() and uses execvp() to execute the command in the child process. If the background flag is set, the child process runs in the background and the parent process moves on to the next iteration of the loop. If the background flag is not set, the parent process waits for the child process to finish using waitpid() before continuing. If the command is not a built-in command and there is an error executing it, the execute() function prints an error message and continues the loop.

- **sigtstp_handler()**: This function is the signal handler for the SIGTSTP signal. When the signal is received, the function prints a message and checks the value of the foreground_pid global variable. If it is greater than zero, it means that a foreground process is currently running, and the function sends the SIGTSTP signal to the process using the kill() function. The foreground_pid variable is then reset to zero.
- **add_to_history()**: This function adds a command to the history of commands. The history is stored in the history array, which is an array of pointers to characters with a fixed size (defined as 10). When a new command is added, the function shifts the history down one position, then concatenates the arguments of the command into a single string and stores it in the first element of the history array.
- **print_history()**: This function prints the history of commands. It iterates through the history array and prints the commands in the order they were entered, starting with the most recent command.
- **list_background_processes()**: This function displays a list of all background processes and their current status. It does this by iterating through the background_pids array, which stores the PIDs of all background processes. For each PID, the function calls the waitpid() function. If the waitpid() function returns a value of 0, this indicates that the process is still running, and the function will print the PID and the status "Running". If the waitpid() function returns the PID of the process, this means that the process has completed and the function will print the PID and the status "Done". If the waitpid() function returns any other value, this indicates that the status of the process is unknown, and the function will print the PID and the status "Unknown".

Testing: To evaluate the shell's functionality, we ran a range of commands including both built-in and external ones. We also tested the ability to run processes in the background using the & operator. Additionally, we tested the shell's handling of the SIGTSTP signal by pressing CTRL+Z while a foreground process was running.

Conclusion: The simple shell implemented in this project provides a basic command-line interface for interacting with the operating system. It can execute external commands and handle several built-in commands for managing the shell environment.The shell can be extended to include additional features, such as command redirection and piping, which allow the user to redirect the input/output of a command to/from a file or another command. Overall, the implementation of the simple shell was a useful exercise in understanding the design and implementation of a command-line interface and the process of interacting with the operating system.