CSE 344

SYSTEM PROGRAMMING

SPRING 2023

Homework 2

AHMET CEBECİ

1901042708

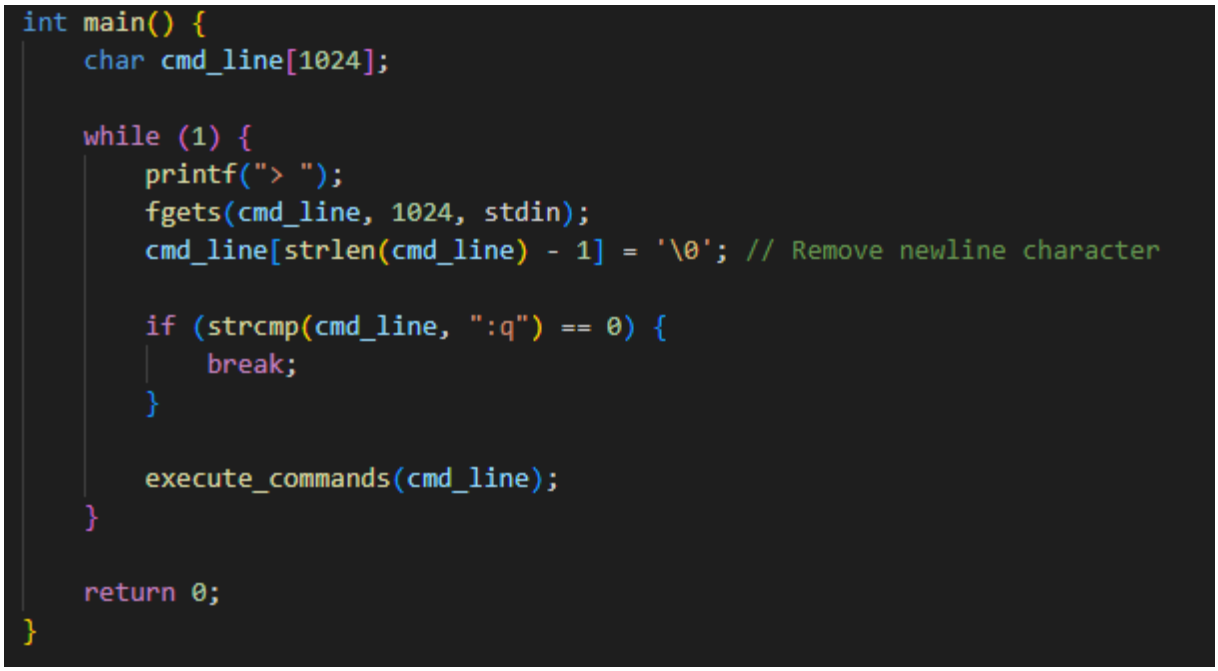
INTRODUCTION

The simple shell described in this report is a basic shell implemented in the C programming language. Its primary purpose is to offer essential functionalities such as executing commands, handling pipes, and managing input/output redirections. In addition, it logs the child process PID and the command executed in a log file for each command. This shell is not as feature-rich or as customizable as more advanced shells, but it serves as an excellent learning tool for those looking to understand the core concepts and workings of a shell or for those who want to build a custom shell from scratch.

CODE FUNCTIONALITY

The simple shell's code is organized into several functions, each responsible for a specific task:

1- main()



```c
int main() {
    char cmd_line[1024];

    while (1) {
        printf("> ");
        fgets(cmd_line, 1024, stdin);
        cmd_line[strlen(cmd_line) - 1] = '\0'; // Remove newline character

        if (strcmp(cmd_line, ":q") == 0) {
            break;
        }

        execute_commands(cmd_line);
    }

    return 0;
}
```

The main function serves as the entry point for the shell. It reads command lines from the user and runs them until the user enters ":q", signaling the program's termination.

2- run_commands(char *cmd_line)

```c
void run_command(char *cmd_line) {
    if (strchr(cmd_line, '|') != NULL || strchr(cmd_line, '<') != NULL || strchr(cmd_line, '>') != NULL) {
        handle_pipes_and_redirection(cmd_line);
    } else {
        run_single_command(cmd_line);
    }
}
```

This function checks if the command line contains pipes or redirections, and then either runs the command using run_single_command or handles the pipes and redirections using handle_pipes_and_redirection.

3- run_single_command(char *cmd)

```c
void run_single_command(char *cmd) {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        if (execl("/bin/bash", "/bin/bash", "-c", cmd, NULL) == -1) {
            perror("execl failed to execute command in child process for command");
            exit(EXIT_FAILURE);
        }
    } else if (pid > 0) {
        // Parent process
        int status;
        waitpid(pid, &status, 0);
        log_child_process(pid, cmd);
    } else {
        perror("fork failed to create child process for command");
        exit(EXIT_FAILURE);
    }
}
```

This function forks a child process and runs the given command using /bin/bash -c.

4- handle_redirection(char *cmd)

```c
void handle_redirection(char *cmd) {
    char *input_file = NULL;
    char *output_file = NULL;

    char *cmd_without_redirection = strtok(strdup(cmd), "<>");
    if (strchr(cmd, '<') != NULL) {
        input_file = strtok(NULL, "<>");
    }
    if (strchr(cmd, '>') != NULL) {
        output_file = strtok(NULL, "<>");
    }

    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        if (input_file) {
        int input_fd = open(input_file, O_RDONLY);
        if (input_fd == -1) {
            fprintf(stderr, "Error: Cannot open input file: %s\n", input_file);
            exit(EXIT_FAILURE);
        }
        dup2(input_fd, STDIN_FILENO);
        close(input_fd);
        }

        if (output_file) {
            int output_fd = open(output_file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (output_fd == -1) {
                fprintf(stderr, "Error: Cannot open output file: %s\n", output_file);
                exit(EXIT_FAILURE);
            }
            dup2(output_fd, STDOUT_FILENO);
            close(output_fd);
        }

        execute_single_command(cmd_without_redirection);
        exit(EXIT_SUCCESS);
    } else if (pid > 0) {
        // Parent process
        int status;
        waitpid(pid, &status, 0);
        log_child_process(pid, cmd_without_redirection);
    } else {
        perror("fork failed to create child process for command");
        exit(EXIT_FAILURE);
    }

    free(cmd_without_redirection);
}
```

This function manages input/output redirections by opening the specified input/output files and duplicating their file descriptors to the standard input/output file descriptors.

5- handle_pipes_and_redirection(char *cmd)

```c
void handle_pipes_and_redirection(char *cmd) {
    char *cmds[20];
    int cmd_count = 0;

    char *token = strtok(cmd, "|");
    while (token && cmd_count < 20) {
        cmds[cmd_count++] = token;
        token = strtok(NULL, "|");
    }

    int pipefds[2 * (cmd_count - 1)];
    for (int i = 0; i < cmd_count - 1; ++i) {
        if (pipe(pipefds + i * 2) < 0) {
            perror("pipe failed to create pipe file descriptors for command");
            exit(EXIT_FAILURE);
        }
    }

    for (int i = 0; i < cmd_count; ++i) {
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            if (i != cmd_count - 1) {
                dup2(pipefds[i * 2 + 1], STDOUT_FILENO);
            }
            if (i != 0) {
                dup2(pipefds[(i - 1) * 2], STDIN_FILENO);
            }
            for (int j = 0; j < 2 * (cmd_count - 1); ++j) {
                close(pipefds[j]);
            }

            if (strchr(cmds[i], '<') != NULL || strchr(cmds[i], '>') != NULL) {
                handle_redirection(cmds[i]);
            } else {
                execute_single_command(cmds[i]);
            }

            exit(EXIT_SUCCESS);
        } else if (pid < 0) {
            perror("fork failed to create child process for command");
            exit(EXIT_FAILURE);
        }
    }

    for (int j = 0; j < 2 * (cmd_count - 1); ++j) {
        close(pipefds[j]);
    }
    for (int i = 0; i < cmd_count; ++i) {
        int status;
        pid_t pid = wait(&status);
        log_child_process(pid, cmds[i]);
    }
}
```

This function manages multiple commands with pipes and redirections, creating pipes and forking child processes for each command, and then handling any redirections if necessary.

6- log_child_process()

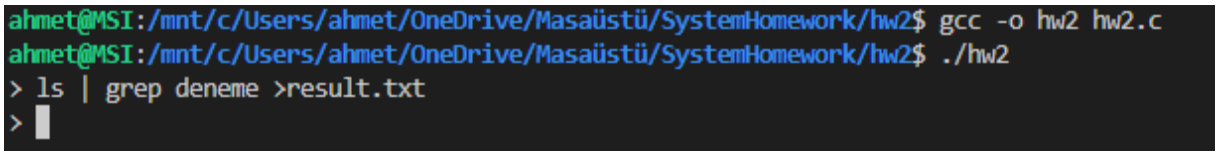
```c
void log_child_process(pid_t pid, char *cmd) {
    time_t rawtime;
    struct tm *timeinfo;
    char timestamp[20];

    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(timestamp, sizeof(timestamp), "%Y%m%d%H%M%S", timeinfo);

    char log_filename[30];
    sprintf(log_filename, "log_%s.txt", timestamp);

    FILE *log_file = fopen(log_filename, "a");
    if (log_file == NULL) {
        perror("fopen log file failed in log_child_process function");
        exit(EXIT_FAILURE);
    }

    fprintf(log_file, "PID: %d, Command: %s\n", pid, cmd);
    fclose(log_file);
}
```

This function logs the PID and command of a child process in a log file with the format: "log_YYYYMMDDHHMMSS.txt".

OUTPUT

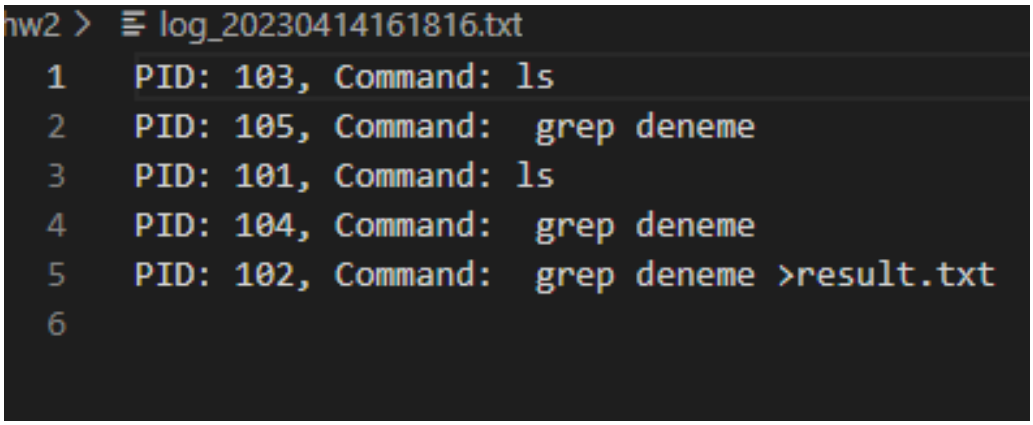
The shell's output consists of the results of the executed commands, which are displayed in the terminal, as well as log files created for each command. The log files contain information about the child process PID and the command executed. Log files are named according to a timestamp format, such as "log_YYYYMMDDHHMMSS.txt".

1 - Sample usage and output



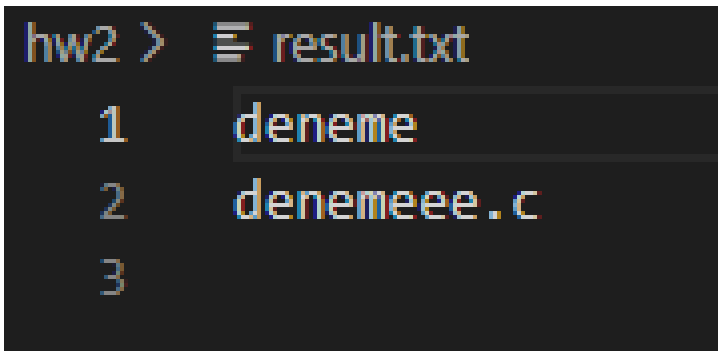
In the given custom shell implementation, the handle_pipes_and_redirection() function is responsible for handling commands that involve pipes and redirection. When you provide a command such as ls | grep deneme > result.txt, the function processes the command, creates the necessary pipes, and forks child processes to execute each part of the command.

The output you see, which includes multiple process IDs (PIDs) and commands, is a result of how the custom shell logs the execution of each child process. Here is a sample output:



The PIDs are assigned by the operating system when a new process is created using the fork() system call. The order in which they are displayed is not guaranteed to be sequential, as it depends on the order of process execution and how the processes are scheduled by the operating system.

The reason you see multiple PIDs for the same command, such as ls and grep deneme, is that the command is being executed in multiple child processes. This can happen when a command is part of a pipeline and is executed concurrently with other commands in separate child processes. In this case, each child process has a unique PID, even though it might be executing the same command.



PERFORMANCE

The performance of the simple shell is generally slower than that of standard shells, such as Bash or Zsh, due to its basic implementation and limited features. However, the simple shell successfully handles basic commands and is suitable for educational purposes or as a foundation for further development.

NOTE

You need to be careful when entering file names. Spaces should be taken into account when creating the name of the file to be processed with an appropriate name, and in some cases they should be enclosed in quotes.

Like this:

```
cat "file1.txt" "file2.txt" | sort | uniq > merged_files.txt
```

There is a situation that needs to be taken into account when using redirection. Note that if a space is placed after redirection, there will be a space in the filename.