Homework 2 Report – CSE344

Mehmet Mahir Kayadelen
Student no. 210104004252

April 2023

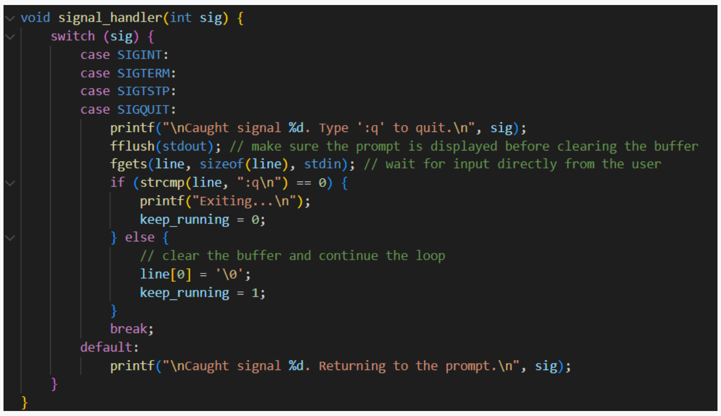
1 Composition

For this project, I used one c file main.c and one makefile for compile run and clean them for easy testability.

2 Program Logic

The main purpose of this program is making user friendly shell emulator. It reads user input from the command line, executes the entered commands, and redirects input and output as required. The program can also handle signals like SIGINT, SIGTERM, SIGTSTP, and SIGQUIT, which allows users to stop the execution of a running command.

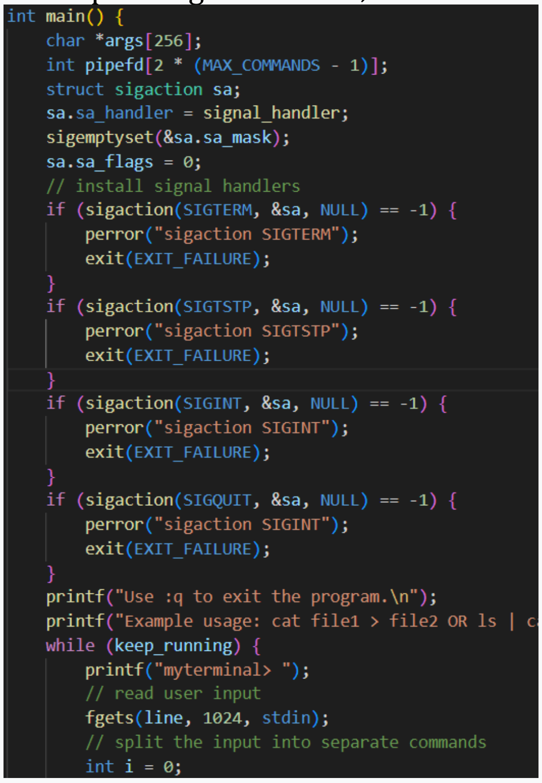
The program defines a signal_handler() function that handles the signals.



```c
void signal_handler(int sig) {
    switch (sig) {
        case SIGINT:
        case SIGTERM:
        case SIGTSTP:
        case SIGQUIT:
            printf("\nCaught signal %d. Type ':q' to quit.\n", sig);
            fflush(stdout); // make sure the prompt is displayed before clearing the buffer
            fgets(line, sizeof(line), stdin); // wait for input directly from the user
            if (strcmp(line, ":q\n") == 0) {
                printf("Exiting...\n");
                keep_running = 0;
            } else {
                // clear the buffer and continue the loop
                line[0] = '\0';
                keep_running = 1;
            }
            break;
        default:
            printf("\nCaught signal %d. Returning to the prompt.\n", sig);
    }
}
```

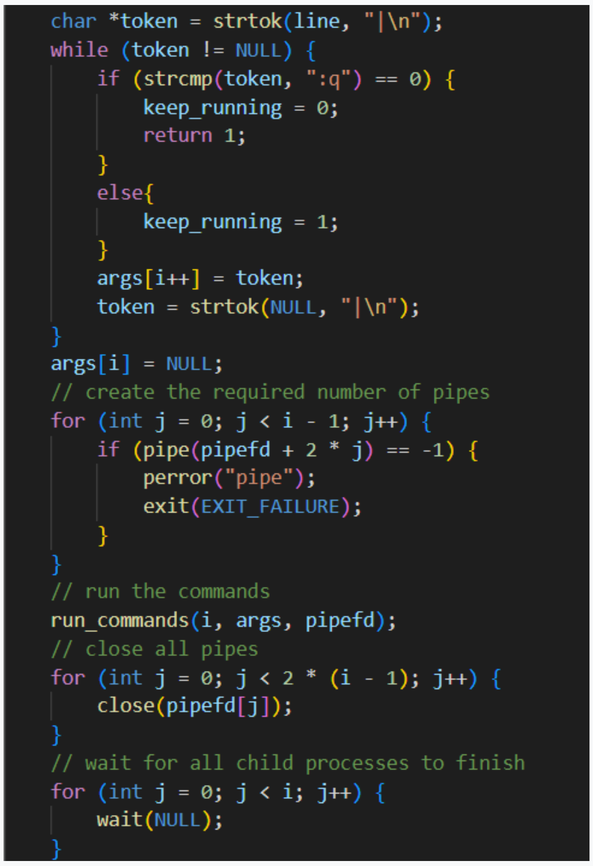
signal_handler() function uses a switch statement to determine how to handle the input signal. If the input signal matches one of the four cases SIGINT, SIGTERM, SIGTSTP, or SIGQUIT, the function prints a message to the console indicating that the signal was caught and prompts the user to input a command.The fflush function is called to ensure that the console output is displayed before the program waits for user input using the fgets function. If the user input matches the string ":q", the program sets the keep_running variable to 0, indicating that the program should exit. Otherwise, the line buffer is cleared, and the program continues running.If the input signal does not match any of the four cases, the function simply prints a message to the console indicating that the signal was caught and returns to the command prompt.Overall, this code is a signal handler that listens for specific signals and prompts the user for input to determine how to proceed based on the received signal.

The main() function reads user input from the command line, splits the input into separate commands, creates the required number(max commands -1) of pipes for communication between commands, runs the commands, logs the child process pids and corresponding commands, and waits for all child processes to finish.



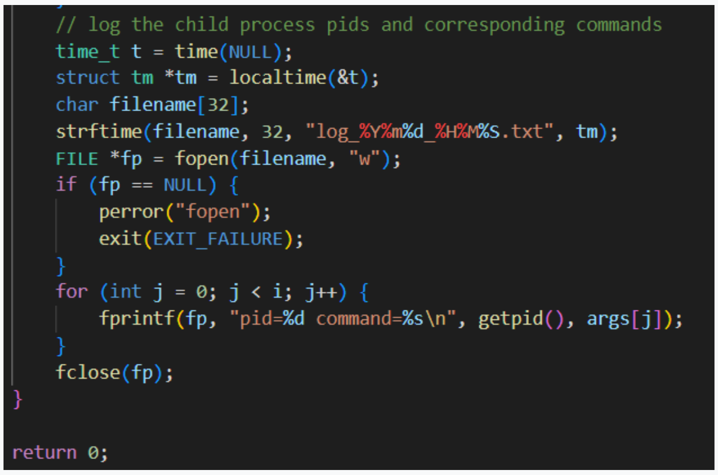
```c
int main() {
    char *args[256];
    int pipefd[2 * (MAX_COMMANDS - 1)];
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    // install signal handlers
    if (sigaction(SIGTERM, &sa, NULL) == -1) {
        perror("sigaction SIGTERM");
        exit(EXIT_FAILURE);
    }
    if (sigaction(SIGTSTP, &sa, NULL) == -1) {
        perror("sigaction SIGTSTP");
        exit(EXIT_FAILURE);
    }
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction SIGINT");
        exit(EXIT_FAILURE);
    }
    if (sigaction(SIGQUIT, &sa, NULL) == -1) {
        perror("sigaction SIGINT");
        exit(EXIT_FAILURE);
    }
    printf("Use :q to exit the program.\n");
    printf("Example usage: cat file1 > file2 OR ls | c
    while (keep_running) {
        printf("myterminal> ");
        // read user input
        fgets(line, 1024, stdin);
        // split the input into separate commands
        int i = 0;
```

*args is an array of character pointers to hold the command line arguments. pipefd is an array of file descriptors for pipes. The maximum number of pipes allowed is defined by MAX_COMMANDS constant. sa is a signal handler for the signals SIGTERM, SIGTSTP, SIGINT, and SIGQUIT. The signal handler is a function named signal_handler. after installed the signal handlers for the four signals. If an error occurs, the perror function prints an error message and the program terminates.After prints two lines some instructions on how to use the program.After loop starts and runs until the keep_running flag is set to 0. After prints a prompt to indicate that the program is ready to accept input.Reads a line of input from the user, up to 1024 characters long, and stores it in the line buffer.



```c
    char *token = strtok(line, "|\n");
    while (token != NULL) {
        if (strcmp(token, ":q") == 0) {
            keep_running = 0;
            return 1;
        }
        else{
            keep_running = 1;
        }
        args[i++] = token;
        token = strtok(NULL, "|\n");
    }
    args[i] = NULL;
    // create the required number of pipes
    for (int j = 0; j < i - 1; j++) {
        if (pipe(pipefd + 2 * j) == -1) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    }
    // run the commands
    run_commands(i, args, pipefd);
    // close all pipes
    for (int j = 0; j < 2 * (i - 1); j++) {
        close(pipefd[j]);
    }
    // wait for all child processes to finish
    for (int j = 0; j < i; j++) {
        wait(NULL);
    }
```

After tokenizes the input string by the pipe character | or newline character \n using the strtok function. The tokens are stored in the args array, and the i variable keeps track of the number of tokens. If the token is :q, the program exits. The last element of the args array is set to NULL.   After creates the required number of pipes(commands -1) between the child processes. After call the run_commands function to execute the commands. Pass the number of commands i, the args array, and the pipefd array as arguments.Close all the pipes created earlier using close.Wait for all child processes to finish using wait array as arguments.



```c
    // log the child process pids and corresponding commands
    time_t t = time(NULL);
    struct tm *tm = localtime(&t);
    char filename[32];
    strftime(filename, 32, "log_%Y%m%d_%H%M%S.txt", tm);
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    for (int j = 0; j < i; j++) {
        fprintf(fp, "pid=%d command=%s\n", getpid(), args[j]);
    }
    fclose(fp);
}

return 0;
```

Create a log file with the name log__.txt using the current date and time. Write the process IDs with getpid() and

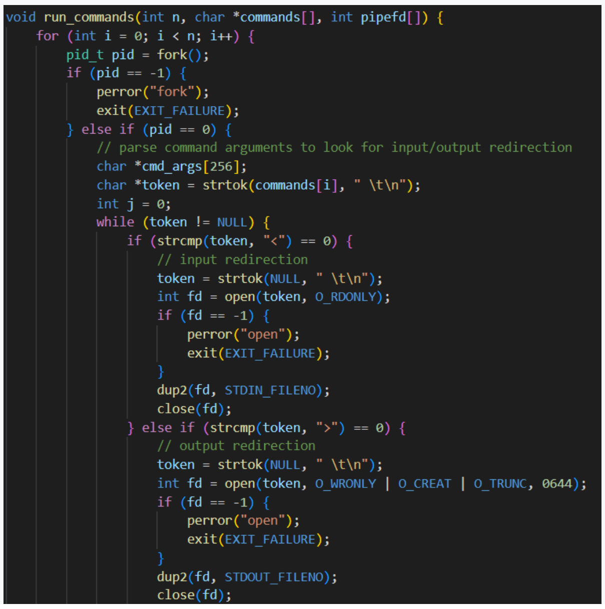
corresponding commands to the file using fprintf

The run_commands() function is called by the main() function to execute the entered

commands. It creates a child process for each command, parses the command

arguments to look for input/output redirection, and redirects input/output as required.

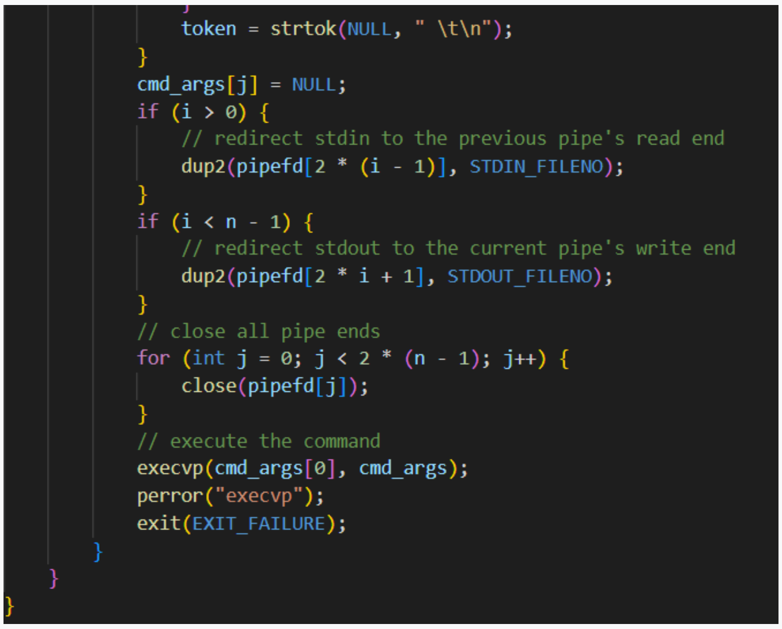
The child process executes the command and then exits.



```c
void run_commands(int n, char *commands[], int pipefd[]) {
    for (int i = 0; i < n; i++) {
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // parse command arguments to look for input/output redirection
            char *cmd_args[256];
            char *token = strtok(commands[i], " \t\n");
            int j = 0;
            while (token != NULL) {
                if (strcmp(token, "<") == 0) {
                    // input redirection
                    token = strtok(NULL, " \t\n");
                    int fd = open(token, O_RDONLY);
                    if (fd == -1) {
                        perror("open");
                        exit(EXIT_FAILURE);
                    }
                    dup2(fd, STDIN_FILENO);
                    close(fd);
                } else if (strcmp(token, ">") == 0) {
                    // output redirection
                    token = strtok(NULL, " \t\n");
                    int fd = open(token, O_WRONLY | O_CREAT | O_TRUNC, 0644);
                    if (fd == -1) {
                        perror("open");
                        exit(EXIT_FAILURE);
                    }
                    dup2(fd, STDOUT_FILENO);
                    close(fd);
```

run_commands function iterates over the commands using a for loop, and for each command it creates a child process using the fork() system call. The child process then executes the command using the execvp() system call. The parent process continues iterating over the remaining commands.

if pid = 0 this means it is child process. It's parsing the current command to check for input/output redirection. It uses the strtok() function to split the command into tokens based on whitespace characters. If a token is <, it means that the next token is the name of a file from which the command should read input. If a token is >, it means that the next token is the name of a file to which the command should write output. The code uses the open() function to open the file, and then uses dup2() to duplicate the file descriptor to either STDIN_FILENO or STDOUT_FILENO. The close() function is then called to close the original file descriptor.



```
                }
                token = strtok(NULL, " \t\n");
            }
            cmd_args[j] = NULL;
            if (i > 0) {
                // redirect stdin to the previous pipe's read end
                dup2(pipefd[2 * (i - 1)], STDIN_FILENO);
            }
            if (i < n - 1) {
                // redirect stdout to the current pipe's write end
                dup2(pipefd[2 * i + 1], STDOUT_FILENO);
            }
            // close all pipe ends
            for (int j = 0; j < 2 * (n - 1); j++) {
                close(pipefd[j]);
            }
            // execute the command
            execvp(cmd_args[0], cmd_args);
            perror("execvp");
            exit(EXIT_FAILURE);
        }
    }
}
```

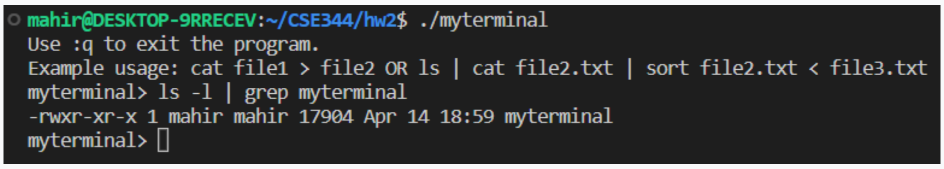
After redirecting the standard input and/or standard output of the current command to a pipe, if necessary. If the current command is not the first command, its standard input is redirected to the read end of the previous pipe. If the current command is not the last command, its standard output is redirected to the write end of the current pipe.

After closing all the file descriptors associated with the pipes that are not needed in the current process. Then code is executing the command using the execvp() system call. If execvp() returns, it means that the command execution failed.

3 Testing

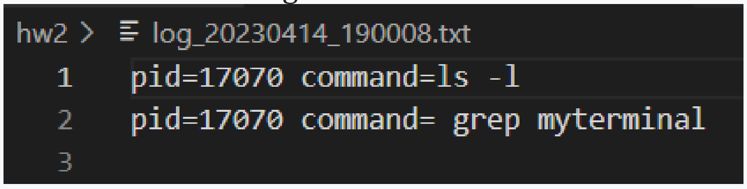
a)Pipe test:

Terminal SS



```
mahir@DESKTOP-9RRECEV:~/CSE344/hw2$ ./myterminal
Use :q to exit the program.
Example usage: cat file1 > file2 OR ls | cat file2.txt | sort file2.txt < file3.txt
myterminal> ls -l | grep myterminal
-rwxr-xr-x 1 mahir mahir 17904 Apr 14 18:59 myterminal
myterminal> 
```

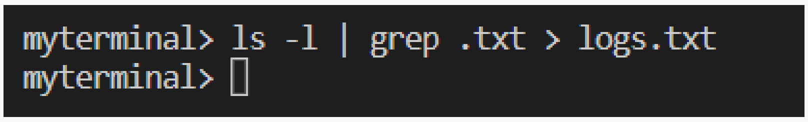
Log file after execution



```
hw2 > log_20230414_190008.txt
pid=17070 command=ls -l
pid=17070 command= grep myterminal
```

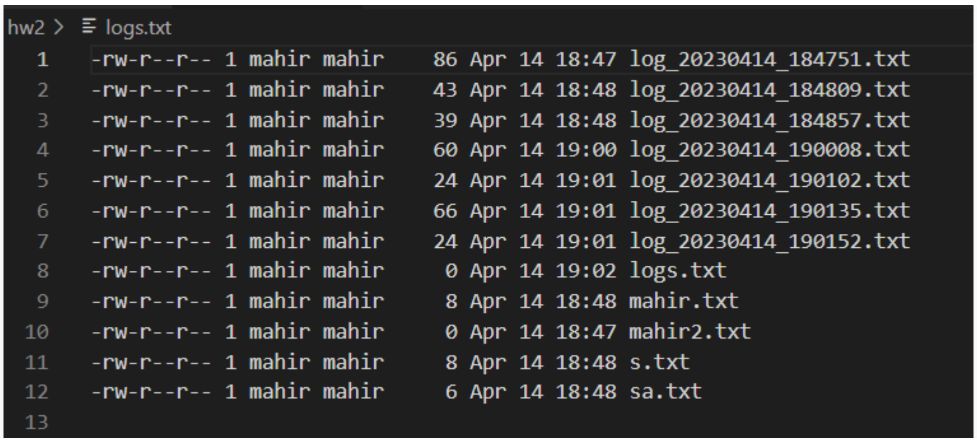
b)Output file descriptor redirection “>” test:

Terminal ss



```
myterminal> ls -l | grep .txt > logs.txt
myterminal> 
```

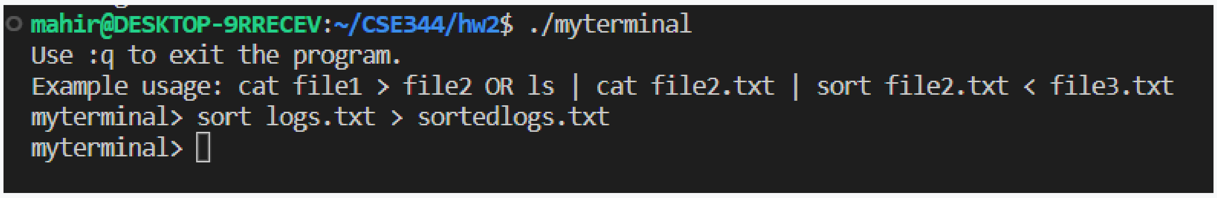
“logs.txt” file after execution command



```
hw2 > logs.txt
-rw-r--r-- 1 mahir mahir    86 Apr 14 18:47 log_20230414_184751.txt
-rw-r--r-- 1 mahir mahir    43 Apr 14 18:48 log_20230414_184809.txt
-rw-r--r-- 1 mahir mahir    39 Apr 14 18:48 log_20230414_184857.txt
-rw-r--r-- 1 mahir mahir    60 Apr 14 19:00 log_20230414_190008.txt
-rw-r--r-- 1 mahir mahir    24 Apr 14 19:01 log_20230414_190102.txt
-rw-r--r-- 1 mahir mahir    66 Apr 14 19:01 log_20230414_190135.txt
-rw-r--r-- 1 mahir mahir    24 Apr 14 19:01 log_20230414_190152.txt
-rw-r--r-- 1 mahir mahir     0 Apr 14 19:02 logs.txt
-rw-r--r-- 1 mahir mahir     8 Apr 14 18:48 mahir.txt
-rw-r--r-- 1 mahir mahir     0 Apr 14 18:47 mahir2.txt
-rw-r--r-- 1 mahir mahir     8 Apr 14 18:48 s.txt
-rw-r--r-- 1 mahir mahir     6 Apr 14 18:48 sa.txt
```

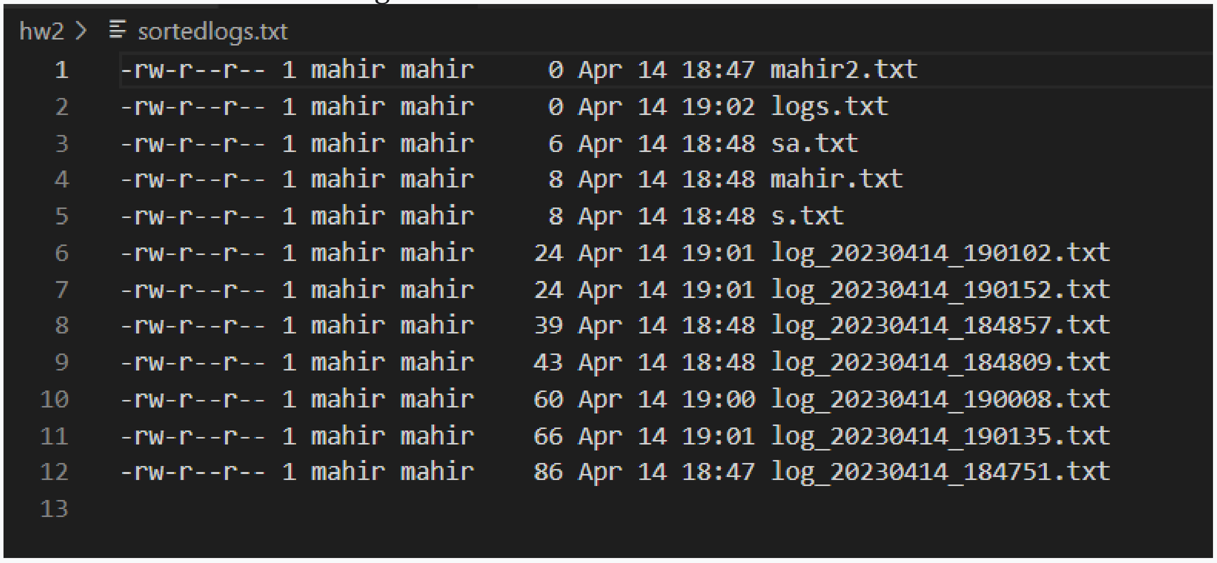
c)Sort and redirection test:

Terminal ss



```
mahir@DESKTOP-9RRECEV:~/CSE344/hw2$ ./myterminal
Use :q to exit the program.
Example usage: cat file1 > file2 OR ls | cat file2.txt | sort file2.txt < file3.txt
myterminal> sort logs.txt > sortedlogs.txt
myterminal> 
```

“sortedlogs.txt” file after execution command

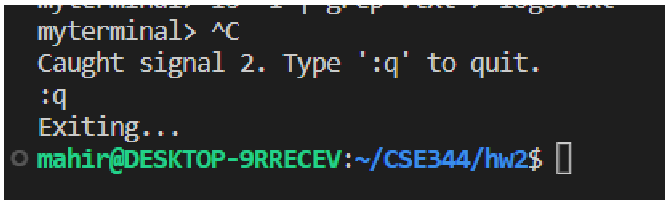


hw2 > sortedlogs.txt

```
-rw-r--r-- 1 mahir mahir     0 Apr 14 18:47 mahir2.txt
-rw-r--r-- 1 mahir mahir     0 Apr 14 19:02 logs.txt
-rw-r--r-- 1 mahir mahir     6 Apr 14 18:48 sa.txt
-rw-r--r-- 1 mahir mahir     8 Apr 14 18:48 mahir.txt
-rw-r--r-- 1 mahir mahir     8 Apr 14 18:48 s.txt
-rw-r--r-- 1 mahir mahir    24 Apr 14 19:01 log_20230414_190102.txt
-rw-r--r-- 1 mahir mahir    24 Apr 14 19:01 log_20230414_190152.txt
-rw-r--r-- 1 mahir mahir    39 Apr 14 18:48 log_20230414_184857.txt
-rw-r--r-- 1 mahir mahir    43 Apr 14 18:48 log_20230414_184809.txt
-rw-r--r-- 1 mahir mahir    60 Apr 14 19:00 log_20230414_190008.txt
-rw-r--r-- 1 mahir mahir    66 Apr 14 19:01 log_20230414_190135.txt
-rw-r--r-- 1 mahir mahir    86 Apr 14 18:47 log_20230414_184751.txt

```

d)Signal handling test:



```
myterminal> ^C
Caught signal 2. Type ':q' to quit.
:q
Exiting...
mahir@DESKTOP-9RRECEV:~/CSE344/hw2$ 
```