Gebze Technical University

Computer Engineering

System Programing(CSE 344)

Homework #2

Berkan AKIN

171044073

Problem

you are expected to develop a terminal emulator capable of handling up to 20 shell commands in a single line, without using the "system()" function from the standard C library. Instead, you should utilize the “fork()”, “execl()”, “wait()”, and “exit()” functions.

Solutions

Firstly, since our emulator needs to be able to initiate and manage multiple processes simultaneously, we will use the "fork()" function. The "fork()" function creates a new process by duplicating the current process at the moment it is called. This way, we can execute the desired number of commands concurrently.

Secondly, we will use the "execl()" function for each new process. This function starts the program that will run in the new process. As a result, our terminal emulator can execute the specified shell commands.

Thirdly, we will use the "wait()" function to wait for all processes to complete. This function blocks the main process from continuing until all subprocesses have completed. In this way, our terminal emulator will operate correctly by waiting for all shell commands to complete.

Lastly, we will use the "exit()" function to ensure each subprocess terminates properly. This function allows the processes to return to the operating system when they finish.

In summary, to make our terminal emulator work, we will follow these steps:

1. First, the terminal emulator must separate the user's input into commands. To do this, you will need a series of string processing to split the input into parts. For example, you can use a function like strtok() to separate the commands.
2. Then, use the "fork()" function to start a new process for each command. You can determine the difference between the main process and child process by checking the value returned by the fork() function.
3. In the child processes, use the "execl()" function to execute the shell commands. This function starts the program that will run in the new process and executes the shell commands.
4. In the main process, use the "wait()" function to wait for all child processes to complete. This ensures the proper operation of the terminal emulator.
5. Once all processes have completed, each process should be properly terminated using the "exit()" function.

By following these steps, you can develop a terminal emulator capable of running 20 shell commands simultaneously.

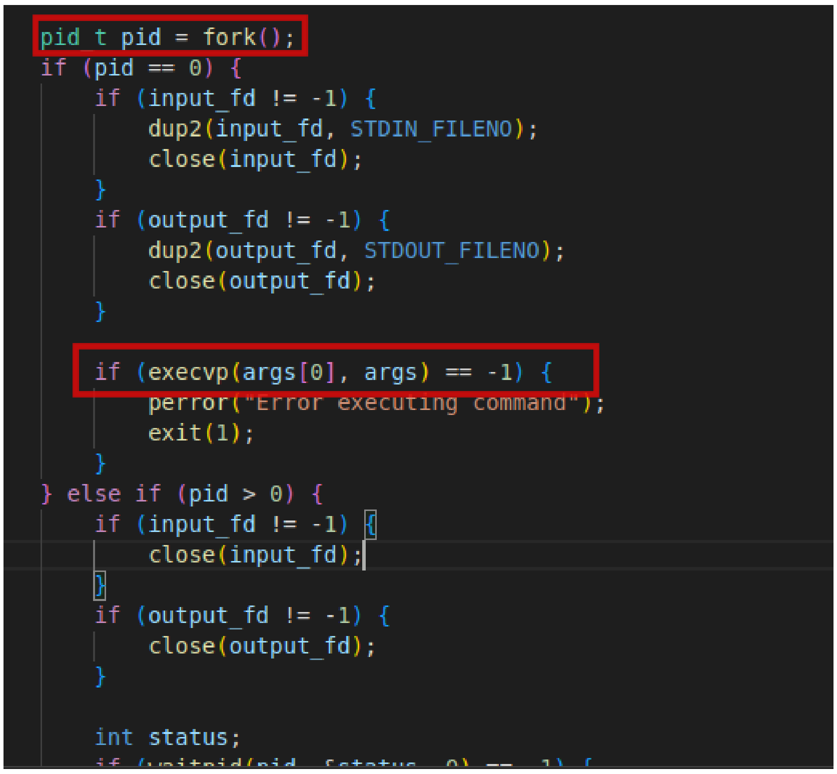
Requirements

1. Each shell command should be executed via a newly created child process, meaning that multiple commands will result in multiple child processes.
2. Proper handling of pipes ("|") and redirections ("<", ">") by redirecting the appropriate file descriptors.
3. Usage information should be printed if the program is not called properly.
4. Error messages and signals that occur during execution should be printed, and the program should return to the prompt to receive new commands.
5. Aside from a SIGKILL (which also should be handled properly) the program must wait for “:q” to finalize its execution.
6. Upon completion, all pids of child processes with their corresponding commands should be logged in a separate file. Each execution should create a new log file with a name corresponding to the current timestamp.

Tests for Requirements

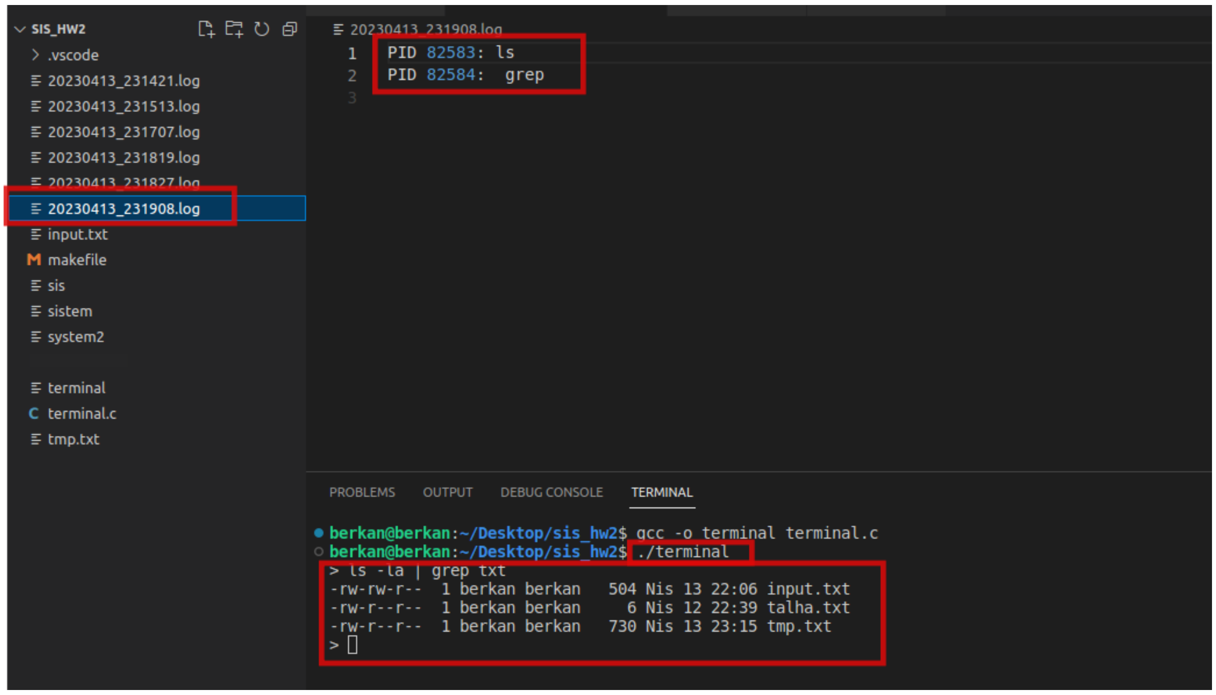
1. Requirements

- The required requirement has been fulfilled. A new child process is created for each command line and the child process is made to execute the command. The sample code is shown below



```
pid_t pid = fork();
if (pid == 0) {
    if (input_fd != -1) {
        dup2(input_fd, STDIN_FILENO);
        close(input_fd);
    }
    if (output_fd != -1) {
        dup2(output_fd, STDOUT_FILENO);
        close(output_fd);
    }

    if (execvp(args[0], args) == -1) {
        perror("Error executing command");
        exit(1);
    }
} else if (pid > 0) {
    if (input_fd != -1) {
        close(input_fd);
    }
    if (output_fd != -1) {
        close(output_fd);
    }

    int status;
```

- It can be observed that processes with different PID values are created



```
PID 82583: ls
PID 82584:  grep
```

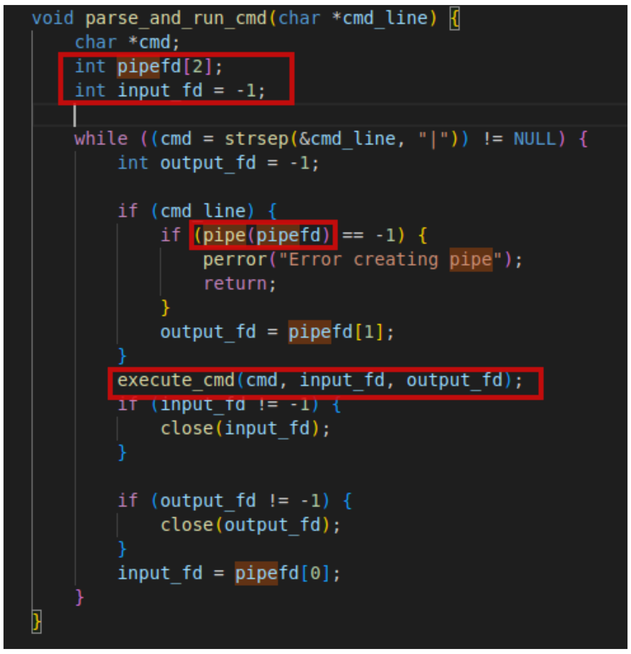
```
berkan@berkan:~/Desktop/sis_hw2$ gcc -o terminal terminal.c
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> ls -la | grep txt
-rw-rw-r--  1 berkan berkan   504 Nis 13 22:06 input.txt
-rw-r--r--  1 berkan berkan     6 Nis 12 22:39 talha.txt
-rw-r--r--  1 berkan berkan   730 Nis 13 23:15 tmp.txt
> 
```

2. Requirement

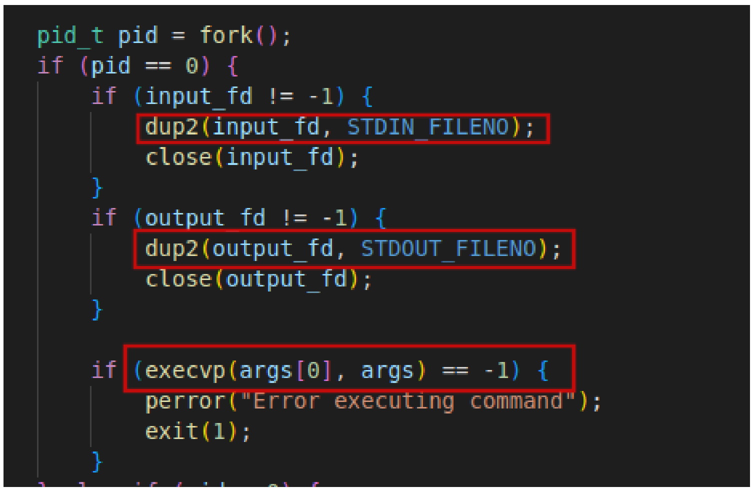
- The execution of the | > < commands is demonstrated with examples of the commands being run below.

pipes ("|")

To use pipes, the dub2() function and the pipe() function have been used in the code.



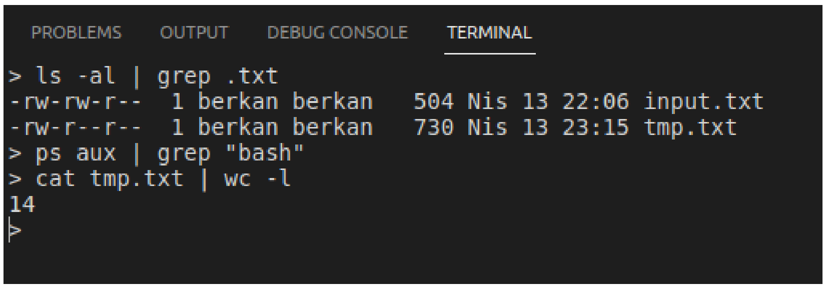
```c
void parse_and_run_cmd(char *cmd_line) {
    char *cmd;
    int pipefd[2];
    int input_fd = -1;

    while ((cmd = strsep(&cmd_line, "|")) != NULL) {
        int output_fd = -1;

        if (cmd_line) {
            if (pipe(pipefd) == -1) {
                perror("Error creating pipe");
                return;
            }
            output_fd = pipefd[1];
        }
        execute_cmd(cmd, input_fd, output_fd);
        if (input_fd != -1) {
            close(input_fd);
        }

        if (output_fd != -1) {
            close(output_fd);
        }
        input_fd = pipefd[0];
    }
}
```



```c
pid_t pid = fork();
if (pid == 0) {
    if (input_fd != -1) {
        dup2(input_fd, STDIN_FILENO);
        close(input_fd);
    }
    if (output_fd != -1) {
        dup2(output_fd, STDOUT_FILENO);
        close(output_fd);
    }

    if (execvp(args[0], args) == -1) {
        perror("Error executing command");
        exit(1);
    }
```

Example

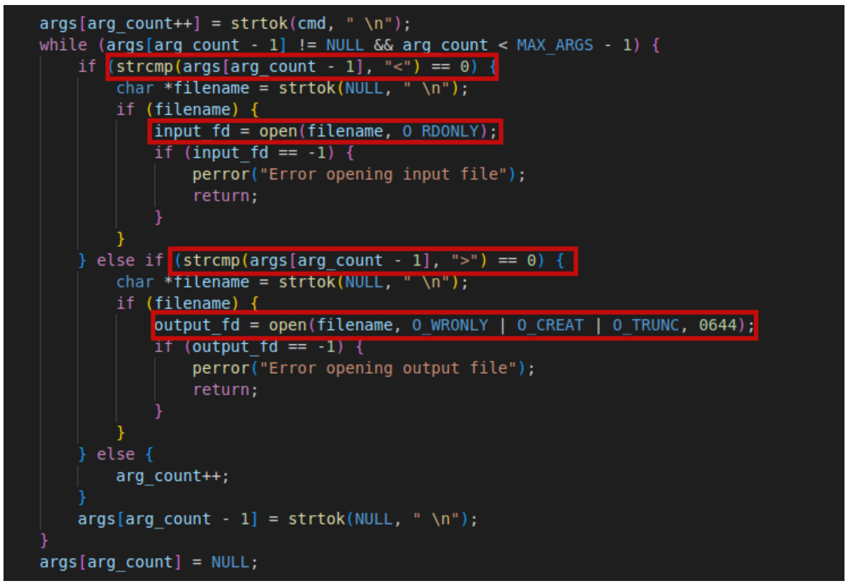
- Two different examples using pipes have been given.
-



```
> ls -al | grep .txt
-rw-rw-r--  1 berkan berkan   504 Nis 13 22:06 input.txt
-rw-r--r--  1 berkan berkan   730 Nis 13 23:15 tmp.txt
> ps aux | grep "bash"
> cat tmp.txt | wc -l
14
>
```

Redirections ("<", ">")

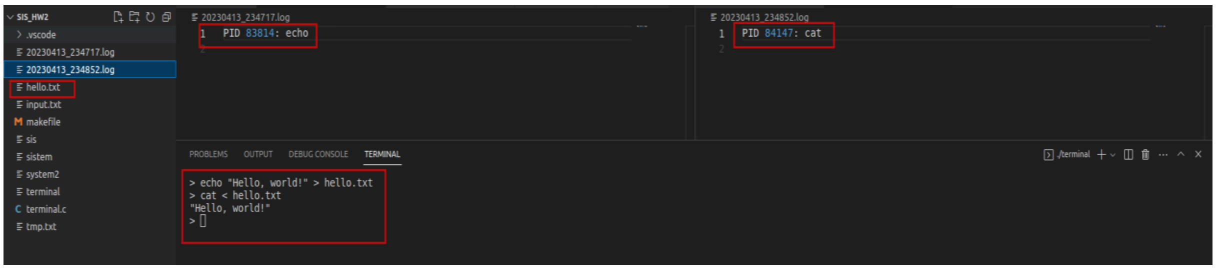
- Below is the code snippet for redirection commands in the written C program.



```c
args[arg_count++] = strtok(cmd, " \n");
while (args[arg_count - 1] != NULL && arg_count < MAX_ARGS - 1) {
    if (strcmp(args[arg_count - 1], "<") == 0) {
        char *filename = strtok(NULL, " \n");
        if (filename) {
            input_fd = open(filename, O_RDONLY);
            if (input_fd == -1) {
                perror("Error opening input file");
                return;
            }
        }
    } else if (strcmp(args[arg_count - 1], ">") == 0) {
        char *filename = strtok(NULL, " \n");
        if (filename) {
            output_fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (output_fd == -1) {
                perror("Error opening output file");
                return;
            }
        }
    } else {
        arg_count++;
    }
    args[arg_count - 1] = strtok(NULL, " \n");
}
args[arg_count] = NULL;
```

Example

- As seen below, two different commands have been executed and their outputs can be seen.



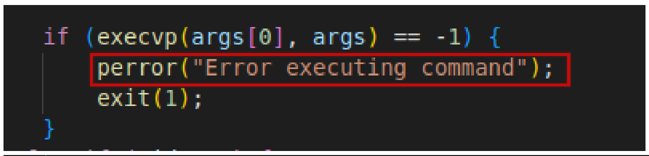
```
20230413_234717.log
1  PID 83814: echo
```

```
20230413_234852.log
1  PID 84147: cat
```

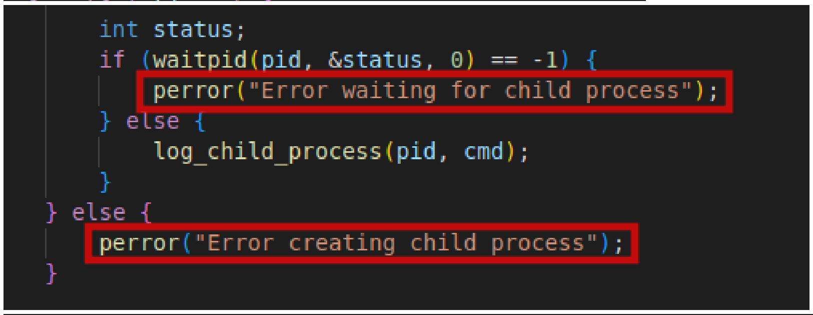
```
> echo "Hello, world!" > hello.txt
> cat < hello.txt
"Hello, world!"
> 
```

3. Requirement

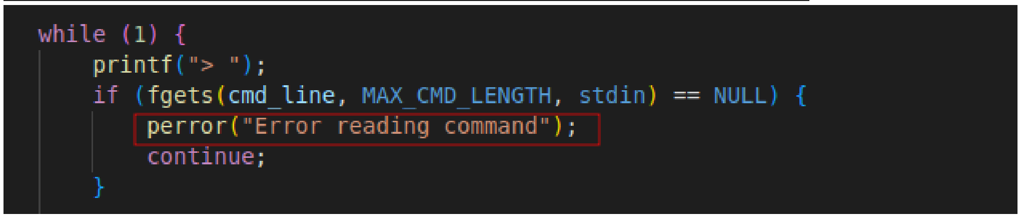
The commands are taken and executed, process creation is performed, and errors regarding the completion of child processes are being checked.



```c
if (execvp(args[0], args) == -1) {
    perror("Error executing command");
    exit(1);
}
```

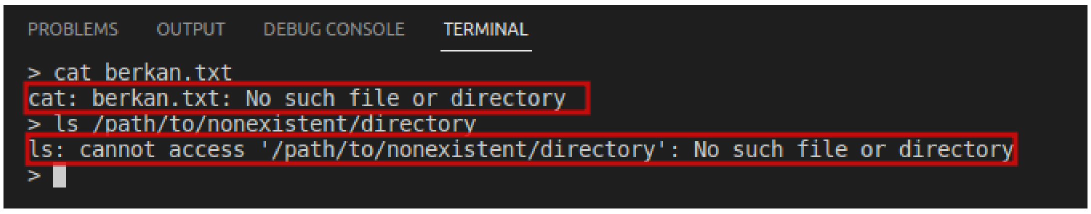


```c
    int status;
    if (waitpid(pid, &status, 0) == -1) {
        perror("Error waiting for child process");
    } else {
        log_child_process(pid, cmd);
    }
} else {
    perror("Error creating child process");
}
```



```c
while (1) {
    printf("> ");
    if (fgets(cmd_line, MAX_CMD_LENGTH, stdin) == NULL) {
        perror("Error reading command");
        continue;
    }
```

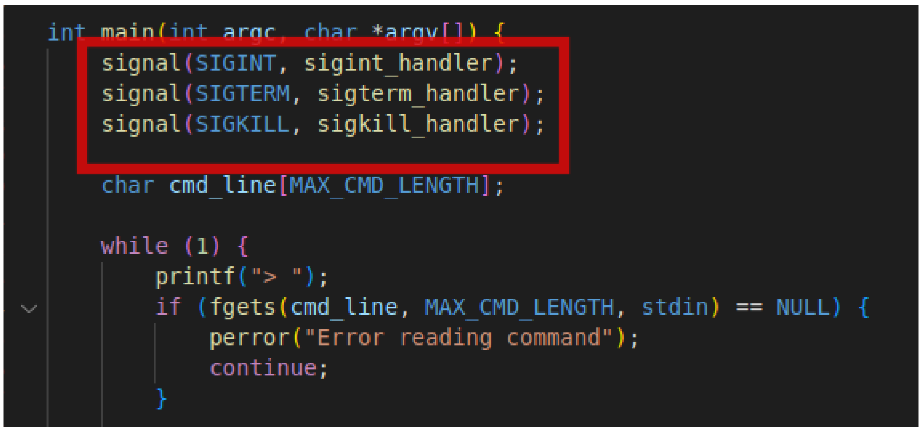
Example



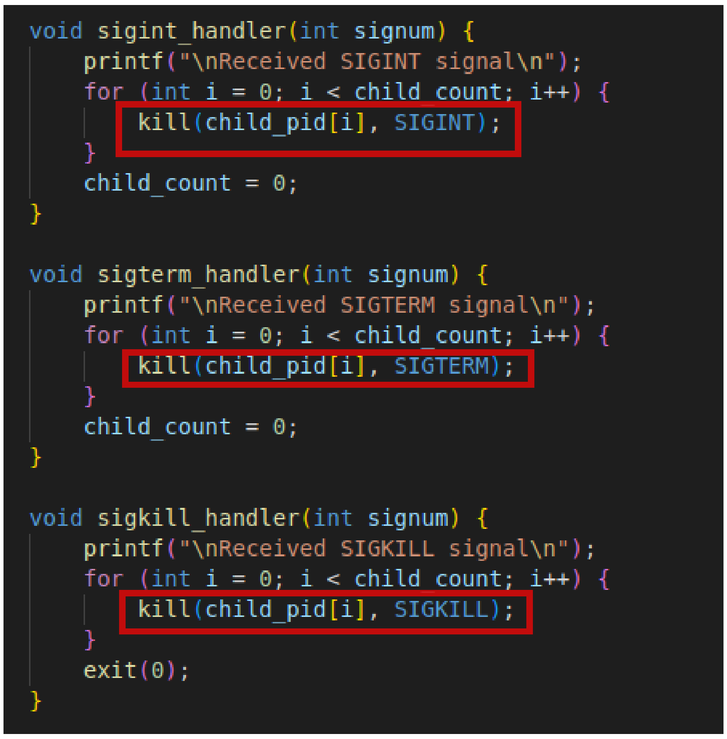
```
> cat berkan.txt
cat: berkan.txt: No such file or directory
> ls /path/to/nonexistent/directory
ls: cannot access '/path/to/nonexistent/directory': No such file or directory
> 
```

4. Requirement

- The desired signal handler functions are listed below.

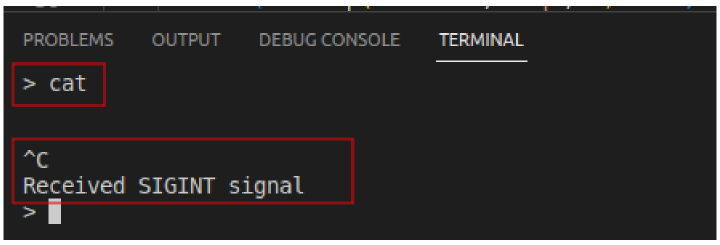


```
int main(int argc, char *argv[]) {
    signal(SIGINT, sigint_handler);
    signal(SIGTERM, sigterm_handler);
    signal(SIGKILL, sigkill_handler);

    char cmd_line[MAX_CMD_LENGTH];

    while (1) {
        printf("> ");
        if (fgets(cmd_line, MAX_CMD_LENGTH, stdin) == NULL) {
            perror("Error reading command");
            continue;
        }
```



```
void sigint_handler(int signum) {
    printf("\nReceived SIGINT signal\n");
    for (int i = 0; i < child_count; i++) {
        kill(child_pid[i], SIGINT);
    }
    child_count = 0;
}

void sigterm_handler(int signum) {
    printf("\nReceived SIGTERM signal\n");
    for (int i = 0; i < child_count; i++) {
        kill(child_pid[i], SIGTERM);
    }
    child_count = 0;
}

void sigkill_handler(int signum) {
    printf("\nReceived SIGKILL signal\n");
    for (int i = 0; i < child_count; i++) {
        kill(child_pid[i], SIGKILL);
    }
    exit(0);
}
```

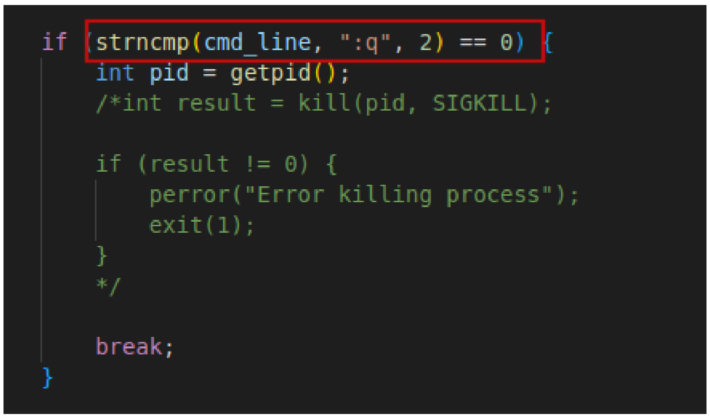
Example



```
> cat

^C
Received SIGINT signal
> 
```

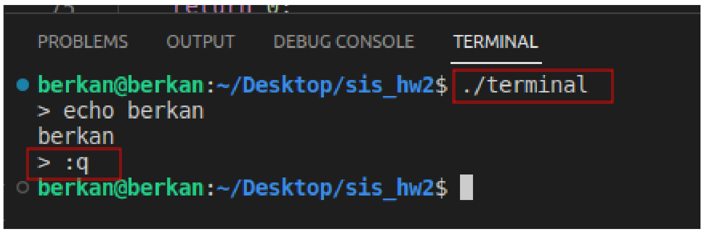
5. Requirement

- We can exit from the terminal emulator we created by typing :q. The code snippet and an example screenshot are shown below



```
if (strncmp(cmd_line, ":q", 2) == 0) {
    int pid = getpid();
    /*int result = kill(pid, SIGKILL);

    if (result != 0) {
        perror("Error killing process");
        exit(1);
    }
    */

    break;
}
```

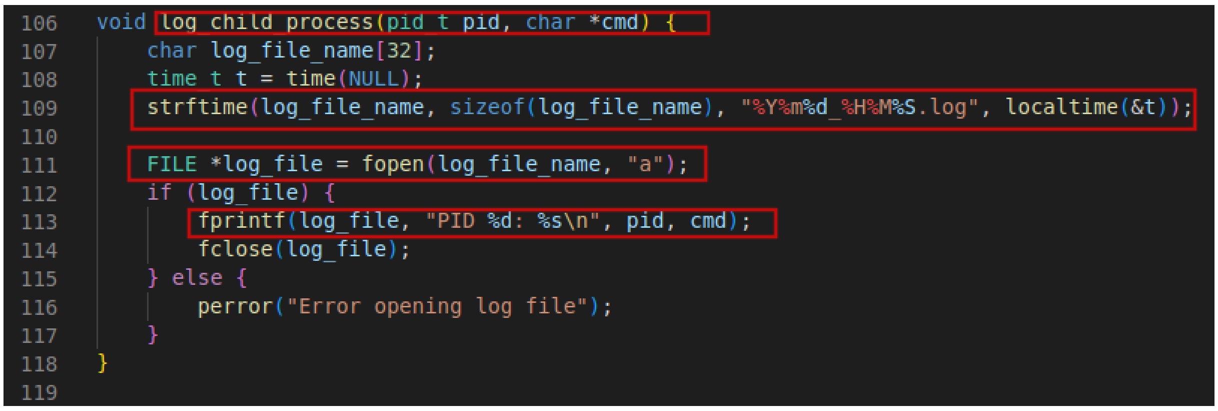
Example



```
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> echo berkan
berkan
> :q
berkan@berkan:~/Desktop/sis_hw2$
```

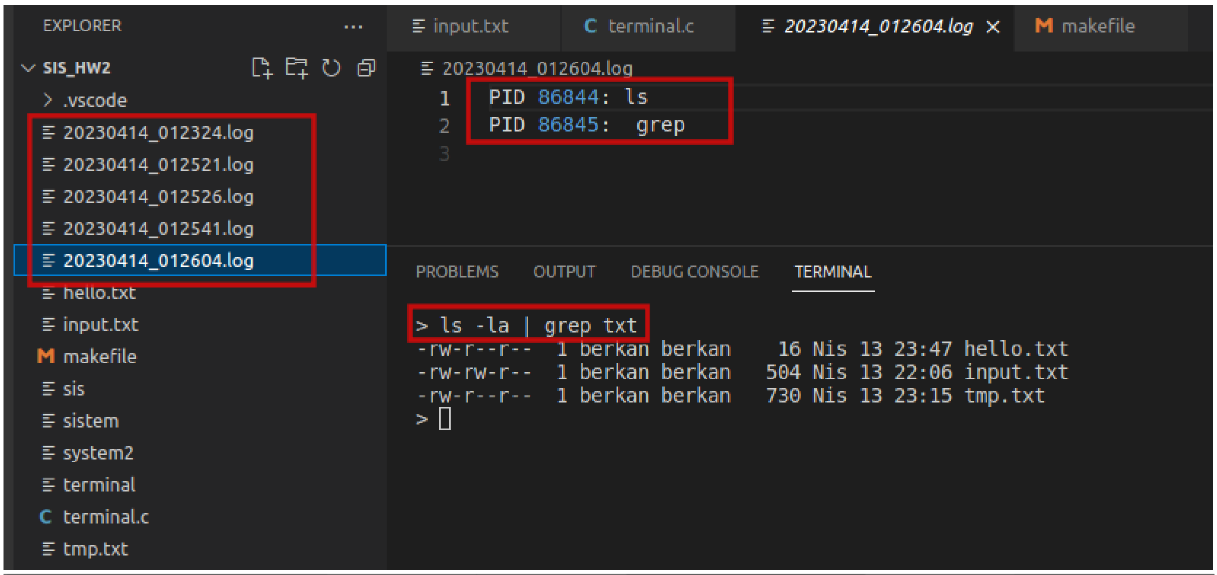
6. Requirement

- A separate log file is created for each executed bash shell. The executed processes and their PIDs are recorded in the log file.



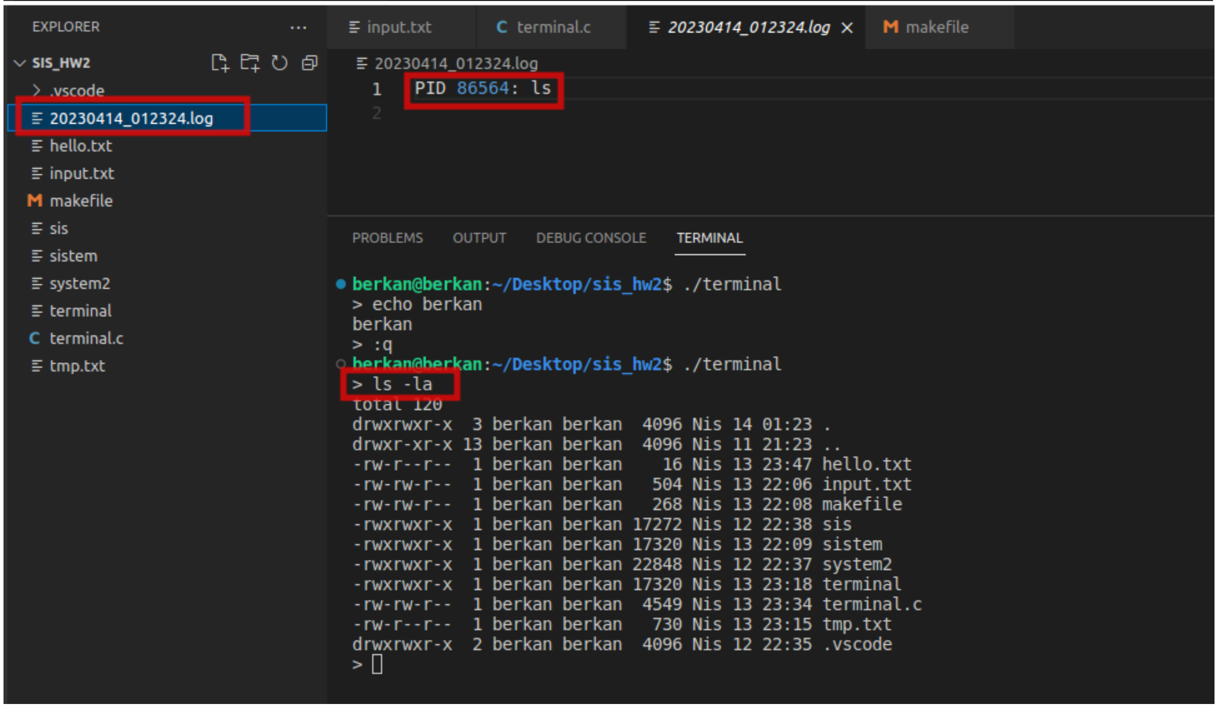
```
void log_child_process(pid_t pid, char *cmd) {
    char log_file_name[32];
    time_t t = time(NULL);
    strftime(log_file_name, sizeof(log_file_name), "%Y%m%d_%H%M%S.log", localtime(&t));

    FILE *log_file = fopen(log_file_name, "a");
    if (log_file) {
        fprintf(log_file, "PID %d: %s\n", pid, cmd);
        fclose(log_file);
    } else {
        perror("Error opening log file");
    }
}
```

Example



```
PID 86844: ls
PID 86845:  grep
```

```
> ls -la | grep txt
-rw-r--r--  1 berkan berkan    16 Nis 13 23:47 hello.txt
-rw-rw-r--  1 berkan berkan   504 Nis 13 22:06 input.txt
-rw-r--r--  1 berkan berkan   730 Nis 13 23:15 tmp.txt
> 
```

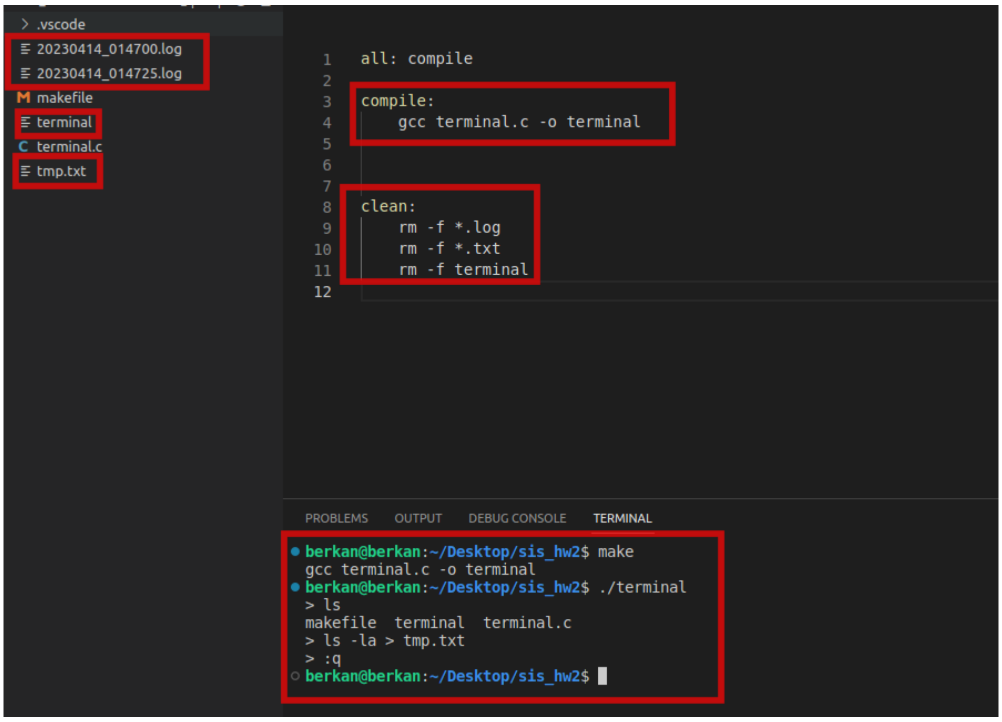


```
PID 86564: ls
```

```
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> echo berkan
berkan
> :q
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> ls -la
total 120
drwxrwxr-x  3 berkan berkan  4096 Nis 14 01:23 .
drwxr-xr-x 13 berkan berkan  4096 Nis 11 21:23 ..
-rw-r--r--  1 berkan berkan    16 Nis 13 23:47 hello.txt
-rw-rw-r--  1 berkan berkan   504 Nis 13 22:06 input.txt
-rw-rw-r--  1 berkan berkan   268 Nis 13 22:08 makefile
-rwxrwxr-x  1 berkan berkan 17272 Nis 12 22:38 sis
-rwxrwxr-x  1 berkan berkan 17320 Nis 13 22:09 sistem
-rwxrwxr-x  1 berkan berkan 22848 Nis 12 22:37 system2
-rwxrwxr-x  1 berkan berkan 17320 Nis 13 23:18 terminal
-rw-rw-r--  1 berkan berkan  4549 Nis 13 23:34 terminal.c
-rw-r--r--  1 berkan berkan   730 Nis 13 23:15 tmp.txt
drwxrwxr-x  2 berkan berkan  4096 Nis 12 22:35 .vscode
> 
```

Result

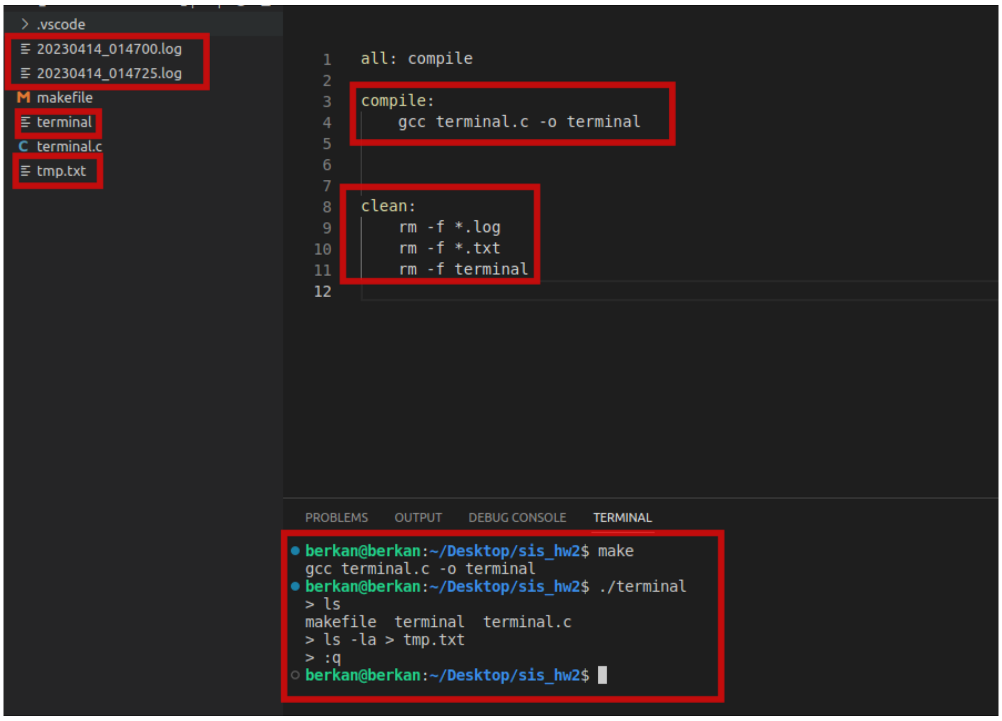
- A shell has been created as desired. Shell commands can be executed. The fulfillment of the requirements specified in the assignment has been illustrated with images. Below is the creation of the file using Makefile
  “make” comment



```
all: compile

compile:
    gcc terminal.c -o terminal



clean:
    rm -f *.log
    rm -f *.txt
    rm -f terminal
```

```
berkan@berkan:~/Desktop/sis_hw2$ make
gcc terminal.c -o terminal
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> ls
makefile  terminal  terminal.c
> ls -la > tmp.txt
> :q
berkan@berkan:~/Desktop/sis_hw2$
```

“**make clean**” comment



```
all: compile

compile:
    gcc terminal.c -o terminal



clean:
    rm -f *.log
    rm -f *.txt
    rm -f terminal
```

```
berkan@berkan:~/Desktop/sis_hw2$ make
gcc terminal.c -o terminal
berkan@berkan:~/Desktop/sis_hw2$ ./terminal
> ls
makefile  terminal  terminal.c
> ls -la > tmp.txt
> :q
berkan@berkan:~/Desktop/sis_hw2$
```