**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.

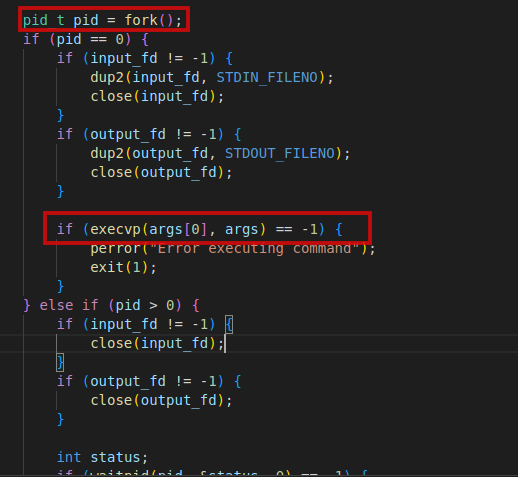
**Requirements**

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.

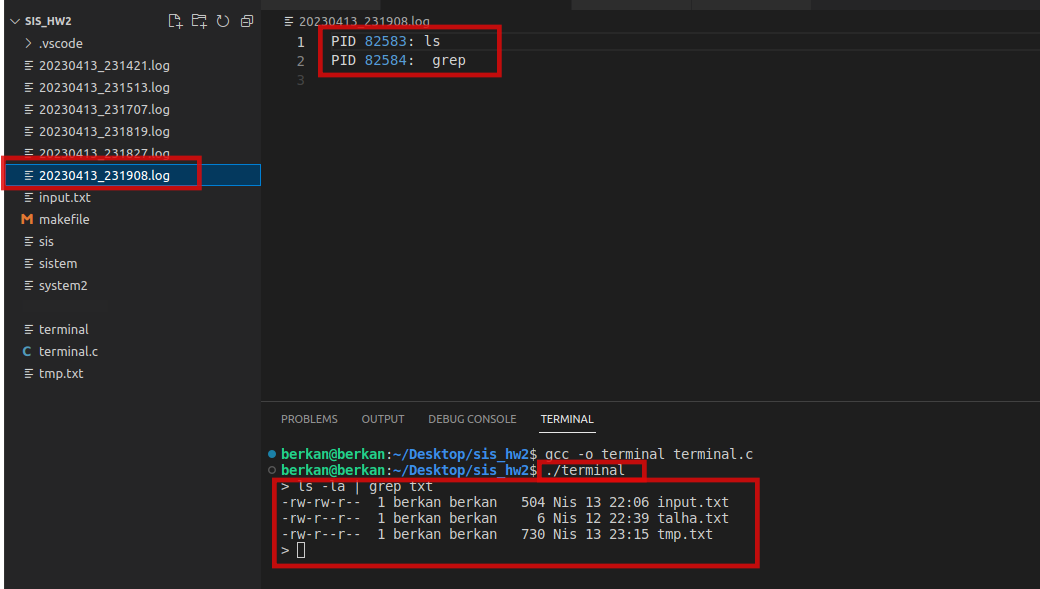
**Tests for Requirements**

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

****

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

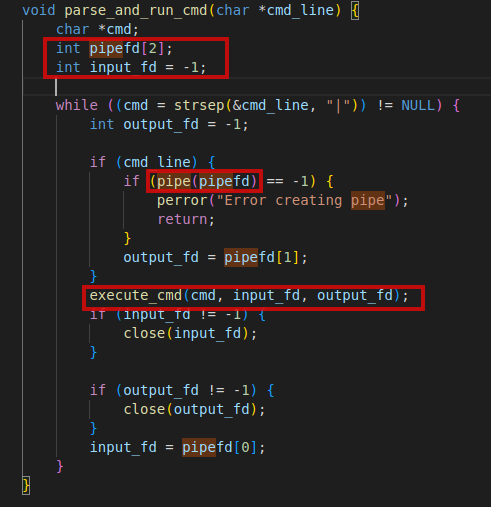


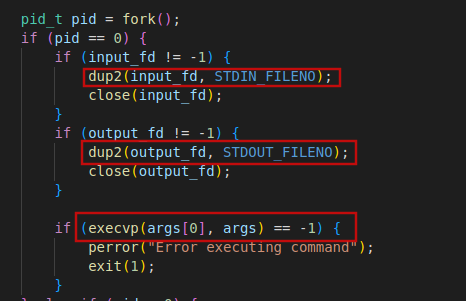
1. **Requirement**

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

**pipes ("|")**

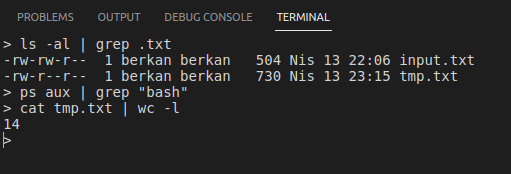
To use pipes, the dub2() function and the pipe() function have been used in the code.





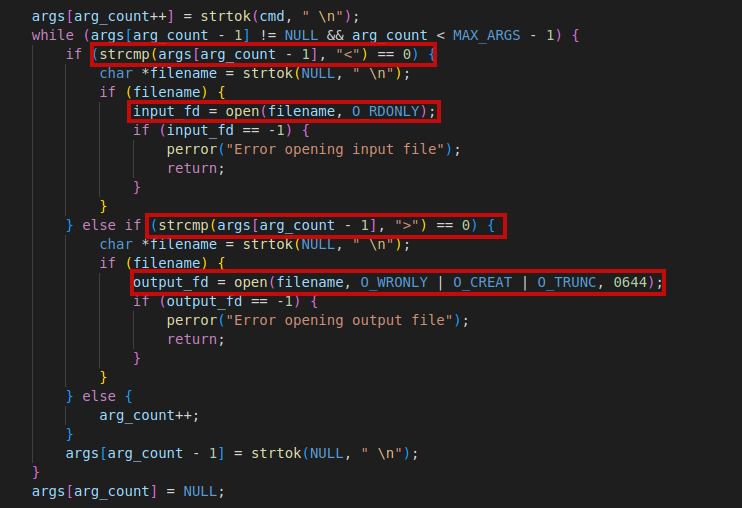
**Example**

* Two different examples using pipes have been given.



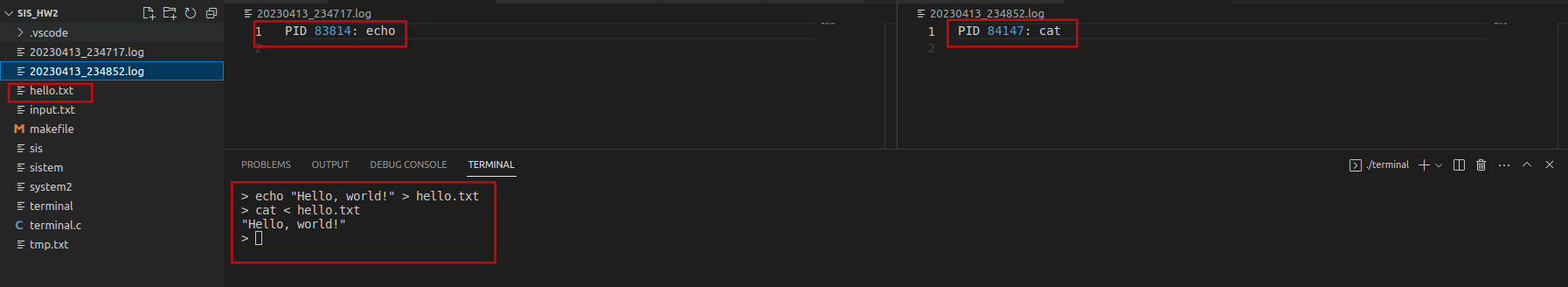
**Redirections ("<", ">")**

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



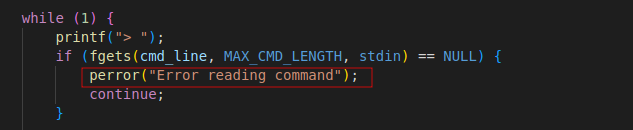
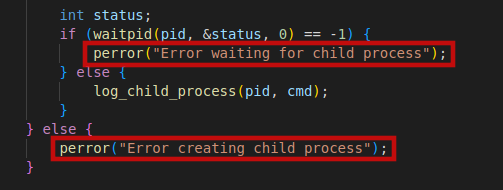
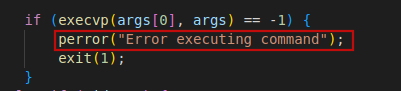
**Example**

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

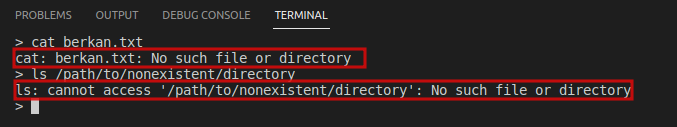
****

1. **Requirement**

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

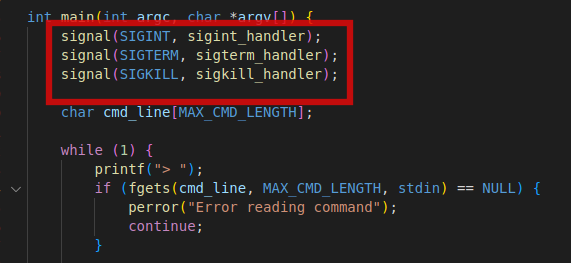
****

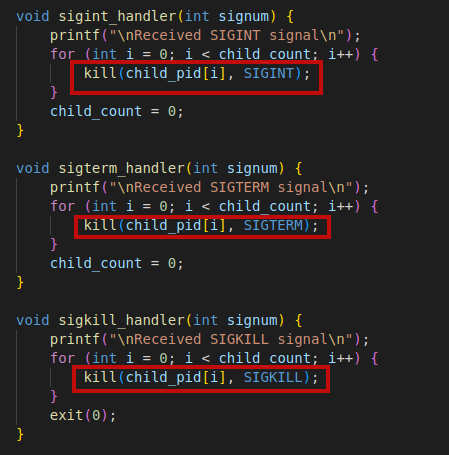
**Example**

****

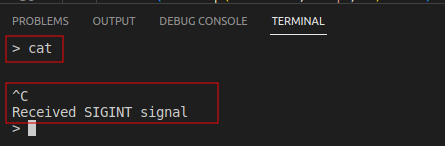
1. **Requirement**

* The desired signal handler functions are listed below.



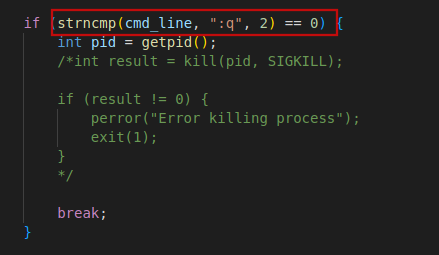


Example

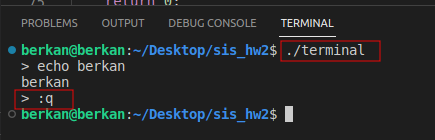


1. **Requirement**

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

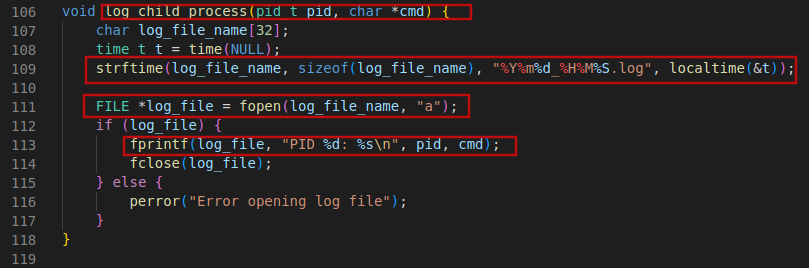


**Example**

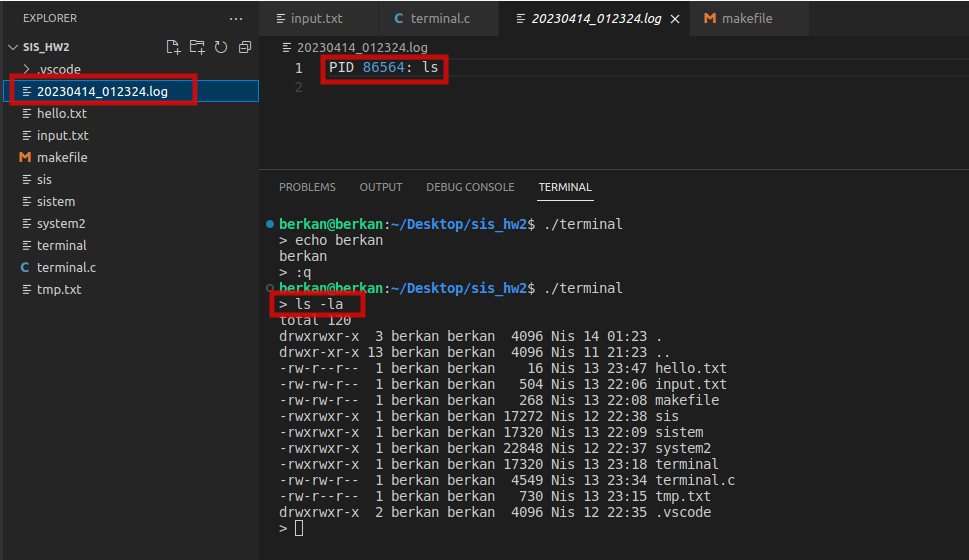
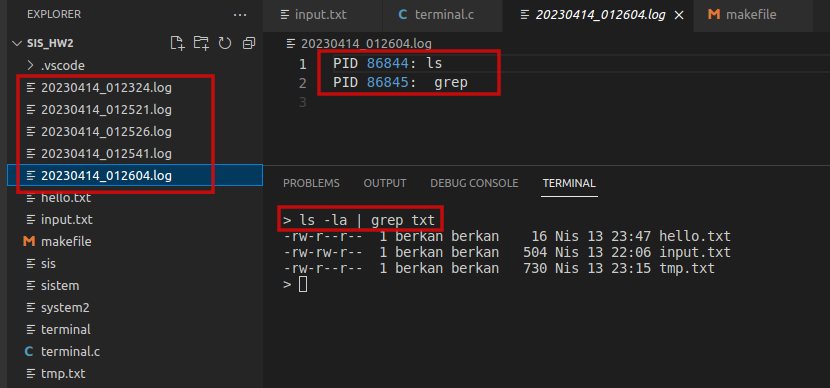


1. **Requirement**

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



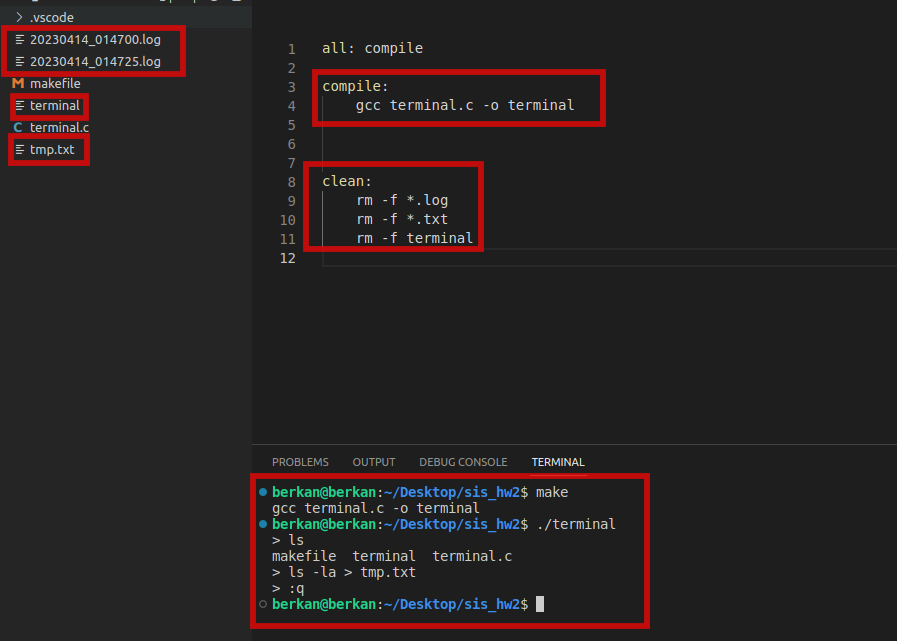
**Example**

****

**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



“**make clean**” comment

