Command-Line User Interface: Shell and Utilities

Halil İbrahim Üstün, Mert Can Pekdemir, Salih Doğaner

I. Problem Statement

The goal of this project was to create a *Command-Line User Interface* that resembles an interface called *shell*, which is used on *Unix-based* operating systems.

II. Subtasks and Major Challenges

A. Prompt

Our shell is supposed to output a prompt when it is ready to accept new commands. The format for the prompt was given to us in the instruction document of the project as follows:

<username>@<hostname>:<working_directory>_$_

A.1 Challenges

The printing prompt was one of the most straightforward challenges we faced during this project. There were determined library functions in C language for this purpose. So, we simply called and assigned them to some variables. Then we implemented those variables into a prompt printing function we created called *printPrompt*. Then we called this *printPrompt* function from the main function to make it possible for the program to print a prompt every time it is ready to take a user input.

B. Program Execution

Whenever the shell is in a ready condition to take an input, a user should be able to enter an input to the program, and the program should execute it in a proper manner. The program should not terminate after either successful or unsuccessful execution.

B.1 Challenges

This part was quite demanding to achieve as it contains the PATH finding process. Sincewe are restricted from using `execvp` system call, which is used to find the path for the given commands automatically, we find the path for the given commands by a function we wrote called *pathFinder*.

The pathfinding process is done as follows: Inside the main function, the program takes a user input, parses it into tokens, and separates the commands from parameters. Then inside our *pathFinder* function, the program fetches the system path and also parses that path variables into tokens. Afterward, the program changes its current directory into that tokenized path variables one by one and tries to match them with a user-inputted command. In the first match, the program stores that path variable and concatenates this path variable with the user command.

By uncommenting line 59, you can see which path variable the program finds for the user-inputted commands. One thing to note here is that in most cases, the commands' matched path will be shown as "/usr/bin" instead of "/bin". This is because in the path variables, the order of "/usr/bin" directory comes before "/bin" directory, and "/bin" is simply a symlink for "/usr/bin". In the Linux shell, you can check the paths of commands by typing `type -a` prior to the command you want to check. Ex: `type -a ls`.

After we find a path for the user-inputted command, we simply call the pathFinder function from our main, create a fork and execute the user-inputted command inside our child process.

C. Background Execution

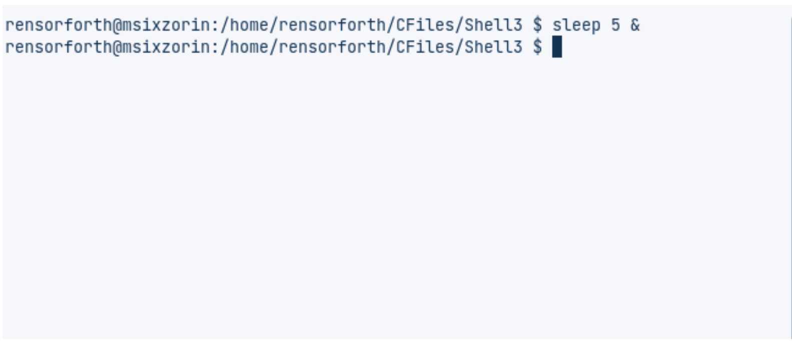
If an ampersand (&) symbol is entered at the end of input as the last parameter, the command should be executed in the background, and the prompt should be printed back to the user in ready condition to take a user input.

C.1 Challenges

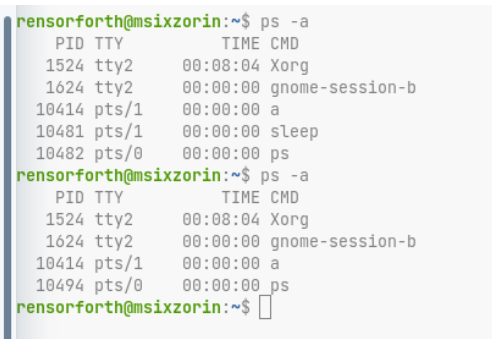
The challenge here was more about preventing the child from being a zombie. Because whenever an execution process is done in the background, the child of that process should be killed for the program to be completed successfully. But the regular execution procedure of the C language is to read lines from top to bottom. Therefore, we needed to go outside of the standard execution flow and use signals because the signals in the C language are executed asynchronously. So that whenever the child finishes its job, SIGCHLD signal is triggered, and the program kills the child process no matter what line the language is currently reading.

We searched the '&' symbol inside a function we created called *checkBg*.

See the demonstration below:



```
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ sleep 5 &
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ 
```



```
rensorforth@msixzorin:~$ ps -a
    PID TTY          TIME CMD
   1524 tty2     00:08:04 Xorg
   1624 tty2     00:00:00 gnome-session-b
  10414 pts/1    00:00:00 a
  10481 pts/1    00:00:00 sleep
  10482 pts/0    00:00:00 ps
rensorforth@msixzorin:~$ ps -a
    PID TTY          TIME CMD
   1524 tty2     00:08:04 Xorg
   1624 tty2     00:00:00 gnome-session-b
  10414 pts/1    00:00:00 a
  10494 pts/0    00:00:00 ps
rensorforth@msixzorin:~$ 
```

We called a sleep function with 5 seconds from our program as a background process, then inside the Linux shell we checked the active processes. After 5 seconds we checked the active processes again and we see that the child process was cleared from the active processes table.

D. ***Input/Output Redirection***

When the program sees one of these characters '<', '>', '>>', it performs the relevant execution depending on what character is found through the line.

'<' symbol is introduced as *input redirection*, both '>' and '1>' symbols are introduced as *output redirection*, '2>' symbol is introduced as *error redirection* and '>>' symbol is introduced as *appending output redirection* process to the program.

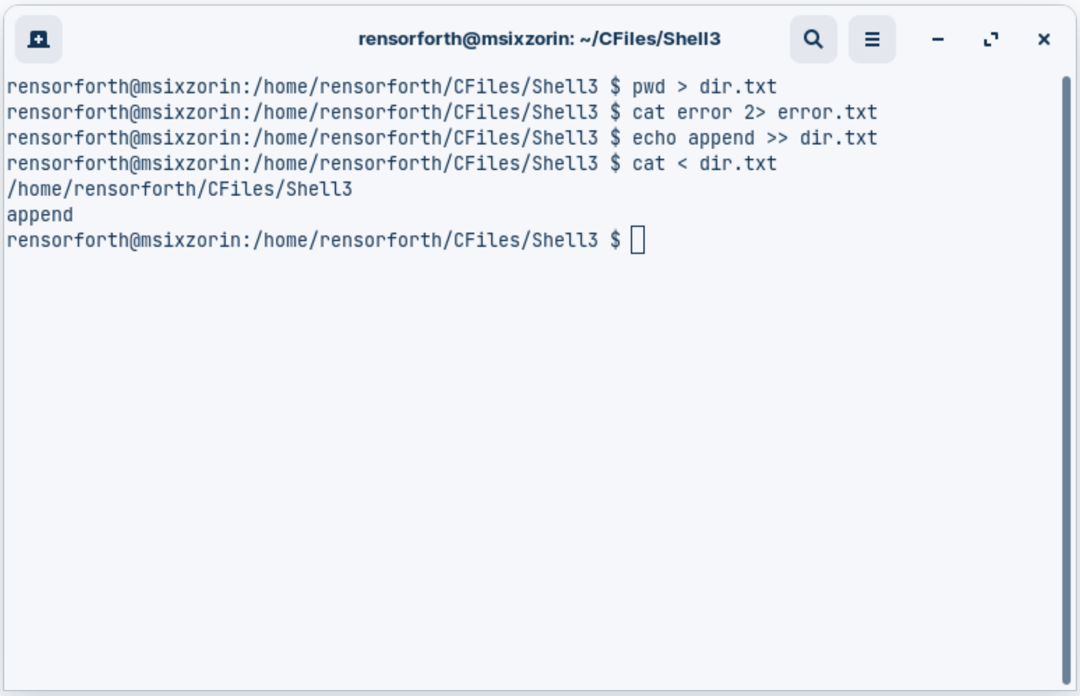
D.1 Challenges

This section's challenge was to deal with file descriptors and learn about flags and access modes (aka `mode_t` values).

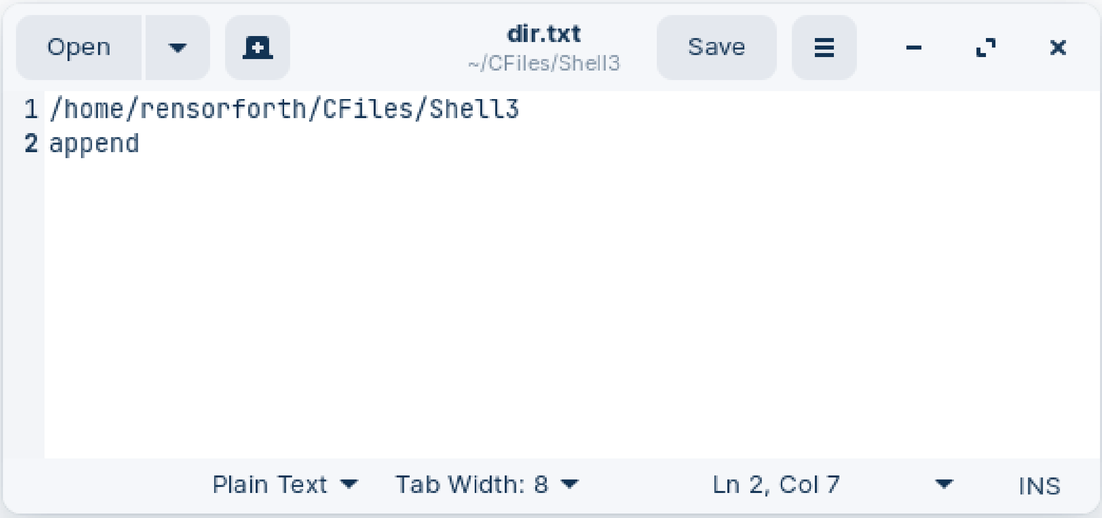
We created a function for redirections called *inOutRedirector*, and inside this function, we introduced the relevant symbols for the redirections. Afterward, we dealt with file descriptors and replaced the standard input (aka *stdin*), standard output (aka *stdout*), and standard error (aka *stderr*) file descriptors of the program with our file descriptors. Then we called this function from the child process inside the main function where commands are executed.

We must mention an important note here: The program supports only one redirection per line, as stated in the *instructions* document. So, multiple redirections are not available.

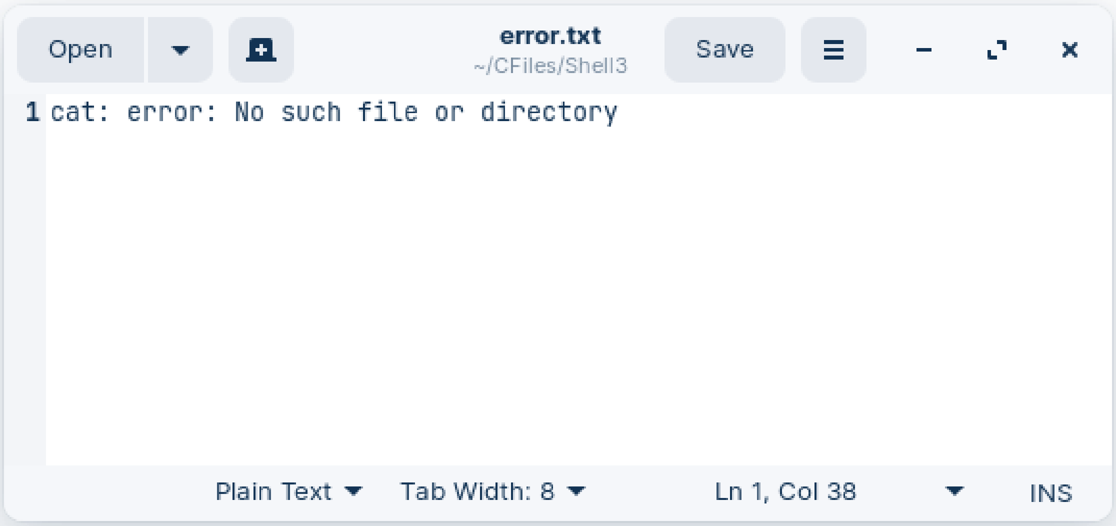
Here is the demonstration of the processes:



```
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ pwd > dir.txt
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cat error 2> error.txt
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ echo append >> dir.txt
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cat < dir.txt
/home/rensorforth/CFiles/Shell3
append
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ 
```



```
/home/rensorforth/CFiles/Shell3
append
```



```
cat: error: No such file or directory
```

E. <u>Pipelines</u>

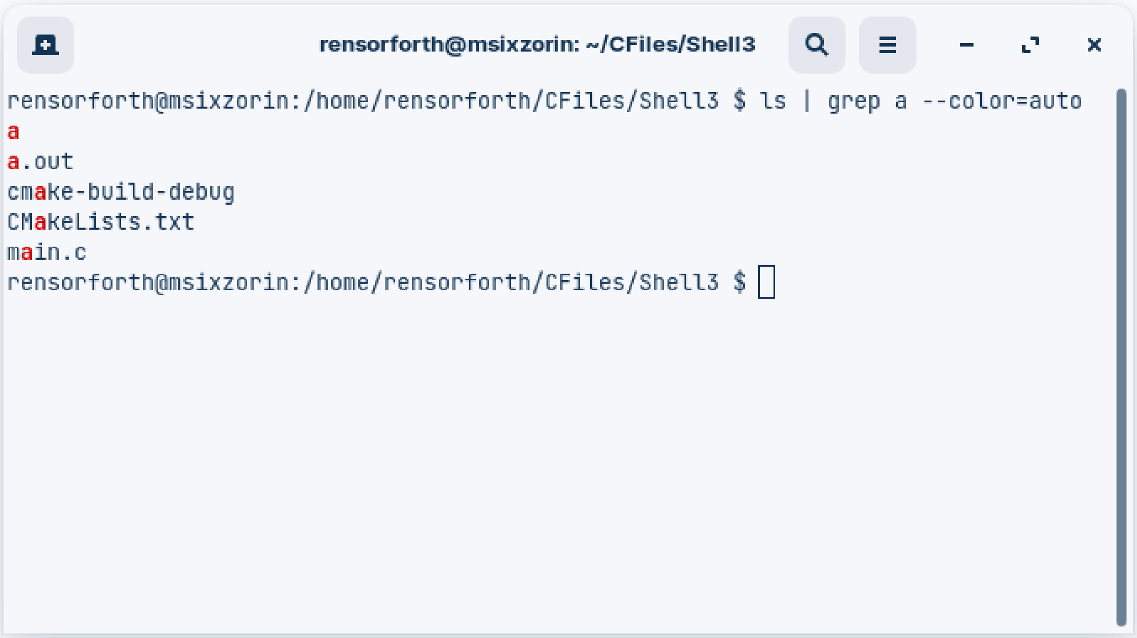
When the program sees the pipeline symbol (|), it creates a pipeline and treats one end of the pipe as a write end and the other as a read end. Then makes an execution process according to the commands written into the two ends of the pipe.

E.1 Challenges

Pipelines look similar to redirects, as they involve a challenge with file descriptors, but here lies the main difference: In the redirect subtask, we only <u>entered</u> commands to the left of the redirection symbols, but in pipelines, we enter commands into both sides of the pipeline.

We handled some part of the process inside a function we created called *pipeExecuter.*

Here is a demonstration of the process:



```
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ ls | grep a --color=auto
a
a.out
cmake-build-debug
CMakeLists.txt
main.c
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ 
```

Also, we need to mention that, as with redirections, pipelines also support only one pipeline per line at a time.

F. exit [n] Built-in

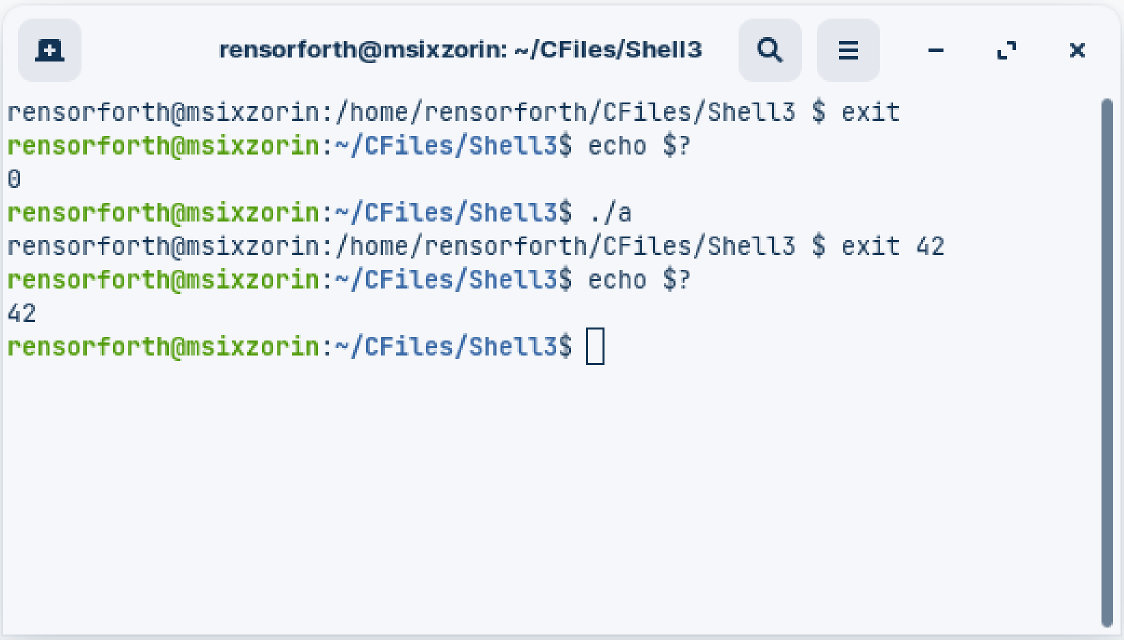
When user input is ‘exit’ shell program is able to terminate itself in a proper manner. The program is also able to support an exit status (n). If no status is indicated, the exit status is assumed to be 0 as default. Otherwise, the program returns the entered exit status to the Linux shell.

F.1 Challenges

This might be the easiest subtask we handled. There were almost no tricky parts.

Wehandled the process inside a function we created called *exitStatus*.

See the demonstration below:



```
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ exit
rensorforth@msixzorin:~/CFiles/Shell3$ echo $?
0
rensorforth@msixzorin:~/CFiles/Shell3$ ./a
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ exit 42
rensorforth@msixzorin:~/CFiles/Shell3$ echo $?
42
rensorforth@msixzorin:~/CFiles/Shell3$
```

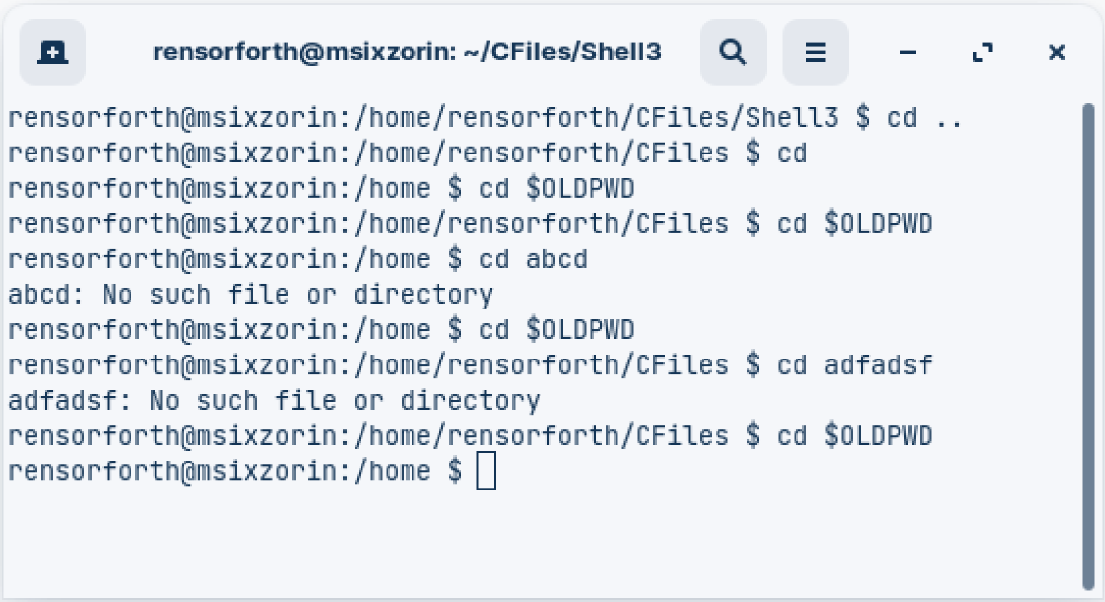
G. *cd [directory] & cd $OLDPWD Built-in*

When a user enters the ‘**cd**’ command and a directory name that follows the command, the program changes the current directory that it is currently in. If the user does not provide any additional directory name, the program changes its current directory to ‘/home’. If the user enters “$OLDPWD” as the directory name, the program handles the process as a special case and changes the current directory to the directory that the program was previously in.

G.1 *Challenges*

Dealing with the ‘**cd**’ command was relatively effortless. However, dealing with the “$OLDPWD” command might be the most challenging and demanding process we dealt with throughout the project. We came up with numerous different algorithm ideas and implemented them consecutively until we became successful. The challenging part was to get back to the directory prior to the current working directory that the program is currently in, even in case of an error while the ‘**cd**’ command is being executed. What made this possible was to save the current directory that the program is in, then change the directory if the ‘**cd**’ command is executed successfully. But suppose there is an error executing the ‘**cd**’ command or any other command. In that case, the program should not save that directory as the prior directory and protect the last saved directory.

See the demonstration below:



```
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cd ..
rensorforth@msixzorin:/home/rensorforth/CFiles $ cd
rensorforth@msixzorin:/home $ cd $OLDPWD
rensorforth@msixzorin:/home/rensorforth/CFiles $ cd $OLDPWD
rensorforth@msixzorin:/home $ cd abcd
abcd: No such file or directory
rensorforth@msixzorin:/home $ cd $OLDPWD
rensorforth@msixzorin:/home/rensorforth/CFiles $ cd adfadsf
adfadsf: No such file or directory
rensorforth@msixzorin:/home/rensorforth/CFiles $ cd $OLDPWD
rensorforth@msixzorin:/home $
```

H. <u>User Input Error Handling</u>

While executing any of the commands, errors should be handled if there is a possible error situation, and the program should not crash.

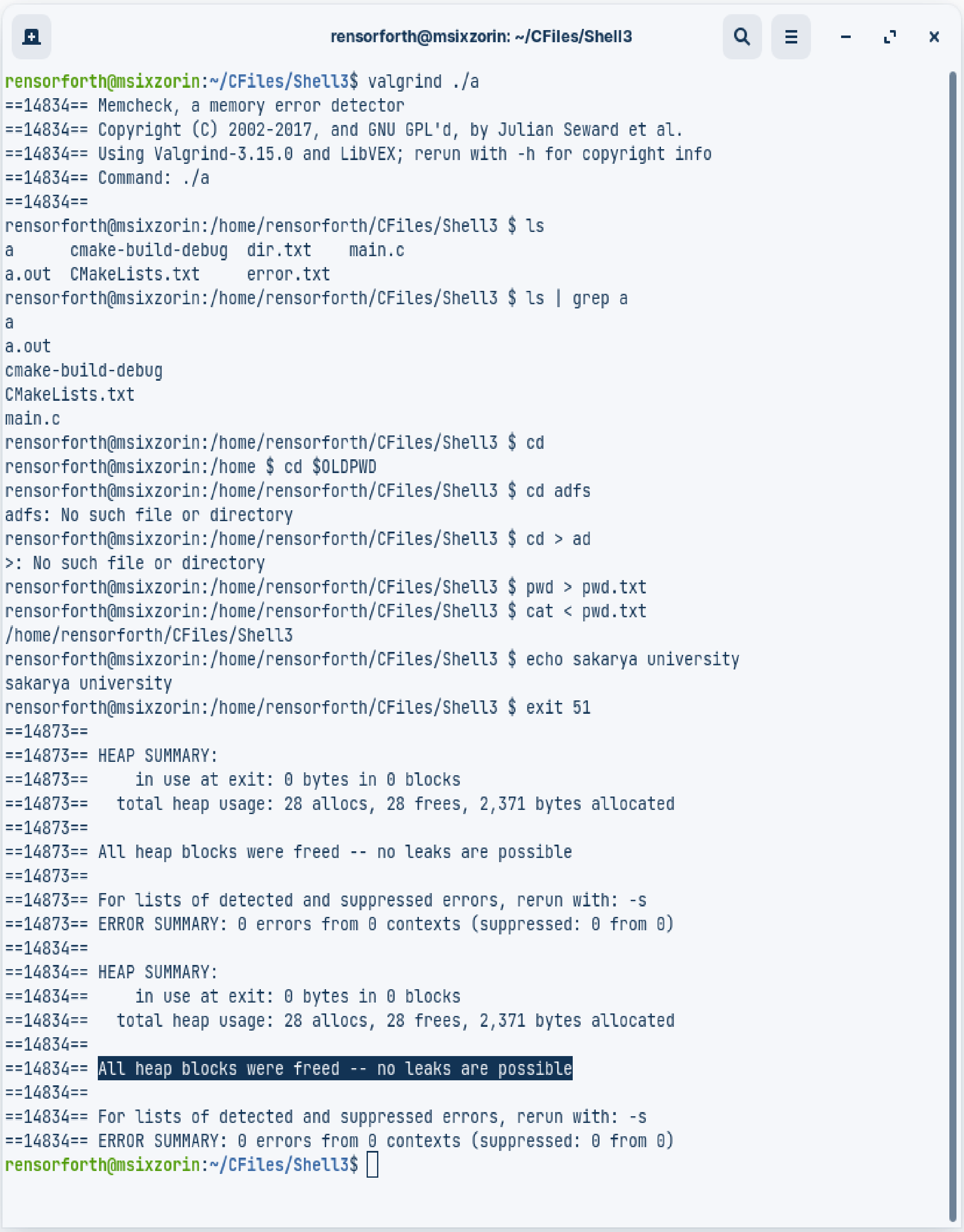
H.1 Challenges

This subtask was one of the most challenging ones since, for every possible error situation that was likely to happen, we needed to handle them and not let the program crash. We tried our best for that subtask and eliminated as many errors as possible. But keep in sight that this is an ordinary program, and program test techniques or test programs are not implemented for the program. In other words, all the possible crash situations are tested by hand, and it is likely that we still might miss some. Currently, we are not detecting any crashes. And all the errors we confront are handled.

I. <u>Memory Leaks</u>

We checked every possible leak situation and handled them inside our program. Every time we allocated memory dynamically or called library functions, such as "*strdup*," which allocates memory dynamically, we freed the memory to prevent memory leaks. We used an open-source application called *Valgrind* to check the memory leaks, and currently, our program has zero leaks. *

See the demonstration below:


rensorforth@msixzorin: ~/CFiles/Shell3

```
rensorforth@msixzorin:~/CFiles/Shell3$ valgrind ./a
==14834== Memcheck, a memory error detector
==14834== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14834== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==14834== Command: ./a
==14834==
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ ls
a      cmake-build-debug  dir.txt    main.c
a.out  CMakeLists.txt     error.txt
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ ls | grep a
a
a.out
cmake-build-debug
CMakeLists.txt
main.c
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cd
rensorforth@msixzorin:/home $ cd $OLDPWD
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cd adfs
adfs: No such file or directory
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cd > ad
>: No such file or directory
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ pwd > pwd.txt
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ cat < pwd.txt
/home/rensorforth/CFiles/Shell3
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ echo sakarya university
sakarya university
rensorforth@msixzorin:/home/rensorforth/CFiles/Shell3 $ exit 51
==14873==
==14873== HEAP SUMMARY:
==14873==     in use at exit: 0 bytes in 0 blocks
==14873==   total heap usage: 28 allocs, 28 frees, 2,371 bytes allocated
==14873==
==14873== All heap blocks were freed -- no leaks are possible
==14873==
==14873== For lists of detected and suppressed errors, rerun with: -s
==14873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==14834==
==14834== HEAP SUMMARY:
==14834==     in use at exit: 0 bytes in 0 blocks
==14834==   total heap usage: 28 allocs, 28 frees, 2,371 bytes allocated
==14834==
==14834== All heap blocks were freed -- no leaks are possible
==14834==
==14834== For lists of detected and suppressed errors, rerun with: -s
==14834== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rensorforth@msixzorin:~/CFiles/Shell3$ 
```

** Please notice that we do these investigations for leak situations by hand and do not use any test program. So, please bear in mind that there might be some holes that we did not test yet that cause memory leaks in the future while you are using the program.*

III. Group Member Contributions

- **Halil İbrahim Üstün**
    1. Path Finding
    2. Pipelines
    3. Input/Output Redirection
    4. Background Execution
    5. cd & cd &OLDPWD Built-ins
- **Mert Can Pekdemir**
    1. Program Execution
    2. MakeFile
    3. exit [n] Built-ins
- **Salih Doğaner**
    1. Printing Prompt
    2. Error Handling
    3. Memory Leaks