Project: Operating systems project, Phase 1

Professor: Azzam Mourad

Lab Instructors: Dena Ahmed

Due Date: Mar 10, 2022

Authors:

Jeongin Lee (jl11640)

Nouf Alabbasi (naa475)

About the shell

How to run the code

1. Download all the needed file
2. In the terminal go to the directory that contains the files
3. Type:
   1) make
   2) ./shell

What commands can be used in this shell

Single commands:

i. ls
ii. cd
iii. cat
iv. find
v. clear
vi. pwd
vii. mkdir
viii. rm
ix. echo
x. move
xi. grep

Redirection:

Example:

Output:

- echo hello > new-file
- ls frhtrhrjy 2> errout-file

Input

- cat < new-file

Pipes:

example:

1 pipe:

- echo hello world | grep hello

2 pipes:

- echo hello world | grep hello | grep world

3 pipes:

- ls | grep f | grep n | grep ne

Test Cases and execution

Testing for correct output for correct commands:

Test case 1:

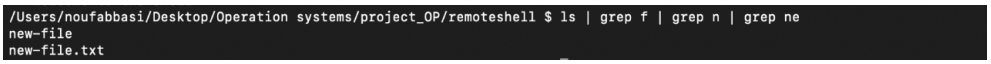
Description: simple single command
Input: ls
output: README.md command.c command.h command.o makefile shell shell.c shell.o
Input is excepted? Yes

Test case 2:

Description: 2 pipes
Input: echo hello world | grep hello | grep world
output: hello world
Input is excepted? Yes

Test case 3:

Description: 3 pipes
Input: ls | grep f | grep n | grep ne
output: new-file
new-file.txt
Input is excepted? Yes



Test case 4:

Description: Output Redirection
Input: echo hello > new-file
output: file is creator containing the string “hello”
Input is excepted? Yes

Test case 5:

Description: Input Redirection (where new file is an existing file containing the string “hello”)
Input: cat < new-file.txt
output: hello
Input is excepted? Yes

Testing error messages for incorrect command input

Test case 1:

Description: command not in list
Input: touch text.txt
output: Invalid command : "touch"
Error was handled? Yes

Test case 2:

Description: command not in list
Input: cp -r old bu
Output: Invalid command : "cp"
Error was handled? Yes

Test case 3:

Description: wrong spelling of mkdir
Input: mkdr
Output: Invalid command : "mkdr"

Error was handled? yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ mkdr
Invalid command : "mkdr"
```

Test case 4:

Description: user input is empty string
Input:
Output: no output
Error was handled? Yes, the program doesn't produce a segmentation error

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $
```

Testing wrong argument

Test case 1:

Description: call non exisiting file(in commands that require an existing file)
Input: cat NonExitsitngFile
Output: cat: NonExitsitngFile: No such file or directory
Error was handled? Yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ cat NonExitsitngFile
cat: NonExitsitngFile: No such file or directory
```

Test case 2:

Description: call non exisiting file(in commands that require an existing file)
Input: cat < new-file
Output: Failed to open file with filename : "new-file"
Error was handled? Yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ cat < new-file
Failed to open file with filename : "new-file"
```

Test case 3:

Description: do no provide required parameters
Input: mkdir
Output: usage: mkdir [-pv] [-m mode] directory …
Error was handled? Yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ mkdir
usage: mkdir [-pv] [-m mode] directory ...
```

Test case 4:

Description: do no provide required parameters
Input: Cat
Output: none and the command line doesn't ask for more inputs
Error was handled? The C shell's reaction is like the terminal's, so it was handled the same way.

Mechanisms used to ensure code doesn't terminate or crash if an error occurs:

1. Using error messages when fork, pipe, or opening file fails and so on.

The structure:

Getting input and parsing it:

After the user types in their input in the command line, the function get_user_input(char * buffer, size_t bufsize) is used to read the input and store it in a string buffer. In this function the strspn(...); C function was used to remove the newline character from the user's input after it is read. Then the user's input is passed to the get_argument_list(...) that we created. This function uses strtok(...) to break the user's input at specified separators and store them in an array. This produces an array with the commands and their arguments.

Running single commands:

To run single commands the user enters into the shell we used execvp with the command and it's arguments. We created a child process that handled this. For the "cd" command we used the chdir(args[1]) c function to change the directory. Creating a child to handle the exec is important as the exec exits the process after it successfully runs. This means that running it in the parent process could risk terminating the parent process after the exec successfully.

Redirection:

The check_if_io_redirection(...) function was used to check whether the user's input contained a redirection sign. We then used the C function strstr(..) to create a pointer to the first occurrence of the signs ">" "<". Then we stored the file name in a string and removed any whitespaces from it. After that, we used the "pos" (the index of the redirection sign) to remove everything behind the redirection sign, including the sign itself from the buffer. Then the filename is returned. Two other functions then take the filename and take care of redirecting the STDIN and out STDOUT values into the commands that are to be executed.

For Input redirection (int redirect_input(char * filename):

In this function, the file is opened in read-only mode and then the content is redirected into the specified command. For example "cat < new-file" would mean that the

content of “new-file” could be used in the “cat” command/process. The function returns default stdin value to restore the default input for next commands.

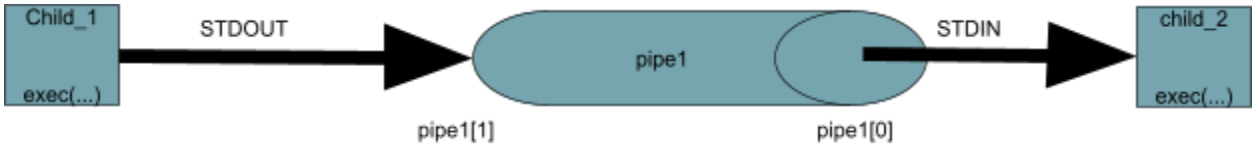
For Output redirection (int redirect_output(char * filename):

In this function a file is created and the output of the first command is written to it. For example, for “echo hello > new-file” the file “new-file” would be created and the output of “echo hello” would be written to it. This was done by opening the file in write mode then redirecting the output(STDOUT) of the command into the specified file. The function returns default stdout value to restore the default output for next commands.

Pipes:

For the pipes, the user’s input was separated into different strings with the “ | “ as the separator (for example if the command was “Echo hello world | grep hello” then then “Echo hello world” would be stored in the index 1 in the $array_1$ and “grep hello” would be stored in index 2). Then each element in the array would be moved to a new array where each word would represent an element in the array. (for example the string “Echo hello world” is divided and moved to $array_2$ where $array_2[0]$= “Echo”, $array_2[1]$= “hello”, $array_2[2]$= “world”, and the string “grep hello” is divided and moved to $array_3$ where $array_3[0]$= “grep” and $array_3[1]$= “hello”).
For the case where only one pipe we do the following:

To execute the commands that utilize pipes we need to create a child that executes the first command before the pipe(using exec) and writes the STDOUT results into the write end of the pipe. Then the parent takes the input needed for the exec from the read end of the pipe and executes the next command after the pipe.



For multiple pipes, we would need to loop over the steps as many times as needed. In other words, the loop would continue the process of forking a child, executing the command(based on the input from STDIN, and sending the output to STDOUT.
Below is a diagram of what the commands separated by 2 pipes would look like: