

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)

Phase 1	3
How to run the code	3
What commands can be used in this shell	3
Single commands:	3
Redirection:	3
Pipes:	3
Test Cases and execution	4
Testing for correct output for correct commands:	4
Test case 1:	4
Test case 2:	4
Test case 3:	4
Test case 4:	5
Test case 5:	5
Testing error messages for incorrect command input	5
Test case 1:	5
Test case 2:	5
Test case 3:	5
Test case 4:	6
Testing wrong argument	6
Test case 1:	6
Test case 2:	6
Test case 3:	6
Test case 4:	6
Mechanisms used to ensure code doesn't terminate or crash if an error occurs:	7
The structure:	7
Getting input and parsing it:	7
Running single commands:	7
Redirection:	7
For Input redirection (int redirect_input(char * filename):	7
For Output redirection (int redirect_output(char * filename):	8
Pipes:	8
Phase 2	9
Socket Communication	9
About the program	9
How to run the code:	9
Steps were taken in the client:	9
Steps were taken in the server/shell:	10
Test Cases and execution	11
Testing for correct output for correct commands:	11
Test case 1:	11
Test case 2:	11

Test case 3:	11
Test case 4:	11
Test case 5:	12
Testing error messages for incorrect command input	12
Test case 1:	12
Test case 2:	12
Test case 3:	13
Test case 4:	13
Testing wrong argument	13
Test case 1:	13
Test case 2:	13
Test case 3:	14
Test case 4:	14
Mechanisms used to ensure code doesn't terminate or crash if an error occurs:	14

Phase 1

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 expected? Yes

```
Welcome to the shell
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ ls
README.md  command.c  command.h  command.o  makefile   shell      shell.c    shell.o
```

Test case 2:

Description: 2 pipes

Input: echo hello world | grep hello | grep world

output: hello world

Input is expected? Yes

```
hello world
```

Test case 3:

Description: 3 pipes

Input: ls | grep f | grep n | grep ne

output: new-file

new-file.txt

Input is expected? Yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ ls | grep f | grep n | grep ne
new-file
new-file.txt
```

Test case 4:

Description: Output Redirection

Input: echo hello > new-file

output: file is created containing the string "hello"

Input is expected? Yes

```
(base) noufabbasi@Noufs-MacBook-Pro remoteshell % echo hello > new-file
(base) noufabbasi@Noufs-MacBook-Pro remoteshell % cat new-file
hello
```

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 expected? Yes

```
(base) noufabbasi@Noufs-MacBook-Pro remoteshell % cat < new-file.txt
hello
```

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

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ touch text.txt
Invalid command : "touch"
```

Test case 2:

Description: command not in list

Input: cp -r old bu

Output: Invalid command : "cp"

Error was handled? Yes

```
/Users/noufabbasi/Desktop/Operation systems/project_OP/remoteshell $ cp -r old bu
Invalid command : "cp"
```

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 $ mkdir
Invalid command : "mkdir"
```

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

```
Welcome to the shell
/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 existing 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 existing 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.

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

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 its 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.

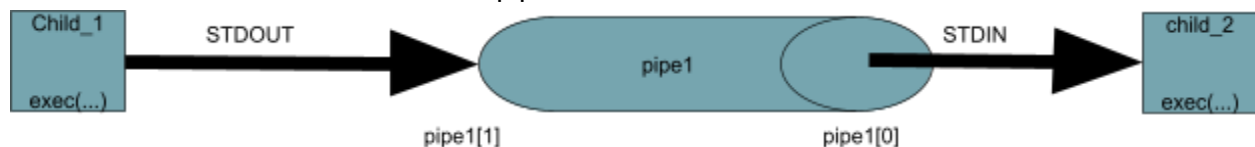
For Output redirection (`int redirect_output(char * filename):`)

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:

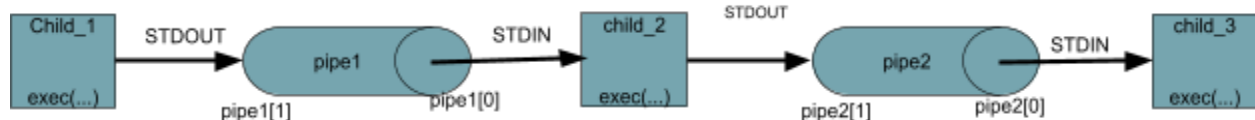
For the pipes, the user’s input was separated into different strings with the “|” as the separator (for example if the command was “Echo hello world | grep hello” then then “Echo hello world” would be stored in the index 1 in the array₁ and “grep hello” would be stored in index 2). Then each element 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 array₂ where array₂[0]= “Echo”, array₂[1]= “hello”, array₂[2]= “world”, and the string “grep hello” is divided and moved to array₃ where array₃[0]= “grep” and array₃[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:



Phase 2

Socket Communication

About the program

Sockets were used in order to allow the client side to communicate with the server-side. The client received the user's input sends it to the server. The server then executes the command and returns the output to be printed on the client end. Pipes were used here to allow the command output to be stored in a string that was sent to the client socket to be printed.

How to run the code:

1. Download all the needed file
2. In two separate terminals
 - a. go to the directory that contains the files
 - b. Type:
 - i. Make
3. in the first terminal type:


```
./server
```
4. In the second terminal type:


```
./client
```
5. Type commands in the client terminal and type "exit" to exit the program

Steps were taken in the client:

1. Define the port
2. Create a socket `socket(AF_INET , SOCK_STREAM, 0);`
 - a. `AF_INET` is the domain/family of the socket which describes the communication properties of the socket.
 - b. `SOCK_STREAM` is the type of socket
 - c. `0` is the protocol for connection but is left as zero when using TCP
3. Then we create a struct to control the properties of the socket we want to connect to
 - a. Then we specify the properties like family, port and so on
4. Then we connect to the socket
 - a. `connect(network_socket, (struct sockaddr *) &server_address, sizeof(server_address));`
 - b. Here we specify the socket and pass a pointer to the address struct and the size of the address
5. Then we sent the user's input and received the output of the command

Steps were taken in the server/shell:

1. Define the port
2. Create a socket `socket(AF_INET , SOCK_STREAM, 0);`
 - a. `AF_INET` is the domain/family of the socket which describes the communication properties of the socket.
 - b. `SOCK_STREAM` is the type of socket
 - c. `0` is the protocol for connection but is left as zero when using TCP
3. Then we create a struct to control the properties of the socket we want to connect to
 - a. Then we specify the properties like family, port, and so on
4. Bind the socket to the local address using the socket's name, a pointer to its address struct and its size.
 - a. `bind(shell_socket, (struct sockaddr *) &shell_address, sizeof(shell_address))`
5. Listen for a connection
6. Accept the connection
7. Receive user input from the client-side, run the command and send output to the client through the socket

Test Cases and execution

Testing for correct output for correct commands:

Test case 1:

Description: simple single command

Input: ls

output: README.md client client.c command.c command.h command.o
 makefile server server.c shell.c

Input is expected? Yes

```
$ ls
README.md
client
client.c
command.c
command.h
command.o
makefile
server
server.c
shell.c
```

Test case 2:

Description: 2 pipes

Input: echo hello world | grep hello | grep world

output: hello world

Input is expected? Yes

```
$ echo hello world | grep hello | grep world
hello world
```

Test case 3:

Description: 3 pipes

Input: ls | grep c | grep m | grep nd

output: command.c command.h command.o

Input is expected? Yes

```
$ ls | grep c | grep m | grep nd
command.c
command.h
command.o
```

Test case 4:

Description: Output Redirection

Input: echo hello > new-file

output: file is creator containing the string "hello"

Input is excepted? Yes

```
$ echo hello > new-file
```

```
$ ls
README.md
client
client.c
command.c
command.h
command.o
makefile
new-file
server
server.c
shell.c
```

```
$ cat new-file
hello
```

Test case 5:

Description: Input Redirection (where new file is an existing file containing the string "hello")

Input: cat < new-file.txt

output: hello world

Input is excepted? Yes

```
$ echo hello world > new-file.txt
```

```
$ cat < new-file.txt
hello world
```

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

```
$ touch text.txt
Invalid command : "touch"
```

Test case 2:

Description: command not in list

Input: cp -r old bu

Output: Invalid command : "cp"

Error was handled? Yes

```
$ cp -r old bu
Invalid command : "cp"
```

Test case 3:

Description: wrong spelling of mkdir

Input: mkdr

Output: Invalid command : "mkdr"

Error was handled? Yes

```
$ 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

```
$
```

```
$
```

```
$ █
```

Testing wrong argument

Test case 1:

Description: call non existing file(in commands that require an existing file)

Input: cat NonExitsitngFile

Output: cat: NonExitsitngFile: No such file or directory

Error was handled? Yes

```
$ cat NonExitsitngFile
cat: NonExitsitngFile: No such file or directory
```

Test case 2:

Description: call non existing file(in commands that require an existing file)

Input: cat < nonexisting-file

Output: Failed to open file with filename : "nonexisting-file"

Error was handled? Yes

```
$ cat < nonexisting-file
Failed to open file with filename : "nonexisting-file"
```

Test case 3:

Description: do not provide required parameters

Input: mkdir

Output: usage: mkdir [-pv] [-m mode] directory ...

Error was handled? Yes

```
$ mkdir
usage: mkdir [-pv] [-m mode] directory ...
```

Test case 4:

Description: do not 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.

```
$ cat
```

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

1. Redirected error messages to the client by `dup2(fd[1], STDERR_FILENO);`