

Operating Systems

UNIX Shell

Project II

I. How to run:

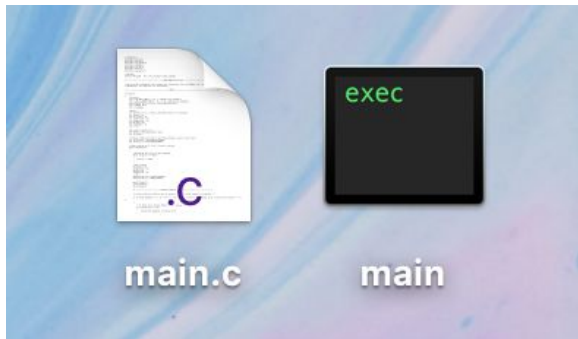
Step 1: Put the main.c file on the desktop

Step 2: On the terminal, enter the commands:

- `cd desktop` (in order to change directory to the desktop)
- `gcc main.c -o main` (main.c is the input file name, and main is the output's name)

```
Last login: Thu Nov  7 18:51:17 on ttys002
[Ashrafs-MacBook:~ ashrafelhawary$ cd desktop
[Ashrafs-MacBook:desktop ashrafelhawary$ gcc main.c -o main
```

This will result in an exec file called main that will appear on the desktop.



II. Code Description:

- **Parser Function**

The parser function dissects the command string into tokens by using the `strtok_r` function and the delimiter “ ” (space) to separate each token and the next. We traverse the string token after token and if the token we are on is not one of the following symbols: ‘ < ’, ‘ > ’, ‘ & ’, ‘ | ’, then the token is saved to the string `args`, with `NULL` being saved between each token and the next.

If the token we are on however happens to be:

1. ‘ < ’ or ‘ > ’ : The token that comes after is the symbol is the file name, which is stored in a variable called `file_name`. The boolean variable that indicates that the corresponding symbol was found turns into `true`.
2. ‘ & ’ : A boolean called `waiting` is turned `true`. The boolean variable that indicates that the corresponding symbol was found turns into `true`.
3. ‘ | ’ : The second command is stored in a different string named `args_two`

- **History Feature**

The history feature allows the user to rerun the most recent command by entering “`!!`”. If there is no command in the history buffer (ie. no previous command run by user), the error “No commands in history.” will be displayed, and the program continues as usual. Otherwise, the most recent command, which is then inside the history buffer, will be placed in the command buffer to be executed like normal.

- **Execution**

→ Firstly, it is checked if the command string is “!!” if so, the history feature mentioned above is provided.

→ If not, the command is parsed to see which kind of command it is.

1. **Exit:**

The program is terminated by setting `should_run` to zero, so the while loop that keeps the program running is no longer entered.

2. **Output / Input:**

Output:

If the boolean value that corresponded to there existing a “>” (which redirects the **output** of a command to a file) is true: `file_desc = open(fileName, O_WRONLY | O_APPEND | O_CREATE | O_TRUNC)`; stores the file descriptor that refers to the open file. Next, `dup2(file_desc, STDOUT_FILENO)`; instructs that the output of command written to the terminal to now be output to the opened file instead of the terminal.

Input:

If the boolean value that corresponded to there existing a “<” (which redirects the **input** to a command from a file) is true: we similarly instruct: `file_desc = open(file_name, O_RDONLY | O_CREAT)`; Next, `dup2(file_desc, STDIN_FILENO)`; instructs that the input of the command come from the open file’s descriptor rather than from the terminal.

Initially, the original `STDOUT` and `STDIN` were saved using the `dup` function and after the above is run, they are set back using `dup2` in the iteration after the input/output instruction so that the output and input is set back to the terminal.

3. **Non-waiting:**

If the `non_waiting` boolean is true which means that there was a “&” during the parsing: the waiting function is skipped. Meaning that the parent no longer waits for the child process to be finished.

4. Piped:

The command is piped if it has the symbol “|” when parsing. Its corresponding boolean variable *command_two* therefore, turns into true and the first command (before the “|”) is stored into *args*, and the second is stored into *args_two* (after “|”) and another child is forked using *fork()*, and the pipe where the two children communicate is setup using *pipe()*. The first child then executes and then communicates its output to the second child and then the second child is executed. At the end, the pipe is closed using *close()* and *wait()* is invoked to ensure the termination of any zombie processes.

5. Normal:

In the case of a normal command: ie. not an exit, history, and has none of the aforementioned symbols:

All values are reset back to their original state, the boolean values are reset to false, the args string is reset to NULL. Next, the command is then parsed and the boolean returned values are how we know it is a normal command. Then, the command is saved in the history buffer. *fork()* is then executed to create a child process, and the parent waits for this child using *wait()*, then the child process is executed using *execvp()*.

III. Tests:

- **Exit:**

```
osh>exit
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.
Deleting expired sessions...74 completed.

[Process completed]
```

Here, the “exit” command is tested, which terminates the program.

- **History Feature**

```
Last login: Thu Nov  7 18:12:53 on ttys002
/Users/ashrafelhawary/Desktop/main ; exit;
Ashrafs-MacBook:~ ashrafelhawary$ /Users/ashrafelhawary/Desktop/main ; exit;
osh>warning: this program uses gets(), which is unsafe.

osh>echo HELLO WORLD
HELLO WORLD
osh>!!
HELLO WORLD
osh>
```

Here the most recent command was “echo HELLO WORLD” so entering the second command “!!”, reruns “echo HELLO WORLD”

- **History Feature Error**

```
Last login: Thu Nov  7 18:12:17 on ttys002
/Users/ashrafelhawary/Desktop/main ; exit;
Ashrafs-MacBook:~ ashrafelhawary$ /Users/ashrafelhawary/Desktop/main ; exit;
osh>warning: this program uses gets(), which is unsafe.

osh>!!
No commands in history.
osh>
```

Here, the command “!!” was entered with no previous command. Therefore, the history buffer is empty, so an error message “No commands in history” is output

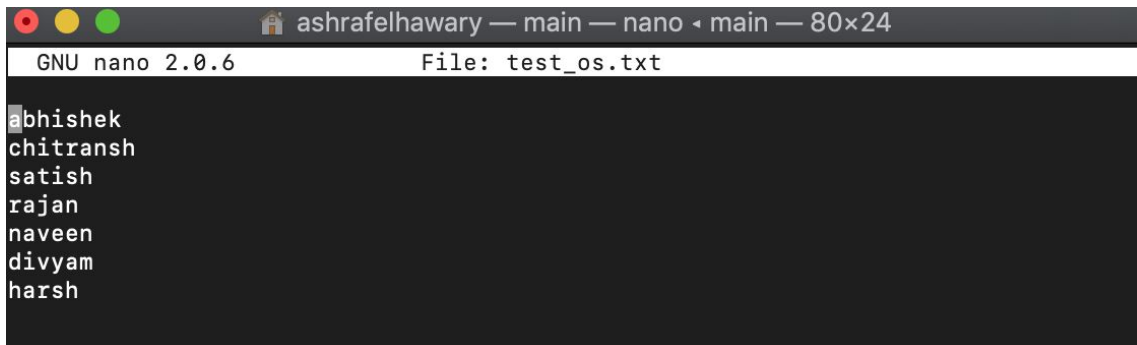
- **Touch, Input, Cat**

```
Last login: Thu Nov  7 16:56:01 on ttys001
/Users/ashrafelhawary/Desktop/main ; exit;
Ashrafs-MacBook:~ ashrafelhawary$ /Users/ashrafelhawary/Desktop/main ; exit;
osh>warning: this program uses gets(), which is unsafe.

osh>touch testing.txt
osh>echo Hello World! > testing.txt
osh>cat testing.txt
Hello World!
```

Here the command “touch testing.txt” is entered which creates the file testing.txt. Next, the command “echo Hello World! > testing.txt” writes the string “Hello World!” as input to the testing.txt file. Lastly, the cat (concatenate) command “cat testing.txt” is run which reads the data inside testing.txt and outputs it.

- **Nano**

A screenshot of a macOS terminal window with a dark background. The title bar shows 'ashrafelhawary — main — nano — 80x24'. The terminal window displays the GNU nano 2.0.6 editor interface. The file being edited is 'test_os.txt'. The content of the file is a list of names: abhishek, chitransh, satish, rajan, naveen, divyam, and harsh, each on a new line. The cursor is positioned at the end of the first line, 'abhishek'.

Here “nano test_os.txt” was entered, which displays the content of the file test_os.txt

- **Sort**

```
osh>nano test_os.txt
osh>sort < test_os.txt
abhishek
chitransh
divyam
harsh
naveen
rajan
satish
osh>
```

After the “nano test_os.txt” was run (whose file contents were shown in the above bullet point), the command “sort < test_os.txt” was entered, which then sorted the words inside test_os.txt alphabetically and output them

- **ls, wc (Pipeline)**

```
Last login: Thu Nov  7 18:50:24 on ttys002
Ashrafs-MacBook:~ ashrafelhawary$ /Users/ashrafelhawary/Desktop/main ; exit;
osh>warning: this program uses gets(), which is unsafe.

osh>ls
Applications                file2.txt
Appointment.pdf             flush
CLionProjects               iCloud Drive (Archive)
Desktop                     in.txt
Documents                   my.txt
Downloads                   mysql
Hotel Booking.pdf           mysqladmin
Library                     out.txt
Movies                      quit
Music                       status
Part 4 Brute Force Algorithms.pdf test
Pictures                    test_os.txt
Public                      testing.txt
VirtualBox VMs              use
file1.txt                   verilog_test
osh>ls | wc -l
    30
```

The first command “ls” output the list of files and directories. The second command “ls | wc -l” uses “wc -l” to count the number of lines on the output of “ls”, which here turns out to be 30.

- **& (Non-waiting)**

```
Last login: Thu Nov  7 21:58:20 on ttys002
/Users/ashrafelhawary/Desktop/main ; exit;
Ashrafs-MacBook:~ ashrafelhawary$ /Users/ashrafelhawary/Desktop/main ; exit;
osh>warning: this program uses gets(), which is unsafe.

osh>cat testing.txt &
osh>Hello World!
```

Here the parent does not wait for the child process, so before the child finishes execution, “osh>” is output.