COMP 304 Shelly: Project 1

Due: Saturday March 17th, 11.00 pm

Notes: The project can be done **individually or teams of 2**. You may discuss the problems with other teams and post questions to the OS discussion forum but the submitted work must be your own work. This assignment is worth 9% of your total grade. START EARLY.
Any material you use from web should be properly cited in your report. Any sort of cheating will be harshly PUNISHED.

Corresponding TA for the project : Doğa Dikbayır

Description

The main part of the project requires development of an interactive Unix-style operating system shell, called **shelly** in C/C++. After executing **shelly**, **shelly** will read both system and user-defined commands from the user. The project has four main parts:

Part I

(20 points)
The shell must support the following:

- Use the skeleton program provided as a starting point for your implementation. The skeleton program reads the next command line, parses and separates it into distinct arguments using blanks as delimiters. You will implement the action that needs to be taken based on the command and its arguments entered to **shelly**. Feel free to modify the command line parser as you wish.
- Commandline inputs should be interpreted as program invocation, which should be done by the shell **fork**ing and **exec**ing the programs as its own child processes. Refer to Part I-Creating a child process from Book, page 155.
- The shell must support background execution of programs. An ampersand (&) at the end of the command line indicates that the shell should return the command line prompt immediately after launching that program.
- Use execv() system call (instead of execvp()) to execute common UNIX commands (e.g. ls, mkdir, cp, mv, date, gcc) and user programs by the child process.

The descriptions in the book might be useful. You can read Project 1- Unix Shell Part-I in Chapter 3 starting from Page 154.

Part II

(20 points)

- (10 points) In this part of the project, you will implement *I/O redirection* for your shell. For the I/O redirection if the redirection character is >, the output file is created if it does not exist and truncated if it does. If the redirection symbol is >> then the output file is created if it does not exist and appended if it does. A sample terminal line is given for I/O redirection below:

```
    shelly> program arg1 arg2 > outputfile
```

- (10 points) You are asked to implement the *script* command, which creates a typescript of a terminal section. User gives an output file name as an input to the command, after the execution of the initial script call, you need to record the terminal session into the output file. When the user wants to end the script session, she simply enters exit and the recording is terminated. Here is a sample run of a recording with the script command:

```
    shelly> script outputfile.txt
    shelly> echo "Hello World!"
    shelly> //any other commands ...
    shelly> exit
```

More info on script command can be found there:
https://www.computerhope.com/unix/uscript.htm

Part III

(35 points) In this part of the project, you will implement three new **shelly** commands:

- (20 points) The first command is the **bookmark** command. Similarly to bookmarking webpages in a browser, this feature will enable users to bookmark frequently used commands. See the following example to add a new bookmark and execute it:

```
    shelly> bookmark conSSH "ssh dunat@lufer.hpc.ku.edu.tr"
    shelly> conSSH  //example usage
```

  The command takes the desired bookmark key as the first input and the command that is going to be linked to it as the second input. You should store the bookmarks as key-value pairs where key is the bookmark name and value is the command that is going to be executed.

  The bookmarks should live between shell sessions. That means you need to load the existing bookmarks when shelly is launched, and at the exit you should save the bookmarks to a file if there is any changes. Use the filename **.mybookmarks** and locate it under your *home* folder.

Finally you should implement a deletion functionality for your new bookmark command. Use input parameter $-r$ for remove mode. In this mode, your command will take the bookmark key from the user as an input parameter and delete the corresponding bookmark.

- (10 points) The second command is called **wforecast**. This command will generate a weather report every morning. The user should specify a filename parameter, then the command should save the weather forecast with the specified file name. The program must execute at 9:00 AM every day.

  In order to implement the *wforecast* command, use the *crontab* command provided by Linux and install *curl wttr.in* utility to pull weather info.

  Once curl is installed, you can test the weather package:

```
    $ curl wttr.in/Istanbul
```

  wttr.in package has a .png output mode. Your command should save the weather report in .png format for a nicer formatted output.

  Here is how wforecast command should look:

```
    shelly$ wforecast reportname.png
```

  We strongly suggest you to first explore curl wttr.in and then crontab before starting implementation of this command.

  More info about crontab and curl wttr.in can be found here:
  http://www.computerhope.com/unix/ucrontab.htm
  http://wttr.in/:help

- (5 points) The third command is any new **shelly** command of your choice. Come up with a new command that is not too trivial and implement it inside of **shelly**. Be creative. Selected commands will be shared with your peers in the class. Note that many commands you come up with may have been already implemented in Unix. That should not stop you from implementing your own.

Part IV

(25 points) In this part of your project, you will simulate a game of **Chinese whispers (kulaktan kulağa)** using processes and POSIX message queues.

Your program will first take a word as an input. Then, you will fork n processes where n is the number of characters in the input word. Once the processes are created, you will implement a ring communication between processes: the input word will be passed from the first process to the second, from the second to the third and so on, until it reaches to the n-th process. Finally, the n-th process will pass the word to the first and the ring will be completed. After receiving the word, the first process should print it to the stdout.

Typically in this game, the word that gets passed around gets distorted. Similarly, you will manipulate the word with a probably of 0.5 as it is passed around: i-th process in the ring can change the i-th character of the word to its alphabetical successor, where $1 \leq i \leq n$. For example, if your input word is *computer*, the first process can change the message to *domputer* and pass it to the second process.

You should traverse the processes in increasing PID order, meaning you should start passing the message from the process with the lowest PID to the one with the highest PID.

We also want you to **ONLY** use **mqueue** (POSIX message queue utility) provided by Linux for your interprocess communication. Other than that, you are free to use any data structure and strategy you want to simulate the game.

Last but not least, add a new command **whispers** to your shell to execute your compiled simulation program. This command should only take the input word as a parameter and display the final manipulated version of it after the simulation is completed.

Here is the an example run:

```
    shelly$ whispers "didem"
    diefm
    shelly$
```

Info and examples about mqueue :
https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html

Deliverables

You are required to submit the followings packed in a zip file (named your-username(s).zip) to blackboard :

- .c source file that implements the **shelly** shell. Please comment your implementation.
- .c source file that implements the Chinese whispers game you implemented in Part IV.
- any supplementary files for your implementations (e.g. Makefile)
- a short REPORT briefly describing your implementation, particularly the new command you invented in Part III. You may include your snapshots in your report.
- Do not submit any executable files (a.out) or object files (.o) to blackboard.
- You are required to implement the project on a Unix-based virtual machine or Linux distribution.

- Finally your team will perform a demo of your **shelly** implementation to TAs after the project submission deadline.

GOOD LUCK.