COMP 304 - seashell - Your Custom Shell: Project 1

Due: Tuesday March 30th, 11.00 pm

Notes: The project can be done **individually or in 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. **Any material you use from web should be properly cited in your report. Any sort of cheating will be punished.** This assignment is worth 12% of your total grade. We recommend you to START EARLY.

Corresponding TA for the project : Najeeb Ahmad, nahmad16@ku.edu.tr

Description

This project aims to familiarize you with the Unix system call interface and the shell by letting you implement several features in a shell, called **seashell**, using C/C++. A shell is a program that provides interface to the operating system by gathering input from the user and executing programs based on that input. Once executed, the **seashell** will read both system and user-defined commands from the user. The projects has five parts:

Part I

(25 points)

- 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 in **seashell**. Feel free to modify the command line prompt and parser as you wish.
- Read *"Project 1- Unix Shell and History Feature"* from the book in Chapter 3 starting from Page 157 (9th edition). That will be useful.
- Use **execv()** system call (instead of **execvp()**) to execute common Linux programs (e.g. ls, mkdir, cp, mv, date, gcc) and user programs by the child process. The difference is that **execvp()** will automatically resolve the path when finding binaries, whereas for **execv()** your program should search the path for the command invoked, and execute accordingly.
- 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.

- Implement the *history* command as discussed in *Part II- Creating History Feature* in the book (Page 159).

Part II

(5+10 points)

- In this part of the project, you will implement I/O redirection for **seashell**. For I/O redirection if the redirection character is >, the output file is created if it does not exist and truncated if it does. For the redirection symbol >> the output file is created if it does not exist and appended otherwise. The < character means that input is read from a file. A sample terminal line is given for I/O redirection below:

```
seashell$ program arg1 arg2 >outputfile >>appendfile <inputfile
```

  dup() and **dup2()** system calls can be used for this part.

- In this part, you will handle program piping for **seashell**. Piping enables passing the output of one command as the input of second command. To handle piping, you would need to execute multiple children, and create a pipe that connects the output of the first process to the input of the second process, etc. It is better to start by supporting piping between two processes before handling longer chain pipes. Longer chains generally can be handled recursively. Below is a simple example for piping:

```
seashell$ ls -la | grep seashell | wc
```

Part III

(15 points) In this part of the project, you will implement a new **seashell** command, *goodMorning*:

- This command will take a time and a music file as arguments and set an alarm to wake you up by playing the music using rhythmbox. In order to implement *goodMorning*, you may want to use the crontab command.

Listing 1: Sample usage of goodMorning command

```
seashell> goodMorning 7.15 /home/musics/muse.mp3
```

  Before implementing the new command inside **seashell**, you should get familiar with crontab (if you decide to use crontab) and rhythmbox on a regular shell.

  More info about rhythmbox:
  http://manpages.ubuntu.com/manpages/hardy/man1/rhythmbox-client.1.html
  More info about crontab:
  http://www.computerhope.com/unix/ucrontab.htm

Part IV

(20 points) In this part, you will implement the *kdiff* utility to compare two files in given paths. Sample usage of *kdiff* to compare two files is given in Listing 2. The utility will operate in two modes:

- $-a$: the utility reads the input files (*file1.txt* and *file2.txt*) as text files and compares them line-by-line. If the two files are different, the utility first prints the differing lines from each file and then prints the count of differing lines as shown in Listing 2. The utility checks for file extensions and if they are not .txt, flags an error. If the two files are identical, it displays the message "The two files are identical".
- $-b$: the utility reads the input files as binary files and compares them byte-by-byte. If the two files are different, the utility prints the message "The two files are different in xyz bytes", where xyz is the number of bytes different between the files. The utility accepts files with any extension in this case. If the files are identical, the message saying "The two files are identical" is displayed. Refer to Listing 2 for the sample output.

  If the user does not provide either $-a$ or $-b$, $-a$ is assumed by default.

  Note: You are not allowed to use existing file comparison tools (such as *diff*) to compare the files. We expect you to use C/C++ file I/O operations to complete this task. You may use the existing tools to verify your output.

Listing 2: *kdiff* utility sample output

```
seashell> kidff -a file1.txt file2.txt
The two text files are identical

seashell> kdiff -a file1.txt file2.txt
file1.txt:Line 23: A quick brown fox jumps over the lazy dog
file2.txt:Line 23: A swift squirrel climbs the tree
file1.txt:Line 41: Corona cases today = 1200
file2.txt:Line 41: Corona cases today = 689
2 different lines found

seashell> kdiff -a file.txt file2.txt
The two files are identical

seashell> kdiff -b file1.bin file2.bin
The two files are identical

seashell> kdiff -b file1.bin file21.bin
The two files are different in 8213 bytes
```

Part V

(10 points) In this final part, you will implement a command of your choice in **seashell**. Come with a new, non-trivial command and implement it inside **seashell**. Creativity will be rewarded and selected commands will be shared with your peers in the class. It is possible

that command you come up with is already implemented in Unix. That should not stop you from implementing your own but do not simply execute the existing Unix version of it.

Deliverables

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

- .c source code file that implements the **seashell** shell. Please comment your implementation.
- any supplementary files for your implementations (e.g., Makefile)
- a short REPORT file briefly describing your implementation, particularly the new command you invented in Part V. You may include your snapshots in your report.
- Do not submit any executable files (a.out) or object files (.o) to blackboard. Do not submit the file for the song you use in Part-III.
- You are required to implement the project on a Unix-based virtual machine or Linux distribution.
- (15 points) Finally there will be project demos after the deadline. Demos account for 15 points towards your grade. We expect that both team members have equally contributed to the project and are competent to answer questions on any part of the implementation. TA will direct questions to any members of the team thus team members may receive different grades based on their demo performance.

GOOD LUCK.