COMP 304 Shelldon: Project 1

Due: Wednesday March 27th, 11.59 pm

Notes: The project can be done **individually or as a team 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 harshly PUNISHED.** This assignment is worth 10% of your total grade. We recommend you to START EARLY.

Corresponding TA for the project : Doğa Dikbayır, Office: ENG 230

Description

The main part of the project requires you to develop an interactive Unix-style operating system shell, called **shelldon** in C/C++. After executing **shelldon**, **shelldon** will read both system and user-defined commands from the user. The project has four main parts. We suggest starting with the first part but continue with any part as you like:

Part I

(20 points)
The **shelldon** shell must support the followings:

- 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 **shelldon**. 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.

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

Part II

(5+15 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:

```
    shelldon> program arg1 arg2 > outputfile
```

- In this part, you are asked to add support for the *history* command to **shelldon**. Read Part II- Creating History Feature of Project-1 in Chapter 3 and implement the history feature as described in the book.

Part III

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

- (20 points) The first command is the **codesearch** command. This command is useful to search a specific keyword in your source code files. The command takes an input keyword and scans all the files in the current directory to match the keyword and then it returns the filenames, line numbers where the keyword occurs and finally the line itself.

  In addition, you are required to support recursive and targeted codesearch. In recursive codesearch, if the user runs the command with $-r$ option, then your command should search all the sub-directories under the current directory including the current directory. In targeted codesearch, your command should search for the keyword in a specified file only provided with the $-f$ option.

  Here is an example call:

```
    shelldon> codesearch "foo"
    45: ./foo.c -> void foo(int a, int b);
    92: ./foo.c -> foo(a,b);


    shelldon> codesearch  -r "foo" //recursive usage
    45: ./foo.c -> void foo(int a, int b);
    92: ./foo.c -> foo(a,b);
    112: ./include/util.h -> void foo(int a, int b, int c);
    254: ./lib/x86/lib64.cpp -> foo(A, B, C);

    shelldon> codeseach  "foo" -f  ./include/util.h //targeted usage
    112: ./include/util.h -> void foo(int a, int b, int c);
```

- (10 points) The second command is called **birdakika**. This command schedules a song specified by the user to play for only one minute every day at a time specified by the user. You are required to use *crontab* in this part of the project.

  Here is how **birdakika** command should look like. You can pick any song of your choice:

  ```
      shelldon> birdakika 7.15 seninle_bir_dakika.wav
  ```

  We strongly suggest you to first explore *crontab* before starting implementation of this command. More info about crontab can be found here: Linux Crontab

- (5 points) The third command is any new **shelldon** command of your choice. Come up with a new command that is not too trivial and implement it inside of **shelldon**. Be creative. Selective 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 but do not simply execute the Unix version of it.

Part IV

(25 points) In this part of your project, you will learn how to develop a kernel module and load it into the Linux kernel. You need to be a superuser in order to complete this part of your assignment.

- The kernel module should take a PID as an argument.
- Your job is to find all the oldest children of processes that appear in the subprocess tree by treating PID as the root. In other words, starting from the process with PID provided by the user and treating it as root, the kernel module traverses the process subtree and prints the PIDs and executable names of the oldest children if exist for the visited processes in the process tree.
- You will need to explore the Linux task_struct in $< linux/sched.h >$ to obtain necessary information such as process name, and process start time.
- Test your kernel module first outside of **shelldon** and make sure it works.
- Then, in **shelldon**, implement a new command *oldestchild* which triggers the kernel module you have developed.

  ```
      shelldon> oldestchild  PID
  ```

  which should internally load the kernel module:

  ```
      shelldon> sudo insmod oldestchild  processID=PID
  ```

- When the command is called for the first time, **shelldon** will prompt your sudo password to load the module. Successive calls to the command will notify the user that the module is already loaded.

- If successive calls use a different process ID, then **shelldon** will unload the previously loaded kernel module and load it again with the new argument. If no process ID is provided or the process ID is invalid, print an error message to kernel log.
- **shelldon** will remove the module when the shell is exited.
- You can use **pstree** to check if the process list is correct. Use -p to list the processes with their PIDs.

We **STRONGLY** suggest you start this part as early as possible because it may require a fair amount of researching. These links might be useful:

- Even though we are not doing the same exercise as the book, Project 2 - Linux Kernel Module for Listing Tasks discussion from the book might be helpful for implementing this part.
- Info about Writing a Simple Module:
  http://devarea.com/linux-kernel-development-and-writing-a-simple-kernel-module.
- Info about task linked list (scroll down to Process Family Tree):
  http://www.informit.com/articles/article.aspx?p=368650
- Linux Cross Reference:
  https://elixir.bootlin.com/linux/latest/source
- Passing Arguments to a Kernel Module
  https://www.tldp.org/LDP/lkmpg/2.4/html/x354.htm

Deliverables

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

- .c source file that implements the **shelldon** shell. Please comment your implementation.
- .c source file of the Kernel Module 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. 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.
- Finally there will be project demos after the deadline. We expect that both team members have equally contributed to the project and are competent to answer questions on any parts of the implementation.

GOOD LUCK.