COMP 304 Shellax: Project 1

Due: Sunday November 13th, 2020, 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 **14%** of your total grade. We recommend you to START EARLY.

Responsible TA: Mohammad Issa (missa18@ku.edu.tr), Office: ENG 230

Description

The main part of the project requires you to develop an interactive Unix-style operating system shell, called **Shellax** in C. After executing **Shellax**, it will read commands from the user and execute them. Some of these commands will be *builtin* commands, i.e., specific to **Shellax** and not available in other shells, while others will be normal commands, i.e., they will run programs available in the system (e.g.,`ls`). The project has four main parts (95 points), in addition to a report (5 points). We suggest starting with the first part and continue with other parts.

Part I

(15 points) **Shellax** 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 in **Shellax**. Feel free to modify the command line prompt and parser as you wish.
- Command line inputs, except those matching builtin commands, should be interpreted as program invocation, which should be done by the shell **forking** and **executing** the programs as its own children 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 Linux programs (e.g. ls, mkdir, cp, mv, date, gcc, and many others) 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.

Part II

(5+10 points)

- In this part of the project, you will implement I/O redirection for **Shellax**. 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:

```
$ 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 **Shellax**. 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:

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

Part III

(10+20+10+10 points) In this part of the project, you will implement new **Shellax** commands (builtin commands).

(a) **uniq** (Must be written in C): you are required to implement an application akin to UNIX's `uniq` command. Given sorted lines, `uniq` should print all the unique values without duplicates. `uniq` Must also have an option, `-c, --count`, which if given as an argument to `uniq`, `uniq` will prefix the unique lines by the number of occurrences.

```
$ cat ingredients.txt
Cinnamon
Egg
Egg
Flour
Flour
Flour
Milk
Milk
```

```
$ cat ingredients.txt | uniq
Cinnamon
Egg
Flour
Milk
$ cat ingredients.txt | uniq -c
      1 Cinnamon
      2 Egg
      3 Flour
      2 Milk
```

(b) **chatroom <roomname> <username>** (Must be written in C): you are required to implement a simple group chat command using named pipes. Each user is represented by a named pipe with their name, and each room is represented by a folder which would contain the named pipes of the users who joined it. To send a message, a user will write to all named pipes within the same room. To read a message, the user would read their own named pipe. The rooms are expected to be located under `/tmp/chatroom-<roomname>/`.

- If room does not exist, create a room folder in `/tmp/chatroom-<roomname>`
- If user does not exist, create a user named pipe in `/tmp/chatroom-<roomname>/<username>`
- A user sends a message by iterating over all named pipes within the room's folder and writing to all other named pipes using separate children
- A user receives a message by continuously reading from their named pipe

Here is an example of sample room folder content with a few users joined:

```
$ ls /tmp/chatroom-<roomname>/
mehmet ali ay a osman ahmet
```

Here is an example of a user joining a room:

```
$ chatroom comp304 mehmet
  Welcome to comp304!
  [comp304] ali: we must unite, this is unfair
  [comp304] ahmet: lets ask for an extension
  [comp304] osman: yes!
  [comp304] mehmet > <write your message here>
```

And here is an example of what happens when the user sends a message:

```
$ chatroom comp304 mehmet
  Welcome to comp304!
  [comp304] ali: we must unite, this is unfair
  [comp304] ahmet: lets ask for an extension
  [comp304] osman: yes!
  [comp304] mehmet: I agree
  [comp304] mehmet > <write your message here>
```

(c) **wiseman <minutes>** (Must be Written in C): This command will utilise `espeak`, a text speech synthesizer, to say a random adage given by the `fortune` program, every `x` many minutes as specified by the user. Here is how *wiseman* command should look:

```
$ wiseman 5 # spits out a wisdom every 5 minutes
```

You will need to install both the `espeak` and `fortune` programs on your system.

```
$ sudo apt install fortune espeak # for those with Ubuntu
$ sudo dnf install fortune espeak # for those with Fedora
$ # <for those who are using something else, google is your friend>
```

You may want to use `crontab` in this part of the project. We strongly suggest you to first explore crontab before starting implementation of this command. More info about crontab can be found here:

http://www.computerhope.com/unix/ucrontab.htm

(d) **Custom Command** (Can be written in any language): The last command is any new **Shellax** command of your choice. Come up with a new command that is not too trivial and implement it inside of **Shellax**. 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.
Note: If you have a partner, you and your partner are required to implement separate custom commands and the project report has to include explanation of both commands.

Part V

(15 points)
Psvis <PID> <outputfile> (Must be written in C): You are required to implement the `psvis` command which uses a kernel module. `psvis` command finds the subprocess tree by treating the given PID as the root and visualizes it in a human-friendly graph form. A node in the graph represents a process and an edge represents the parent-child relationship between two processes. In each node, show the PID and creation time of the process. The heir nodes (eldest child of a parent) should be colored with a district color. For visualization, the graph should be dumped into an image file.

- To develop this command, an underlying kernel module is required to handle the process tree. The visualization part does not need to be handled inside the kernel module. You need to be a superuser in order to complete this part of your assignment. The command `psvis` should trigger the kernel module.
- 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 **Shellax** and make sure it works.
- When the command is called for the first time, **Shellax** will prompt sudo password to load the module into the kernel. Successive calls to the command will notify the user that the module is already loaded.
- **Shellax** should remove the module from kernel when the shell is exited.
- You can use `pstree` command to check if the process list is correct. Use -p to list the processes with their PIDs.
- You may want to gnuplot or a similar tool to draw the process graph and save as an image file.

References

We strongly recommend you to start your implementation as early as possible as it may require a decent amount of researching. The following 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
- To develop this command, an underlying kernel module is required to handle the process tree. The visualization part does not need to be handled inside the kernel module. You need to be a superuser in order to complete this part of your assignment. The command `psvis` should trigger the kernel module.
- 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 **Shellax** and make sure it works.
- When the command is called for the first time, **Shellax** will prompt sudo password to load the module into the kernel. Successive calls to the command will notify the user that the module is already loaded.
- **Shellax** should remove the module from kernel when the shell is exited.

- You can use `pstree` command to check if the process list is correct. Use -p to list the processes with their PIDs.
- You may want to gnuplot or a similar tool to draw the process graph and save as an image file.

Deliverables

You are required to submit the followings packed in a zip file (username1-username2.zip) to blackboard:

- Each team must create a Github repository for the project and add the TA as a contributor (username is mktip). Add a reference to this repo in your REPORT. We will be checking the commits during the course of the project and during the project evaluation. This is useful for you too in case if you have any issues with your OS.
- .c source file that implements the **Shellax** shell. Please add comments to your implementation.
- .c source file of the Kernel module used by `psvis`.
- any supplementary files for your implementations (e.g., Makefile).
- (5 points) a Report describing your implementation, particularly the new builtins in Part III. You may include screenshots in your report.
- You should keep your Github repo updated from the start to the end of the project. You should not commit the project at once when you are done with it; instead make consistent commits so we can track your progress. Otherwise a penalty may apply.
- Do not submit any executable files (a.out) or object files (.o) to blackboard.
- You are required to implement the project on Linux.
- Selected submissions may be invited for a demo session. Note that team members will perform separate demos. Thus each project member is expected to be fully knowledgeable of the entire implementation.. Not showing up at the demo will result in zero credits.

GOOD LUCK.