COMP 304 Shellgibi: Project 1

Due: Tuesday March 10th, 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 **10%** of your total grade. We recommend you to START EARLY.

Responsible TA: Mandana Bagheri (mmarzijarani20@ku.edu.tr), Office: ENG 110

Description

The main part of the project requires you to develop an interactive Unix-style operating system shell, called **ShellGibi** in C. After executing **ShellGibi**, it will read commands from the user and execute them. Some of these commands will be *builtin* commands, i.e., specific to **ShellGibi** 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 three main parts. We suggest starting with the first part and continue with other parts.

Part I

(20 points) **ShellGibi** 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 **ShellGibi**. 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 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 **ShellGibi**. 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:

```
shellgibi$ 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 **ShellGibi**. 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:

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

Part III

(10+15+10+10+10 points) In this part of the project, you will implement new **ShellGibi** commands (builtin commands).

(a) **Auto-complete**: you are required to handle auto-completion of existing and newly introduced commands in your shell. While typing a command if the Tab key is pressed, **ShellGibi** should automatically complete the command. If there are more than one match found, it should list all the possible matches. If the command is fully typed, then the Tab key is pressed, it should list the files in the current directory. Your implementation should support newly introduced commands in this project.

(b) **Job management**: you are required to implement a set of simple job management builtins such as `myfg`, `mybg`, `myjobs` and `pause` without using a kernel module. These commands should only consider user processes owned by the current user (not system or other user processes).

- **myjobs**: list the processes owned by the current user as a table on the command line. For each job, pid, program name and status should be displayed.
- **pause** <PID>: Given a job's pid, pauses the execution of this job.
- **mybg** <PID>: Given a job's pid, brings the paused job to background and running state.
- **myfg** <PID>: Given a job's pid, brings the paused job to foreground and running state.

You can test whether these commands are working or not by running two instances of **ShellGibi**. For example, create processes on one shell, and test your commands on the second one.

(c) **Psvis <PID> <outputfile>**: The `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 **ShellGibi** and make sure it works.
- When the command is called for the first time, **ShellGibi** 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.
- **ShellGibi** 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.

(d) **Alarm <time> <musicfile>**: This command will play a sound file every day at a time specified by the user. Here is how *alarm* command should look like. You can pick any music file of your choice. Here is how *alarm* command should look:

```
1    shellgibi$ alarm 7.15 wakeupwakeup.wav
```

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

(e) **Custom Command** The last command is any new **ShellGibi** command of your choice. Come up with a new command that is not too trivial and implement it inside of **ShellGibi**. 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.

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

Deliverables

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

- .c source file that implements the **ShellGibi** 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).
- (10 pts) a Report describing your implementation, particularly the new builtins in Part III. You may include screenshots 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 Linux.
- There will be project DEMOs after the deadline. We expect both team members to equally contribute to the project and are competent to answer questions on any parts of the implementation. Not showing up at the demo will result in zero credits.

GOOD LUCK.