COMP 304 - Operating Systems: Project 1

Due: 23:59 April 14, 2023

Didem Unat Spring 2023

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 discussion forum, but the submitted work must be your own. **Any material you use from external sources such as the internet 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.

Contact TA: Javid Baydamirli (jbaydamirli21@ku.edu.tr)
Office Hours: 17:30-19:00 Tuesday and Wednesday in ENG 230, or email beforehand.

GitHub submissions: We will be using `GitHub classroom` for this assignment. Find your KU username in the student list and clone the repository with starter code. You must push your work to your assigned GitHub repository. We will be checking the commits throughout the course of the project and during evaluation. This is useful for you too in case if you have any issues with your OS, as your work will be backed up online. Note that you still need to upload a zip file to Blackboard.
GitHub Classroom Link: https://classroom.github.com/a/s8hwXLGK

Description

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

Part I (15 points)

- Use the skeleton program provided as a starting point for your implementation. The skeleton program reads a line of commands from `stdin`, parses it, and separates it into arguments using whitespace as the delimiter. You will implement the action that needs to be taken based on the command and its arguments entered in **Mishell**. Feel free to modify the command line prompt and parser as you wish.
- Use the provided `Makefile` to compile your code. Type `make help` to get a list of build targets.

- Command line inputs, except those matching builtin commands, should be interpreted as program invocation. The shell must **fork** and **execute** the requested programs. Refer to *Part I - Creating a child process* of the book.
- 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 the program.
- Do not use the `exec()` family of calls that are prefixed with `p` such as `execp()` that automatically search for executable files. Instead, use the `execv()` library call and implement path resolution yourself.
- Implement `cd` and `exit` builtin commands.

Part II (10 points)

- In this part of the project, you will implement I/O redirection for **Mishell**. Implement the following operators:
  - **Operator** > : The file is created if it does not exist, and truncated otherwise.
  - **Operator** >>: Same as above, but the output is appended if the file already exists.
  - **Operator** < : The input of the program on the left-hand side is read from a file.

  A sample command line is given for I/O redirection below.

```
$ program arg1 arg2 > output.txt >> append.txt < input.txt
```

  You can use `dup()` and `dup2()` system calls for this part. Refer to the bash manual on redirections for more info.

Part III (10 + 15 + 15 + 10 points)

In this part of the project, you will implement new **Mishell** commands (builtin commands).

1. Dice Roll (10 points)

Write a program to roll multi-sided dices and display the results individually with the sum at the end.
Syntax:

```
$ roll [number of rolls]d<number of sides>
```

Example output:

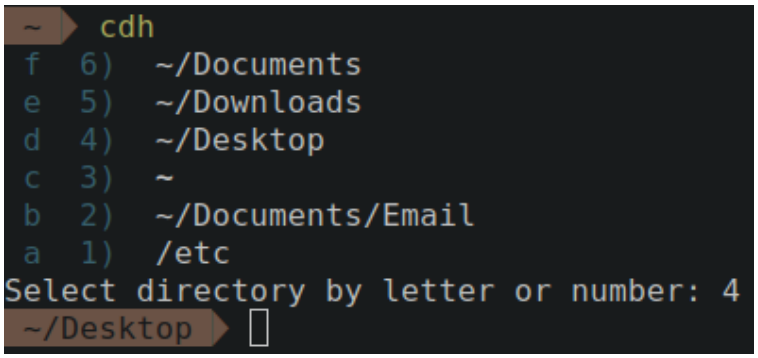
```
$ roll 3d6
Rolled 17 (3 + 5 + 4)
$ roll 4d10
Rolled 27 (3 + 10 + 9 + 5)
```

```
$ roll d3
Rolled 2
```

- Notice that the number of rolls is optional. If not provided, assume a single roll.
- Make sure to handle malformed input and other possible errors.

2. cd history (15 points)

Implement a command called **cdh**. The command takes no arguments. When called, the command should output a list of **10** most recently visited directories. The list should also include an index for each directory as both a number and an alphabetic character. The command should then prompt the user the directory they want to navigate to. The user can select either a letter or a number from the list. After this, the shell should change into the selected directory. Make sure not to include the same directory twice. Below is example output for 5 most recent directories.



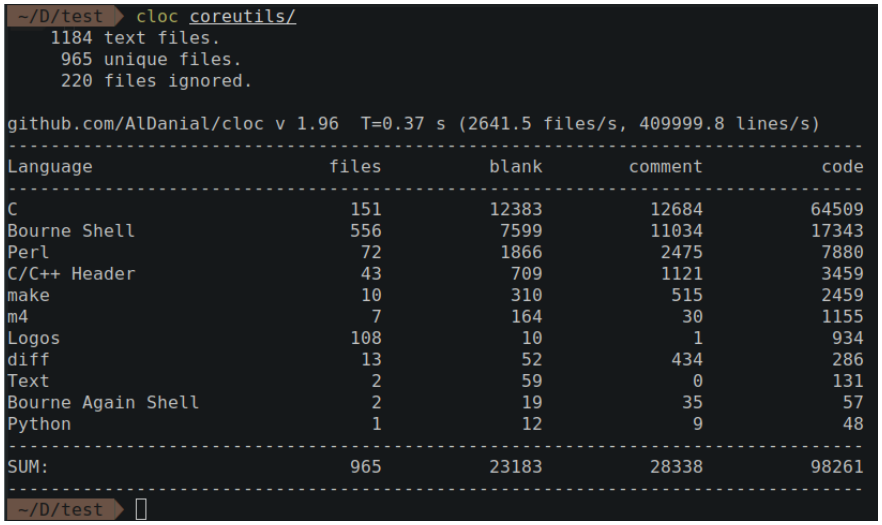
Keep in mind that **this command lives across shell sessions**; when a new shell session is started, it should remember recently visited directories of previous sessions.

Note: the command is inspired by the cdh command from the fish shell. You can take a look at its documentation for more details.

3. Count Lines of Code (15 points)

Implement a command called **cloc** to recursively count lines of code for all source code files in a given directory.
The output below is from the `cloc` utility that has support for many languages. Your version does not need to be similarly comprehensive, however, and you are only expected to implement it for `C`, `C++`, `Python`, and can assume every other file type to be Text (`.txt`). You also do not need to calculate performance metrics such as lines per second as shown in the screenshot.

- Print the total number of files found



- Print the number of files that are processed, ignoring binary files and dotfiles (files and directories that start with a `.`)
- List each file type or language with the number of files, total number of blank lines, comment lines, and code lines. You can alternatively list by file extension as opposed to language names shown in the screenshot, for example, `.c`, `.py`, `.txt`, and so on.
- Print a collective sum of all blank, commend and code lines in all files as shown in the screenshots.

4. Custom Command (10 points) [Can be written in any language]

The final command is any new **Mishell** command of your choice. **Come up with a new command that is not too trivial** and implement it inside your shell as a builtin. Be creative. Selected commands will be shared with your peers in the class. You are allowed to take inspiration from existing Linux programs or existing shell builtins, but do not make an exact clone.

Note: If you are implementing this project as a team, both partners are required to implement their own custom commands, and the report must include explanations of both commands.

Part V (20 points)

Psvis <PID> <output file> (Must be written in C):
You are required to implement the **psvis** command which uses a kernel module. The command finds the subprocess tree by treating the give 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 (the 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 defined in `linux/sched.h` to obtain necessary information such as process name and process start time.
- Test your kernel module first outside of **Mishell** and make sure it works.
- When the command is called for the first time, **Mishell** will prompt sudo password to load the module into the kernel. Successive calls the command will notify the user that the module is already loaded.
- **Mishell** 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 the **-p** flag to list the processes with their PIDs.
- You may want to use **gnuplot** or a similar tool to draw the process graph and save it as an image file.

Note: You need to be in a working directory with no spaces in the path to build the kernel module with the provided Makefile.

References

We strongly recommend you to start your implementation as early as possible as it may require a decent amount of research. The following links might be useful:

- 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 in implementing Part V.
- Writing a simple kernel module:
  https://devarea.com/linux-kernel-development-and-writing-a-simple-kernel-module/
- Task Linked List (scroll down to Process Family Tree):
  https://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 and Requirements

You are required to submit the following in a zip file (name it username1-username2.zip) to Blackboard.

- You must push your work to GitHub classroom in addition to Blackboard submissions. We will be checking the commits throughout the and during project evaluation.
- Although not *required*, we highly encourage you to use the provided `.clang-format` file to autoformat your code. Run the following command to apply formatting.

```
find . -name '*.[ch]' -exec clang-format -i {} \;
```

- `.c` source file that implements the **Mishell** shell. Please add comments to your implementation.
- `.c` source file of the Kernel module used by `psvis`.
- Any supplementary files for your implementation (Makefiles, header files, etc.)
- (5 points) a report describing your implementation, particularly the new builtins in Part III. You may include screenshots in your report. Please submit a `pdf` file.
- You should keep your GitHub repo updated from the start to the end of the project. Do not commit at the very end when you are finished, instead make consistent commits as you make progress. You will be penalized otherwise.
- Implement and test your code in a Linux environment.
- Selected submissions may be invited to a demo session. Note that team members will perform separate demos. As such, each team member is expected to be fully knowledgeable of the entire implementation. Not showing up at the demo will result in zero points.