COMP 304 Shellfyre: Project 1

Due: 31 March 2022 midnight

Didem Unat Spring 2022

Notes: The project can be done **individually** or **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. This assignment is worth 15% of your total grade. Read this document carefully before you start and START EARLY.

Any material you use from web should be properly cited in your report. Any sort of cheating will be harshly PUNISHED.

Contact TA: Ismayil Ismayilov (ENG 230)
iismayilov21@ku.edu.tr

Description

The main part of the project requires development of an interactive Unix-style operating system shell, called **shellfyre** in C/C++. After executing **shellfyre**, **shellfyre** will read both system and user-defined commands from the user. The project has three main parts:

Part I - Basic Commands

20 points

The shell must implement basic commands and support the following:

- 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 **shellfyre**. Feel free to modify the command-line parser as you wish.
- Command-line 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.
- 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 UNIX commands (e.g. ls, mkdir, cp, mv, date, gcc etc) and user programs by the child process.

The descriptions in the book might be useful. You can read Project 1 - Unix Shell Part-I in Chapter 3 starting from Page 157 (9th edition).

Part II - Custom Commands

In this part, you are asked to implement a number of user-defined commands in **shellfyre**.

filesearch (15 pts)

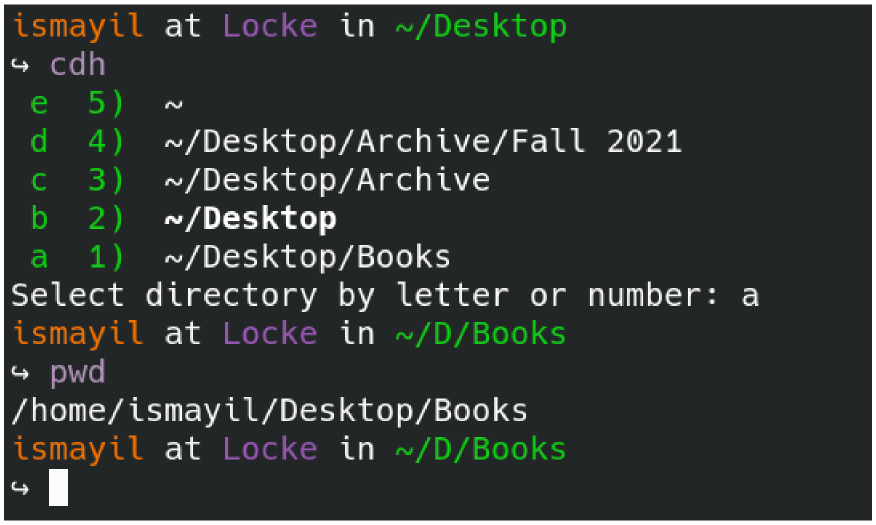
The first command you will implement is called **filesearch**. This command is useful to search filenames matching a keyword in a directory. The command takes an input keyword and scans all the files in the current directory to match the keyword with the file name and then it returns the list of file(s) with this name regardless of the file extensions.

In addition, you are required to support recursive and open options. In recursive **filesearch**, 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. If the **-o** option is passed to **filesearch**, it should open file(s).

```
shellfyre> filesearch "foo"
    ./foo.txt
    ./food.pdf
shellfyre> filesearch -r "foo"
    ./foo.txt
    ./food.pdf
    ./dir1/abcfoo
    ./dir1/foot.out
```

cdh (15 pts)

The next command you should implement is called **cdh**. The command takes no arguments. After calling **cdh**, the shell should output a list of most recently visited directories. The list should also include the index of the directory as both a number and an alphabetic letter. The shell would then prompt the user which directory they want to navigate to; the user can select either a letter or a number from the list. After this, the shell should switch to that directory. If there are no previous directories to select from, the shell should output a warning. You should keep track of 10 of the most recently used directories. There is no need to handle duplicates (it is allowed for directories to be duplicated in the output). Example output is shown in the screenshot below for 5 directories:



```
ismayil at Locke in ~/Desktop
↪ cdh
 e  5)  ~
 d  4)  ~/Desktop/Archive/Fall 2021
 c  3)  ~/Desktop/Archive
 b  2)  ~/Desktop
 a  1)  ~/Desktop/Books
Select directory by letter or number: a
ismayil at Locke in ~/D/Books
↪ pwd
/home/ismayil/Desktop/Books
ismayil at Locke in ~/D/Books
↪ 
```

Note that **this command lives across shell sessions**; when a new shell session is started, it should remember the most recently visited directories.
Note: the idea for the command adapted from the cdh command from the *fish* shell. You can take a look at its description if you want more details.

take (10 pts)

In this part, you will implement a command called **take** which takes 1 argument: the name of the directory you want to create and change into. The command will create a directory and change into it. The command must create the intermediate directories along the way if they do not exist. For example: if you call **take A/B/C**, the command should create the directories that do not exist and change into the last one (i.e, A/B/C).
Note: the idea for the command was adapted from the **take** command from *zsh*.

joker (15 pts)

The last command is called **joker**. The command will generate a random joke every 15 minutes and output it to the screen.

To implement this command you will need to use the **crontab** command provided by Linux and the **curl** command (install it if you do not have it). To get a random joke you can use *https://icanhazdadjoke.com*.

We suggest you to explore *crontab* and *notify-send* utilities before implementing this command. You can learn more about *crontab* from `http://www.computerhope.com/unix/ucrontab.htm` and *notify-send* from `http://manpages.ubuntu.com/manpages/xenial/man1/notify-send.1.html`.

Your awesome command (10 pts)

This command is any new **shellfyre** command of your choice. Come up with a new command that is not too trivial and implement it inside of **shellfyre**. 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. Also note that **if you are in a team of 2, each team member should implement one command each**.

Part III - Kernel Modules

20 points

In this part of the project, you will write a kernel module that will be triggered using commands from **shellfyre**. You need to be a superuser or have a sudo access in order to complete this part of the project. The shell command that invokes the kernel module is as follows:

- **pstraverse <PID> <-d or -b>**: The `pstraverse` command finds the subprocess tree by treating the given PID as the root and traverse the tree in depth-first-search (DFS) or breadth-first-search (BFS) order depending on the second command line argument, i.e. -d or b. If the value is -d, then the tree is traversed in DFS order. Otherwise, if the value is -b, the tree is traversed in BFS order.

  When a `task_struct` node in the subprocess tree is visited during the traversal, the PID and the name of the executable run by the process are printed using printk by the kernel module.

  The followings are some suggestions and details for your implementation.

  - You will need to explore the Linux task struct in `linux/sched.h` to obtain necessary information such as process id (PID) and name.
  - Test your kernel module first outside of **shellfyre** and make sure it works.
  - When the command is called for the first time, **shellfyre** will prompt sudo password to load the module into the kernel. After the module is loaded by the first call, the tree traversal operation on the targeted PID will run. Successive calls to the command will not load the module again. They will only invoke tree traversal operations by on targeted PIDs. In the first call of the command, the traversal operation can be run by the initialization function of the kernel module. In the following calls, you can use ioctl function calls to trigger the operations.
  - **shellfyre** 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. Note that there might be some processes that are shown by pstree but not shown by your kernel module. These processes are the ones whose names are in curly brackets when printed by `pstree -p`.

Useful References:

- Info about task linked list - http://www.informit.com/articles/article.aspx?p=368650
- Linux Cross Reference - https://elixir.bootlin.com/linux/latest/source/include/linux
- You can use **pstree** to check if the sibling list is correct. Use -p to list the processes with their PIDs
- 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

READ CAREFULLY

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

- .c source code file that implements the **shellfyre** shell. Please comment your implementation.
- .c sourse code file that implements the kernel module you developed 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. You may include your snapshots in your report.
- Both in the report and source code, mention the partners' names and their KUSIS IDs.
- Do not submit any executable files (a.out) or object files (.o) to Blackboard.
- Each team must create a **github repository for the project** and add the TA as a contributor (username is readleyj). 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.
- Insufficient amount of detail in the REPORT or in the code repository (e.g. one commit only on the day of the deadline) will result in a 10 point penalty in the project grade.
- 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.
- 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.

GOOD LUCK and START EARLY