COMP 304 Shell-ish: Assignment 1

Due: Feb 25th, 2026, 11.59 pm

Notes: The assignment must be done **individually**. You may discuss the problems with other students but the submitted work must be your own work. **Any material you use from web (or AI model) should be properly cited in your submission. Any sort of cheating will be harshly PUNISHED.**

This assignment is worth 6% of your total grade (3% for the implementation and invited demos, and 3% for the project exam). The project exam will be conducted during the midterm exam. To be eligible to take the project exam, you must complete and submit the assignment before the deadline. We strongly recommend that you start early.

Responsible TAs: Mohammad Issa (missa18@ku.edu.tr), Office: ENG 230, 4:00-5:30 PM Tuesdays (or by appointment via email)

Description

The main part of the assignment requires you to develop an interactive Unix-style operating system shell, called **Shell-ish** in C. After executing **Shell-ish**, it will read commands from the user and execute them. Some of these commands will be *builtin* commands, i.e., specific to **Shell-ish** and not available in as an executable, while others will be normal commands, i.e., they will run programs available in the system (e.g.,`ls`). We recommend starting with the first part and proceeding with other parts afterwards.

The assignment has 3 main parts (100 points). Please note that some students will be randomly selected for in-person demos. Points may be deducted in cases of poor performance during the demo.

To build and run your shell:

```
$ gcc -o shell-ish shell-ish-skeleton.c
$ ./shell-ish # or
$ rlwrap ./shell-ish # better for line editing
```

Part I

(10 points) **Shell-ish** 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 **Shell-ish**. 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

(10+10 points)

- In this part of the assignment, you will implement I/O redirection for **Shell-ish**. 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.

  Also note the lack of a space before the redirection targets, the parser only supports those. So do not test with spaces between your redirection symbols and redirection target.

- In this part, you will handle program piping for **Shell-ish**. 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

(25+25+20 points) In this part of the assignment, you will implement new **Shell-ish** commands (builtin commands).

(a) **cut** (Must be written in C): you are required to implement an application similar to UNIX's `cut` command. The program reads lines from standard input and prints only the specified fields. By default, input lines are assumed to be separated by a TAB character. The program must support the option `-d, --delimiter`, which, when followed by a single character, specifies the delimiter to be used instead of TAB. It must also support the option `-f, --fields`, which accepts a comma-separated list of field indices (e.g., `1,3,10`); only these fields should be printed for each input line, in the order specified.

```
$ cat /etc/passwd
root:x:0:0:Super User:/root:/bin/bash
bin:x:1:1:bin:/bin:/usr/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/usr/sbin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/usr/sbin/nologin
missa18:x:1001:1001::/home/missa18:/bin/bash
...

$ cat /etc/passwd | cut -d ":" -f1,6
root:/root
bin:/bin
mail:/var/spool/mail
nobody:/
missa18:/home/missa18
...
```

(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 alara osman ahmet
```

Here is an example of a user joining a room:

```
$ chatroom comp304 mehmet
  Welcome to comp304!
  [comp304] ali: 2 weeks for completing the first assignment is criminal
  [comp304] alara: 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: 2 weeks for completing the first assignment is criminal
  [comp304] alara: lets ask for an extension
  [comp304] osman: yes!
  [comp304] mehmet: I agree
  [comp304] mehmet > <write your message here>
```

(c) **Custom Command** (Can be written in any language): The last command is any new **Shell-ish** command of your choice. Come up with a new command that is not too trivial and implement it inside of **Shell-ish**. 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. Create a README.md explaining your command and some sample usage.

Deliverables

You are required to submit the followings packed in a zip file (username.zip) to LearnHub:

- You must create a Github repository for the assignment and add the TA as a contributor (username is mktip). Add a reference to this repo in the README in your submission. We will be checking the commits during the course of the assignment and during the assignment evaluation. This is useful for you too in case if you have any issues with your OS.
- .c source file that implements the **Shell-ish** shell. Please add comments to your implementation.
- an imgs folder containing Screenshots of each of the features you added (including commands implemented)
- any supplementary files for your implementations (e.g., Makefile).
- A README describing how to use your shell, and your custom command. Make sure to include your repo-link here as well.
- You should keep your Github repo updated from the start to the end of the assignment.
- You should not commit the assignment 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 LearnHub in your zip.
- You are required to implement the assignment on Linux.
- Selected submissions may be invited for a demo session. You are expected to be fully knowledgeable of the entire implementation. Not showing up at the demo will result in zero credits.

GOOD LUCK.