CSci 4061: Introduction to Operating Systems

Programming project 2 **Due: Thursday, March 5th, 2020**

Ground Rules. You may choose to complete this project in a group of up to three students. Each group should turn in **one** copy with the names of all group members on it. The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc. All submissions must compile and run on any CSE Labs machine located in KH 4-250. A zip file should be submitted through Canvas by 11:59pm on Thursday, March 5th. **Note**: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo.

Objectives: This project aims to create your `command line interpreter`, also known as a shell program. The shell program will prompt the user to type in a command. The shell creates a child process to execute the command and prompts the user for more once it has finished. The shell program will be simpler than the ones offered in Unix systems. You have to implement your versions of `cd`, `ls`, and `wc`. This gives you more flexibility to add functionality to these programs and not restrict yourself to the version in `'/bin/'`. This exercise is aimed to help you familiarize yourself with the Linux programming environment.

The main focus of this project is to implement (1) three Unix file system–related commands (`cd`, `ls`, and `wc`) and (2) a unix shell which is capable of executing any Unix commands, including the commands which you implemented. Moreover, the shell should be able to handle redirection and pipes in addition to executing the commands.

1 Part 1: Implementation of Unix Commands

You have to implement the cd, ls and wc commands in their C files which are already provided in the codebase. Those C files have its own *main* function. After you finish the implementation of the commands in their C files, the next step is to generate the executable files. To create their executable files, you can just execute `'make build'` which builds the executable files for all the commands; the Makefile has been provided to you. Once you have the executable files for the commands, you should be able to run the commands from the terminal. For example, if you want to run the "ls" command, you can just execute `'./ls'` like any other C program.
You have to implement all the below commands using basic system calls instead of just directly reusing (e.g., using $fork$ and *exec*) the existing commands provided by the operating system. In the following, we describe the details about the commands. Note that all paths used in the following can be both absolute or relative.

1.1 `cd`

The `cd` command is used to change the working directory to the given path. ".." indicates the parent directory. Refer to the `cd` man page for more details.

List of test cases:

```
./cd <path>
./cd ..
```

(Hint: Use the `chdir` system call)

1.2 `ls`

The `ls` command is used to list files and directories. When used with the argument `-R`, it recursively displays the content of every subdirectory in the given path. You have to implement a version of ls command which takes the path as input and displays the names of every file and subdirectory in it. Additionally, if `-R` is passed as an argument to your ls command, it should display the contents of every subdirectory as well.

List of test cases:

```
./ls <path>
./ls -R <path>
```

If the path is not given, use the current working directory as the path.
(Hint: Use directory-related system calls, like `scandir`, `readdir` and `stat`. Recursion might be useful.)

1.3 wc

The `wc` command is used to find the line,word and character count of a file. Your version of the `wc` should have a three columnar output. First column shows number of lines present in a file specified, second column shows number of words present in the file and the third column shows number of characters(including whitespaces) present in file. Additionally, `wc -l` should print the number of lines, `wc -w` should print the number of words and `wc -c` should print the number of characters.

List of test cases:

```
./wc filename
./wc -l filename
./wc -w filename
./wc -c filename
```

Note that all paths used in the above sections can be both absolute or relative

2 Part 2: Implementation of the Unix Shell

The second part of the project is to implement a unix shell that supports file redirection and pipes. The implementation should be done in the file `'shell.c'` in the given codebase. The shell should always display the current working directory along with a dollar while prompting the user to enter a command, as the original unix shell does. To differentiate from the original shell, add the tag

`[4061-shell]` at the start. For example, if `'home/csci4061/project'` is the current working directory, it should display:

```
[4061-shell]/home/csci4061/project $
```

You must use `STDIN` to read the user input using the read system calls. The shell should be able to parse any of the above commands and then execute the correct command along with user provided arguments. In the shell, you must use `fork` and `exec` to execute the command. If a specified command is not one of the above 3 commands implemented by you, you should call the system's original command executable directly using the `exec` function. If any error occurs during the execution of the given command, you have to output `'Command error'`.

Additionally, you have to implement `exit` command to terminate the shell. You can perform all the necessary steps to quit the shell(like freeing memory etc.) when handling the `exit` command. (Hint: You may find the `strtok()` function useful for parsing the command line arguments)

2.1 File redirection

The shell should be able to handle redirection of any command's output to a file. In other words, the shell should be able to handle `'>'` and `'>>'` operators. For example,

```
[4061-shell]/home/csci4061/project $ ls > out.txt
```

The above command will redirect the output of `ls` command to a file named out.txt. The `'>'` operator will truncate the old contents in the file if any. And the `'>>'` operator will append the new contents to the old contents of the file if any.
(Hint: Use dup2 system call)

2.2 Pipes

The shell should also be able to handle pipes between multiple commands. For simplicity, your shell should have the ability to handle one pipe between two commands. For example,

```
[4061-shell]/home/csci4061/project $ cat data.txt | grep student
```

The above command will search for the word `'student'` in the output of `'cat data.txt'`, which is the content of the file `'data.txt'`. Basically it will pipe the output of the cat command into the input of the grep command.
(Hint: Use pipe along with fork and exec system calls)
And also, the shell should support the combination of redirection and pipes in a single command. For example,

```
[4061-shell]/home/csci4061/project $ ls | wc > out.txt
```

If you are curious, try implementing the support for multiple pipes. (But no extra credit will be provided)

3 Possible division of labour

- First person: Implementing `cd` and `exit` + Command parsing and execution logic
- Second person: Implementing `ls` + file redirection
- Third person: Implementing `wc` + pipes

It is not necessary to follow the above division of labour. Feel free to change it as needed. *(Note: Start the project early)*

4 General notes

1. Use cselab machine (or similar environment) to implement the solution. Mac/Windows environment will not be suitable for this project as you have to read input from `STDIN`.
2. The list of test cases given is the only test cases upon which your command will be tested. If your command works correctly for those basic test cases, you will get full credit for those commands.
3. You are given some of the helper functions in `'util.c'` which might be helpful. Feel free to change the implementation in `'util.c'`
4. Handle errors from system call by checking the return values of the call. We recommend you start doing this from the very beginning of your work as you'll often catch errors.

5 Deliverables

Students should upload to Canvas a zip file containing their C code, a makefile, and a README that includes the group member names, what each member contributed, any known bugs, test cases used (we will also use our own)and any special instructions for running the code.

6 Grading Rubric

- 5% For correct README contents
- 5% Code quality such as using descriptive variable names and comments
- 40% Implementation of all the commands (cd and exit - 5% each, ls and wc - 15% each)
- 25% Implementation of basic unix shell
- 10% Implementation of file redirection in shell
- 10% Implementation of pipes in shell
- 5% Ability to handle both redirection and pipe in a single command