Project 1

(Pseudo-Shell: a single-threaded UNIX system command-line interface)

CIS 415 - Operating systems

Fall 2025 - Prof. Jieyang Chen

Due date: **11:59 pm, Friday, October 24, 2025**

Introduction:

If we were to ask an average person what a shell is, one of the first things that will come to their mind is the outside of an egg, which I highly recommend you remove before eating. For us computer scientists, however, we think of a shell as a software program, or interface, that interprets commands from the user so that the operating system can understand them and perform the appropriate functions.

In Unix systems, the shell is a command-line interface, which means it is solely text-based. The user can type commands to perform functions such as run programs, open and browse directories, and view processes that are currently running. The catch is, you need to know the correct syntax when typing the commands or the shell will angrily spit out errors at you.

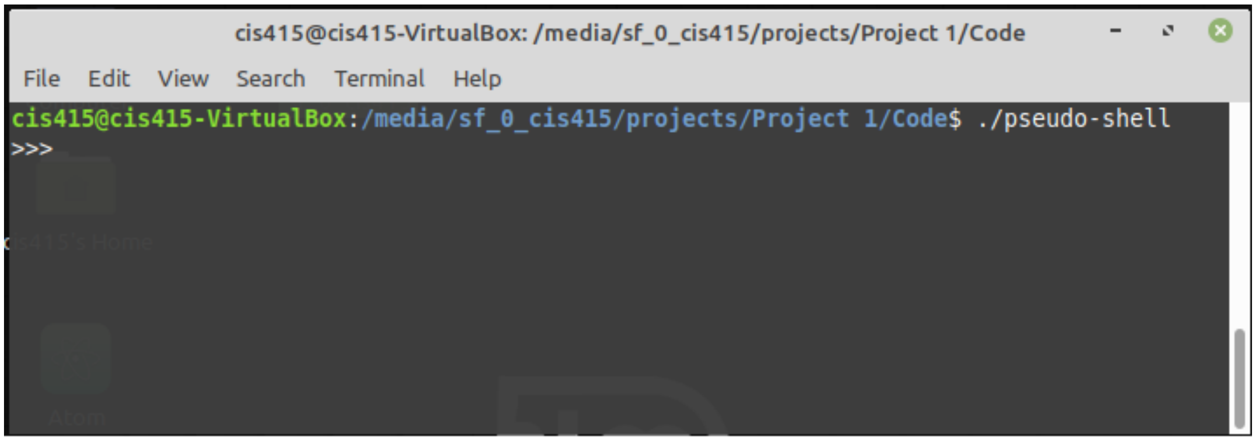
For the first project in this class, you will be building a simpler version of the shell from scratch. We call this shell the "Pseudo-Shell". The pseudo-shell is a single-threaded, synchronous, command-line interface that allows the user to perform a limited set of functions centered around file system navigation, manipulation, and I/O.

Project Details:

In this project, you will implement a pseudo-shell that is capable of executing a number of UNIX system commands. The following are the functionalities of this pseudo-shell:

Shell modes:

The shell can be started in one of two modes: A) File mode, B) Interactive mode. To run the shell interactively, simply type: **./pseudo-shell**. Upon executing, we should see a shell-like CLI that is ready to receive commands input into the console as shown in Fig. 1 below. This CLI sits at the ready state, indicated by the prompt **">>>"**, until the user inputs a command string and presses enter. In this shell mode, the user interacts with the shell by typing in commands one line at a time and pressing enter. The shell then executes those commands and displays any output if necessary (some commands execute with no output such as **rm** or **mv**).


cis415@cis415-VirtualBox: /media/sf_0_cis415/projects/Project 1/Code
File Edit View Search Terminal Help

```
cis415@cis415-VirtualBox:/media/sf_0_cis415/projects/Project 1/Code$ ./pseudo-shell
>>>
```

Fig. 1 pseudo-shell in interactive mode

The program is also capable of reading commands from a batch file whose name is specified in the command line via file mode. To run the shell in file mode, type: **./pseudo-shell -f <filename>,** where **<filename>** is the name of the batch file we wish to read commands from. The input file will contain lines of text, where each line represents either a single command to be executed or a sequence of commands to be executed. When we use this shell in file mode, it will generate a file "**output.txt**". The output file "output.txt" will have all the same STDOUT as running in interactive mode for the given commands. Your shell should create this file in the same directory that your shell is in.

Base shell functionalities:

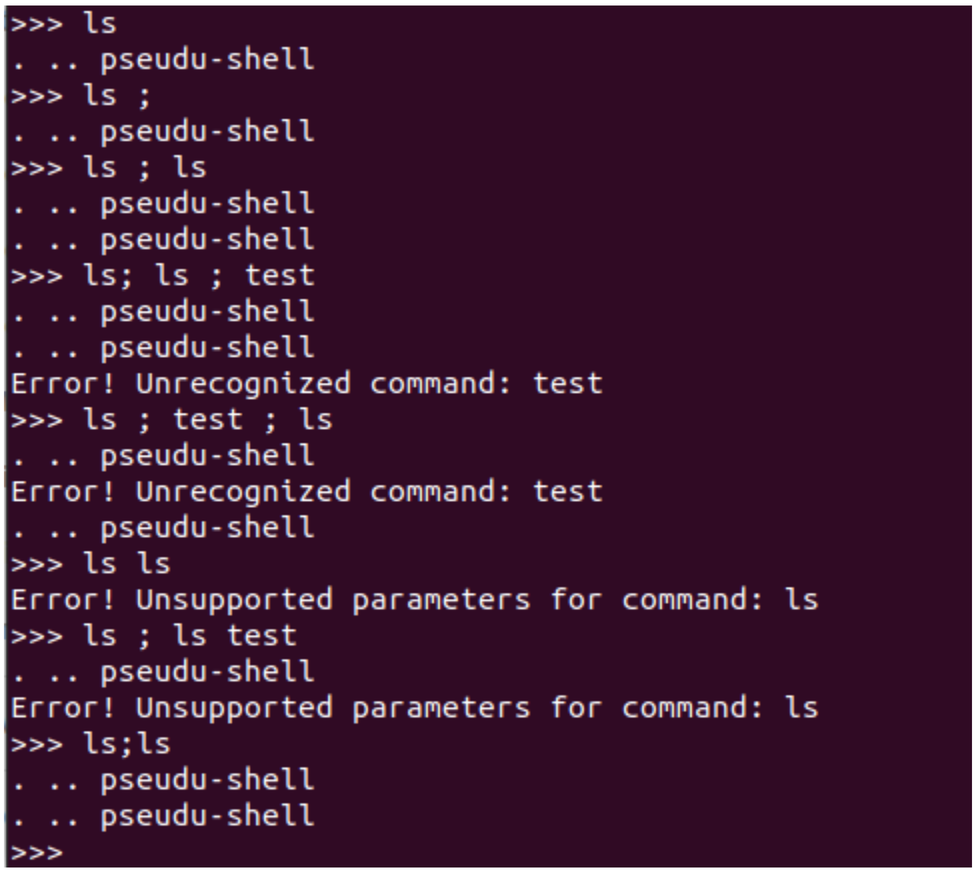
Input commands have the following format: **<cmd> <param1> <param2> ; <cmd2> …** Where **<cmd>** is a command from the list of acceptable commands shown in table 1 below. And **<param>** are its input parameters. **Note: some commands have no input parameters.** These keywords are each delimited by a space with individual command calls delimited by the control code **";"** as shown in Fig. 2. You **must** use **getline(3)** to grab input from either the file or console. When the control code is given, each command is executed sequentially one after another until the end of the line is reached at which time the shell will either go back to waiting for input from the user (in interactive mode) or will read the next line in file mode. Of course, multiple commands can be executed one after another when delimited by the control code **";"**, in this case the commands are executed and the output is displayed sequentially. There is one special command: **"exit"**. If the user enters this command then the pseudo-shell closes down. **Use this command (exit) to break out of your main loop.**

Error Handling:

Take care to handle the following error cases in your code:

1. Incorrect parameters:
    a. Missing parameters (i.e. **mv ../test.txt ,** mv must have 2 parameters)
    b. Too many parameters (e.g. **pwd ../test.txt**, pwd has 0 parameters)
2. Incorrect syntax:
    a. Missing/unrecognized command (e.g. **grep ./test** or **; ls**).
    b. Repeated command without control code (e.g. **ls ls)**

These errors can occur at any point in the line. If an error is encountered, you can skip processing the rest of the line. See the following image for the exact output the shell should display when given the corresponding input. (we are using ls to demonstrate the error handling but it should be the same for all commands). Your shell must follow this format and behavior **exactly.**



```
>>> ls
. .. pseudu-shell
>>> ls ;
. .. pseudu-shell
>>> ls ; ls
. .. pseudu-shell
. .. pseudu-shell
>>> ls; ls ; test
. .. pseudu-shell
. .. pseudu-shell
Error! Unrecognized command: test
>>> ls ; test ; ls
. .. pseudu-shell
Error! Unrecognized command: test
. .. pseudu-shell
>>> ls ls
Error! Unsupported parameters for command: ls
>>> ls ; ls test
. .. pseudu-shell
Error! Unsupported parameters for command: ls
>>> ls;ls
. .. pseudu-shell
. .. pseudu-shell
>>>
```

Your program should work regardless of the extra space in the picture. For instance
$ls;ls
Should behave the same as
$ls ; ls

Commands:

Implement the following UNIX-like commands, details shown in the table below, using system calls. ***You **must** implement these functions using Linux system calls.** You are not allowed to use the following functions (printf, fprintf, fopen, freopen). This rule only applies to the **command.c** file where you implement these commands. However, in the command.c you are allowed to use string manipulation from string.h. The following is the man page listing for system calls (if it is in here, then you can use it):
http://man7.org/linux/man-pages/man2/syscalls.2.html
Note: Pay attention to the kernel version in the list of system calls. (string.h is permitted)

Command	#Args	Example	Description
ls	0	ls	Unix ls command. Will show the names of files and folders of the current directory.
pwd	0	pwd	Unix pwd command. Will show the current directory.
mkdir	1	mkdir < name>	Unix mkdir command. Will create a directory with the filename specified by: <name>.
cd	1	cd directory	Unix cd command. Will change directory based on the destination specified in the argument.
cp	2	cp <src> <dst>	Only for copying files. Copy one file from the path specified by <src> to the location specified by <dst>. <src> can be any form below: “File”, “dir/file” <dst>can also be any form below: “file”,”dir/file”, “dir”
mv	2	mv <src> <dst>	Only for moving files(cut-paste). Move a file from the location specified by <src> to the location specified by <dst>. <src> can be any form below: “file”, “dir/file” <dst>can also be any form below: “file”,”dir/file”, “dir”
rm	1	rm <filename>	Only for removing files (not a directory). Will remove the specified file from any directory.
cat	1	cat <filename>	Displays a file’s content on the console.

Project Structure Requirements:

For a project to be accepted, the project must contain the following 5 files and meet the following requirements:

main.c : This file contains the main function. The main function **must** meet the following requirements:

1. **main.c** must be able to start in both of the following modes:
   a. File mode:
      i. File mode is called by using the -f flag and passing it a filename (e.g. pseudo-shell -f <filename>)
      ii. The shell only accepts filenames in file mode.
      iii. All input is taken from the file whose name is specified by **<filename>** and all output is written to the file **"output.txt".** There should be no console output in this mode. Place all output in the file.
   b. Interactive mode:
      i. If no command line argument is given the project will start in interactive mode.
      ii. The program accepts input from the console and writes output from the console.
2. The main function must have the following input parameters: **argc** and **argv**
   a. **argc** contains the number of input parameters for the main function.
   b. **argv** contains a 2D array of strings that correspond to the parameters input into the program on execution. (e.g. the filename)
3. You must use dynamic memory allocation via **malloc()** and **free()**.
   a. Hint: it might be helpful to do these in a function.
   b. Valgrind must not show any memory leaks for your program.
4. Your solution must use a loop that reads commands on a line-by-line basis using getline() from the command line or a file.
5. Your solution must be able to parse the input string command into tokens using the provided header file to identify the commands and their arguments.
   a. The string_parser.h will be provided.
6. Your solution must be able to call the corresponding function for Unix command execution.
   a. We will provide a header file (see below).

NOTE: You **must not** alter the header files!

string_parser.h: A header file that contains user defined command_line struct and functions headers related to them.
string_parser.c: This file contains all the functions for the string_parser header file.

command.h: A header file that contains all the function names that execute the Unix commands. We will provide this file with all the function names. Make sure to read it. You cannot edit the header in any way. We will be using the original command.h in grading your code, thus if you edit command.h your code will not work.

command.c: This file contains the function definitions for our shell commands. Your program **must** meet the following requirements:

1. The command function names must match the function names provided in command.h.
2. You must use the specified system calls. (See the table in Project Details)

Makefile: Your project must include a standard makefile.

README: Complete README.txt

Submission Requirements:

Once your project is done, do the following:

1. Use make to compile your code.
2. Use Valgrind to check for memory leaks.
3. Tar or zip the project folder and submit it to Canvas.
    a. The zip/tar file must contain the following content
        i. A screenshot of you successfully compiling your project in Ubuntu.
        ii. main.c
        iii. makefile
        iv. command.c & command.h
        v. README.txt
        vi. Any additional header file or c file

Valgrind can help you spot memory leaks in your code. As a general rule any time you allocate memory you must free it. Points will be deducted in both the labs and the project for memory leaks so it is important that you learn how to use and read Valgrind’s output. See (https://valgrind.org/) for more details.

Grading Rubric:

Base shell functionalities	10	8 Use getline(), strtok, strtok_r, and the provided command_line struct to grab the string and tokenize command. 5 Able to deal with “;” 5 exit implemented
Error Handling	10	2 Handles unrecognized commands. 2 Handles wrong number of parameters. 2 Print error messages. 5 None of the above errors cause crash or memory errors.
File mode	10	Program being able to enter file mode and execute commands without crashing. At least able to deal with one command.
Interactive mode	10	10 Program being able to enter interactive mode takes the user command line by line without crashing. At least able to deal with one command.
ls	5	Use syscall functions and print out the correct content.
pwd	5	Use syscall functions and print the correct path
mkdir	5	Use syscall functions. Able to create a directory. Report error with directory already..
cd	5	Use syscall functions. Able to go to the correct directory.
cp	5	Use read(), write(), open() syscall functions. Able to handle the following case (cp file dir)&(cp dir/file dir) cp dir/file dir/file cp file file
mv	5	Use read(), write(), open(), unlink() syscall functions. Able to handle the following case mv file dir mv dir/file dir/file mv file file
rm	5	Use syscall functions. Able to remove a file. Report error when the file does not exist
cat	5	Use syscall functions. Able to display file content to terminal
Valgrind	10	1 point for any form of memory leak up to 5 points. 1 point for each memory errors up tp 5 points
Report	10	1 to 2 page report.

Conventions of how to turn on Canvas:
Firstname_lastname_image_ProjectX (an example is given below)

- firstname: alex
- lastname: summers
- image: ex. DebainUTM, DebianVbox
- Submission for: Project1
- So the name of the zip/tar file should be:
  alex_summers_DebainUTM_Project1.tar

List of allowed system calls

- **Readdir**
- **Getcwd**
- **Write**
- **Closedir**
- **Stat**
- **Mkdir**
- **Chdir**
- **Opendir**
- **Basename**
- **Open**
- **Read**
- **Remove**
- **Getopt**

Examples of NOT allowed system calls

- **Execvp**
- **Syscall**

Note:
All string methods from string library are allowed. Also, this list is just what I used to make this project. There are a bunch more system calls out there that you are allowed to use! The only ones I don't want to see are ones that just do the work for you. For example, execvp() which you can give it "cp main.c cool.c" and it'll do it all for you.

Note: some sections may have more detailed points than the total points, meaning there are more than one way you can get a 0 in that section.

1. 0/100 points if a function calls upon the Linux native command function
2. 0 if your program does not work in the required environment.
3. 20 points deduction if your Makefile does not work.
4. 0/5 for a command function if any of the listed error handling cases above cause your program to crash.
5. 0/5 for a command function to use a non-syscall to complete its core task.

6. 1 point deduction for each command function containing printf().

(If you got 0/100 because your program does not compile on my machine and you did turn in a screenshot, you may resubmit for 5 points deduction, otherwise it is a 15 point deduction)

Late Policy:

- 10% penalty (1 day late)
- 20% penalty (2 days late)
- 30% penalty (3 days late)
- 100% penalty (>3 days late) (i.e. no points will be given to homework received after 3 days)