A simple Linux/Unix Shell

In this programming project, you'll build a simple Linux/Unix shell. The shell is the heart of the command-line interface, and thus is central to the Linux/Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux/Unix programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably `bash`. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

Program Specifications

Basic Shell: `tsh`

Your basic shell, called `tsh` (short for Tiny Shell), is basically an interactive loop: it repeatedly prints a prompt `tsh> ` (note the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types `exit`. The name of your final executable should be `tsh`.
The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument way:

```
prompt> ./tsh
tsh>
```

At this point, `tsh` is running, and ready to accept commands. Type away!

The mode above is called *interactive* mode, and allows the user to type commands directly. The shell also supports a *batch mode*, which instead reads input from a batch file and executes commands from therein. Here is how you run the shell with a batch file named `batch.txt`:

```
prompt> ./tsh batch.txt
```

One difference between batch and interactive modes: in interactive mode, a prompt is printed (`tsh> `). In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception are *built-in commands*, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types `ls -la /tmp`, your shell should run the program `/bin/ls` with the given arguments `-la` and `/tmp` (how does the shell know to run `/bin/ls`? It's something called the shell **path**; more on this below).

Structure

Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. That's it!

For reading lines of input, you should use `getline()`. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in *interactive mode*, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support *batch mode*, in which the shell is given an input file of commands; in this case, the shell should not read user input (from `stdin`) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call `exit(0)` and exit gracefully.

To parse the input line into constituent pieces, you must use `strtok() (or strsep())`. Read the man page (carefully) for more details.

To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. See the man pages for these functions, and also read the relevant book chapter for a brief overview. You will note that there are a variety of commands in the `exec` family; for this project, you must use **`execvp`**. You should **not** use the `system()` library function call to run a command. Using system() in your program will result in 0 point for your project. Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. You can find example usages of execvp() in page 51

(Chapter 3) in *Advanced Linux Programming*, you should have already read as part of an reading assignment.

Paths

In our example above, the user typed `ls` but the shell knew to execute the program `/bin/ls`. How does your shell know this?
It turns out that the user must specify a **path** variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the *search path* of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

Important: Note that the shell itself does not *implement* `ls` or other commands (except built-ins). All it does is find those executables in one of the directories specified by `path` and create a new process to run them.
To check if a particular file exists in a directory and is executable, consider the `access()` system call. For example, when the user types `ls`, and path is set to include both `/bin` and `/usr/bin`, try `access("/bin/ls", X_OK)`. If that fails, try "/usr/bin/ls". If that fails too, it is an error.
Your initial shell path should contain one directory: `/bin`
Note: Most shells allow you to specify a binary specifically without using a search path, using either **absolute paths** or **relative paths**. For example, a user could type the **absolute path** `/bin/ls` and execute the `ls` binary without a search path being needed. A user could also specify a **relative path** which starts with the current working directory and specifies the executable directly, e.g., `./main`. In this project, you **do not** have to worry about these features.

Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your tsh source code, which then will exit the shell.
In this project, you should implement `exit`, `cd`, and `path, and logging` as built-in commands.

- `exit`: When the user types `exit`, your shell should simply call the `exit` system call with 0 as a parameter. It is an error to pass any arguments to `exit`.

- `cd`: `cd` always take one argument (0 or >1 args should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.
- `path`: The `path` command takes 0 or more arguments, with each argument separated by a colon character (i.e., :) from the others. A typical usage would be like this:
  -
  - `tsh> path /bin:/usr/bin`
    - It would add `/bin` and `/usr/bin` to the search path of the shell.
  - `tsh> path /bin:/usr/bin:./`
    - It would add `/bin`, `/usr/bin, and the current directory` to the search path of the shell.
  - `tsh> path`
    - `It just print out the current path information to the screen, e.g.,`

      -
        ```
        tsh> path
        path is set to /bin:/usr/bin:./
        ```
  - If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The `path` command always overwrites the old path with the newly specified path.
- `logging`: The `logging` command takes 1 or two arguments, with each argument separated by whitespace from the others. There are two possible usages of `logging command:`
  - `option-1)  logging on outputfile.txt`
    - `All internal and external commands typed by the user will be logged into the outputfile.txt after running this program.`
  - `option-2) logging off`
    - `The output file that has been used for logging will be closed and no further logging activities will be done to the output file.`

Program Errors

The one and only error message. You should print this one and only error message whenever you encounter an error of any type:

```
    ~~char error_message[30] = "An error has occurred (from yourInitials)\n";~~
        char error_message[] = "An error has occurred (from yourInitials)\n";
    write(STDERR_FILENO, error_message, strlen(error_message));
     // note that the “yourInitials” should be replaced by the intials of your first
and last name
     // For example, if your name is John Doe, “yourInitials” should be replaced by
“JD”.
```

The error message should be printed to stderr (standard error), as shown above.

After most errors, your shell simply *continue processing* after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling `exit(1)`.
There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to `ls` when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

Miscellaneous Hints

Remember to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as `ls`).
Next, add built-in commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces (` `) and tabs (`\t`). In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators~~; however, the operators (redirection and parallel commands) do not require whitespace.~~
Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be.

Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

Deliverables:

A single zip file should be submitted to the ReggieNet website. It should contain the following files:

- Source code files (C files and header files if you also have header files)
- Makefile
- A single text file describing the logic of your program and also how each API has been used. (300 words)

Plagiarism

If you started to search Internet websites for sample implementations of a shell program in C (example websites: Github.com, freelancer.com, Stackoverflow,com, and chatgpt3 etc.), you are putting yourself in great danger. For your information, the following is a sample list of activities considered as cheating:

1) use of code implementing a shell program, which you have found on Internet websites including Github.com, Chegg.com, and Stackoverflows
2) hire freelancer to do the work for you (e.g., use of freelancer.com)
3) Google search for sample implementations of Shell program
4) Use of code that you have obtained from other students inside or outside IT383 class
5) ask ChatGPT3 to write code for you

The following is a sample list of activities that I am going to use to catch cheatings:

1) I am going to use special software tool check plagiarism.
2) A certain percent of students in class will be interviews with me. During the interviews, each student will be asked to explain in detail how each API/algorithm/function has been implemented.

Any form of cheating will result in three forms of penalties:

1) 0 point will be given for this assignment.
2) Their final letter grade will be lowered by one letter graded.
3) They will be reported to Dean of Students.

<u>**In short, DO NOT use someone else's code!!**</u>

Grading rubric:

- If any extra feature which has not been mentioned in his document is implemented (e.g., pipe and redirection), you will lose 50% of the maximum points automatically and also we will have a meeting to discuss about Academic Dishonesty. If you are not sure about this policy, please talk to me.

- If writeup is missing, you will lose 10%.
- If makefile is missing, you will lose 5%.
- Compilation error results in a failing grade (i.e., 0 point)
- Runtime error (e.g., segmentation fault) will result in at least 10% off.
- Command-line user interface (parsing input commands): 10 points
- Handling external commands: 40 points out of 80 points
- Handling internal commands: 30 points out of 80 points
    - Exit: 5 points
    - Cd: 5 points
    - Path: 10 points
    - Logging: 10 points
- I will also check if each API mentioned in this document has been used correctly.

Note that you MUST implement only the features mentioned in this document. Extra features not mentioned in this document will result in loss of lots of points and you will need to prove that the code was solely written by yourself in front of me in my office.