CS406 Project 1 "Simple Shell" (Due March 2nd, 11:59pm)

Project Assignment

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the 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 familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary minimal functionality in shells.

You will need to download the project assignment from the Moodle, which can be found from the moodle under the Projects section.

The name of the zip file is: `cs406_s25_project1.zip`

There you will also find this PDF document desccribing how to configure your development environment.

Your solution will be the modified version of this zipped directory, with all your code residing in a single file (`lsh.c`).

To zip and unzip the directory, you can use the `zip` and `unzip` commands, which are in your development environments.

```
pfaffmaj@pfaffmaj-mbpro ~ % zip -r cs406_s25_project1.zip cs406_s25_project1
updating: cs406_s25_project1/ (stored 0%)
  adding: cs406_s25_project1/.DS_Store (deflated 97%)
  adding: cs406_s25_project1/run-tests.sh (deflated 69%)
  adding: cs406_s25_project1/Makefile (deflated 18%)
  adding: cs406_s25_project1/SimpleShellREADME.pdf (deflated 2%)
  adding: cs406_s25_project1/tests/ (stored 0%)
  adding: cs406_s25_project1/tests/8.err (stored 0%)
  adding: cs406_s25_project1/tests/2.in (deflated 10%)
  ...

pfaffmaj@pfaffmaj-mbpro ~ % unzip cs406_s25_project1.zip
Archive:  cs406_s25_project1.zip
   creating: cs406_s25_project1/
```

To move the directory to the server use the Secure Shell Copy command (`scp`), which is used when configuring SSH paired keys.

```
scp cs406_s25_project1.zip yourID@139.147.9.135:.
```

To download the zipped archive from the server, you can reverse the process.

```
scp yourID@139.147.9.135:cs406_s25_project1.zip .
```

The TL;DR

This is an individual assignment that can be explored in any Unix environment (Linux, OSX, WSL), but will be graded in the provided server environment `139.147.9.135`. Your credentials for this server are shared from the professor's Google Drive. There is also a PDF document describing the configuring your system on the course Moodle page.

To make your development process simpler, some scripts have been provided. The first is a `Makefile` that provides commands for compiling and cleaning the project. The file is displayed here:

```
pfaffmaj@cs203-0:~/cs406_s25_project1_solution$ more Makefile

compile:
        gcc -o lsh lsh.c -lc -g

doc:
        pandoc -i README.md -o README.pdf

test:
        ./test-lsh.sh

clean:
        rm -f lsh *~
        rm -fr tests-out
```

`Makefile`s are very useful for creating simple commands and naming them. This `Makefile` provides commands for compiling, creating this document, running a code test suite, and cleaning your code. An example for each command is given here.

```
pfaffmaj@mbpro cs406_s25_project1 % make compile
gcc -o lsh lsh.c

pfaffmaj@mbpro cs406_s25_project1 % make doc
pandoc -i README.md -o README.pdf

pfaffmaj@mbpro cs406_s25_project1 % make test
./test-lsh.sh
test 1: 1.err incorrect
  what results should be found in file: tests/1.err
  what results produced by your program: tests-out/1.err
  compare the two using diff, cmp, or related tools to debug, e.g.:
  prompt> diff tests/1.err tests-out/1.err
  See tests/1.run for what is being run
make: *** [test] Error 1

pfaffmaj@mbpro cs406_s25_project1 % make clean
rm -f lsh *~
```

The provided tests can also be run directly from the shell command line using the `./test-lsh.sh` command. But to run individual tests, you can use `./run-tests.sh` (demonstrated below on the server).

```
pfaffmaj@cs203-0:~/cs406_s25_project1_solution$ ./run-tests.sh -h
usage: run-tests.sh [-h] [-v] [-t test] [-c] [-s] [-d testdir]
  -h                help message
  -v                verbose
  -t n              run only test n
  -c                continue even after failure
```

```
  -s                skip pre-test initialization
  -d testdir        run tests from testdir

pfaffmaj@cs203-0:~/cs406_s25_project1_solution$ ./run-tests.sh -t 5
test 5: passed
```

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. On the sever you are running `bash`, but on your personal machine it can be an alternative. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

Manual pages, or man pages, are provided through your shell. For example the following is obtained by running `man 3 exec` on my OSX terminal. The three indicates this page comes from the third manual. This document describes the diffferent types of exec commands that are used to load a program over a child process forked by a parent.

```
EXEC(3)                                                 Library Functions Manual

NAME
     execl, execle, execlp, execv, execvp, execvP - execute a file

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <unistd.h>

     extern char **environ;

     int
     execl(const char *path, const char *arg0, ..., /*, (char *)0, */);

     int
     execle(const char *path, const char *arg0, ..., /* (char *)0 char *const envp[] */);

     int
     execlp(const char *file, const char *arg0, ..., /*, (char *)0, */);

     int
     execv(const char *path, char *const argv[]);

     int
     execvp(const char *file, char *const argv[]);

     int
     execvP(const char *file, const char *search_path, char *const argv[]);
```

As a second example, if you were to run the command **man printf** you will recieve the manual page for the printf command.

```
PRINTF(1)                                               General Commands Manual
```

```
NAME
     printf - formatted output

SYNOPSIS
     printf format [arguments ...]

DESCRIPTION
     The printf utility formats and prints its arguments, after the first, under control of the format.
     string which contains three types of objects: plain characters, which are simply copied to standar
     sequences which are converted and copied to the standard output, and format specifications, each o
     next successive argument.
```

But instead, maybe you wanted the manual page for the C printf, then you would run `man 3 printf` obtaining the following.

```
PRINTF(3)                                                  Library Functions Manual

NAME
     printf, fprintf, sprintf, snprintf, asprintf, dprintf, vprintf, vfprintf, vsprintf, vsnprintf, vas
     output conversion

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     int
     printf(const char * restrict format, ...);
```

The best part about the man system, is that these pages give a great amount of detail and do not require access to the internet.

Running the Program Environment

If you do need more of an overview to the different shell commands, you can find them in this handy webpage.

Learning the Shell (link:https://linuxcommand.org/lc3_learning_the_shell.php)

Running the Program Environment

Described previously, a `Makefile` is provided that will perform the following:

- **compile** This will compile the `lsh.c` file using gcc.
- **doc** Rebuild the README.md file that will allow you to see it as a PDF file.
- **test** will run a set of unit tests against your program if you are just doing a normal make file compile. For this option, you will need to install pandoc.
- **clean** This will remove the executable and any modified files.

This project can be developed using any IDE you wish on your personal machines. But you need to use the makefile to build the program and test it on the server, where it will be graded. This can be done from the command-line. Please be aware, just because it runs perfectly in your environment does not mean it will run the same on the server.

Program Specifications

Importantly, all program code in your `lsh.c` must be fully commented explaining what you are doing.

Basic Shell: `lsh`

Your basic shell, called `lsh` (short for Lafayette Shell), is an interactive loop that: repeatedly prints a prompt `lsh>` (with a 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 `lsh`.

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> ./lsh
lsh>
```

At this point, `lsh` 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> ./lsh batch.txt
```

One difference between batch and interactive modes: in interactive mode, a prompt is printed (`lsh>`). 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

In general, the shell is (conceptually) very simple: 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 might want to use `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 `execv`. You should **not** use the `system()` library function call to run a command. Remember that if `execv()` 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.

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 lsh source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, and `path` 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 whitespace from the others. A typical usage would be like this: `lsh> path /bin /usr/bin`, which would add `/bin` and `/usr/bin` to the search path of the shell. 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.

Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the `>` character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types `ls -la /tmp > output`, nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. In addition, the standard error output of the program should be rerouted to the file `output` (the twist is that this is a little different than standard redirection).

If the `output` file exists before you run your program, you should simple overwrite it (after truncating it).

The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors.

Note: don't worry about redirection for built-in commands (e.g., we will not test what happens when you type `path /bin > file`).

Parallel Commands

Your shell will also allow the user to launch parallel commands. This is accomplished with the ampersand operator as follows:

```
lsh> cmd1 & cmd2 args1 args2 & cmd3 args1
```

In this case, instead of running `cmd1` and then waiting for it to finish, your shell should run `cmd1`, `cmd2`, and `cmd3` (each with whatever arguments the user has passed to it) in parallel, *before* waiting for any of them to complete.

Then, after starting all such processes, you must make sure to use `wait()` (or `waitpid`) to wait for them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

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\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

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

If you are not sure where to start, read chapter 5 (Interlude: Process API) of the text book and work through the coding homework. This will give you the essential parts to create children processes and see how fork/exec work.

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. Then, try working on redirection. Finally, think about parallel 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.