CSCI-442 Project 2: UNIX Shell

> You'll want to read this **entire document** before beginning the project. Please ask any questions you have on the discussion board, but only if this README does not answer your question.
>
> Finally, this is a **large project**. Be sure to start early. If you wait until a few days before the due date, you are unlikely to finish in time.

For this project, you will implement a UNIX Shell (similar to shells Like `bash`), using the C programming language. The shell serves as the user interface of an operating system, and some shells provide additonal functionality, such as scripting.

1) Learning Objectives:

- Understand the system call ABI to UNIX systems by writing a program which uses system calls such as `fork`, `execvp`, and more.
- Become familiar with UNIX file descriptors, both associated to files on the filesystem and to pipes, by implementing a shell which supports pipelines, file input, file output, and file appending.
- Sharpen systems programming skills by working on a large project.

Here are the important deadlines for you to know:

Deliverable	Due Date
Deliverable 1	Check Canvas Assignment
Deliverable 2	Check Canvas Assignment

The function of multiple deliverables is to prevent you from falling behind on this large project. Please note:

- Solutions to D1 will not be provided after the due date. You are expected to make D2 function on-top of your own work from D1.
- It is recommended to finish D1 significantly before the provided due date so that you can get a head-start on D2. **D2 is much more difficult than D1**.

2) General Requirements

Using the starter code *is not a requirement of this project*. You are free to discard any amount of the starter code, as long as your final project conforms to the requirements outlined here.

General requirements for all deliverables:

- You are free to develop your code in whichever environment you like, but the code you submit code must compile and function on Gradescope. The Gradescope enviroment is the same as Isengard (Ubuntu 20.04).
- Your code must be written using only the C programming language. Do not extend using additional languages. The exception is for the Makefile for your project, you may write the build system for your code using any available language on Isengard.
- To compile your code, the grader should be able to `cd` into your repository and run `make`. To run your code, the grader should be able to type `./shell`.
- Your code cannot depend on any shells installed on the system, such as `/bin/bash`, `/bin/sh`, or `/usr/bin/zsh`, except at build time. As a corollary to this, you may not use the `system(3)` or `popen(3)` library functions, as these depend on the availability of a system shell.
- Your code should not assume that system calls succeed. **Always, always, always, check the return code from system calls**, and print an error message where appropriate.

- You should follow good code formatting and software engineering practices. This includes avoiding monolithic functions, freeing heap memory you allocate, writing comments where appropriate, and following The Linux Kernel Style Guide.

3) Project Requirements

The specific requirements, as well as the associated deliverable, are outlined below. Every requirement is prefixed with [D1] or [D2] to indicate the corresponding deliverable it must be working by.

All functionality due in Deliverable 1 must continue to function when you submit Deliverable 2.

Should you choose to go without starter code (or discard portions of it), all functionality provided by the starter code is due with the first deliverable, with the exception of the "history" and "help" builtin commands, noted below.

Through this document, the term *whitespaces* or *whitespace characters* refers to any of the following characters:

- Spaces
- Tabs
- Newlines
- Carriage returns
- Vertical tabs
- Form feeds

The term *word characters* refers to any characters which are not whitespace characters, nor >, <, or `|`.

3.1) External Commands (Deliverable 1)

If you are using all of the starter code, this is likely to be the first thing you implement.

这里的path指的是什么?

[D1] When the user types a command which is not known as a builtin to the shell, the shell should find the command in the `PATH` and execute the command using `fork(2)` and `execve(2)`. You may use any of the `exec*` family of library functions (such as `execvp(3)`) to help you find the command in the `PATH` before executing it, if you wish.

[D1] The shell should wait on the external command finishing before returning to the prompt. As an example, you should be able to type `gedit`, the editor will open, and you won't get your shell prompt again until the editor is closed. See `man 2 wait` for info on how to do this.

3.2) Pipes (Deliverable 2)

[D2] Your shell should be able to handle an arbitrary number of commands piped together. For example:

```
command1 | command2 | command3 | command4
command1 arg1 arg2 | command2 | command3 arg1 arg2
command1 < inputfile | command2 | command3 > outputfile
```

For an example of a real piped command, try this (which gives the number of lines in `mains/parseview.c` which contain the word `int`):

```
cat mains/parseview.c | grep int | cat
```

Or maybe use `cat` on a previous command:

```
date | cat | cat | cat
```

For this command, you should get the current date (assuming your shell handles pipes properly).

> Note: You may not make use of temporary files in your implementation of pipes. This means you are going to have to use the `pipe(2)` system call.
>
> Additionally, you are not expected to support more than `PIPE_BUF` bytes sent between two processes. On Isengard, this value is 64 kilobytes.

3.3) File Redirection (Deliverable 2)

[D2] Your code must handle file redirection using > (overwrite to a file), >> (append to a file), or < (input from a file).

For example:

```
command > file-to-write-or-overwrite.txt
command >> file-to-append-to.txt
command < file-to-get-input-from-as-stdin.txt
```

[D2] For > and >>, you should create the file if it does not exist.

[D2] You should support < at the beginning of a pipeline, and >> or > at the end of a pipeline.

[D2] Think carefully about the permissions you create files with. With `open(2)`, you provide the value which gets paired with `umask`. What number should you use for files then? (Hint: files should not have the executable bit set)

3.4) Terminal Interaction

The following information has already been implemented in the starter code. If you want to use all the starter code, then you can skip ahead to Section 4 and then begin working on handling an external command.

If you do not wish to use the starter code, you must read sections 3.4 - 3.4.4 to have the proper formatting that the grader will expect when running your code.

3.4.1) Input Logic

The general operation of your shell should consist of the following:

1. Initialize the last exit status to `0`.
2. Use the `readline` function to accept a command on input. The prompt string passed to the `readline` function is described below. If `readline` returns `NULL` (EOF), your shell should behave as if `exit` was typed.
3. If the line is blank, or consists of only whitespaces, goto step 2.
4. Execute the command, and wait until completion of the command.
5. Update the last exit status to reflect the result of the command.
6. If the command typed was the `exit` builtin (see below), your shell should exit, returning either the provided exit status from your process, or the last exit status if none was provided.
7. Goto step 2.

3.4.2) The Prompt

The prompt should be entirely on one line (no newline characters), and must contain `:)` if the last exit status was zero, or `:(` if the last exit status was non-zero.

The prompt must end with `$`, followed by a single space.

You can include any additional text in the prompt as you wish, so as long as the above requirements are held true.

> Note: The starter code implements the prompt functionality in `src/interact.c`, and provides the following pre-implemented prompt line, which meets the above requirements:

- Username
- @ symbol
- Hostname
- Space
- Current working directory
- Space
- `:)` or `:(`
- Space
- $ symbol
- Space

3.4.3) Input Parsing

The input consists of a *pipeline*. A *pipeline* is one or more *commands* chained together by a `|` character. Note: the `|` character may be surrounded by zero or more whitespace characters on either side.

A *command* consists of one or more ordered arguments, each of which is a sequence of one or more word characters. The special prefix operators >, >>, and < consume the following argument, and designate the file path a truncated output, appended output, or input file respectively.

Note that these prefix operators may appear at the beginning, in the middle of, or at the end of the arguments list.

It is invalid to specify a command for which any of the following scenarios apply:

- The command has mutiple input files.
- The command has an input file, but is not the first command in a pipeline.
- The command has multiple output files (either truncated or appeneded).
- The command has an output file, but is not the last command in a pipeline.

The following are valid example commands:

- `arg`
- `arg1   arg2`
- `arg1 arg2 >outfile <infile`
- `<  infile arg1 >outfile arg2`
- `arg1 arg2   arg3  <  input_file arg4 | cmd2_arg1 >> append_file`

> For Deliverable 1 only...
>
> You may assume the characters `|`, <, and > do not appear in the input.
>
> Note: The starter code already implements (and uses) an input parser for you. The documentation for this parser can be found in `include/parser.h`.
>
> Regardless of whether you choose to use our provided input parser or develop your own, it is recommended that you observe the interpretation of the above examples and more using the provided `parseview` program in the starter code.

3.4.4) Builtin Commands

Builtin commands are commands supported by the shell which do not require running an external program.

For all bulitin commands, if the user provides an invalid input (such as incorrect number of arguments, provides a non-existent file or directory, etc.), your shell should print an approprite error message on `stderr` and indicate the command failure status in the prompt.

> Note: All required builtin commands are implemented in the starter code (in `src/shell_builtins.c`) for you already. This is provided for reference if you decide to not use that portion of the starter code.
>
> The provided starter code also implements `history` and `help` builtin commands. It is not required that you implement these. You are free to discard these commands if you don't want them.

Finally, builtin commands (`cd` and `exit`) do not need to work with pipelines of more than a single command, nor input or output files.

The `exit` command takes zero or one arguments. If zero, the shell should exit with the last return code. If one argument is passed, it should be a number indicating the exit code to exit with.

The `cd` command takes one argument, the directory to change to, which can be a relative or absolute path.

If `cd` is called with no arguments, it should change to your home directory (as specified by the `HOME` environment variable).

4) An Introduction to the Starter Code

The starter code provides an input loop, input parser, and builtin commands, as well as a Makefile, but does not dictate how you should structure the code for external commands, pipelines, or redirection.

It's **up to you** to break your code into useful helper functions, or even separate files entirely.

You'll want to start taking a look at `src/dispatcher.c`, where `dispatch_external_command` will be the *entry point* into your code. If you do it right, hopefully you'll find that `dispatch_external_command` turns out to just be a very short function that calls out to some of your other functions that you've written.

If you are just opening the starter code and want to look for somewhere to start, try this:

1. Run `make` and observe the output directories and where the programs end up.
2. Run `./shell`. Observe the working builtin commands, and what happens if you try to run an external command.
3. Run `./parseview` and type some commands. Each output shows the resultant `struct command` structure than you'll get at `dispatch_external_command`. Use this tool to rationalize the meaning of each of the fields in the struct.
4. Open the `src/dispatcher.c`, read thru the comments in there.
5. Start hacking away at `dispatch_external_command`!

5) Reference Executables

Provided for you are two reference executable files called `shell_d1` and `shell_d2`. They are working versions of D1 and D2 that score 100% in the autograder. You may use them to help understand the behavior of a working project. They are included in this template repository and can be run with `./shell_d1` or `./shell_d2`. Things to keep in mind about the reference executables:

- They were developed on isengard and are only guaranteed to work on isengard. See the video on Ed for information about setting up isengard with vscode.
- They are instructor versions and you may not execute them from within your own code. The autograder checks for this and you will get a zero.

6) Grading

Each deliverable is 50% of the grade. For each deliverable, you'll be graded on:

- Functionality (the specific features we ask for in this document). This is most of the grade.
- Code Quality
  - Follows Linux Kernel Style Guide.
  - Useful error messages are printed to `stderr` when a system call fails.
  - All opened files are closed before your program exits.
  - All heap memory allocated explicitly is freed with `free()`.
  - Memory safety:
    - `strcat`, `sprintf`, and `strcpy` are not used.
    - Corruption impossible.
    - No segmentation faults.
    - Use `./shell.debug` for additional memory safety testing.
  - Avoids monolithic functions. Makes good use of helper functions.
- Correct Submission
  - Code compiles without errors (note: you will receieve a zero on functionality as well if this is not true)
  - Code submitted with `make-submission`
- For D2 only: no regressed features from D1.

> Warning
>
> You will receive a zero on the project if your code anyhow depends on one of the system shells. Do not use `system()` or `popen()`: these depend on the system shells.

7) Additional Resources

- Don't forget the man pages! System functions are under `man 2`, and library functions under `man 3`.
- Ask questions in the online discussion board. If you are going to post code, please keep the post private as to comply with the collaboration policy.
- Please attend office hours if you find yourself falling behind. Don't wait until the last week to seek help.

8) Collaboration Policy

This is an **individual project**. All code you submit should be written by yourself. You should not share your code with others.

Please see the syllabus for the full collaboration policy.

WARNING: Plagarism will be punished harshly!

9) Access to Isengard

Remote access to Isengard is quite similar to ALAMODE, but the hostname is `isengard.mines.edu`.

For example, to `ssh` into the machine with your campus MultiPass login, use this command:

```
$ ssh username@isengard.mines.edu
```

A tutorial has been linked in the discussion board to `ssh` via Visual Studio Code.

Note: you need to be on the campus network or VPN for this to work. If you are working from home, use either the VPN or hop thru `jumpbox.mines.edu` first.

10) Submitting Your Project

Submission of your project will be handled via **Gradescope**.

1. Create the submission file using the provided `make-submission` script:

   ```
   prompt> ./make-submission
   ```

2. This will create a `.zip` file named `$USER-submission` (e.g., for me, this would be named `lhenke-submission.zip`).
3. Submit this `.zip` file to Gradescope. You will get a confirmation email if you did this correctly.

   WARNING: You are **REQUIRED** to use `make-submission` to form the `.zip` file. Failure to do so may cause your program to not compile on Gradescope.