Advanced - Unix
Writing a Shell

EECS 201 Fall 2025

Submission Instructions

This assignment will be submitted as a repository on the EECS GitLab server (see Basic - Git 1). Create a private, blank, **README-less (uncheck that box!)** Project on it with the name/path/URL `eecs201-adv-unix` and add `mmdarden` as a Reporter. The submission branch will be `submit`. If this branch is not already the default initial branch, you initialize the local repo with an additional argument: `git init --initial-branch=submit` or `git init -b submit` if your version of Git is recent enough. Otherwise you can create a branch with this name after your first commit.

To finish the submission process and get your grade, you will need to run the autograder. You will need to SSH into the course server (see Basic - Intro):
`local$ ssh uniqname@peritia.eecs.umich.edu`
(the `local$` referring to a shell prompt on your system)

and then run `eecs201-test`:
`peritia$ eecs201-test adv-unix`
(the `peritia$` referring to a shell prompt on the server)

The repository should have the following directory structure, starting from the repository's root:

```
/
|-- report.txt
|-- Makefile
|-- (source files)
```

Do not commit any build output (compiled executables and object code) or any file system data like `.DS_STORE` on macOS.

Preface

This assignment can be done on both macOS and Linux.

First, we'll need to acquire a set of starter files. This assignment can be done in C or C++, whichever you prefer Use your preferred method of downloading files from a given URL:

```
# C version
https://www.eecs.umich.edu/courses/eecs201/fa2025/files/assignments/adv-unix-c.tar.gz

# C++ version
https://www.eecs.umich.edu/courses/eecs201/fa2025/files/assignments/adv-unix-cpp.tar.gz
```

Inside these archives there is a starter file and a Makefile. You can compile the program by running `$ make`.

1 Programming with POSIX (10)

This week's class exit survey asked if you think you could write a shell. No matter what your answer was, I believe that you are capable of writing a basic shell :)

One video briefly mentioned how proccesses are created in Unix by `fork`-ing and `exec`-ing. Let's showcase some POSIX programming and get some practice reading `man` pages with this exercise. In this exercise you'll be creating

"μShell", or "`mush`", using C or C++ (whichever you prefer). `mush` is a simple, minimalist shell whose only job is to execute the commands presented to it.
Its specifications are:

- Presents a prompt of "`<username>:<current working directory>$ `" on **standard output**.

  For example, for a `USER` "doe": "`doe:/home/doe$ `". Note the space at the end. Since users type in their command after this, don't print a newline. Assume that the current working directory can only be represented by at most 255 characters. **The `PWD` environment variable does not automatically update to where your current directory is: you should not use it for your prompt. It's actually shells that manage this variable!** However, feel free to use the `USER` environment variable.

- Assume that the entered input has at most 255 characters and that there will be at most 15 arguments. Feel free to handle more, but the input is guaranteed to meet these restrictions in testing. When dealing with C strings, remember that you need space for the null terminator as well. Also keep in mind the structure of the array that holds the arguments: what size does the array need to be if it holds 15 arguments? It sounds like a weird question, but if you read the documentation it'll make more sense ;)

- If the entered input is empty, prompts the user again.

- Stops, prints a newline, and exits (i.e. returns from `main`) with a 0 if the end of the file (EOF) is reached is encountered with an empty line. You can send the EOF with Ctrl-D at a terminal (try it out and see what happens with Bash or Zsh!).

- Stops and exits with a 0 if the command is `exit` with no additional argument. If there is an additional argument in the form of an integer, it exits with that integer value. **You may assume that this additional argument will always be an integer.** This is the return value of your `main` function.

- Changes the current working directory if the command is `cd`; if no path is specified, the current working directory is set to the `HOME` environment variable. If the directory does not exist, prints
  "`mush: cd: no such file or directory '<directory path>'`" on **standard error**.

  For example: "`mush: cd: no such file or directory '/emoh/doe'`". This string should include a newline at the end so that the prompt appears on the next line.

- Executes the entered command and with its arguments. These commands are either in the `PATH` or specified with a path (i.e. has a forward slash in it). That means you do not have to implement any other shell built-ins besides `exit` and `cd`.

- Waits for the command run to be complete (see `waitpid`).

- If the command does not exist, prints "`mush: command '<command name>' not found`" on **standard error**. For example: "`mush: command 'iamnotacommand' not found`". This string should include a newline at the end so that the prompt appears on the next line. If a child process was created, the child should exit.

Note that there are no built-in commands (besides `exit` and `cd`), job control, file redirection, or signal handling specified (if you want to, you can do them for personal edification).
Some helpful functions (the string processing ones are more for C; if you're using C++ you might have other mechanisms that fulfill the same role):

- `getenv`
- `getcwd`
- `fgets`
  - `fgets` has a gotcha where it'll include the newline character. Be sure to deal with it accordingly
- `strtok` and its reentrant sibling `strtok_r`
  - `strtok(_r)` has a gotcha where additional invocations on the same string have the `str` argument be `NULL`.
- `strcmp`
  - Note that 0 is returned when strings match.

- `chdir`
  - Note that `chdir` does not change the `PWD` environment variable for you (you don't have to worry about `PWD`).
- `fork`
  - Take note of how the return value differs between the parent and child process.
- `execvp`
  - The `man` pages for `execve` and `execvp` may offer some more info. Be very careful to read up on the data format of the arguments...
- `waitpid`

You can see documentation for these functions by using `man`. Sometimes, there may be multiple `man` pages for a given function. `man` pages have multiple sections: sections 2 and 3 are the programmer's manual, with 2 being the operating system API and 3 being about library functions. You may often see things like `fork(2)`: this refers to `fork` under section 2. You can specify a section to look at as an argument: `man 2 fork`.
Here are some helpful variables/macros for error checking:

- `errno`
- `ENOENT`

You can read about them in the `errno` `man` page: `man 3 errno`.
Below is an example way to structure the program. Remember that there are many different solutions possible:

```
1  // forever loop
2    // print prompt
3    // receive user input
4    // handle EOF
5    // tokenize input
6    // handle "exit" and "cd"
7    // handle non-builtins
8      // fork
9      // child
10       // execute command
11       // handle errors
12     // parent
13       // wait for child
```

In the file archive you downloaded, you can find a Makefile and starter file for your preferred language in the appropriate directory. You may only use the C standard library, the C++ standard library, and any POSIX function. **When you work on this, keep in mind the header files required by each function: the manpages will tell you.** This can be implemented in surprisingly little code: it's possible to do this in less than 50 lines of plain old C!
For me the receiving and parsing input was the most difficult part of this.

This is an autograded assignment: you need to run the autograder to get credit. Make sure that your code is able to compile on the course server! You can clone your repo onto the course server if you create a keypair on the server and share that public key with the EECS GitLab. **It's been noticed some compilers on macOS will implicitly handle the `#include` of certain POSIX header files while the compiler on the course server does not, resulting in a situation where code written on macOS will compile but not on the course server since the course server is more strict about the `#include`s. Be mindful of what `#include`s you need!**

Things to keep in mind:

- ☐ Does it present the correct prompt **on standard output**?
- ☐ Does it execute commands with arguments?
- ☐ Does it print the command not found message?
- ☐ Does it handle empty input?
- ☐ Does it print a newline and return 0 when an EOF is encountered??

☐ Does it handle both cases of `exit`?

☐ Does it change directories using `cd`?

☐ Does it print the directory not found message with `cd` **on standard error**?

☐ Does it print the command not found message **on standard error**?

☐ Why should the parent handle `exit` and `cd` and not the child?

☐ Have your run this assignment on the autograder?

Report

Create a file `report.txt` in your repository's root: `<repo path>/report.txt`. On the first line of the file, put down an estimate of how long you took to do this assignment in minutes as an integer (e.g. "37", "84": just numbers, no letters). On the second line and onwards of the file, put down what you learned (if anything) by doing this assignment. If you already knew how to do all of this, just put "N/A".