Assignment 1
CSE 134: Embedded Operating Systems, Spring 2021

Due: Monday, April 12 at noon

Goals

The goals of this assignment are to ensure that you have FreeBSD set up on your Raspberry Pi and to get you familiar with user-level access to the operating system by writing a simple shell. This assignment will also ensure that you're familiar with the tools you'll need to effectively program kernel-level C code, as you'll be doing in the rest of the class.

Basics

For this assignment, you'll be writing a simple shell that parses commands and executes them using system calls. You'll be handling I/O redirection.

We are providing you with `shell.l`, which will parse a line of text into an array of character strings, each of which corresponds to a single token from your shell's point of view. Note that a token might include blanks if there are quotes in the input! Your shell program calls `get_args`, which returns an array of pointers to character strings, each of which is one argument. If the token has the same quote character (`"` or `'`) as both the first and last character, remove both the first and last characters from the token.

To compile `shell.l`, you have to use the `flex` command. This produces a file called (by default) `lex.yy.c`. You must then compile and link `lex.yy.c` and the `.c` file(s) that contain your shell code in order to get a running program. In the link step you also have to use `-lfl` to get everything to work properly. Your link step should look like this:

```
cc -o myshell myshell.o lex.yy.o -lfl
```

Note that `-lfl` *must* come last, after all of the other files being compiled. As with all assignments, your build must be done using a `Makefile`, arranged so that `make` with no arguments builds your shell. We'll have a review on this the first week of class in section, and you may use online tutorials to help build your `Makefile` as well. Keep in mind that FreeBSD doesn't use GNU `make`.

Details

Supported features

Your shell should print a prompt (`shell>` , including the space), and read a line of input. It then processes the input using `getargs()`, providing the following features:

- Run programs using `fork()` and `execve()`.

  Single programs Your shell must be able to run a single program with zero or more arguments, such as `ls -l foo`. There can be any number of arguments, subject to the maximum number of tokens that can be handled by the `flex` file (1024). Please read the man pages for `fork()` and `execve` for help in how to pass arguments and actually do the fork. A single command consists of tokens (each string returned from `getargs()` is a token), with a *metacharacter* token ending the command. Metacharacters are those that have special meaning to the shell, and will be listed in descriptions below. Note that `exit` and other internal shell commands are *not* metacharacters.

  Input and output redirection Your shell must support input redirection (via <) and output redirection (via >). This is typically done using a combination of `open()` and some variety of `dup()`. You may also need to use `close()`. For example, `sort -nr < scores` takes the file `scores` and connects it to `sort`'s standard input. `sort -nr < scores > sorted` takes input from `scores` and connects standard output to `sorted`. if the output redirection symbol is `>&`, both standard output and standard error are redirected. A single > (or `>&`) *replaces* the named file with the output. If, instead, there's a double >> (or `>>&`), the output is *appended* to the file.

Multiple programs not connected If you have multiple programs on a single line with semicolons (`;`) between them, they will run *in series*, one after the other. They can each have their own input/output redirection, but are otherwise unconnected except that a program must exit *before* the next one on the line can start. Example: `ls ; echo "done"`

Multiple programs connected by pipes Your shell must support multiple commands connected by *pipes*. A pipe connects the output of the first command to the input of the second command, using the `pipe()` system call. Note that the first command in a pipe can have its input redirected from a file, and the last command in a pipe can have its output redirected to a file. For example,
`du -sk | sort -nr | head -10 > big`
will take the output of `du -sk` and send it to `sort -nr` as input. `sort -nr` will send its output to `head -10`, and `head -10` will send its output to the file `big`. If the pipe symbol is `|&`, both standard output and standard error are redirected.

- Internal shell commands.

  cd Change the current working directory, using the `chdir()` system call.

  exit Exit the shell (cleanly), using the `exit()` C library call.

Your shell doesn't need to support wildcards or ~ as the home directory.

Error handling

You must check and correctly handle all return values. This means that you need to read the manual pages for each function and system call to figure out what the possible return values are, what errors they indicate, and what you must do when you get that error.

Your shell must not crash *for any reason*. Don't depend on the underlying programs to be "well-behaved"—assume nothing! The underlying program may crash, but a properly-written shell will not; it'll just get a non-zero return code, which isn't a problem. You don't have to worry about more than 1024 tokens on a command line, but everything else is fair game.

Deliverables

Your code must be written in C (not C++), and must conform to the C99 standard. It must compile and build properly on a Raspberry Pi running FreeBSD 13, with the standard (`llvm`-based) compiler and `make`. This is where we will test your code. You must use the following compiler flags when you build your code, and your code must build without error. The only exception is for warnings in `lex.yy.c`: there are two functions that you'll get warnings for.

```
-Wall -Wextra -Wpedantic -Wshadow -std=c99 -O2
```

All of your code *must* be written in the `asgn1` subdirectory of your repository on `git.ucsc.edu`. For this assignment, as for all others, we *strongly* recommend that you commit code to `git` regularly, and that you push your code to `git.ucsc.edu` at least once a day. You don't have to push after each commit if you don't want to.

At the time of submission, your `asgn1` subdirectory must include your source code files, your `makefile`, and a design document called `DESIGN.pdf`. In addition, include a `README.md` file to explain anything unusual to the TA. Obviously, your `DESIGN.pdf` must be in PDF—you're welcome to write it in plain text, LaTeX, or anything else, as long as you convert it to PDF and name the resulting file `DESIGN.pdf`. Your `README.md` must be in the plain-text Markdown language; a regular plain-text (ASCII) file is valid Markdown, so you don't need to use any other features of Markdown if you don't want to.

Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs. A sample design document is available on the course web page.

Committing and submitting

You must do regular commits of your code using `git`; you'll lose points if you don't do regular commits. You need not push your commits to `git.ucsc.edu` every time you commit, but you should push on a regular basis. Please see the course Canvas page for details.

Do not commit object files, assembler files, or executables to your `git` repository. Each file in the submit directory that could be generated automatically by the compiler or `flex` (including `lex.yy.c`!) will result in a 10 point deduction from your programming assignment grade.

You must follow the full submission instructions listed under Assignment 1 on Canvas.

Hints

- Start early!
- Get your Raspberry Pi environment set up first. It shouldn't take long (much less than an hour), and will let you start developing your code quickly.
- Write your design document *before* starting to write your code. It's ok to experiment a bit to say how things work, but make sure you come up with a good design first, so you don't end up rewriting a lot of code.
- Get your design done by Friday of this week, Monday at the latest. That'll give you a full week to write your code and debug it. The code for the shell isn't that difficult *if you have a good design*.
- Break down your shell implementation into steps and ensure that each step works before proceeding, start with the Makefile, make sure that the parsing works correctly for some edge input cases (multiple indirects, excessive whitespace, use of quotes), handle creation and execution of child processes, read system call manual pages to check for the possible return values and their meaning.
- Try to leverage common code as much as possible. For example, there are four different operators for redirecting output (`>`, `>>`, `>&`, `>>&`), but they all share a lot in common—write one piece of code that handles them all, with a few conditions in it.
- Use `lldb` to debug your code.
- If you have generic questions, ask them on Piazza.

Grading

Your grade for this assignment will be determined based on the rubric posted on Canvas. A well-done design document, good coding style, and good comments are all important to your overall grade, as is functionality of your shell.

We can only grade what you commit and submit. We ***strongly*** recommend that, before making your submission, you clone your `git.ucsc.edu` repo to a new directory and try to build it. If you can't, figure out why not and fix it *before* submitting. **If you submit a commit ID that doesn't build with `make` with no arguments, your maximum grade on the assignment is 5 points**.

