Assignment 05

If any of this is unclear, please get in touch with me ASAP. There are definitely things in the course I want you to struggle with. I absolutely do NOT want you struggling with understanding my expectations.

03 Dec 22: Added Coda, at the bottom

In this assignment, you will write your own shell. It must print a prompt, accept commands from the user, support input/output redirection, and pipelines. A description of these behaviors follows.

Input redirection

Normally, a program accepts input from stdin, which is typically attached to the terminal in which the program is running. A program can be made to read from a file instead using the following syntax.

```
$ ./foo < input-file
```

Note that the program itself *does not know the difference*. It just reads from file descriptor 0; the shell just does some tricks behind the scenes to make sure file descriptor 0 is associated with `input-file` instead of the terminal.

Output redirection

Normally, a program's output goes to stdout, which is typically the terminal in which the program is running. That output can be caused to go to a file instead using the following syntax.

```
$ ./bar > output-file
```

As with input redirection, the program doesn't know this is happening: it just writes to file descriptor 1, which the shell has caused to be associated with `output-file` instead of the terminal.

In the previous invocation method, the file `output-file` should be truncated (made zero-length) before any writing begins. An alternate invocation method causes any data output by the program to be *appended* to the file instead:

```
$ ./baz >> output-file2
```

Piping

Instead of saving the output in a file, the shell can cause a program's output to be delivered directly to *another* program's input. This technique is called *piping* and is indicated using the pipe character: "|".

```
$ ./program1 | ./program2
```

In this instance, `program1` happily writes to file descriptor 1 and `program2` happily reads from file descriptor 0. Except the shell has done some magic to ensure that whatever `program1` writes to its fd1 appears on fd0 of `program2`.

With two exceptions, the net effect is equivalent to the following command.

```
$ ./program1 > temp
$ ./program2 < temp
```

The two exceptions are that, in the case of piping, the programs are running *simultaneously*, whereas in the latter example they are running sequentially; and the latter example creates a file to temporarily store the output before passing it on to the second program, whereas with piping there is no temporary file.

New Mechanics

You might find these useful in this assignment.

New functions

- `fork(2)`
- `exec(3)`
- `wait(2)`
- `pipe(2)`
- `dup(2)`
- `fgets(3)`
- `strtok(3)`

Some of these are not strictly necessary (ie, you can implement the assignment without them) but they are sufficiently standard for the types of problems you're going to be solving herein that you should be aware of their existence.

Behavioral requirements

You will produce a source file, `mysh.c`, that implements your shell. When compiled and run, it should behave just like the shell you interact with:

- it should print a prompt (it doesn't have to be fancy, but it should be clear when the shell is awaiting a command);
- when a command is typed (ending in newline), it should run the command;
- when the command has finished, it should print a new prompt.
- It should continue reading and running commands until it reads end-of-file or the user enters the command "exit". (From the keyboard, you can indicate end-of-file by hitting Ctrl-D.)
- If the command contains input redirection (<), output redirection (> and >>), or pipes (|), it should behave as described above.
- In the case of piping, your shell should support an arbitrary number of pipes in a single command.

You may make some assumptions about commands being inputted:

- A single input will never exceed 4096 characters in length (including the trailing newline and null terminator).
- Individual commands (ie, single entities for which the shell will fork(2), and which may be separated by pipes) will have a maximum of 10 arguments, arguments will be separated by a single space, and commands will features neither leading nor trailing spaces. (I don't want you to get too hung up on parsing.)
- Any I/O redirection or piping will be correctly specified and not contradictory (eg, I won't enter a command that uses both ">" and "|" for the same program's output).

Things you don't need to worry about:

- the environment.

(I may add clarifications here as questions arise.)

Deliverable requirements

- `mysh.c`, which should contain the implementation of your shell.
- `Makefile`, whose default target should build `mysh` (out of `mysh.c`), It may have as many other targets as you like, to make your life easier), but the default one *must* build that program.
- `README`, which will include a list of authors, a list of known bugs, and a list of resources you consulted in your work (a list of URLs is totally acceptable—if I can see the kinds of references you're using, I can get a better idea of how to tailor the instruction to your needs—just keep the README file open as you work and paste them there when you find something useful).

Style requirements

Your code must exhibit:

- appropriate use of comments;
- appropriate use of stack, heap, global, and static variables;
- appropriate use of constants;
- appropriate decomposition;
- clear organization (eg, grouped header files, struct definitions, function prototypes);
- well-named variables, functions, and other identifiers;
- verification of command-line arguments, if applicable;
- checking and appropriate handling of all return values;
- consistent style (capitalization, indenting, etc).

Submission Process

Your submission files should be in a git repository on basin in this directory:

```
/home/<username>/<secret>/assignment5
```

Email me when you want feedback, with a description of the type of feedback you would like.

Coda

You've seen me use pipes in class a few times, but the real power of them is only realized when you have both knowledge of a sufficiently broad collection of programs to choose from and a problem well-suited to those tools. I've documented some such programs here.

As for potential problems, here is an account of using programs in that collection to process data extremely efficiently:

- https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html

Last modified: 12/03/2022 13:02:34