CSPB 3753

Shell Assignment

Creating Your Own Shell

Your task is to extend the Fork Lab and the Pipe Lab to create a basic shell with a small memory footprint. The shell should operate as follows:

1. Print a customized welcome message to `stdout`.
2. Display a customized command prompt.
3. Fetch a command from `stdin` and store it into a buffer.
4. Execute this command.
5. Report the exit status of this command to `stdout` if not run in background.
6. Repeat the steps 2 through 5 until the user enters `DONE`.

To keep things simple, we will not do any sophisticated parsing of the user's input (e.g., no quotation matching, checks for escape characters, etc..). In particular, you may assume that the user will input data that obeys the following grammatical structure:

```
cmd_1 arg arg ... arg <pipe> cmd_2 arg arg ... arg <pipe> ... <pipe> cmd_n arg arg ... arg <bg>
```

Background Support

Note that `<bg>` is either `&` or empty. If the former, then the command will run in the background (like what `bash` does); otherwise, it runs in the foreground, and we'll wait for the child to finish, reporting its exit status. You may assume that no `arg` has `&` in it.

Piping

You are urged to only add pipe functionality to your shell once you get the rest of the shell working 100%. If you have reached this point, then you should first try to get your shell working with a single pipe. Once you have this special case working, extend your solution to handle any number of pipes. You may assume that no `arg` has a pipe `|` in it.

Dynamic Buffer

Observe that when a command is fetched from the user we do not know in advance how many characters this will be. This means that you cannot define at compile time how large the input buffer will be, even though most of the users commands will typically be no more than 32 characters long. Since our shell aims to keep a low memory profile, you should not initialize the buffer to have a large size, and you must *dynamically* allocate more memory to the buffer should it be required to run the user's command. You may find the `fgetc` function handy for implementing a dynamic buffer.

Error Handling

In your code, there are many places where system errors can arise (errors that are out of your control). In these places, you must always provide some sort of error checking. Typically, if a system call returns something negative, then this indicates an error, but you should check the man pages to be sure. If an error occurs, you should use the `perror` function to print the error to `stderr` as follows:

```
perror("some informative message");
```

Visit its `man` page for more details.

Reaping

Recall that a shell should not create any zombie processes, which will happen when we do not `wait` on our children. However, we do not care about the exit status of our forked process if we are running it in the background, so we'd like to just delete our child's process entry from the process table to prevent it from becoming a zombie. This can be done by having the parent ignore the child's signal `SIGCHLD` using the `signal` system call. Check out the `signal` system call for more details.

Other

1. If you are having trouble getting started, or seeing the big picture, our book has a simple outline of a shell at the end of Chapter 3.
2. When passing strings around in `C`, remember that a string is just a sequence of characters that *ends with the null character*, and be on the lookout for trailing whitespace.
3. Your shell needs to do something clever for when the user wants to change directories using `cd` (hint: `man chdir`). *Think about why one needs to do this.*

4. When you're done, you should make sure that your shell is not leaking any memory. Most of the time (but not always) if you have a matching `free` for every `malloc`, then you're in the clear, but the tool `valgrind` will sniff out any memory leak should you have one. We will check for memory leaks so be sure to run `valgrind` on your solution before you submit.

Submission Format

To submit your assignment, create a zip file (use filename: <your last name>_shell.zip) with all your source code including a make file (no folders!). Your program should compile if I just run `make` with no additional arguments. Submit that zip file as your submission on Moodle.

Grading

The majority of your grade (70%) is based on the *correctness* of your program, and the remaining 30% will be determined by *style*. *If your final submission does not compile, then you cannot earn more than a 30, so be sure that you submit code that compiles.* The correctness of your program will be determined as follows:

- Parsing user input (10 pts),
- Unbounded/dynamic input buffer (10 pts),
- Background support (10 pts),
- Proper use of `fork` + `execv` (5 pts),
- Handling `cd` properly (5 pts).
- Proper handling of errors (5 pts).
- Single pipe functionality (7 pts),
- Multiple pipe functionality (8 pts),
- No zombie processes (10 pts).

The remainder of your grade will be determined as follows:

- Documentation: Your code should be well-commented and give good explanations of the complicated parts of your program. (10 pts)
- Legibility/Formatting: Your style should be consistent and readable (e.g., make good use of whitespace). Also, if you don't follow the submission format instructions, then you will lose some points here. (10 pts)

- Elegance: Subroutines of your solution shouldn't be needlessly complicated (e.g., there should be no unused variables, redundant code, etc). (10 pts)

Partial credit (to be decided) will be awarded if any of the parts above are not 100% correct.