Sean Varie, CS452 Project 1: Simple Shell

This was a very interesting project. It has been a while since I have written C code that is meant to run on an OS (as opposed to bare metal microcontroller programming). It took me a while to get used to memory management again and some of the common design patterns of software writing with C. I enjoyed diving into the system calls and learning how to use a bunch of new operations that I wasn't aware of before, like the history library or the waitpid function or even the fork and exec family of functions, which I was aware existed, but have never actually used before.

The design ideas were mostly laid out for us, at least until task 7. Printing the version was just following the standard format for parsing arguments using getop. The code for user input was given to us. The custom prompt just involved making a few system calls with some error handling and setting a variable, and the built in commands were just a series of if/else statements. The first thing I had to think about in terms of the design of the project was how to handle creating another process to run a command. The instructions gave us a strong hint to use the execvp function so that we didn't have to worry about searching the path for the command to run, but I still needed to create another process, as the exec family of functions replaces the calling process. This meant that I would need to use the fork function before execvp. It seemed most elegant to do the fork itself in main and then defer to functions in lab.c once I had forked. Handling the signals was fairly simple, much of the information was given to us. The only question was where to do the ignoring/defaulting of the signals. It seemed best to me to set them to ignore in sh_init and back to default in sh_destroy, and set them to default in children as necessary. The biggest design decision made was how to handle the background processes, mostly how to keep track of them. I decided to keep them in a single linked list and hold a pointer to the head of the list in a global variable. This made it simple to remove an element when a process completed and to add elements onto the end, increasing their job number by 1 from the previous end of the list. It also made executing the built in jobs command as simple as traversing the single linked list and printing out the information in the proper format for each element of the list (and removing completing jobs as necessary).

I achieved all the functionality laid out in the instructions. The shell can create new processes to handle commands issued which are not built into the shell itself. It will wait until the command is finished to prompt the user for another input, or will background the command and prompt again immediately if indicated with the '&' character at the end of the command. All four built-in commands are implemented. The history command prints out each command that was previously issued on its own line, in chronological order, the first command ran will be at the top, and so on. Some extra formatting for the history command could be desired, but I decided to keep it simple.

No AI was used in the making of this project.

There is a memory leak in the provided test code for the test: test_cmd_parse2. I was instructed to not modify the test code, but if I were to put this code into production, I would fix that memory leak by adding the following code to the bottom of that test:

```
    free(stng);
    cmd_free(actual);
```