Programming Project #1

Building a Shell

Introduction

After completing this assignment you will gain experience in systems programming, learn how to use the system call interface, and get familiar with the programming tools and environment you will be using for the following assignments. You task is to create a working command line shell program. What good is an operating system if you can't tell it what to do? A shell lets you type in commands to run programs.

You should have some experience using the standard Unix shell. Read up a bit on the features and capabilities of your favorite shell. Popular choices of shell are BASH (the Bourne-Again shell) and CSH (the C shell). You can't implement a feature if you don't know how it works.

You are provided files `lex.c` and `shell.c` which contain template code to get you started. Without any modifications, the files parse the command line input and split out each argument. The default "shell" just prints out the arguments and waits for more input. The shell gets input by calling `getline()` which returns an array of pointers to strings. Each string is either a word containing letters, numbers, period, minus, and slash, or a single special character from "`()<>|&;`".

To compile `lex.c` you need to run `flex lex.c` to produce `lex.yy.c`. Then you need to compile `lex.yy.c` and `shell.c` using:

```
cc -c lex.yy.c
cc -c shell.c
```

This produces output files ending in .o. To link them together into a working executable do:

```
cc -o shell shell.o lex.yy.o
```

Make sure you get the default shell with no functionality working first before you start modifying the code.

Requirements

Your shell must support the following features.

1. **An internal command "exit" which terminates the shell.**
   Concepts: shell commands, exiting the shell
   System calls: `exit()`
2. **A command with no arguments**
   Example: `ls`
   Details: Your shell must block until the command completes. If the return value is abnormal, it must print out an error message.
   Concepts: Forking a child process, waiting for children to finish, synchronous execution
   System calls: `fork(), execvp(), exit(), wait()`
3. **A command with arguments**
   Example: `ls -l`
   Details: Argument 0 is the name of the command (e.g. ls in the example)
   Concepts: Command-line arguments
4. **A command (possibly with arguments) executing in the background with &**
   Example: `crunch numbers.txt &`
   Details: For simplicity you may assume the & character is always the last thing on the line. Your shell must execute the command and return without blocking.
   Concepts: Background execution, asynchronous execution, signals, signal handlers, processes
   System calls: `sigset()`
5. **Redirection to a file**
   Example: `ls -l > listing`
   Details: This takes the output of the command and puts it into the named file.
   Concepts: File operations, output redirection
   System calls: `freopen()`
6. **Redirection of input**
   Example: `sort < numbers.txt`
   Details: This takes the named file and uses it as input to the command.
   System calls: `freopen()`
7. **[EXTRA CREDIT] Redirect output of one command into input of another**
   Example: `ls -l | sort`
   Details: This takes the output of ls and makes it the input of the sort command.
   Concepts: Pipes, synchronous operation
   System calls: `pipe()`
8. **[EXTRA CREDIT] Invent your own shell feature.**
   Ideas: something with console control codes (colors!), better ways to keep track of background tasks, log commands somewhere
   Details: Remember to design your feature before coding it (think about how to do it)

You must check and handle all possible return values correctly from every function call you make. Read the documentation for each call you use, make sure you understand how it can fail and what you need to do in that case.

Deliverables

Turn in a `.tgz` file for your project directory into eCommons. It should include:

- Build and run instructions in `README.txt`
- Your design document, named `design.pdf`
- Source files (no generated files!)
- Any extra build files (e.g. `Makefile`)
- Any necessary test files

Your project must be submitted in this format or you will be asked to fix your submission and resubmit (using grace days or getting a penalty). If we try to follow the instructions in your `README.txt` file and it doesn't work, you will be asked to resubmit (using grace days or getting a penalty). We will try your submissions in a fresh MINIX 3.1.8 installation. Don't forget to include any files needed. Also, do not include files that could be generated automatically from source files (each such file included will be penalized).

Coding Style

All code submitted must follow these guidelines. Deviations may be penalized. Unclear or sloppy code will always be penalized. Good coding style makes your code more readable to others and is critical for keeping complex software maintainable.

Comments

- Every file must contain a comment on top that indicates whether you created it, or it was part of MINIX and you modified it. If you created the file, put CREATED in the command and a date. If you modified an existing MINIX file, put CHANGED in the comment with a date.
- Every function must have a comment immediately before it that describes what the function does. It should describe the parameters, return value, intended action, and any assumptions the function makes.
- Code should be well commented. Trickier parts of code need more comments, simple parts need fewer comments. You should end up with about as many lines of comments as you have lines of code.
- If you changed an existing MINIX file, put comments that indicate where the change starts and ends. Put “CHANGE START” and “CHANGE END” in these comments.

Indentation

- Only use spaces, no tabs.
- Indent a fixed number of spaces at every level (e.g. 2 or 4).
- Wrap long lines manually in a visually reasonable way.

Spacing

- Every significant comment should be preceded by a blank line. Do not put a blank line after comments before the code that is being described.
- Put a blank line between chunks of code that do different things.
- Put a blank line between variable declarations and other code.
- Put a blank line after every right brace unless the next line is also a right brace.

Braces

- For function declarations, put the opening brace in column 0.
- For other structures, put the left brace on the same line as the keyword (if, while, etc.). Put the right brace to align with the opening keyword (if, while, etc.)

Identifiers

- Functions and variable names should be all lowercase, with underscores between words.
- Names should be descriptive without being too long (this is an art).
- Single letter names may only be used as trivial loop counters.

Global Variables

- Global variables may not be used unless they are the correct solution to a particular implementation issue.
- Any global variables must be explained in the design document.