She Sells C Shells

CMS 330, Spring 2018

Due Tuesday, March 11 at 11:59 PM

Description

You may work with a partner on this project.

In this project, you'll implement a *shell*, similar to the command line interpreter programs you've used in Cloud9. The shell displays a prompt where the user types commands. After each command is entered, the shell creates a child process to execute the requested program, then prompts for another command once the child process has finished. This project will let you use some of the standard commands for working with processes such as `fork` and `execvp`.

Your program should be called `shell.c`. Include a `Makefile` to automatically build your project.

Shell Design

Building the shell is mostly a matter of identifying the correct C library functions and using them in the correct order. The most important tip is to *build incrementally*. Don't attempt to code the entire shell all at once; develop step-by-step, one piece at a time.

The basic shell runs in a loop:

- Display the command prompt
- Get a line of input from the prompt
- Parse the input line to identify the requested program and any options
- Call `fork` to create a new child process
- Use `execvp` on the child to make it run the requested program with the given options
- Have the parent process wait until the child finishes

Note that you don't need to implement any other programs for this project. When you type a command like `ls` at the shell, the code for `ls` is not executed by the shell itself. Rather, the shell simply runs the `ls` program in `/bin`. This is the case for all of the common terminal commands, with a small number of exceptions described below.

Take a look at the example programs included with this document. They demonstrate how to use `fork`, `execvp`, and `wait`.

You can use `strtok` to tokenize the input. Recall that `strtok` changes the input string, so it can't be applied to statically allocated string variables. Here's a short example program that uses `strtok` to identify tokens separated by any number of spaces or tabs:

```
#include <stdio.h>
#include <string.h>

int main() {
    char s[64] = "This    is \t   a\t\ttest";

    // Identify the first token
   char *token = strtok(s, " \t");

   // Use NULL as the first argument to find subsequent tokens
   while(token) {
      printf( "%s\n", token );
      token = strtok(NULL, " \t");
   }
}
```

You assume that each command line argument will be formed from contiguous non-whitespace characters. Therefore, you do *not* need to deal with complex verbatim string inputs that might contain spaces, as in the following example using `echo`:

```
shell$ echo "Hello, World!"
```

Built-In Commands

There are a few commands that should be executed directly by the shell program. These commands *do not* go through the regular path of calling `fork` and then `execvp`.

If the user types `exit`, your shell should terminate.

If the user types `pwd`, you should print the present working directory. You can do this by calling `getcwd` ("get current working directory") and printing the string that it returns.

You must support the `cd` command, which you can implement using the `chdir` function. `chdir` takes a string as its input, then changes the current working directory of the current process to the directory given in the input string.

Output Redirection

Your shell must support a special feature. If the user types the `>` symbol followed by a file name, the shell must redirect any output from the command into the specified file. For example:

```
shell$ ls -l > list.txt
```

Look at the example in the project directory to see how file descriptors can be used to redirect output.

Strategy

Here's a recommended implementation plan.

- Write basic loop that displays a prompt and reads from the terminal
- Implement the three built-in commands: `exit`, `pwd`, and `cd`
- Use `fork` and `execvp` to execute single commands with no arguments, like `ls`
- Make the parent wait until the command finishes
- Add `strtok` to tokenize the input string
- Make sure that you can run full commands with multiple arguments, like `ls -l -a`
- Add support for redirection

Testing

There is no automated test script for this project. You'll need to think about how to adequately test your shell. One warning: if your shell program terminates, you'll go back to the real `bash` shell. I have seen previous students who thought their shells were running perfectly, but they were actually executing all of their commands inside the regular terminal!

Here is an example sequence of commands you can use to test your shell.

```
shell$ ls
shell$ ls -l -a
shell$ sleep 10
shell$ mkdir temp
shell$ cd temp
shell$ pwd
shell$ ps -u ubuntu > ps.txt
shell$ cat ps.txt
shell$ cd ..
shell$ rm -rf temp
shell$ exit
```