CSCI 4100 – Assignment 5 – Writing a Simple Shell

Learning Outcomes

Write a Linux shell using the POSIX process interface.

Required Reading

Linux - Chapter 5

Instructions

For this programming assignment you are going to implement a simple shell called `turtle` in C. This shell will not be able to implement pipes or redirects, but it will be able to execute any Linux command with any number of command line arguments.

The shell should repeatedly prompt the user for a command then execute that command, unless the user types `exit` in which case the shell should exit sucessfully (using `exit(0)`), or the user enters only whitespace in which case the shell should prompt the user again.

If the user does not type exit or a blank line the shell should break up the line of text the user entered into an array of C-strings.

Here is a sample of what running your shell might look like:

```
[colemann@apmycs2 ~]$ ./turtle
> ls
colemanAssign5.c turtle
> ls -l
total 32
-rw-r--r--  1 colemann  staff  2146 Sep 14 16:04 colemanAssign5.c
-rwxr-xr-x  1 colemann  staff  9076 Sep 14 16:05 turtle
>
> fnord
turtle: command fnord not found.
> exit
[exiting turtle shell]
```

I have provided the following functions to get you started:

- `read_line(line)` - reads a line of text from input and stores it in a string.
- `is_blank(str)` - returns true (1) if a string contains only whitespace, false (0) otherwise.
- `parse_args(cmd, argv)` - takes a command line and breaks it up into a sequence of strings containing the command and the arguments, then stores the strings in the array provided.

I have also defined the following constants using the `#define` preprocessor statement:

- `MAX_LINE` - the maximum number of characters allowed in a single line. Use this as the size when you declare a character array to represent the string to pass to `read_line`.
- `MAX_ARGS` - the maximum number of arguments (including the command) that may be used in a command line. Use this as the size when you declare an array of C-strings to represent the array of arguments to pass to `parse_args`.

To execute a command you must do the following:

- Use the `fork` function to fork off a child process. The `fork` function takes no arguments and returns a value of type `pid_t` that can be treated like an integer value. WARNING: be very careful with using `fork` in a loop. If you are not careful you will fork too many processes for your shell to handle.
- Check to see if the current process is the child or the parent by checking the return value of `fork`. If the return value is 0, the current process is the child process. Otherwise, it is the parent process.
- The parent process should call the `wait` function with a single argument of `NULL` to wait for the child to complete.
- The child process should call the `execvp` function with two arguments:
  - The name of the command being executed (argument 0 of the array.)
  - The entire array of command line arguments.

If `execvp` succeeds, the process will no longer be running the shell code. This means that the next line is executed only when `execvp` fails. If this is the case, display an error message and terminate the child process by calling `exit(1)`.

Displaying Output to the Console

In previous assignments I suggested using `puts` to display output to the console.

```
puts("Hello World!\n");
```

If you want to insert a C-string into your output you can use the `printf` function with the special format code `%s` and provide the string as an extra argument:

```
printf("Hello %s!\n", name);
```

We also used the `fputs` function with the stream `stderr` to write to standard error:

```
fputs(stderr, "Something went wrong.\n");
```

If you want to insert a C-string into your error message you can use `fprintf`:

```
fprintf(stderr, "Something went wrong with %s.\n", str);
```

String Comparison

When you are checking to see if the user has entered the `exit` command it may be tempting to do something like this:

```
if(line == "exit")
  // display message and exit program
```

This will not work because the `==` operator compares the addresses of the strings, not the contents of the strings.

Instead you will need to use the `strcmp` function. This function takes two strings as arguments and returns zero if they are the same:

```
if(strcmp(str1,str2) == 0)
  puts("The strings are the same");
```

Compiling Your Code

Your source file should be called `yourlastnameAssign5.c` except with your actual last name. To compile this code you should use the `gcc` compiler on the Linux server. To create an executable file called `turtle` use the following command:

```
gcc -o turtle yourlastnameAssign5.c
```

What to Hand In

Your source file should have comments at the top listing your name, CSCI 4100, Assignment 5, and a brief explanation of what the program does. Download the source file to your local machine, then upload it to D2L in the dropbox called Assignment 5.