Homework 2: The Shell - Part 1

Due: 11:59 PM, September 16, 2021

This homework *project* will require you to implement the beginnings of a functioning shell. Ultimately, your shell will be able to run and control the operation of programs. Once you have implemented your shell, you will be able to navigate the available file systems, execute commands, pass a variable number of arguments to those commands, and redirect standard input and standard output for those programs.

In this particular assignment, you will begin with the basics which includes creating a shell prompt, parsing user input, and navigating the file system.

The shell prompt

The shell "prompt" serves to give a user an indicator of *where* a user is in relation to the system environment. This is also where the user types in their commands to operate the system. Shell prompts often include information including a person's user name, the working directory, the host name of the computer, and perhaps even the current time. In many shells, this prompt can often be customized to fit a user's preferences.

In the shell *you* will implement, you are to simply list the current working directory followed by a dollar sign ($) which would then be followed by user input. Therefore, a shell waiting to be used might look like the following if your working directory was your home directory (in this case, my home directory).

```
/home/backman $
```

The input a user may provide follows relatively clear formatting guidelines which are depicted as follows:

```
commandName [arg1 arg2 ... argN]
```

Let's break down what each of these parts entails:

- **commandName:** This refers to the name of the command to be executed and is the only part of a user's input that is not optional. [Note: The term "command" is a bit odd here as these can be either built-in shell commands or names of programs/binaries that you wish to execute.] A user may supply the absolute path to this program (the path starting from the root directory) or they may supply the relative path to this program (the path starting from the working/current directory).
- **args:** This list of arguments is optional and will be passed to the command specified by `commandName`. There is no limit to how many arguments will be included here — there may be 0 or many.

Spaces (one or many) are *required* between a command name and the beginning of its list of arguments. Between all of the other arguments in the list of optional arguments there ought to be

spaces as well (1 to many). A lack of spacing between two arguments will result in those arguments being captured as a single argument.

All of the following are **valid examples** of user input:

```
/home $ ls -l /home/backman
/home $ ls -l -t -r -h backman

/ $ ping -c 3 google.com
```

Note that the first example referenced the program `ls` and provides two arguments (`-l` and `/home/backman`). The second example contains 5 arguments and the third example contains 3 arguments.

Executing Commands

At this stage in the development of your shell, you will not be launching arbitrary programs/binaries – that will come next. You will, however, implement a number of shell commands outlined in the following section.

Built-in commands

Aside from executing external applications, you will also have to implement three built-in shell commands. They are:

- `cd` - To change directories
- `pwd` - To print the working directory
- `exit` - To quit your shell

The `cd` command allows your shell to update its working directory to the one specified by the user. This working directory provides the programs that the shell executes with a specific location context – any relative path that application chooses to follow will begin in the current working directory. If a program you execute writes the file `out.txt` (a relative path name) then the file will be written to the working directory last specified by the Unix `chdir` C function. You can read more on this C function by reading its documentation via: `man chdir`.

As users navigate with the `cd` command in your shell, you should invoke `chdir` appropriately to provide the correct location context for any applications that may run. Note that `cd` will only take a single argument: the location of a directory to move to.

Important Note: If a user requests to `cd` to a directory that doesn't exist, you must print an appropriate error to the screen. You can check for these types of errors by noting the return value of the `chdir` function. Again, see the man pages for how to handle this return value.

Another handy function you may use to set the current working directory of the shell when you start is `getcwd`. Look up the documentation on it via: `man getcwd`.

In your program you should maintain a notion of your current working directory. After all, it should be displayed within your shell prompt and you need to keep track of the directory that you are in. Also, you need to be able to print it back to the user on-demand as they call `pwd` (even though they can see it in the prompt anyway).

The `exit` command is an obvious one – it terminates your shell.

Debugging Arguments

There is one last built-in command that I would like you to implement that is not a standard built-in shell command (you wont find it in bash, csh, zsh, etc.). This will be useful to you for debugging and identifying arguments as you perform string processing to isolate the arguments provided by a user.

You must implement a command called `debugargs` that, when called, will isolate and display each of the arguments provided after it in a nicely displayed format. For example, suppose that I type the following into my shell:

```
/home/backman $ debugargs    one 2 3.0     four   FIVE   6    seven   E_I_G_H_T
```

Your program should then print out the command as well as the list of arguments (in which we ignore white-space) in the nice, readable format illustrated below. **Important Note!:** When you print your list of args, you must also print two *extra* args: in the first position, you must repeat the command name and the last position must be the text `(null)`. This work will prepare you for the 2nd half of this project which will require you to construct array containing exactly these elements. Also, you must print brackets around each of your arguments so that you can ensure that they do not contain any extra white-space. Therefore, invoking the previous `debugargs` command would produce the following output:

```
/home/backman $ debugargs    one 2 3.0     four   FIVE   6    seven   E_I_G_H_T
cmd: [debugargs]
arg: [debugargs]
arg: [one]
arg: [2]
arg: [3.0]
arg: [four]
arg: [FIVE]
arg: [6]
arg: [seven]
arg: [E_I_G_H_T]
arg: [(null)]
```

Again, excess spaces should not be captured in the list of arguments. This work will be immensely helpful for the upcoming part 2 of this project in which you pass both this `cmd` string as well as this `argv` "vector" to the `execv` system call to be used in the execution of the binaries. See: `man execv`.

Implementation: function use

As a large purpose of this project is to get you acquainted with operating system abstractions and the fundamentals of C, you will be restricted to using mostly POSIX system calls. Functions accessible for use include those found in the following table:

function name	location	purpose
open	fcntl.h	open/create files to read/write
close	unistd.h	closes open file descriptors
read	unistd.h	reads a quantity of bytes from an open file descriptor
write	unistd.h	writes a quantity of bytes to an open file descriptor
execv	unistd.h	copies program to address space & begins execution
fork	unistd.h	creates a new, cloned, process
chdir	unistd.h	changes the working directory
getcwd	unistd.h	gets the current working directory
wait	sys/wait.h	waits for child process termination
malloc	stdlib.h	allocates heap memory for use
free	stdlib.h	releases allocated memory for re-use
strcpy	string.h	copies a string from a source to destination array
strncpy	string.h	copies a string from a source to destination array
memcpy	string.h	copies data from a source to destination array
strlen	string.h	gets the number of chars in a string excluding terminator
strcat	string.h	concatenates two strings together
strcmp	string.h	compares two strings to see if they are similar
memchr	string.h	identifies an instance of a character in a string
memset	string.h	fills a region of memory with a value
strerror	string.h	provides nice string-output for errno values
	errno.h	provides access to the errno global/static variable

Prohibited functions include: printf, scanf, and strtok.

If you are unfamiliar with any of the functions above and you wish to learn more, you can read their documentation with the `man` command. For instance, type `man strcpy` to learn about how to use the `strcpy` function. Some functions have multiple manual entries if they have multiple connotations for different libraries. For instance, `open` is one of these and calling `man open` leads you to a less-than-helpful man page. For such functions, you can type `whatis` and then specify the function name to see what kinds of man pages are available for that function. Typing `whatis open` will inform you that there are two types and that a man page with identifier (2) is the C function that is applicable to us. Therefore, we can type `man 2 open` to access the appropriate documentation.

You may also read about `errno` in the documentation for the above functions. `errno` is a static and global variable (not a function) that provides additional context when system calls fail. Often times the documentation for a function will say that when a system call fails it will return a non-zero value and that `errno` will be "set accordingly". Using the function `strerror` and passing in the value `errno` in the following manner will produce an error message in a C string that will give more detail regarding the error: `strerror(errno)`. For example, when trying to open a file, you might get an error which informs you that either the file doesn't exist or that you don't have permission to access the file.

Submission: Turning in the assignment

To turn in your assignment, place your work on the CS-Data server (\\`CS-Data`) under the following directory within your student folder:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_2\
```

Make sure your folder structure is identified **exactly** as it is outlined here or you will lose 5 points for not following directions and hindering my scripts which run and test your code.

The only file you *n*eed to have in your folder is the following:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_2\shell.c
```

Again, points will be deducted if your files are not stored in exactly these locations. Make sure that you test the functionality of your shell to ensure that it works in a variety of scenarios.