Homework 3: The Shell - Part 2

Due: 11:59 PM, September 23, 2021

This homework *project* will require you to implement a functioning shell. The shell should operate, similarly to other shells, 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.

The shell prompt

The visual styling of your shell prompt will be unchanged from what you did in the previous assignment. An example below shows what you are familiar with.

```
/home/backman $
```

The input a user may provide will now be slightly different from what you saw previously but it will follow a very strict form. Existing shells used in Linux/Unix today allow for much more lax formatting requirements but we will make things a bit more rigid and straightforward for the purpose of this project. Therefore, a user's input will follow these formatting guidelines:

```
commandName [arg1 arg2 ... argN] [input redirection] [output redirection]
   or
commandName [arg1 arg2 ... argN] [output redirection] [input redirection]
```

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. 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.
- **input redirection:** This is also optional and provides a file that will replace standard input for the application. Input redirection is specified with two parts: 1) the standard input redirection symbol (<) followed by the file name (with an absolute or relative path) to replace standard input with. There may be 0 or many spaces between the symbol and the filename. Notice that input redirection can occur either before or after output redirection.
- **output redirection:** This is also optional and provides a file that the application's standard output will be written to. Output redirection is specified with two parts: 1) the standard output redirection symbol (either > or >>) followed by the file name (with an absolute or relative path) to replace standard ouput with. Notice that there are two forms of output

redirection that can occur (as seen by the two sets of symbols). The single arrow symbol (>) denotes that if the specified file exists, it will be overwritten when written to – that is the file will be deleted and then started anew. The double arrow symbol (>>) denotes that if the specified file eixsts, it will be appended to. Only one of these options can be used when choosing to redirect standard output. There may be 0 or many spaces between the symbol and the filename. Notice that output redirection can occur either before or after input redirection.

Spaces (one or many) are only *required* between a command name and its list of arguments. Between all of the other optional fields in the user input, there may be 0 to many spaces.

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

```
/home $ ls -l /home/backman > /home/backman/output.txt
/home $ ls -l backman >> backman/output.txt
/home/backman $ cat < input.txt > /home/backman/output.txt
/home/backman $ cat< input.txt >> /home/backman/output.txt
/home/backman $ cat   >  /home/backman/output.txt<input.txt
/home/backman $ cat>>/home/backman/output.txt<input.txt
/ $ ping -c 3 google.com>/home/backman/output.txt
```

All of the following are **invalid examples** of user input for the purposes of this assignment (although some work in the Bash shell!):

```
/home $ < in.txt echo hello world
/home $ echo hello > out.txt world
/home $ echo hello world > out1.txt >> out2.txt
/home $ cat < in1.txt < in2.txt
```

You do not need to support any additional operators found in shells such as: (&, &&, ;, |, etc.). The only operators you need to support are the redirection operators explained above.

Executing applications

When applications are executed from the shell, the shell should `fork()` off a process and utilize the `exec` family of function calls to load the application into the child process's address space. Specifically, you will need to use the `execvp` system call to pass in an array of arguments provided by the user. (hint: read `man exec`).

In the man pages for `exec` you will learn exactly how to call the `execvp` function. Pay particular attention to the formatting of the 2nd argument to `execvp`. Again, **read the man pages for `exec`** and pay special attention to the description of the `execvp` function!

Built-in commands

Aside from executing external applications, you must also support all existing functionality that you implemented for the previous assignment. Keep everything that you did previously as we will continue to build upon them. Therefore you should continue to support the following built-in commands:

- `cd` - To change directories
- `pwd` - To print the working directory
- `exit` - To quit your shell
- `debugargs` - To provide handy feedback on your argument parsing

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
execvp	unistd.h	copies program to address space & begins execution
fork	unistd.h	creates a new, cloned, process
chdir	unistd.h	changes the 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. Although, you may definitely use functions such as printf for your own debugging purposes – just make sure that you do not include it in your final submission.

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.

Error Reporting

I would also like you to report errors that occur in your program as they relate to the built-in `cd` function and when executing arbitrary commands. That is, if a user tries to `cd` into a directory that doesn't exist, print that information instead of simply doing nothing. Similarly, if `cd` fails because a user does not have permission to access the directory that they asked for, print a corresponding message instead of simply doing nothing.

In the same way, if a user tries to execute a program that doesn't exist or that they don't have permission to execute, print the corresponding error instead of doing nothing.

You do *not* need to provide error reporting for errors that happen relative to file descriptors that did not open correctly for the purpose of stdin/stdout redirection. Unless you're printing to stderr, that would get a bit complicated as the stdout file descriptor may have already been closed. Therefore, don't worry about such error reporting. Only handle `cd` and program execution.

`Errno` and `strerror(errno)` will be immensely helpful for the purpose of error reporting. Detect when functions fail and then print the corresponding error message accordingly.

Final Tips

Writing Files: When writing to files (for the purpose of output redirection) it will be necessary to use the system call `open()` (see `man 2 open`). Sometimes that file will not exist before-hand so you will be need to "create" that file; the `O_CREAT` flag will enable this. Also, you'll want to set the permissions of these files that you'll create; the `S_IRWXU` mode parameter will be suitable. You can learn more about these parameters and how to set the file in append mode or truncate mode in the man pages. The following invocations may be relevant for you.

```
    open(fileName, O_WRONLY|O_CREAT|O_APPEND, S_IRWXU)

    open(fileName, O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU)
```

Managing failed calls to execvp: In a shell, prior to any call to `execvp` there will be a call to `fork()`. Which means we've effectively got 2 processes running by the time we hit `execvp`. There's an interesting problem here when `execvp` fails. Side note: It's easy for `execvp` to fail – a user may specify a program to run that doesn't exist, for example. The problem is that, when this happens, the child process that was forked off continues to exist. And, this child probably lives within the context of your primary shell loop. Which means that you'll have **two** processes simultaneously facilitating shell functionality! You may observe this if you type "exit" and find yourself still in the shell (you may have terminated one shell but not yet the other – I've seen some need to type "exit" numerous times to kill all of their extra/redundant shell processes). There is a simple solution to all of this – call `exit(1)` immediately after `execvp` so that you can kill any child processes that manage to live past `execvp`.

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_3\
```

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.

After completing all of the exercises in this homework assignment, the only file you *n*eed to have in your folder is the following:

```
\\CS-Data\Students\your_user_name\cmsc432\hw_3\shell.c
```

It is absolutely fine for you to have additional files (a header file that defines commonly used functions, for example). I will copy over your entire folder and compile only your `shell.c` file. If that file includes other header files then they will be successfully brought into your code. Make sure to include in your folder any such additional user-defined header files that you code requires.

Again, points will be deducted if your files are not stored in exactly these locations. Make sure that you test your functions to ensure that they work in a variety of cases (including using test cases that I didn't provide to you in the example portion of each exercise).