Pseudo shell

A shell is another program that allows a user to execute other programs. There are several popular shells available. Some example shells include **bash** (born again shell), **sh** (bourne shell) and **csh** (C shell) to name a few. The shells are in */bin/bash* and */bin/sh* on most Linux systems.

You are asked to write a ***myshell,*** which is a very simple shell that replicates some of the basic features of a traditional shell. You shell should be written in **C** and exists mainly in the file *myshell.c*, but you are free to add additional source code files as long as your Makefile works, and compiles an executable named **myshell**.

The shell should run continuously and display a prompt when waiting for input. The prompt is the '$' symbol, no spaces or extra characters! An example of your shell running is

```
$ls –la /home/osboxes
```

or

```
$gcc –g /home/osboxes/hello.c –o /home/osboxes/hello
```

Your shell will read a line from **stdin** one line at time. An input line is delimitated by a carriage return (the enter key). After reading in the line:

- parse the line into its' command and all supplied arguments. This is essentially tokenizing the input line.
- You are not required to hand any special characters like (', ", `, $, tabs) you will have to handle (;, &, <, >) characters.
- The maximum line length is 2048 characters.

Execute the command. After parsing and building an appropriate command to execute, execute the command. Valid executable commands are executable files or built-in shell commands.

- if the command is an executable file. you will do a fork() and exec(). Note you may have to do some work before you do the fork() or exec(). NOTE: you must search the path to find the executable, not the system.

Built-in Command implementation

Implement exit, cd and history

- cd – change director will be used to change directories. Initially you will only have to support

```
$cd pathToChangeTo
```

- exit – terminate you shell program
- history [-c] [offset] – modeled after the bash history command, but much simpler.
- history (without arguments) displays the last 100 commands the user ran, with an offset next to each command. The offset is the index of the command in the list, and valid values are 0 to 99,

inclusive. 0 is the oldest command. Do not worry about persisting this list to a file; just store it in memory. Once more than 100 commands are executed, remove the oldest entry from the list to make room for the newer commands. Note that history is also a command itself and therefore should also appear in the list of commands. If the user ran invalid commands, those should also appear in the list.

- history -c clears the entire history, removing all entries. For example, running history immediately after history -c should show history as the sole entry in the list.
- history [offset] executes the command in history at the given offset. Print an error message of your choosing if the offset is not valid.

Pipes

Add pipes to your shell. This will allow a sequence of processes to communicate through a pipe. An example was demonstrated in class e.g.

```
$ls –l | wc
```

In this example, you will fork() two processes (ls) and (wc). LS will write data (the directory listing) to the pipe. wc will read the data in the pipe and printout the number of newlines, words and byte count from the “ls –l” command.

You should not assume that there are only one pipe. For example

```
$cat hello.c | nl | more
```

In the above example, **cat** writes the contents of hello.c to **nl,** which adds line numbers, then sends the data to **more**. More displays the text with line numbers on the screen.

Error messages should be printed using exactly one of two string formats. The first format is for errors where errno is set. The second format is for when errno is not set, in which case you may provide any error text message you like on a single line.

```
"error: %s\n", strerror(errno)
```

OR

```
"error: %s\n", "your error message"
```

So for example, you would likely use: fprintf(stderr, "error: %s\n", strerror(errno));.

Check the return values of all functions utilizing system resources. Do not blithely assume all requests for memory will succeed and all writes to a file will occur correctly. Your code should handle errors properly. Many failed function calls should not be fatal to a program.

Typically, a system call will return -1 in the case of an error (malloc will return NULL). If a function call sets the errno variable (see the function's man page to find out if it does), you should use the first error message as described above.