Command Shell

The shell or command-line interpreter is a fundamental user interface to an operating system. Your first project is to write a simple shell called "`myshell`" that has the following properties.

The shell will loop continuously to accept user commands; it will terminate when "`quit`" is entered. The command line prompt must contain the pathname of the current directory.

Internal commands

The shell must support the following internal commands. Internal commands should be handled by the shell itself and should not be handled by using `exec()` to call an external program.

`cd` `[[<pathname>]<directory>]`
Change the current default directory to <directory>. If the <directory> is not present, report the current directory. If the directory does not exist an appropriate error message should be reported. The command should also change the `PWD` environment variable.

`clr`
Clear the screen.

`dir` `[[<pathname>]<directory>]`
List the contents of `<directory>.` If the `<directory>` is not present, list the contents of the current directory. If the directory does not exist an appropriate error message should be reported.

`environ`
List all the environment strings.

`echo` `<string>`
Display `<string>` on the display, followed by a new line.

`help myshell`
Display the user manual as a `man` page. Note that you must use the nroff markup language and standard Unix man page format for this. You will not be able to "install" your page into the man facility, but the help command should display it using the nroff processor (i.e. `nroff -man <yourpage>`). You should not simply format a text file to look like a man page – you must use the nroff markup language. There are many resources on the web about how to write man pages. An on-line article from the CS Department at Harvey Mudd College at <http://www.cs.hmc.edu/qref/writing_man_pages.html> may provide a good starting point.

`pause`
Pause operation of the shell until the enter key is pressed.

`set` `<varname> = <value>`
Sets a shell variable `<varname>` to the value `<value>`. Both `<varname>` and `<value>` may consist of a string of case-sensitive alphanumeric characters [A-Za-z] and [0-9] and each may be up to 32 characters long. Your shell should allow the creation of at least 255 distinct shell variables. Shell variables should be able to be used as part of any `<string>`, `<pathname>`, `<directory>`, or `<value>`. When using a shell variable as part of a `<string>`, `<pathname>`, `<directory>`, or `<value>`, the effect should be that the variable is replaced with its corresponding value.

`quit`
Quit the shell.

Program invocations

All the other command line input is interpreted as program invocation, which should be done by the shell forking and executing the program as its own child processes. The programs should be executed with an environment that contains the entry: `parent =<pathname>/myshell`. Upon finding the executable, the shell will echo the full path from the system root to the directory where the executable was found. If the executable is not found, the shell will issue an informative error message.

Path specifications

When appropriate, the user may include path specifications in commands, as indicated by `<pathname>` in the internal command specifications above, and elsewhere. The shell will accept path specifications that start with "`/`", "`./`" and "`../`".

However, the user should not be required to include path specifications. In a program invocation, when no explicit path is given to an executable, the shell will search for the executable according to the values in the environment variable `PATH`. This value must be retrieved using the UNIX system call `getenv()`.

The shell environment should contain `shell=<pathname>/myshell` where `<pathname>/myshell` is the full path for the shell executable (not a hardwired path back to your directory, but the path from which it was executed).

Other considerations

The shell must take into account the attributes of relevant files. For example, if the command "`/usr/home/me/foo`" is entered and the specified file exists in the specified location, but is not executable, the shell will issue an informative error message.

The shell must be able to take its command line input from a file; i.e. if the shell is invoked with a command line argument:

```
myshell < batchfile
```

then `batchfile` is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. If the shell is invoked without a command line argument it solicits input from the user via a prompt on the display.

The shell must support I/O redirection on either or both `stdin` and `stdout`. That is, the command line

```
programmname arg1 arg2 < inputfile > outputfile
```

Will execute the program `programname` with arguments `arg1` and `arg2`, the `stdin` file stream replaced by `inputfile` and the `stdout` replaced by `outputfile`.

`stdout` redirection should also be possible for the internal commands `dir`, `environ`, `echo` and `help`.

With output redirection, if the redirection token is '>', then the output file is created if it does not exist, and truncated if it does and its write permissions are set. If the redirection token is '>>', then the output file is created if it does not exist, and appended if it does. When an output file is created using redirection, its access permission must at least include read permission for the owner. If redirection targets an existing file whose write permissions are not set, the shell will issue an informative error message.

The shell must support background execution of programs. An ampersand '`&`' at the end of a command line indicates that the shell should return to the command prompt immediately after launching that program.

Changes to shell environment variables should be registered using `setenv()` or `putenv()` so those values will be visible when external program invocations are made. When your shell exits, the environment should be restored to the same state as before the shell was started.

User documentation

You must write a simple user manual describing how to use the shell. The manual should contain enough detail for a UNIX beginner to use it. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. Your manual should be formatted in the standard UNIX man page format and be written using the nroff markup language as described earlier in this document.

For an example of the sort of depth and type of description required, you should take a look at the online manuals for one or more of the common UNIX shells (`csh`, `tcsh`, etc.) Those shells are much more complex than yours will be, so your manual does not need to be so large.

The user manual should not contain build instructions, an included file list, or source code. The purpose of the manual is to be for end-users, not shell developers.

Design and implementation requirements

There are some explicit requirements, in addition to those on the Programming Style page of the course website:

- Your shell must be implemented in ISO-compliant C/C++ code. You may make use of any language-standard types and/or containers you find useful.
- You must decompose your implementation into separate source and header files, in some sensible manner that reflects the logical purpose of the various components of your design.
- You must document your implementation according to the *Programming Standards* page on the course website.
- You must properly allocate and de-allocate memory, as needed.
- If your shell does not implement a specified feature, it should write an appropriate disclaimer when the user attempts to use that feature, something distinguishable from a normal error message resulting from a logically invalid command. Any such omissions should also be documented in the User Manual.

In general, you are expected to apply the design and implementation guidelines and skills covered in your previous computer science courses. You may implement your solution in pure C, or in C++.

Suggestions and assumptions

There are some explicit assumptions you may make:

- No command line will be longer than 100 characters, and no command will be given more than 10 arguments, not counting redirections and '`&`'.
- Each command argument, redirection symbol and '&' will be preceded by at least one blank space.

You may find it helpful to consult the UNIX man pages on `fork()`, exec(), `getenv()`, access(), and `waitpid()`.

You may also find it helpful to look up the C library function `freopen()`. Note that there are many good online C/C++ language references.

Evaluation

Remember that your implement will be tested in the McBryde 124 lab environment (GCC/Linux). No help will be given for any other development environments. It is your responsibility to make sure your implementation works correctly in the lab.

Your implementation will be evaluated for documentation and design, as well as for correctness of results. Note that the evaluation of your project may depend substantially on the quality of your code and documentation.

What to turn in and how

This assignment will be collected on the Curator system. The testing will be done under the McBryde 124 lab environment.

Submit a single gzip'd tar file containing the C/C++ source and header files for your implementation to the Curator system. Submit only the source and header files. Submit nothing else. Be sure that your header files only contain `include` directives for Standard C/C++ and UNIX header files; any other `include` directives will probably cause compilation errors. It must be possible to unpack the file you submit using the following command:

```
tar –zxf <name of your file>
```

The unpacked files will then be compiled, typically using the following command syntax:

```
[g++ | gcc] –o <name we give the executable> [*.cpp | *.c]
```

You may use the termcap library, in which case the following line will be used to compile your code:

```
[g++ | gcc] –o <name we give the executable> [*.cpp | *.c] -ltermcap
```

Instructions, and the appropriate link for submitting to the Curator system will be posted on the course web site. You will be allowed to submit your project up to three times before the due date in order to fix errors you discover before your project is tested. Your latest submission will be the one that is tested.

Pledge

Your program submission must be pledged to conform to the Honor Code requirements for this course. Specifically, you must include the pledge statement provided on the Programming Standards page in one of your submitted files.