Command Shell

The shell or command-line interpreter is the fundamental user interface to an operating system. Your first project is to write a simple shell 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:

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>]
Llist 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 <comment>
Write the string <comment> on the display, followed by a new line.

help
Display the user manual using the more filter.

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

quit
Quit the shell.

Internals commands are handled by the shell itself. This may involve making calls to API functions provided by the underlying operating system. You should not simply invoke features of an underlying shell from which your shell may have been started.

Program invocations

All 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 as described above. 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 "../".

Path specifications are not required. 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 one 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 it is not executable, the shell will issue an informative error message.

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

```
programname arg1 arg2 < inputfile > outputfile
```

will execute the program `programname` with arguments `arg1` and `arg2`, the `stdin` file stream replaced by `inputfile` and the `stdout` file stream 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 to 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. The manual must be named `readme` (no extension) and must display properly if opened in `vi` or `emacs`.

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`, `bash`, etc.). Of course, those shells are much more complex than yours will be, so your manual will not be so large.

The user manual should not contain build instructions, an included file list, or source code. This is a User Manual, not a developer's guide.

Design and implementation requirements

There are some explicit requirements, in addition to those on the *Programming Standards* 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 deallocate 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. There is no requirement that you base your design on an object-oriented analysis of the problem, nor are you discouraged from doing so. 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 redirection 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()`, `waitpid()`, `opendir()`, and related UNIX features cited in those man pages.

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 and that if you have installed the Microsoft Visual C++ documentation that does include a comprehensive language reference.

Here is an example of a C++ style internal command handler:

```
void cdHandler( istringstream& User ) {

   string newDirectory;
   User >> newDirectory;                // get target directory

   if ( newDirectory.length() == 0 ) {  // no target directory, echo current
         cout << Environ.PWD << endl;
   }
   else if ( newDirectory == "./" || newDirectory == "." ) {   // no change
         return;
   }
   else {
      string newPWD = findNewPWD( newDirectory );  // clean up vble
      Environ.PWD = newPWD;                        // set internal vble
      setenv("PWD", newPWD.c_str(), true);         // set system vble
   }
}
```

Evaluation

Shortly before the due date for the project, the TA will post signup sheets inside the McB 124 lab. You will schedule a demo with the TA. At the demo, you will perform a build, and run your program on the demo test data, which I will provide to the TA. The TA will evaluate the correctness of your results. In addition, the TA will evaluate your project for good internal documentation and software engineering practice.

Remember that your implementation will be tested in the McB 124 lab environment. If you use a different development platform, it is entirely 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 Fedora Core 4 using gcc 4.0.1. The Curator will not begin accepting submissions of this assignment before September 10.

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 certainly result in 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]
```

Instructions, and the appropriate link, for submitting to the Curator are given in the *Student Guide* at the Curator website:

http://www.cs.vt.edu/curator/.

You will be allowed to submit your solution up to three times in order to fix errors you discover before your project is tested. Your latest submission will be tested.

Pledge

Each of your program submissions 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.