PROGRAMMING PROJECT ONE

DEVELOPING A 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—`myshell`—that has the following properties:

1. The shell must support the following internal commands:
   - i. `cd <directory>`—Change the current default directory to `<directory>`. If the `<directory>` argument is not present, report the current directory. If the directory does not exist, an appropriate error should be reported. This command should also change the `PWD` environment variable.
   - ii. `clr`—Clear the screen.
   - iii. `dir <directory>`—List the contents of directory `<directory>`.
   - iv. `environ`—List all the environment strings.
   - v. `echo <comment>`—Display `<comment>` on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
   - vi. `help`—Display the user manual using the `more` filter.
   - vii. `pause`—Pause operation of the shell until "`Enter`" is pressed.
   - viii. `quit`—Quit the shell.
   - ix. 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).
2. All other command line input is interpreted as program invocation, which should be done by the shell `fork`ing and `exec`ing the programs as its own child processes. The programs should be executed with an environment that contains the entry: `parent=<pathname>/myshell` where `<pathname>/myshell` is as described in 1.ix above.
3. The shell must be able to take its command line input from a file. That is, 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. Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.
4. The shell must support I/O redirection on either or both *stdin* and/or *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 character is > then the `outputfile` is created if it does not exist, and truncated if it does. If the redirection token is >> then `outputfile` is created if it does not exist, and appended to if it does.

5. The shell must support background execution of programs. An ampersand (`&`) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.
6. The command line prompt must contain the pathname of the current directory.

Note: You can assume all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, `&`) will be delimited from other command line arguments by white space—one or more spaces and/or tabs (see the command line in 4. above).

PROJECT REQUIREMENTS

1. Design a simple command line shell that satisfies the above criteria and implement it on the specified UNIX platform.
2. Write a simple manual describing how to use the shell. The manual should contain enough detail for a beginner to UNIX 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` and must be a simple text document capable of being read by a standard Text Editor.

   For an example of the sort of depth and type of description required, you should have a look at the online manuals for `csh` and `tcsh` (`man csh`, `man tcsh`). These shells obviously have much more functionality than yours and thus, your manuals don't have to be quite so large.

   You should NOT include building instructions, included file lists, or source code—we can find that out from the other files you submit. This should be an Operator's manual not a Developer's manual.
3. The source code **MUST** be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret, and it is in your interests to ensure the person marking your project is able to understand your coding without having to perform mental gymnastics!
4. Details of submission procedures will be supplied well before the deadline.
5. The submission should contain only source code file(s), include file(s), a `makefile` (all lowercase please), and the `readme` file (all lowercase, please). No executable program should be included. The person marking your project will be automatically rebuilding your shell program from the source code provided. If the submitted code does not compile, it cannot be marked!

6. The `makefile` (all lowercase, please) **MUST** generate the binary file `myshell` (all lowercase please). A sample `makefile` would be

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
myshell: myshell.c utility.c myshell.h
    gcc -Wall myshell.c utility.c -o myshell
```

   The program `myshell` is then generated by just typing `make` at the command line prompt.

   Note: The fourth line in the above `makefile` **MUST** begin with a tab.

7. In the instance shown above, the files in the submitted directory would be:

```
makefile
myshell.c
utility.c
myshell.h
readme
```

SUBMISSION

A `makefile` is required. All files in your submission will be copied to the same directory, therefore, do not include any paths in your `makefile`. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not hand in any binary or object code files. All that is required is your source code, a `makefile`, and a `readme` file. Test your project by copying the source code only into an empty directory then compile it by entering the command `make`.

We shall be using a shell script that copies your files to a test directory, deletes any preexisting `myshell`, `*.a`, and/or `*.o` files, performs a `make`, copies a set of test files to the test directory, and then exercises your shell with a standard set of test scripts through *stdin* and command line arguments. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, and so on, then the marking sequence will also stop. In this instance, the only marks that can be awarded will be for the tests completed at that point, and the source code and manual.

REQUIRED DOCUMENTATION

Your source code will be assessed and marked as well as the `readme` manual. Commenting is definitely required in your source code. The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual **MUST** be named `readme` (all lowercase, please, and **NO** `.txt` extension).