SOFE3950U: Operating Systems

Lab 2 - A Simple 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, then report the current directory.
   If the <directory> does not exist, then 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)**
      1. When your shell starts, it should figure out its own executable path and set this path as the value of the shell environment variable.

2. **All other command line input is interpreted as program invocation which should be done by the shell forking and execing 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.**
   i. After reading a command and determining it's not an internal command, your shell should use **fork** to create a child process.
   ii. Before using **exec** to run the command, set the parent environment variable in the child process. This can be done using the **setenv** function.
   iii. Your shell (the parent process) should wait for the child process to finish executing before taking more user input. This is done using the **wait()** or **waitpid()** functions.

3. **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 (modifying the shell program so that it can read and execute commands from a file, in addition to taking commands**

from standard input (like the keyboard). This feature is commonly referred to as batch mode or script mode in many shells.): **myshell batchfile**

i. 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*.** i.e. 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**.

   Input Redirection (**<**):

   programname arg1 arg2 < inputfile
   The **programname** read the **inputfile**.
   wordcount < essay.txt

   Output Redirection (**>** and **>>**):

   '**>**' truncates it (deletes all content) before writing new content.
   '**>>**' appends the new content to the existing content.

   programname arg1 arg2 > outputfile
   The **programname** writes **arg1 arg2** on **outputfile**.
   echo "Hello World" > outputfile

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

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.**
   i. When you run a program from a shell, it usually waits for the program to finish before returning the command prompt. Background execution allows the shell to immediately return the command prompt while the program runs in the background.
   ii. Appending an ampersand (&) to the end of a command instructs the shell to run that command in the background. For example, typing 'some_command &' causes 'some_command' to execute in the background.

iii. To support this in your shell, you need to: Detect if the user's input ends with an ampersand (&).

6. The command line prompt must contain the pathname of the current directory.

Note: you can assume that 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 specifed 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 on-line 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 that 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 lower case please), and the **readme** file (all lower case please). No executable program should be included. The marker will be automatically rebuilding your shell program from the source code provided. If the submitted code does not compile it can not be marked!

6. The **makefile** (all lower case please) **MUST** generate the binary file **myshell** (all lower case please). A sample **makefile** would be:

```
# Joe Citizen, 1234567 - Operating Systems - Lab2
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.

To make this clear: ***do not hand in any binary or object code files***. All that is required are **myshell.c**, **utility.c**, **myshell.h**, **makefile** and **readme** file. Test your project by copying the source code only into an empty directory and 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 pre-existing **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, non-existence of files etc 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

Firstly, 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 execution. The manual **MUST** be named **readme** (all lower

case please, **NO .txt** extension).

Note: Your name, student number, and work group must appear in ALL submitted files.

Marking Criteria (150 marks total)

- Submission of required files only, with name, student number etc on all submitted files (5 marks)
- Warning free compilation and linking of executable with proper name (5 marks)
- Support for both keyboard and batchfile inputs (5 marks)
- Performance of internal commands and aliases (30 marks)
- External command functionality (10 marks)
- I/o redirection (10 marks)
- Background execution (10 marks)
- Readability, suitability & maintainability of source code and makefile (25 marks)
- User manual (50 marks)
    - Description of operation and commands (10 marks)
    - Description of environment concepts (10 marks)
    - Description of i/o redirection (10 marks)
    - Description of background execution (10 marks)
    - Overall layout and display of understanding (10 marks)

Ensure that you address all the points specified in this project documentation.