CS 146 Assignment #5 (Project)

Due dates: Part 1: Thu. 19 May. (**no extension, solution handed out on that date!!!**);
Part 2: Thursday of Week 10..

Write a simple Unix command-line shell called "nsh". It will support, at the very least: shell scripts; the running of arbitrary commands with arguments; the "cd" command; at least one pipe; and I/O redirection (including append). Beyond that, there is considerable choice as to what you decide to implement. **The assignment is out of 30 marks, but bonus marks are possible**.

MINIMAL SHELL (30 points)

You are to write a shell that is similar to a subset of the Bourne Shell. The syntax for everything that is implemented will be identical to the Bourne Shell. So, for example, it needs to handle the following command lines:

```
? who | fgrep -i .edu > foo
? sort < foo | uniq -c >> bar
? ^D
$
```

The prompt is '? ', ie., a question mark followed by a space; nsh should exit when it encounters EOF (which is ^D on the keyboard). In addition, it must support shell scripting so that if precisely one argument is given, then that argument is interpreted as an nsh shell script and the commands in it are run, without printing a prompt. If you implement this first, it will facilitate easy testing of your nsh program as you develop it, because you can write test scripts and compare the output of "nsh foo" with the output of "sh foo", since nsh is supposed to act just like the Bourne shell. You do not need to perform syntax checking on the command line; incorrect command lines are allowed to crash your program without loss of marks on the assignment (I'm being easy on you because this assignment is hard enough as it is).

Note there are spaces around every token; this makes it easier for you. You'll need to look (at least) at the following manual pages: fork(2), exec(2), pipe(2), open(2), creat(2), close(2), dup(2), dup2(3c). Note that you must be careful with files: you may be working with both Unix file descriptors (because those are the only things pipe(2) and dup(2) work with), and ANSI C (FILE *) pointers (because it will make some of the other stuff easier). You can, if you want, choose to work only with Unix file descriptors. For example, in the simple version of the shell above, the only thing you ever need to print is the prompt, so you don't really need the functionality of printf(3c). You are free to use versions of exec(2) that search the PATH for you (eg., execlp, execvp); you don't need to search it yourself.

PART 1: The Parser (15 marks. Due Thu 19 May, when a solution will be distributed)

Since this will be quite a large project, you will split it into manageable chunks and put each chunk in a separately compiled module, and then use a Makefile. However, the Makefile is not due until Part 2. In part 1 you will write the "parser". A parser's job is to break up the line into its constituent logical parts. That means it needs to figure out if there is any input/output redirection (the < and > characters), how many pipes there are on the command line (which is one less than the number of commands between pipes---eg., "who | wc" is two commands separated by one pipe), and what the command line arguments are to each command. You should create a **struct** which will hold a full command line with all this information, and put its definition into a file called **parse.h**. The file **parse.h** also contains a prototype for the function **Parse**, which will be defined in the file **parse.c**. Then create **parse.c**, which contains the code for the actual parser. The file **main.c** will contain a loop which (for now) just reads a line, pasess that line to the function **Parse**. **Parse** will populate the **struct** with the logical info on the command line, and then your **main** program will print out the parsed version of the command line. For example, given "cat –v <infile | grep foo | wc > outfile", the Part 1 version of your main program should print:

3: <'infile' 'cat' '-v' | 'grep' 'foo' | 'wc' > 'outfile'

The '3' represents the number of commands, and each "word" is printed with quotes around it. Input/output redirection should only be printed if they are present. So for example "who | wc" should give:

2: 'who' | 'wc'

A correct executable of the parser is in ~wayne/pub/cs146/nsh-parser. You can use it to see what yours should output for Part 1 of the assignment.

PART 2: Executing Commands (15 or more marks, due Thu of Week 10.)
In Part 2 you will execute the commands using system calls. As mentioned above, the shell MUST implement shell scripting to facilitate easy testing of your shell, and your shell should be able to handle commands with at least one pipe. Handling multiple pipes (ie., more than 2 commands piped together) is worth extra marks, as described below.

EXTENSIONS TO THE SHELL --- do not start until minimal shell is working, or you won't get the points.
The following extensions are possible, **BUT ONLY IF YOUR MINIMAL SHELL IS WORKING**. If you'd like to try something not listed, talk to me. You may choose to do up to 10 marks worth of extensions, and your maximum mark will be 40---ie., you can earn up to 10 "bonus" points on this assignment.

[1 mark] Implement background execution using '&'.
[1 mark] Comments. Anything after a '#' to the end-of-line is ignored. Note that this automatically allows you to begin your nsh scripts with "#!/home/you/bin/nsh", giving automatic execution of executable nsh scripts.
[1 mark] Allow meta-characters to appear without spaces around them, so "who|wc" works, rather than needing spaces like "who | wc". (You don't get the marks if you end up using my parser from Part 1, of course.)
[1 mark] Implement the shell escape character using "\", so "echo \|" will output a single "|".
[1 mark] Implement the "exit" command. With no arguments, it exits with the exit status of the most recently executed command, or it accepts a single integer argument which will be nsh's exit status.
[2 marks] Implement single quoting so arguments can have spaces inside them. For 1 more mark, allow the argument to have single quotes inside, so 'now\'s the time' becomes a single argument "now's the time".
[3 marks] Handle more than one pipe. Eg., "who | grep foobar | uniq –c | sort –nr | less"
[4 marks] Implement environment variables. You don't need to distinguish between shell and environment variables (ie, no "export" command). Just use putenv(3c) and getenv(3c). Duplicate the Bourne Shell's use of them. This can be tricky, you should carefully experiment with how the Bourne Shell handles variables. Note especially what happens with

```
$ foo='hello world'    # there's a space in there
$ prargs $foo          # how many arguments is that?
$ bar=$foo             # does bar become 'hello' or 'hello world'?
```

[4 marks] Implement back-quotes, ie. an arbitrary nsh command line can be put in back-quotes, and its output is substituted onto the current command line. Remember to parse the spaces after substitution.
[4 marks] Implement double-quotes, so variables and back-quotes can be put inside them. (Of course this only makes sense if you've implemented at least one of variables or back-quotes.) It must support allowing double-quotes inside the double quotes using "\"".
[6 marks] Use lex(1) and yacc(1) to do the parsing for you. If you do this, it may actually make some of the other stuff easier, but there's alot to learn to do it. You may use GNU's flex(1) and bison(1) instead.

TESTING
You must hand in test scripts that test and demonstrate the functionality of each of the basic and extended features you implement. Your grade will depend strongly on how well your test scripts actually test the features you claim to have implemented; the grader will not spend time testing your shell to see if it does what you say it does; you must demonstrate this yourself with your test scripts. We will, of course, run your test scripts to ensure they do what you say they do.

DOCUMENTATION
Provide a cover page that lists the extensions that you implemented, so I know what extra marks to give you. Document any known bugs you have found in your shell. If you document a bug, you may get partial marks for a feature that is partially implemented; undocumented bugs will significantly lower your mark.

WHAT TO HAND IN
Hand in paper copies of the cover page, your manual page, all your source files, all your test scripts and their output, and your Makefile. Also electronically sumbit everything (except the cover page) using the **submit** command as in previous assignments.