Project 1 – Writing your own unix shell in C

You will write your very own shell in this assignment, similar to bash or tcsh, but with only a few of the features. Your shell will be able to run programs and support file redirection, pipes, and the use of & to run programs in the background. You will complete this assignment with your partner.

Deliverables: You will hand in just one file, simpleShell.c. Be sure BOTH PARTNERS names appear in comments at the top of the file!

Shell specifications

- The input to the shell is a sequence of lines. A line may not be longer than 100 characters.
- Each line consists of tokens. Tokens are separated by one or more spaces. A line may contain as many tokens as can fit into 100 characters.
- There are two kinds of tokens: operators and words. The only operators are <, >, &, and | .
- Words consist of the characters A–Z, a–z, 0–9, dash, dot, forward slash, and underscore. If a word in a line of input to the shell contains any character not in this set, then the shell should print an error message and continue processing with the next line of input.
- The only legal input to the shell, other than lines consisting of valid tokens, is the end-of-file, which should call the shell to exit with a return of 0.
- Lines of input are divided into token groups. Each token group will result in the shell forking a new process and then executing a program.
- Every token group must begin with a command. This may be followed by any number of arguments. For example the following are all legal token groups
  - `ls`
  - `ls -l`
  - `echo Hello World`
  - `java HW/Proj1/Test.java`

- A token group may include one or more file redirections. A file redirection consists of one of the operators < or > followed by a single word that is the filename. For example the following are all legal token groups
  - `python test.py > output.txt`
    - runs program test.py but writes any output to output.txt instead of terminal
  - `python test.py < input.txt`
    - runs program test.py but reads input from input.txt instead of keyboard
  - `python test.py < input.txt > output.txt`
    - runs test.py with both input and output redirected to/from the given files

  The > operator indicates an output redirection, e.g. STDOUT of the command should be redirected to the given file. This file should be created if it does not exist, and the shell should report an error if the file cannot be created. Similarly, the < operator indicates an input redirection, e.g. STDIN of the associated command should be redirected from the given file. The shell should report an error if this file cannot be opened for reading.
  - You should use the dup2 system call to achieve redirection, see dup2example.c for a simple example of how to do this.

- Token groups are separated by pipe operators. In other words, each valid line of shell input must begin and end with a valid token group, and the only place pipe operators are allowed is in between token groups.

- a pipe redirects the output of the preceding token group to become the input of the following token group, for example `ls | tail -1` would run `tail -1` on the output of ls, resulting in only the name of the last file being displayed. Try it!
- A token group cannot be followed by a | if it contains output redirection, and similarly a token group that follows a | cannot contain input redirection.
- there can be multiple pipes on a line, for example `ls | tail -1 | python test.py` would run the program test.py with its input not coming from the keyboard but instead the output of `tail -1` as explained above
- you will use the pipe system call to achieve this, see the program pipeexample.c for a simple example of using a pipe.

- The token & may appear only as the last token on a line of input, and indicates that the shell should not wait for its forked() processes to finish before attempting to read the next line of input. If there is no &, the shell should wait for all forked() processes to finish before continuing.
- Lines of shell input that violate any of the parsing rules should cause the shell to print an error message and then move on to the next line of input.
- The program execexample.c provides a function to read a line of input and split it into a NULL terminated array of tokens. It also shows how to use execvp to execute a simple command given this array of tokens.

Getting Started

Be sure you understand this assignment before starting to write code! Especially spend some time looking at the example files included in the assignment folder and understanding how they work. The execexample.c file has a function "readLineOfWords" that parses the user input and puts it into an array of separate tokens, you do not need to do anything more fancy than this, just use it as is!

Here is a rough outline of steps you might take to complete this project:

1. Basic shell – use the example programs forktest.c and execexample.c to create a basic shell that reads a line of input, forks a child process to exec the given command, and waits for the child process to finish before looping to read the next line of input.
2. Add some processing of the array of tokens to determine which of the operators may be present (<, >, |, &)
3. Handle waiting vs. not waiting based on the &, and make sure you remove the & from the array before calling exec.
4. To support I/O redirection, modify the child process created by fork() by adding some code to open the input and output files specified on the command line. This should be done using the open() system call. Next, use the dup2() system call to replace the standard input or standard output streams with the appropriate file that was just opened. Finally, call execvp() to run the program after removing the redirect tokens from the array.
5. Pipes are a little trickier: you will need to call a fork/exec for each command on the line. Then you should use the pipe() system call to create a pair of pipe file descriptors before calling fork(). After the fork both processes will have access to both sides of the pipe. The reading process should immediately close the write file descriptor, and the writing process should immediately close the read file descriptor. At this point each process uses dup2() to copy the remaining pipe descriptor over STDIN or STDOUT as appropriate.
6. When you are done, bask in the glory of a working shell! You should now have a good operational understanding of the user-mode side of some of the most important UNIX system calls.

7. Code Size: My finished shell is ~300 lines of C code, including plenty of debugging code, blank lines, and comments. Your shell should not be a lot larger than this (in other words, if it gets a lot larger then you're probably doing something wrong).

Use the example programs, man pages, and other documentation on system calls for help, and of course ask questions!

One important note: this is a classic OS assignment and full solutions are easy to find online. Try to avoid the temptation to look at these at all, even if you don't intend to copy but just want to check how to do something, it will be difficult to not do it the same way as the example you happen to see instead of figuring it out for yourself. You will learn so much more by working through it yourself!

Good luck and have fun!