Design Document: Assignment 1 Shell

Alexander Sung
cruzID: acsung

Goals

The goal of this assignment is to create a fully functional simple shell that will perform a set of actions the shell does without terminating unexpectedly. The shell will parse input commands and will execute those commands using system calls. Additionally, the shell will also be able to handle I/O redirection.

Guidelines

One of the files provided is `shell.l`, which is used to parse a line of text into an array of character strings, where each character matches a single token from the shell's point of view (important to note that if there are quotes in the input to the shell, some tokens may include blanks). The file can be compiled using the `flex` command, which produces a file named `lex.yy.c`, which must be compiled and linked to the shell code `.c` files in order to run the program. The linking step can be done using the command:

```
cc -o myshell myshell.o lex.yy.o -lfl
```

Be sure to include `-lfl` ***after*** all of the other files that are to be compiled. The shell program must be built using a `Makefile`; while keeping in mind that we are using FreeBSD `make` rather than GNU `make`.

The Makefile will create separate object files created from compiling the required files: `myshell.c lex.yy.c`. Once the object file targets are built, we link the files created, as well as the object file created from compiling file `shell.l` using the `flex` command. When the object files we need are created, we can link the files together with the above command, giving us an executable file `myshell`, that can be run and will start the shell code of `myshell.c`. I've also included extra Makefile targets, `clean` and `spotless`, which can be run by typing `make <target>` which will remove any unnecessary object and executable files from the directory.

Design Details

When the shell is running, a prompt ("`shell> `") should be printed to the console and the program will read in a line of input that will be processed and executed using the `get_args()` method.

Single programs

The shell program should be able to run a single program followed by zero or more arguments (up to a maximum number of tokens the `flex` file can handle which is 1024 tokens), such as the command `ls -l foo`. A single line command consists of a number of tokens, where each string returned by the `get_args()` function is a token. The end of the command is denoted by a *metacharacter* (`exit` and other shell commands are not metacharacters).

Functions:

`void handleChild(char** args)`

- Input: argument list, where the first argument must be the command
- Output: void
- This function handles the child process after calling `fork()`. It takes the argument list passed in from the `get_args()` call, and passes the argument list to `execvp()` to execute the input command. When `execvp()` completes without error, the child process terminates. At this point, we have completed executing the given command, we can let the parent process loop back to the top of the while loop, and prompt the user for another command to execute. If there was an error with executing the command, `execvp()` will return an error code -1. In this case, we print the associated error message to the standard error, and we exit the child process cleanly using `exit(0)`.

`void handleParent()`

- Input: NULL
- Output: void
- This function handles the parent process after calling `fork()`. The parent process calls this function, which will call the system call `wait()`, which will wait for the child process to terminate before the parent process can proceed to going back to the top of the shell interactive while loop. This ensures that the result is `execvp()` is completed before prompting the user for another command to execute.

Input and output redirection

The shell program should be able to handle input redirection (using <) and output redirection (using >). Depending on the files provided for redirection, I will use `open(), dup(),` and `close()` to handle file input/output redirection.

- Example: `sort -nr < scores` uses the file contents of `score` and feeds it to the standard input of the `sort` function.
- Example: `sort -nr < scores > sorted` uses the file contents of `scores` and returns the output of the `sort` function, writing the standard output to the `sorted` file, rather than to the console.

The shell must be able to handle single `>&`, which will redirect standard output as well as standard error to the provided output file. A single > (also of form `<&`) will **replace** the contents of the given output file. A double >> (also of form `<<&`) **appends** the output to the contents of the given file.

I have created multiple flags to indicate which types of redirection commands we receive in the input command. I first parse the command, argument by argument, using the predefined argument list, `args`. If any of the arguments indicate an input redirection symbol, I open a file descriptor for reading only for the appropriate file, then continue parsing the arguments. If any of the arguments are one of the three output redirection symbols, I open an output file descriptor for the given filename, and depending on which form of redirection symbol is used, the file descriptor will be opened with certain permissions, such as `O_WRONLY | O_APPEND` or `O_RDWR`.

Multiple programs not connected
The program will be able to handle multiple single-line input commands so long as they are separated by semicolons (;). To do this, I will parse the single-line input command for the semicolon (;) character, and separate each of the independent commands to be run in series, ensuring that one command finishes before the other starts.

Multiple programs connected by pipes
The shell should also be able to handle commands connected via **pipes**. We will parse the input command for this special character, and we will handle the piping process using a function for that purpose. The pipe uses the output of the first command, and inputs that output to the second command, via the `pipe()` system call. Both inputs and outputs of pipes may be redirected to a provided file.

- Example: `du -sk | sort -nr | head -10 > big` : the output of `du -sk` is sent to the input of `sort -nr` whose output is then the input of `head -10` whose output will be redirected to the file `big`

If pipe symbol includes ampersand (`|&`), then standard output and standard error is redirected to the provided output file.

Internal Shell Commands

- `cd` : change the current directory to the provided directory using `chdir()` syscall. Handle errors if the directory provided is a file. This should be handled before forking processes. Use an if else statement to either perform `cd` or handle commands through `execvp()`.
- `exit` : exits the shell gracefully, using the `exit()` library call.