Assignment 6: Shell, pt. 2

Due: Thursday, March 3, 10pm

Starter code: The starter code for this assignment is your basic shell from Assignment 5. Take the first few days to improve the basic shell functionality if some tests weren't passing. Continue developing in your A5 repository.

Submission: This is a **pair assignment**, but you can work alone, if you so choose.

Submit the contents of your repository via Gradescope. See Deliverables below for what to submit. If you are working with a partner, do not forget to include their name with the submission.

In this assignment, we will expand on the basic shell you wrote as part of Assignment 5. The basic shell functionality will form part of this assignment.

You are asked to implement builtin commands, as well as the following 4 operators:

- Sequencing, e.g., `echo one; echo two`
- Input redirection, e.g., `sort < foo.txt`
- Output redirection, e.g., `sort foo.txt > output.txt`
- Pipes, e.g., `sort foo.txt | uniq`

Note that these operators can be combined. Follow the implementation strategy suggested below. This will give you the relative priorities of the operators.

Task 1: Built-in Commands

In addition to running programs, shells also usually provide a variety of *built-in commands*. Let's implement some.

The shell should support at least the following built-in commands, in addition to `exit` from Assignment 5:

`cd` (change directory) This command should change the current working directory of the shell.
Tip: You can check what the current working directory is using the `pwd` command (not a built-in).

`source` Execute a script.
Takes a filename as an argument and processes each line of the file as a command, including built-ins. In other words, each line should be processed as if it was entered by the user at the prompt.

`prev` Prints the previous command line and executes it again.

`help` Explains all the built-in commands available in your shell

Task 2: Sequencing Using ;

The behavior of ; is to execute the command on the left-hand side of the operator, and once it completes, execute the command on the right-hand side.

Task 3: Input Redirection <

A command may be followed by < and a file name. The command shall be run with the contents of the file replacing the standard input.

Task 4: Output Redirection >

A command may be followed by > and a file name. The command shall be run as normal, but the standard output should be captured in the given file. If the file exists, its original contents should be deleted ("truncated") before output is written to it. If it does not exist, it should be created automatically.

Task 5: Pipe |

The pipe operator | runs the command on the left hand side and the command on the right-hand side simultaneously and the standard output of the LHS command is redirected to the standard input of the RHS command.

Deliverables

All tasks Implement the extensions in `shell.c`.

Include any `.c` and `.h` files your implementation depends on and commit everything to your repository. Do not include any executables, `.o` files, or other binary, temporary, or hidden files.

Grading

About 25% of the grade for this assignment will be for the functionality of the basic shell from Assignment 5. This means that if you've completed that assignment, you get 25% for free on this one. If you didn't get it quite right, completing it will give you 25% of the grade for this assignment right away. You won't get a lesser grade for the functionality than you did in the previous assignment.

Shell Implementation Strategy

Here's a set of "rough and ready" guidelines for tackling the extra shell functionality. Note that each subcommand might contain other operators as well. You might want to implement sequencing or redirection first.

1. Sequencing: `command1; command2`
    a) Split the token list on semicolon.
    b) Fork child A & execute `command1` (recursively).
    c) In parent: wait for child A to finish.
    d) Fork child B & execute `command2` (recursively).
    e) In parent: wait for child B to finish.
2. Pipe: `command1 | command 2`
    a) Fork child A.
    b) In child A: create a pipe.
    c) In child A: fork child B.
    d) In child B: hook pipe to `stdout`, close other side.
    e) In child B: execute `command1`.
    f) In child A: hook pipe to `stdin`, close other side.
    g) In child A: execute `command2`.
    h) In child A: wait for child B.
    i) In parent: wait for child A.
3. Redirection: `command <OP> file`
    a) Fork a child.
    b) In child: replace the appropriate file descriptor to accomplish the redirect.
    c) In child: execute `command` (recursively).
    d) In parent: wait for child to finish.

All this functionality needs to be implemented by you, using system calls. Relying on the default shell does not fulfill the requirements.

Hints & Tips

- `man` is your friend. Check out `fork`, `open`, `close`, `read`, `write`, `dup`, `pipe`, `exec`, ...
- **Write your own (unit) tests.** Doing so will save you time in the long run, especially in conjunction with the debugger. In office hours, the instructors or the TAs may ask you to show how you tested code that fails.
- Follow good coding practices. Make sure your function prototypes (signatures) are correct and always provide purpose statements. Add comments where appropriate to document your thinking, although strive to write self-documenting code.
- Split code into short functions. Avoid producing "spaghetti code". A mutli-branch `if-else if-else` or a multi-case `switch` should be the only reason to go beyond 40-50 lines per function. Even so, the body of each branch/case should be at most 3-5 lines long.
- Use `valgrind` with `--leak-check=full` to check you are managing memory properly.
- Avoid printing extra lines (empty or non-empty) beyond what is required above. This goes both for the tokenizer and the shell. Extra output will most likely confuse our tests and give false negatives.

Examples

Here are some examples you can use to test the shell functionality.

- The line

  ```
  echo one; echo two
  ```

  should print

  ```
  one
  two
  ```

- Running

  ```
  echo -e "1\n2\n3\n4\n5" > numbers.txt; cat numbers.txt
  ```

  should print

  ```
  1
  2
  3
  4
  5
  ```

  and result in a file called `numbers.txt` being created in the current directory.

- Running

  ```
  sort -nr < numbers.txt
  ```

after the above, should print

```
5
4
3
2
1
```

- Running

  ```
  shuf -i 1-10 | sort -n | tail -5
  ```

  should print

  ```
  6
  7
  8
  9
  10
  ```

Going Further

You might consider some of the following optional features in your shell to challenge yourself (there is no extra credit for this):

1. Switching processes between foreground and background (`fg` and `bg` commands).
2. Grouping command expressions. E.g.:

   ```
   ( cat prologue.txt ; ( cat names.txt | sort ) ; cat epilogue.txt ) | nl
   ```

Changelog

02/25/2022

- Strategy: Switch pipe and redirection to reflect the desired operator priority.
- Going Further: Fix list rendering.