Project 1: Shell

Due: Friday, February 21, 10pm

Starter code: See Project 1 on Canvas for the Github Classroom link.

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.

Do not use `login.khoury.northeastern.edu` to work on this assignment. Instead use Github Codespaces or your local Docker container.

The first project of this class, is to write a Shell. This project is more involved than the previous assignments and requires more planning as well as programming. You are asked to develop a moderately complex piece of C code from scratch. Start early with planning, experimenting, and prototyping.

Part 1: Tokenizer & Basic Shell

The first part of this assignment is to get a basic working shell. For this, we will first need to develop a *tokenizer*, that is, a piece of code that helps us split up an input line into meaningful *tokens*. The second task is to develop a basic shell that uses this tokenizer to process input from the user.

Task 1.1: Shell Tokenizer

Before we can execute commands (or combination of them) we need to be able to process a command line and split it into chunks (lexical units), called *tokens*. The input of a tokenizer is a string and the output is a list of tokens. Our shell will use the tokens described in the table below. The tokens `(`, `)`, `<`, `>`, `;`, `|`, and the whitespace characters (space `' '`, tab `'\t'`) are *special*. Whitespace is not a token, but might separate tokens.

Token(s)	Description / Meaning
`( )`	Parentheses allow grouping shell expressions
`<`	Input redirection
`>`	Output redirection
`;`	Sequencing
`|`	Pipe
`"hello < (world;"`	Quotes suspend the meaning of special characters (spaces, parentheses, ...)

Token(s)	Description / Meaning
`ls`	Word (a sequence of non-special characters)

Your first task is to write a function that takes a string (i.e., `char *` in C) as an argument and returns a list (array, linked list, etc.) of tokens. The maximum input string length can be explicitly bounded, but needs to be at least 255 characters.

You also need to provide a demo driver, `tokenize.c` that will showcase your function. The driver should read a single line from standard input and print out all the tokens in the line, one token per line. For example:

```
$ echo 'this < is > a demo "This is a sentence" ; "some ( special > chars"' | ./tokenize
this
<
is
>
a
demo
This is a sentence
;
some ( special > chars
```

In this example, we print the example string to standard output, but immediately *pipe* that output into the input of the `tokenize` program. We will implement piping in our own shell in the next assignment.

Whitespace that is not in quotes should not be included in any token.

To help you get started writing a tokenizer, see the example included with the starter code.

What we are implementing here is a Deterministic Finite-state Automaton (DFA), which is a recognizer for Reguar languages. While not necessary to complete this assignment, you might want to read up on those to get a deeper understanding if you are interested.

Task 1.2: Basic Shell

The next step is to implement basic shell functionality running a single user-specified command at a time. The shell should display a prompt, read the command and its arguments, and execute the command. This should be performed in a loop, until the user requests to exit the shell. The commands can have 0 or more arguments and these arguments may be enclosed in double quotes `"`, in which case the enclosed string is treated as a single argument.

Example interaction:

```
$ ./shell
Welcome to mini-shell.
shell $ whoami
ferd
shell $ ls -aF
./        .git/     shell*    shell.o   tokens.h  vect.c    vect.o
../       Makefile  shell.c   tokens.c  tokens.o  vect.h
shell $ echo this should be printed
this should be printed
shell $ echo this is; echo a new line
this is
a new line
shell $ exit
Bye bye.
```

Here are the requirements for the basic shell

1. After starting, the shell should print a welcome message: `Welcome to mini-shell.`
2. You must have the following prompt: `shell $` in front of each command line that is entered.
3. The maximum size of a single line shall be at least 255 characters. Specify this number as a (global) constant.
4. Each command can have 0 or more arguments.
5. Any string enclosed in double quotes (`"`) shall be treated as a single argument, regardless of whether it contains spaces or special characters.
6. When you launch a new child process from your shell, the child process should run in the foreground by default until it is completed. The prompt should be printed again and the shell should wait for the next line of input.
7. If the user enters the command `exit`, the shell should print out `Bye bye.` and exit.
8. If the user presses *Ctrl-D* (End-Of-File, aka EOF), the shell should exit in the same manner as above. *Implement this early on. Our tests rely on your shell being able to terminate on EOF.*
9. If a command is not found, your shell should print out an error message, `[command]: command not found` (replacing "`[command]`" with the actual command name), and resume execution.

   For example:

   ```
   shell $ dfg
   dfg: command not found
   shell $
   ```

10. System commands should not need a full path specification to run in the shell.

    For example, issuing `ls` should work the same way it works in BASH and run the `ls` executable that might be stored in `/bin`, `/usr/bin`, or elsewhere in the *system path*.

Part 2: Advanced Shell Features

Part 2 expands on the basic shell from Part 1. You are asked to implement 4 builtin commands, as well as the following 3 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 2.1: 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 Part 1:

`cd` (change directory) This command should change the current working directory of the shell to the path specified as the argument.
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, without becomming the new command line.
You do not have to support combining `prev` with other commands on a command line.

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

Task 2.2: 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.

For example:

~~~
```
shell $ echo Boston; echo San Francisco; echo Dallas
Boston
San Francisco
Dallas
shell $ dfg; uptime
~~~

~~~
dfg: command not found
20:04:40 up 44 days,  6:14, 60 users,  load average: 2.05, 1.93, 1.70
shell $
```
~~~

Task 2.3: 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 2.4: 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. You do not need to implement output redirection for built-ins.

Task 2.5: 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. You do not have to support piping the output of built-ins.

Deliverables

Parts 1 and 2. Implement the shell 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; or any extra directories.

All the functionality needs to be implemented by you, using system calls. Writing code that relies on the default shell in any form does not fulfill the requirements.

The Grammar of Shell

A grammar for a language specifies all the *valid* examples of expressions (or sentences) in that language. Our shell has the following grammar. This should help decide what is a valid command line, but also to help you structure your code. If you took *Fundies*, it might help to think of a grammar as a collection of related (inductive) union definitions.

Shell Grammar

- A **command line** is one or more *pipe commands* separated by a semicolon: `;`.
- A **pipe command** is one or more *redirections* separated by a pipe: `|`.
- A **redirection** is a *simple command*, optionally followed by a `>` or a `<` and a file name.
- A **simple command** is either a built-in command or a program name followed by zero or more arguments.

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.

   Note, that you may have success processing a sequence of commands using an ordinary loop too.

2. Pipe: `command1 | command2`

   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.

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
```

Using the Provided Makefile

As before, we provide you with a Makefile for convenience. It contains the following targets:

- `make all` – compile everything
- `make tokenize` – compile the tokenizer demo
- `make tokenize-tests` – run a few tests against the tokenizer
- `make shell` – compile the shell
- `make shell-tests` – run a few tests against the shell
- `make test` – compile and run all the tests
- `make clean` – perform a minimal clean-up of the source tree

Hints & Tips

- The starter code contains an example of a tokenizer. A good start is to try to modify the example to recognize the tokens of a shell.
- A very basic tokenizer can also be written using the function `strtok`, which provides a somewhat different approach. However, trying to handle string tokens using this approach might prove tricky.
- Use the function `fgets` or `getline` to get a line from `stdin`. Pay attention to the maximum number of characters you are able to read. Avoid `gets`.
- Figure out how `fgets/getline` lets you know when the shell receives an end-of-file.
- If you get unexpected results (such as extra output) when implementing redirections, try using `fflush(stdout)` *before* closing/replacing standard output. Look at `fflush`'s manpage.
- Use the provided unit tests as a minimum sanity check for your implementation. Especially before the autograder becomes available.
- **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. Pick meaningful names for your functions and variables. The larger the scope of the variable, the expressive the variable name should be.
- Think about and design your program in a top-down manner and split code into short functions. Leverage your knowledge of program design from previous classes.
- Avoid producing "spaghetti code". A mutli-branch **`if-else if-else`** or a multi-case **`switch`** should be the only reason to go beyond 30-40 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.
- A string vector implementation might be useful.
- 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.
- `man` is your friend. Check out `fork`, `open`, `close`, `read`, `write`, `dup`, `pipe`, `exec`, ...