Assignment #1: Building an OS Shell

Due: October 3, 2024 at 23:59

1. Infrastructure Description

Welcome to the first OS assignment where we will build an OS Shell!

This is the first of a series of three assignments that build upon each other. By the end of the semester, you will have a running simulation of a simple Operating System and you will get a good idea of how the basic OS modules work.

You will use the **C programming language** in your implementation, since most of the practical operating systems kernels are written in the C/C++ (e.g., including Windows, Linux, MacOS, and many others).

The assignment is presented from a Linux point of view using a server like Mimi. Our grading infrastructure will pull your code from GitLab automatically and will run the unit tests for this assignment every day on the Mimi server. The autograder may not run for the first few days as we take all the administrative steps to set it up.

Starting shortly after the team registration deadline, you will receive a daily report stating whether your code passes the unit tests. ***Please make sure your assignment runs on our server, as this is the reference machine we use for grading.*** To get your code picked up by our grading infrastructure we will rely on GitLab. **It is mandatory to use GitLab for this coursework.** For more information about GitLab and the grading infrastructure, please refer to the tutorial posted on MyCourses as part of Lab 1.

For local development, and quicker testing turn-around, you can use the SOCS server `mimi.cs.mcgill.ca`, which you can reach remotely using `ssh` or `putty`. You need a SOCS account to access the mimi servers, and you need to either be on campus or connected to the McGill VPN. If you do not have a SOCS account (e.g., you might not have one if you are an ECSE student) please follow the instructions here to obtain one.

To get started, fork the following repository, which contains the starter code and the testcases for this assignment.

https://gitlab.cs.mcgill.ca/mkopin/comp310-ecse427-coursework-f24

IMPORTANT: If you have already forked your **repository before the release date of the assignment**, please make sure that your version of the starter code is up-to-date (i.e., sync your fork with the upstream repository *mkopin/comp310-ecse427-coursework-f24*) before starting to work on the code.

1.1 Starter files description:

We provided you with a simple shell to start with, which you will enhance in this assignment. Take a moment to get familiar with the code.

Compiling your starter shell

- Use the following command to compile: `make mysh`
- Re-compiling your shell after making modifications: `make clean; make mysh`
- *Note: The starter code compiles and runs on Mimi and on our server. If you'd like to run the code in your own Linux virtual machine, you may need to install build essentials to be able to compile C code:* `sudo apt-get install build-essential`

Your code should be able to compile in any environment with a C compiler so long as you do not hardcode assumptions about the machine into your code. For example, write `sizeof(int)` rather than assuming that this value is 4.

Running your starter shell

- **Interactive mode:** From the command line prompt type: `./mysh`
- **Batch mode:** You can also use input files to run your shell. To use an input file, from the command line prompt type: `./mysh < testfile.txt`

Starter shell interface. The starter shell supports the following commands:

COMMAND	DESCRIPTION
`help`	*Displays all the commands*
`quit`	*Exits / terminates the shell with "Bye!"*
`set VAR STRING`	*Assigns a value to shell memory*
`print VAR`	*Displays the STRING assigned to VAR*
`run SCRIPT.TXT`	*Executes the file SCRIPT.TXT*

More details on command behavior:

- The commands are case sensitive.
- If the user inputs an unsupported command the shell displays *"Unknown command"*.
- `set VAR STRING` first checks to see if VAR already exists. If it does exist, STRING overwrites the previous value assigned to VAR. If VAR does not exist, then a new entry is added to the shell memory where the variable name is VAR and the contents of the variable is STRING. For now, each value assigned to a variable is a single alphanumeric token (i.e., no special characters, no spaces, etc.). For example:
  - `set x 10` creates a new variable x and assigns to it the string 10.
  - `set name Bob` creates a new variable called name with string value Bob.
  - `set x Mary`, replaced the value 10 with Mary.

- `print VAR` first checks to see if VAR exists. If it does not exist, then it displays the error *"Variable does not exist"*. If VAR does exist, then it displays the STRING. For example: `print x` from the above example will display Mary.

- `run SCRIPT.TXT` assumes that a text file exists with the provided file name, *in the current directory*. It opens that text file and then sends each line one at a time to the interpreter. The

interpreter treats each line of text as a command. At the end of the script, the file is closed, and the command line prompt is displayed once more. While the script executes the command line prompt is not displayed. If an error occurs while executing the script due a command syntax error, then the error is displayed, and the script continues executing.

1.2 Your tasks:

Your task is to add the following functionality to the starter shell.

1.2.1. Enhance the `set` command.

The `set` command in your starter shell assumes that the STRING variable is a single alphanumeric token. Extend the `set` command to support values of at most 5 alphanumeric tokens. If the command has more than 5 tokens *for the value*, the shell will not set the new value and will return the error: *"Bad command: Too many tokens".*

Assumptions:

- You can assume the tokens are separated by a single space.
- You can also assume that the length limit for the tokens is < 100 characters each. We will not test with larger values.
- Only STRING can have multiple alphanumeric tokens. VAR remains unchanged (i.e., a single alphanumeric token).

Example execution (interactive mode):

```
$ set x 10
$ print x
10
$ set x 20 bob alice toto xyz
$ print x
20 bob alice toto xyz
$ set x 12345 20 bob alice toto xyz
Bad command: Too many tokens
$ print x
20 bob alice toto xyz
```

1.2.2. Add the `echo` command.

The `echo` command is used for displaying strings which are passed as arguments on the command line. This simple version of `echo` only takes **one token string** as input. The token can be:

- **An alphanumeric string.** In this case, `echo` simply displays the string on a new line and then returns the command prompt to the user.

Example execution (interactive mode):

```
$ echo mary
mary
```

```
$
```

- **An alphanumeric string preceded by $.** In this case, `echo` checks the shell memory for a variable that has the name of the alphanumeric string following the $ symbol.
    - If the variable is found, `echo` displays the value associated to that variable, similar to the `print` command and then returns the command prompt to the user.
    - If the variable is not found, `echo` displays an empty line and then returns the command prompt to the user.

Example execution (interactive mode):

```
$ echo $mary
                    // blank line
$
$ set mary 123
$ echo $mary
123
$
```

Assumptions:

- You can assume that the token string is <100 characters.

1.2.3. Enhance batch mode execution.

1. Batch mode execution in your starter shell enters an infinite loop if the last command in the input file is not `quit.` Fix this issue so the shell does not enter an infinite loop. Instead, the shell should display the `mysh` command prompt (i.e., entering interactive mode) after running all the instructions in the input file.
2. Batch mode execution in your starter shell displays $ for every line of command in the batch mode. Change the batch execution so that $ is only displayed in the interactive mode.

1.2.4. Add the ls, mkdir, touch, and cd commands.

Add three new commands to your shell:

1. `my_ls` lists all the files present in the *current directory*.

    - If the current directory contains other directories, `my_ls` displays only the name (not the contents) of the directory.
    - Each file or directory name needs to be displayed on a separate line.
    - The file/directory names are shown in alphabetical order, similar to the sort command in Linux:
        - Names starting with a number will appear before lines starting with a letter.
        - Names starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
        - Names starting with an uppercase letter will appear before lines starting with the same letter in lowercase.

2. `my_mkdir dirname` creates a new directory with the name `dirname` in the *current directory*.

- `dirname` can be (1) an alphanumeric string, or (2) an alphanumeric string preceded by $.
- If `dirname` is an alphanumeric string, `my_mkdir` creates a directory with the given name.
- If `dirname` is an alphanumeric string preceded by $, `my_mkdir` checks the shell memory for a variable that has the name of the alphanumeric string following the $ symbol.
  - If the variable exists in the shell memory and contains a single alphanumeric token, `my_mkdir` creates a directory using the value associated to that variable as the directory name.
  - If the variable is not found or the variable contains something other than a single alphanumeric token, `my_mkdir` displays *"Bad command: my_mkdir"* and then returns the command prompt to the user.

3. `my_touch filename` creates a new empty file inside the current directory. `filename` is an alphanumeric string.

4. `my_cd dirname` changes current directory to directory `dirname`, inside the current directory. If `dirname` does not exist inside the current directory, `my_cd` displays *"Bad command: my_cd"* and stays inside the current directory. `dirname` should be an alphanumeric string, you do not need to consider the case where `dirname` is a shell variable.

Assumptions:

- You can assume that file/directory names are <100 characters.

1.2.5. One-liners.

The starter shell only supports a single command per line. This is not the case for regular shells where multiple commands can be chained. Your task is to implement a simple chaining of instructions, where the shell can take as input multiple commands separated by semicolons (the ; symbol).

Assumptions:

- The instructions separated by semicolons are executed one after the other.
- The total length of the combined instructions does not exceed 1000 characters.
- There will be at most 10 chained instructions.
- Semicolon is the only accepted separator.

Example execution (interactive mode):

```
$ set x abc; set y 123; print y; print x
123
abc
$
```

2. TESTCASES

We provide you with 10 testcases and expected outputs for your code in the starter code repository. Please run the testcases to ensure your code runs as expected, and make sure you get similar results in the automatic tests.

IMPORTANT: The grading infrastructure will use batch mode when testing your code, so make sure that your program produces the expected outputs when testcases run in batch mode. You can assume that the grading infrastructure will run one test at a time in batch mode, and that there is a fresh recompilation between two testcases.

3. WHAT TO HAND IN

The assignment is **due on October 3, 2024 at 23:59, no extensions.**

Your final grade will be determined by running the code in the GitLab repository that is crawled by our grading infrastructure. We will use the most recent commit that happened before the deadline, on the main branch of your fork.

In addition to the code, please include a README mentioning the author name(s) and McGill ID(s), any comments the author(s) would like the TA to see, and mention whether the code uses the starter code provided by the OS team or not.

The project must compile on our server by running `make clean; make mysh`. You should confirm this when the autograder runs **before** the deadline.

The project must run in batch mode, i.e. `./mysh < testfile.txt`

Note: You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations. Submissions are automatically checked via plagiarism detection tools.

4. HOW IT WILL BE GRADED

Your program must compile and run on our server to be graded. If the code does not compile/run using the commands in Section 3, in our grading infrastructure, you will receive **0 points** for the entire assignment. If you think your code is correct and there is an issue with the grading infrastructure, contact rahma.nouaji@mail.mcgill.ca.

Your assignment is graded out of 10 points. You were provided 10 testcases, with expected outputs. If your code matches the expected output, you will receive 1 point for each testcase. You will receive 0 points for each testcase where your output does not match the expected output. Formatting issues such as tabs instead of spaces, new lines, etc. **will not be penalized.** The TA will look at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you implemented the requirement as requested. Specifically:

- **Hardcoded solutions will receive 0 points for the hardcoded testcase**, even if the output is correct.
- **Programming expectations.** Your code needs to have a consistent and reasonable programming style. **If not, your TA may remove up to 3 points, as they see fit.** If you need a reference, the Google C++ Style is applicable to C. The starter code style does not quite match it, however.
- **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.

5. EXTRA CHALLENGE

Add the command `exec` which uses the "fork-exec" pattern discussed in class to invoke other commands. That is, it forks the shell and calls one of the flavors of exec (see the man page) to execute the given command. You should use a flavor of exec that lets you pass the rest of the user input as command-line arguments to the command.

Example:

```
$ exec cat shell.h
#define MAX_USER_INPUT 1000
int parseInput(char inp[]);
$
```

There are no additional points for this, but it is good practice. The '`exec`' word being expected at the start of the line is just to ensure that our testcases work properly, in particular that the Bad Command error still appears when it should. After submitting, you can extend your shell into a proper shell by removing this expectation and attempting to fork-exec *any* command name that isn't built into the shell. You could also try adding more chaining operators like `&&` and `||` and you can try adding file redirection or pipes. The world is your oyster!