# Assignment #1: The Operating System Shell

Due: January 29, 2021 at 23:55 on myCourses

## 1. Preliminary Notes

- The Week 1 and Week 2 in-class exercise sessions, as well as the C Labs, will provide some important background for this assignment.
- The question is presented from a Linux point of view using the computer science server `mimi.cs.mcgill.ca`, which you can reach remotely using `ssh` or `putty` from your laptop (explained in the Week 1 in-class exercise session). We suggest you do this assignment directly from `mimi.cs.mcgill.ca` since these are the computers the TA will be using.
- **You must write this assignment in the C Programming language.**

## 2. Assignment Question: Building an OS Shell

### 2.1 Your source files:

Your application must be built from three C files:

- **shell.c** : This is the primary programming file. It contains the main() and parse() functions.
- **interpreter.c** : The file interpreter.c contains the interpreter() function. Each command the interpreter() function accepts has a corresponding function that implements the command's functionality. Give the function the same name as the command.
- **shellmemory.c** : The file shellmemory.c contains the *private data structures and public functions* that implement the shell memory (The Week 2 Monday/Wednesday exercise session will also be helpful here).

These three source files must be built using modular programming techniques that we discussed in the in-class exercises.

### 2.2 Compiling your shell:

Compile your application using `gcc` with the name **`mysh`**. To do modular programming, you must compile your application in the following manner:

```
gcc -c shell.c interpreter.c shellmemory.c
gcc -o mysh shell.o interpreter.o shellmemory.o
```

### 2.3 Running your shell:

From the command line prompt type: `./mysh`

### 2.4 What `mysh` displays to the user:

```
Welcome to the <your name goes here> shell!
Version 1.0 Created January 2020
$
```

The above dollar sign is the prompt. The cursor flashes beside the prompt waiting for the user's input. From this prompt the user will type in their command and the shell will display the result from that command on the screen (or an error message), and then the dollar sign is displayed again prompting the next command. The user stays in this interface until they ask to quit.

The command line **syntax** for your shell is: **COMMAND ARGUMENTS**.

Each command line command is a single line of text (a string) separated by spaces and terminated by a carriage return. Some commands are a single word, while other commands contain multiple words. Processes each command in a way that is similar to how it was presented in class. You do not need to follow the class slides exactly.

**2.5 Shell Interface:**

The shell needs to support 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* |

- The commands are **case sensitive.**
- There are no other commands (for now).
- If the user inputs an **unsupported command** the shell displays *"Unknown command"*.

**More notes on command behavior:**

- `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 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.

  **Important:** Implement the shell memory as an array of struct, not as a linked list.
  Struct MEM { char *var; char *value; };

- `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. It opens that text file and then sends each line one at a time to the interpreter (as seen in class). The interpreter treats each line of text as a command. Each line affects the shell and the UI. 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. For example: `run test.txt` will begin

by opening the file test.txt. If that fails, then an error message is displayed: "Script not found". If the file is opened, then each line of the file is interpreted. At the end, the file is closed, and the command line prompt is displayed. If an error occurs while executing the script due a command syntax error, then the error is displayed and the script stops executing.

## 2.6 Testing your shell:

The TAs will use *and modify* the provided text file to test your shell. You can also use this file to test your shell or you can test it the old fashion way by typing input from the keyboard. To use the provided text file, you will need to run your program from the OS command line as follows:

```
$ ./mysh < testfile.txt
```

Each line from testfile.txt will be used as input for each prompt your program displays to the user. Instead of typing the input from the keyboard the program will take the next line from the file testfile.txt as the keyboard input.

**Make sure your program works in the above way.**

## 2.7 System calls:

The heart of any operating system is the set of **system calls** that it can handle, through the Kernel API. In this assignment, you likely used various system calls through **libc, the standard C library,** that wraps the kernel API, among other functionalities.

**Important:** Even though libc makes a system call look like function call, it is not a function call. It is a user–kernel transition, from one program (user) to another (kernel). This is much more expensive.

In this exercise, you will measure the time difference between library calls that stay in user-mode and library calls that use system calls to go to kernel mode. Create a **SYSTEMCALLS.TXT** file where you report the following:

- In your code, identify 2 libc library calls that do not use system calls, and 2 libc library calls that use system calls. For the latter two, what system calls are being used? *Hint: Chapter 1.6 in the course textbook will be useful to answer this question.*

- Using the code snippet below, time the 4 library calls you selected in the previous point and report your measurements. Don't forget to remove the code snippet after you're done.

```
#include <time.h>

clock_t start, end;
long cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((long) (end - start));
printf("%s %ld\n", "CPU cycles elapsed:", cpu_time_used);
```

# 3. WHAT TO HAND IN

Your assignment has a **due date on January 29, 2021 at 23:55** plus two late days. If you choose to submit your assignment during the late days, then your grade will be reduced by -5% per day. Submit your assignment to the Assignment 1 submission box in myCourses. You need to submit the following:

- A README.TXT file stating any special instructions you think the TA needs to know to run your program.
- Your version of TESTFILE.TXT. This will be you telling the TA that you know for sure that your program can at least do the following. The TA will run your program with this file and they will also run it with their own version.
- Submit shell.c, interpreter.c and shellmemory.c (you may want to create .h files, if so, please hand those in as well).
- Submit the executable (compiled on the appropriate machine).
- The SYSTEMCALLS.TXT file where you report the measurements from Section 2.7.

*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 assignment is graded out of **20 points** and it will follow this rubric:

- The student is responsible to provide a working solution for every requirement.
- The TA grades each requirement proportionally. This means, if the requirement is only 40% correct (for instance), then the student receives only 40% of the points assigned to that requirement.
- **Your program must run to be graded.** If it does not run, then the student receives **zero** for the entire assignment. If your program only runs partially or sometimes, you should still hand it in. You will receive partial points.
- The TA looks at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you (A) implemented the requirement as requested, and (B) to check if the submission was copied.

  **Mark breakdown:**
  - **1 point** - Source file names. The TA must verify that the student created the three source files shell.c, interpreter.c and shellmemory.c as the only source files in the application. In addition the source files must be populated with at least what was specified in the assignment description for each file. The student is permitted to supply additional helper functions as they see fit, as long as it does not hinder the assignment requirements.
  - **2 points** - Modular programming. If the student wrote their application using modular programming techniques as described in the in-class exercise sessions (or seen from another course) then they receive these points.
  - **1 point** - Executable named mysh. The students compiled program must be named mysh.
  - **2 points** - Shell UI as specified. See the assignment specification for the opening UI screen layout, the error messages, and the general flow of the application as described. For

example, prompts are displayed continually until the user exits the application using the quit command.

- **1 point** - The help command. Displays all the command and description as seen in the assignment description.
- **1 point** - The quit command. Terminates the application. Prints “Bye!” and returns the user to their OS.
- **3 points** - The set command. Creates or overwrites a variable with the new string.
- **2 points** - The print command. Displays the string or the error message.
- **4 points** - The run command. Executes a text file of valid or invalid commands.
- **1 point** - The SYSTEMCALLS.TXT file (0.25 points per each correctly identified and measured libc library call).
- **2 points** - Program can be tested with the TESTFILE.TXT file.