Assignment #1: Building an OS Shell

Due: Oct 3, 2025 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 (or use key authentication). 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/operating-systems-f25

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/operating-systems-w25**) before starting to work on the code. You should also do this whenever we announce that there has been an important starter code or test case update.

1.1 Starter files description:

We provided you with a simple shell to start with, which you will enhance in this assignment. Once you sync your fork with ours, the starter code will be in `A1/starter-code`. Whether or not you want to use the starter code, run `mkdir -p project/src` to create the src folder where you will write your code. **You must place your work in the project/src folder, or our autograder will not be able to find it.** If you want to use the starter code, now run

```
cp A1/starter-code/* project/src/
cp A1/starter-code/.indent.pro project/src/
```

This will copy the starter code, Makefile, and style configuration into your new src folder. The first command *might* copy the style configuration, but since the name starts with a dot it is safest to copy it explicitly.

Navigate to the src folder with cd and take a moment to get familiar with the code.

- Use the following command to compile: `make mysh`
- Re-compiling your shell after making modifications: `make clean; make mysh`
- Running the automatic code formatter: `make style`
    - This command will leave behind lots of files with names like "`shell.c~`". These are backups of your code before it was formatted. We have configured the git repository to ignore these backup files, but you can also safely delete them with `rm *~`. **Do not run this command without the ~ at the end** -- it will delete all of your code!
- *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 # or appropriate package manager
```

Your code will be able to compile in any linux 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. However, you can assume that the code is running on Mimi machines, and in particular that you have access to standard Linux/Unix header files.

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*
`source 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.

- `source 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. 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.

1.2.2. 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 terminate 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.
   This step is extremely important – if you do not complete this step, you will fail every test.

1.2.3. 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.

- Additionally, my_ls should include the special entries "." and ".." in the output, which represent the current directory and the parent directory, respectively.
- Hint: investigate the `dirent.h` header file.

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.

1.2.4. 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. (Spaces are still ignored, of course.)

Example execution (interactive mode):

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

1.2.5. Implementing the 'run' Command with Fork-Exec-Wait.

Add the command `run` which uses the "fork-exec-wait" 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 flavour of exec that lets you pass the rest of the user input as command-line arguments to the command.

You will need to read the documentation for exec and wait. You can get the man pages from your terminal with `man 2 wait` and `man 3 exec`, or you can look it up online.

Example:

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

The '`run`' command is only needed 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 the `run` command 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 `||`, or you can try adding file redirection or pipes. The world is your oyster! (If you follow any of these suggestions, we recommend implementing them on a separate branch so that you don't accidentally break your submission.)

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.

To run a test, navigate to `A1/test-cases`. Then run `../../project/src/mysh < [testfile].txt`. Alternatively, you can move or copy the mysh executable to the test directory to shorten the path. Use `diff` to compare the output with the expected output. **We highly recommend writing a bash or python script to help you run your tests automatically.** We are giving you all the test cases and will not grant requests for any extra autograder runs. Writing testing scripts is part of C programming!

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. Regrade requests of the form "I never tested my code in batch mode, but it works in interactive mode" will be denied.

3. WHAT TO HAND IN

The assignment is **due on Oct 3, 2025 at 23:59.** You have 4 late days that you can use across the 4 assignments however you choose. Information on the form you will use for this will follow separately. Beware that you **must** submit the form before the due date – a late submission should be a conscious, planned decision.

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 replace the README at the top level of the repository. Your README should include any additional 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 TA Mansi Dhanania at mansi.dhanania@mcgill.ca.

We will use **your Makefile**. You may change your Makefile however you wish to fit your project. If you do not use the starter code, or if you add additional source files, you will have to modify the Makefile to correctly compile your source files. You should also fix the style target so that it correctly styles your code. Failure to do so will result in the autograder being unable to run your code, and TAs will not spend time reviewing code that does not run. In other words, if the autograder cannot run your code, you will get a 0. As soon as the autograder becomes available, you should ensure that your project setup is correct – and fix any problems ASAP.

Your assignment is graded out of 100 points. Up to 80 of these points are awarded for passing test cases. The other 20 are awarded by a TA performing code review.

Test Cases: You were provided 10 test cases, with expected outputs. If your code matches the expected output, you will receive 8 points for each testcase. You will receive 0 points for each test case where your output does not match the expected output. When comparing output, we ignore differences in capitalization and whitespace. Your output must have the same words, in the same order, as the expected output.

Any test cases for which you have hardcoded the output will receive 0 points. "Hardcoding" means you've copied the expected output directly to your program output, or otherwise obtained the correct output without implementing the assignment requirements yourself. Note that this means that you cannot use the `system` function that you saw in 206, nor other functions with similar behavior like `popen`. Helper functions in the C standard library, such as those in `<dirent.h>`, are allowed.

Code Review: If your code runs at all, regardless of how many points it scores (even 0!), a TA will perform code review. This entails looking at your code, spending some time trying to understand it, and

then evaluating it for how straightforward it was to understand and for systems design and structure. **The goal of code review is to get you high-quality, actionable feedback on your programming.**

- **Your code needs to follow a consistent and reasonable programming style**. To ensure this, a configuration file for the `indent` tool has been provided, and it is installed on all the mimi machines. A Makefile target (`make style`) is also included to format your code automatically using `indent`. You are **not required** to use `indent` specifically. If you'd rather use a different code formatter, or even take your code style into your own hands, you are free to do so. You can also change the indent configuration (in the hidden file `.indent.pro`) if you'd prefer different options. As long as you are **consistent and reasonable** in your style, we'll be happy.
- **Your code must have useful, meaningful comments.** The TA will be trying to understand your code, and they didn't write it! Part of writing systems software is being able to write an expansive program that *other people can understand*. Whenever your code is doing something non-trivial, you should probably have a comment explaining why.
  Personally, I feel that you can never have too many comments, especially when working on a complex or tightly-integrated part of a system. I would suggest that you have a comment above each function definition explaining what it does, how to call it correctly, and briefly how it works. Prototypes in header files should get a copy of that comment, usually without the "how it works" part. I'd also suggest that there should be a comment on or near any line of code that takes you more than a couple of minutes to figure out how to get right. If it took you a few minutes to disentangle the complexity of *writing* it, it's going to be much harder for a reader to understand the decisions that you had to make to reach your final version.
  On the other hand, useless comments like "increment x" on the line "x++;" hamper comprehensibility, as they take time to read but do not contribute anything. Avoid these.
  A solid understanding of how many comments are necessary to make your code understandable can only be developed with time, and by working on large-scale projects. Therefore, for A1, we will assign lower weight to this component than on later assignments. If there aren't enough comments in your code for A1, your feedback will tell you so.
- **Your software system should be reasonably organized and structured.** A helper function that is used in only one place should be near the function that uses it. A function that's part of an API, e.g. the `shellmemory`, should be in the file implementing that API. If you need to design a new API, such as for a new data structure like a queue or PCB, that API should get its own header/C file pair. Code related to parsing the input and running the shell's outer interactive loop probably belongs in `shell`. Code implementing the behavior of a shell command probably belongs in `interpreter`. In general, don't be afraid to add new files to the project. Having more header files with APIs open on the side, and trusting that their APIs are implemented correctly, is a low mental burden. Having more C code open on the side, and having to read the implementations to see how they work, is a high mental burden. We want low burdens 😊.
- **You cannot hardcode test cases.** See the warning under the 'Test cases' section. If the TA sees hardcoding, or the use of functions like `system` and `popen`, we will deduct all of the points from the affected test cases.
- **Your code must be written in C.** If the TA goes to review your code, and finds a project written in a language other than C (even if you've correctly set up a Makefile for that language….) you will get a zero. **C++ is not C.**