UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS3010A
Operating Systems

Shell Project

Semester 2, 2022

Instructors

COMS3010A Lecturer:
Branden Ingram branden.ingram@wits.ac.za

COMS3010A Tutors:
William Hill 2115261
Andrew Boyley 2090244
Oluwatimileyin Obagbuwa 2134111
Sheslin Naidoo 2094701
Derrin Naidoo 2127039
Phindulo Makhado 1832463
Mohammed Gathoo 2089236
Sedzani Ramathaga 2083519
Ryan Alexander 1827474
Talion Naidoo 1448771
Jonathan Nunes 2087190
Phoenix Krinsky 2233063
Christopher Walley 1846871
Alice Govender 1847313

1 Description

In this assignment, you will implement a command line interpreter (CLI) or, as it is more commonly known, a shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably **bash**. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

2 Program Specifications

2.1 Basic Shell - witsshell

Your basic shell, called **witsshell**, is basically an interactive loop: it repeatedly prints a prompt **witsshell**> (note there should be a space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types **exit**. The name of your final executable should be **witsshell**.

The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument way:

```
prompt> ./witsshell
witsshell>
```

At this point, **witsshell** is running, and ready to accept commands. Type away!

The mode above is called interactive mode, and allows the user to type commands directly. The shell also supports a batch mode, which instead reads input from a batch file and executes commands from therein. Here is how you run the shell with a batch file named **batch.txt**:

```
prompt> ./witsshell batch.txt
```

One difference between batch and interactive modes: in interactive mode, a prompt is printed (**witsshell**>). In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception are built-in commands, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types **ls -la /tmp**, your shell should run the program /bin/ls with the given arguments **-la** and **/tmp** (how does the shell know to run **/bin/ls**? It's something called the shell path; more on this below).

3 Structure

3.1 Basic

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command **exit**, at which point it exits. That's it!

For reading lines of input, you should use the **getline()** function. This allows you to obtain arbitrarily long input lines with ease. Your first objective should be run a loop repeatedly reading in input from the user until the user types **exit**.

Generally, the shell will be run in interactive mode, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support batch mode, in which the shell is given an input file of commands; in this case, the shell should not read user input (from **stdin**) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call **exit(0)** and exit gracefully. In interactive mode the EOF is when you press Ctrl-D.

To parse the input line into constituent pieces, you might want to use **strsep()**. Read the man page (carefully) for more details. Separating an input command into useable pieces should be your next step. In other words if you were to input the following command:

```
witsshell> echo help
```

you should be able to break it down into its components i.e. "echo" and "help" and then store that in some structure, either an array or list for example. I recommend you putting time into creating useful functions and or data-structures to help handle the processing of strings as it will help you throughout.

To execute commands, like **echo** look into **fork()**, **exec()**, and **wait()/waitpid()**. See the man pages for these functions, and also read the relevant lecture notes for a brief overview.

You will note that there are a variety of commands in the **exec** family; for this project, you must use **execv**. You should not use the **system()** library function call to run a command. Remember that if **execv()** is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

3.2 Paths

In our example above, the user typed **echo** but the shell knew to execute the program **/bin/echo**. How does your shell know this?

It turns out that the user must specify a path variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the search path of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

Important: Note that the shell itself does not implement **ls** or other commands (except built-ins). All it does is find those executables in one of the directories specified by path and create a new process to run them.

To check if a particular file exists in a directory and is executable, consider the **access()** system call. For example, when the user types **ls**, and path is set to include both **/bin/** and **/usr/bin/**, try **access("/bin/ls", X_OK)**. If that fails, try "/usr/bin/ls". If that fails too, it is an error. Your initial shell path should contain one directory: **/bin/**.

Using a list to store these path will be useful as whenever you need to check if particular file exists you can iterate through that list in order to get the required path for that file.

3.3 Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the exit built-in command, you simply call exit(0); in your witsshell source code, which then will exit the shell.

In this project, you should implement **exit**, **cd**, and **path** as built-in commands.

- **exit**: When the user types exit, your shell should simply call the exit system call with 0 as a parameter. It is an error to pass any arguments to exit.
- **cd**: cd always take one argument (0 or more than 1 args should be signaled as an error). To change directories, use the **chdir()** system call with the argument supplied by the user; if **chdir** fails, that is also an error. Errors are discussed later.
- **path**: The path command takes 0 or more arguments, with each argument separated by whitespace from the others. A typical usage would be like this:

  ```
  witsshell> path /bin/ /usr/bin/
  ```

  which would add **/bin/** and **/usr/bin/** to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The path command always overwrites the old path with the newly specified path.

3.4 Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the > character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types:

```
witsshell> ls -la /tmp > output
```

nothing should be printed on the screen. Instead, the standard output of the **ls** program should be rerouted to the file output. In addition, the standard error output of the program should be rerouted to the file output (the twist is that this is a little different than standard redirection).

If the output file exists before you run your program, you should simple overwrite it (after truncating it).

The exact format of redirection is a **command** (and possibly some **arguments**) followed by the **redirection** symbol followed by a **filename**. Multiple redirection operators or multiple files to the right of the redirection sign are errors.

Note: don't worry about redirection for built-in commands (e.g., I will not test what happens when you type **path /bin** > **file**).

3.5 Parallel Commands

Your shell will also allow the user to launch parallel commands. This is accomplished with the ampersand operator as follows:

```
witsshell> cmd1 & cmd2 args1 args2 & cmd3 args1
```

In this case, instead of running **cmd1** and then waiting for it to finish, your shell should run **cmd1**, **cmd2**, and **cmd3** (each with whatever arguments the user has passed to it) in parallel, before waiting for any of them to complete. Remember this is possible since the mother process continues after creating a child, therefore we can create more children to handle the other commands.

Then, after starting all such processes, you must make sure to use wait() (or waitpid) to wait for them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

3.6 Errors

The one and only error message. You should print this one and only error message whenever you encounter an error of any type:

```
    char error_message[30] = "An error has occurred\n";
    write(STDERR_FILENO, error_message, strlen(error_message));
```

The error message should be printed to **stderr** (standard error), as shown above.

After most errors, your shell simply continue processing after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling **exit(1)**.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to ls when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

4 Hints

- USE A DEBUGGER - https://gourav.io/blog/setup-vscode-to-run-debug-c-cpp-code
- The hardest part about the project is string handling, so take the time to develop the necessary functions or data structures you need in order to help make everything easier.
- Remember to get the basic functionality of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as **ls**). **ls** is a program which lists the files and directories in a given directory. If no arguments are given it assumes you want the current directory in which the process is executing in.
- Next, add built-in commands. Then, try working on redirection. Finally, think about parallel commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.
- At some point, you should make sure your code is robust to white space of various kinds, including spaces ( ) and tabs (**\t**). In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection and parallel commands) do not require whitespace.
- Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense.
- Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle – other users certainly won't be.
- Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

5 Testing

A shell tester is provided on moodle along with a starting file called witsshell.c. Once you have compiled this file into an executable called "witsshell" you can use the tester to test your shell. To do this simply run:

```
.\test-witsshell.sh
```

You should see a tonne of console messages populating the terminal indicating to you how you have done on various tests. These tests can be found in the "tests" folder

- .desc - gives a description of the test
- .err - indicates what error should be returned
- .in - indicates the input given to the shell
- .out - indicates the shells output

The tester will generate output files in "test-out" which correspond to the how your shell performed given a particular test. Compare these versus the desired output to learn where you went wrong. **NB: These tests are EXAMPLES** and are not the exact strings that will be used for your final grade. i.e. don't just have 25 if statements and expect to get marks. Try to get the basic tests working first. Lastly due note that the tester is essentially testing your shell in batch mode.

6 Mark Allocation

- 4x2 marks for basic tests = **8**
- 1x22 marks for normal tests = **30**
- Total **30** Marks

Academic Integrity

There is a zero-tolerance policy regarding plagiarism in the School. Refer to the General Undergraduate Course Outline for Computer Science for more information. Failure to adhere to this policy will have severe repercussions.

During assessments:

- You may not use any materials that aren't explicitly allowed, including the Internet and your own/other people's source code.
- You may not access anyone else's Sakai, Moodle or MSL account.
- You may not use any device other than the lab machines.
- You may not edit your submissions using any other device either inside or outside the designated venue.

Offenders will receive 0 for that component, may receive FCM for the course, and/or may be taken to the legal office.