**Name: Jacob Chan**

**Email:** [**jacob.chan@afit.edu**](mailto:jacob.chan@afit.edu)

**Class: CSCE489**

**Date: 20 July 21**

**Project 1 README**

Git Repo: <https://github.com/jacobc40/CSCE489.git>

\*Note\* Project 1 repo is in Master branch while Project 0 repo was in Main branch.

**Questions**

1. Why was it necessary that your shell use the fork command before running the exec() system call?

**If I did not call the fork command before actually executing commands, it would be the main process that executes the commands. If we call fork before executing the commands, we can specify which process, either parent or child process, would be the one to execute the command.**

1. Explain, in as much detail as possible, what happened when you called the fprintf function? Keep in mind, I’m looking for an answer that discusses the process from when the function is entered, through the different “layers” until the function is complete.

**I first passed in 3 arguments to the fprintf function: the file, a %s formatter for strings, and the string to append to the file. When I called the fprintf function in the update command with those arguments, it adds to buffer the string I wanted to append, in the format I specified, and then waits. It will only output to the file after everything has been written to the buffer, i.e. when fflush() or fclose() is called. Understandably, if you run an update command with many lines of text (and calls to sleep() to artificially increase run time) and run it in the background, if you call the list command, it will show no new text has been added to the file until the update command exits. You will not see any new line of text until the whole text has been buffered and forced to output to the output device via fflush() or fclose().**

**Under the hood, fprintf is a function that is a part of the C library/API. C, being a systems programming language, runs in user mode but will call the write() system call on behalf of the programmer/user. In kernel mode, the kernel will determine whether this system call is valid and if so, will execute it on behalf of the user before returning control back to the systems program.**

**Introduction**

This is Project 1 of CSCE489: Operating Systems. In this project, we are tasked with writing a simple Unix shell that executes 5 different commands:

1. Create <name>
2. Update <name> <number> <text>
3. List <name>
4. Dir
5. Halt

Each of these commands will take in the mandatory arguments specified above and optionally, include an & symbol to specify that the command should be run in the background instead of the foreground.

<name> represents a file name and can be any valid Unix file name

<number> represents an integer value of how many times to append a line of text to a file

<text> represents a line of text to append to the file

Overall, the project was fairly challenging. The shell functions were easy to write but it required a lot of googling on different C functions that were supported. I first wrote the code on VS Code on Windows. It worked perfectly fine until I ported the code over to Linux. Then I ran into a lot of compiling and linking errors. That had to do with Windows not able to run a lot of Unix system call functions such as fork() as well as many dependencies not being supported outside of VS Code. A big bulk of time was spent on error checking and user input validation. Great project and was definitely an eye-opening experience.

**Noticeable Design Decisions**

In an attempt to save time and not produce a PDF that is overly long, I will only go over some key design decisions I had to make. More detail can be found in the comments of my source code.

* Gets user input in one line using fgets() and tokenizes them
* Instead of looping through the whole line to get tokens, I chose to hardcode strtok() 5 times because I knew the max number of tokens possible should be 5 in the case of an update command with an & symbol at the end. This may not be the best idea because I’m storing a bunch of variables. However, the tradeoff is that I get to check for user input validity easier (i.e. whether they inputted the correct number of arguments).
* Utilized lots of short-circuiting to do my input validation checks, i.e. for OR statement, if A checks out, no need to check out B. In an AND statement, if A doesn’t check out, no need to check out B.
* Because I use “” as delimiter for <text>, if I don’t put my text between quotation marks, an & symbol is treated as part of the text if I enter it. Thus, I chose to do a check that if the last character in <text> is an & symbol, I remove it from text and treat it as an actual & symbol. Bad edge case is if user wants to type a command similar to update file1 3 hello, world&, it will remove the ‘&’ from the text and run the command in the background. The user can avoid this by surrounding their text in quotes.
* I chose not to use the sprintf(). Main reason is I couldn’t get it to work but other reason is execl() was more than enough to get the job done.

**Error Checking**

My program checks for the below error cases and reports an error to the user if found. Almost all user input errors are treated as non-fatal errors so we continue running the shell and ask for a different input from the user.

* If user presses enter without typing anything, it will return an error stating command was not found
* If user enters only space characters, it will return an error stating command was not found
* If user enters a space proceeding a command, it will return an error stating command was not found
* If user mistypes a command or enters an incorrect command, it will return an error stating command was not found
* If user enters a correct command but not enough arguments, it will return an error message specific to the command stating there were not enough arguments entered and what arguments are accepted
* If user enters a correct command but too many arguments, it will return an error message specific to the command stating that the user entered too many arguments
* If user attempts to create a file that already exist, it will return an error message stating the file already exists
* If user attempts to list a file that does not exist, it will return an error message stating the file does not exist

**Limitations/Flaws**

* To make processing easier, the user input is taken in as 1 string and tokenized into multiple tokens. Because I know that the max number of arguments that can be provided for this shell is 5, in the case with “update <name> <number> < text> &”, I tokenize the string into 5 tokens. I named these tokens intuitive names:

command = <command>

fileName = <name>

n = <number>

text = <text>

amp = &

However, the names above may only make sense for the commands that accept the right number of arguments. For those commands that accept less arguments, I reuse the token names. For example, list <name> only takes in one argument. If it is given a second argument, it must be & symbol. However, the second argument is already named “n” so it may be confusing if someone sees my code. They will see that I’m checking for n in the list command but list only takes in a file name. However, what I’m really doing in that case is checking to see if n is an & symbol.

* Another limitation is that I print the current working directory of the user before and every command like a normal Unix shell does. However, if the user specifies to run a command in the background instead of the foreground, it will mess up the formatting of the directory printing. I haven’t figured out how to fix this yet.
* My program doesn’t check whether a user inputted an integer instead of a string and vice versa. This specifically affects the update command. If the user inputs a string instead of an integer for <number>, it will still run the command but the file would not be updated.