Project 1 Midpoint Report  
Chase Lamkin and Caleb Shafer  
Sept 12, 2023

**Project Description**  
In part 1 of the project we created a container with a class called Param to store commands from the user. However, we also created a parser to tokenize these commands and allow the storage of information in Param to be understandable by applications. A command line shell, just like a command line tool, needs to take in arguments to know how to process the users request. Being able to understand these arguments and relate them to a functionality of the shell is a critical step or else the shell will be unable to do what you want it to. This is why many tools like argparse and CLI11 exist. By doing this in part 1 we have taken the first big step in creating a shell by creating functionality that can give the shell understandable input.

**Achievements**

To solve this problem I figured out needed functionality to incorporate param in this project. I decided to leave param as purely a container for arguments and give all functionality to my class Parse. I decided the most vital function to have was a run and a grabArg. grabArg would be instrumental because I would be able to give it an index and it would grab the full word up until it reaches the end of the string, a space, or a special character. Run would then fill in the gaps by giving logic to when and how to use grabArg and it would integrate it with the Param container. The data structure used was an array of pointers to dynamically store a varying number of arguments of varying sizes. I used the cstring library, to allow me to perform operations on these c style strings easily. The largest issue encountered was creating an LL parser that was able to parse the string and tokenize effectively without causing segmentations faults, and successfully creating grabArg through lots of experimentation was instrumental in solving this.

**Preliminary Testing**

Test cases I used for my code were primarily for the run() function to ensure that its logic created errors when invalid input was received and that it would also give the specified output. Some test cases I included were:

1. Test test \t (to ensure \t was not added as an argument
2. Test test < test (to ensure filenames did not have have a space next to specifier
3. Test test \j (to ensure only correct special characters were used)
4. Test test $ -Debug (to ensure $ is properly set, and the printParam method works)

Next Steps

The next step in the problem I believe is dependency injection. Creating a interface and a factory for this interface will allow me to easily solve the largest problems that I foresee: foreground and background, file locking, and directory management. Dependency injection will allow me to inject foreground and background requirements into all commands. To handle foreground and background I will implement a counter that ensures all forks are properly handled before a certain number are created. To handle input and output of files, the terminal as well as directory management I will also implement a locking mechanism in the factory that ensures only one process is accessing a input output method at a time. It will force programs attempting to access it to wait, and if too many processes become active, the factory will handle the waiting processes.