CSSE332 Operating Systems

Team 1-F Reflection

Known Bugs

Due to extensive testing and carefully written code, our operating system currently has no known bugs.

Special Features

Our operating system supports all of the required basic features for an operating system such as reading a file, copying a file, printing out a message from a file and much more. However, our system also has several special features. One special features that our operating system has is that when the user uses the dir command to print out all of the files in the system, the size of each file is also printed. Also, our system has a command to clear the terminal's screen. Our system also has a command which displays all of the commands our system supports.

Furthermore, our system prints out "FNF" which stands for "file not found" if the user tries to use a command on a non-existent file. Also, our system prints out "invalid command, type "help" to see the available commands" if the user tries to use an invalid command. The last special feature that our system supports is quit, which kills all process including the shell.

Interesting Implementations

The most interesting implementation that we used was to use character buffers instead of string literals. We did this because string literals fill up the global space in C and cause strange and cryptic bugs. We found that character buffers were a way to solve this problem; however, some of our longer strings such as our error message for an invalid command resulted in some awkwardly long pieces of code.

Another interesting implementation strategy that we used was to use several small functions instead of fewer large functions. For instance, we created an initializeProcessTable function in which we initialized the process table. In main, where we need to initialize the process table, we simply call this function instead of having to initialize it there.

Morgan Cook Individual Reflection

I learned several lessons from this project. The first and most important lesson that I learned is that planning ahead is important. Just because our team finished one milestone in a certain amount of time didn't mean that we would finish the next one as quickly. Also, I learned that everyone has a busy schedule and you have to plan meetings early in order to get a time that works for everybody. Another lesson that I learned is that just because you think you wrote a certain piece of code right, if you haven't tested it, you can't be sure. Also, if you're experiencing an issue in your code, the problem might be in the function you're calling and not the one you're writing.

In addition to the non-technical lessons that I learned in this project, I also learned several technical lessons. For instance, I learned not to use C literals or global variables in 16-bit C. Instead of using global variables, I learned to use #define statements. I also learned the power of a make file. Putting all of the steps necessary to compile bochs in our compileOS.sh then just running that instead of writing in 4 different lines made the whole process much simpler. Lastly, I learned to appreciate built-in functions and debuggers. Having to write our own simple functions (like mod) and use print statements to debug was incredibly painful and made me grateful for the support modern operating systems and coding languages provide.

Josh Green Individual Reflection

From this project, I learned a lot about debugging and writing in C, as well as applying several concepts learned in class. Programming in 16 bit c provided numerous unique challenges that I had never had to worry about in previous projects. One example is managing the memory used in the global space. Since string literals in c fill up the global space, we had to adapt our code to fill up character arrays instead of using strings. Prior to learning about this issue with 16 bit c, we spent a lot of hours looking at strange bugs, like our shell simply not launching, but it became much easier after learning about this limitation. Another technical thing I learned is how to handle process scheduling by managing stack pointers and memory segments, which prior to this project was just a class concept that had not been practically applied in code.

Phillip Ross Individual Reflection

I learned a lot of things from this project, notably, how much I appreciate higher level languages. This project required us to program in 16 bit C without any of the standard libraries, so things like converting an int to a string suddenly became an issue. Manually managing memory was also a pain when I was trying to implement a command history into our shell. One of the difficulties I had was that when I was trying to have a global array of commands entered, it was the array was in the kernel's data segment, but the program still had a reference to the array, which wasn't correct, so when I thought I was writing to my global array, I was really just writing over random parts of our program, which caused weird bugs. This problem wasn't easily solved, since setting the data segment to the kernel data segment in order to have the right reference to my global array, the reference to my char array turned into garbage. The solution to this would be to write some new assembly functions to move my string to the kernel's data segment and get the string off of the stack. However there wasn't enough time to implement this without bugs.