Sophie Brusniak
Joseph Knierman
Stella Park
CSSE332-03

Operating Systems Project Reflection

Known Bugs in the Project

Although our operating systems project functions fairly smoothly, there are some existing bugs in the system. One such bug is that on occasion, if an executed function prints to the terminal, the “SHELL>” cursor is sometimes off by one. For instance, when “execute tstpr2” is called, it prints “SHELL> [printed message],” but the cursor waiting for the next command on the next line, which still works properly, does not have the “SHELL>” text before it. Another bug is present in handling logically invalid commands, such as when a user tries to execute a non-executable file. Instead of exiting gracefully, the shell merely freezes, disallowing the user to enter any further commands. An similar bug occurs when trying to run a deleted file. If the user were to delete an existing file then attempt to execute it afterwards, the shell freezes and once again the user is unable to enter any additional commands. Collectively, these bugs can potentially be frustrating for the user to work with, but they are for the most part harmless.

Special Features

After finishing the base features for our operating system we moved on to implementing additional features. The first feature we added was the ability to close the terminal from the shell. To do this we invoked a specific interrupt, and when executed the assembly code would close the terminal. The next feature we added was a help menu. All of the help instructions are stored in the helpmes.txt file and when help is typed into the shell it will print the contents of this

file into the terminal. We also added commands for changing the text and background color of the shell. Both of these shell calls were given separate functions that take in a specific color index. This color index is then given to a helper function called get color which returns the specific hexadecimal value for a color. This translated hex value can then be passed directly into an interrupt resulting in a color change for either the background or text. The last feature implemented is a clear function. When called from the shell this function will place 24 new lines into the terminal and print the shell prompt at the top of the screen.

Interesting or Clever Implementation Techniques

A clever technique used to implement some of the additional features was referring to existing assembly code to extract values to use in some of the interrupts used to ensure the feature functionality. For instance, for the quit command, the numbers in the method interrupt(0x15, 0x5307, 0x0001, 0x0003, 0) were determined from the following existing assembly code from a related post on stackoverflow: MOV AX, 5307; MOV BX, 0001; MOV CX, 0003; INT 15. Similarly, this technique of referring to assembly code to determine the appropriate interrupt parameters was used to create the color-related commands changeBackGroundColor (“bgcolor” command in the shell) and changeFontColor (“txtcolor” command in the shell) methods as well. Another interesting implementation technique used is in regards to reading user input from the terminal. The way our team chose to implement reading input was to use many nested if statements that went through the command, one by one, to determine which command the user wanted to run. Although there may be more efficient ways of doing this, the implemented style makes it somewhat easier to understand the shell code upon first glance.

Lessons Learned

Sophie:

From the project, I learned about the importance of being consistent in the code and how sometimes apparent shortcuts may actually not work as anticipated. For instance, when accessing the process table in the code, we thought we could sometimes get away with calling setKernelDataSegment without calling restoreDataSegment at the end of the method because it seemed like it would be closed elsewhere in the program or wouldn't be necessary to do in some cases to achieve the same desired functionality. Although this shortcut worked fine in some cases, it ended up leaving us with a bug in our killProcess method that would have been avoidable had we just passively assumed our shortcut would work. Another lesson that I learned is in regards to the actual benefit of pair/triple programming as opposed to the divide-and-conquer approach. Since our team chose not to divide the work and work on it all together at the same time, we were able to understand all aspects of the program and have a quicker debugging process. This, I believe, not only allowed us to save time by fixing bugs early on, but made the overall work time more productive.

Stella:

The most important lesson I learned from this project was the importance of committing. For instance, committing pushed each team member to have their codes updated. Bugs were easier to spot due to the number of participants constantly looking over the code. Furthermore, as all the team members became actively involved in every part, I was able to better comprehend the material by discussing the different aspects of the code as a group. Also, we were able to achieve our goal faster and were able to find the bugs more easily.

Joseph:

After completion of the project one of the most important lessons I learned was code errors are not always where you would expect. For instance there were many point when making the OS we encountered errors , but the problem was not in the code we had just written. Often the error was in previous code that had not been thoroughly tested, and now cascaded errors into new sections. These errors were often difficult to track down and required us to have some type of notification to signal every step in the process. This stressed the importance of thoroughly testing all of the code before moving. This might cost more time upfront but will save time later on.

Technical Things Learned

Sophie:

The biggest technical takeaways that I got from the project were understanding how to handle actions that interact with stack pointers, sectors, and memory, whether that be through actions that require reading or actions that require writing and storing. This sometimes got as specific as using a base memory location and an index to determine a specific segment. Another technical thing that I learned from the project is how to write working code using bcc. After using gcc for so long, it was interesting to learn of the distinct similarities and differences between the two. For instance, I learned that bcc required all declarations of functions and variables to come at the beginning of the program or method whereas gcc did not require this to compile and run a program. Collectively, I was able to learn about the inner workings of a simple OS along with the distinct differences of the gcc and bcc compilers.

Stella:

Throughout the project, I learned how different exceptions were handled and how new exceptions were implemented. One of the default interrupts that was used was 0x10 which provided video service. Also, I learned how to set up customized interrupt. The interrupt number that was used for this project was interrupt number 0x21 which allowed other programs to call the function without having the actual function. Furthermore, I learned that the parameters of the exceptions were the same as the parameters of the assembly code. Apart from handling exceptions, I learned how to program in bcc. Even though bcc was a lot like gcc, it had small but significant differences. For example, bcc required that all variables be initialized before any other code could compile or run while gcc did not. Accordingly, including other technical materials such as sectors and memory, I have also learned basic concepts of OS and was able to implement through hand-on experience.

Joseph:

While completing this project I learned a lot about how low level assembly is utilized by higher level programs written in higher level languages much like the kernel. For example in the project all code that interacted with the kernel required specific interrupts. Once called these interrupts could directly change values in the hardware. Another Important skill that I learned was how to compile code using bcc. This was a challenge and required a good amount of research, but eventually was able to get it into a working state. At the beginning of the project this was difficult but through enough trial and error we were able to get adquently acquainted to the syntax of bcc.