CSSE332
Final Project Reflection

Team 1-L

Members:

Michael Trittin

RJ DeCramer

Jaron Goodman

Known Bugs in the Project

Currently the behavior of create is to create a new file regardless of pre-existing files with the same name. Thus you can have two files of the same name on the operating system, and can only access the first one created (i.e. the file earlier in the directory). However, you can access the newer file by using the delete command, since it will delete the first file it comes across in the directory.

Special Features and How to Use Them

“clear” command

The *clear* command allowed us to clear the terminal (to help clean up the interface). It could be invoked from the terminal by simply typing “clear”, and the terminal would clear. Implementation just involved printing multiple newlines until the screen was cleared—a simple and effective solution.

File sizes with “dir”

The *dir* command—along with the functionality that was required— also prints out the sizes of the files that we have loaded into memory (in sectors). To do this, we loaded the directory sector into a buffer. For each of the files in the directory, we used a loop that would increment a count variable until a one of the sectors (after the file name) was equal to zero (at which point the count would equal the number of sectors that the file is occupying.

"Help" command

The *help* command prints out a simple help menu in the event that you forget a command or would like to see the arguments for a command. It's usage is simple: just type *help* at the console.

StringMaker enhancement

The StringMaker code that we received was useful, but it printed the output (the converted string) to the console. We decided to pipe the output to a program called “xclip”,

which would copy the output from StringMaker to the clipboard. This allowed us to quickly create a string and paste it without having to deal with copying the output from the terminal manually. **Note: “xclip” needs to be installed for the enhancement to work!**

Utility functions

Our OS repeatedly made use of commands like “atoi”, “itoa”, string comparison, and so on. Since this repeated functionality was necessary, we created a “lib” directory which held files like *math.c, string.c,* and *mem.c*. We linked this “lib” directory (using -I./lib) to allow users to include new header files and .c files in a very easy manner (since the new files could be included like system header files). The files included functions like **strcmp** and **strncmp** (which would compare two strings), **zero** (which would “zero-out” an array), **atoi** (which would convert a string version of an integer into its integer value), **itoa** (the reverse of atoi), and so on.

Interesting Implementation Techniques Used

The organization of our codebase was very important to us as we progressed through the project. To help make adding new code easier, our design was made with modularity in mind. Including a new module was is easy as adding the module files to the “lib” directory (as seen above, the code is compiled with “-I./lib”); this coupled well with the organization of our codebase—it was not necessary to add paths to includes (**i.e. #include “../../math.c”**), since it was in the “lib” directory and could be included with **#include <math.c>**.

Another technique that was featured in our design was the modularity of our kernel and shell code. To add a new command, all that was necessary was to:

1. Create a function that handled the logic of the command that shall be added.
2. Add an “else if” statement to the “runCommand” function in **shell.c**.
3. Add an another case to the switch/case in the “handleInterrupt21” function in **kernel.c**.

After these steps were completed, the shell and OS had the capabilities of using the new command.

Technical and Life Lessons

RJ

Life Lessons Learned:

I learned a number of lessons from working on this project. The first one is more an idea that was reinforced instead of learned. It was that paired programming is essential for projects such as this. Trying to split up group members and assign individual tasks may seem like it will cause the work to get done faster, but in reality, the number of errors that go unnoticed cause the time to take much longer and be much more frustrating. By working together, it allowed two of the group members to be able to catch mistakes that would have otherwise gone unnoticed while the other group member types.

Another lesson I learned was the importance of starting early on milestones you know will be more challenging. During the early milestones, we made a note of starting and finishing early so as to have more time to make sure the milestones were completed to specification. As the project progressed, we started to mistakenly assume that the milestones wouldn't get as hard as they did. We began starting the milestones later and later because we were able to finish them rather quickly. Then, once we got to milestone five, we were blindsided and had to work for almost two straight entire nights in order to get the project working. Had we stuck with the strategy of starting early, this problem would have never arisen, and we likely could have gotten more extra features working with the extra time provided.

Technical Lessons Learned:

As for technical lessons, I learned a little bit more about how interrupts and processes work within an operating system. There's only so much you can learn in class about how they work, but actually implementing them yourself really helps you envision and understand the process. I also learned just how fragile memory really is, and how you need to be careful when allocating it to certain spots. As we discovered, if the kernel or shell isn't allocated enough sectors, then they won't operate. The amount of sectors they need can be changed simply by adding in a few imports, or adding in new functions into the file. These are all considerations that need to be made when you can't take memory for granted.

Jaron

Life Lessons Learned:

This project was very intense and low-level from the beginning. While concepts from class were already solidified in my mind, the actual transition from learning to application is always an in-depth experience. This project especially taught me that even if an idea *looks* easy to implement, it does not mean that it will actually be easy to implement it. For example: during Milestone 5, we were tasked with incorporating multiprocessing into our OS. We read through Step 9 multiple times to get an understanding of what needed to be done, but when we began implementing it, we ran into some major issues (discussed in “Technical Lessons Learned”).

Not necessarily a lesson learned, but an affirmation of the importance of pair programming was produced from this project. Many hours were spent with all of us staring at a screen, bouncing ideas off of one another until it eventually worked, and then we would analyze why the particular solution worked compared to our previous attempts at making it work.

This project also hit home the importance of time management. Each of the members of our team have multiple team projects going on at once, so it becomes hard to schedule our milestone meetings when everyone is available. Luckily, we were able to manage our time very well, which (I believe) helped us to develop an OS that we could be proud of.

Technical Lessons Learned:

There were a lot of interesting technical things that I learned during the course of this project. Michael is a Sublime wizard, and he showed me a lot of neat tricks to help speed up the process of creating code. I also learned how to use new commands like “dd”, “bcc”, and “ld86”.

Project-specifically, I learned about the importance of doing as little as possible when switching back and forth with the KERNEL data segment; we ran into some interesting behavior (like garbage data being printed when “Hello World” should have printed), and it took a little bit of experimentation to discover how finicky the KERNEL could get.

Michael

Life Lessons Learned:

Start early, don't work too long. This project in particular emphasized that point to me. When we were working on the project from five/six until midnight or later for milestone 4, I knew that doing all of a single milestone in one night was not a good idea. For milestone 5, we gave the project two days, but since milestone 5 was also about twice as long as milestone 4, I think we should have given it more like 4. The worst part about working from five until midnight or later isn't the lateness, but just that staring at code for that long is a bad idea and not productive. If we had split the coding up into much smaller segments, our lives would have been much easier and overall we would have spent less time on the project.

Another lesson I learned was the value of pair programming. Every milestone we worked on was done together, and I think that this was an extremely good way to get things done. By ourselves, this project would have been a monumental task since many of the components require interactions with one another and the system may become complicated quickly. However, since we all programmed on one computer and rotated the task of programming around, this job was a lot easier and also meant that each one of us understands the details of our operating system. Having three pairs of eyes to check our work was extremely valuable.

Technical Lessons Learned:

I learned a ton in the implementation of this OS. I think my personal favorite technique was learning how to include header files using the –I flag. I will likely be using that trick in the future for some of my c projects.

I also have a love/hate relationship with the difficulty and challenge that comes with implementing the operating system from scratch. It was enjoyable, although sometimes frustrating, to learn how common functions like atoi, printInt, etc. might be implemented in a real operating system.

Another interesting technical lesson learned was how to create and use modular makefiles. Makefiles are a powerful tool in the right hands, but many of the tricks that they are capable of go unexplored in smaller projects. The OS project was large enough to give me valuable insight with regards to creating and maintaining a larger makefile.