Jake Gathof, Eric Moorman
Team 3
CSSE332 - Operating Systems
Defoe - 01
February 7th, 2018

Operating System Technical Documentation

Introduction

In the Bare-Metal Operating Systems project, our team developed an operating system capable of being stored on a 3.5’ floppy disk. We utilized the emulation software Qemu to simulate the booting and usage of our OS. Throughout the five milestones, we have implemented several features with a strong interest on the graphical interface aspect of an operating system.

Known Bugs

As of now, there are a limited number of known bugs in our system. One bug is that if a shell is run in the foreground, and another process is executed, the system will crash. Otherwise, executing any other process in the foreground will produce no such error. When switching terminal colors using the “set-b” and “set-f” commands, all text is cleared from the screen. This is caused by the fact that we use page scrolling to reset page color. We have also found that setting the background color initially will change the foreground color to black. This is a one-time occurance, and is usually more helpful than not, especially when switching to lighter colors. Additionally, by pressing backspace, the user can delete the shell prompt. This does not, however, interrupt the functionality of the shell.

Bonus Features

Beyond the bonus features suggested in the Milestone 5 document, we have implemented a couple additional bonus features. The first bonus feature is the “status” command. This command will output a list of active or waiting processes. This can be used to debug as well as to see which threads are being used.

The second and larger bonus feature that we implemented was image loading. In our current system, we can load 64 x 64 images, given a couple of factors. Because we wished to conserve space in the system, we have made it so that each image can only take up one sector. With a traditional 64 x 64 image, we would need to store 4096 individual pixels. However, a sector will only hold 512 bytes. Because of this and the way images are processed, the image must be black and white and be pre-processed by our custom algorithm. This algorithm represents each pixel of the image with only one bit, and then stores the bits into a text file. By reading the characters from this text file, we can successfully reduce an image to 512 bytes in order to reproduce the image on the OS screen. Due to this, we can successfully store a 64 x 64 image in a single sector of the system. The best example of this is our title screen. Whenever the shell starts, we execute our graphics drawing in the foreground. This blocks the shell from continuing until the user has terminated the home page by hitting the enter key.

Part of our goal in this project was to incorporate graphics as much as possible. This can be seen in the image loading functionality, but we also made an effort to make the UI as customizable as we found possible. With the “set-b” and “set-f”, a user is able to set the color of the background and the foreground of the terminal to fit their tastes. One can “reset” to bring the entire shell back to default settings. Going further, out team had plans to mesh together text and images to create a rudimentary GUI.

Interesting Implementations

Our most interesting implementation technique was the compression of our images. While the process may have been possible in C, our implementation actually required the use of Java to reduce an image down into a black-and-white representation, before compressing the file down to 512 bytes.

Non-Technical Reflection

Eric: I would say the biggest lesson that I learned from this project was to not procrastinate until the end of the project. Throughout this project we continued to put things off until the latest we could possibly do them. In every case, we still managed to get everything done for that milestone. However, this practice also made it so that we could not implement some of the bonus features that we had hoped to implement.

Jake: For me, the hardest part of this project was keeping up to date with the milestones. It felt like we would be just finishing a milestone when the next one was almost due. If I went back, I would encourage working ahead by a few days and starting the milestones before they were assigned. In addition, towards the end of the project, we ran out of space in our kernel, as the program had already grown too large. Looking back, we did not properly organize our project, so we had duplicated code throughout the entire document that could have been refactored to save space.

Technical Reflection

Eric: I learned several technical things from this project. Before this class, I had no idea how an operating system worked, and I feel that this project has made me understand it much better. I would say I learned the most when we did the last milestone, Milestone 5. There were several things in this milestone that I had never thought about that were causing us problems. For example, the fact that when you switch memory spaces, you lose the parameters that were given to the current method. While I knew that this happened, it is never something that I have had to deal with in the past. Additionally, dealing with process switching was challenging and had some very interesting problems; such as switching segments and stack pointers, and dealing with the different memory locations. Finally, I learned a good amount about how the Kernel is actually programmed. In the past, this has always been kind of a black box. But now, I feel that I better understand at least some of the Kernel's inner workings.

Jake: Throughout this project, I believe the most important takeaway is both how an operating system is much simpler than I originally thought, and also much harder. Before this class, the prospect of writing our own operating system seemed like a goliath task, one that was completely out of the scope of our capabilities. However, after only five weeks, we have developed a respectable system. In doing so, however, I have realized how much really goes into an operating system like Windows or OSX. I found it interesting to be working with 16-bit C-code. I have never worked in such an environment where we do not have access to standard libraries, and even declaring variables at the wrong spot can cause the entire compilation to fail.