Group 1-E Operating System Team (MSBOS1)

Tyler Rockwood, Kevin Trizna, Eric Downing

Known Bugs

There are currently no known bugs! :)

Special Features & Techniques

Shell Commands

clear - Clears the screen and resets the cursor to the top of the screen. This is done by setting the video mode using the 15h interrupt on BIOS.

quit - Closes the bochs emulator. This is done using the 15h interrupt on BIOS that requests a shutdown operation.

color - Does a color change on the background and the foreground, while clearing the screen. Like the previous three functions, this comes from the 15h interrupt.

help - prints a list of all functional commands. The help file is stored in memory and we type it to the screen.

Code Features

iota - (named atoi in the kernel.c) prints a string representation of an integer. Works for any valid positive integer.
parseCommand - (in shell.c) parses a command into an array of character arrays, with the first argument being the command name and the reset of the array being the arguments to said command. This made adding new commands very easy and saved us the time when trying to deal with multiple arguments for functions such as copy.

Technical Lessons

1) More so than our past experiences in computer architecture courses, building an operating system was a huge lesson in the style of programming that bridges the gap between the actual hardware and the code that needs to work with it.
2) The project was a great refresher on the nit-picky aspects of programming that are often overlooked at the higher level. For instance, attention to detail was critical when it came to null-terminators and other similar line-ending characters such as the newline and carriage return. Every single part of this project demanded an assiduous work ethic with line endings and it made you pay if you were careless with them.
3) This project showed how to build and structure the code within an operating system. Before this project, if you had told me how to, from a management perspective, I would have no idea. Know I know to have a kernel, a shell, and define all of the system calls within the kernel.

4) Learning how key the bios is in providing functionality to the operating system. Without specific hardware to do certain things within the computer, the OS would be useless. We have to rely on the hardware being there and well documented to develop and use the interrupts to get the functionality you need.
5) Just how many interrupts happen within a computer. Interrupts happen all of the time, and interrupts even happen within other interrupts.
6) Debugging is very difficult for an operating system if your simulation platform does not have something built in to see what code is currently being executed, what the values are in memory and give error messages when your code breaks instead of 'hanging'.
7) Memory management is often taken for granted with higher level programming. When designing at the lowest level, all of a sudden memory is a very precious resource that needs to be accounted for.
8) Pair Programming is necessary and needed for building an operating system, as calculating addresses, and making bios calls are much easier with two people trying to keep it in their heads and there is much less likely to be an address calculation error, or an error in shifting the bits.
9) Testing concurrent systems is difficult, especially when you are the one building the framework for concurrency. Our system's concurrency was very spotty for a long time.

Life Lessons

1) If the cursor blinks fast, you're going to pass. If the cursor blinks slow, you're bound for woe.
2) When working with very low-level code, its entirely possible that perfectly correct code will not run perfectly. Black magic or low-level (memory management) errors are very possible and having to take that into account forces you to program with a different mindset as opposed to high-level Java programming where you debug convinced that there is a logical error.
3) Test and work extremely incrementally. It just makes life and work much easier, in terms of software development, we found pair programming to be much more useful than in higher-level languages because you are more careful with memory and how you structure your code.
4) Not everything has to be done at once, or even early. There are a multitude of factors that can detract from your programming ability or debugging capacity, and sometimes the best debugging choice is simply to walk away from the code for a few hours or days and come back to it with a fresh mind and a new perspective.
5) Writing this operating system gave us a whole new perspective on just how well-written and useful certain algorithms and functions are such as printf(), automatic garbage collection, and string concatenation. The project showed us what it would take to actually write some of these and we also realized how lucky we are to have them in our everyday code.

6) As always, be assiduous in your SVN maintenance while committing and pulling from the repo. Binary files such as the kernel.o and the shell.o should be carefully maintained, if kept at all.
7) When planning out software design, it helps to keep a healthy list of responsibilities to work on. As referenced in lesson 4, sometimes you just need a break from code. Having other tasks to work on, no matter how small or insignificant, saves you from feeling backed into a corner by one massive, looming feature.