RandyOS Reflection

Austin Derrow-Pinion, Anthony De Rada, George Finn

CSSE 332-03 Winter 2016-2017

This reflection walks through various aspects for the implementation of RandyOS and the shell, named "Magic Shell."

Instructions for building RandyOS:
In the m5 directory

```
% make
```

Instructions for testing and using RandyOS:

```
% make bochs
```

This should open up bochs and present Magic Shell's prompt as below:

```
1F:>
```

Known Bugs

RandyOS does not contain any known bugs at this time. It performs all the required tasks and extra features perfectly.

Special Features

A few special features have been integrated into Magic Shell. These being:

- A command to list all programs in the directory with their corresponding size in sectors.
- A command to neatly print out the process table with each process's index, whether it is active or not, and the process's waiting status.
- A command to print out three different help pages of instructions for all available shell commands.
- A command to make an existing process wait on another existing process.

Implementation Techniques

RandyOS was created using a lot of enhanced loops to reduce code overhead. Several *'helper'* functions were implemented to keep both kernel.c and shell.c clean and readable. Some of these functions helped debug the programs while others just took care of redundancy, practicing good software design. An interesting implementation technique was adding additional functionality to the *execute* function. Instead of implementing a separate *'wait and execute'* function, an additional parameter was added to the old *execute* function to handle both operations. This parameter, if not -1, represents the process index in the process table that needs to wait for this new process to finish execution.

Lessons Learned (Individual)

Anthony De Rada

I learned that it is really important to pay attention to details when building an OS. We had to be really careful when accessing the kernel's data segment form a running process. I also learned that it is really hard to debug any code when using really basic compilers. This is why we used many helper functions to make it easier to detect bugs and fix our code. Another thing I learned was that keeping your code clean and readable is really important to avoid problems in the future. By keeping our code well organized and clean we saved a lot of time in finding specific places where our OS failed and fixing it.

George Finn

I learned that when working with processes and memory, there is little room for error. Because of the strictness of implementation on this project, I realized that to prevent getting lost, all code and methods must be very organized and modularized. From working with my teammates, they enlightened my understanding of how to connect all the pieces, how to debug, and how to find out where the problem is. From all of the peer programming done in this project, I found a new way to approach debugging, solving problems, and figuring out where the project is going wrong when there is little debugging tools available to use. In a program working so directly with memory management, debugging by simply guessing where the the program is going wrong is never the answer and will slow down progress. Sitting down, thinking about the solution, and carefully stepping through what the program should be doing vs what it is actually doing will always make the debugging process a lot faster.

Austin Derrow-Pinion

If I were to provide advice for future students doing this project, I would tell them to test their shell fully and often. This is because when working with this low level version of C and managing memory by yourself, it can be very easy to break functionality. It can be very difficult to debug the operating system because the lack of supported tools and information given when functionality breaks. This is why it is so vital to do best practices for debugging and finding problems early so that they don't waste a lot of your time when you realize something stopped working.

Technical Things Learned (Individual)

Anthony De Rada

The most important thing I learned was how to implement a basic kernel program for an operating system with a working shell. Also got to see how a scheduler with interrupts works . Implementing and using the kernel's process table really helped understand how an operating system can handle multiple processes. This part of the project was the most challenging and engaging. It was a confusing at first because the machine code functions that handled hardware were handed to us but by the end I was understood how memory was managed and how the registers were modified for multiple process to run.

George Finn

The project milestones really made it easy to connect all the dots when it came to building a bare metal operating system from scratch. The milestones stepped through the complexity of the operating system at a refreshing rate and enabled all of the different components of the operating system to come together in a comprehensive manner. The final milestone really forced a better understanding of how memory management worked and how the operating system utilized scheduling and timer interrupts to enable multiple processing. The biggest challenge in the final milestone were making proper use of the assembly functions and the implementation of the process table with the loading of programs. By overcoming the issues dealt with in the final milestone, I gained a new understanding of how multiprocessing works as a whole. By learning how the setKernelDataSegment and restoreDataSegment assembly functions actually worked with the registers, I finally understood how all the registers and usage of memory linked together to enable multiprocessing to be feasible.

Austin Derrow-Pinion

In class, we all learned about memory management, but this project allowed me to fully understand the i-node method of memory management by implementing it. It was also very interesting to see this one method of how multi-processing could be implemented. Before multi-processing was implemented, the order of events were all pretty logical so it was easy to see what was being executed and when. When the timer interrupt was introduced for the scheduler, it was not obvious at all when certain functions would be executing so it was interesting figuring out how to implement it in a way that won't ever break. Something I wish we could have worked on was virtual memory with paging, but I understand why that may be out of the scope and time for this project.