A Browser-based Operating Systems Project

JavaScript Adventures in Dinosaur Slaying

Alan G. Labouseur
School of Computer Science and Mathematics
Marist College
Poughkeepsie, New York 12601 USA
Alan.Labouseur@Marist.edu

Abstract: This paper presents one educator's experience with a browser-based project for an upper-level/graduate Operating Systems course. The author explains the project goals, why the browser in general and JavaScriptin particular are so well suited for this task, challenges andtheir solutions, the incremental assignments that ultimately result in a fairly complex OS simulation by the end of the semester, the response to the project, and some ideas about where to go next.

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education; D.4. [Operating Systems]
General Terms: Theory, Practice
Keywords: Operating Systems project, Browser-based software, JavaScript programming

1. INTRODUCTION

The Operating Systems project is a tough nut to crack. There are many OS concepts and constructsto cover before students can actually write one, yet we cannot delay the project too long lest they run out of time. These concepts are hard enough when handled individually, and their complexity builds as we combine them. We must attack this complexity if class work and project work are to compliment each other. Our weapon: an incremental project that's done in a modern, *friendly and immediate* environment.

Friendly and immediate means that our students can concentrate on what we're teaching and avoid the exasperating nonsensearound infrastructure. I don't care what version of the JVM is installed. I'm not worried about whether or not the IT staff loaded GCC libraries in exactly the right place. I do not want to hear that student logins don't have the privileges to alter process priority or overwrite a device driver. And I could not care less about the "group policy editor" that IT uses[1] to change these things. I want to get down and dirty in the first class, playing with interrupts and screen I/O. But how?

I got to thinking about web browsers. They are *friendly*: everybody in our target audience uses them. They are *immediate*: you change some code or markup, press refresh, and experience the result. They are also increasingly ubiquitous as they find their way into mobile devices, consumer electronics, appliances, game consoles, and more. So as a practical matter, experience with this most modern of platformswould be beneficial for Computer Science majors who may not have had the benefit of web programming courses in their curriculum.

Can we do a browser-based OS project? If so,with what language? All browsers run JavaScript. But JavaScript is a toy, right?So I asked myself, "An OS written in JavaScript . . . Seriously?"

2. JAVASCRIPT?

Upon further review: Yes, seriously.I realized that the web is full of Rich Internet Applications ("RIA" to us geeks) such as GMail, Yahoo Maps, LinkedIn, Facebook, Twitter, and more. They are all heavily JavaScript-based, at least on the client side. Microsoft is releasing Office Web Apps written in standards-compliant JavaScript and CSS.You'd think they might have used Silverlight since it's their own technology, but they chose JavaScript for their cross-browser RIA. Since there are browsers for every platform, JavaScriptapplications run anywhere, with no software to install or configure.*Friendly and immediate.*

But even as the ubiquitous programming language of browsers, can JavaScript provide the tools we need for systems programming?

JavaScript smells like Java and C#, but it's weakly typed. It is very (but not purely) Object-oriented, supporting prototypical inheritance, polymorphism, and some degree of encapsulation, but probably not enough. JavaScript is also somewhat Functional, supporting higher order functions and lambda expressions.

[1] but only on Tuesdays under a full moon

```
Clock simulation via JavaScript's setInterval and callback:
_hardwareClockId = setInterval(
      krnOnCPUClockPulse, CLOCK_INTERVAL);

Keyboard interrupt via JavaScript's event listener and callback:
functionsimEnableKeyboardInterrupt()
{
   document.addEventListener("keydown",
                 simOnKeypress, false);
}

Device driver "base" class:
functionDeviceDriverBase()
{
   // Base Attributes
   this.version = "0.07";
   this.status = "unloaded";
   // Base Method pointers to override
   this.driverEntry = null;
   this.isr = null;
}

Keyboard device driver subclass with prototypical inheritance:
DeviceDriverKeyboard.prototype =
                 new DeviceDriverBase;
functionDeviceDriverKeyboard()
{
   // Override the base method pointers.
   this.driverEntry = krnKbdDriverEntry;
   this.isr = krnKbdDispatchKeyPress;
}
```

It seemed possible that we'd have enough in there to build an OS. But I wasn't sure. So I experimented, on my students of course.

3. PLANNING THE WORK

I anticipated a few of the dinosaurs thatwe'd need to slay in advance. (I missed a few others. More on those later.)

3.1 Issue: JavaScript Might be New to Some

Browser-based programming might be new to CS students unfamiliar with "webby" issues like event handling, style sheets, and the HTML5 Canvas element. That there is no Eclipse or Visual Studio-like tool[1] for JavaScript development to ease this pain further exacerbates this issue.This being an OS class, and not one on web programming, I did not want to spend significant time on the subject.

Solution: Initial project and external support

I wrote a tiny initial project[2] that I distributed on the first day of class wherein I handled some ofthose "webby" issues. It includedJavaScript example code cross-referenced to pages in our textbook[3] for the keyboard interrupt and canvas drawing. (I also included an "Easter egg" to reward close reading of my code. More on this later.) My code showedJavaScript's class/prototype pattern in use and served as a reference for the entire semester.Figure 1 contains a few OS constructs in JavaScript.

To empower my students to help themselves and each other, I set up some external support via our class web site[3], including JavaScript syntax and Canvas resources, Software Engineering Radio podcast links, our project blog, and more.

Lastly, I distributed information about JavaScript development tools for interactive debugging (FireBug), documentation (JSDoc), static code analysis (JSLint, JSMeter), unit testing (FireUnit, YUI Test, JsUnit), and acceptance testing (Selenium).

3.2 Issue: This is a HUGE development project

While certainly a good thing, I didn't want to overwhelm the students. But I did want the class work and project work to complement each other as we went along.

Solution: Incremental projects

We did five incremental projects ("iProjects") spread over the semester in sync with our class material.

1. Add some fun kernel-level shell commands.
2. Load and run one user program.
3. Load and run many user programs (round-robin).
4. Disk I/O with MS-DOS[4].
5. Load and run many user programs, support disk-based virtual memory,FCFS and Priority scheduling.

This allowed us to divide and conquer the OS such that each part lead into the next. Of course, mistakes made or shortcuts taken in earlier parts continued to haunt students in their subsequent projects, just like in real life.

3.3 Issue: CPU Instruction Set

What instruction set should we use?

Solution: 6502 with a tweak

I learned to program on a Commodore PET 4016 and an Apple //+, both of which ran on the classic 6502 microprocessor. So I indulged my sense of nostalgia and used the 6502 instruction set. This was particularly nice because it's quite straightforward and there's a "Virtual 6502" for testing already available on the web[5] (as a JavaScript application, naturally).The tweak: a new op-code (FF) to support system calls.

[1] although Aptana (based on Eclipse) is promising

[2] available at www.3nfconsulting.com/students-os.aspx

[3] *Operating System Concepts, 8th edition* by Silberschatz, Galvin, and Gagne

[4] No, not that one! The **M**arist **S**tudent **D**atabase-backed **O**perating **S**ystem

[5] www.e-tradition.net/bytes/6502

4. WORKING THE PLAN

Let's see how things went and the pitfalls for which I was not prepared.

4.1 iProject 1

Up first: practice JavaScriptand get intimate with my initial code. Students were asked to add some simple kernel-level commands to the OS, some of which I specified and some that they could invent. They had to modify shell objects and talk to standard out, which I had accidentally named StdIn[1]. (The students quickly discovered this and corrected me, which I loved.) I also encouraged them to add scrolling to the console display. It turns out that's harder than it sounds, though several students handled it nicely, one building an entire TTY abstraction.

Everyone did quite well, which was the idea for the first project. The (30 student) average grade was 91%. Some added creative and unusual commandssuch as "WhatsForDinner" which printed a random menu from a predefined collection, "BondJamesBond" which listed all of the 007 films, and "Dungeon" which was essentially a "lite" version of the first few locales of Zork.

A few students found my Easter egg, discovering that if they typed insome "more colorful metaphors"[2] the OS entered sarcastic mode until they apologized by typing "sorry".Error messages in sarcastic mode were a little more abrasive than one might normally encounter.Those who took a real close look at the codediscovered this and a few… um… enhanced it.

4.2 iProject 2

For their second project, students were asked to enhance their code from the first project so that a user could enter a program in 6502 assembly and run it. They were to display the memory and CPU status (accumulator, registers, instruction pointer, etc.) in real time. See figure 2.

I read in the class blog that there were a few speed bumps around synchronizing the CPU simulation with the timer interrupt, so I wrote a tiny demonstration of this and posted it on the blogto clear things up as this was not clear enough in my initial project.

Once again the class did quite well, with an average grade of 91%.

4.3 iProject 3

Project three asked the students to take their second project and enhance it to run many programs in memory at once in a round-robin fashion. This meant implementing a ready queue (easy), a scheduler (hard), and handling context switches (also hard). See figure 2.

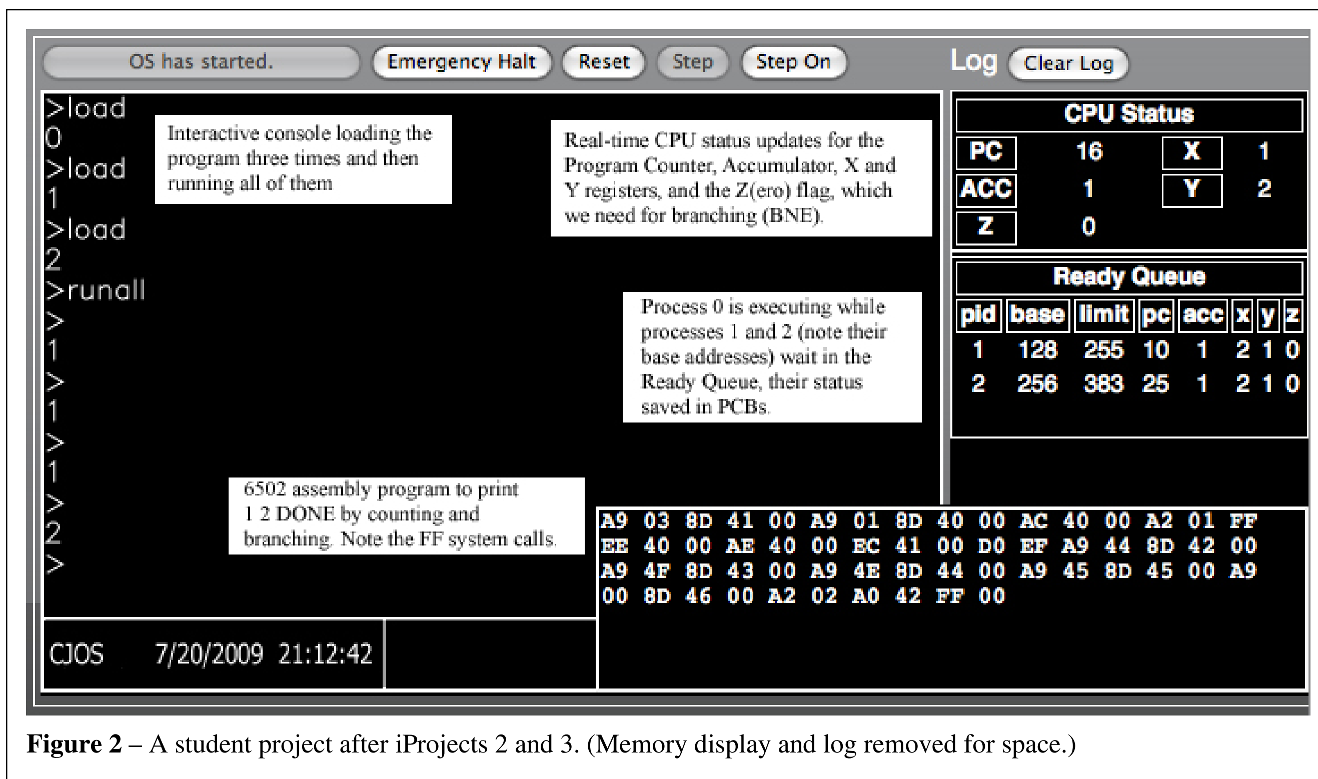

Figure 2 – A student project after iProjects 2 and 3. (Memory display and log removed for space.)

[1] I really didn't intend to do that. Both StdOut and StdIn pointed to the Canvas, so I didn't catch the mistake.

[2] I stored these in rot-13 form so that the source code remained family-friendly.

We were now fairly deep into the OS, and the class average of 87% reflected this. They handled base and limit registers with surprising ease, but there was some confusion about the division of labor between memory management in the OS and memory access in the CPU. I should have expected this, since the students were programming both. But once we cleared it up via in-class discussions, there was little further trouble.

4.4 iProject 4

This project gave the students a break from CPU and memory issues and let them focus on implementing a file system. They were happy about this at first, but it didn't last long once they realized that JavaScriptcan't access the file system. Needingan alternative,we turned to XML web services and JSONP. (Google Gears[1]was another option, and one student took that route.)

I had already decided that students could use only two primitive (and asynchronous) operations: GetBlock() and SetBlock(), and had developed a Web-based API for them:

```
GET GetBlock?username=str&track=str
      &sector=str&block=str HTTP/1.1

GET SetBlock?username=str&track=str
      &sector=str&block=str&data=str HTTP/1.1
```

I also wrote a utility service to return the entire "disk" given a username. This was not for use in their OS code, but as a tool to help in the development process.

```
GET UserDisk?username=str HTTP/1.1
```

The backing store was a SQL Server table hosted on my web site, giving each student 1 disk with 4 tracks of 8 sectors of 8 blocks of 256 bytes; or 65,536 bytes per student, way more than enough space.

This worked very well so long as the students ran their code from my web site. One of the issues I should have seen but did not was that running from any location other than my web site caused cross-site scripting (XSS) security violations in the browser. Not wanting to force my students to FTP their code to my server for every little change (because I encouraged testing early and often, after even the smallest change), I wrote them a JavaScript Object Notation[2] (JSON)implementation of the API.

JSON does not solve the XSS security restriction by itself, but including within the JSON call the name of a JavaScript function you want to execute as a callback does(usingscript tag injection). This is called JSONP, and with it my students could call the XML web services (well, technically,JSON services now) from anywhere they cared to develop.Figure 3 shows some sample I/O code with JSONP calls.

HardDrive object with function pointers to read and write methods:

```
functionHardDrive()
{
this.writeBytes = hdWriteBytes;
this.readBytes = hdReadBytes;
}
```

WriteBytes method using JSONP call (via the "callback" variable):

```
functionhdWriteBytes(track, sector, block,
 DATAS, callback)
{
var getUrl = ".../SetBlock.aspx?" +
"u=" + escape(DISK_USER) +
"&t=" + escape(track) +
"&s=" + escape(sector) +
"&b=" + escape(block) +
"&d=" + escape(DATAS);
  $.getJSON(getUrl +
"&format=json&jsoncallback=?",
    callback
  );
}
```

One (common) usage in the file system device driver code:

```
_krnFSDriver.writeBytes(0, 0, dirBlockNum,
newDirectoryEntry, dirCreateMsg);
```

Figure 3 I/O code featuring JSONP

Now that the students had the web service and JSONP APIs to work with, all they had to do was write device driver wrappers around GetBlock() and SetBlock(). Easier said than done.After a long in-class discussion, conveniently timed to coincide with those chapters in our textbook, we came up with a scheme for storing directories, a file allocation table, and system of pointers to block-by-block file contents (with chaining).

While conceptually easy to see, this was a complicated assignment to implement. There are a ton of moving parts in an I/O device driver. Each piece is fairly simple, but managing complexity (the heart of programming) is hard. Also, the asynchronous nature of JSONP calls gummed up the works a bit, forcing students to sequence the calls and responses themselves, just like a "real" OS.

The average grade reached only 83% despite my being relatively forgiving on several issues. By now I could tell I was pushing the students to their limits and beyond.

4.5 Final Project

The ultimate project challengedmy students to build on everything they had done before, adding disk-based virtual memory to the mix. They already had multiprogramming from project three and disk I/O from project four, so implementing swapping (not paging)[3] was a matter of creating and using a swap file, enhancing the Process Control Block data structure to keep track of what's in

[1] a browser plug-in for access to local storage

[2] a lightweight data interchange format that's easy for humans to read and for JavaScript to parse.

[3] Swapping is a process at a time, paging is more granular.

memory and what's not, and modifying the context switching code to handle swaps. Again, easier said than done, but very doable. I also asked the students to add first-come first-served (FCFS) and priority scheduling.

While many students were successful at adding FCFS scheduling (most realizing that FCFS can be implemented as round-robin with a huge quantum) and turning the ready queue into a priority queue, virtual memory – even at the level of swapping andnot paging – is really hard, as the average grade of 61% shows. Those who never got project four in good shape did poorly here.

5. DISCUSSION

This was a fun experiment; the class being a highlight of my academic year. The material is interesting, detailed, interrelated, and naturally makes for a large-scale software development project.It was a real pleasure.

5.1 Grades

As you can see from the decreasing project grades, the programming gets harder as we get further into implementingOS internals. This seems right, especially when you consider that each project relies on the one before it. The test grades, however, were very consistent, with averages of 83% for the first test and 82% for the second. The tests were based on in-class material only and independent, not cumulative.

5.2 Student Comments

Here are some (unedited) comments I received as part of the school's official student feedback survey:

"Managed to make OS fun"
"Very cool class"
"JSON proved frustrating"
"Building this OS has been very satisfying"
"Use of JavaScript was an innovative approach"

5.3 Anecdotal Responses

Many students expressed tremendous satisfaction at the end of the semester for having written such a large project. Several told me that they were grateful to (finally) get some web programming experience in a CS class.

5.4 My Assessment

The experiment was a success. While not perfect, JavaScript can provide the tools we need for systems programming. Seeing the results, I am convinced that we can do a browser-based OS project.

I'm pleased that my students could jump right in duringthe first class.This project presenteda low barrier to entry, unlike many other types of OS projects wheretoo much of the semester goes by before students can get going, or the infrastructure requirements cause delays in getting their feet wet. The modularized and interactive structure meant students couldbegin immediately and code as they go, receiving immediate feedback. Most found this both helpful and encouraging.

I'm also pleased this project developed modern webskills within the structure of a traditional CS class, unlike many other types of OS projects.

Lastly, I'm very proud of what they accomplished: writing OS simulationsthat implemented a file system with character and disk I/O that could load and run many programs simultaneously viaprocess scheduling and device drivers, all while being platform independent.Not bad for a single semester.

6. FUTURE WORK

This is only the beginning. Here are a few areas that I'd like to explore.

6.1 Browser-based Systems Programming Stack

This OS simulation is pretty thin when considered in a larger context. It's a middle tier, needing at least a compiler above it and a more detailed CPU below it.

While it's fine to have our students program in assembly language[1], it would be nice to have a compiler supporting a higher-level language and generating code for the simulated CPU within the OS[2].Given this compiler and our OS, several opportunities arise to explore larger systems programming topics such as Software Isolated Processes and many others.

The current CPU simulation is very primitive in that it's too high-level. A sophisticated CPU simulation would expose more low-level primitives (signaling, timing, etc.), the types of things we'd find in a Computer Architecture course.

6.2 Transactional Memory

This is another topic that works nicely with a *friendly and immediate* compiler and OS. Students would enhance theirmemory manager, memory accessor, CPU objects, and file systemto support ACID properties via TM compiler semantics.

6.3 Concurrent Programming

The CPU simulation is a JavaScript object. So far we've been using only a single CPU, but we could easily create as many instances as we like and address advanced topics such as processor affinity (asymmetric), load balancing (symmetric), and more. This is easier said than done, but the point is that it provides a*friendly and immediate* environment in which to explore multi-core programming on many levels.

ACKNOWLEDGEMENTS

I'd like to give special thanks to my OS students, who inspired me as much as I (hope to have) inspired them. I also want to thank the following people for their insight and suggestions, all of which improved this work: Ron Coleman, Eitel Lauria, Anne Matheus, and Roger Norton.

---

[1] because for learning there must be pain
[2] I have since done this in my Compiler Design course.