Proposal for Funds for Independent Student Research
Richter Scholars Program

George Fox University

Brian R. Snider

December 2005
Introduction

To develop a very thorough understanding of a particular subject, it is often helpful for one to be able build on previous experience. No matter what that subject is, when a student is able to use the knowledge gained in one course and directly apply it to another course, the student develops a deeper understanding of the fundamental concepts being presented. Additionally, the student is able to delve into the intricacies of the more complex course, having a solid background in those necessary fundamental concepts.

At George Fox University, the computer science program is very good at building a solid foundation. However, when a student reaches the upper-division courses, the topics no longer continue to explicitly build upon each other; these courses are mostly very specific to one facet of computer science. This necessary specialization of courses results in a disjoint in the pattern of building upon the previous. In the current upper-division computer science curriculum, the core topics of operating systems, computer architecture and assembly language programming, and compiler design are taught using several different computer platforms.

The operating systems course gives the student an overview of a generic operating system, and sometimes uses actual, in-use operating systems as case studies. Similarly, the computer architecture course presents the student with a very horizontal knowledge set that is common to many architectures; however, the assembly language programming portion of the course is very specifically targeted to one architecture, usually the ubiquitous Intel x86 platform. Likewise, the compiler design course is generalized at first to give understanding of the topic, but the majority of the work done by the student is targeted at one selected platform, the Java Virtual Machine.

While general knowledge is required to enable a student to progress through the course, the work done in each of these courses targets a different architecture. Though very intertwined in real-world use, during his or her academic career at this university, the student is not able to build upon the knowledge gained in one of these core upper-division courses in another course, except at the most general level. It would be highly advantageous to the student, and to the instructors of these courses, to utilize one target platform for all three courses. This ideal of a standard architecture for learning can be realized, but much work needs to be done to choose a system of hardware and software to be used.

The Problem

The problem is this: the current modern operating system is far too complex for the undergraduate student to digest in one semester. Even widely-accepted open-source operating systems such as Linux contain over ten years’ worth of source code to understand. There are currently very few (if any) suitable “teaching” operating systems, that allow an instructor to use that one single system as the case study; hence, the trend in all modern texts to use numerous case studies from several different operating systems.
Likewise, modern mainstream hardware architecture is far more complex than desired for a learning environment. A scaled-down or miniaturized architecture is needed also.

Proposal

If a suitable basic architecture could be found, so that a specialized operating system could be developed, the three core computer architecture-related courses could be reunited. The purpose of my research is to determine the best candidate architecture to build a “teachable” operating system upon. The task at hand is to assess each relevant architecture and categorize the traits of each, with respect to being “teachable” (meaning the level of complexity inherent in each) and develop a set of requirements for a complete yet simplistic system. Critical to the selection of an optimal architecture is the availability and usability of good development tools, e.g., compilers and debuggers.

From this notion of simplicity, it follows that a small, embedded platform with only the core functionality included should be used as the target architecture. Various potential embedded architectures exist, all available as a single-board computer with all the required hardware (CPU, memory, I/O subsystem) included in a small form factor. It is just a matter of applying the emergent properties of the ideal teaching system to the available architectures to generate a list of likely candidates, then testing each candidate for real-world suitability.

Methodology

- Assess existing operating systems for complexity, making note of the traits that are desirable in a “teaching” operating system.

- Define requirements for new operating system and associated hardware, and generate system requirements documentation.

- Using emergent properties of the ideal teaching system, select likely candidates for use as the target architecture.

- Acquire necessary hardware and supporting documentation, and set up development environment.

- Determine suitability of candidate architectures for use as the target architecture based on actual use programming within each development environment.

- Select best candidate and generate documentation to lay the groundwork for development of a “teaching” operating system.

The Value of Current Literature

The existing knowledge of porting operating systems to miniaturized architecture can be utilized in this project. For example, the vast work done on various Linux kernels,
such as the work done by the open-source uClinux project, can be referenced for proven methods of implementation. Additionally, the hardware documentation found at microcontroller and single-board computer manufacturer web sites provide valuable information about the architecture that not only will ease development by providing a “map” to the hardware, but also provide the foundation for much of the documentation that will accompany the resulting system, for potential use in courses by other students. Also, traditional operating system texts such as Andrew S. Tanenbaum’s Modern Operating Systems and others will give insight into the key components of the operating system and its underlying hardware that are deemed necessary for the computer scientist to learn. The necessity of the principles presented in these texts mandates a good deal of reading along with the initial research of current operating systems and related hardware.

Benefits

The benefit of this project is of immediate gain to George Fox University’s computer science department. Upon completion, the instructors of the three aforementioned courses will have one unified platform to teach from; they can build on one strong foundation. The operating systems course will have one understandable, concise source listing to reference; the architecture course will have one standard set of hardware to analyze, and to write assembly language programs for; the compiler design class will write their compiler to target one machine. Each architecture-related course the upper-division student takes will build upon the same case study as the previous; instead of starting with an empty slate, each course will build upon a foundation of prior knowledge. It would serve to strengthen, to make concrete and tangible, abstract notions of a computer’s inner workings – the processor, the memory management unit, job priority queues, filesystems – they would all be familiar components that a student has seen with his or her own eyes and has not only read about, but written code that makes them actually do something. This project could be presented informally on the computer science department’s web server, for other universities to make use of and emulate; more formally, it would be appropriate to present the results at an Association for Computing Machinery conference or publish in one of the various academic computing journals.

As a further incentive, the unification of the computer science and engineering departments can be achieved. Engineering students can work hand in hand with computer science students to build more complex machines that use the embedded platform as the controller. Each would bring his or her expertise to the table, the engineer designing the mechanics of the system – the gears and pulleys, the electronics – and the computer scientist writing the code that is the “brains” of the machine. Students in both majors would reap the benefits of working in a more real-world environment that crosses the barriers between one’s own specialty and another. An entire new breed of computer engineers would result, some computer scientists with engineering experience, some engineers with computer science experience.
# Proposed Budget

<table>
<thead>
<tr>
<th>Item</th>
<th>Description/Purpose</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-board computer development kits</td>
<td>Candidate target architectures; cost includes development tools such as the compiler, debugger, power supply, LCD readout, serial link to host computer.</td>
<td>$500</td>
</tr>
<tr>
<td>Student stipend</td>
<td></td>
<td>$3000</td>
</tr>
</tbody>
</table>
References

Printed texts:


Online documentation:


