Linux Kernel Projects for an Undergraduate Operating Systems Course

Rob Hess and Paul Paulson
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331, USA
{hess.paulson}@eecs.oregonstate.edu

ABSTRACT
In this paper, we present a series of programming projects based on the Linux kernel for students in a senior-level undergraduate operating systems course. The projects we describe cover several key operating systems concepts, including process scheduling, I/O scheduling, memory management, and device drivers. In addition, we assess these projects along several dimensions, from their difficulty to their capacity to help students understand operating systems concepts, based on six terms (three years) of detailed student exit surveys along with observations and anecdotal evidence. Through this assessment, we conclude that our Linux-based projects are an effective means by which to teach operating systems concepts and, additionally, that students’ response to these projects is overwhelmingly positive.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education; D.4.7 [Operating Systems]: Organization and Design

General Terms
Human Factors, Design, Experimentation

Keywords
Linux, Kernel, Operating Systems, Projects, Open Source

1. INTRODUCTION
Effectively teaching operating systems requires providing students with a good balance of theory and practice. How to teach operating systems theory is fairly well established, and a number of good textbooks exist for this purpose. How to teach operating systems practice, however, is a topic that still receives a great deal of attention in the literature.

The most basic approach to teaching operating systems practice is to have students implement solutions to hypothetical problems that are analogously related to operating systems concepts. Examples of such problems include the dining philosophers problem, which illustrates concurrency issues, or the elevator problem, which illustrates issues related to disk I/O. These problems and their solutions, however, really skirt the issue of operating systems practice by sidestepping most of the important details that lie at the heart of a functional operating system. And, in the end, they really only reinforce operating systems theory rather than teaching operating systems practice.

An alternative approach has students program within an instructional operating system such as Nachos [4] or Minix [12]. Anderson and Nguyen [2] provide an excellent survey of the many existing such systems. The motivation behind the use of these instructional systems is to expose students to some of the issues involved in making an operating system “work” while, at the same time, protecting them from as many as possible of the more formidable, nuts-and-bolts aspects of a real production operating system.

In this paper, we argue that protecting senior-level students completely from the details of a production operating system is not necessary and, further, that exposing them to some of these details can, in fact, prove to be beneficial.

Many educators have already documented their success in having students work on production software in other domains [7, 3, 6, 11, 1]. Some of the benefits of classroom production programming described by these educators include teaching students that most real programming projects are not started and finished by a single programmer alone, but are rather the collaborative effort of many; teaching them how to break down, understand and contribute to very large pieces of software written by other people; introducing them to programming environments similar to those they will use later in their careers; teaching them the importance of good documentation and well-written, self-documenting code; increasing their interest in programming and hence their motivation to program; and, most importantly, giving them a sense of achievement and pride in their work.

In fact, some educators have already begun having students do production programming in their operating systems courses. Indeed, as of 2005, an almost surprising 14% of the top 100 computer science schools reported using the Linux kernel in some form in their undergraduate operating systems course [2]. In the literature, Nieh and Vaill [10], like us, describe several projects that involve students programming directly within the Linux kernel. Similarly, Lawson and Barnett [8] describe having students modify a custom Linux kernel designed to run on the iPod. In both of these cases, the authors report not only that their students were able to successfully program within a complex production...
operating system, but that they did so with tremendous enthusiasm.

The goal of this paper is thus not to introduce Linux programming in an operating systems course as a novel practice, but rather to present a set of Linux kernel-based operating systems projects which can be seen as complimentary to the ones already described in the literature. Specifically, we describe five projects that cover system calls, process scheduling, memory allocation, I/O scheduling, and device drivers. Each of these projects (with the exception of the process scheduler, for which we provide students with virtual machine and skeleton scheduler code derived from the Linux process scheduler) is implemented directly within the Linux kernel and involves understanding and modifying existing Linux code to create new functionality.

We have run our senior-level operating systems course for six terms (three years) using these Linux-based projects, and like previous authors, we too have seen a great deal of enthusiasm in our students as they complete these projects. Our previous experience using an instructional operating system in the same course suggests that it is far more difficult for students to achieve the same level of enthusiasm using an instructional operating system as we have seen them achieve when programming in the Linux kernel.

We have seen other benefits to having students do Linux programming in our course, as well. The foremost of these is the exposure it gives them to the type of programming many of them will perform later in their careers, involving understanding and modifying a large, complex piece of software written by someone else. After doing this type of programming, many students demonstrate noticeably improved self-confidence in their own computer science abilities. Again, this is markedly different from our experience using an instructional operating system in the same course.

After outlining our set of Linux-based projects, we more fully assess their effectiveness in an analysis based on data from exit surveys distributed in each of the six terms we used our Linux-based projects. Through this analysis we show that students find the difficulty of our Linux-based projects to be appropriate for a senior-level course and, moreover, that students find these projects to be an effective way to learn operating systems concepts in practice. In addition, we combine our survey results along with personal observation and anecdotal evidence to demonstrate the level of enthusiasm our course generates and, further, to describe some of the benefits students see as a result of having completed our Linux kernel development-based projects.

2. COURSE DESCRIPTION

Our work is set in a one-quarter, senior-level course on the principles of operating systems. The course’s learning objectives include concepts related to concurrency, process management, CPU scheduling, synchronization, deadlock, memory management, disk management, I/O scheduling, etc. We address the theoretical aspects of these concepts in our lectures. However, another primary course objective is to have students design, implement and test functions related to these concepts within a large, complex code base.

For many years, we used the Java-based Nachos instructional operating system [4] as our code base for student development, but, while this package included a virtual machine and a small set of projects with auto-graders which made it fairly easy to use, we found that it was poorly documented and poorly maintained. In addition, because we had assigned the projects for several years, students were beginning to reference other students’ solutions from previous offerings of the course. Students also commonly complained that the projects had no relevance to the theoretical topics discussed in the course, and exit surveys showed that they would prefer more realistic projects.

Instead of developing new projects for a system we felt was too limited, we decided to choose a new system, but using another instructional operating system seemed like only a partial solution to our problems. Linux was attractive to us because it is both a real operating system and an open-source project with which students are familiar. This latter point was important to us because we felt it would help students approach our operating systems course enthusiastically, knowing that part of our course would involve them modifying a well-known and well-respected piece of software.

2.1 Development Environment

In the six terms we have run our course using Linux, we have evolved a development environment similar to the one described by Nieh and Vaill in [10]. Specifically, students work in teams of three to four members, and each team is provided with a virtual machine (VM) for testing their kernel code. Our VMs are hosted using VMware Server (freely available at www.vmware.com). This setup allows students to access their VMs from anywhere on or off campus, and it provides an inexpensive and less risky alternative to supplying each team with its own physical testing machine.

Each student team is also provided with a Subversion repository for revision control of the code they develop. This is useful, as it permits different team members to easily work on separate parts of their code in parallel. More importantly, using Subversion helps us to simulate a real production development environment where some form of version control software is generally employed.

To help students browse the Linux source code and easily track down function, structure and variable definitions, we refer them to an indexed, cross-referenced version of the Linux source, available online at lxr.linux.no. This has proved to be an invaluable resource for students, as has a course email list, over which we permit a certain degree of collaboration among student teams.

3. LINUX KERNEL-BASED OS PROJECTS

Below we describe our set of five Linux-based projects. Following that, we discuss several ideas for projects that are under development. As a compliment to these projects, we recommend using a good Linux kernel reference, such as the excellent book by Love [9]. Note that, because the projects we describe here involve low-level Linux kernel programming, familiarity with C or a similar language is a prerequisite. Also, note that the descriptions below are based on version 2.6.23 of the Linux kernel, though they likely apply to other versions as well.

3.1 Simple System Call

Our first programming project is designed to help students set up and acclimate themselves to our Linux kernel development environment and to help them overcome any initial

1Full project descriptions and the virtual machine and skeleton code for our process scheduling project can be found online at http://eecs.oregonstate.edu/~hess/cs411.html.
design a scheduler that assigns a time-slice to each of these processes as submitted in the Linux kernel. Students must machine are submitted to the scheduler in the same manner as specifically, processes which are created by the virtual machine operates on process “profiles” which can be hand Linux's constant time scheduler. In addition, our virtual Linux's rather elegant linked list implementation (located

### Memory Allocator

The Linux kernel is equipped with three memory allocators: SLAB and SLUB, which are complex frameworks designed for use in resource-rich systems to reduce internal fragmentation of memory and to permit efficient reuse of freed memory; and SLOB, which is a lightweight, efficient framework designed for use in embedded systems and other systems with limited resources.

The SLOB (Simple List Of Blocks) allocator, located in the Linux kernel tree at \texttt{mm/slob.c}, is a piece of the kernel that, unlike the process scheduler, can easily be extended by students. It works by maintaining a linked list of available blocks of contiguous memory called the free list, and, when a request for memory is made, it is serviced from the free list in first-fit fashion. If there are no available blocks large enough to service a given request, a new page of memory is allocated and prepended to the free list. Requests larger than one page are serviced separately.

As a project, we have students modify the SLOB allocator to service requests from the free list in either best-fit or worst-fit fashion, both of which are classical allocation strategies discussed in most operating systems textbooks. To verify their implementation works, students use configuration scripts to select their allocator and then compile and install the modified kernel. A defective implementation will typically not survive a small amount of normal usage.

To test the correctness of their implementation, we also have students write system calls to compute the total amount of memory on the free list as well as the total amount claimed by the SLOB allocator for allocations less than one page (i.e. memory that is either on the free list or has been allocated off the free list and not released). The values returned by these functions can be used to compute a rough measure of internal fragmentation, and students can use these values to compare the amount of fragmentation that results from using different allocation strategies in the SLOB allocator (typically, the best-fit and worst-fit strategies will suffer less from fragmentation than does the first-fit strategy).

This project is also straightforward and does not require a great amount of coding, but it begins to bring students into the inner workings of the Linux kernel and requires them to understand the code they are modifying before they start hacking. We typically allot two weeks for this project.

### I/O Scheduler

Another piece of the Linux kernel that is simple and easily extensible by students is the no-op I/O scheduler, located at \texttt{block/noop-iosched.c} in the kernel tree. This scheduler services I/O requests in first-in, first-out (FIFO) order and is designed to be used with devices such as flash drives, where disk access time is constant and independent of physical location on the disk. To determine the FIFO ordering of requests, the no-op scheduler maintains a request queue using the Linux kernel's linked list implementation.

As a project, we have students extend the no-op scheduler to implement the shortest seek time first algorithm (SSTF), which services I/O requests in order of increasing distance from the current location of the disk head. This is another classical algorithm discussed in most operating system textbooks. Extending the no-op scheduler to implement SSTF can be achieved in several ways. The most straightforward of these involves sorting the list of pending requests by their physical location on the disk and selecting the one closest
to the current location of the disk head whenever the kernel signals that a request should be serviced.

This project is quite versatile. For example, it can be changed from term to term by having students implement a different classical scheduling algorithm, such as SCAN or LOOK. In addition, it can be made more challenging by having students implement the sorted request list using the Linux kernel’s red-black tree implementation (include/linux/rbtree.h and lib/rbtree.c); by having them include additional features, like request aging; or by having them implement various optimization features available in Linux’s I/O scheduler API, such as request merging.

To verify the correctness of their implementation, we find it easiest to have students print to a log file the order in which requests are submitted to their scheduler and the order in which they are serviced. A quick examination of this file should suffice to show that requests are serviced in proper order. Alternative—though perhaps more complicated—methods for testing the I/O scheduler are available, such as the blktrace functionality built in to the kernel.

This project is more challenging than the previous two, as it requires significantly more design and coding, and we typically give students three weeks to finish it.

### 3.5 Device Driver

A common boast among Linux kernel developers is that Linux supports more devices than any other operating system in history. Indeed, device drivers account for more than half of the code in the Linux kernel. However, many devices exist that are still unsupported by Linux, and one of the best ways for a developer to get his or her code into the Linux kernel is by writing a driver for one of these. Writing a driver is also one of the most common tasks initially assigned to those hired by a company to do Linux kernel development. For these reasons, students find it particularly interesting to write a device driver as a project.

Unfortunately, since our development environment involves VMs hosted on a remote server and because we are constrained to work within a ten-week quarter, it is difficult to find real devices for which students can easily write drivers. Moreover, providing devices to students might be financially infeasible, even if an appropriate device could be found. We have thus found it easiest for students to write a driver for a virtual device. Specifically, we have them implement a RAM disk driver, which allocates a large block of memory and presents it in the form of a block device (i.e. a disk).

Chapter 16 of Corbet et al.’s book on Linux device drivers [5] provides a nice step-by-step tutorial on writing a RAM disk driver, and we have students follow this tutorial to produce a basic, working RAM disk. We typically have students extend their basic driver, for example, by using the Linux kernel’s built-in cryptography API to encrypt their disk or by having them add additional functionality, such as an “eject” function, which can be implemented as an ioctl command. Validating a RAM disk driver implementation involves mounting the disk, creating a filesystem on it, and then using that filesystem and observing whether data is correctly stored to and retrieved from the disk.

This project serves not only to have students write a real Linux device driver; it also introduces them to several key kernel APIs, such as the block device API and the cryptography API. Because this project does not deal directly with any of the theoretical material we cover in lecture, because students find it more interesting than other projects, and because they are allowed to follow the RAM disk driver tutorial in [5], we typically assign this project last, at a time when students are generally busy with final projects for other classes and studying for final exams. Two weeks are usually sufficient for students to complete this project.

### 3.6 Projects Under Development

Our course is under continual development, and there are a number of projects we are developing that we have not yet assigned to students or which we decided to develop further due to poor outcomes. Here, we discuss some of these.

**Filesystem.** Filesystems are another important part of the operating system, so we are working to develop a filesystem project. We are specifically considering one in which students modify the RAM filesystem (located at fs/ramfs/) to support version control functionality. As an alternative, we are also considering a project in which students would implement a filesystem in userspace using FUSE, an open-source Linux kernel module that allows users to develop filesystems outside of the kernel (available at fuse.sf.net).

**Process Synchronization.** Process synchronization is also an important part of an operating system, and we would like to develop a project in which students implement a process synchronization method and replace calls to existing synchronization methods in Linux with calls to their own. Nieh and Vaill describe a project along these lines [10].

**More Device Drivers.** Although the RAM disk driver project is one of our more popular ones, it does not give students the satisfaction of writing a driver for a real, physical device and seeing that device function properly when the driver is finished. Virtual machine-related issues aside, to have students write a driver for a real, physical device would require finding an inexpensive device that was enough and that came with a clear set of hardware specifications so that students could be reasonably expected to write a driver for it as a two- to four-week project. We do believe, though, that such a device exists, and we are considering options such as a simple USB device, like a thermometer, or a simple serial port device. Recently, our department also instituted an initiative in which students complete various course-related projects on their own Linux-based handheld devices as they progress through the Computer Science curriculum, and we are now working to develop driver projects for these devices.

### 4. COURSE ASSESSMENT

In every term we taught our course using the Linux kernel development-based projects, we conducted a detailed, open-ended exit survey of all of the students. In all, we received 243 responses to these surveys over six terms. While our surveys were designed to gather information of our own interest rather than as the basis for a formal research study, they still provide a great deal of information that is both useful and relevant to this paper.

Here, we use these surveys to answer three main questions:

1. Are our projects appropriately difficult for a senior-level undergraduate course?
2. From our students’ perspective, are our projects an effective means by which to teach operating systems concepts in practice?
3. What degree of interest and enthusiasm do our projects engender in our students?
The first two of these questions we ask students directly in the surveys. To answer the third, we use the following question as a proxy:

3’. Have the projects increased your interest in doing work on the Linux kernel or on other open-source projects?

Our rationale here is that students who are enthusiastic about the work they do in our course will be interested to continue doing that work after the course has ended.

**Difficulty.** To gauge whether our projects are appropriately difficult, we classified each student’s responses to question 1 as indicating that the projects were either appropriately difficult, too difficult, or too easy. In this way we determined that 66% of the 243 responding students found our projects to be appropriately difficult, and 7% found them to be too easy, while only 24% found them to be too difficult. These numbers suggest fairly conclusively that our projects are appropriately difficult for senior-level undergraduates.

One student’s answer to this question sums up students’ general attitude about our projects’ difficulty: “kernel programming... isn’t the black magic I used to think it to be.”

**Effectiveness.** To determine whether students felt that our projects were an effective means through which to learn operating systems concepts, we similarly classified students’ responses to question 2. Through this classification, we determined that 74% of the 243 responding students did feel the Linux-based projects were a good way to learn operating systems concepts, while only 23% felt they were not.

Again, we consider these numbers to be fairly conclusive proof of our students’ satisfaction with these projects. However, it is also informative to read some of what they wrote in this regard. For example, one student writes, “this was a great way to teach operating system concepts, just because it’s so practical, which makes us want to really understand what we’re doing.” Another writes, “I’ve always had a conceptual idea of how an OS works, but diving in like this made it very clear.”

**Enthusiasm.** We analyzed question 3’ in the same manner as we did questions 1 and 2, and found that 48% of students described an increased interest in open-source programming or Linux kernel development in particular, while 52% did not. Note, however, that this question, unlike questions 1 and 2, is not a zero-sum question. In other words, here, we are looking not for a majority opinion of the students surveyed but, rather, for the total number of students who reported increased interest. Indeed, we find it quite compelling that nearly half of our students have wanted to continue working on Linux or another open-source project after completing our course.

Again, it is informative to read some of what students wrote here. For example, one writes, “thanks to this class, I can follow my interest in developing device drivers for the kernel.” Another writes, “I’ve been interested in kernel development, but afraid to try. This course has given me a whole new perspective.”

**Self-confidence.** The last quote above also touches on a benefit of our projects that we didn’t try to measure formally through our surveys. Specifically, after completing our projects, many students appear to gain a good deal of self-confidence in their computer science abilities. Many of them speak to this in their survey responses. One student writes, for example that our projects “gave [him] confidence that [he] could do this as a career.” The following quote, however, epitomizes the attitude with which we believe many students leave our course.

_I was always so scared of the Linux kernel and now I feel like most operating systems have similar concepts, and I don’t have to be scared [of] those either... The transitive property of working on the Linux kernel has given me confidence for other systems as well._

**Hirability.** Finally, we note that many students see increased hirability as a benefit of their work on our Linux kernel-based projects. Many students who have graduated since taking our course have described to us how their work on our projects helped them earn a job. Some students, such as this one, even discuss this in their surveys: “I will be working on and contributing to another open-source project... during an internship at Intel, and telling them about what we’ve done in this class helped me get the job.” Others have told us informally that the Linux programming environment they used in our class prepared them well for the build environments they work with in industry.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we described a set of Linux kernel-based projects for a senior-level undergraduate operating systems course, and we assessed those projects along several dimensions, both formally and informally. There are many directions for future work, most notably in developing new projects, but also, for example, in devising better methods to grade our current projects.

6. ACKNOWLEDGMENTS

Much of our students’ enthusiasm is attributed to guest-lectures by Linux kernel maintainer Greg Kroah-Hartman, to whom we are extremely grateful. We are also indebted to Mike Marineau, Yunrim Park, and Ian Oberst for their hard work to help develop and run our course. We would also like to thank Google for their financial support.

7. REFERENCES


