Structured Linux Kernel Projects for Teaching Operating Systems Concepts

Oren Laadan
Dept of Computer Science
Columbia University
New York, NY 10027
orenl@cs.columbia.edu

Jason Nieh
Dept of Computer Science
Columbia University
New York, NY 10027
nieh@cs.columbia.edu

Nicolas Viennot
Dept of Computer Science
Columbia University
New York, NY 10027
nviennot@cs.columbia.edu

ABSTRACT

Linux has emerged as a widely-used platform for enabling hands-on kernel programming experience to learn about operating system concepts. However, developing pedagogically-effective programming projects in the context of a complex, production operating system can be a challenge. We present a structured series of five Linux kernel programming projects suitable for a one semester introductory operating systems course to address this issue. Each assignment introduces students to a core topic and major component of an operating system while implicitly teaching them about various aspects of a real-world operating system. Projects are of modest coding complexity, but require students to understand and leverage core components of the Linux operating system. The learning benefits for students from this approach include learning from real-world operating system code examples by expert kernel designers and gaining software engineering experience managing production code complexity. We have successfully used these structured Linux kernel projects to teach over a thousand students in the introductory operating systems course at Columbia University.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, Experimentation, Human Factors

Keywords

Operating systems, computer science education, open-source

1. INTRODUCTION

Kernel programming projects provide crucial hands-on experience for helping students to understand operating system concepts. Linux has emerged as a widely-used platform for teaching operating systems because of its many advantages. Its open-source software base and widely available development tools make it easy for students to access its internals. It provides students the ability to explore a real production operating system, not just a toy system, thereby also enhancing skills that can be immediately applied in the workforce after graduation. It is already maintained by real developers for real users, so there is no need to invest any effort maintaining a separate pedagogical operating system. It provides real code examples by real developers for students to learn from by example.

However, developing effective programming projects that teach students systematically about real-world operating system design and implementation in the context of a production operating system can be a challenge. Production operating systems are large, complex pieces of software. To gain a broad understanding of operating systems, it is useful to have hands-on experience with several different major components of an operating system. However, the sheer size of these systems makes them difficult to understand. It may be tempting to create projects that only touch on fringe aspects of a production operating system to avoid its complexity. However, fringe projects do not provide the opportunity to learn about the core functionality of the kernel. Production operating systems are also not primarily designed as teaching tools, but instead to be used in commercial deployment. Such operating systems are already fully functional and do not have major missing subsystems that students can design and implement. Redesigning several major subsystems of a production kernel may also not be feasible in the time constraints of a course.

We present our efforts at Columbia University over the past decade to leverage the benefits of Linux and address the challenges of designing kernel programming projects that provide educational value for an operating systems course. We have developed a structured series of five Linux kernel programming projects that exposes students to core components of a production operating system with modest coding complexity. These projects start with the basics of writing system calls and understanding the structure of processes, progress into synchronization which is necessary for understanding other aspects of operating system design and implementation, then explore three core operating system components in processor scheduling, virtual memory, and file systems.

Our projects are designed to require students to read and understand core components of the operating system to make the modifications necessary to complete the projects. At

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

the same time, the coding complexity is modest enough for groups of students with no previous operating systems background to successfully complete the projects in less than a few hundred lines of code. In addition to learning about the operating system components themselves, the projects are designed to implicitly teach students about various aspects of operating system structure and understanding how to modify a large code base and manage its complexity.

We have successfully used these structured Linux kernel projects to teach over a thousand students in the introductory operating systems course at Columbia University. We describe our experiences using these projects and analyze various aspects of students' submitted solutions. We demonstrate that the designed coding complexity of the projects is achieved as example project solutions are each under 400 lines of code and students' submitted solutions on average fall within the same range. We also demonstrate that common programming errors in students' submitted solutions become notably less frequent in subsequent projects as later projects reinforce the concepts taught in earlier projects, demonstrating that the projects provide an effective framework for progressive learning.

2. KERNEL PROGRAMMING PROJECTS

The kernel programming projects focus on five core topics in operating systems: system calls and processes, synchronization mechanisms, processor scheduling, virtual memory, and file systems. The topics and order of topics are intentionally chosen as they introduce key operating system concepts and build on one another. In particular, system calls and processes start with less complex components of the operating system and introduce students to basic kernel structures. Synchronization is next as understanding proper use of synchronization mechanisms is a prerequisite for being able to design and implement all other operating system components. Processor scheduling builds on the knowledge students have gained on the structure of processes from the first assignment and the use of synchronization on the second assignment. Virtual memory is deferred until the fourth assignment to only then focus on one of the most complex aspects of a production operating system. File systems is last among the assignments to follow the ordering of topics reflected in most common textbooks.

The assignments are carefully designed so that a group of students with no previous operating systems background can implement a correct solution in a few hundred lines of code. We accomplish this by choosing specific, scope-limited assignments that require intimate understanding of how the existing operating system kernel operates. It is not uncommon for students to spend at least as much time studying and understanding existing kernel code as they do implementing their solutions. Once students understand the relevant mechanisms in the stock kernel well, the implementation itself is fairly straightforward in terms of kernel code.

For the assignments, we direct students to pay careful attention to the handling of errors and unexpected conditions, including resource unavailability, security restrictions, and invalid inputs. While this is important for programming in general, this is particularly important for kernel code. Unlike user-space programs, the operating system kernel may not simply give up and terminate when it encounters an error. For example, when user-space programs terminate, their memory is implicitly freed, but the kernel must explicitly clean up all used resources such as locks held and memory allocated when an operation fails. Furthermore, given the increasing dominance of multicore systems, we require that students generate code in a manner that is SMP-safe. To do proper testing, students must generate test programs in user-space that demonstrate the correctness of their kernel code, including identifying corner cases.

2.1 System Calls and Processes

System calls lay the foundation for the interface between processes and the operating system. Processes are the main abstraction for application programs and the most important entity managed by the operating system. The first kernel project teaches students about basic concepts related to system calls and processes by adding a new system call. We design the project to acclimate students to Linux kernel programming including the development environment such as how to compile and install a new kernel and how to apply debugging techniques.

Both system calls and processes are good topics for a first kernel assignment because of their simplicity. Adding a new system call does not require in-depth knowledge of any kernel subsystem, and accessing process data is fairly straightforward. Moreover, both are fundamental notions that will be used in all of the remaining assignments.

To teach students about the basics of processes, the system call should perform a task that requires access to the kernel process data so that students can learn about the internal representation of processes in the kernel. We focus on process relationships, creation and termination of processes, and how process properties are inherited. Because in this assignment the students are not yet familiar with synchronization methods, we provide template code to handle synchronization and instruct the students to configure a uniprocessor kernel.

While there are many choices about what a system call might perform, we suggest designing the assignment to teach students about two key concepts in addition to the technical details of adding a new system call. First, the system call should involve data transfer to or from the calling program to illustrate the kernel-space versus the user-space view of memory. Second, to exercise handling of unexpected conditions in the kernel, we ask students to identify possible error conditions and select meaningful values to return in response. This also teaches how errors are propagated to the calling program. These two concepts exemplify important differences between implementing system calls and standard library calls in user-space.

Example assignment: system calls (1). One example assignment is to have students write a new system call that outputs records about descendant processes of a given process in DFS order. Each process record consists of some information about the process, such as its process identifier (PID), state and runtime statistics, and the PIDs of its parent, first child, and next sibling. The system call takes three arguments: the PID of the top process, a buffer to store the data, and an integer that indicates the size of the buffer.

Pedagogically, this assignment helps students develop a useful mix of skills. Students learn to traverse the process hierarchy and collect data about individual processes. They learn how to copy data back to user-space. They learn how to use basic kernel data structures, especially kernel linked list structures, which are used throughout the core compo-

nents of the kernel. They learn how to handle bad input (e.g., invalid PID) or failed memory allocations in a graceful manner. Finally, in testing the code, students need to produce process trees of specific forms, e.g. flat, deep, or very large trees, which improves their knowledge about process creation mechanisms and process relationships.

Example assignment: system calls (2). Another example assignment is to have students write two different types of system calls, one that modifies the behavior of existing system calls and the other to understand how processes run. The first system call accepts two integer arguments used to set a maximum limit on the number of bytes that can be read or written in a single invocation of the *read* and *write* system calls. This can be useful for testing the correctness of a user-space program in handling unexpected behavior of *read* and *write* system calls. The second system call accepts two arguments–a buffer and its size–and fills the buffer with a sequence of records describing when the caller was scheduled to run. Each record indicates the start-time and end-time of a scheduling period. Records are reported only once, and data older than one minute is discarded to avoid memory pressure.

While the first example assignment provides more experience with manipulating process list structures, this assignment provides experience with modifying the kernel process structure. In this example assignment, both system calls require adding a new field to the process structure in the kernel. Students need to adjust process creation code to correctly inherit (or not) the new fields by a child process. The second system call also requires adjusting process termination for proper cleanup and provides students some insight into the notion of context-switches.

2.2 Synchronization

Synchronization mechanisms are at the heart of an operating system's ability to function safely and correctly in the presence of concurrent or interleaving access to shared state. The second kernel project teaches students about synchronization and race conditions by implementing new kernel synchronization primitives. Students are assumed to be able to write system calls and use kernel list structures from their experience with the first project.

The new primitive is a high level primitive to be implemented using Linux's basic synchronization building blocks. We aim to achieve two learning objectives. First, students will learn about and gain first-hand experience with the native basic kernel synchronization primitives such as spinlocks, atomic counters, and wait queues. This is useful not only for this assignment, but also for all of the following assignments. Second, students will become familiar with higher level synchronization concepts such as fairness and the thundering herd, instead of re-implementing an already existing mechanism in the kernel.

To further deepen students' understanding of the topic, we introduce in the assignments three additional twists that we feel are often overlooked in operating systems classes. First, we introduce the notion of reference counting for shared resources by designing the new synchronization primitive to have global scope so that it is visible to all processes. To protect the validity of the new primitive in the presence of concurrent create, access, and destroy operations, a solution must maintain a reference count for the primitive and allow an object to be released only by the last user. Second, we direct students to select appropriate kernel primitives to produce a solution that is *correct* and *efficient*, in this order. For instance, students should prefer a spinlock over a semaphore if and only if they do not expect the process that holds it to block (sleep) on it. Finally, on occasion it is useful to forbid the use of certain synchronization primitives, either to avoid trivializing the assignment, or to guide students toward a path to a solution.

Example assignment: synchronization (1). One example assignment is to have students implement a synchronization barrier primitive that allows multiple processes to block on an event until some other process triggers the event. When a process triggers the event, all processes that are blocked on the event are unblocked; the operation has no effect if no processes are blocked. This is accomplished with four new system calls: `eventopen` to create a new event and return an event identifier; `evenclose` to destroy an event and notify all blocked processes; `eventwait` to block on an event; and `eventsig` to unblock all processes waiting on an event. The scope of an event is system wide so that any process may access an event by its identifier.

In this assignment we encourage students to leverage the kernel *wait queue* primitive and to learn from the many existing examples in the kernel that show how to use it properly. We also remind students that a reference count must be used to protect against concurrent access, using either explicit synchronization or atomic types. Finally, students are required to write a test program for their code. The test program should show that the kernel functions work for the usual cases, for example, with one process waiting, and also for corner cases such as a call to `eventsig` with no processes or multiple processes waiting, multiple concurrent calls to create a new event, a call to `eventclose` when processes are waiting, and a call to `eventclose` that coincides with an `eventsig` or `eventwait` call to the same event instance.

Example assignment: synchronization (2). Another example assignment is to have students implement a user-space policy mutex mechanism. The mutex primitive is provided by the kernel, but the policy for deciding which task to run in case of contention is determined by a user-space daemon that picks the task that gets the mutex when available. This is accomplished with six new system calls: `mutex_open` to create a new mutex and return its identifier; `mutex_close` to close a mutex; `mutex_down` and `mutex_up` to grab and release the mutex, possibly blocking if already taken; `mutex_list` to fill a buffer with the list of all mutexes in the system and their waiters; and `mutex_grant` to grant a given task that waits for a mutex access. The latter two system calls are intended to be used by the user-space daemon. The scope of a mutex is system wide so that any process may open it by its name, but each task has its own set of mutex identifiers.

This example assignment introduces three additional elements compared to the previous example. First, because mutex identifiers are per process, the students learn about the idea of a descriptor table, conceptually similar to the file descriptor table. Furthermore, they combine skills from the assignment on system calls to implement this table as an extension to the in-kernel process structure, which reinforces their knowledge. Second, we restrict students to use only spinlocks and atomic counters, as using the native mutex primitive would trivialize the assignment. Finally, moving some functionality to user-space (specifically, the policy that

determines the order in which waiting tasks gain access to a mutex) is useful for two reasons. First, working in user-space makes it easier to debug and experiment with different policies, e.g., to avoid the thundering herd effect. Second, through this split of functionality, students learn about the tradeoffs and possibilities in designing operating system interfaces and deciding on the boundary between the kernel and user-space.

2.3 Scheduling

Scheduling is a key concept crucial for an operating system to provide time-sharing and multitasking, while considering metrics such as throughput, response time, and fairness. The third kernel project teaches students about scheduling by having them modify the existing Linux kernel scheduler to add a new scheduling policy. Recent Linux kernels are particularly amenable to this approach because they provide a framework designed for adding new scheduling policies, as well as several existing scheduling policies which can be used as examples for implementing new scheduling policies.

The new scheduling policy can be any new policy, but we have several learning objectives to achieve. Students will learn about and gain first-hand experience with the structure of a kernel scheduler and how scheduling algorithms fit within the context of that structure. Students will learn how processes switch among different states of execution, such as sleeping, runnable, and running, and how those state transitions are handled as part of scheduler functions. Students will learn how multiple scheduling policies are typically supported by a common scheduling mechanism. Students will learn about what scheduling parameters can affect scheduling and how they are set or modified over time.

This is the first assignment that requires students to deal with a more substantial kernel subsystem, but the scheduler code is localized to just a few files and the assignment still only involves modification of only a relatively modest number of lines of kernel code. For simplicity, one can focus on uniprocessor scheduling policies to avoid the additional multiprocessor scheduling complexity of managing distributed processor queues and load balancing among them. The assignment builds on the previous two kernel programming projects as students need to write new system calls to control the scheduler and they need to be careful to protect kernel data structures with proper locking mechanisms.

Example assignment: scheduling (1). One example assignment is to have students implement a User-Weighted-Round-Robin (UWRR) scheduling policy. It operates by switching, round-robin, among users with runnable processes, giving each user a share of the CPU when it is that user's turn. The scheduler uses a hierarchical scheme to choose which process to run: first, a user is chosen, then, a process associated with that user is chosen. Each user has an associated weight that determines the proportional share allocation of that user. This scheduler was chosen both because it is easy to understand and implement, and because students can easily see why such a scheduling policy might be useful. It also does not cause starvation of processes if implemented correctly, making it easier for students to debug.

Example assignment: scheduling (2). Another example assignment is to have students implement a Container-Weighted-Round-Robin (CWRR) scheduling policy. It operates in a manner similiar to the UWRR scheduling policy, but with Linux containers being used as the grouping mechanism for processes instead of user identifiers. Process containers are a powerful isolation mechanism that is increasingly being included in production operating systems, so this assignment provides an opportunity for students to learn about this mechanism in the context of scheduling.

2.4 Virtual Memory

Virtual memory is another fundamental concept in operating systems. The fourth kernel project teaches students in detail about virtual memory and the paging system by having them implement mechanisms to measure and visualize how memory pages are managed and used, and then create programs that produce different memory access patterns.

We focus on measuring and visualizing how memory pages are managed and used, as opposed to implementing new virtual memory mechanisms. Because the virtual memory subsystem is typically the most complex part of the operating system, it can be difficult to cover any portion of it in any reasonable depth. We chose not to focus on relatively isolated parts of the virtual memory subsystem, such as memory allocators or the page replacement policy. Instead, we focus on mechanisms related to memory pages to cover a wider range of topics central to virtual memory including page table management, page faults, copy-on-write, and address translation as just a few examples. Implementing new virtual memory management mechanisms that cover these topics would be difficult for students to do in a reasonable amount of time for a production operating system. Our measurement mechanism approach focuses on students gaining working knowledge of the kernel's memory subsystem. Once students understand the stock kernel well, the implementation itself is fairly straightforward and requires under a few hundreds lines of new kernel code.

Example assignment: virtual memory (1). One example assignment is to have students implement a mechanism to visualize how memory pages are used by modifying the kernel so that the `/proc/[PID]/maps` file show additional information about pages in each memory region. For each memory region displayed we add a string that represents all the pages of that region, such that each page is denoted by a '.' if not present, a '1-9' for the page reference count if less than 10 if the page is present, or 'x' otherwise.

This assignment is useful because it helps students become thoroughly familiar with virtual memory concepts like the page table and paging, and process memory layout. To test their solutions, students must write user-space programs that generate memory regions with specific patterns, such as only not present pages ('.....'), all pages with a given reference count ('222222'), and even arbitrary valid strings. By writing test programs that produce different memory usage behaviors, students gain further insight into virtual memory internals. Finally, we reinforce the knowledge by also requiring students to examine the output of `/proc/[PID]/maps` for common programs and explain the role of each region displayed (e.g., code, stack, heap), its page table mappings, and how that data differs between different programs.

Example assignment: virtual memory (2). Another example assignment is to have students implement a framework to track the *working set* of processes, and then add a new system call to report to user-space the data collected. The system call fills a user supplied buffer with records. Each record describes a single memory region and carries a bitmap of the pages accessed recently in that region. Re-

gions without recently accessed pages are not reported. The system call then resets the monitored state to start collecting only new data.

One way to approach this assignment is to use the hardware access bit of the page table as an indication to whether a page was accessed. Because this flag is also used and reset by other functions in the kernel, students must study the related code and understand the implications of changes they introduce. For simplicity, students are allowed to ignore changes to address space layout such as due to mapping new memory regions, as long as their code does not crash. However, we do require students to identify the respective system calls and discuss the changes required to accommodate their effects, to raise students' awareness to this caveat.

Example assignment: virtual memory (3). Another example assignment is to have students implement a framework to track the *modified set* of processes. This is a variation of the previous example. The modified set is similar to the working set, but includes only pages that were modified, not just accessed. As before, students also add a system call to report the data collected to user-space and reset the monitored state.

Unlike before, using the hardware access bit of the page table is insufficient to track the modified set as it does not indicate whether a page is modified when accessed. Rather, students need to alter the page table of the process and convert all writable pages to be write-protected so that write accesses are intercepted. When a write-fault occurs, the kernel records the page address and restores the original protection of said page. By implementing this mechanism and writing test programs that produce write-faults, read-faults followed by writes, writes without faults, and copy-on-write faults, students experiment with and demonstrate knowledge about the page fault handler mechanism.

2.5 File Systems

Files are the other main abstraction other than processes that are managed by operating systems. As a fifth kernel project, we have students implement extensions to an existing file system code. This assignment requires students to gain practical understanding of how the virtual file system (VFS) infrastructure is designed, which is the key file system abstraction layer that every file system designer needs to understand. The project also gives students an opportunity to learn about the underlying file system realization of some VFS methods.

Disk-based file systems can be very complex, so we focus on implementing pseudo file systems. Pseudo file systems do not represent real, physical storage, but instead reside entirely in main memory and store files and directories that represent runtime information, such as `procfs`, `sysfs`, `ramfs`, and `devpts`, to name a few. Pseudo file systems make good candidates for pedagogic purposes: not only are they an effective platform to practice file system development, but they are also often used to report other kernel state and thus expose students to additional kernel mechanisms. To ease students into a successful solution we provide a skeleton implementation of a pseudo file system, saving the gory details of developing one from scratch.

Example assignment: file systems (1). One example assignment is to have students implement a pseudo file system to show the users currently running on a system, and the processes that they own. When mounted, the top level directory contains subdirectories that correspond to active users and are named after the respective user identifier. Each subdirectory holds two files: *"proc"* lists the processes of that user, and *"signal"* provides a way to signal all the processes of a given user in an atomic manner. In this way, students learn not only about file system principles, but also about users and groups. Moreover, a correct solution requires students to apply skills acquired from the second project on synchronization to manage race free access to shared objects.

Example assignment: file systems (2). Another example assignment is to have students implement a pseudo file system that gives information about the process hierarchy, by mirroring the tree structure. In this file system, directories correspond to processes and are named after the respective process's PID. Each directory holds one file *"status"* which contains select information about the process, and as many subdirectories as it has children. An advantage of this variation on the previous example is that it builds directly on skills and code developed in the first project on system calls and processes. By presenting an alternative mechanism to achieve the same goal, we illustrate the design options available to operating system builders, and foster comparisons between the two approaches in terms of usability and implementation difficulty.

3. EXPERIENCES

We have taught the introductory operating systems course at Columbia University using these Linux kernel projects for about a decade. Over a thousand students have taken this course. We assign five projects for a semester course, one project for each of the core topics: system calls and processes, synchronization, scheduling, virtual memory, and file systems. Students work in teams of two or three and are given two weeks to complete each project. We currently provide a VMware virtual appliance for doing the kernel project assignments, which can be easily deployed and run on students' personal computers without interfering with any existing software already on the students' computers. A distributed version control system is used to provide reliable storage for students' homework assignments, support students working together on group homework assignments, and manage the submission and grading of homework assignments. The experience of both students and instructors has been quite positive [4, 7].

An important consideration in designing the projects was to keep the coding complexity modest enough for groups of students with no prior experience with operating systems to successfully complete the projects with reasonable effort. To assess our approach in this aspect, we compared the size of our example solutions for a representative set of projects, to the average size of the students' solutions. The results are given in Table 1 in lines of code after stripping comments and blank lines. The results show that in all the projects,

Project	Example solution	Students' solutions
System calls and processes	146	164
Synchronization	353	465
Scheduling	399	389
Virtual Memory	66	63
File System	132	120

Table 1: Projects solutions sizes in lines-of-code

our proposed solution requires under 400 lines of code, and in nearly all cases students produce solutions of comparable size. The exception is the assignment on synchronization, where students produced more complicated solutions for the problem, with approximately 30% more lines of code on average. Producing an example solution in advance was an important step in the process of designing the specifics of assignments to estimate the expected complexity, and instrumental to ensure that it would be possible for students to complete with a reasonable limited amount of time coding.

It is also noteworthy to examine common errors encountered in the students' solutions as an indication to how students' knowledge and skills evolve as they proceed from one project to the next. For instance, two common mistakes in the first assignment on system calls were lack of input argument validation and incomplete error handling. However, they became notably less frequent in subsequent projects in which students had to repeatedly code additional system calls. Similarly, two common mistakes in the second assignment on synchronization were omitted locking around access to shared data and allowing a process to sleep while holding a spinlock. These errors became increasingly scarce later on, even though virtually all subsequent projects required repeated use of similar synchronization techniques. In this context, arranging the order of projects according to their dependencies is crucial to produce an effective framework for progressive learning.

Finally, although many students find kernel-level programming very difficult at first, they often say the work investment is warranted because they are learning a useful and applicable skill. We have received many comments from alumni who say the course turned out to be most useful course to them after graduation.

4. RELATED WORK

Courses that involve kernel-level projects with production operating systems have gained popularity in recent years; in 2005, 14% of the top 100 computer science schools reported using the Linux kernel in some form in their undergraduate operating systems course [1]. Nieh *et al* [6, 7] presents some early work on teaching operating systems by programming directly with the Linux kernel. Our work builds on this prior work and experience by describing a new, more structured series of Linux kernel projects with extensive examples for teaching core operating system concepts. Our work also leverages this prior work by using virtual machines to provide the necessary infrastructure to enable students to do kernel-level development.

Other Linux kernel projects have also been used for teaching. The SOFTICE [2] project advocates Linux Kernel Modules (LKM) to enhance students' hands-on experience. However, LKMs provide a narrow view of the kernel with a limited set of interfaces, and thus severely restrict the potential scope of projects. Lawson *et al* [5] describe one project in which students modify a custom Linux kernel designed to run on the iPod. Hess *et al* [3] describe some projects taught during six terms, with a course assessment. However, these projects lack structure and are suboptimal in that they may oversimplify a topic or focus on a simpler but tangential kernel component, degrading the utility of the assignment. We address these issues by providing guidelines and rationale for designing new projects to achieve specific learning objectives, as well as multiple examples from five core topics of operating system concepts. The examples are carefully chosen to meet the objectives and ease students incrementally into the world of kernel programming.

5. CONCLUSIONS

We have developed a structured series of Linux kernel projects for teaching operating system concepts. The projects progressively introduce students to core topics and major components of operating systems, namely system calls and processes, synchronization, scheduling, virtual memory, and file systems. At the same time, they implicitly teach students about various aspects of a real-world operating system and managing the code complexity of a large software system. The projects require students to understand and leverage core components of the operating system while keeping the coding complexity modest. We presented guidelines and rationale for designing new projects to achieve specific learning objectives, and provide examples of eleven different projects that can be used.

Our experiences teaching over a thousand students in the introductory operating systems course at Columbia University demonstrate the pedagogical value of these kernel projects in practice. We have shown that the designed coding complexity of the projects is achieved as example project solutions are each under 400 lines of code and students' submitted solutions on average fall within the same range. We have also shown that common programming errors in students' submitted solutions become notably less frequent in subsequent projects as later projects reinforce the concepts taught in earlier projects, demonstrating that the projects provide an effective framework for progressive learning.

6. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-0905246, CNS-0914845, and CNS-1018355.

7. REFERENCES

[1] C. L. Anderson and M. Nguyen. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. *Journal of Computing Sciences in Colleges*, 21(1):183–190, 2005.

[2] A. Gaspar and C. Godwin. Root-kits & Loadable Kernel Modules: Exploiting the Linux Kernel for Fun and (Educational) Profit. *Journal of Computer Sciences in Colleges*, 22(2):244–250, 2006.

[3] R. Hess and P. Paulson. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *Proceedings of SIGCSE 2010*, Milwaukee, WI, Mar. 2010.

[4] O. Laadan, J. Nieh, and N. Viennot. Teaching Operating Systems Using Virtual Appliances and Distributed Version Control. In *Proceedings of SIGCSE 2010*, Milwaukee, WI, Mar. 2010.

[5] B. Lawson and L. Barnett. Using iPodLinux in an Introductory OS Course. In *Proceedings of SIGCSE 2008*, Portland, OR, Mar. 2008.

[6] J. Nieh and Özgür Can Leonard. Examining VMware. *Dr. Dobb's Journal*, pages 70–76, Aug. 2000.

[7] J. Nieh and C. Vaill. Experiences Teaching Operating Systems Using Virtual Platforms and Linux. In *Proceedings of SIGCSE 2005*, St. Louis, MO, Feb. 2005.