Experiences Teaching Operating Systems Using Virtual Platforms and Linux

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

Chris Vaill
Department of Computer Science
Columbia University
New York, NY 10027
cvaill@cs.columbia.edu

ABSTRACT

Operating system courses teach students much more when they provide hands-on kernel-level project experience with a real operating system. However, enabling a large class of students to do kernel development can be difficult. To address this problem, we created a virtual kernel development environment in which operating systems can be developed, debugged, and rebooted in a shared computer facility without affecting other users. Using virtual machines and remote display technology, our virtual kernel development laboratory enables even distance learning students at remote locations to participate in kernel development projects with on-campus students. We have successfully deployed and used our virtual kernel development environment together with the open-source Linux kernel to provide kernel-level project experiences for over nine hundred students in the introductory operating system course at Columbia University.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General; K.3.1 [**Computers and Education**]: Computer Uses in Education—*distance learning*; 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, virtualization, virtual machines, open-source software

1. INTRODUCTION

Programming projects are an important aspect of learning about operating systems. The hands-on experience is crucial for helping students to understand how the textbook concepts can be applied in practice. However, developing and administering programming projects that teach students about real-world operating system design and implementation is difficult for two important reasons.

First, unlike other application software which runs at user-level, operating system code runs in supervisor mode. To write or modify operating system code that runs in supervisor mode, students must be given root privileges to do many of the things required for kernel development such as installing a new kernel, rebooting, kernel testing and debugging, etc. In addition, the kernel development cycle of plan-implement-reboot-test-debug results in system downtime that necessitates exclusive access to a computer to avoid inconveniencing others. For a large introductory operating systems class, providing each student with a computer on which to run as root to do kernel development would be difficult to administer and prohibitively expensive.

Second, real production operating systems are large, complex pieces of software. The sheer size of these systems makes them difficult to understand and learn about. These operating systems are also not primarily designed as teaching tools, but instead to be used in commercial deployment. For example, such operating systems are already fully functional and do not have major missing subsystems that students can design and implement. These factors make it difficult to design projects for students that enable them to learn about interesting and important aspects of real-world operating system design.

To address these issues, we created a virtual kernel development environment in which operating systems can be developed, debugged, and rebooted in a shared computing lab environment without affecting other application users. Using virtual machines and remote display technology, our virtual kernel development laboratory enables even distance learning students at remote locations to participate in kernel development projects with local on-campus students. We have combined our virtual kernel development laboratory with a set of novel programming projects using the Linux kernel to provide real-world project experiences for students. These projects enable students to modify and replace substantial subsystems of Linux in a manner that provides many of the benefits of building new operating system subsystems from scratch. We leverage the benefits of the Linux open-source software base and widely available development tools to enable students to effectively manage the software complexity associated with a production operating system.

Both our virtual kernel development environment and pro-

http://doi.acm.org/10.1145/1047124.1047508

gramming projects build on production technologies that are widely used. Our virtual kernel development environments builds on virtual machines provided by VMware [9] and remote display mechanisms such as VNC [7]. Our programming projects are based on Linux. Building on production technologies enables us to leverage other large development efforts to support operating system education without the need to to maintain substantial pedagogical infrastructure on our own. We do not need to maintain our own simulators or pedagogical operating systems, which is difficult to do given the rapid pace of industry practice and the limited teaching resources available at most universities. Instead, our approach inherently keeps up and evolves as technology changes without requiring an in-house development effort.

We have deployed, refined, and used our virtual kernel development environment together with Linux kernel programming projects in teaching an introductory operating systems course at Columbia University. We were the first to use this approach for teaching operating systems [6], and have used it successfully over the last five years to teach over nine hundred students. These students range from sophomore undergraduates to doctoral students and include even distance learning students located halfway around the world. Our experiences using our virtual kernel development lab together with Linux demonstrate the utility of this approach for enhancing operating systems education by providing effective kernel-level project experiences for students.

This paper describes how we developed our approach to operating systems education and our experiences with it in the classroom. Section 2 discusses related work. Section 3 describes the development of our virtual kernel development environment. Section 4 describes the design of our Linux kernel programming projects. Section 5 discusses experiences using the virtual kernel development laboratory and the Linux kernel projects. Finally, we present some concluding remarks.

2. RELATED WORK

The two main approaches to providing programming experience in operating system courses can be loosely categorized as *user-level* or *kernel-level*. User-level projects require only developing code designed to run in unprivileged mode. Examples of such projects include writing modules for a user-level simulator such as Nachos [1], user-level threads programming, or systems programming with a production operating system such as Linux, Solaris or Windows XP. Operating systems courses with only user-level student projects are much more common because the projects are typically no more difficult to setup and administer than other user-level applications. While user-level projects do provide students with some hands-on experience with operating systems, they do not provide direct kernel-level development experience. As a result, user-level projects do not effectively address important issues such as bootstrapping, handling interrupts, the kernel-level development and debugging process, or understanding the kernel internals of a full-featured operating system.

Some operating system courses offer kernel-level programming projects. Kernel-level projects require writing or modifying code designed to run in supervisor mode. Kernel-level projects can provide a better pedagogical vehicle for learning about real-world operating system design and implementation. However, because of the complexity of production operating systems, such projects are typically based on a small pedagogical operating system like MINIX [8]. While MINIX can be setup and run on modest PC hardware, this would require providing every student with their own machine on which to do kernel development, which is impractical in most university settings and does not scale to large class sizes or supporting distance learning students effectively. Since pedagogical operating systems are small though, they can often be used with a machine simulator instead, such as Bochs [5]. One may even design a pedagogical operating system to be used only in a simulated environment [4]. At the same time, because pedagogical operating systems are smaller than production operating systems, developing code in one does not expose students to many of the real-world issues that arise in practice. Furthermore, because pedagogical operating systems are not used in practice, keeping them from becoming dated can require substantial effort and they may have more limited lifetimes due to changes in technology and operating system practice [4]. Examples of pedagogical operating systems that have become obsolete include Xinu [2] and TOY, which was originally developed by Brian Kernighan in 1973. In contrast, a production operating system such as UNIX predates both TOY and Xinu yet continues to be widely used today.

3. A VIRTUAL KERNEL DEVELOPMENT LABORATORY

To support kernel-level programming projects for students to learn about operating systems, what we would like to do is provide each student with effectively root access on a dedicated machine which the student can use to test kernel code and crash and reboot at will. However, we would like to do this without the expense and administrative difficulties of providing a dedicated physical machine for each student. Note that the key issue is testing and debugging kernel code, not writing kernel code. Many users can share the same machine to write and compile kernel code since that activity does not result in frequently crashing and rebooting a machine due to student programming errors.

Our solution was to use VMware [6, 9] virtual machine technology to create a virtual development platform that looks like a real machine but in fact is only a virtual one. We used VMware's Workstation product, a virtual machine monitor [3] for the x86 architecture. A virtual machine monitor is an additional layer of software between the hardware and the operating system that virtualizes all of the hardware resources of the machine to provide a virtual hardware execution environment called a *virtual machine* (VM). Multiple VMs can be used at the same time, and each VM provides isolation from the real hardware and other activities of the underlying system. Because it provides the illusion of standard PC hardware within a VM, VMware can be used to run multiple unmodified PC operating systems simultaneously on the same machine by running each operating system in its own VM. An operating system running as a user-level application on top of VMware is called a *guest OS*. The native operating system originally running on the real hardware is called the *host OS*. VMware provides a GUI as the visual interface to the VM which makes it look like a real computer from the moment the VM boots.

VMware is low-level enough to make a guest OS appear to be receiving hardware interrupts, such as timer interrupts,

and behave as if it were the only operating system on the machine. At the same time it provides isolation so that a failure in or misbehaving of a guest OS does not affect other guest OSes or the underlying system. For instance, a guest OS crashing will not crash the underlying system. As opposed to a software simulator, much of the code running in a VM executes directly on the hardware without interpretation. Only privileged instructions are trapped and impose additional overhead. A key advantage to using a VM as opposed to a simulator is the performance improvement possible through direct execution of unprivileged instructions.

We designed our virtual kernel development platform to be used in a shared computer lab facility. As is typical at many universities, such a facility is generally not available to be dedicated as an operating systems lab. VMware enabled us to configure a guest OS as a kernel development environment for a group of students working together on a kernel programming project. We could give students root access to guest OSes running in VMs without compromising the security of the lab machines. Since faults that occur in running the guest OS are contained within its respective VM, students could crash and reboot their guest OSes without interfering with the operation of the host. Students debugging their kernel could thus work in a shared computer lab facility and share the same computer as other students without disrupting the work of the other students.

To properly isolate VMs and their guest OSes from the underlying host machine, there are a few important VMware disk and network configuration options that need to be set properly. VMware allows the guest OS to mount raw disk partitions or use virtual disks. A virtual disk in VMware is simply a large file in the host OS file system that is treated by the VM as a disk. We configured the VMs to use virtual disks so that disk problems caused by a misbehaving guest OS that a student installed would not affect the host disk partitions. The use of virtual disks made it much simpler for students to re-install a guest OS in the event of a disk crash in the VM. Since a virtual disk is just a regular file on the host OS file system, we just provided clean versions of the virtual disk so that students could use them to overwrite their own in case of unrecoverable disk crashes in the VM. Restricted access to the virtual disks was achieved using user groups. Each virtual disk was owned by a different user group and permissions were set on the virtual disks so that they could only be accessed by the teaching staff or members of the respective user group. Students were each assigned to one user group.

The VM network configuration options determine the type of networking available to the guest OS. The options are no networking, host-only networking, and bridged networking. The no networking option does not export a network interface to the VM. The host-only networking option exports a network interface to the VM that only allows communication between the VM and the host machine. Under bridged networking, the host OS acts as a bridge between the VM and the LAN, effectively allowing the VM to run as a real networked machine with an IP address. To allow students to access their files on the host machine, we configured the VMs for host-only networking. Using host-only networking, students were able to use ftp to backup their work onto their regular home directories, which alleviated much of the problems of a virtual disk crash while working on an assignment. We did not provide full bridged networking to the VMs because of the security implications of allowing students to have root access on a full networked machine on a LAN.

Because VMs are virtual and there is no need to provide additional hardware in creating additional VMs, we could easily provide multiple VMs for each student. By design, our virtual development platform provides two VMs per student group, one which serves as a primary test VM and the other which serves as a backup test VM. Because students frequently crash their VMs when running their kernel code, it is not uncommon for a student to corrupt the virtual disk associated with the VM. Just like a real machine with a bad disk a corrupted virtual disk prevents a VM from booting. By providing a backup test VM for each student, a student can continue working with the backup test VM while the primary test VM is repaired by copying over a clean virtual disk and thereby restoring the VM to its initial starting configuration. However, any modifications stored on the corrupted virtual disk are lost.

Because of the likelihood that a VM can lose its persistent storage state, VMs are only used for testing and debugging, not actual kernel code development. On each host machine, we allocate a separate disk partition for each student group to use. That partition includes space for the student group's VM virtual disk as well as a development area in which to do kernel builds and write kernel code. All code developed by the students resides on the host machine itself, not within the VM. As a result, VM disk crashes and reinstalls do not cause any loss of kernel development work by the students. This separation of the development environment from the testing and debugging environment is critical for preventing students from losing their work.

Perhaps the most important advantage of using a VMware VM for kernel debugging is that a VM can be powered on and off with the click of a mouse as opposed to a physical machine which requires that its power be physically turned on or off. Machine problems that can only be fixed by power cycling are therefore much more convenient to fix when the machine is a VM. Furthermore, power cycling a VM can be done by a student who is not physically co-located with the respective host machine without the need for any specialized power management hardware.

Since distance learning students are increasingly common and many commuter students often work from off-campus, we augmented our virtual machine platform with remote display functionality to support remote kernel development and debugging. While VMware Workstation runs like a normal X application under Linux that can displayed its GUI on another machine via X, any extended loss network connectivity between machines would cause the X application to terminate, the equivalent of powering off the VM without properly shutting down the guest OS. This could result a corrupted virtual disk. We instead used VNC [7] to provide remote access to a VM because it allows a VM to continue running even when its display on another machine is interrupted due to loss of network connectivity. Using VNC, students did not have to compete for console access to use VMware. VNC could be used by distance learning and off-campus students in industry who could not access VMware using X because of corporate firewalls. VNC also provides screen sharing technology so that users can see and control the exact same screen on multiple machines, which made it much easier for students to collaborate in their project. This remote display functionality enabled us to teach operating

systems using the virtual kernel development environment to students at remote locations around the world.

4. LINUX KERNEL PROJECTS

Our virtual kernel development laboratory enabled us to provide kernel-level programming projects with either real production operating system or a pedagogical one. We chose to use Linux, a production operating system, for this purpose for eight important reasons. First, since Linux is used in the real world, it enables students to learn about real-world operating system issues that are difficult to glean from simplified pedagogical tools alone. Second, because Linux is open-source and widely used, there is a wealth of documentation and tools available to learn about the system. For example, there are a number of Linux source code navigators available that make it extremely easy to follow code through the system and search for various functions to understand how the system works. Third, there are many utilities such as kernel debuggers available for use with Linux which are of high quality because many people use them given the popularity of Linux. The same useful tools that are available to real kernel developers are available to students. Fourth, since Linux is immensely popular, students were more interested in doing the projects since they were working with something they felt was practical and relevant. Fifth, by applying operating system concepts to Linux, students gain skills in a real production system that can be immediately applied in the workforce after graduation. Sixth, by using a real production operating system, students gain experience dealing with a large, complex piece of software and understanding of how to read production code to figure out how a system works. Being able to manage software complexity is of tremendous importance in the real world given that operating system and other software developers spend much of their time working with others on large software projects, not developing isolated single-person systems. Seventh, as Linux evolves to keep up with the pace of innovation necessarily in production systems, it also naturally evolves as a pedagogical tool that enables students to learn in the context of modern operating system design trends. Finally, all of this support for Linux is provided without any need for us to maintain the operating system or any of its utilities, allowing us to focus limited teaching resources on teaching rather than a difficult in-house development effort that becomes outdated in less than a decade [4].

Because Linux is a production operating system, it does not naturally provide pedagogical opportunities by leaving out various operating system functionality for students to implement as is commonly done with pedagogical operating systems. However, many interesting aspects of important operating system subsystems can be designed in a variety of ways. Our approach to developing kernel programming projects for students is to create projects that allow them to add or replace existing operating system functionality. This provides two important advantages over the approach of leaving out parts of an operating system for students to fill in. First, the opportunities for projects are not fixed to the set of holes that were created at a given point in time which may not be the right set of projects for students to work on at a later time as technology evolves. Second, by replacing existing operating system functionality, students can learn by example from the design of existing code as written by real-world operating system developers.

Using this approach, we developed a number of kernel programming projects that can be used for an operating systems course to provide hands-on experience for students in understanding key operating system concepts. While discussing these projects in-depth here is not possible due to space constraints, we highlight five representative projects that we have developed corresponding to major operating system topics such as operating system structure, synchronization, scheduling, memory management, and file systems. Each assignment is designed to be done in two weeks or less by small student groups of two to three students.

As a first kernel project, we have students learn basic Linux kernel development and operating system structure. They first learn how to build a kernel and install and boot it. We then teach students how to apply patches and use a kernel debugger by downloading and applying the patch for the KDB kernel debugger for Linux. Students install the kernel with debugger support and walk through some simple debugging instructions. The final part of the assignment allows students to learn about operating system structure by adding a new system call that obtains some basic information about a process from its internal kernel structure. This assignment not only eases students into the sometimes intimidating process of kernel development, but it also illustrates the operation of a system call and its difference from a normal library function call.

As a second kernel project, we have students learn about synchronization through an assignment consisting of two parts. The first part involves a user-level POSIX-like threading implementation, similar to the GNU libc's *linuxthreads* library. Students are given an incomplete version of the library and a test-and-set function, and asked to implement mutexes, semaphores, and reader-writer locks. We start with synchronization in user-level to stress that the concepts are not specific to kernel programming, and in fact are necessary in any threaded programming environment. The second part involves implementing a new kernel synchronization primitive that allows multiple processes to block on an event until some other process signals the event. When the signal occurs, all processes blocking on the respective event are unblocked. This assignment exposes students to synchronization issues for both user-level thread libraries as well as kernel-level synchronization primitives.

As a third kernel project, we have students implement a new kernel CPU scheduler. This scheduler is called User-Weighted Round-Robin (UWRR), and operates by switching, round-robin, between users, giving each user's processes a full share of the CPU when it is that user's turn. 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. This is the first assignment that requires students to deal with a more substantial kernel subsystem but still only involves modification of a relatively modest number of lines of kernel code. The assignment builds on the previous two kernel programming projects as students need to write new system calls to control the UWRR scheduler and they need to be careful to protect kernel data structures with proper locking mechanisms.

As a fourth kernel project, we have students replace the stock Linux kernel's page replacement algorithm with the classic two-bit clock algorithm taught in operating system textbooks. This assignment requires students to learn in detail how the stock Linux kernel's page replacement mecha-

nisms are implemented and to understand the Linux paging system in reasonable detail. Because the virtual memory subsystem of an operating system is often its most complex part, our focus in this assignment is less on building a new virtual memory subsystem and more on demonstrating working knowledge of the stock kernel's memory subsystem. Once students understand the stock kernel well, implementing the two-bit clock algorithm is fairly straightforward and requires less than thirty lines of new kernel code.

As a fifth Linux kernel project, we have students implement a new access control list mechanism for the commonly used Linux ext2 or ext3 file systems. 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. In addition to learning about file systems, the project also gives students an opportunity to learn a bit about security issues as well.

5. EXPERIENCES

The experiences of both students and instructors with this approach to teaching operating systems have been very positive. We have taught the course in this manner for five years running, and enrollment has consistently increased. When we started this program, operating systems was not a required course in the computer science undergraduate curriculum at Columbia, yet enrollment increased 50% in just the first year. Since we started this approach to the course, we have had to double the number of sections taught, and the course has gone from a once-per-year offering to a staple course offered every semester to accommodate the large course enrollments. In Fall 2004, the course had the highest enrollments of any Computer Science course at Columbia.

The popularity of the course is attributable to what students see as its relevance to popular and modern real-world operating systems. 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 very useful to them after graduation. While our intent in teaching an introductory operating systems course is to convey understanding of general principles, and not to teach Linux kernel programming as such, our approach leverages the natural interest many students have in working with such a popular system. This has turned out to be a very powerful incentive—we have had students take the course as early as their sophomore year and perform extremely well. Not surprisingly, students are more willing to put forth extra effort to learn difficult material if they perceive that the material may be useful to them again after the course is over.

The virtual kernel development laboratory itself has also led to positive student experiences with the course. The virtual machine and VNC setup allows students to collaborate more easily without necessarily being physically in the lab. We have had students as far away as Japan take the course in a distance learning capacity. Our approach holds benefits for students that can work locally as well. The virtual machines reboot more quickly and are more easily recovered from kernel-bug catastrophes than physical machines are, leading to fewer frustrations for students.

The virtual kernel development laboratory has also been favorably received by machine administrators in our department's information technologies staff. Virtual hardware is not subject to failure and virtual machines for students mean that administrators do not need to deal with the difficulties attendant upon managing extra machines running potentially buggy kernels. If a student corrupts her virtual root disk, she merely gets a new copy—no administrator must reinstall a base operating system.

Our success with using virtual kernel development platforms and open-source Linux in teaching operating systems has prompted educators at a number of other universities to adopt our approach for their own operating system courses. These universities include both top-tier and smaller schools, demonstrating the viability of our approach for a variety of educational settings.

6. CONCLUSIONS

We have developed a virtual kernel development platform that enables kernel-level projects to be conducted by students in shared computer lab facilities without affecting other application users. We have used this platform together with kernel programming projects in the Linux kernel to teach students about important operating system concepts in conjunction with real-world operating system design issues. We have used this approach to teach a wide range of students, including sophomore undergraduates, doctoral students, and distance learning students located in distance countries around the world. Our experiences in deploying this approach to teach more than nine hundred students have demonstrated the effectiveness of learning about real production operating system kernel development using virtual platforms. We hope our experiences can continue to serve as a basis for improving operating system education at other institutions as well.

7. ACKNOWLEDGMENTS

This work was supported in part by an NSF CAREER Award and NSF ITR grant CNS-0426623.

8. REFERENCES

[1] W. Christopher, S. Proctor, and T. Anderson. The Nachos Instructional Operating System. `http://http.cs.berkeley.edu/~tea/nachos/nachos.ps`.

[2] D. E. Comer. *Operating Systems Design: The XINU Approach*. Prentice-Hall, 1984.

[3] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[4] D. A. Holland, A. T. Lim, and M. I. Seltzer. A New Instructional Operating System. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 111–115, Feb. 2002.

[5] K. Lawton. Bochs. `http://bochs.sourceforge.net/`.

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

[7] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[8] A. Tanenbaum. A UNIX Clone with Source Code for Operating Systems Courses. *Operating Systems Review*, 21(1):20–29, Jan. 1987.

[9] VMware. `http://www.vmware.com/`.