Teaching Virtualization by Building a Hypervisor

Abhinand Palicherla
Tintri, Inc.
303 Ravendale Dr.
Mountain View, CA 94043
apalicherla@tintri.com

Tao Zhang
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
zhtao@cs.stonybrook.edu

Donald E. Porter
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794
porter@cs.stonybrook.edu

ABSTRACT

Virtual machines (VMs) are an increasingly ubiquitous feature of modern computing, yet the interested student or professional has limited resources to learn how VMs work. In particular, there is a lack of "hands-on" exercises in constructing a virtual machine monitor (VMM, or hypervisor), which are both simple enough to understand completely but realistic enough to capture the practical challenges in using this technology. This paper describes a set of assignments to extend a small, pedagogical operating system (OS) to form a hypervisor and host itself. This pedagogical hypervisor, called HOSS, adds roughly 1,000 lines of code to the MIT JOS source, and includes a set of guided exercises. Initial results with HOSS in an upper-level virtualization course indicate that students enjoyed the assignments and were able to apply what they learned to solve different virtualization-related problems. HOSS is publicly available.

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

Keywords

Operating Systems; Virtual Machines; Instructional Tools

1. INTRODUCTION

Virtual Machines (VMs) are an increasingly important component of modern computing. VMs offer several attractive benefits, including security isolation, the ability to run applications designed for different operating systems (OSes) on the same hardware platform, and the ability to migrate applications from one physical machine to another. As a result, VMs have become the backbone of cloud computing, grown increasingly common on desktops, and are even emerging as a way to separate business and personal applications and numbers on a single mobile phone [4, 14].

Unfortunately, there are very few tools available to help students and instructors learn how virtualization works, especially "hands-on" exercises. A few OS textbooks have added overviews of virtualization [29, 33], but there is very little between these very high-level descriptions and the very low-level details of the ISA programming manual, technical research papers [7, 11] and other references [12, 30]. Although open source hypervisors are widely used, Xen [7] is hundreds of thousands of lines of code, and KVM is scattered across 15 million lines of Linux source code. The interested student lacks a "middle ground" environment that is simple enough to completely understand, but realistic enough to appreciate the real-world challenges of working with this technology.

This paper describes **HOSS**, a simple OS that can host instances of itself as virtual machine guests. In the HOSS exercises, students directly program Intel's hardware virtualization extensions (called VT-x) [34], create extended page tables, handle hypercalls and other traps, as well as implement a paravirtual guest system. To build HOSS, we extended the MIT JOS [20] pedagogical OS. In both JOS and HOSS, boilerplate code is provided, and students implement the interesting components of the system. The provided code is small enough to understand (about 14,000 LoC), but structurally similar to much larger OS codebases, such as Linux. HOSS adds roughly 1,000 LoC to JOS.

Initial experiences with HOSS in the classroom are positive. The HOSS assignment was initially assigned in a new course on virtualization, including a mix of senior undergraduate and graduate students. After completing the HOSS assignments, students were able to complete an open-ended final project that involved modifying a production hypervisor. The HOSS assignments are publicly available at `http://www.cs.stonybrook.edu/~porter/hoss` and solutions are available to instructors upon request. Enhancements and support for HOSS are ongoing (§6).

2. RELATED WORK

To our knowledge, HOSS is the first pedagogical hypervisor. In addition to JOS, a number of pedagogical OSes have been developed and used [6, 10, 13, 19, 24, 27, 32], but these OSes do not include exercises in hypervisor construction. This paper demonstrates that adding a hypervisor to such a pedagogical OS is straightforward—adding roughly

SIGCSE'15, March 4–7, 2015, Kansas City, MO, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2966-8/15/03 ...$15.00.
http://dx.doi.org/10.1145/2676723.2677254.

one thousand lines of additional code. Northwestern has a course which involves adding features to a research hypervisor [16], but the coding assignment is an open-ended project, not structured assignments implementing core hypervisor features.

Lguest is perhaps closest to HOSS—a simple, but functionally limited hypervisor for Linux [28]. Lguest is written primarily to document and test Linux-internal interfaces for writing a hosted hypervisor; KVM also uses these internal interfaces. Lguest would be useful in the classroom as a code review assignment, but it would be difficult to create assignments that worked with core virtualization concepts that did not amount to reimplementing existing code.

A number of OS educators have shifted to using real-world OSes, such as Linux, in the classroom [3, 18, 23]. To our knowledge, published experiences and exercises have not introduced virtualization concepts. Andrus and Nieh [5] argue for the use of Android in the OS classroom because it is an increasingly ubiquitous technology with unique characteristics; we believe this argument applies equally to hypervisors.

Finally, we note that several papers have been published on the effectiveness of virtual machine platforms as *classroom infrastructure*—scaling limited hardware to large classrooms, simplifying administrative tasks, and addressing other classroom challenges [15, 21, 22, 25, 31]. HOSS helps students learn how such useful technology works.

3. BACKGROUND

This section summarizes background on the JOS operating systems courseware, as well as concepts unique to virtual machine monitors (VMMs, or hypervisors) compared to a traditional OS kernel.

3.1 JOS

The JOS [20] operating system is written in C and assembly, and runs on 32-bit x86 (PC) hardware. A completed JOS solution is about 14,000 lines of C code and 600 lines of assembly[1]. Over the course of 6 assignments, students complete substantial portions of core system components, including the memory manager, scheduler, file system, shell, and network driver. In particular, JOS requires students to program x86 hardware including configuring the page tables, context switching the CPU, handling interrupts, and writing an e1000 network card driver.

The JOS architecture is best described as an exokernel [17], wherein the OS kernel manages hardware and exports hardware-like interfaces to the application. Unlike a monolithic kernel, most familiar POSIX-style abstractions, such as file descriptors or even `fork` and `exec` are implemented in an application *library OS*. The only exception is the file system, which runs as a shared, microkernel-style daemon.

An essential feature of JOS is that students develop intuitions which apply to much larger OS code bases, such as Linux or FreeBSD. Although JOS supports fewer architectures and features than a production-quality OS, it retains a similar kernel structure and requires students to write core system components. JOS was developed at MIT, and has been used at a number of universities, including Stanford, Texas, UCLA, Washington, Harvard, and Stony Brook.

[1]We note that JOS also includes a 25,000 LoC user-level socket library, but students do not need to understand this code to complete the assignments.

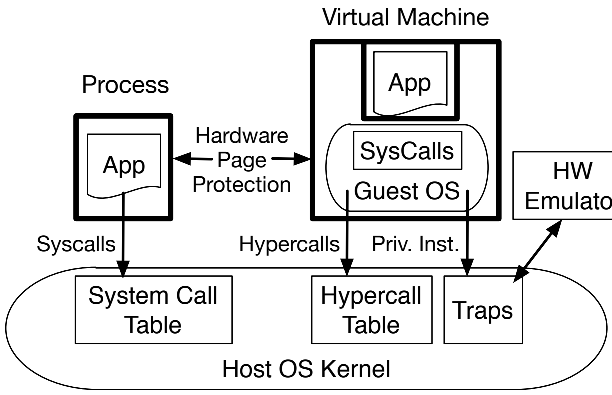

Figure 1: Comparison of an application in a process (left) to a virtual machine (right). Both execute in an isolated address space, implemented with hardware page tables. The primary difference is the ABI—system calls versus hypercalls and emulated hardware. A host OS kernel can execute a mixture of processes and VMs.

3.2 Hypervisor Implementation Techniques

Virtual machines are, in many respects, quite similar to OS processes (illustrated in Figure 1). VMs execute in a hardware-isolated virtual address space, and at a lower privilege level than the host OS kernel. The primary difference is the application binary interface (ABI) exported by the host OS kernel to the guest. The process ABI includes higher-level abstractions, such as file descriptors, signals, and network sockets, whereas a VM manipulates emulated hardware abstractions, such as a virtual disk, CPU interrupts, and a virtual network card. In other words, adding VM support to a legacy OS, such as adding KVM to Linux, requires replacing the system call table with a hypercall table, the ability to trap accesses to privileged hardware, and a PC hardware model, generally running as a separate process. As a result, desktop and server OSes can run both traditional processes and VMs (this is sometimes called a Type 2, or hosted hypervisor); a Type 1 hypervisor, such as VMware ESX server or Xen, runs on bare metal and only executes VMs.

We summarize three essential concepts specific to modern hypervisor implementations, which are not required to understand a traditional OS kernel.

Trap-and-emulate. VMs run a potentially-unmodified guest OS in a lower hardware privilege level (or ring in x86 parlance), yet include privileged instructions that directly access hardware. Hypervisors must be able to trap each of these privileged instructions, emulate its effect, and then advance the CPU's program counter transparently to the guest OS. We note that, before the introduction of Intel's VT extensions [34], the x86 architecture did not trap on all privileged instructions, but failed silently on some, rendering it unvirtualizable. Thus, VMware Workstation and other early hypervisors required binary translation techniques to detect these instructions [11].

Once a privileged instruction is trapped, the instruction's effects must be emulated. For instance, sending a packet on a virtual network card should ultimately result in a packet being sent on the physical network (or to another VM within a virtual network). As a result, most hypervisors also require

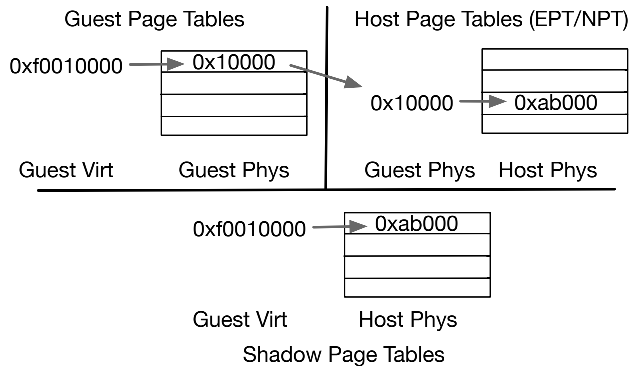

Figure 2: Both a guest and host OS include page tables, which translate virtual to physical addresses, or the guest's view of physical memory to the hosts' view of physical memory. Newer hardware directly traverses both page tables using extended, or nested page tables (EPT/NPT). For older hardware with only one level of paging, the hypervisor must construct a shadow page table (bottom), that directly translates guest virtual to host physical addresses.

models of common PC hardware, such as a widely-supported network card, disk controller, and video device. KVM obtains this from the qemu machine emulator [8], which runs as a process on the host OS. VMware Workstation has a similar process, called `vmx`.

Memory Translation. A VM runs in a virtual address space, similar to a process, but then creates nested virtual address spaces for each application. This creates a problem for CPUs that only export one level of page translation hardware, requiring the hypervisor to combine two levels of page translation into a single *shadow page table* for the hardware to use. Shadow page tables are illustrated in Figure 2, which directly map guest virtual addresses onto host physical addresses. Emulating paging hardware can be particularly expensive and complex because the hypervisor must trap every write to the memory containing the guest OS's page tables, and then update the shadow page tables.

To reduce this complexity and improve performance, recent Intel and AMD CPUs now support two levels of hardware page tables [1], allowing the hypervisor and VM to independently manage their own address translations. We note that, even with two-level paging, running a hypervisor inside of a VM will still require shadow paging (e.g., to map three page tables onto two) [9]. Although conceptually similar, hypervisor-level page tables and CPU control structures are not identical to traditional page tables. Even if a student has written context switching code for an x86 OS, context switching into and out of a VM requires understanding different control structures and a different control flow. For instance, returning from a system call leaves the kernel stack empty, whereas entering a VM leaves frames on the kernel stack.

Paravirtualization and hypercalls. Paravirtualization makes small modifications to a guest OS so that particularly expensive hardware emulation can be replaced with simpler software abstractions, such as replacing an emulated disk controller with a simple software ring buffer [7]. The hypervisor must export these abstractions to the guest using a *hypercall* interface (similar to an OS kernel's system call table), and the guest OS must be modified to invoke these hypercalls rather than using privileged instructions to modify hardware directly.

Intuitively, paravirtualization improves performance by reducing the total number of traps to the hypervisor, as well as reducing the overhead of emulating specific hardware. The changes required of the guest OS are simple, small, and often encapsulated in dynamically loaded device drivers. We observe that modern paravirtual hypervisor designs share many similarities with an exokernel: exporting an interface to applications that either directly accesses hardware features where safe, or using system calls that export low-level, hardware-like abstractions to the process. As a result, the conceptual and implementation gap between supporting an exokernel-style library OS and a paravirtualized guest OS is relatively small.

4. HOSS EXERCISES

Similar to JOS, course staff provided students with HOSS "skeleton code", and students implement the remaining portions of the hypervisor. Students also make changes to the guest code to support paravirtualization. Both the JOS guest and HOSS host share a code base; all guest or host-specific code is delimited with C preprocessor macros (e.g., `#ifdef VMM_GUEST`). In order to increase the virtual address space for the host system, we ported JOS to the 64-bit x86 architecture; we omit the details of the 64-bit port, as these details are orthogonal to the hypervisor.

Students completed the assignments using the bochs PC emulator, which can emulate Intel's VT-x extensions. Using bochs eliminated the need for each student to have access to physical hardware, removing artificial constraints on enrollment. Moreover, bochs allows the student to connect a debugger directly to the running hypervisor or guest OS, facilitating debugging compared to bare metal.

The HOSS exercises reflect essential lessons, including the principles of hypervisor construction summarized in §3.2, which we explain in the rest of this section.

4.1 A VM is (almost) a Process

In HOSS, each process (`struct Env`) includes a flag that indicates whether it is a normal process or a VM. Most of the process-management code works unmodified. The two pieces of VM-specific code students must implement are for memory management and context switching.

Students must write code that creates and manages extended (or nested), hypervisor-level page tables. The processor expects a slightly different layout in these page tables than in a single-level page table; we provide the students with helper functions and pointers to the Intel documentation to manipulate these structures. Moreover, at this point, the students have a small, working implementation of guest-level page tables, so they have a related reference implementation. The kernel data structures to manage allocated and free memory are unchanged. When a virtual page mapping is created for a process, HOSS uses the VM flag to determine whether to call the handler written for extended or single-level page tables.

Students must also write context switching code that uses the `vmlaunch` and `vmresume` instructions. This component introduced the most differences. When an application traps to a traditional OS kernel, an x86 CPU transfers control to a handler specified in the interrupt descriptor table (IDT) and pushes the application's registers onto an address specified by the kernel. By ubiquitous convention, this is an empty stack frame; when the kernel finishes handling an interrupt, the final return (`iret` instruction) pops the registers and returns to the application. In other words, between any two interrupts or system calls, there is no active state on the kernel stack. In contrast, Intel's VT instructions maintain live state on the stack when context switching into a VM. Students had to write code that compensated for this difference by properly switching to a different stack when switching between multiple VMs.

Beyond these complication with context switching, students also had to understand the fields of the VM control structure (VMCS)—a region of physical memory where guest and host registers are saved and restored, as well as a structure that includes flags controlling VM behavior, such as which events cause traps. Students wrote short inline assembly code to detect whether the CPU supports VT and extended page tables, as well as context switching into and out of the VM. As a result, students gained experience working with these hardware abstractions.

4.2 The Guest Kernel is Just a Program

Students are asked to take an ELF loader, for loading application binaries into a process, and convert this to an ELF loader that launches a guest kernel in a VM. Students discover that this requires very few changes, except emulating the behavior of a PC BIOS, such as loading the bootloader at an expected address in memory and communicating the layout of (virtualized) physical memory to the guest kernel.

4.3 Hypercalls, Traps, and Emulation

We export the JOS system call table to the guest kernel as hypercalls in HOSS. Students implement the code that handles traps from the VM and redirects hypercalls to the system call table.

Students also get an introduction to hardware emulation by implementing the `cpuid` instruction. This instruction places information about the CPU's capabilities in several registers, according to a format defined by the CPU vendor. Most OSes use this instruction during boot for runtime configuration. The VMCS in HOSS is configured to trap on a `cpuid` instruction. The students write a trap handler that places appropriate flags in the correct registers and manually advances the program counter before returning control to the VM. In practice, this feature is often used to hide the presence of features such as extended paging from a guest VM when a hardware capability is already in use and software emulation is not available.

4.4 Paravirtualization

Finally, students implement a paravirtual disk. The baseline JOS code has a file system daemon, which issues hardware-level commands directly to an IDE disk controller. In the HOSS lab code, this is placed in a host-only preprocessor macro, and students are asked to instead request blocks of a disk image file from the host. These disk image file requests are issued using host-level IPC, as illustrated in Figure 3. The host-level file system transparently handles requests for the disk image file just like any other file. This design does require the guest-level IPC mechanism to translate guest virtual to guest physical addresses. Representing a hard drive as a disk image file is extremely common, and this gives students a clear understanding that a hard drive can easily be represented as a file.

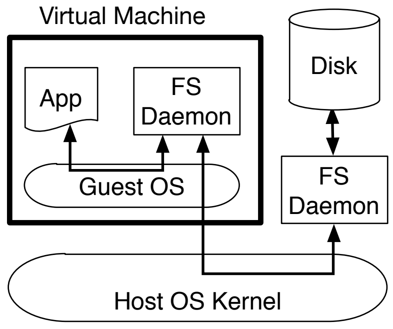

Figure 3: HOSS paravirtualized file system architecture. Both guest and host kernel execute a microkernel-style file system daemon. The guest FS daemon uses a hypercall to issue an IPC to the host daemon to access its disk image. The host FS daemon accesses the disk directly.

5. THE VIRTUALIZATION COURSE

HOSS was used as part of a new course on virtualization for senior undergraduates and graduate students. The course included two undergraduates, nineteen M.S. students, and four doctoral students, for a total of 25 students. All students in the course had completed an undergraduate OS course, and some had completed a graduate OS course.

In addition to the HOSS assignments, students read seminal and recent research papers on hypervisor design, virtual I/O, and security. About three quarters of class sessions were discussions about the assigned research papers, and one quarter were in-class programming days to help students work the assignments. Paper discussions spent equal time on understanding the technical details of how a prototype system was constructed (and its relevance to the HOSS assignment), as well as critically assessing the merit of the research idea and its evaluation.

The HOSS assignments were completed about halfway through the semester, and the remainder of the semester was spent on open-ended final projects. The final projects could be research-oriented or development tasks, and could be extensions to HOSS or any other open-source hypervisor. Students were allowed to work the HOSS assignments and final project in self-selected teams of up to three people, although some students chose to work alone.

5.1 Final Projects

One underlying goal of the final project is to evaluate whether students can apply what they have learned to new projects or codebases. This is a somewhat noisy, but mostly positive indicator, and cases that were not clear successes indicate further gaps in the instructional tools available (§6).

Here we list a sample of final projects and outcomes:

- **Adding suspend and resume support to lguest.** Lguest is a simple hypervisor written for a Linux host, using the same kernel-internal interfaces as KVM [28]. The undergraduate students were able to independently read,

understand, and extend the lguest source code with suspend and resume support.
- **Virtual Networking.** Several groups added virtual networking support to HOSS, but with different strategies. In general, most successful groups adopted a paravirtual strategy, which exposed a new interface for packet delivery to the guest via ring buffers, and wrote a new guest driver using these paravirtual interfaces.
- **Better VM benchmarks.** Several students felt that virtualization research would benefit from standardized microbenchmarks. These teams successfully added timing and other instrumentation to benchmark KVM.
- **Multi-Processor Support.** One team developed multiprocessor support for HOSS, such that multiple guests can run concurrently on different cores. This team was delayed somewhat by an underlying bug in JOS's support for multiple processors, which they fixed. By the end of the semester, the host kernel supported multiple CPUs, but guests were still single-CPU.
- **AMD Virtualization Hardware.** One student tried to port HOSS from Intel's VT to AMD's SVM equivalent [1]. Intel and AMD expose comparable, but different interfaces and abstractions for virtualization support. Unfortunately, no emulator exposes the AMD instructions, and the student worked directly on bare metal. Without the ability to attach a debugger to the system, and lacking debugging information from the emulator itself, this project proved too ambitious for the time frame.

In general, these results indicate that students understood the architecture of the system, and were able to solve new problems and work with new code bases.

5.2 Student Feedback

At the end of the course, students were given both an anonymous, university-issued survey, as well as the opportunity to give feedback directly to the instructor. Numerically, the course evaluation scores were positive: overall a 4.65 out of 5, compared to a 4.34 university average. However, the qualitative comments from both surveys tended to be more informative. We observe the following trends:

- The course contained a mix of students who had previously completed JOS and those who had not. The students who did not complete JOS previously felt that they needed to complete most of the previous assignments to fully understand the code, even when given a solution.
- Comments about HOSS itself were quite positive. Students repeatedly commented that they learned a lot and found the exercises worthwhile. Students generally requested more HOSS exercises, which we will add in future versions of the course.
- Most students enjoyed learning to read research papers, but a minority prefer a textbook-and-lecture style course. Most found a useful connection between programming assignments and the higher-level performance and functionality analysis of the research papers.

Based on this feedback, HOSS's hands-on approach to learning about virtualization was effective, and students generally wanted more system implementation exercises.

6. FUTURE WORK

HOSS is an ongoing effort, which we will use in future courses. We plan to add the following features to HOSS:

- **More exercises with device emulation in software.** A little device emulation goes a long way, as this task quickly becomes tedious. We envision providing a mostly-complete model of a disk or network card, and then asking students to add missing feature and use the model to implement trap-and-emulate behavior.
- **Execute an unmodified Guest OS.** The current JOS guest requires light modifications, primarily to avoid such emulation. We ultimately want to be able to support an unmodified JOS guest, as well as a mainstream OS, such as Linux or Windows.
- **Add binary translation for systems without virtualization hardware.** For older x86 systems, binary translation was necessary to properly catch and replace certain privileged instructions. We plan to add exercises that program a binary translator, such as the one used internally by qemu.

Most of these tasks require relatively straightforward engineering effort, and exercises using these features can be kept manageable for students. The current HOSS assignments cover the essential concepts, but the ability to boot an unmodified binary kernel within a semester would make the project more attractive for students.

We would also like to provide exercises programming recent IOMMU hardware that allows direct device management from a VM [2] and network cards that can be exported as multiple virtual NICs [26]. Unfortunately, current hardware emulators do not support this feature. In fact, most emulators and simulators do not include virtualization extensions at all. In our experience, students can grasp concepts and solve problems more quickly with familiar debugging tools—tools that are harder to support on bare metal. Thus, an important direction for future work is simply building emulator infrastructure for teaching and debugging.

A substantial limitation of this work is that it is part of an advanced, two-course sequence. An important direction for future education research is to develop a stand-alone variation of the course or to adapt components of this work to exercises in isolation, emulation or performance analysis for a core undergraduate systems course.

7. CONCLUSION

HOSS gives students hands-on experience with hypervisor construction. These exercises target an advanced, two-course OS/virtualization sequence. The exercises are suitable for an advanced undergraduate or graduate student, and yield intuitions that are applicable to new problems and other hypervisors. Our initial experiences with HOSS are positive, although there is room for improvement in the assignments as well as current emulation tools. Our assignment code and exercises are publicly available at `http://www.cs.stonybrook.edu/~porter/hoss` and solutions are available to instructors upon request.

8. ACKNOWLEDGMENTS

We thank the many contributors to JOS for creating a wonderful building block for HOSS. We thank Michael Tashbook and the anonymous reviewers for insightful comments on earlier drafts of this paper. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, and the Office of the Vice President for Research at Stony Brook University.

References

[1] AMD. AMD-V Nested Paging. White Paper, AMD: http://developer.amd.com/assets/NPT-WP-1\%201-final-TM.pdf, July 2008.

[2] AMD. AMD I/O Virtualization Technology (IOMMU) Specification Revision 1.26. White Paper, AMD: http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf, Nov 2009.

[3] C. L. Anderson and M. Nguyen. A survey of contemporary instructional operating systems for use in undergraduate courses. *J. Comput. Sci. Coll.*, 21(1):183–190, Oct. 2005.

[4] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *SOSP*, pages 173–187, 2011.

[5] J. Andrus and J. Nieh. Teaching operating systems using android. In *SIGCSE*, pages 613–618, 2012.

[6] B. Atkin and E. G. Sirer. Portos: An educational operating system for the post-pc environment. In *SIGCSE*, pages 116–120, 2002.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[8] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC*, pages 41–41, 2005.

[9] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and implementation of nested virtualization. In *OSDI*, 2010.

[10] D. Brylow. An experimental laboratory environment for teaching embedded operating systems. In *SIGCSE*, pages 192–196, 2008.

[11] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, Nov. 2012.

[12] D. Chisnall. *The Definitive Guide to the Xen Hypervisor.* Prentice Hall Press, 1st edition, 2007.

[13] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *USENIX Winter Conference*, pages 4–4, 1993.

[14] S. Cooper. Vmware android handset virtualization hands-on. Retrieved Sept. 1, 2014 from http://www.engadget.com/2011/02/15/vmware-android-handset-virtualization-hands-on/, 2011.

[15] R. Davoli and M. Goldweber. Virtual square (v2) in computer science education. In *ITiCSE*, pages 301–305, 2005.

[16] P. Dinda. Eecs 441: Resource virtualization, winter 2014. Retrieved Sept. 1, 2014 from http://pdinda.org/virt-w14/.

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995.

[18] R. Hess and P. Paulson. Linux kernel projects for an undergraduate operating systems course. In *SIGCSE*, pages 485–489, 2010.

[19] D. A. Holland, A. T. Lim, and M. I. Seltzer. A new instructional operating system. In *SIGCSE*, pages 111–115, 2002.

[20] JOS OS Lab. Retrieved Sept. 1, 2014 from http://pdos.csail.mit.edu/6.828.

[21] M. Ketel. A virtualized environment for teaching IT/CS laboratories. In *SE*, pages 92:1–92:2, 2010.

[22] O. Laadan, J. Nieh, and N. Viennot. Teaching operating systems using virtual appliances and distributed version control. In *SIGCSE*, pages 480–484, 2010.

[23] B. Lawson and L. Barnett. Using ipodlinux in an introductory os course. In *SIGCSE*, pages 182–186, 2008.

[24] H. Liu, X. Chen, and Y. Gong. Babyos: A fresh start. In *SIGCSE*, pages 566–570, 2007.

[25] J. Owens. Using virtualization to teach Linux system administration in online courses. In *IBM ICVCI*, 2007.

[26] PCI-SIG. Address translation services 1.1 specification. http://www.pcisig.com/members/downloads/specifications/iov/ats_r1.1_26Jan09.pdf.

[27] B. Pfaff, A. Romano, and G. Back. The pintos instructional operating system kernel. In *SIGCSE*, pages 453–457, 2009.

[28] R. Russell. Lguest: The simple x86 hypervisor. Retrieved Sept. 1, 2014 from http://lguest.ozlabs.org.

[29] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* 9th edition, 2012.

[30] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann Publishers Inc., 2005.

[31] W. Sun, V. Katta, K. Krishna, and R. Sekar. V-NetLab: An approach for realizing logically isolated networks for security experiments. In *CSET*, pages 5:1–5:6, 2008.

[32] A. S. Tanenbaum. A unix clone with source code for operating systems courses. *OSR*, 21(1):20–29, Jan. 1987.

[33] A. S. Tanenbaum and H. Bos. *Modern Operating Systems.* 4th edition, 2014.

[34] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, lain Kägi, F. H. Leung, and L. Smith. Intel virtualization technology. In *IEEE Computer*, pages 48 – 56, 2005.