The Pintos Instructional Operating System Kernel

Ben Pfaff
Nicira Networks
Palo Alto, CA
blp@nicira.com

Anthony Romano
Stanford University
Palo Alto, CA
ajromano@stanford.edu

Godmar Back
Virginia Tech
Blacksburg
gback@cs.vt.edu

ABSTRACT

Pintos is an instructional operating system, complete with documentation and ready-made, modular projects that introduce students to the principles of multi-programming, scheduling, virtual memory, and filesystems. By allowing students to run their work product on actual hardware, while simultaneously benefiting from debugging and dynamic analysis tools provided in simulated and emulated environments, Pintos increases student engagement. Unlike tailored versions of commercial or open source OS such as Linux, Pintos is designed from the ground up from an educational perspective. It has been used by multiple institutions for a number of years and is available for wider use.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Design, Experimentation

Keywords

Pintos, instructional operating system, instructional kernel

1. INTRODUCTION

Despite the wide use of higher-level languages and environments, gaining a robust understanding of operating systems (OS) fundamentals and training in the current design and implementation practices of OS remains a cornerstone of undergraduate computer science education.

Approaches to teaching OS courses generally fall along two axes: whether the treatment of the material is abstract or concrete [12], and whether they adopt an internal or external perspective [8]. An abstract approach discusses algorithms and techniques used in operating systems and may include partial implementation or simulation exercises, whereas a concrete approach stresses the design and creation of realistic artifacts. When adopting the internal perspective, an operating system is considered from the point of view of the OS designer, whereas the external perspective assumes the role of a user or programmer using an OS's facilities [5].

The approach advocated in this paper is concrete and adopts the internal perspective. Students who have studied, implemented, and evaluated core OS techniques attain a deeper understanding than those who have merely studied them. Finally, adopting a concrete approach brings significant secondary benefits, including training in modern software development techniques and tools. The C language remains the implementation language of choice for operating system kernels and for many embedded systems. Practice and debugging experience in C, particularly using modern tools, not only increases students' "market value," [9] but provides students with the insight that a low-level programming and runtime model is not incompatible with high-level tools.

Designing course material for the internal and concrete approach is challenging. While realistic, assignments should be relatively simple and doable within a realistic time frame. Whereas assignments should use current hardware architectures, they must not impart too much transient knowledge. Assignments should include and emphasize the use of modern software engineering practices and tools, such as dynamic program analysis.

This paper introduces Pintos, an instructional operating system kernel that has been in use at multiple institutions for about 4 years. Pintos provides a bootable kernel for standard x86-based personal computers. We provide four structured assignments in which students implement a basic priority scheduler, a multi-level feedback queue scheduler, a process-based multi-programming system, page-based virtual memory including on-demand paging, memory-mapped files, and swapping, and a simple hierarchical file system. An overview of the projects enabled by Pintos is given in Figure 2, which shows which software is provided as support code and test cases, the parts that are created by students, and their relationship.

Although Pintos follows in the tradition of instructional operating systems such as Nachos [6], OS/161 [11], and GeekOS [12], PortOS [2], BLITZ [15], JOS [1], or Yalnix [1], we believe that it is unique in two aspects. First, Pintos runs on both real hardware and in emulated and simulated environments.[1] We believe that having the ability to see

SIGCSE '09 Chattanooga, Tennessee, USA

[1] GeekOS is the only other system that claims to also run on real PC hardware; it requires, however, a dedicated disk and does not

the outcome of their project interact with actual hardware increases student engagement. Second, we have created a set of analysis tools for the emulated environment that allows students to detect programming mistakes such as race conditions. Figure 1 shows the three environments in which the same kernel can be run.

Others have used Linux, either on dedicated devices (e.g., iPodLinux [13]), or in virtualized environments [7, 10, 14], to provide an internal, concrete perspective. Compared to those approaches, Pintos provides a similar level of realism in that students can see the results of their work on concrete or virtualized hardware, but does not require that students understand the often arcane and ill-documented interfaces of the Linux kernel, which were not designed from an educational perspective. By contrast, all Pintos code is written to be studied by students.

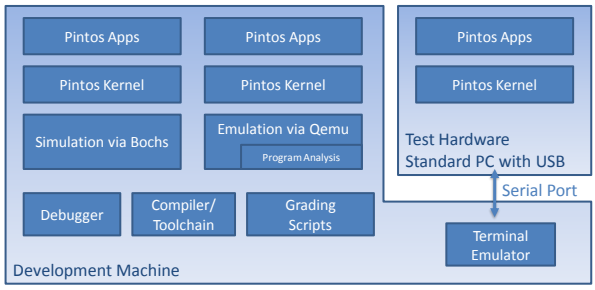

Figure 1: The same Pintos instructional kernel runs in a fully reproducible simulated environment, in an enhanced emulated environment with dynamic analysis capability, and on actual hardware.

2. DESIGN PRINCIPLES

Pintos's projects are built on a number of principles.

Read before you Code.

Each project involves a significant amount of reading code before students write the first line of code. Because software maintenance constitutes the vast majority of all software development efforts [4], this setup mirrors the environment in which most software engineers work. Simultaneously, we limit the amount students have to read by encapsulating lower layers, such as device drivers. We went to great lengths to write the entire Pintos baseline code, and in particular the portions students must read, in a style that shows, by example, the coding style we expect from students. We continuously refined the internal code documentation over several semesters, focusing on those portions that initially proved difficult to understand.

Maximize Creative Freedom.

OS design involves a tremendous amount of creative freedom, both in the choice of algorithms and data structures. Our projects are designed to stimulate creativity by avoiding the prescription of specific approaches to accomplish each project's goals. Instead, students design their own data structures and associated algorithms as much as possible.

support running off USB devices, making it impractical for many laboratory settings.

Project	Functionality	Robustness	Regression
1	27	-	-
2	41	35	-
3	20	14	75
4	39	7	75

Table 1: Pintos test cases by project.

Practice Test-driven Development.

Each project includes a large number of test cases that are accessible to students, as shown in Table 1. They must implement the API that is exercised by these test cases. Students are encouraged to add their own test cases.

Work in a Team.

The projects presented in this paper are designed to be accomplished by teams of 2-4 students. Working in a team provides an environment that more closely resembles industrial software development, and it provides a way for students to brainstorm and implement together. In addition, we teach and require the use of group collaboration tools, notably shared source code version control systems such as CVS.

Justify your Design.

Design justification and rationale is as important for learning as creating an artifact that fulfills a set of given requirements. We designed a set of structured questionnaires in which students describe their design and discuss choices and trade-offs they made.

Provide a Reproducible, Manageable Environment.

Operating Systems are inherently concurrent environments, which can be difficult to debug. For educational use, we must provide an environment that is manageable and reproducible, which we do by providing the option of running Pintos in a simulated, fully deterministic environment. As a result, Pintos kernels can be debugged in a manner that is substantially similar to how user programs are being debugged.

Include Analysis Tools.

Dynamic analysis tools are now being widely used in software development; an OS course should be no exception. In Section 4, we describe how we extended the QEMU emulator [3] to perform tailored analyses that find errors such as race conditions.

Provide Extensive and Structured Documentation.

If using an instructional system requires too much undocumented knowledge, the system is often not shared or falls into disuse because the learning curve for instructors is too steep and training teaching assistants is difficult. Pintos includes an extensive 129 page manual, a sample solution, and grading instructions for teaching assistants. The project documentation highlights sections students must read from sections that merely provide supplemental information.

3. PINTOS PROJECTS

The Pintos instructional operating system is split into four projects.

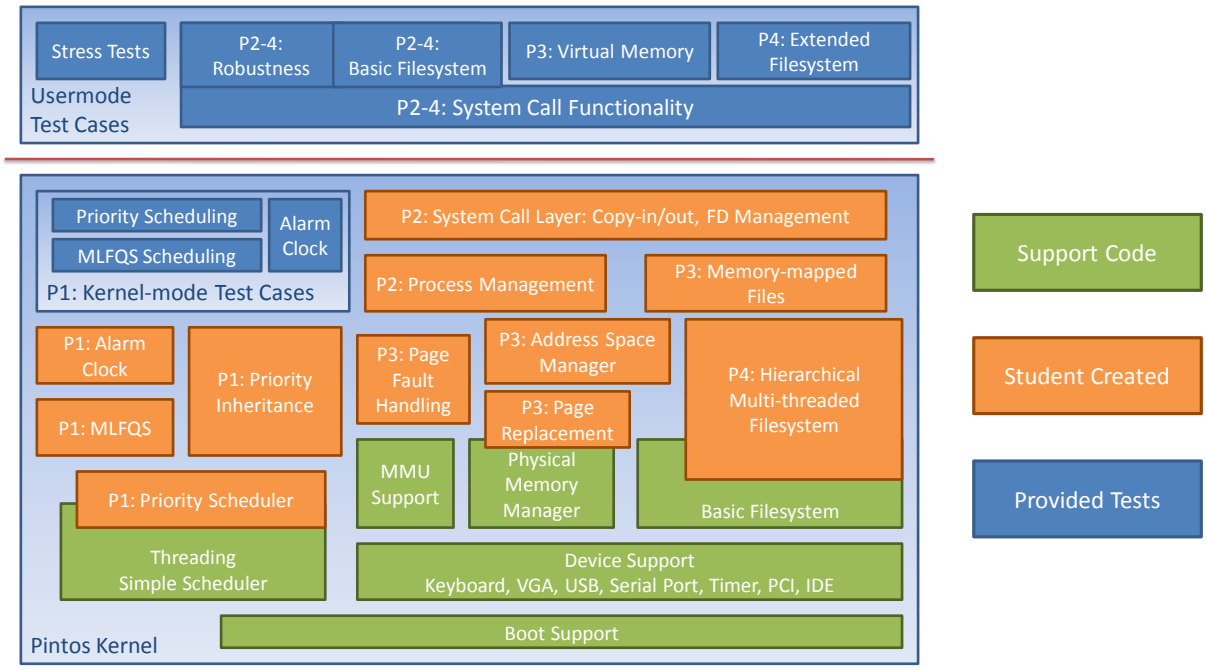

Figure 2: Components of Pintos split in provided support code, test cases, and components created in assignments. Overlapping components indicate when students have to replace parts of the support code.

3.1 Project 1 – Threads

Project 1 revolves around threads. The baseline Pintos code boots into a kernel that supports multiple in-kernel threads. It provides code for initialization, thread creation and destruction, context switches, thread blocking and unblocking as well as a simple but preemptive round-robin scheduler. Students study the existing, barebones threading system (about 600 lines of C code) to understand how threads are created and destroyed, and to understand the transitioning of threads between the READY, RUNNING, and BLOCKED states. They also study how a thread's internal memory is managed, which is used to store its runtime stack and thread control block. Students can examine the context switch code, but the projects do not involve any modifications to it.

After reading the baseline code, the project asks students to implement several features that exercise thread state transitions. The first part of this project includes a simple alarm clock, which requires maintaining a timer queue of sleeping threads and changing the timer interrupt handler to unblock those threads whose wakeup time has arrived. Students learn how to protect data structures that are shared between a thread and an interrupt handler. The second part of the project constitutes a strict priority-based uniprocessor scheduler; students learn about the different ways in which such a scheduler must react to thread state changes.

Based on the priority scheduler, students implement priority inheritance, which deepens their understanding of the interaction of threads and locks. We use the example of the near-failure of the Mars PathFinder mission to motivate the need for priority inheritance. Separately, students build a multi-level feedback queue scheduler on top of the strict priority scheduler. This scheduler adjusts threads' priorities based on a sampling of how much CPU time a thread has received recently.

Testing and Grading.

Project 1 is accompanied by 27 tests as shown in Table 1, which are run by a grading script using the Bochs simulator. Most of the tests are designed to produce deterministic output; the grading script will point out differences between expected and actual output. Usually, a test failure leads students to study the documented source code of the test and understand how the expected output derives from it.

The MLFQS scheduler tests require a different approach. Since those tests rely on estimating CPU usage, they depend on how much CPU time a specific implementation uses, which in turn depends on how efficient it is. We compute the expected CPU consumption values by simulation and provide an envelope within which the output is accepted. The envelope is large enough to allow for minor inefficiencies, but major inefficiencies will usually lead to test failures. Such failures convey the importance of using efficient algorithms and data structures within an OS kernel.

Learning Objectives.

Project 1 has three learning objectives. First, students will understand how the illusion that "computers can do multiple things at once" is created by a sequence of thread state transitions and context switches. Second, they will understand how sophisticated scheduling policies can be built on top of a simple priority-based scheduler. Third, having seen the mechanisms a preemptive scheduler uses to create apparent concurrency, students gain a better intuition of the non-determinism inherent in concurrent systems.

3.2 Project 2 – User Programs

The second project illustrates how an OS implements pro-

tection and isolation between user processes, how user processes access kernel services, and how user processes lay out the virtual address space in which their program and data is contained. Students first add support to Pintos to load and execute user programs. We kept the provided code purposefully minimal to only a library that reads the text and data segments of ELF binaries. These binaries are loaded into a new address space; the baseline code includes functionality to allocate physical memory and set up a page directory to establish the process's virtual address mappings.

Students implement support for a small set of system calls that allow processes to perform I/O, start new processes, wait for their termination, and exit. Both the Pintos user process model and system call API are modeled after traditional Unix, with the exception that Pintos does not separate process creation (i.e., fork()) from program loading (i.e., exec) - instead, Pintos's exec() system call combines these two pieces of functionality into one. The implementation of these calls requires the students to keep track of per-process resources such as file descriptors, which deepens their understanding of how an operating system provides the abstraction of a process.

Like most OS, Pintos exploits dual-mode operation in which user processes run in a nonprivileged mode. Processes attempting to bypass these restrictions are terminated. Students implement the system call handling code, a key aspect of which includes the careful examination of arguments passed by user processes.

Project 2 also illustrates concurrent programming techniques, notably fork/join parallelism, which students implement using rendezvous-style synchronization when providing support for the exec()/wait()/exit() system calls.

Testing and Grading.

All tests for Project 2 are user programs written in C. They are divided into functionality and robustness tests. Functionality tests check that the operating system provides the specified set of services when it is used as expected. Robustness tests check that the OS rejects all attempts at passing invalid input to system calls. To pass those tests, the student's kernel must be "bullet-proof." We include a stress test in which we artificially induce low memory conditions by creating a large number of processes and pseudo-randomly introducing failures in some of them. We expect the kernel to fully recover from such situations.

Learning Objectives.

Students learn how the thread abstraction introduced in Project 1 is extended into the process abstraction, which combines a thread, a virtual address space, and its associated resources. Project 2 enables students to understand how operating systems employ dual-mode operation to implement isolation between processes and to protect system resources even in the presence of failing or misbehaving processes. Students understand how processes transition into the kernel to access its services, and how kernels implement such services in a robust way. The principles learned in this exercise carry over to all scenarios in which applications must be robust in the face of input coming from untrusted sources and uncertain resource availability, as is the case in many server systems.

3.3 Project 3 – Virtual Memory

Project 3 asks students to implement several virtual memory techniques, including on-demand paging of programs, stack growth, page replacement, and memory-mapped files. This functionality is primarily implemented in a page fault handler routine. We provide supporting code to create and maintain page directories, which hide the x86-specifics of how to program the memory management unit (MMU) and how to ensure consistency with the CPU's translation lookaside buffer (TLB). As a result, students can treat the MMU as an abstract device in which to install, update, or remove virtual-to-physical address mappings. Consequently, they are free to choose any design for the data structures needed to keep track of the state of each page or region in a process's virtual address space.

In early offerings, this significant creative freedom came at the cost that some students were lost as to how to accomplish set goals. We added an intermediate design review stage to this project using a structured questionnaire in which students outline their planned design. We also provided a suggested order of implementation.

Like Project 2, Project 3 requires the use of concurrent programming techniques. Since the Pintos kernel is fully preemptive, students must consider which data structures require locking, and they must design a locking strategy that both avoids deadlock and unnecessary serialization.

Testing and Grading.

Project 3 relies on Project 2, therefore, we include all tests provided with Project 2 as regression tests to ensure that system call functionality does not break in the presence of virtual memory. Furthermore, we provide tests for the added functionality that lends itself to such testing, namely, memory-mapped files and stack growth. Some of the project tasks, such as on-demand paging, are performance-enhancing techniques that do not directly add functionality that is apparent to user programs; these tasks are graded by inspection. We test the students page replacement code by varying the amount of physical memory available to the kernel when run under emulation, relative to the amount of memory that is accessed by our test programs. Grossly inefficient page replacement schemes result test failures due to time-outs.

Learning Objectives.

Students learn how an OS creates the environment in which a user program executes as it relates to the program's code and variables. It provides a thorough understanding of how OS use resumption after page faults to virtualize a process's use of physical memory. Students gain hands-on experience with page replacement algorithms and have the opportunity to directly observe their performance impact.

3.4 Project 4 – Filesystems

Project 4 asks students to design and implement a hierarchical, multi-threaded filesystem and buffer cache. In Projects 2 and 3, students use a basic filesystem we provide to access the disk, which supports only fixed-size files, no subdirectories, and which lacks a buffer cache. Although we suggest a traditional, Unix-like filesystem design, which stores file metadata in inodes and treats directories as files, students have complete freedom in designing the layout of their filesystem's metadata. Since our host tools will not know how to interpret the student's filesystems, we attach an intermediate "scratch" disk or partition to the physical

or virtual computer on which Pintos runs, and use the student's kernel to copy files into and out of their filesystems. Similarly, we encourage students to experiment with different replacement strategies for their buffer cache, although we require that their algorithm behaves at least as well as a least-recently-used (LRU) strategy.

As with all projects, this assignment includes additional concurrent programming tasks: in this project, we suggest that students implement a multiple-reader, single-writer access scheme for individual buffer cache blocks to allow shared access when reading file data and metadata.

Testing and Grading.

Project 4 adds a new set of test cases for the extended functionality, including stress tests that check correct behavior for deeply nested directories and for nearly full disks. For each functionality test, we provide a sibling persistence test that verifies that the changes done to the filesystem survive a shutdown and restart. Since the project does not require the virtual memory functionality, it can be built on either Project 2 or 3, depending on the instructor's judgment.

Learning Objectives.

This project provides a deep understanding of how OS's manage secondary storage while avoiding fragmentation and providing efficiency for commonly occurring disk access patterns. Students learn how the use of a buffer cache helps absorb disk requests and improves performance. They also gain insight into filesystem semantics in the presence of simultaneously occurring requests.

4. DYNAMIC ANALYSIS TOOLS

Data races and invalid memory accesses are some of the most common and difficult to debug errors that may occur in concurrent C code. We developed dynamic analysis tools that run on top of the QEMU system emulator [3] to help detect these mistakes. Since these tools do not require additional support from the Pintos kernel, students can use them without complicating their code.

Data races are found by using a semaphore-aware modification of the RaceTrack algorithm [16]. Calls to Pintos's synchronization primitives are instrumented at runtime to track every thread's data sharing pattern. Meanwhile, every memory access records synchronization information to shadow memory maintained by the analysis tool. When the synchronization information for a memory address indicates that a data race occurred, a report including heap information for the data location and the call stacks for the racing threads is generated.

Invalid memory accesses, such as a read from newly allocated but uninitialized data, are detected by tracking all memory accesses. Heap allocation calls are instrumented to map a range of addresses as uninitialized. When data is written to a memory address, it is marked as initialized. If an address marked as uninitialized is read from, an error is reported.

Each of our tools presents students with one or more concrete backtraces that show where the error occurred, which not only helps students debug their code, but makes the concept of race conditions more concrete.

5. FUTURE WORK

In the future, we will expand Pintos's analysis capabilities to provide quantitative information and include realistic device models. We are also considering extending Pintos to multiple CPUs and assignments that involve networking and interprocess communication (IPC). Although we have received highly favorable feedback from our industrial affiliates, who compare students having used Pintos to students having taken courses that use less concrete or external approaches, we need to perform a formal evaluation to compare learning outcomes using Pintos to other alternatives.

6. REFERENCES

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

[2] B. Atkin and E. G. Sirer. PortOS: an educational operating system for the Post-PC environment. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 116–120, New York, NY, USA, 2002. ACM.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATC'05: Proc. USENIX Annual Technical Conference*, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[4] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[5] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, us ed edition, August 2002.

[6] W. A. Christopher, S. J. Procter, and T. E. Anderson. The nachos instructional operating system. In *USENIX'93: Proc. USENIX Winter 1993 Conf.*, page 4, Berkeley, CA, USA, 1993. USENIX Association.

[7] R. Davoli. Teaching operating systems administration with User Mode Linux. In *ITiCSE '04: Proc. 9th annual SIGCSE conf. on Innovation and tech. in comp. sc. ed.*, pages 112–116, 2004.

[8] H. M. Deitel, P. J. Deitel, and D. R. Choffnes. *Operating Systems*. Prentice Hall, 3 edition, December 2003.

[9] A. Gaspar, N. Boyer, and A. Ejnioui. Role of the c language in current computing curricula part 1: survey analysis. *J. Comput. Small Coll.*, 23(2):120–127, 2007.

[10] A. Gaspar, S. Langevin, W. D. Armitage, and M. Rideout. March of the (virtual) machines: past, present, and future milestones in the adoption of virtualization in computing education. *J. Comput. Small Coll.*, 23(5):123–132, 2008.

[11] D. A. Holland, A. T. Lim, and M. I. Seltzer. A new instructional operating system. In *SIGCSE '02: Proc. 33rd SIGCSE technical symposium*, pages 111–115, New York, NY, USA, 2002. ACM Press.

[12] D. Hovemeyer, J. K. Hollingsworth, and B. Bhattacharjee. Running on the bare metal with GeekOS. In *SIGCSE '04: Proc. 35th SIGCSE technical symposium*, pages 315–319, New York, NY, USA, 2004. ACM.

[13] B. Lawson and L. Barnett. Using iPodLinux in an introductory os course. *SIGCSE Bull.*, 40(1):182–186, 2008.

[14] J. Nieh and C. Vaill. Experiences teaching operating systems using virtual platforms and Linux. In *SIGCSE '05: Proc. 36th SIGCSE technical symposium*, pages 520–524, New York, NY, USA, 2005. ACM Press.

[15] H. H. Porter. An overview of the BLITZ system.

[16] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proc. 20th ACM Symposium on Operating Systems Principles*, pages 221–234, New York, NY, USA, 2005. ACM.