Running on the Bare Metal with GeekOS

David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee
Dept. of Computer Science, University of Maryland, College Park, MD, 20742 USA
{daveho,hollings,bobby}@cs.umd.edu

ABSTRACT

Undergraduate operating systems courses are generally taught using one of two approaches: *abstract* or *concrete*. In the abstract approach, students learn the concepts underlying operating systems theory, and perhaps apply them using user-level threads in a host operating system. In the concrete approach, students apply concepts by working on a real operating system kernel. In the purest manifestation of the concrete approach, students implement operating system projects that run on real hardware.

GeekOS is an instructional operating system kernel which runs on real hardware. It provides the minimum functionality needed to schedule threads and control essential devices on an x86 PC. On this foundation, we have developed projects in which students build processes, semaphores, a multilevel feedback scheduler, paged virtual memory, a filesystem, and inter-process communication. We use the Bochs emulator for ease of development and debugging. While this approach (tiny kernel run on an emulator) is not new, we believe GeekOS goes further towards the goal of combining realism and simplicity than previous systems have.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design

Keywords

Operating systems, education, hardware, emulation

1. INTRODUCTION

A fundamental question that instructors must answer when developing projects for an operating system course is "to what extent should I conceal the underlying hardware from the students?" Many points exist along this continuum. A very high level approach is to simulate processes running on the operating system in terms of abstract actions, such as 'compute', 'perform I/O', and 'map a page into virtual memory'. This is the approach taken by Dickinson in [5]. An intermediate approach is to build student projects on top of the user mode of a host operating system. This approach is used by PortOS [1] and by Donaldson in [6]. This approach may expose some low-level hardware details by requiring students to implement a user space thread library. The Nachos kernel [4] is run under a host operating system, but its user processes are executed using a CPU simulator. A variation on the user-level approach is to use a Java Virtual Machine to run the student projects; examples of this approach can be found in [18] and [15]. Finally, projects may be run on real hardware. Perhaps the most famous instructional OS kernel for real hardware is Minix [19].

Running operating system projects on real hardware has several advantages. The main advantage is that it allows the instructor complete control over the level of complexity to which students are exposed. Depending on how much code is provided to the students, projects can range from very high level to very low level. In systems relying on simulation, such as Nachos, there is a fundamental limit on how realistic the projects can be. Another advantage is that running on real hardware provides a good opportunity for students to learn about computer architecture in a hands-on setting. Finally, hardware-level programming experience is useful considering the increasing importance of embedded systems in everyday life.

Working at the hardware level has two main disadvantages. First, hardware devices can be tricky to program correctly. A more fundamental problem is that debugging kernel code running on real hardware is difficult, even for experts. The contribution of our work is to show that both of these difficulties can be overcome without requiring heroic measures from students or instructors. We have implemented a tiny OS kernel, called GeekOS, which provides a sufficient abstraction layer over the hardware to hide the genuinely difficult details. By providing well-commented source and ample supporting documentation, students quickly get up to speed with hardware-level programming. Students develop and run their projects on the Bochs [3] emulator rather than actual PC hardware. Bochs provides extensive diagnostics and debugging support, which makes writing and debugging kernel code almost as easy as user code.

The structure of this paper is as follows. In Section 2 we give a brief overview of GeekOS. In Section 3 we describe projects in which students add important operating system features to GeekOS. In Section 4 we describe our experiences using GeekOS in an undergraduate operating system course. In Section 5 we describe related work. We conclude in Section 6.

2. OVERVIEW OF GEEKOS

GeekOS is a tiny OS kernel for x86 PCs, written in C with a small amount of assembly. The source currently contains about 3700 lines of C and 502 lines of assembly, not including comments and blank lines.[1] We have tried to make the code as easy to understand as possible. GeekOS and the project materials based on it are available under an open source license, and its development is hosted on SourceForge [10]. We are committed to making the source and documentation to GeekOS available on a long-term basis.

In its most basic form, GeekOS supports the following functionality:

- Handling interrupts
- Kernel threads with a static priority scheduler; the timer interrupt is used to make scheduling preemptive
- Allocation of physical memory pages
- Allocation of physical memory buffers from a kernel heap
- Device drivers for the interrupt controller, keyboard, VGA text screen, IDE hard drives, and floppy drive

GeekOS is written to conform to well documented hardware specifications: specifically, an AT-class PC with a 486 or higher processor. Hardware documentation for this type of system is readily available in books [17, 12, 20] and from online sources. GeekOS satisfies our requirement for realism, since it is possible to write a kernel image to a floppy or hard drive and boot it on a real machine.

Because GeekOS targets an x86 PC, which is by far the most common mainstream hardware platform, we have access to a wide variety of well-supported open source development tools. We use gcc and binutils [8, 2] as our compiler toolchain, and nasm [14] as our assembler. These are often available as standard components of popular open source operating systems such as GNU/Linux and FreeBSD, meaning that in many cases no special software (other than Bochs) needs to be installed in order to compile and run GeekOS. For users of non-x86 systems, such as Mac OS X, Solaris, etc., it is quite easy to build a cross compiler and toolchain capable of building GeekOS.

2.1 Why Emulate?

While targeting student projects for a mainstream hardware architecture satisfies our requirement for realism, we do not require students to use physical machines to develop their projects. Debugging a kernel on real hardware is difficult. In addition, system administrators are likely to take a dim view of allowing students to boot their (probably buggy) kernels on machines in an open workstation lab. Having dedicated machines for student projects is possible, but inconvenient.

Fortunately, a number of excellent PC hardware emulators are available [3, 21]. Because emulators run as ordinary user processes on a host operating system, they have many advantages over running on physical hardware:

- They boot extremely quickly
- They can be run anywhere: workstation labs, timesharing clusters, students' home machines, etc.
- They generally offer better diagnostic and debugging support than physical hardware

The idea of using hardware emulators to teach operating systems is not new [6, 11]. However, it is only in the past few years that the CPU and memory resources (both of which are needed in abundance for effective emulation) of commodity hardware have become sufficient to make emulation practical; (perhaps) for this reason, the use of hardware emulation in operating system courses does not seem to be widespread. We use the Bochs [3] emulator in our operating systems course. Bochs has a number of features that make it a good choice. First, it is open source software with active developer and user communities. It runs on a variety of host operating systems, including GNU/Linux, varieties of Unix, Mac OS X, and Windows, and is not tied to any particular host CPU architecture. For these reasons, it is easy to deploy in almost any kind of workstation lab. It offers extensive debugging support, including stubs for remote debugging using gdb [9]. This makes developing kernel code almost as easy as ordinary user code.

3. STUDENT PROJECTS

On top of the base GeekOS system, we have developed a sequence of projects in which students add important operating system features. In this section, we briefly describe each project.

3.1 Processes

In the first project students implement processes. Program executables are loaded from a provided PFAT[2] filesystem residing on a virtual floppy disk.

The project uses segmentation to prevent processes from accessing memory outside their own address space. Note that the form of segmentation available in 32 bit protected mode (used in GeekOS) is superior to the dreaded segment/offset addressing found in DOS and early versions of Windows. Each process has its own local descriptor table (LDT), in which identical code and data segments are defined. The code and data segments refer to the single chunk of memory allocated for the process. So, from the process's viewpoint, it is running in a flat address space, with any access beyond the limit of the code/data segment resulting in a trap to the kernel.

We require the students to implement `Spawn` and `Wait` system calls to allow processes to create child processes and to wait for them to exit.

What the students learn. Students learn about how processes are isolated from each other. Although segmentation on the x86 does not afford the same flexibility as paging—for example, it cannot be used to trap null pointer references—it is easier to understand. Because the executable images are in ELF format, students learn about linking and loading. The `Spawn` and `Wait` system calls require

[1]The heap memory allocator, which is opaque as far as students are concerned, accounts for 1300 lines of C.

[2]PFAT (or 'pseudo-FAT') is a simple read-only filesystem used to store executables to be loaded after the kernel boots. A PFAT filesystem image is created by an offline tool.

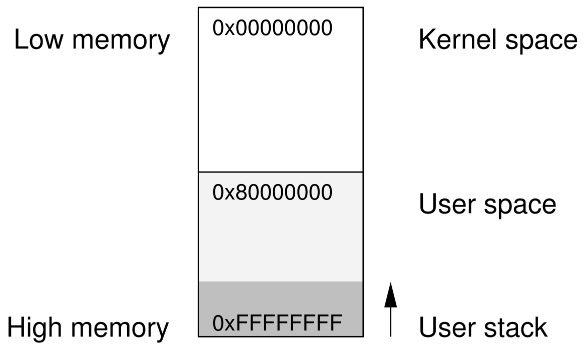

Figure 1: Virtual memory in GeekOS.

passing parameters to the kernel, including (for `Spawn`) a user mode character string. Ensuring that these parameters are valid is a practical lesson in the kinds of properties that the kernel must check in order to prevent misbehaving user processes from corrupting the state of the system. For example, it should not be possible for a malicious user process to cause the kernel to access or modify kernel data structures or the memory of other processes.

3.2 Scheduling

In this project, students implement a multi-level feedback scheduler, which coexists with the original static priority scheduler. In addition, they implement named semaphores which may be accessed by user processes using Dijkstra's `P` and `V` operations. We provide a synthetic workload which allows students to measure the behavior of the two scheduling policies.

What the students learn. A key goal of the scheduling project is to have students read and modify code written by someone else. A thoughtful implementation supporting both static and dynamic priority scheduling can be done by modifying only a few lines of code. Having students include a short writeup about the two different scheduling policies on different workloads also serves an important pedagogical function. In particular, it helps to reinforce the idea that programming is not simply picking a 'best' algorithm, but requires considering the expected workload when making design choices.

3.3 Virtual Memory

One of the strongest motivations for running projects on real hardware is to enable students to implement a real virtual memory system. (This is much harder to do if the projects are run in a host operating system or JVM.) At first, implementing virtual memory on a real processor might seem like a daunting task. However, the MMU in the x86 CPU architecture is easy to program. Its TLB is implemented entirely in hardware, so the kernel is only responsible for keeping page tables up to date, and for flushing the TLB when page mappings are removed. Also, its cache is physically indexed, so no special cache management by the kernel is required for context switches.

In this project, students replace the segmentation-based memory protection scheme from the first project with protection based on paging. The virtual address space is divided into kernel space (starting at address 0) and user space (starting at address 80000000h). All of physical memory is identity-mapped in the kernel space; this allows the kernel to enable paging without any need for address remapping.[3] It also allows the kernel to access any physical address without needing to create temporary mappings. The user code and data segments are defined so that their base address is the beginning of the user portion of the virtual memory space. In this way, user executables may be linked starting at address 0; at runtime, the segmentation hardware will translate user addresses into the high virtual memory space before the MMU sees them. This arrangement gives user processes the illusion that their address space ranges from 0 to 2G. The layout of virtual memory is described in Figure 1.

To incorporate virtual memory in their kernels, students must make several modifications. First, they must build an initial set of page tables covering kernel space, and enable paging. The second and more difficult step is to modify their process creation and switching code to use page tables instead of segmentation. We do not require students to demand page the executable; instead, they preallocate memory for user code and data, which allows them to reuse some of their executable loading code from the first project. We do, however, require them to implement a growable stack at the end of the user address space. Finally, students implement a page fault handler and a paging file, which allows the kernel to evict pages to disk when the system is running low on memory. The paging file is accessed using the provided IDE disk driver. We simplify management of the paging file by allowing pages to be completely purged from the paging file when they are brought back into memory; this simplifies the bookkeeping, at the expense of always requiring pages to be written to disk when paged out.

We feel that the relative ease with which virtual memory can be added to GeekOS demonstrates two important points. First, it lends support to our view that programming at the hardware level does not impose an undue burden on students, even when sophisticated hardware features are used. Second, and more importantly, it shows that having access to the full power of modern hardware greatly expands the possibilities of what students can accomplish.

What the students learn. Virtual memory is fundamental to how modern computers work. By gaining first hand experience with how an operating system uses virtual memory to efficiently allocate memory resources to processes, students are better equipped to understand system performance issues affecting all software.

3.4 A Filesystem

In this project, students implement a hierarchical read/write filesystem. The filesystem uses a traditional indexed allocation scheme using 4K blocks, in which the first eight blocks of a file are direct blocks, and the next 1024 are singly indirect blocks. This approach is similar to the original Unix filesystem design [16].

Because the base GeekOS system already contains an implementation of the read-only PFAT filesystem, we provide a VFS layer which calls down to functions in PFAT or the student's filesystem as appropriate. The VFS layer defines Mount, Open, Close, Delete, Read, Write, Stat, Seek, CreateDirectory, Sync, and Format operations. Students are responsible for adding system calls so all of the VFS opera-

[3]Prior to enabling paging, the kernel runs without any address translation.

Question	yes	somewhat	no
Was GeekOS easy to understand?	10	14	1
Did GeekOS help you understand the concepts in the course?	10	15	
Did you find it interesting that your project could run on real hardware?	18	5	2
Was GeekOS a good basis for the student projects?	19	5	1

Figure 2: Responses to student survey.

tions can be invoked by user processes. We also provide code to keep track of mounted filesystems and open file handles. GeekOS does not include support for disk partitioning, so we simply dedicate IDE disk 0 to PFAT and IDE disk 1 to the student's filesystem.

What the students learn. Several important topics are covered in this project. Students learn how disks are accessed, and how filesystem data structures create higher level abstractions on top of them. The VFS layer shows how multiple filesystem types may coexist. Finally, students learn how locking can be used to protect shared data structures when they are used concurrently.

3.5 Inter-Process Communication

In the final project, students implement an inter-process communication mechanism based on message queues. The queues are named, and thus can be used by unrelated processes. They support buffered half-duplex communication. Students also implement a permissions system based on user IDs and access control lists (ACLs), which are used to limit access to files and message queues. Finally, students extend the VFS from the previous project to support message queues and the console (keyboard and text screen), and modify the `Spawn` system call to specify input and output file descriptors for each created process.

What the students learn. Upon completing this project, GeekOS resembles a very simple version of Unix. We provide a tiny shell which can execute programs, allowing I/O redirection using files and message queues. Using the shell, students can test the complete range of functionality implemented in their projects, seeing how all of the features work together.

4. EVALUATION

In this section, we discuss our experiences using GeekOS in the undergraduate operating system course at the University of Maryland.

4.1 Background

Several of the projects we developed using GeekOS were originally developed for an earlier incarnation of the project sequence which ran under MS-DOS using C and 16-bit real mode assembly. These projects were similar to the GeekOS-based projects in that they made significant use of hardware features, such as the keyboard and timer interrupts. This approach provided substantial realism, but had several drawbacks. The programming model and development tools offered by MS-DOS were inferior to those available in more modern systems. Bugs in student projects were likely to corrupt the state of the host operating system, making debugging difficult. Also, the distinction between services offered by MS-DOS, the computer's BIOS, and the hardware is muddled.

We found that the important features of the original projects translated easily to GeekOS. We were able to use modern development tools, and the Bochs emulator provided an isolated environment for debugging the projects. In addition, we were able to expand the scope of the projects to include new features, such as virtual memory, that would have been much more difficult under MS-DOS. GeekOS retained the good features of the original projects, while making development and debugging easier.

4.2 Student Experience

One concern we had in using GeekOS in the classroom was that programming at the hardware level would be perceived as too difficult by the students. To evaluate the students' experiences, we distributed a short survey to students in the first class (Fall 2001) which used GeekOS as the basis for class projects. The survey results are shown in Figure 2.

In general, student experience was positive. Only one student felt GeekOS was not easy to understand. Given the complex nature of kernel programming, we saw this as a success. The students also indicated that GeekOS was either helpful (10/25) or somewhat helpful (15/25) in understanding the concepts taught in the course. In the four semesters since the survey was taken, GeekOS has become much better integrated into the course, so we expect that if another survey were taken, students would rate GeekOS more highly as helping to reinforce the course concepts. A clear majority of students felt GeekOS was a good basis for student projects (19/25), and that there was value in having projects capable of executing on real hardware (18/25).

4.3 Instructor Experience

As instructors, the Bochs based project sequence has allowed us to focus class time and office hours on concepts and how the components in the operating system fit together. In the old projects, a significant amount of time was consumed answering questions about the obscure development environment or how to debug when the OS crashed the hardware.

GeekOS also provides ample opportunies for defining alternative projects. Possibilities include dynamic linking, log based or other alternative filesystem implementations, slab memory allocation, filesystem and virtual memory integration, etc. These alternative projects can be used either to help reduce plagarism by providing a larger pool of possible projects, or to allow team projects rather than the current individual projects.

Since their introduction, some of the projects have evolved in the direction of requiring less hardware-level programming by the students. This trend is an example of the freedom running on real hardware offers; we can conceal the low-level details where they are less important, and focus on them where they are more important. For example, in the first project, we now provide the students with code to set up the user code and data segments and LDT, rather than requiring them to write it. In contrast, in the virtual memory project, students are responsible for setting up the page tables and managing the TLB themselves, because these are essential to understanding how virtual memory works.

5. RELATED WORK

A large number of instructional operating system kernels have been developed over the years. Two of the best known are Nachos [4] and Minix [19]. The approach taken by Nachos is quite similar to ours, in that it provides a very minimal set of services upon which students build more complex features. However, it uses a CPU simulator embedded in the kernel to execute user processes, which creates practical and conceptual difficulties. Minix runs on real hardware; however, it is a complete operating system implementation, with about 30,000 lines of source code in the kernel alone. We feel this is too much code for students to get up to speed with in a one-semester course. Production quality kernels such as Linux and FreeBSD are at least an order of magnitude more complicated than Minix, and are thus even less practical to use in a course.

Two existing instructional OS kernels are similar to GeekOS in spirit. OS/161 [11] runs on a custom MIPS hardware emulator, System/161. It is slightly more complete than GeekOS, having more kernel functionality and a proper C library. Topsy [7] is an instructional microkernel for the MIPS architecture. Like GeekOS, both OS/161 and Topsy leave major kernel subsystems to be implemented by students. GeekOS differs from Topsy mainly in that it is not a microkernel design. GeekOS differs from OS/161 in being somewhat less ambitious in the range of services it offers to user processes. Because it targets the x86 architecture, it is much easier to obtain up-to-date development tools for GeekOS than for Topsy and OS/161.[4]

6. CONCLUSIONS

From our experience with GeekOS, we have drawn several conclusions. First, having students work on top of (emulated) real hardware is a viable approach for an undergraduate course. The key to making this approach practical is to strive for simplicity wherever possible, and to find an appropriate level of abstraction which exposes only those hardware details which are necessary. Second, hardware emulation brings many of the conveniences of ordinary user mode software development to kernel development. Finally, the x86 PC hardware platform is a good choice for operating systems projects due to the wealth of excellent open source development tools, ease of system level programming, and the availability of good documentation.

In the future, we would like to make GeekOS even easier to understand and modify. Some of the internal interfaces could be simplified. We would also like to expand the supporting documentation.

It would be interesting to explore the possibility of compiling GeekOS using a 'safe' C dialect, such as Cyclone [13]. The kinds of memory errors common in C programs are especially problematic in operating system kernels. Being able to catch memory errors at compile time would be tremendously useful for both students and instructors.

[4]An x86 port of Topsy exists, but is not included in the standard distribution.

7. REFERENCES

[1] B. Atkin and E. G. Sirer. PortOS: An Educational Operating System for the Post-PC Environment. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2002.

[2] GNU Binutils: http://www.gnu.org/software/binutils, 2003.

[3] Bochs: http://bochs.sourceforge.net, 2003.

[4] W. A. Christopher, S. J. Procter, and T. E. Anderson. The Nachos Instructional Operating System. In *Proceedings of the 1993 Winter USENIX Conference*, pages 481–488, 1993.

[5] J. Dickinson. Operating Systems Projects Built on a Simple Hardware Simulator. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2000.

[6] J. L. Donaldson. An Architecture-Dependent Operating System Project Sequence. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2001.

[7] G. Fankhauser, C. Conrad, E. Zitzler, and B. Plattner. Topsy — A Teachable Operating System: http://www.tik.ee.ethz.ch/˜topsy/book/topsy_1.1.pdf, 2000.

[8] GCC Home Page: http://gcc.gnu.org, 2003.

[9] GDB: The GNU Project Debugger: http://www.gnu.org/software/gdb, 2003.

[10] GeekOS: http://geekos.sourceforge.net, 2003.

[11] D. A. Holland, A. T. Lim, and M. I. Seltzer. A New Instructional Operating System. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2002.

[12] Intel. *Intel Architecture Software Developer's Manual.* Intel, 1997.

[13] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.

[14] The Netwide Assembler: http://nasm.sourceforge.net, 2003.

[15] T. Nicholas and J. A. Barchanski. TOS - An Educational Distributed Operating System in Java. In *Proceedings of the ACM Technical Symposium on Computer Science Education*, 2001.

[16] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7), 1974.

[17] T. Shanley. *Protected Mode Software Architecture.* Addison-Wesley, 1996.

[18] A. Silberschatz and P. Galvin. *Applied Operating System Concepts.* John Wiley and Sons, 1999.

[19] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation.* Prentice Hall, second edition, 1997.

[20] F. van Gilluwe. *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas.* Addison-Wesley, second edition, 1996.

[21] Virtutech: Simulators for hardware and software engineering: http://www.virtutech.com, 2003.