A New Instructional Operating System

David A. Holland, Ada T. Lim,[1] and Margo I. Seltzer
Harvard University
Cambridge, MA 02138
{dholland,ada,margo}@eecs.harvard.edu

Abstract

This paper presents a new instructional operating system, OS/161, and simulated execution environment, System/161, for use in teaching an introductory undergraduate operating systems course. We describe the new system, the assignments used in our course, and our experience teaching using the new system.

1 Introduction

Traditionally, undergraduate classes in operating systems are taught using special-purpose instructional OSes. These systems are meant to be simple and easily comprehensible; they have pieces intentionally left out as exercises for students.

Several instructional OSes of various types have been written. They seem to have a working lifetime of around ten years and then become dated, due both to changes in the OS community and to changing expectations and prior knowledge bases of students.

We present a new instructional OS, called OS/161, and accompanying platform, called System/161. Our goals in its development were to (1) provide a realistic execution environment; (2) facilitate debugging; (3) retain the simplicity and assignment-oriented structure necessary for course use; (4) help to familiarize students with the structure and layout of real OSes; (5) have the completed OS at the end of the course be capable of running real, if small-scale, user applications; and (6) provide a cleanly written and robust code base.

In the remainder of this paper, we discuss other teaching OSes and how they relate to OS/161, along with an outline of why we do not believe in using production OSes for teaching; then we provide an overview of OS/161 and System/161 from an instructional perspective. Following that, we discuss our course structure and assignments, and our experience teaching OS/161 this past year (2001). Finally, we conclude with an analysis of the extent to which we met our goals.

[1] Also University of New South Wales.

2 Other operating systems

The last new full-scale instructional OS was Nachos [1], which appeared circa 1993. Nachos has an unconventional architecture: the operating system kernel and a machine simulator are compiled into a single executable, which then runs as an ordinary process on some "host" operating system. User-level processes run *in* Nachos are run in the machine simulator, but the Nachos kernel runs on the host system's native hardware. We refer to this design as "mixed mode".

In some ways it is quite clever: it means the Nachos kernel can be run in a standard debugger. It also means Nachos does not need to include its own implementations of standard C functions, such as `printf` or `strcpy`, because the Nachos kernel is linked to the host OS's standard C library.

There are, however, some disadvantages to mixed mode. The machine simulator is for a fixed processor type, a MIPS r2000 running in little-endian mode. The host machine may be anything, which today means that it is *not* a MIPS r2000. It might not even be little-endian. This means that, depending on the choice of host platform, data sizes and representations may not be the same in the Nachos kernel and in Nachos user-level programs. This causes a great deal of confusion and opportunity for mysterious and unrealistic bugs. Furthermore, because structure sizes may not be preserved, it is difficult to create a fully functional system call interface.

Nachos has some additional related drawbacks: first, because the kernel does not appear in the simulated machine's RAM, it is not limited by memory constraints.

This means that the size of kernel data structures does not significantly affect system performance, which makes the size/space tradeoffs for many design decisions unrealistic. And second, perhaps worst of all, all interfacing to the machine simulator and simulated hardware devices takes place via C++ objects. This means students receive no exposure to the way real hardware is actually accessed, even though that is a significant aspect of kernel programming.

For teaching, a certain amount of realism is desirable. Too much realism, however, becomes both too complicated and, sometimes, realistically painful. We take the position that methods and mechanisms should be realistic, but details should be simplified: the System/161 hardware is accessed the way real hardware is, but the devices themselves are simple and easy to work with. By contrast, handling real devices, especially on i386 platforms, involves a large collection of complicated device drivers, making a system large and less readily comprehensible than it might otherwise be.

Recently Nachos has been modified to run as a native kernel on top of SimOS [4]. This alleviates many of the problems with mixed mode described above. SimOS is a MIPS machine simulator similar to System/161; however, SimOS simulates real hardware and can in fact support real OSes like Irix, and the modified Nachos must accomodate this.

Minix [5] is often used as an instructional OS. It can be run on real hardware, or on an i386 simulator such as Bochs [3]. It thus offers a substantial, perhaps excessive, degree of realism.

Another new teaching OS is Topsy [2], which runs natively on MIPS simulators such as SimOS. However, it includes no native support for either paged virtual memory or file system operations; it does not appear to be intended to illustrate the design and implementation of a complete system but rather to offer a platform for teaching concurrency and hardware manipulation.

Other past instructional OSes that appear to be more or less obsolete include Xinu, from Purdue circa 1984, and TOY, developed in 1973.

The ultimate in realism is to use a real production OS. This has drawbacks, however, beyond the issue of device drivers: real OSes are immensely large and complicated, and are full of complexities and constructs for coping with real-world issues that have little instructional value. Furthermore, real OSes are already fully functional, so students do not get to design or implement major subsystems or even minor interesting parts. Finally, production OSes are usually too slow for simulators; running them on bare hardware makes debugging much more difficult and requires making hardware available to students (often problematic). We feel that for these reasons, introductory OS classes involving kernel programming at all should avoid production OSes.

3 System/161

We came to the conclusion that computers are now fast enough that, even on the chronically overloaded undergraduate computer facility, it is practical to use a completely simulated platform for teaching the operating systems class.

We thus created a simulated hardware platform, System/161. System/161 is extremely simple: the first and foremost design goal was to provide the necessary functionality without complicating the operating systems running on it. Thus, it has a straightforward bus architecture, into which simulated devices, supporting disks, serial ports, timer/clock, and random generation, can be inserted. These devices are accessed via registers in the same way real devices would be. (There is, however, no DMA; instead some of the devices have memory-mapped transfer buffers.) The devices are minimalistic: even the most complex one has only six registers. The explicit intent was that an experienced kernel programmer would be able to write a complete set of device drivers in an afternoon.

System/161 also contains a complete integer MIPS r2000 processor simulation running in big-endian mode. The code is structured to support multiple processor types. Support for several other processors has been proposed and/or begun, but not completed. Floating-point support is not planned.

System/161 is written in portable C; it was developed under NetBSD on i386 hardware and used by students under Digital Unix on Alpha hardware. It consists of approximately 9,000 lines of code. On a 550MHz Pentium-III, the most recent version manages a little over 3MHz running cpu-intensive tasks. (This could undoubtedly be improved.) While not ideal, such performance is acceptable, as OS code is rarely cpu-intensive.

Perhaps the most significant feature of System/161, however, is the debugging support. System/161 contains its own remote `gdb` hooks, so any kernel loaded into System/161 can be debugged with `gdb` without needing support of any kind from that kernel. Even better, the debugging is completely transparent and does not affect the simulation's timing. It is thus possible to trace right into the context switch, for instance, or to tackle timing-related problems in the debugger.

By contrast, remote `gdb` to a real kernel running on real hardware requires placing the debugging hooks within the kernel itself. This unavoidably complicates the kernel and affects its operation; furthermore, it is generally impossible to debug low-level parts of the kernel. While debugging a "mixed mode" system like Nachos does not require the kernel's cooperation, it still affects timing. Also, in a mixed mode environment, the debugger debugs the machine simulator as well as the kernel, which is generally not illuminating and sometimes confusing.

None of these features require the use of OS/161; System/161 is an independent entity. In fact, an unexpected side benefit of building it has been that it offers a fairly nice platform for quick and dirty kernel hacking. It will load and run any ELF[2] format kernel compiled for the right processor type.

OS/161 is not necessarily tied to System/161 either; there is no barrier, besides complexity we did not wish to introduce, to porting it to other platforms, simulated or real. However, OS/161 and System/161 were designed together and are intended to be used together.

4 OS/161

OS/161 is intended to feel like a real OS, while still being simple enough to hand out to undergraduates. It is intentionally similar to BSD Unix in general organization and structure. It comes with a dozen or so of the basic Unix shell commands and is laid out to use a Unix-like system call interface. It includes a skeleton standard library (to be extended in the future), which is also Unix-like. The advantage of this is that simple code will compile on Unix, Linux, BSD, etc. as readily as it will on OS/161. Thus, students can test and debug user-level code elsewhere before running it under OS/161.

Since OS/161 runs in a completely simulated environment, rather than mixed mode, all the interrupt and exception handling mechanisms work exactly as they would on real hardware. We provide most of the code that implements these things; we do not require students to write assembly code, although we do encourage them to read it. We take only one shortcut: we do not implement the confusing MIPS cache flush code, as we know System/161 does not include a cache simulation.

Similarly, the OS/161 kernel must include implementations of all the C library functions it uses. We provide these (including `printf`); much of this code is shared with the user-level C library.

In a teaching OS, the file system serves two quite separate purposes: it serves as both a subject of instruction and as a repository for materials (executable programs and data) used in the course of operation. If these purposes are made to overlap, defects in the students' file systems can cause the operational functions to fail. This is frustrating for the students and a source of wasted time. Thus, while it is desirable for the students to be able to use their file systems for the operational functions for testing, it is not desirable to require such use.

Thus we provide two file systems in OS/161: one is a "real" file system that sits on top of a simulated disk device and accesses blocks from it. The other is the "emulator" file system, `emufs`, which works through a special System/161 device and accesses the file system of the *host* OS. The former is used for instruction; the latter is used, under ordinary circumstances, for operation. A side benefit of this organization is that there is no need to copy the OS/161 user programs onto a disk image file after every recompile.

In order to make these two file system types function together, and to allow students to add additional file system types if they so desire, we provide a VFS (virtual file system) layer. The VFS layer uses a `device:path` syntax, like AmigaDOS or MS-DOS, so as to avoid the complications associated with mount points; apart from splitting off the device name, if any, it does no interpretation of paths or filenames. (Usage of the path "`lhd0:my/files/test.txt`" causes the string "my/files/test.txt" to be passed to the file system associated with `lhd0`, the first simulated disk.) The VFS layer code does not require student attention in any of our assignments, although the students are encouraged to understand how it works.

Having the separate file system for operational use also means that general-purpose file system activities are possible even before students tackle the instructional file system. As much as possible, we have structured the system and assignments so that they may be completed in any order.

For instance, to avoid requiring the virtual memory assignment to come first, we provide a simpleminded virtual memory system, which we call "dumbvm", described below. This provides just barely enough functionality to allow user-level code to run.

Other functionality included in the base OS/161 kernel is kept to a minimum, in order to try to minimize the amount of unfamiliar code students have to absorb. It includes basic thread and context switch code, basic support for threads to block and be awakened, device drivers, console I/O routines, code to read ELF executables, some assorted test code, and a skeleton instructional file system for the file system assignment. The base kernel is 11,000 lines of code with another 8,000 lines of comments; the user-level tools and test programs add another 7,000 lines of code.

[2]Embedding and Linking Format, a widely used file format for executables.

5 Assignments

Our course is set up with five assignments and a preliminary exercise. (The preliminary exercise consists of reading portions of the code and answering some questions about it, building and running a kernel, and trying out the debugger.) Starting with assignment 2, the students work in pairs. The organization and content derive from the original Nachos assignments; in the past we used Nachos in our course and we intentionally preserved the assignment structure.

Assignment 1 covers synchronization and concurrent programming. The students are given a BSD-like sleep/wakeup interface to the thread system and a semaphore implementation that uses it. They must implement locks and condition variables (with Mesa semantics), and then use these synchronization primitives to implement concurrent solutions to three or four synchronization problems. We change the problems every year, but typically one is a signalling problem of some kind and another is a producer-consumer problem. This coding takes place within the OS/161 kernel. In order to make this interesting, we make the kernel fully preemptible.

Assignment 2 covers processes and system calls. The students implement a dozen or so basic system calls, complete with proper argument handling. This requires implementing file handles, as well as the argument handling associated with `execv`, process creation with `fork`, and waiting for processes to exit with a subset of `waitpid`. The students are also responsible for arranging that processes get killed correctly upon fatal faults, and they implement a simple shell.

They then implement two different schedulers and do some simple system performance analysis based on choice of scheduler and tuning of scheduler parameters.

Assignment 3 involves writing a VM (virtual memory) system. The "dumbvm" hack is turned off for this assignment and students get to write nearly everything from scratch. We require support for swapping and for malloc at user level; however, we do not require memory-mapped files or copy-on-write shared regions, although students have implemented both these things in past years. We then ask students to tune their VM systems and report their performance measurements.

Since the students are writing even the most fundamental parts of the VM system, it is important that the system not expose any avoidable rough edges. Thus the base system is arranged so that VM initialization comes *after* console initialization, so debug messages can be printed. We provide a framework for grabbing memory during bootup and before VM initialization. Some students find this distasteful and rip it out, but many do not, and we believe it worthwhile.

In assignment 4, the students extend the skeleton file system we provide. What we provide is primitive: it does not support directories, files larger than about 72k, multiple concurrent processes, or a disk cache. The students are supposed to remove these restrictions.

Additionally, they must implement a half-dozen file system-related system calls, and write up performance analysis of their disk cache.

The original Nachos assignment 5 involved implementing a simple network protocol; in our course over the years we have used that assignment and other similar ones. However, networking is in many ways its own field; in our department it has its own class, and so we decided instead to open up the assignment to be a student-selected project. We provide a list of several dozen suggestions; they must do either one large project (for example, a file system integrity checker) or two smaller projects (such as implementing pipes).

Unfortunately this past year we did not have simulated network hardware ready in time to allow network-related assignment 5 projects, but that will not be a problem in the future.

6 Experience

One thing we discovered this past year is that our teaching of synchronization concentrated too much on high-level synchronization primitives and not enough on the realities of synchronizing with interrupt handlers by disabling interrupts. While the high-level primitives are preferable in various ways, in real life one needs to deal with interrupts from time to time, and disabling interrupts is not functionally equivalent to any of the conventional primitives. We found that our students tended to get into trouble using interrupt synchronization when it proved necessary in certain contexts, such as the VM system. So we are considering adding a synchronization problem that must be solved by disabling interrupts.

We also found that our original implementation of "dumbvm" was too dumb. The original dumbvm had no concept of processes, address spaces, or valid address ranges: whenever a page fault occurred in user space, no matter where the fault was, it grabbed a blank page of memory and mapped it in using the MIPS memory management unit. When it ran out of slots in the MMU, it gave up. It could be made to clear everything out and start over, but even then it never reused pages.

This allowed only one process at a time. In order to allow running user-level code that used `fork` and `exec`, we contrived a hack: fork would do nothing and just return and claim to be the child process, so the parent process would evaporate. While this worked, it proved undesirable: it postponed implementation of the fork system call itself to the VM assignment, which helped make the VM assignment too long. Furthermore, because it did no bounds checking, it made it harder than necessary to test anything involving user space, and rendered certain kinds of failures invisible. Worse, because it never reused memory, even for in-kernel allocations, it masked a wide variety of student implementation bugs in assignment 2 code that then surfaced later.

These problems proved serious. The new dumbvm implementation, that we will use this coming year and presumably thereafter, can handle more than one process and keeps track of the legal addresses in each, although it is still heavily restricted and limited by the number of slots in the MMU.

In the orignal OS/161, the implementation of `malloc` and `free` in the kernel were essentially stubs that called into dumbvm to grab pages. They never reused memory either. Part of the VM assignment (assignment 3) was to implement the *kernel* versions of `malloc` and `free`. This was a serious mistake. Since many parts of the kernel use `malloc`, bugs therein would cause bizarre crashes, the kind that undergraduates generally do not have the experience to deal with effectively. Furthermore, the various bugs in students' assignment 2 code that were hidden by never reusing memory came to life. Some of these were detected in time by course staff grading assignment 2, but not all. These factors made life rather unpleasant for all concerned. So we now provide a fairly robust implementation of `malloc` and `free` as part of the base system.

On the other hand, the project assignment worked quite well; some of our students did quite elaborate projects, while the less ambitious could pick things that were relatively straightforward.

We chose MIPS as the simulated processor platform because we already had some experience with it from having used Nachos in the past. Unfortunately, it turned out that the various freely available compilers and compiler tools for MIPS were in a rather poor state. We had some difficulty preparing a toolchain that would itself compile on all the host platforms we wished to use, and even then it turned out to have several annoying bugs. There is some reason to hope that the situation may be improving, but we are seriously contemplating moving to a different processor architecture in the future.

In general, while we did make some poor design decisions, described above, we believe that these have been rectified. We also had a few problems arising from using a new and thus immature code base. A significant amount of time has been invested in improving the robustness of the code, and on general polishing, and we expect a much smoother experience this coming year.

7 Conclusions

In the introduction, we listed six design goals for OS/161 and System/161. In our estimation, with one exception, we have accomplished them. We created a realistic execution environment that supported easy debugging; we wrote an operating system that remains sufficiently simple for instructional use, but reflects the design and layout of real OSes; and we believe that the code we have is cleanly written and well organized. We did have several students go out of their way to tell us that they liked the code.

Unfortunately, it is not yet the case that the finished OS/161 properly supports "real" user applications. This is, however, largely because OS/161's C library is not yet sufficiently complete; this problem is easily solved, and we hope to port over some of BSD /usr/games before long.

We believe we have produced a useful system and instructional tool and hope others may find it useful for themselves as well.

8 Availability

OS/161 and System/161 are freely redistributable under a BSD-like license and are available for download from `ftp://ftp.eecs.harvard.edu/pub/os161`.

References

[1] Christopher, W. A., Procter, S. J., and Anderson, T. E. The nachos instructional operating system. In *USENIX Winter* (1993), pp. 481–488.

[2] Fankhauser, G., Conrad, C., Zitzler, E., and Plattner, B. Topsy - a teachable operating system, 2000. Online. Internet. September 7, 2001. Available WWW: `http://www.tik.ee.ethz.ch/~topsy/Book/Topsy\_1.1.pdf`.

[3] Lawton, K. bochs: The open source ia-32 emulation project (home page), 2001. Online. Internet. Available WWW: `http://bochs.sourceforge.net/`.

[4] Rosenblum, M., Herrod, S. A., Witchel, E., and Gupta, A. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications 3*, 4 (Winter 1995), 34–43.

[5] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems: Design and Implementation*, second ed. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1997. Includes CD-ROM.