The Nachos Instructional Operating System

Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

In teaching operating systems at an undergraduate level, it is very important to provide a project that is realistic enough to show how real operating systems work, yet simple enough that the students can understand and modify it in significant ways. A number of these instructional systems have been created over the last two decades, but recent changes in hardware and software design, along with the increasing power of available computational resources, have changed the basis for many of the tradeoffs made by these systems.

We have implemented an instructional operating system, called Nachos, and designed a series of assignments to go with it. Our system includes CPU and device simulators, and runs as a regular UNIX process. Nachos illustrates and takes advantage of modern OS technology, such as threads and remote procedure calls, recent hardware advances, such as RISC's and the prevalence of memory hierarchies, and modern software design techniques, such as object-oriented programming and distributed computing.

We have used Nachos in the undergraduate operating systems class at Berkeley, with positive results. Nachos is freely available, and we would like to see it widely used for undergraduate instruction.

1 Introduction

In undergraduate computer science education, course projects provide a useful tool for teaching basic concepts and for showing how those concepts can be used to solve real-world problems. A realistic project is especially important in undergraduate operating systems courses, where many of the concepts are best taught, we believe, by example and experimentation.

This paper discusses an operating system, simulation environment, and set of assignments that we developed for the undergraduate operating systems course at Berkeley.

Over the years, numerous projects have been developed for teaching operating systems; among the published ones are Tunis [Holt 1983] and Minix [Tanenbaum 1987b, Aguirre et al. 1991]. Many of these projects were motivated by the development of UNIX [Ritchie & Thompson 1974] in the mid 70's. Earlier operating systems, such as MULTICS [Daley & Dennis 1968] and OS/360 [Mealy et al. 1966] were far too complicated for an undergraduate to understand, much less modify, in a

A copy of Nachos can be obtained by anonymous ftp from sprite.berkeley.edu, file "nachos/nachos-2.0.tar". The authors' e-mail addresses are {faustus,procter,tea}@cs.berkeley.edu. This work was supported in part by a grant from the College of Engineering, University of California at Berkeley.

semester. Even UNIX itself is too complicated for this purpose, but UNIX showed that operating systems need only a few simple but powerful interfaces, and that the core of an operating system can be written in only a few dozen pages [Lions 1977]. Indeed, the project previously used at Berkeley, the TOY Operating System, was originally developed by Brian Kernighan in 1973.

The introduction of minicomputers, and later, workstations, also aided the development of instructional operating systems. Rather than having to run the operating system on the bare hardware, computing cycles became cheap enough to make it feasible to execute an operating system kernel using a simulation of real hardware. The operating system can run as a normal UNIX process, and invoke the simulator when it would otherwise access physical devices or execute user instructions. This vastly simplifies operating systems development, by reducing the compile-execute-debug cycle and by allowing the use of off-the-shelf symbolic debuggers. Because of these advantages, many commercial operating system development efforts now routinely use simulated machines [Bedichek 1990].

However, recent advances in operating systems, hardware architecture, and software engineering have left many operating systems projects developed over the past two decades out of date. Networking and distributed applications are now commonplace. Threads are crucial for the construction of both operating systems and higher-level concurrent applications. And the cost-performance tradeoffs among memory, CPU speed and secondary storage are now quite different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

For these reasons, we decided to design and implement a new teaching operating system and simulation environment. Our system, called Nachos, makes it possible to give assignments that require students to write significant portions of each of the major pieces of a modern operating system: thread management, file systems, multiprogramming, virtual memory, and networking. We use these assignments to illustrate concepts that we believe are necessary to understand the computer systems of today and of the future: concurrency and synchronization, caching and locality, the tradeoff between simplicity and performance, building reliability from unreliable components, dynamic scheduling, the power of a level of translation, distributed computing, layering, and virtualization.

In building Nachos, we were continually faced with a tradeoff between simplicity and realism. For example, a complete UNIX-like file system would be too complicated for students to understand in only a few weeks. Our approach was to build the simplest implementation we could think of for each sub-system of Nachos; this provides students a working example, albeit overly simplistic, of the operation of each component of an operating system. The assignments ask the students to add functionality to this bare-bones system and to improve its performance on micro-benchmarks that we provide. As a result of our emphasis on simplicity, the Nachos operating system is about 2500 lines of code, about half of which are devoted to interface descriptions and comments.[1] It is thus practical for students to read, understand, and modify Nachos during a single semester course.

We have used Nachos for one term as the project for the undergraduate operating systems course at Berkeley; we then revised both the code and the assignments based on our experiences with using it. Nachos currently runs only on DEC MIPS workstations, but we believe that it would be straightforward to port to other platforms.

The rest of this paper describes Nachos in more detail. Section 2 provides an overview of Nachos; Section 3 describes the Nachos assignments. Sections 4 and 5 summarize our experiences.

[1] The hardware simulator takes up another 2500 lines, but students need not understand the details of its operation.

2 Nachos Overview

Like many of its predecessor instructional operating systems, the Nachos kernel and hardware simulator run together in the same UNIX process.[2] Nachos has several significant differences with earlier systems:

- Because we simulate a standard, well-documented, instruction set (MIPS R2/3000 integer instructions [Kane 1987]), we can run normal C programs as user programs on our operating system. In the past, operating systems projects typically simulated their own ad hoc instruction set, requiring user programs to be written in assembly language. Because the R2/3000 is a RISC, it can be simulated with only about 10 pages of code.
- We accurately simulate the behavior of a network of workstations, each running Nachos. We connect Nachos "machines", each running as a UNIX process, together via sockets. A thread on one "machine" can then send a message to a thread running on a different "machine"; of course, both are simulated on the same physical hardware.
- Our simulation is deterministic. Debugging non-repeatable execution sequences is a fact of life for professional operating systems engineers, but it did not seem advisable for us to make that experience our students' first introduction to operating systems. Instead of using UNIX signals to simulate asynchronous devices such as the disk and the timer, Nachos maintains a simulated time that is incremented whenever a user program executes an instruction and whenever a call is made to certain low-level operating system routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point.[3]
- Our simulation is also randomizable to add unpredictable, but repeatable, behavior. For instance, the network simulation randomly chooses which packets to drop; provided the initial seed to the random number generator is the same, however, the behavior of the system is repeatable.
- Nachos is implemented in a subset of C++. Object-oriented programming is becoming more popular, and we found that it was a natural idiom for stressing the importance of modularity and clean interfaces in building operating systems. To simplify matters, we omitted certain aspects of the C++ language: derived classes, operator and function overloading, and C++ streams. We also kept inlines to a minimum. Although our students did not know C++ before taking our course, we found that they learned the language very easily.
- The Nachos assignments take a quantitative approach to operating system design. Frequently, the choice of how to implement some piece of operating system functionality comes down to a tradeoff between simplicity and performance. We believe that teaching students how to make informed decisions about tradeoffs is one of the key roles of an undergraduate operating systems course. The Nachos hardware simulation reflects current hardware performance characteristics;

[2] Minix takes the different approach of running directly on personal computers. While this approach is more realistic, it makes debugging more difficult.

[3] The one aspect of the simulation we did not make reproducible was the precise timing of network communications, but since this came at the end of the semester, it did not seem to cause problems.

we exploit this by having students measure the performance of their implementations on some simple workloads that we provide.

3 The Assignments

Nachos contains five major components, each the focus of one assignment given during the semester: thread management and synchronization, the file system, user-level multiprogramming support, the virtual memory system, and network support. Each assignment is designed to build upon previous ones; for instance, every part of Nachos uses thread primitives for managing concurrency. This reflects part of the charm of developing operating systems: you get to "use what you build."

In this section, we discuss each of the five assignments, including the hardware simulation facilities and the operating system structures we provide, along with what we ask the students to implement. Students worked in pairs, and we conducted 15 minute graded design reviews after every assignment. We found that the design reviews were very helpful at encouraging students to design before implementing.

3.1 Thread Management

The first assignment introduces the concepts of threads and concurrency. We provide students with a basic working thread system and an implementation of semaphores; the assignment is to implement Mesa-style locks and condition variables [Lampson & Redell 1980] using semaphores, and then to implement solutions to a number of concurrency problems using these synchronization primitives. For instance, we ask students to program a simple producer-consumer interaction through a bounded buffer, using condition variables to denote the "buffer empty" and "buffer full" states.

In much the same way as pointers for beginning programmers, understanding concurrency requires a conceptual leap on the part of students. Contrary to Dijkstra [Dijkstra 1989], we believe that the best way to teach concurrency is with a "hands-on" approach. Nachos helps in two ways. First, thread management in Nachos is *explicit*: students can trace, literally statement by statement, what happens during a context switch from one thread to another, both from the perspective of an outside observer and from that of the threads involved. We believe this experience is crucial to de-mystifying concurrency. Precisely because C and C++ allow nothing to be swept under the covers, concurrency may be easier to understand (although harder to use) in these programming languages than in those explicitly designed for concurrency, such as Ada [Mundie & Fisher 1986], Modula-3 [Nelson 1991], and Concurrent Euclid [Holt 1983].

Second, a working thread system, as in Nachos, allows students to practice writing concurrent programs and to test out those programs. Even experienced programmers find it difficult to think concurrently; a widely used OS textbook had an error in one of its concurrent algorithms that went undetected for several years. When we first used Nachos, we omitted many of the practice problems we now include, thinking that students would see enough concurrency in the rest of the project. In retrospect, the result was that many students were still making concurrency errors even in the final phase of the project.

Our thread system is based on FastThreads [Anderson et al. 1989]. Our primary goal was simplicity, to reduce the effort required for students to trace the behavior of the thread system.

Our implementation is a total of about 10 pages of C++ and a page of MIPS assembly code. For simplicity, thread scheduling is normally non-preemptive, but to emphasize the importance of critical sections, we have a command-line option that causes threads to be time-sliced at "random", but repeatable, points in the program. Concurrent programs are correct only if they work when "a context switch can happen at any time".

3.2 File Systems

Real file systems can be very complex artifacts. The UNIX file system, for example, has at least three levels of indirection — the per-process file descriptor table, the system-wide open file table, and the in-core inode table — before one even gets to disk blocks. As a result, in order to build a file system that is simple enough for students to read and understand in a couple of weeks, we were forced to make some hard choices as to where to sacrifice realism.

We provide a basic working file system that is as stripped of as much functionality as possible. While the file system has an interface similar to that of UNIX [Ritchie & Thompson 1974] (cast in terms of C++ objects), it also has many significant limitations with respect to commercial file systems: there is no synchronization (only one thread can access the file system at a time), files have a very small maximum size, files have a fixed size once created, there is no caching or buffering of file data, the file name space is completely flat (there is no hierarchical directory structure), and there is no attempt at providing robustness across machine and disk crashes. As a result, our basic file system takes only about 15 pages of code.

The assignment is first, to correct some of these limitations, and second, to improve the performance of the resulting file system. We list a few possible optimizations, such as caching and disk scheduling, but it is up to the students to decide which are the most cost-effective for our benchmark (the sequential write and then read of a large file).

At the hardware level, we provide a disk simulator, which accepts "read sector" and "write sector" requests and signals the completion of an operation via an interrupt. The disk data is stored in a UNIX file; read and write sector operations are performed using normal UNIX file reads and writes. After the UNIX file is updated, we calculate how long the simulated disk operation should have taken (from the track and sector of the request), and set an interrupt to occur that far in the future. Read and write sector operations (emulating hardware) return immediately; higher level software is responsible for waiting until the interrupt occurs.

We made several mistakes along the way of developing the Nachos file system. In our first attempt, the file system was much more realistic than the current one, but it also took more than four times as much code. We were forced to re-write it to cut it down to something that students could quickly read and understand. When we handed out this simpler file system, we did not provide enough code for it to be completely working, leaving out file read and write operations to be written by the students. Although these are fairly straightforward to implement, the fact that our code did not work meant that students had difficulty understanding how each of the pieces of the file system fit together.

We also initially gave students the option of which limitations to fix; from our experience, we found that students learned the most from fixing the first four listed above. The result is that, even though virtually all modern file systems include some form of write-ahead logging or log-structure [Rosenblum & Ousterhout 1992], the assignment now completely ignores the issue of crash recovery. This is simply a tradeoff; in the limited time available, we focus on how basic file systems work, how the file abstraction allows disk data layout to be radically changed without changing the file system interface, and and how caching can be used to improve I/O performance.

3.3 Multiprogramming

In the third assignment, we provide the code to create a user address space, load a Nachos file containing an executable image into user memory, and then to run the program. Our initial code is restricted to running only a single user program at a time. Students expand on this base to support multiprogramming. Students implement a variety of system calls (such as UNIX fork and exec), as well as a user-level shell. We also ask them to optimize the multiprogramming performance of their system on a mixed workload of I/O- and CPU-bound jobs.

While we supply relatively little Nachos code as part of this assignment, the hardware simulation does require a fair amount of code. We simulate the entire MIPS R2/3000 integer instruction set and a simple single-level page table translation scheme. (For this assignment, a program's entire virtual address space must be mapped into physical memory; true virtual memory is left for assignment four.) In addition, we provided students an abstraction that hid most of the details of the MIPS object code format.

This assignment requires few conceptual leaps, but it does tie together the work of the previous two assignments, resulting in a usable, albeit limited, operating system. Because our simulator can run C programs, our students found it easy to write the shell and other utility programs (such as UNIX "cat") to exercise their system. (One overly ambitious student attempted to port emacs.) The assignment illustrates that there is little difference between writing user code and writing operating system kernel code, except that user code runs in its own address space, isolating the kernel from user errors.

One important topic we chose to leave out (again, as a tradeoff against time constraints) is the trend toward a small-kernel operating system structure, where pieces of the operating system are split off into user-level servers [Wulf et al. 1974]. Because of its modular design, it would be straightforward to move Nachos towards a small-kernel structure, except that (i) we have no symbolic debugging support for user programs and (ii) we would need a stub compiler to make it easy to make procedure calls across address spaces.

3.4 Virtual Memory

Assignment four asks students to replace their simple memory management code from the previous assignment with a true virtual memory system, that is, one that presents to each user program the abstraction of an (almost) unlimited virtual memory size by using main memory as a cache for the disk. We provide no new hardware or operating system components for this assignment.

The assignment has three parts. First, students implement the mechanism for page fault handling — their code must catch the page fault, find the needed page on disk, find a page frame in memory to hold the needed page (writing the old contents of the page frame to disk if it is dirty), read the new page from disk into memory, adjust the page table entry, and then resume the execution of the program. This mechanism can take advantage of what the students have built in previous assignments: the backing store for an address space can be simply represented as a Nachos file, and synchronization is needed when multiple page faults occur concurrently.

The second part of the assignment is to devise a policy for managing the memory as a cache — for deciding which page to toss out when a new page frame is needed, in what circumstances (if any) to do read-ahead, whether or not to write unused, dirty pages back to disk in advance to speed later page fault handling, and how many pages to bring in before initially starting to run a program [Levy & Lipman 1982, Leffler et al. 1989].

These policy questions can have a large impact on overall system performance, in part because of the large and increasing gap between CPU speed and disk latency — this gap has widened by two orders of magnitude in only the last decade. Unfortunately, the simplest policies often have unacceptable performance. To encourage students to implement realistic policies, the third part of the assignment is to measure the performance of the paging system on a benchmark we provide — a matrix multiply program where the matrices do not fit in memory. This workload is clearly not representative of real-life paging behavior, but it is simple enough that students can understand the impact of policy changes on the application. Further, the application illustrates some of the problems with caching — small changes in the implementation of matrix multiply can have a large impact on performance [Lam et al. 1991].

3.5 Networking

Although distributed systems have become increasingly important commercially, most instructional operating systems have not included any networking components. To address this, the capstone of the project is to write a significant and interesting distributed application.

At the hardware level, we simulate the behavior a network of workstations, each running Nachos, by connecting the UNIX processes running Nachos via sockets. The Nachos operating system and user programs running on it can communicate with other "machines" running Nachos simply by sending messages into the emulated network; the transmission is actually accomplished by socket send and receive. The Nachos network provides unreliable transmission of limited-size packets from machine to machine. The likelihood that any packet will be dropped can be set as a command-line option, as can the seed used to choose which packets are randomly chosen to be dropped. Packets are dropped but never corrupted, so that checksums are not required.

To demonstrate how to use the network and at the same time, how to take advantage of layering, we built a simple post office protocol on top of the network. The post office layer provides a set of "mailboxes" that serve to route incoming messages to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowlegements.

The assignment is first to implement protocol layers to provide for the reliable transmission of arbitrary-sized messages, and then to build a distributed application on top of that service. The fragmentation protocol is straightforward to implement — one need merely to split the message into pieces, add fragment serial numbers, and send them one by one. Reliability is more interesting, requiring a careful analysis and design to be implemented correctly.

The choice of how to complete the project is left up to the students' creativity. We did make a few suggestions: multi-user UNIX talk, a distributed file system with caching, a process migration facility, distributed virtual memory, a gateway protocol that is robust to machine crashes. Perhaps the most interesting application a student built was a distributed version of the "battleship" game, with each player on a different machine. This illustrated the role of distributed state, since each machine kept only its local view of the gameboard; it also exposed several performance problems in our hardware simulation code which we have since fixed.

Perhaps the biggest limitation of our current implementation is that we do not model network performance correctly, because we do not keep the timers on each of the Nachos machines synchronized with one another. There are well-known ways of doing this [Chandy & Misra 1981, Jefferson et al. 1987], but we have not implemented one of them yet. With this, we would have been able to include a benchmark of the student's network protocols; it would also allow students to implement a parallel algorithm (albeit using message-passing) as the final part of the project.

4 Lessons Learned

Designing and implementing Nachos taught us a lot about how instructional software should be put together, and provided insights on how students learn about complex systems. In this section, we discuss some of the lessons that we learned.

In devising the assignments, we had to decide which pieces of the Nachos code to provide students and which pieces to leave for students to write themselves. At one extreme, we could have provided students only the hardware simulation routines, leaving a *tabula rasa* for students to build an entire operating system from scratch. This seemed impractical, given the scope of what we wanted students to achieve during the semester.

Thus, when we taught the course for the first time, our goal was to provide students with the mundane and/or technically difficult parts of the operating system, such as generic list and bitmap management routines on the one hand, and low level thread context switch code on the other. We did this by writing the entire operating system from scratch, and then ripping out the parts that we thought students should write for themselves.

We found, however, that code (if simple enough), can be very useful at illustrating how some piece of the operating system should behave. The key is that the code has to be able to run standalone, without further effort on the part of students. Our thread system, although limited, could show exactly what happens when one thread relinquishes a processor to another thread. By contrast, when we provided students with less than a working file system, students had difficulty understanding how the pieces of the file system fit together. Similarly, we initially left to students the definition of the system call interface, including how parameters were to be passed from user code to the kernel. A simple example would have largely eliminated the resulting confusion.

Of course, reading code by itself can be a boring and pointless exercise; we addressed this by keeping our code as simple as possible, and by asking students to modify it in fairly fundamental ways. The result is that the assignments focus on the more interesting aspects of operating systems, where tradeoffs exist so that there is no single right answer.

Another lesson that we learned from using Nachos for a semester was the need to add a quantitative aspect to the assignments. We explicitly encouraged students to implement simple solutions to the assignments, to avoid sprawling complexity. But because we initially had no standard benchmarks for measuring the performance of student implementations, we had no counterbalance to show when complexity was justified. Students tended to devise overly simplistic solutions, where only a bit more effort was needed to be realistic. We hope that the performance tests that we've added will encourage students to identify which added complexity is justified by its benefits.

5 Conclusions

We have written an instructional operating system, called Nachos. It is designed to take advantage of advances in hardware and software technology, and to illustrate the principles of modern operating systems. These include concurrency, caching, and distributed computation. We have used Nachos for one semester in the undergraduate operating systems course at Berkeley, and the results were positive. We plan to use Nachos in future semesters, and we have made it publicly available in the hope that others will also find it useful.

6 Acknowledgements

We would like to thank the Spring 1992 CS 162 class at Berkeley for serving as guinea pigs while Nachos was under development. We would also like to thank Brian Bershad, Ed Lazowska, John Ousterhout, and Dave Patterson for their very helpful advice during the design of Nachos; John Ousterhout also wrote the MIPS simulator that we used. We credit Lance Berc with the acronym for Nachos: Not Another Completely Heuristic Operating System.

References

[Aguirre et al. 1991] Aguirre, G., Errecalde, M., Guerrero, R., Kavka, C., Leguizamon, G., Printista, M., and Gallard, R. Experiencing MINIX as a Didactical Aid for Operating Systems Courses. *Operating Systems Review*, 25:32–39, July 1991.

[Anderson et al. 1989] Anderson, T., Lazowska, E., and Levy, H. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Bedichek 1990] Bedichek, R. Some Efficient Architecture Simulation Techniques. In *Proceedings of the 1990 USENIX Winter Conference*, pp. 53–63, January 1990.

[Birrell 1989] Birrell, A. An Introduction to Programming with Threads. Technical Report #35, Digital Equipment Corporation's Systems Research Center, Palo Alto, California, January 1989.

[Chandy & Misra 1981] Chandy, K. and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, November 1981.

[Daley & Dennis 1968] Daley, R. and Dennis, J. Virtual Memory, Processes and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[Dijkstra 1989] Dijkstra, E. On the Cruelty of Really Teaching Computer Science. *Communications of the ACM*, 32(12):1398–1404, December 1989.

[Holt 1983] Holt, R. *Concurrent Euclid, the UNIX System, and TUNIS*. Addison-Wesley, 1983.

[Jefferson et al. 1987] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontabas, P., Laroche, P., Studevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 77–93, November 1987.

[Kane 1987] Kane, G. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[Lam et al. 1991] Lam, M., Rothberg, E., and Wolf, M. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, April 1991.

[Lampson & Redell 1980] Lampson, B. and Redell, D. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):104–117, February 1980.

[Leffler et al. 1989] Leffler, S., McKusick, K., Karels, M., and Quarterman, J. *Design and Implementation of the 4.3 BSD Unix Operating System*. Addison-Wesley, 1989.

[Levy & Lipman 1982] Levy, H. and Lipman, P. Virtual Memory Management in the VAX/VMS Operating System. *Computer*, pp. 35–41, March 1982.

[Lions 1977] Lions, J. A Commentary on the UNIX Operating System, June 1977. Department of Computer Science, University of New South Wales.

[McKusick et al. 1984] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[Mealy et al. 1966] Mealy, G., Witt, B., and Clark, W. The Functional Structure of OS/360. *IBM Systems Journal*, 5(1):3–51, January 1966.

[Mundie & Fisher 1986] Mundie, D. and Fisher, D. Parallel Processing in Ada. *IEEE Computer Magazine*, 19(8):20–25, August 1986.

[Nelson 1991] Nelson, G., editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Patterson 1992] Patterson, D. Has CS Changed in 20 Years? *Computing Research News*, 4(2):2–3, March 1992.

[Ritchie & Thompson 1974] Ritchie, D. and Thompson, K. The Unix Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Rosenblum & Ousterhout 1992] Rosenblum, M. and Ousterhout, J. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[Tanenbaum 1987a] Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

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

[Wulf et al. 1974] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–344, June 1974.