Appendix B

THE NACHOS SYSTEM

By Thomas E. Anderson

University of California, Berkeley

I hear and I forget, I see and I remember,

I do and I understand.

– Chinese proverb

A good way to gain a deeper understanding of modern operating-system concepts is to get your hands dirty — to take apart the code for an operating system to see how it works at a low level, to build significant pieces of the operating system yourself, and to observe the effects of your work. An operating-system course project provides this opportunity to see how you can use basic concepts to solve real-world problems. Course projects can also be valuable in many other areas of computer science, from compilers and databases to graphics and robotics. But a project is particularly important for operating systems, where many of the concepts are best learned by example and experimentation.

That is why we created *Nachos,* an instructional operating system intended for use as the course project for an undergraduate or first-year graduate course in operating systems. Nachos includes code for a simple but complete working operating system, a machine simulator that allows it to be used in a normal UNIX workstation environment, and a set of sample assignments. Nachos lets anyone explore all the major components of a modern operating system described in this book, from threads and process synchronization, to file systems, to multiprogramming, to virtual memory, to networking. The assignments ask you to design and implement a significant piece of functionality in each of these areas.

Nachos is distributed without charge. It currently runs on both Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations; ports to other machines are in progress. See Section B.4 to learn how to obtain a copy of Nachos.

Here, we give an overview of the Nachos operating system and the machine simulator, and describe our experiences with the example assignments. Of necessity, Nachos is evolving continually, because the field of operating systems is evolving continually. Thus, we can give only a snapshot of Nachos; in Section B.4 we explain how to obtain more up to date information.

B.1 ■ Overview

Many of the earliest operating-system course projects were designed in response to the development of UNIX in the mid-1970s. Earlier operating systems, such as MULTICS and OS/360, were far too complicated for an undergraduate to understand, much less to modify, in one semester.

Even UNIX itself is too complicated for that purpose, but UNIX showed that the core of an operating system can be written in only a few dozen pages, with a few simple but powerful interfaces. However, recent advances in operating systems, hardware architecture, and software engineering have caused many operating-systems projects developed over the past two decades to become 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 different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

Nachos is intended to help people learn about these modern systems concepts. Nachos illustrates and takes advantage of modern operating-systems technology, such as threads and remote procedure calls; recent hardware advances, such as RISCs and the prevalence of memory hierarchies; and modern software-design techniques, such as protocol layering and object-oriented programming.

In designing Nachos, we faced constantly the tradeoff between simplicity and realism in choosing what code to provide as part of the baseline system, and what to leave for the assignments. We believe that a course project must achieve a careful balance among the time that students spend reading code, that they spend designing and implementing, and that they spend learning new concepts. At one extreme, we could have provided nothing but bare hardware, leaving a *tabula rasa* for students to build an entire operating system from scratch. This approach is impractical, given the scope of topics to cover. At the other extreme, starting with code that is too realistic would make it easy to lose sight of key ideas in a forest of details.

Our approach was to build the simplest possible implementation for each subsystem of Nachos; this provides a working example — albeit an overly simplistic one — of the operation of each component of an operating system. The baseline Nachos operating-system kernel includes a thread manager, a file system, the ability to run user programs, and a simple network mailbox. As a result of our emphasis on simplicity, the baseline kernel comprises about 2500 lines of code, about one-half of which are devoted to interface descriptions and comments. (The hardware simulator takes up another 2500 lines, but you do not need to understand the details of its operation to do the assignments.) It is thus practical to read, understand, and modify Nachos during a single semester course. By contrast, building a project around a system like UNIX would add realism, but the UNIX 4.3BSD file system *by itself,* even excluding the device drivers, comprises roughly 5000 lines of code. Since a typical course will spend only about 2 to 3 weeks of the semester on file systems, size makes UNIX impractical as a basis for an undergraduate operating-system course project.

We have found that the baseline Nachos kernel can demystify a number of operating-system concepts that are difficult to understand in the abstract. Simply reading and walking through the execution of the baseline system can answer numerous questions about how an operating system works at a low level, such as:

- How do all the pieces of an operating system fit together?
- How does the operating system start a thread? How does it start a process?
- What happens when one thread context switches to another thread?
- How do interrupts interact with the implementation of critical sections?
- What happens on a system call? What happens on a page fault?
- How does address translation work?
- Which data structures in a file system are on disk, and which are in memory?
- What data need to be written to disk when a user creates a file?
- How does the operating system interface with I/O devices?
- What does it mean to build one layer of a network protocol on another?

Of course, reading code by itself can be a boring and pointless exercise; we addressed this problem by keeping the code as simple as possible, and by designing assignments that modify the system in fundamental ways. Because we start with working code, the assignments can focus on the more interesting aspects of operating-system design, where tradeoffs exist and there is no single right answer.

B.2 ■ Nachos Software Structure

Before we discuss the sample assignments in detail, we first outline the structure of the Nachos software. Figure B.1 illustrates how the major pieces in Nachos fit together. Like many earlier instructional operating systems, Nachos runs on a *simulation* of real hardware. Originally, when operating-system projects were first being developed in the 1970s and early 1980s, the reason for using a simulator was to make better use of scarce hardware resources. Without a simulator, each student would need her own physical machine to test new versions of the kernel. Now that personal computers are commonplace, is there still a reason to develop an operating system on a simulator, rather than on physical hardware?

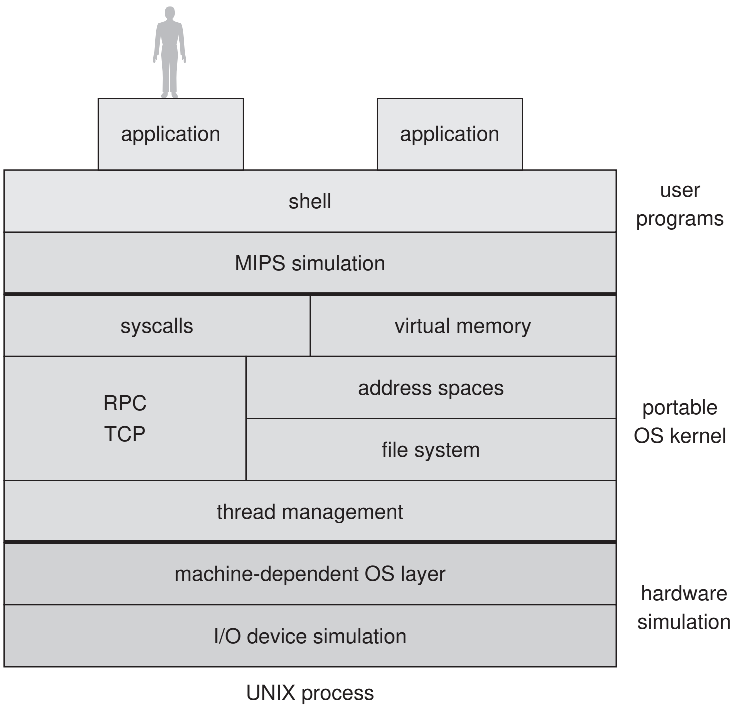

Figure B.1 How the major pieces in Nachos fit together.

We believe that the answer is yes, because using a simulator makes debugging easier. On real hardware, operating-system behavior is nondeterministic; depending on the precise timing of interrupts, the operating system may take one path through its code or another. Synchronization can help to make operating-system behavior more predictable, but what if we have a bug in our synchronization code such that two threads can access the same data structure at the same time? The kernel may behave correctly most of the time, yet crash occasionally. Without being able to repeat the behavior that led to the crash, however, it would be difficult to find the root cause of the problem. Running on a simulator, rather than on real hardware, allows us to make system behavior repeatable. Of course, debugging nonrepeatable execution sequences is part of life for professional operating-system engineers, but it did not seem advisable for us to make this experience part of anyone's introduction to operating systems.

Running on simulated hardware has other advantages. During debugging, it is important to be able to make a change to the system quickly, to recompile, and to test the change to see whether it fixed the problem. Using a simulator reduces the time required for this edit–compile–debug cycle, because otherwise the entire system has to be rebooted to test a new version of the kernel. Moreover, normal debugging tools do not work on operating-system kernels, because, for example, if the kernel stops at a breakpoint, the debugger cannot use the kernel to print the prompt for the next debugging command. In practice, debugging an operating-system kernel on real hardware requires *two* machines: one to run the kernel under test, and the other to run the debugger. For these reasons, many commercial operating-system development projects now routinely use simulators to speed development.

One approach would be to simulate the entire workstation hardware, including fetching, decoding, and executing each kernel- or user-mode instruction in turn. Instead, we take a shortcut for performance. The Nachos kernel code executes in native mode as a normal (debuggable) UNIX process linked with the hardware simulator. The simulator surrounds the kernel code, making it appear as though it is running on real hardware. Whenever the kernel code accesses an I/O device — such as a clock chip, a disk, a network controller, or a console — the simulator is invoked to perform the I/O activity. For instance, the simulator implements disk I/O using UNIX file routines; it implements network packet transfer via UNIX sockets.

In addition, we simulate each instruction executed in user mode. Whenever the kernel gives up control to run application code, the simulator fetches each application instruction in turn, checks for page faults or other exceptions, and then simulates its execution. When an application page fault or hardware interrupt occurs, the simulator passes control back to the kernel for processing, as the hardware would in a real system.

Thus, in Nachos, user applications, the operating-system kernel, and the hardware simulator run together in a normal UNIX process. The UNIX process thus represents a single workstation running Nachos. The Nachos kernel, however, is written as though it were running on real hardware. In fact, we could port the Nachos kernel to a physical machine simply by replacing the hardware simulation with real hardware and a few machine-dependent device-driver routines.

Nachos is different from earlier systems in several significant ways:

1. We can run normal compiled C programs on the Nachos kernel, because we simulate a standard, well-documented, instruction set (MIPS R2/3000 integer instructions) for user-mode programs. In the past, operating-system projects typically simulated their own ad hoc instruction set, requiring user programs to be written in a special-purpose assembly language. However, because the R2/3000 is a RISC, it is straightforward to simulate its instruction set. In all, the MIPS simulation code is only about 10 pages long.

2. We simulate accurately the behavior of a network of workstations, each running a copy of Nachos. We connect Nachos "machines," each running as a UNIX process, via UNIX sockets, simulating a local-area network. A thread on one "machine" can then send a packet to a thread running on a different "machine"; of course, both are simulated on the same physical hardware.

3. The simulation is deterministic, and kernel behavior is reproducible. 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 kernel routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point. At present, the precise timing of network packet delivery is not reproducible, although this limitation may be fixed in later versions of Nachos.

4. The simulation is randomizable to add unpredictable, but repeatable, behavior to the kernel thread scheduler. Our goal was to make it easy to test kernel behavior given different interleavings of the execution of concurrent threads. Simulated time is incremented whenever interrupts are enabled within the kernel (in other words, whenever any low-level synchronization routine, such as semaphore P or V, is called); after a random interval of simulated time, the scheduler will cause the current thread to be time sliced. As another example, the network simulation randomly chooses which packets to drop. Provided that the initial seed to the random number generator is the same, however, the behavior of the system is repeatable.

5. We hide the hardware simulation from the rest of Nachos via a machine-dependent interface layer. For example, we define an abstract disk that accepts requests to read and write disk sectors and provides an interrupt handler to be called on request completion. The details of the disk simulator are hidden behind this abstraction, in much the same way that

disk-device–specific details are isolated in a normal operating system. One advantage to using a machine-dependent interface layer is to make clear which portions of Nachos can be modified (the kernel and the applications) and which portions are off-limits (the hardware simulation — at least until you take a computer-architecture course).

B.3 ■ Sample 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 networking. Each assignment is designed to build on previous ones; for instance, every part of Nachos uses thread primitives for managing concurrency. This design reflects part of the charm of developing operating systems: You get to use what you build. It is easy, however, to change the assignments or to do them in a different order.

In Sections B.3.1 through B.3.5, we discuss each of the five assignments in turn, describing what hardware-simulation facilities and operating-system structures we provide, and what we ask you to implement. Of course, because Nachos is continuing to evolve, what is described here is a snapshot of what is available at the time of printing. Section B.4 explains how to obtain more up-to-date information.

The assignments are intended to be of roughly equal size, each taking approximately 3 weeks of a 15-week (semester) course, assuming that two people work together on each. The file-system assignment is the most difficult of the five; the multiprogramming assignment is the least difficult. Faculty who have used Nachos say that they find it useful to spend 1/2 to 1 hour per week discussing the assignments. We have found it useful for faculty to conduct a design review with each pair of students the week before each assignment is due.

Nachos is intended to encourage a quantitative approach to operating-system design. Frequently, the choice of how to implement an operating-system function reduces to a tradeoff between simplicity and performance. Making informed decisions about tradeoffs is one of the key tasks to learn in an undergraduate operating-system course. The Nachos hardware simulation reflects current hardware performance characteristics (except that kernel execution time is estimated, rather than being measured directly). The assignments exploit this feature by asking that you explain and optimize the performance of your implementations on simple benchmarks.

The Nachos kernel and simulator are implemented in a subset of C++. Object-oriented programming is becoming more popular, and it is a natural idiom for stressing the importance of modularity and clean interfaces in building systems. Unfortunately, C++ is a complicated language; thus, to simplify matters, we omitted certain aspects from standard C++: derived classes, operator and function overloading, C++ streams, and generics. We also kept inlines to a minimum. The Nachos distribution includes a short primer to help people who know C to learn our subset of C++; we have found that our students pick up this subset quickly.

B.3.1 Thread Management

The first assignment introduces the concepts of threads and concurrency. The baseline Nachos kernel provides a basic working thread system and an implementation of semaphores; the assignment is to implement Mesa-style locks and condition variables using semaphores, and then to implement solutions to a number of concurrency problems using these synchronization primitives.

In much the same way as understanding pointers can be difficult for beginning programmers, understanding concurrency requires a conceptual leap. We believe that a good way to learn about concurrency is to take a hands-on approach. Nachos helps to teach concurrency in two ways. First, thread management in Nachos is *explicit*: it is possible to trace, literally statement by statement, what happens during a context switch from one thread to another, from the perspectives of an outside observer and of the threads involved. We believe that this experience is crucial to demystifying concurrency. Precisely because C and C++ allow nothing to be swept under the carpet, concurrency may be easier to understand (although more difficult to use) in these programming languages than in those explicitly designed for concurrency, such as Ada or Modula-3.

Second, a working thread system, like that in Nachos, provides a chance to practice writing, and testing, concurrent programs. Even experienced programmers find it difficult to think concurrently. When we first used Nachos, we omitted many of the practice problems that we now include in the assignment, thinking that students would see enough concurrency in the rest of the project. Later, we realized that many students were still making concurrency errors even in the final phase of the project.

Our primary goal in building the baseline thread system was simplicity, to reduce the effort required to trace through the thread system's behavior. The implementation takes a total of about 10 pages of C++ and one page of MIPS assembly code. For simplicity, thread scheduling is normally nonpreemptive, but to emphasize the importance of critical sections and synchronization, 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.

B.3.2 File Systems

Real file systems can be 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 you even get to disk blocks. As a result, to build a file system that is simple enough to read and understand in a couple of weeks, we were forced to make some difficult choices about where to sacrifice realism.

We provide a basic working file system, stripped of as much functionality as possible. Although the file system has an interface similar to that of UNIX (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 at a time can access the file system), 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 to provide robustness across machine and disk crashes. As a result, the basic file system takes only about 15 pages of code.

The assignment is (1) to correct some of these limitations, and (2) to improve the performance of the resulting file system. We list a few possible optimizations, such as caching and disk scheduling, but part of the exercise is to decide which solutions are the most cost effective.

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 are 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 requests (emulating hardware) return immediately; higher-level software is responsible for waiting until the interrupt occurs.

We made several mistakes in 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 rewrite it to cut it down to code that could be read and understood quickly. When we handed out this simpler file system, we did not provide sufficient code for it to be working completely; we left out file read and file write to be written as part of the assignment. Although these functions are fairly straightforward to implement, the fact that the code did not work meant that students had difficulty understanding how each of the pieces of the file system fit with the others.

We also initially gave students the option of which limitation to fix; we found that students learned the most from fixing the first four listed. In particular, the students who chose to implement a hierarchical directory structure found that, although it was conceptually simple, the implementation required a relatively large amount of code.

Finally, many modern file systems include some form of write-ahead logging or log structure, simplifying crash recovery. The assignment now completely ignores this issue, but we are currently looking at ways to do crash recovery by adding simple write-ahead logging code to the baseline Nachos file system. As it stands, the choice of whether or not to address crash recovery is simply a tradeoff. In the limited amount of time available, we ask students to focus on how basic file systems work, how the file abstraction allows disk data layout to be changed radically without changing the file-system interface, and how caching can be used to improve I/O performance.

B.3.3 Multiprogramming

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

Although we supply little Nachos kernel 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 4.) In addition, we provide an abstraction that hides 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 the simulator can run C programs, it is easy to write utility programs (such as the shell or UNIX **cat**) to exercise the system. (One overly ambitious student attempted unsuccessfully 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 that 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. Because of Nachos' modular design, it would be straightforward to move Nachos toward a small-kernel structure, except that (1) we have no symbolic debugging support for user programs, and (2) we would need a stub compiler to make it easy to make remote procedure calls across address spaces. One reason for adopting a micro-kernel design is that it is easier to develop and debug operating-system code as a user-level server than if the code is part of the kernel. Because Nachos runs as a UNIX process, the reverse is true: It is easier to develop and debug Nachos kernel code than application code running on top of Nachos.

B.3.4 Virtual Memory

Assignment 4 is to replace the simple memory-management system 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. The first part is to implement the mechanism for page-fault handling — the kernel 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 the page frame 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 the code written for the previous assignments: The backing store for an address space can be represented simply 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, when to write unused dirty pages back to disk, and how many pages to bring in before starting to run a program.

These policy questions can have a large effect 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 past decade. Unfortunately, the simplest policies often have unacceptable performance. So that realistic policies are encouraged, the third part of the assignment is to measure the performance of the paging system on a matrix multiply program where the matrices do not fit in memory. This workload is not meant to be representative of real-life paging behavior, but it is simple enough to illustrate the influence of policy changes on application performance. Further, the application illustrates several of the problems with caching: Small changes in the implementation can have a large effect on performance.

B.3.5 Networking

Although distributed systems have become increasingly important commercially, most instructional operating systems do not have a networking component. To address this omission, we chose the capstone of the project to be to write a significant and interesting distributed application.

At the hardware level, each UNIX process running Nachos represents a uniprocessor workstation. We simulate the behavior of a network of workstations by running multiple copies of Nachos, each in its own UNIX process, and by using UNIX sockets to pass network packets from one Nachos "machine" to another. The Nachos operating system can communicate with other systems by sending packets into the simulated network; the transmission is accomplished by socket send and receive. The Nachos network provides unreliable transmission of limited-sized 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 determine which packets are "randomly" chosen to be dropped. Packets are dropped but are never corrupted, so that checksums are not required.

To show how to use the network and, at the same time, to illustrate the benefits of layering, the Nachos kernel comes with a simple post-office protocol layered on top of the network. The post-office layer provides a set of mailboxes that route incoming packets to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowledgments.

The assignment is first to provide reliable transmission of arbitrary-sized packets, and then to build a distributed application on top of that service. Supporting arbitrary-sized packets is straightforward — you need merely to split any large packet into fixed-sized pieces, to add fragment serial numbers, and to send the pieces one by one. Ensuring reliability is more interesting, requiring a careful analysis and design. To reduce the time required to do the assignment, we do not ask you to implement congestion control or window management, although of course these are important issues in protocol design.

The choice of how to complete the project is left open. We do make a few suggestions: multiuser 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 that a student built (that we know of) was a distributed version of the "battleship" game, with each player on a different machine. This application illustrated the role of distributed state, since each machine kept only its local view of the gameboard; it also exposed several performance problems in the hardware simulation, which we have since fixed.

Perhaps the biggest limitation of the 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. We are currently working on fixing this problem, using distributed simulation techniques for efficiency. These techniques will allow us to make performance comparisons between alternate implementations of network protocols; they will also enable us to use the Nachos network as a simulation of a message-passing multiprocessor.

B.4 ■ Information on Obtaining a Copy of Nachos

You can obtain Nachos by anonymous ftp from the machine ftp.cs.berkeley.edu by following these steps:

1. Use UNIX ftp to access ftp.cs.berkeley.edu:

   ftp ftp.cs.berkeley.edu

2. You will get a login prompt. Type the word anonymous, and then use your e-mail address as the password.

   Name: anonymous
   Password: tea@cs.berkeley.edu (for example)

3. You are now in ftp. Move to the Nachos subdirectory.

   ftp> cd ucb/nachos

4. You must remember to turn on "binary" mode in ftp; unfortunately, if you forget to do so, when you fetch the Nachos file, it will be garbled without any kind of warning message. This error is one of the most common that people make in obtaining software using anonymous ftp.

   ftp> binary

5. You can now copy the compressed UNIX tar file containing the Nachos distribution to your machine. The software will automatically enroll you in a mailing list for announcements of new releases of Nachos; you can remove yourself from this list by sending e-mail to nachos@cs.berkeley.edu.

   ftp> get nachos.tar.Z

6. Exit the ftp program:

   ftp> quit

7. Decompress and detar to obtain the Nachos distribution. (If the decompress step fails, you probably forgot to set binary mode in ftp in step 4. You will need to start over.)

   uncompress nachos.tar.Z
   tar -xf nachos.tar

8. The preceding steps will produce several files, including the code for the baseline Nachos kernel, the hardware simulator, documentation on the sample assignments, and the C++ primer. There will also be a README file to get you started: It explains how to build the baseline system, how to print out documentation, and which machine architectures are currently supported.

```
cat README
```

Mendel Rosenblum at Stanford has ported the Nachos kernel to run on Sun SPARC workstations, although user programs running on top of Nachos must still be compiled for the MIPS R2/3000 RISC processor. Ports to machines other than Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations are in progress. Up-to-date information on machine availability is included in the README file in the distribution. The machine dependence comes in two parts. First, the Nachos kernel runs just like normal application code on a UNIX workstation, but a small amount of assembly code is needed in the Nachos kernel to implement thread context switching. Second, Nachos simulates the instruction-by-instruction execution of user programs, to catch page faults and other exceptions. This simulation assumes the MIPS R2/3000 instruction set. To port Nachos to a new machine, we replace the kernel thread-switch code with machine-specific code, and rely on a C cross-compiler to generate MIPS object code for each user program. (A cross-compiler is a compiler that generates object code for one machine type while running on a different machine type.) Because we rely on a cross-compiler, we do not have to rewrite the instruction-set simulator for each port to a new machine. The SPARC version of Nachos, for instance, comes with instructions on how to cross-compile to MIPS on the SPARC.

Questions about Nachos and bug reports should be directed via e-mail to nachos@cs.berkeley.edu. Questions can also be posted to the alt.os.nachos newsgroup.

B.5 ■ Conclusions

Nachos is an instructional operating system designed to reflect recent advances in hardware and software technology, to illustrate modern operating-system concepts, and, more broadly, to help teach the design of complex computer systems. The Nachos kernel and sample assignments illustrate principles of computer-system design needed 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, object-oriented programming, the power of a level of translation, protocol layering, and distributed computing.

Familiarity with these concepts is valuable, we believe, even for those people who do not end up working in operating-system development.

Bibliographical Notes

Wayne Christopher, Steve Procter, and Thomas Anderson (the author of this appendix) did the initial implementation of Nachos in January 1992. The first version was used for one term as the project for the undergraduate operating-systems course at The University of California at Berkeley. We then revised both the code and the assignments, releasing Nachos, Version 2 for public distribution in August 1992; Mendel Rosenblum ported Nachos to the Sun SPARC workstation. The second version is currently in use at several universities including Carnegie Mellon, Colorado State, Duke, Harvard, Stanford, State University of New York at Albany, University of Washington, and, of course, Berkeley; we have benefited tremendously from the suggestions and criticisms of our early users.

In designing the Nachos project, we have borrowed liberally from ideas found in other systems, including the TOY operating system project, originally developed by Ken Thompson while he was at Berkeley, and modified extensively by a collection of people since then; Tunis, developed by Rick Holt [Holt 1983]; and Minix, developed by Andy Tanenbaum [Tanenbaum 1987]. Lions [1977] was one of the first people to realize that the core of an operating system could be expressed in a few lines of code, and then used to teach people about operating systems. The instruction-set simulator used in Nachos is largely based on a MIPS simulator written by John Ousterhout.

We credit Lance Berc with inventing the acronym "Nachos" Not Another Completely Heuristic Operating System.

Credits

This Appendix is derived from Christopher/Procter/Anderson, “The Nachos Instructional Operating System,” Proceedings of Winter USENIX, January 1993. Reprinted with permission of the authors.