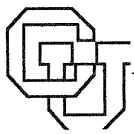
Windows NT in the OS Curriculum

Gary J. Nutt

CU-CS-880-99



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Windows NT in the OS Curriculum

Gary J. Nutt
Department of Computer Science, CB 430
Boulder, CO 80309-0430
University of Colorado
nutt@cs.colorado.edu

CU-CS-880-99 February 1999

University of Colorado at Boulder

Technical Report CU-CS-880-99
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Windows NT in the OS Curriculum

Gary J. Nutt
Department of Computer Science
University of Colorado

Abstract

There is a continuing evolution in the way operating systems are taught to undergraduates. While the focus on concepts grows with new technologies, there is also a strong trend toward incorporating significant laboratory materials into OS courses. These laboratory materials are usually based on a pedagogical system (such as Nachos) or a public domain version of UNIX (such as Linux). This papers describes our experiences with these types of systems compared with using the Win32 API and Windows NT in a first OS course.

1 Introduction

Today's undergraduate computer science curricula generally comply with (or are inspired by) the 1991 ACM/IEEE Curricula Recommendation [Tucker, et al., 1990]. The recommendation is intended to be a guideline for constructing classes at 2-year colleges, 4-year colleges, and Ph.D granting institutions. For example the appendix of the recommendation describes an OS class that might be well-suited to a Bachelor's curriculum in computer science. In an actual undergraduate class, the textbook is often the most influential element in defining the course content. The OS textbooks that appear to be most widely used in upper division undergraduate classes are those by Silberschatz and Galvin [1998] and Tanenbaum [1992].[1] While the authors of these books were undoubtedly influenced by the ACM/IEEE recommendation, they also factored in their own experience in selecting content, and in the amount of emphasis they placed on any given element. As instructors adopt a book, they tend to define the content of their course according to the content and balance of the book, i.e., instructors adopt the book that they feel is both pedagogically sound and which is the best match of material the teachers wants to discuss.

Since the mid 1980s there has been a trend to focus on general, conceptual material. "This book ... provides a clear description of the *concepts* [authors emphasis] that underlie operating systems. ¶In this book, we do not concentrate on any particular operating system or hardware. Instead we discuss fundamental concepts that are applicable to a variety of systems." Preface of [Silberschatz and Galvin, 1998]. Instructors that use Silberschatz and Galvin understand that if they desire a laboratory component to their course, they will have to augment the material in the textbook.

Tanenbaum has taken a different approach with some of his textbooks. *Operating Systems: Design and Implementation* introduced MINIX [Tanebaum, 1987], and the second edition [Tanenbaum and Woodhull, 1997] updates the discussion. In both the first and second edition books, though the focus is on the design and implementation of MINIX, concepts are introduced as a generalization of the software. However his *Modern Operating Systems* book (apparently his most popular book for undergraduate OS courses) "... began as an attempt to edit my earlier book, *Operating Systems: Design and Implementation*, and remove all the material on MINIX from it to make it suitable for traditional 'theory," as opposed to 'laboratory' courses." Preface to [Tanenbaum, 1992]. Regarding the ratio of space dedicated to the ACM/IEEE topics, Tanenbaum notes that he "... personally helped design and implement three different operating systems: TSS-11 (PDP-11), MINIX (IBM PC, Atari, Amiga, Maintosh, and SPARC), and Amoeba (80386, Sun-3, SPARC, and VAX)." Preface of [Tanenbaum, 1992]. The result was a book that addressed the topics from the ACM/IEEE recommendation, but proportional to their "... practical value in real systems ...". Where Silberschatz and Galvin used 644 pages to describe conceptual material, Tanenbaum used only 264 pages [Tanenbaum, 1992]. The MINIX book has a strong laboratory orientation, while the [Tanenbaum, 1992] is independent of any OS.

---

[1] Generally, publishers are reluctant to provide very many details about book sales, so our statements about a book's popularity are somewhat subjective.

The three textbooks [Silberschatz and Galvin, 1998], [Tanenbaum, 1992], and [Tanenbaum and Woodhull, 1997] illustrate the dilemma faced by the instructor of the first undergraduate OS course. How much emphasis should be placed on the laboratory aspect compared to the conceptual aspects? If there is an emphasis on the laboratory work, what target OS should be used? There appear to be three categories of OS courses being offered today, all complying with the ACM/IEEE recommendation:

- **Conceptual courses**. These courses follow the Silberschatz and Galvin model, spending essentially all the time on concepts. There is no significant laboratory component to such a course.
- **Laboratory courses**. These courses generally follow the MINIX model, spending almost all the time learning how MINIX, Linux, FreeBSD, Nachos, or some other OS is designed and implemented. Conceptual material is derived as an abstraction of the laboratory work.
- **Mixed courses**. These courses attempt to strike a balance (chosen by the instructor) for conceptual and laboratory material. They often use two textbooks – a "conceptual" book such as [Silberschatz and Galvin, 1998] or [Tanenbaum, 1992] supplemented by a book describing the target system such as [Beck et al., 1998] or [McKusick, et al., 1996].

In laboratory and mixed courses there is another important consideration: Should the course concentrate on teaching students about operating systems using an external or internal view?

- **The external view**. These courses are sometimes called "systems programming" courses since the exercises focus on learning about concepts by *using* the features of a particular OS. For example, the course may illustrate process management mechanisms by having students write software to use kernel fork, synchronization, and address space sharing mechanisms.
- **Kernel internals**. These courses concentrate on the design and implementation of a target OS, having the student read, modify, and augment the kernel code. There are two categories of internals targets: User space and kernel space systems.
    - **User space**. The target OS executes on a hardware simulator that executes with the CPU in user mode (e.g., Nachos). The simulator provides a software testbed for debugging and inspecting the target kernel.
    - **Kernel space**. These target kernels execute directly on the hardware (e.g., Linux, FreeBSD, or a kernel that the students build from scratch).

In the last few years we have written a textbook for undergraduate operating systems [Nutt, 1997], a Windows NT lab manual [Nutt, 1999]; we have also taught the undergraduate operating systems course at the University of Colorado as a "mixed course" with different ratios of conceptual/laboratory materials using both Linux (including internals) and Windows NT (using the Win32 API). Our colleagues at the University of Colorado have also used Nachos for the OS laboratory. We have reviewed the research distribution of the Windows NT source code as a potential basis of a laboratory component for an OS course. In the remainder of this paper we describe our experience with the OS courses, then offer some opinions regarding the suitability of Windows NT for both laboratory and mixed courses.

2 The OS Course

The undergraduate OS course at the University of Colorado is a required, 4 credit-hour, 15-week course for computer science majors, usually taken during the student's junior year. The prerequisites for the course are introductory programming, data structures, and a sophomore course in machine organization. Students sometimes enroll during their sophomore year, after having taken only the minimum prerequisites; other students take the course at the end of their junior year after they have gained substantially more programming experience. One of the difficulties in teaching this course is the diversity of backgrounds of the students. Students attend 3 hours of lectures per week (taught by an instructor), 1 hour of recitation (taught by a teaching assistant), and spend an unspecified amount of lab time (some of which is led by the teaching assistant).

The conceptual material presented in the class dominates the 45 hours of lecture time. In the 15 contact hours of recitation, all specialized information that students need to solve laboratory exercises is presented. This introduces a significant limitation to the amount of laboratory material that can be required of the students. Even though they are expected to spend considerable time outside of lecture and recitation time learning materials and writing solutions, any reasonable multiplier is going to restrict the total number of

hours they will apply to the class to something like 10-12 hours per week[2] As a result the course is offered with 4 to 9 assignments per semester, though usually no more than 4 of the assignments can be significant programming assignments (see Section 3 for additional comments on the amount of homework).

In the last four years, three different instructors at the University of Colorado have used [Silberschatz and Galvin, 1998] and [Nutt, 1997] – both of which focus on conceptual material. These courses all use the mixed course strategy, but with different target operating systems in the laboratory. Each time the course has been taught, the laboratory component has been substantial; students spent considerable time working on the laboratory assignments.[3] Since fall 1995, the course has been taught four times using a set of laboratory exercises based on Linux, twice using Nachos, and once using Windows NT.

Because of the place the OS course appears in our computer science curriculum, we must incorporate a certain amount of external view system programming. That is, many of the students who fulfill the prerequisites are simply not prepared to dive into a kernel for the first several weeks of the course, e.g., most students have never written a multithreaded program before taking the course.

2.1 Conceptual Exercises and Linux Exercises

In fall 1995 the course had 9 exercises. Approximately a third of these exercises were analytic exercises taken from the problems at the ends of chapters; another third had students design and analyze – but not implement – kernel components (e.g, simulate page replacement algorithm behavior); the last third of the exercises focused on the internal and external views of OS code:

1. Write a program to create another process to execute a program (i.e., write a simple shell program).
2. Learn how a Linux device driver is organized (i.e., design a device driver in pseudo code)
3. Design and implement a loadable device driver to implement a FIFO queue.

Only one assignment in this list requires the student to actually write kernel mode software – the FIFO device driver. The difficulty in the assignment is not in creating the device driver itself, but in learning all the details associated with device driver design: How to make this code be compatible with the Linux kernel interface, to learn what kernel facilities can be used by a device driver, to install a device driver, and to test the solution, i.e., the student learns only a limited amount about the kernel itself. In an attempt to give the student more exposure to the Linux kernel in spring 1996 (and fall 1997) we added another exercise:

4. Read the kernel timer code, then use it (with signals) to profile a simple application.

2.2 Focusing on Linux Laboratory Exercises

The exercises in the earlier Linux course instances were very time-consuming to solve, and the teaching assistant was spending an inordinate amount of time evaluating the conceptual homework. In the spring 1998 instance we decided to eliminate the conceptual problems (except on examinations, of course) and focus on laboratory assignments. We were also searching for opportunities for our students to learn more about kernel internals (even though we could also see that perhaps three-fourths of the students were not prepared to do so – see the experience with Nachos in Section 2.3).
In spring 1998 we eliminated all conceptual exercises and used the four exercises described in Section 2.1 along with these additional exercises:

5. Write a successive overrelaxation program (an exercise in shared information and synchronization).
6. Refine the shell program to support I/O redirection, pipes, and background processing.
7. Design and implement a simple user space file manager.
8. Design and implement a simple "talk" program using BSD sockets and TCP.

---

[2] In industry, it is often said that "good programmers" are up to 10 times more efficient than "bad programmers," suggesting that the range of hours a student spends on a laboratory exercise ranges from 3 to 30 hours per week.

[3] This observation is based on a significant number of informal comments from students as well as on the formal, anonymous comments provided by students on the faculty-course questionnaire administered at the end of each semester.

In this course instance, the driver assignments were given relatively early in the semester. Students struggled, but surprisingly, not substantially more than if they were given the assignment near the end of the term. We interpreted this as evidence that the conceptual knowledge required to write a device driver was not very significant, i.e., a student did not have to know many aspects of the kernel design to write a simple driver. Again, students spent most of their time on system administration type tasks, rather than on kernel internals or generalized device management.

In an attempt to provide more experience with internals issues (without modifying the kernel) the file manager assignment was designed to have students think about the internal design of a file manager. The assignment statement provided routines to read and write disk blocks; students were required to develop a simple file descriptor, directory operations, and file manipulation operations. The solution was very time consuming, though none of the design concepts were difficult: we concluded that the yield from this assignment (for the amount of work involved) was too low.

2.3 Nachos Laboratory Exercises

In fall 1996 and again in spring 1997 Nachos was used as the basis of our laboratory exercises. These instances used a subset of the exercises available with the Nachos distribution (no network exercises were assigned). In both these instances, the lectures focused on the conceptual OS material from the textbook, leaving the laboratory materials to the teaching assistant to handle within recitations. A quick review of the Nachos web page shows that the UC-Berkeley instances of the course created a strong link between the conceptual material provided in lecture and the laboratory provided in the recitation. That is, there appears to have been the same kind of integrated presentation in the Berkeley courses that a student might see if they were learning MINIX using [Tanenbaum and Woodhull, 1997].

Our experience was that the recitations were overloaded with material, and that only a TA with outstanding background and experience could make Nachos succeed in this lecture environment. The TA for the Fall 1996 course had an extensive OS background, but the TA for spring 1997 did not; we had to have the first TA help the second TA with Nachos halfway through the spring 1997 semester.

The comments of the experienced TA, and the written comments on the course questionnaires for both terms made clear that the background of the instructor and TA were crucial elements in the course. The top students were able to grasp the material, and became extraordinarily enthusiastic about kernels and Nachos. However the majority of the students were unable to master even a minimum threshold of the material and came away from the course without learning much about operating systems. The spring 1997 students were especially unhappy with the laboratory exercises. We concluded that while Nachos was a good alternative, it was difficult to assure that it would be compatible with the conceptual materials from a standard textbook.

2.4 Windows NT Laboratory Exercises

In fall 1998, the course used Windows NT as the laboratory system. Assignments were taken from the prepublication version of [Nutt, 1999]. All exercises used the Win32 API to experiment with various kernel concepts – the course used the externals approach exclusively. In this manual assignments use the `cmd.exe` interface, so they avoid using any of the USER and GDI kernel space components of Windows NT. An immediate difference between this course and the Linux and Nachos courses was that students no longer had any access to kernel source code. However, they were provided with a lab manual that explained various designs and implementations of concepts in the Windows NT Executive and Kernel related it the exercise that they were to solve.

The premise behind this switch to the pure external view was that we had not successfully introduced any substantial amount of kernel internals into the course, except in the Nachos instances. It was also apparent that students were not sufficiently well-versed in using the OS to study its internals. Finally, we were influenced by the growing importance of both the Windows NT technology and its commercial presence. Our students needed more understanding of the kinds of technology that are used in Windows NT than we had offered in previous instances of the course.

We assigned 8 exercises that addressed different aspects of the Windows NT Executive and Kernel:

1. Familiarization with the Visual C++ environment (most of our students had only used a UNIX environment before they took the class).
2. Use Windows NT tools to observe process, thread, and memory behavior.
3. Write a multiprocess and a multithread program (similar to the Linux shell assignment).
4. Use kernel object handles and waitable timers.
5. Use kernel event object for different forms of thread synchronization.
6. Use the memory-mapped file mechanism.
7. Design and implement a user-mode floppy disk driver and selected directory operations for a FAT12 file system.
8. Design and implement a simple "talk" program using BSD sockets and TCP (same problem as the Linux problem)

The first two exercises were very easy; students were able to complete them in minutes rather than hours. The first was intended only to be sure the students could use the Visual C++ environment with a console application. The second exercise was to ensure that they recognized the distinction between processes and threads, and would become familiar with various tools for inspecting the machine behavior. The remaining exercises all required that students learn how the Executive implemented some system functionality, then to write a Win32 API program to test the functionality. These assignments appear to have succeeded in teaching students about most aspects of process and thread management, device management, and file management.

However, it was challenging to derive exercises related to some aspects of the kernel. By using the Win32 API, it is difficult to provide an exercise in which students gain a real appreciation for how the virtual memory is designed an implemented, the details of how the scheduler works, and to appreciate the design of the Windows NT security mechanism. (On the other hand, we were not really able to assign exercises to study these notions when we had the Linux source code available either.) We did not use the Device Development Kit, since we did not feel we had adequate time for students to learn how to write kernel space drivers. Instead, we exploited the ability to use the `CreateFile` call on a floppy disk. Though this provides a byte stream interface, we were able to have students use this as if it were a raw disk I/O interface, thus the file management exercises were far more useful than the file manager exercise we tried with Linux.

Students were visibly excited about this instance of the class. They viewed this course as an opportunity to learn material that they thought would be especially relevant in the commercial world (i.e., they thought this experience would help them get a job). However there was some initial disappointment that they would not be looking at the kernel source code, though by the end of the term, the general feeling was that they would not have had time to learn enough about a kernel to study and change it.

3 Comparison

All instances of the course since fall 1995 have had a strong conceptual component, but with the laboratory component varying from Linux, to Nachos, to Win32 API Windows NT. Table 1 summarizes the results from the official course questionnaires for all instances of the course since fall 1995. Linux was the lab target system in fall 1995, spring 1996, fall 1997, and spring 1998. Nachos was the lab system in fall 1996 and spring 1997. Windows NT was used in fall 1998.

The course questionnaire provides about twenty different questions for which a student can select a letter grade. They also are provided with space to write comments regarding the least and most effective parts of the course, how to improve it, etc. The "Assignment Relevance" question asks students to rate the relevance of the assignments. "Course Value" asks for a rating on the value of the course to the student. "Recommend Course" asks if the student would recommend this course to other students. And the "Course Rating" reflects the students' overall rating of the course. The instructor row is provided to show which course instances were taught by the same instructor. These data illustrate the general student perception of the different instances, though student ratings are bound to reflect many other subjective concerns (e.g.,

how much they like the instructor and/or teaching assistant, or the quality of the lab equipment). The data are used as a quantitative means to compare the different strategies for the lab component of the course.

	Linux	*Linux*	*Nachos*	*Nachos*	*Linux*	*Linux*	*WinNT*
Term	Fall 95	Spr 96	Fall 96	Spr 97	Fall 97	Spr 98	Fall 98
Assignment Relevance	2.94	3.12	3.48	2.97	2.91	2.95	3.67
Course Value	3.47	3.19	3.30	2.59	2.91	3.12	3.23
Recommend Course	3.25	3.02	3.35	2.41	2.68	2.63	3.14
Course Rating	3.24	2.95	3.14	2.26	2.77	2.55	3.00
Instructor	N	N	E	B	N	N	N

Table 1: Course Questionnaire Results

The two Nachos classes used essentially the same assignments, so the ratings for the relevance of the assignments almost certainly reflects the instructor and TA's ability to teach the subject as opposed to the actual relevance of the assignements. In fall 1996, the TA made heavy use of the Berkeley material in recitations, and relied heavily on his own kernel internals experience These data support the hypothesis that students would be very happy to learn kernel internals if the material were presented in such a way that it was possible to master it in the limited time available. The fall 1996 TA stated that he believed that in his course, the best students could learn internals, but that the majority of the students had significant difficulties and were not able to learn as much he thought they should. The spring 1997 data supports the hypothesis that students will have a great deal of difficulty with an internals course if they do not have an extraordinary instructor and/or TA.

The results from the Linux courses indicate that the assignments were adequate, but not as good as well-presented kernel internals (Nachos in Fall 1996) labs or the Windows NT material. The course value, recommendation and ratings vary wildly across different instances with the same content/organization, so we infer that it probably provides a general indicator of the quality of the instructor and TA with only a suggestion of the relevance of the lab OS.

The Windows NT course has the highest assignment relevance of any instance of the course. The written comments on the questionnaires indicate that this high relevance is related to students learning about a technology for which they see an immediate use. The course value, recommendation and rating were also relatively high compared to Instructor N's other evaluations. We conclude that the focus on the external view of Windows NT (via Win32 API) provided a valuable learning environment – at least in their view – for the students in that course.

The experience with the various instances of the first course in operating systems has led us to believe that the conceptual material that appears in contemporary OS textbooks can be complemented with an externals view laboratory. If the laboratory component is to incorporate material on kernel internals, it is clear that the student must have learned the external view at some time prior to focusing on internals. In our curriculum, there is no assurance that students will learn how to use an OS before they enroll in the first OS course; we are obliged to provide that information in the course.

Once the decision has been made to incorporate external view material, the question is "which OS provides the best target?". There is an argument that the Windows NT is superior because of the breadth of technology it incorporates (that is visible from Win32 API). We believe that there are more than enough useful exercises for an entire semester course.

When we have used Linux as the laboratory system, it was apparent that the UNIX system call interface could be learned in less than a semester of exercises. Hence it is natural to attempt to complement the externals exercises with kernel exercises. However, the only reasonable module students can grasp in the

remainder of a semester is the simple device driver. But, most of the value in such an exercise is in system administration rather than learning kernel internals.

The unsatisfying aspect of this experience is that neither the Linux nor Windows NT approaches provide an adequate introduction to OS internals. With the Nachos approach, there has been concern about the difficulty of teaching the material, its user space approach, and the difficulty in creating new exercises. This situation has led us to consider an approach in which there is a second course in undergraduate OS that is explicitly an internals course.

4 A Second Course on OS Internals

In our curriculum much, if not all, of the time for exercises in the first course in OS has to address the external view. Only after undergraduates has seen the conceptual material and experimented with the system call interface are they prepared to study system internals. Therefore we advocate that the OS curriculum focus on the external view in the first course; this is completely consistent with the ACM/IEEE recommendation. However the curriculum should also include a second elective class which has the traditional class as its prerequisite, and which focuses on OS internals. The second course should presume the kind of experience gained in either the Windows NT instance or the Linux instance of the first course, i.e., it should provide the conceptual material and laboratories and ensure that students know how to use the OS. The second course should then be an intense internals course in which the student spends a full 12-15 hours per week studying how a particular OS implements processes, resource management, memory management, device drivers and file systems.

Since the second course is devoted to kernel internals, we think that the OS ought to be a kernel mode implementation. There an important argument against using a "real" OS for a course: The practical details of any real implementation can obscure the pure OS concepts. (It might be argued that this is yet another argument for why internals should not be studied until after the first course has been completed.) Today, a kernel internals course (that runs on the hardware using kernel mode) would likely be one of Linux or FreeBSD (although it could be a system designed as a pedagogical system, e.g., MINIX). There are good arguments for choosing either of these systems.

4.1 The Suitability of Windows NT for an Internals Course

Microsoft has been distributing the Windows NT source code to universities for use in research projects for a few years now. The license under which the source code is distributed explicitly forbids the use of the source code for educational purposes. This *research distribution* source code takes about 2.4 GB of disk space (it is distributed on 4 CD-ROMs). Before the source code can be compiled, the build machine must be configured to have an acceptable build environment, and have about 5 GB of available disk space. Under the current circumstances, even if Microsoft were to allow universities to use the source code for a internals class, it would be very difficult to use it in most university teaching labs.

However, from the perspective of a kernel internals course, the interesting parts of the research distribution are the code that make up the `ntoskrnl.exe` module – the part of Windows NT that executes in kernel mode. (This part of the source code is still an order of magnitude larger than, say, the Red Hat distribution of Linux.) Suppose that the source code to build the kernel was made available to universities, with the remainder of the Windows NT code distributed as binary code. What advantages or disadvantages would this distribution have over UNIX distributions for a kernel internals course?

The main advantage to using the Windows NT kernel is that its architecture and interface were chosen 15-20 years after the UNIX organization (that is not meant to say that a modern UNIX kernel looks just like the Version 6 kernel – Linux uses a newer code base than does the Windows NT kernel). The Windows NT kernel was designed as a Kernel and an Executive that are combined at compile time. This approach results in the generation of a monolithic runtime kernel, but one which is designed and analyzed as a microkernel and server. Further, the Kernel services are built around low level objects that can be used by the Executive to build OS objects with an omnipresent object interface with a built-in security mechanism. Students who learn kernel internals with this kind of model would be taught how to design systems that

divide functionality using a microserver/system strategy. There are two principle disadvantages to using the Windows NT kernel code: First, it is not available, and second, it is an order of magnitude larger than a UNIX kernel.

5 Conclusion

Since 1995 we have experimented with various versions of the first course in operating systems. Instructors' judgement (and the nature of OS textbooks) have encouraged us to design our course on top of a considerable body of conceptual information. Our experiments with the course have been to explore different ways to provide laboratory materials to our students. We have concluded that it is very difficult to incorporate much practice with kernel internals in the first course, since students need to spend a substantial amount of time learning the external view of an OS. Our preliminary experience with using the Win32 API to Windows NT has been very positive. It has caused us to provide a broader set of exercises for the external view than we used with UNIX-based systems, but it has also suggested that a second course is really required for those who wish to address kernel internals.

References

1. Beck, Michael, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Dirk Werworner, *LINUX Kernel Internels*, Second Edition, Addison Wesley, 1998.
2. McKusick, Marshall Kirk, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesly, 1996.
3. Nutt, Gary J., *Operating Systems: A Modern Perspective*, Addison Wesley, 1997.
4. Nutt, Gary J., *Operating Systems Projects for Windows NT*, Addison Wesley, 1999.
5. Silberschatz, Abraham and Peter Baer Galvin, *Operating System Concepts*, Fifth Edition, Addison Wesley, 1998.
6. Tanenbaum, Andrew S., *Modern Operating Systems*, Prentice Hall, 1992.
7. Tanenbaum, Andrew S., *Operating Systems: Design and Implementation*, Prentice Hall, 1987.
8. Tanenbaum, Andrew S. and Albert S. Woodhull, *Operating Systems: Design and Implementation*, Second Edition, Prentice Hall, 1997.
9. Tucker, Allen B., et al.. (editors), "Computing Curricula 1991: Report of the ACM/IEEE – Joint Curriculum Task Force," ACM and IEEE, December, 1990 (available at http://www.acm.org/education/curricula.html).