A Three Pronged Approach to Teaching Undergraduate Operating Systems

Steven Robbins
Department of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

ABSTRACT

This paper describes an approach to teaching an undergraduate operating systems course that relies on three aspects. First, a standard textbook is used for the basic theoretical material. Second, programming projects are used to reinforce some of the material covered from the textbook. Lastly, simulators are used to illustrate other material. A key to the approach is to use experimentation by the student to enhance understanding and prepare them for research.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*

General Terms

Experimentation

Keywords

operating systems, systems programming, curriculum

1. INTRODUCTION

The undergraduate operating systems course is taught in many formats. Courses range from purely theoretical, in which students do not use computers at all, to completely hands-on, in which they design and write an operating system or pieces of an operating system from scratch. This paper describes a course that is intermediate between these approaches. A standard operating systems book [4, 17, 18, 19] is used to cover the basic theoretical material. This is supplemented by a UNIX systems programming book [10] that emphasizes communication, concurrency, and threads. Programming projects play a major roll in the course and give students hands-on experience. The emphasis is on issues of concurrency and synchronization of communicating processes and threads. Lastly, there are some operating systems topics that do not easily lend themselves to programming projects. For these, simulators are used both to demonstrate principles and to provide a workbench for doing experiments. Having students propose hypotheses and then test them with experiments can lead to a deeper understanding and prepare students for doing research.

The rest of the paper is organized as follows. Section 2 discusses the format of the course. Section 3 describes the major topics covered in the course. Section 4 gives details on what is covered in the recitations. Section 5 describes the programming projects and section 6 describes the simulators. Finally, section 7 discusses some observations about the course.

2. COURSE FORMAT

Operating Systems is a required course for the undergraduate computer science degree at UTSA. Most of the required courses for the CS major, including Operating Systems, are 4 credits with a 3-hour per week lecture and a one hour per week recitation. The recitations are taught by graduate teaching assistants, usually first year PhD students. As in most CS programs, many of the graduate students are not native English speakers and some of them have difficulty teaching undergraduates. Few of them have any teaching experience. It is not always possible to find a teaching assistant that can do a good job running problems sessions or teaching concepts.

The recitations for the undergraduate Operating Systems course were designed so that the teaching assistants would not have to do any teaching. The recitations are self-paced labs in which the students are given detailed scripts to follow. The same scripts are used each year. Often the goal is to have the students try something out or learn how to use a tool. Each lab is designed to be completed in 50 minutes with no previous preparation, other than having attended the lecture and having a basic understanding of the course material to date. The recitation labs are graded on a credit/no credit basis. Each completed lab counts one point in the student's final average. The recitations are described in Section 4.

Students have access to labs containing PCs running Linux and Windows XP as well as Sparc computers running Solaris. All of the machines share a common file system containing the student accounts. Most of the PCs, including all of those in the teaching labs are dual-boot. The Linux and Solaris machines are available remotely through SSH.

3. THE CURRICULUM

The course uses two textbooks. The first can be any traditional undergraduate operating systems book [4, 18, 19], but currently we are using the text by Silberschatz, et al. [17]. We take most of our case studies and specific examples from UNIX. For this we use a UNIX systems programming book [10]. This book forms the basis for the programming assignments. The general topics of the course are shown in Figure 1. For each topic, the relevant chapters of the two books are given, along with which simulators and programming projects relate to that topic.

Topic	Text	Simulator	Programming
Early Systems	Notes		
Safety	USP 2		R02
Processes	OSC 3		
Scheduling	OSC 5	PS	P01
Process Creation	USP 3		P08
Unix I/O	USP 4	IO	
Files	USP 5		P04
Special Files	USP 6	ring	
Communication and Concurrency - rings	USP 7	ring	
Threads	USP 12	IO	P02, P03
Synchronization	OSC 6	PC, SP, ring	
Signals	USP 8		P05, P07, P08, P10, P11, P12
Network Communication	USP 18		P03-P08, P10-P14
Disk Head Scheduling	OSC 12	disk	
Memory	OSC 8,9	address	

Figure 1: The curriculum. The simulators are described in Section 6 and the programming projects refer to Figure 3. (USP = UNIX Systems Programming [10], OSC = Operating System Concepts [17])

4. THE RECITATIONS

Figure 2 lists the recitations that have been used. Often, there is not time for all of these since there is no recitation during the first week of the semester and another recitation is lost due to a holiday. One of the recitation weeks is used for catch up, in which a student can do the recitations that he/she has missed.

R01	Using the system
R02	Timing and static variables
R03	Process scheduling simulator
R04	Process scheduling simulator (continued)
R05	POSIX threads
R06	Ring simulator
R07	Ring simulator (continued)
R08	POSIX thread synchronization
R09	Producer-consumer simulator
R10	Network communication
R11	Disk head simulator
R12	Disk head simulator (continued)
R13	Address translation simulator

Figure 2: Recitation Topics

The complete set of recitations as well as the actual recitations used in each of the past 5 years are available online [15].

The recitations meet in a computer lab with one Linux computer per student. It would also work to have 2 students share a computer if resources are limited. The goal of the first recitation is to have the students successfully log onto the system and be able to compile programs. Of the other 12 recitations, four involve programming assignments and eight involve using simulators. Most of the programming assignments start with a working program that the students download and modify.

These labs could easily be used in an environment without a separate recitation. They could be used as weekly assignments since they are all self-paced. Most of the recitations require that the students create a web page containing their results. The created web pages could be used to check that they have been completed. The assignments have been designed to be graded pass-fail.

As an example of a programming recitation, in R08: POSIX thread synchronization, students download a simple program [10, Program 13.1] along with a test program and a makefile. They compile and run the program. Most students log on and complete this part in the first 5 minutes of the lab. Program 13.1 is a simple counter that can be incremented or decremented. It is made thread-safe by protecting the count with a single static POSIX mutex lock. The main program creates a number of threads that each increment the counter a fixed number of times. All of the threads are waited for and the final count value is displayed. If all goes well (as it should with the original program) the count will have a predicted value. The students then perform three types of experiments.

In the first experiment they remove (comment out) some of the mutex calls and see if this has an effect on the output. Surprisingly (or not) the locking seems to be unnecessary in that the correct results are still achieved, at least most of the time. This is an example of a program that (almost always) produces correct output, but is still an incorrect program. Students have a difficult time grasping this concept.

In the original program, the count is incremented with a single count++ statement. This may actually be atomic on some systems, while on others it still takes a very short time to execute, being implemented with 3 machine instructions. Failure can occur only if the CPU is lost during execution of this short code segment. In the second experiment students replace the `count++` line with three lines in which the count is first stored in a temporary variable, that variable in incremented, and then the result if stored back in count. This makes the unprotected code more likely to fail. Since they have both Linux and remote Solaris environments available, they do the tests on both systems and compare them.

In a third experiment, they add timing code to their main program to see what penalty is occurred when the mutex calls are used and calculate the time it takes to do one mutex lock and unlock pair.

R06 and R07 are examples of simulator labs. In the first of these, students learn how to use the fork-pipe simulator. In the first lab, they are guided through starting up and running the simulator on a simple example and using most of the features of the simulator in a closely guided procedure. In the second of these labs, students use the simulator to explore several methods of protecting critical sections. They explore the consequences of small modifications in the code, basically following the procedure from [10, Section 7.3].

5. PROGRAMMING PROJECTS

It is hard to engage students when operating systems is taught primarily as a theoretical course. Some concepts such as threads, I/O and networking lend themselves to programming assignments. Since concurrency and synchronization are underlying themes in operating systems, several assignments illustrate these concepts. The use of programming projects (other than to write or modify an operating system) in an operating systems course is not new [3, 16]. What is different here is the extensive use of projects and the way they are integrated into the course. One of the last topics in the course is network communication, and the projects usually end with an assignment that uses the UICI networking library [10, Chapter 18] which is described below. Network implementations often expose synchronization issues that were not present or not apparent in the non-networked version. A list of some of the projects used in the last ten years appears in Figure 3. All of these are available on the web [15].

P01	Four scheduling algorithms
P02	Comparing Java and POSIX threads
P03	Threaded network server
P04	File synchronization
P05	Parallel make
P06	Marshaling parameters
P07	License manager
P08	Proxy server
P09	Process ring
P10	Peer to peer communication
P11	Network audio server
P12	Web redirection
P13	Parallel calculator
P14	Network ring

Figure 3: Programming Projects

Until recently, the programming assignments each year have had a theme. Before starting the semester I would decide what the last programming assignment would be and then give preliminary assignments to build the support structure for this last assignment, giving it the flavor of a term project.

An example of this would be a simple parallel make. Programming in C on a UNIX system, the final program would be given a list of C source files, the first of which contains a main function. They would farm out the compilation of each source file into an object file. When they were completed, all of the object files would be linked together to give an executable. Early assignments involve parsing the input parameters (the source file names). They would compare the modification dates of the source and object files and only recompile those modules that were out of date. For each source file that needed to be compiled, they would pass the name to a separate process using pipes. Later they would use threads of a single process to do the compiles. These two implementations would provide the basis of a discussion of the differences between communication and synchronization using separate processes and threads with shared memory. Since the students are (at least they were until recently) running on machines with a single CPU, these implementations had little speedup. One of the last topics of the semester is network communication and they would modify their program to run on a network of workstations. These machines share a common file system through NFS, so only the names of the files need to be passed among machines. For extra credit, they could pass the files through the network.

The semester-long projects gave the course a unifying theme. Each project had several parts with due dates throughout the semester. These worked well for many years. A few years ago, however, UTSA switched to Java (from C) in the beginning programming courses, so students taking operating systems now have only one semester of C programming. Students with weak programming (or study) skills would often fall behind. Since each assignment built upon the previous one, these students often completely gave up on the programming projects. During the last year, I have moved to shorter, independent assignments, and this has worked out better for these students.

One such shorter assignment (P01 in Figure 3) requires students write code to produce a Gantt chart for each of four scheduling algorithms: first-come, first-served; shortest job first; preemptive shortest job first; and round robin. This assignment assumes that there are only two processes, each with two CPU bursts and one I/O burst. I have also introduced programming projects that used both Java and C. One such project (P02 in Figure 3) compares POSIX and Java threads. Students start by writing threaded programs in each language to calculate the average value of $\sin^2 x$ by summing random values. Students do this without any synchronization and determine whether the programs give the correct answer. After this is turned in, I distribute sample code so the students can use it in the next part of the assignment if their first part was not working. In the next part they add synchronization and compare the accuracy and timing with their earlier, non-synchronized, code.

5.1 Network Communication

We spend about one week of lectures near the end of the semester discussing network communication. UTSA has a separate networking course, but unlike operating systems, the networking course is not required of all majors. In operating systems we concentrate on the client-server model of communication, mainly connection-oriented communication. Earlier in the semester, the client-server model is introduced using named pipes (FIFOs) for communication between unrelated processes on the same machine. Some of the programming projects use pipes for communication and synchronization between processes.

Typically, there is a large amount of overhead in discussing how to write client-server code in a UNIX environment. The socket API uses complicated data structures to store connection information. Dealing with translations between host names and IP addresses as well as network byte order adds complications that are irrelevant to the higher level concepts. The goal is to get students to be able to use network programs that illustrate concurrency and synchronization issues that may not be apparent when programs are running on a single-CPU system, even if multiple processes or threads are being used.

UICI prototype	description (assuming no errors)
`int u_open(u_port_t port)`	creates a TCP socket bound to `port` and sets the socket to be passive returns a file descriptor for the socket
`int u_accept(int fd, char *hostn, int hostnsize)`	waits for connection request on `fd`; on return, `hostn` has first `hostnsize-1` characters of the client's host name returns a communication file descriptor
`int u_connect(u_port_t port, char *hostn)`	initiates a connection to server on port `port` and host `hostn`. returns a communication file descriptor

Figure 4: The UICI Interface

In the fall of 1991, I invented the UICI (Universal Internet Communication Interface) as part of a final exam problem. The goal was to abstract out the higher level concepts of network communication. Students could obtain file descriptors for doing network communication using an interface that is based on host names and port numbers. The current version of UICI is not much different from the one described in 1991. The UICI interface is shown in Figure 4. There are just three functions which take simple parameters, integers and strings. After a connection is set up, all communication is done with standard UNIX `read` and `write` calls. Students are given the source code for UICI and are encouraged to use it in other courses or after graduation.

6. THE SIMULATORS

Seven simulators are used to give students hands-on experience with several concepts that would otherwise need to be covered in a purely theoretical way. Each of the simulators can be run as a Java applet on the web or as a Java application with additional functionality. Running as an application gives the simulator access to the local machine so that local configuration is possible. Each simulator has a user's guide and is freely available on the web [14]. A brief explanation of each simulator follows.

6.1 Process Scheduling (PS)

The process scheduling (CPU scheduling) simulator [6] supports the following algorithms: first-come, first-served (FCFS); shortest job first (SJF); shortest job first approximation (SJFA); preemptive shortest job first (PSJF); preemptive shortest job first approximation (PSJFA); and round robin (RR). It allows users to specify probability distributions for CPU and I/O burst times for processes as well as interarrival times and durations. It will generate statistics and Gantt charts allowing students to compare the action of different algorithms with the same set of processes. Students experiment with this simulator in two recitations. The first recitation introduces them to the simulator and guides them through setting up a simple experiment. The second recitation requires more complex experiments comparing FCFS, SJF, and RR with three different values of the quantum. Students learn that as the quantum becomes larger, Round Robin approaches FCFS in performance.

After the recitation, students are given a project in which they are to use the simulator on their own. Several different projects have been developed. In some of these, students are given a statement and asked to design an experiment to test its validity. Some of the statements used include:

a) If 20 similar processes are scheduled using Round Robin (RR) with a small enough quantum, the result for one of the processes is similar to running it on a non-timesharing system with no other processes, but using a CPU that is 20 times slower.

b) Ignoring context switch time, the average waiting time for RR with a given quantum will be close to that of SJF when the quantum is small.

c) The average waiting time for RR increases (decreases) with increasing quantum.

In other types of projects students are asked to design an experiment that shows a particular behavior, such as one in which PSJF has less than half the average waiting time of SJF. To challenge the students I have asked them to design an experiment in which PSJF performs much worse (as measured by average waiting time) than SJF, or in which FCFS performs much worse than SJF. These seem counter-intuitive to most students (and to me also), but such experiments do exist. In all of the projects, the student needs to have a good understanding of the conditions under which each algorithm performs well.

I often use this simulator to introduce the concept of the load average, the average number of processes in the ready queue. UNIX systems sometimes report this value as an indication of the load on the CPU and traditionally use this value in calculating process priorities [5, Section 4.4]. Little's Law [2, Section 30.3] implies that the load average is the total waiting time divided by the time for the experiment. Since these two latter values are directly available from the simulator statistics, students can use these to calculate the load average. The load average is also useful in determining the cost of a context switch, since each context switch delays those processes that are ready.

Students can make predictions about how a non-zero context switch time will affect different scheduling algorithms. Often they predict that preemptive algorithms such as PSJF which generate more context switches will pay a bigger penalty as the context switch time grows than nonpreemptive algorithms (such as SJF). This is often not the case as PSJF will typically have a smaller load average than SJF. They can test this both analytically and by running the simulator.

6.2 Synchronization (PC)

This simulator [7] models a simple bounded buffer producer-consumer problem on a single CPU with one producer and one consumer. The code is similar to that given in several standard textbooks [17, page 192][18, page 221][19, page 108] and does not include any protection for the critical sections. This simulator is usually used as a demonstration of what can go wrong when shared variables are not protected. Students can run the simulator to completion or single step thought the code and force a context switch at any time. The internal state of the variables is always displayed. An interesting insight is that when a problem does occur and the internal state of the program becomes inconsistent, it may not have

an effect on the output generated (which items are consumed) until long after the failure has occurred. This is useful in illustrating how difficult it is to debug programs with synchronization errors, since by the time the error has been detected (if at all) it is long past the time at which the error occurred.

6.3 Starving Philosophers (SP)

This simulator [8] animates a standard solution to the dining philosophers problem using monitors. It is useful as a demonstration of how monitors are implemented using various queuing strategies and how processes move among these queues as the program progresses. It also illustrates how the standard solution to this problem allows for starvation, hence the name of the simulator.

Students can step through the program and see how the states of the processes (philosophers) vary. The entry and exit queues of the monitor are shown, as well as the waiting queue for each philosopher's condition variable. The simulator can also execute a more complicated solution that does not allow starvation.

6.4 Forks and Pipes (ring)

This simulates [9] a C program that uses `pipe`, `fork`, `dup2`, and `wait`. It allows students to write their own programs or run canned programs included with the simulator. The simulator provides a pictorial view of the relationship among the processes, pipes, and file descriptors as you single-step through the program. One of its main purposes is to show how redirection can be used to make several topologies of pipes connecting processes that have a common ancestor. One program is a ring of processes [10, chapter 7] that can communicate through pipes using standard input and output.

Before having access to this simulator, I would often draw diagrams on the board showing the results of various combinations of `fork`, `pipe`, `dup2` and `close`. It was not possible for the students to reproduce these diagrams in their notes, since `dup2` and `close` require that lines from the diagram be erased. The simulator allows a simple demonstration of these diagrams, which the students can reproduce at their leisure.

Back when computers were a lot slower, one of my standard examples would be a C program that would create a number of processes in a loop and then have each process print an identifying string. The order of the output would be different on each run and sometimes would be garbled as pieces of one output line would be intermingled with lines generated by other processes. With current machines which are much faster, such programs very rarely (one run in a hundred) garble lines and almost always produce the output in the same order. The simulator can be used to show what can happen. I use this as an example to illustrate that just because a program produces the correct output (almost) all of the time, this does not necessarily imply that the program is correct.

The simulator also supports loops, creating variables, assignment statements, printing to standard error, reading from standard input, writing to standard output, semaphores, and random number generation. The main example illustrates what can happen when a number of processes write to a shared resource, with or without exclusive access to that resource. Students can experiment with how scheduling of the processes affects the output generated and how the non-atomic printing in a C program can cause garbled output when no additional synchronization is used. The simulator is robust enough to illustrate a leader election algorithm on a ring of processes [1] [10, Section 7.8].

6.5 Disk Head Scheduling (disk)

Disk head scheduling is a topic that has traditionally been included in an operating systems course because the scheduling used to be done by the operating system. As disk drives became more complex, the internal structure of the disk became hidden from the operating system and scheduling was more appropriately done by the disk drive itself. The main ideas remain the same and most current operating systems books still address this issue [17, Section 12.4] [18, Section 11.5] [19, Section 5.4.3]. The simulator [11] allows students to obtain graphical and statistical information about the standard algorithms. It also allows examination of the consequences of having the operating system schedule the head movement when the geometry of the disk is different from the model that the operating system uses. This is most apparent when the drive itself remaps bad blocks in a way that is transparent to the operating system.

We use this simulator in two recitations. In the first recitation, the students use the simulator to reproduce the results from the examples in their text. They also make longer runs that would be difficult to trace by hand. Almost all textbook examples assume that all requests have been made before the algorithm executes. The simulator allows requests to come in at random times and the students can adjust the distribution of requests and arrivals. The second recitation has them explore the consequences of bad blocks that are transparently remapped by the disk. In this case a file that the operating systems considers contiguous may be fragmented on the disk. The students explore the relationship between the fraction of bad blocks and the disk access time.

6.6 Address Translation (address)

The address translation simulator [12] illustrates address translation in a demand paged virtual memory system with one or two-level page tables. The simulator has a number of canned examples for the students to work through and presents them in a randomized order. The students can turn in a generated HTML file containing their work. The simulator has a built-in help section that can either give hints or do the next step if students get stuck. Student's understanding of address translation (as indicated on exams) increased after we started using this simulator in the recitations.

The simulator presents information about the machine architecture including physical and logical address size, page table sizes and translation lookaside buffer (TLB) size. Students are given a binary logical address and must translate this into a physical address. They do this by segmenting the logical address and cutting and pasting the segments into parts of the simulator. For example, the simulator can display the TLB (which is just a table of 0's and 1's) and the student must find the appropriate entry in the TLB if it exists. First they must segment the TLB into page number and frame number by determining how many bits of each are in an entry. Then they paste the page number into the TLB, which highlights the corresponding frame number if it is found. If unsuccessful they must do lookups in one or more page tables. They can display the page tables and scroll through them until the appropriate entry is found, or paste the page number into the page table to highlight the corresponding entry. A sophisticated help facility is available if they get stuck. It can show the the steps they need to perform and which steps were successfully completed so far. It can give a description of the next thing to do, and can even perform the next step for them. The simulator keeps a record of all the steps performed including how much help was needed.

This simulator is best used to allow students to experiment with address translation and allow them to make as many mistakes and get as much help as needed until each translation is done successfully. I grade this on a credit/nocredit basis. They get credit as long as they complete a few single level problems and a few 2-level page table problems, even if they let the simulator do most of the work for them. Experience has shown that most students require little assistance from the simulator after they have used it for a while.

6.7 UNIX I/O (IO)

This simulator [13] demonstrates the relationship among several tables that are involved in UNIX I/O. It displays the contents of the file descriptor table, the system open file table, and the in-memory inode table as students single-step through programs. The simulator supports process (using fork) and thread (POSIX) creation. It is used mainly for demonstrations in class. A nice feature is the ability to step back through executed programs. This is very useful during classroom demonstrations to answer student questions about what happened in a previous step. At any point the state can be saved and later restored to show the result of a particular set of operations.

This simulator is useful for illustrating how the file offset behaves when a child process is created after a file is opened. The file offset in this case is a variable that is shared between processes. It is the simplest such example of a shared variable, since students only need to understand the basic UNIX I/O operations (open, read, write, close) and process creation (fork). Before this simulator was available, I had to draw complex diagrams on the board which were difficult for the students to absorb. With the simulator, they can rerun the demonstrations on their own and use the results as a basis for asking questions that would otherwise be hard to phrase.

7. CONCLUSIONS

This paper described an undergraduate operating systems course that combines traditional textbook material with programming projects and simulators to increase student understanding and experience. Although the course uses recitations to introduce the students to some of the simulators and short programming assignments, these recitations are self-paced and could be used as weekly assignments without a formal recitation.

Semester-long projects that build on previous assignments work well for strong, well-prepared and well-motivated students. However, weaker students or those with insufficient programming skills tend to not complete the early assignments and cannot continue with the later ones. Shorter, independent assignments work better for these students.

The course described here has evolved slowly over time so there is no way to compare student performance between this version of the course and a more traditional one. Almost all of our evaluations are anecdotal. Students seem to enjoy using the simulators and have made favorable comments about the programming assignments. The address translation simulator, in particular, has improved student performance on related questions on exams.

All of the material described here is freely available (except for the textbooks which most students still pay for). The most unique aspect of the course is the simulators. Instructors are encouraged to visit the web site [14] and try running the simulators directly from the browser. No installation or configuration is needed to try them. The simulators are independent and any combination of them can be easily incorporated into an existing operating systems course.

8. ACKNOWLEDGMENTS

The simulators and curriculum development for the course described here have been supported by several NSF grants: USE-0950407, DUE 9750953, DUE-9752165, DUE-0088769. Some of the projects described in Figure 3 were developed by K. Robbins.

9. REFERENCES

[1] A. Itai and M. Rodah, "Symmetry breaking in distributive networks," *Proc. Twenty-Second Annual IEEE Symposium on the Foundations of Computer Science*, 1981, pp.150-158.

[2] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc, 1991.

[3] G. Nutt, *Operating System Projects for Windows NT*, Addison-Wesley, 1998.

[4] G. Nutt, *Operating Systems, Third Edition*, Addison-Wesley, 2003.

[5] S. Leffler, M. McKusick, M. Karels, and J. Quartermain, *The Design and Implementation of the 4.3BSD UNIX Operating System,*, Addison Wesley, 1988.

[6] S. Robbins, S. and K. A. Robbins, "Empirical exploration in undergraduate operating systems," *Proc. 30-th SIGCSE Technical Symposium on Computer Science Education*, (1999) pp. 311-315.

[7] S. Robbins, "Experimentation with bounded buffer synchronization," *Proc. 31-st SIGCSE Technical Symposium on Computer Science Education*, (2000), pp. 330–334.

[8] S. Robbins, "Starving philosophers: Experimentation with monitor synchronization," *Proc. 32-nd SIGCSE Technical Symposium on Computer Science Education*, (2001), pp. 317–321.

[9] S. Robbins, "Exploration of process interaction in operating systems: A pipe-fork simulator," *Proc. 33-rd SIGCSE Technical Symposium on Computer Science Education*, (2002), pp. 351–355.

[10] K. Robbins and S. Robbins, *UNIX Systems Programming*, Prentice Hall, 2003.

[11] S. Robbins, S., "A disk head scheduling simulator," *Proc. 35-rd SIGCSE Technical Symposium on Computer Science Education*, (2004), pp 325–329.

[12] S. Robbins, "An address translation simulator," *Proc. 36-th SIGCSE Technical Symposium on Computer Science Education*, (2005), pp. 515–519.

[13] S. Robbins, "A UNIX concurrent I/O simulator," *Proc. 37-th SIGCSE Technical Symposium on Computer Science Education*, (2006), pp. 303–307.

[14] S. Robbins, Simulators for teaching operating systems, 2006. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/simulators**

[15] S. Robbins, Operating Systems Curriculum, 2008. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/OS**

[16] R. Rybacki, K. A. Robbins, and S. Robbins, "Ethercom: A study of audio processes and synchronization," *Proc. 24-th SIGCSE Technical Symposium on Computer Science Education*, (1993) pp. 218–222.

[17] A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts, Seventh Edition*, John Wiley and Sons, Inc, 2005.

[18] Stallings, W., *Operating Systems: Internals and Design Principles, 5th Edition,*, Prentice Hall, 2004.

[19] A. Tanenbaum, *Modern Operating Systems, Second Edition*, Prentice Hall, 2001.