Multithreaded Programming Can Strengthen an Operating Systems Course

Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
Houghton, MI 49931–1295, USA
E-mail: shene@mtu.edu

Abstract

Today, virtually all operating systems support multithreaded programming. In fact, threads are easier to use and more efficient than processes. This paper presents a possible way of using multithreaded programming to strengthen an operating systems course. More precisely, the lecture hours are divided into a theory track and a programming track. The former concentrates on the traditional topics, while the latter consists of seven programming assignments and one mini-project that can provide students with a comprehensive coverage of the use of multithreaded programming in the design of various operating system components. These assignments also serve as a vehicle for introducing interesting materials (*e.g.*, PRAM models) that are normally not available in a typical operating systems course.

1 Introduction

The concept of multithreaded programming first appeared in late 60's when IBM added the task feature and completion event variables into its PL/I F compiler and made the same available in all IBM VS (*i.e.*, virtual storage) operating systems as system calls. In early 80's, threading became popular in the Unix community. A number of multithreaded libraries for MS-DOS were published in trade magazines such as *Dr. Dobb's Journal* and *C Users Journal*. Well-known ones include [19, 31], which were available around 1995. However, multithreaded programming was not very popular in the PC community until Microsoft and IBM released Windows 95 and OS/2, respectively. Today, virtually all operating systems have multithreaded capability and the popular POSIX Pthreads standard is also available. Many well-known operating systems textbooks added sections on threads in their newest editions and numerous books about threads were published in recent years. Therefore, it is the time to teach students this new technology.

The best place to introduce the concept of threads is naturally an operating systems course for a number of reasons. First, the use of threads is much simpler than the use of processes. Second, multithreaded libraries, even the most primitive ones, support more synchronization primitives. Third, many implementations are at the user level, which means that all threads of a program run in the *same* address space, and hence debugging is easier. However, learning multithreaded programming requires a paradigm shift, which is not always easy for many students, and using synchronization primitives correctly without race conditions and deadlocks is very challenging. To smooth the transition from sequential to multithreaded and to make sure students will grasp the merit and skills of using synchronization primitives, topics must be covered in a slower pace than usual, and programming assignments must be chosen carefully so that students can practice and use multithreaded programming to solve a number of operating system related problems.

This paper presents the author's experience of teaching multithreaded programming in an operating systems course, and demonstrates the way of strengthening this course with this new technology. This course has been taught ten times in the past six years, and a preliminary report was published in [37]. The author's approach divides the lecture hours into a theory track and a programming track. The former covers most materials that can be found in a typical operating systems textbook. The latter, with the help of seven programming assignments and one mini-project, covers the concept, merit, and skills of multithreaded programming. In this way, students will not only learn multithreaded programming, but also be familiarized in many topics that are not available in a typical operating systems course.

In the following, Section 2 reviews some related work, Section 3 contains a course overview which includes a discussion of the theory and programming tracks, Section 4 presents a detailed discussion of programming assignments and the mini-project, Section 5 is an evaluation of our approach over the past four years, and Section 6 has some concluding remarks.

2 Previous Work

There are many ways of teaching an operating systems course. A popular one is the use of a pedagogical operating system (*e.g.*, MINIX, Nachos and their variants) for students to add, modify and/or extend some components [1, 25]. Some instructors believe that programming with a particular architecture directly (*e.g.*, the Intel platform) would benefit the students the most, and, as a result, students are asked to implement an architecture-dependent system [17, 28, 29]. Some others may use a hardware simulator, usually greatly simplified, to achieve the same goal [16, 33]. Yet another approach prefers to use a simple operating system, usually MS-DOS, to explore the system structure, system calls, and interrupt vectors [27, 32, 42]. There also are simulators and visualization tools developed for teaching a number of components of an operating systems such as CPU schedulers and page replacement algorithms [30, 36]. These approaches permit students to implement some operating system components. However, C. A. R. Hoare once pointed out that "you cannot teach beginning programmers top-down design because they do not know which way is up." It would be difficult for students to design and implement a system component correctly, if

they do not know how to use that component correctly and efficiently. This is particularly true for implementing synchronization primitives.

Ever since Brinch Hansen's seminal work [7], concurrent programming and synchronization have occupied a permanent place in an operating systems course. Along this line of development, some instructors prefer to use a system interface directly and others choose to use a high-level programming language. The most commonly seen approach of using a system interface usually involves the `fork()` and `exec()` system calls and message queues of Unix and/or Linux [18, 42, 44]. The shared memory and semaphores of Unix are rarely touched upon, perhaps because the Unix semaphore feature is very difficult to comprehend and be used properly. The language approach is also pioneered by Brinch Hansen with his Concurrent Pascal [8]. In the CS education community, Ben-Ari's interpreter [4] and its variants [11, 12, 35] are very popular and widely used. Instructors taking this approach emphasize concurrent execution and mutual exclusion built upon a number of popular synchronization primitives (*i.e.*, semaphores, monitors and CSP channels). The language SR was also used [20]; however, Java has been gaining some momentum in recent years [21, 22, 23, 24] although its parallelism mechanism is insecure [9]. Recently, instructors have started to use thread programming interfaces. For example, Berk [5] uses SunOS's light-weight process library, Downey [18] uses Pthreads, and the author uses Solaris thread library [37, 38].

The language approach is more structured, robust and safer, because the compiler and language syntax can prevent many unwanted programming errors and allow a user to concentrate on handling concurrency and synchronization. On the other hand, the thread programming interface approach requires a user to call a number of functions to achieve a task that can be done with one statement in a language that supports concurrency. We take the thread programming interface approach. We believe that before a student can implement a system component, s/he must have a reasonable understanding of the use and the meaning of that component. Fortunately, this can usually be done at the user-level rather than digging into the system itself. Thus, programming assignments are designed to fully explore a particular thread programming interface (Sun Solaris threads) with challenging problems that cover not only concurrency and synchronization but also multiprocess programming and signal handling. After students are well-equipped with all fundamental programming skills and the knowledge of these system components, they are provided with a small user-level kernel that supports non-preemptive multithreaded programming for a two-week mini-project. With this small kernel, students can implement thread creation, termination, joining, suspension and resume, thread scheduling with and without priority, and various synchronization primitives. We believe that in an introduction course, this approach has the unique advantage that students not only learn operating systems from a user's point of view but also have an opportunity to implement a significant portion of a kernel. In this way, students will not be limited to the theory nor overwhelmed by the implementation of a significantly larger system (*e.g.*, if they failed in an early stage, it would be difficult to recover and catch up in later stages).

3 Course Overview

This course, Introduction to Operating Systems, is divided into two tracks: *theory* and *programming*. The theory track, which consumes approximately two-third of the lecture hours, covers most traditional topics (*e.g.*, processes and threads, virtual memory, input/output, device management and file systems) [39, 40, 41]. Parallel to the theory track, the programming track focuses on programming skills that can reinforce the understanding of the theory and offer a chance for introducing important topics in related areas (*e.g.*, parallel programming). Combining both tracks, students will learn much more than they can in a typical operating systems course. Table 1 shows the relationship between these two tracks. In the following, Section 3.1 provides some background information, and Section 3.2 and Section 3.3 discuss the theory and programming tracks, respectively. Note that we only concentrate on the topics that are directly related to multiprocess and multithreaded programming.

Table 1: Relationship among programming and course topics

Week	*Course Topics*	*Programming*	*Exam*
1	Introduction		
2	Processes and Threads		
3	Synchronization Primitives	Warm-up	
4			
5			
6		Semaphore	Exam 1
7	Process Scheduling		
8	Process vs. Threads	Monitor	
9	Memory Management	Message Passing	
10		Shared Memory	
11			Exam 2
12		Scheduler	
13	Input/Output and File System	Disk Scheduling	
14			
15		Mini-Project	

3.1 Background Information

Introduction to Operating Systems is a 3-credit junior level course for juniors and seniors. Course prerequisites include C/C++, data structures, and computer architectures. There is a senior and graduate elective course Operating Systems for those who wish to learn more. We also have a graduate level course Distributed Systems. Therefore, this course serves as the entry point of all subsequent system related courses, including a Computer Network course.

The working environment includes Sun workstations running Solaris and Linux workstations. However, except for a few process related assignments, students' programs are run and graded on the Sun system because Solaris thread is a very robust system.

3.2 The Theory Track

As shown in Table 1, the theory track covers topics in a traditional operating systems course. We emphasize threads instead of processes because the former is easier to use and more efficient. Our experience shows that it is difficult and takes time for students to change their mindset from sequential to multithreaded. Even more difficult is the correct use of synchronization primitives. Hence, four weeks are allocated to a very detailed and thorough discussion of synchronization primitives. Topics include critical sections, mutual exclusion, busy waiting, simple hardware and software locks, semaphores, monitors, message passing, and a number of classic problems. In the past six years of teaching this course more than ten times, we found that the most commonly seen problem in student programs is the presence of race conditions rather than deadlocks. Most textbooks only provide a simple definition of race conditions with one or two trivial examples. Unfortunately, these unrealistic examples do not provide sufficient help for students to pinpoint and eliminate race conditions in their programs. Moreover, since general race condition detection is an NP-complete problem [34], efficient algorithms for precisely locating race conditions do not exist. To address this difficulty, we developed a set of materials [14] which are used along with the discussion of semaphores because race conditions can easily occur when semaphores are used.

Of the three popular synchronization primitives, the use of semaphores is the most difficult, because it is the first encountered synchronization primitive and because the use of semaphores is not as well-structured as monitors. We summarize three commonly used patterns: **(1)** locks, **(2)** counting-down counters, and **(3)** notification. Then, we identify these patterns in every semaphore related program. This approach has been very effective because the successful rate of solving semaphore related problems increases significantly.

Monitors are more structured, easier to use, and less prone to programming mistakes. Two types of monitors are covered: the Hoare style and the Mesa style. In the Hoare style, the signaler yields the monitor to the awoke thread, and, as a result, after the execution of a signal on a condition variable, the signaler and the awoke threads become inactive and active, respectively. On the other hand, in the Mesa style, the signaler continues to be active and the awoke thread becomes inactive (from waiting). Since monitor is usually not a programming interface feature, students must decompose a textbook structured example program into unstructured calls to a number of functions. Moreover, textbooks normally assume that monitors are of the Hoare style, and, to the best of the author's knowledge, most system level implementations of monitors use the Mesa style for efficiency. Therefore, students may be confused by what they learn in class and what they use in programming assignments. This is the main reason we discuss monitors of both types.

The next is the concept of synchronous and asynchronous channels. In addition to the theory and semantics of channels, we also emphasize that in a threaded or non-network

environment, a many-to-many channel (*i.e.*, multiple threads can send and receive messages to and from the same channel) is simply an instance of the bounded-buffer problem. We also point out that the message queue primitive available on Unix and its variants is a version of many-to-many asynchronous channels.

Barriers and reader-writer locks in both reader-priority and writer-priority versions are also mentioned. Some equivalence constructions among these primitives are established either in the programming track or in exams. For example, we may ask students to implement (1) semaphores using monitors or message queues, (2) FIFO semaphores using counting semaphores, (3) Unix semaphores using locks and condition variables (*i.e.*, monitors), (4) condition variables using counting semaphores, and (5) barriers using semaphores. As a result, students will know how to obtain a particular primitive when it is not available on a system.

3.3 The Programming Track

The programming track is designed to provide students with an opportunity to practice what they learn in the theory track, and to cover additional topics in system programming and topics that are normally not available in a typical operating systems course. Because the process dispatcher that supports multiprocess and multithreaded capability is part of the lowest layer of an operating system, students must know the use of multiprocess and multithreaded programming well before they can implement a threaded kernel correctly. As a result, we do not choose a pedagogical operating system (*e.g.*, MINIX and Nachos) for programming projects. Instead, we start with user-level multithreaded programming for students to familiarize the concept and various aspects of multithreaded programming.

To make sure that students will be equipped with sufficient knowledge and skills to start their first significant assignment, we take a very slow pace in the beginning and cover topics as detail as possible in the theory track. However, to stimulate students' interests, we give them early a simple warm-up assignment to gain some exposure in multithreaded programming. This strategy has been very successful because students are excited about the fact that a program can be split into multiple threads running concurrently.

Programming skills covered in this track include the use of Sun Solaris thread programming interface, building monitors using mutex locks and condition variables, channels, Unix processes and shared memory, signal handling, and non-local goto using ANSI C functions `setjmp()` and `longjmp()` and coroutines. The use of coroutines is worth mentioning in some details. With a function call, the control flow enters the callee from its very beginning and goes back to the next instruction of the caller. When a coroutine is called initially, the control flow enters the callee from the first instruction as usual. However, when the callee is called again, the control flow starts from the next instruction below the previous return point. Figure 1 shows three coroutines A, B and C. In the diagram, vertical arrows with different patterns indicate the execution flow of different functions, arrows in lighter color indicate control flow switches, and small circles are switching/returning points. Thus, this diagram shows the following activities: A executes (1), A calls B and B executes (2), B calls C and C executes (3), C calls A and A executes (4), and so on. As a result, A, B and C

can be considered as three threads and are context switched at the points indicated by the small circles. To implement a context switch at the user level, we need a jump buffer of type `jmp_buf` which is defined in the ANSI C header file `setjmp.h`. Then, `setjmp()` is called to save the execution environment before switching out and `longjmp()` is called to switch to a place recorded in a particular jump buffer. With this technique, we can implement a non-preemptive threaded system with minimal capability in about 100 lines! Note that a separate stack is required for each executing function. This is the only machine-dependent place in the implementation of coroutines; however, it is very easy to implement and only requires a couple of lines.

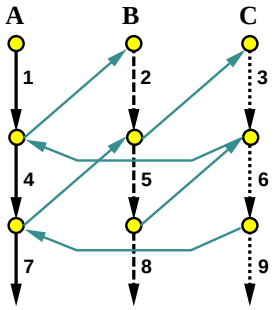

Figure 1: The concept of coroutines

All materials for the programming track are in electronic book form and available for student access on the Internet.

4 Programming Assignments

This section provides a detailed discussion of the seven programming assignments and one mini-project that form the core of the programming track. These programming assignments must have a broader impact on student learning, must be able to reinforce and enhance the theory track with an eye on system level programming practice, and must provide additional topics that are normally not covered in a typical operating systems course. Students must submit their programs and `README` files electronically. A `README` file must include the answers to a number of critical questions such as a discussion to show that his/her implementation does not have race conditions and deadlocks, an elaboration of the program logic, and a convincing argument to show the correctness of an implementation. For the mini-project, a `JOURNAL` file is also required to record a student's software design and implementation process. The journal events may include, but are not limited to, new design ideas, policy changes, mechanism modifications, bugs, bug reports and fixes. Students can organize these events in chronicle or problem-oriented order and include date, problem, impact on the project, solution, and other important information. We found out that most students kept very detailed and honest journals.

Because of the dynamic behavior of threads, grading student programs has always been a very challenging and difficult task. This means that we may encounter or discover a race

condition or deadlock that has never occurred in students' tests. It is also possible that we may not be able to duplicate race conditions or deadlocks reported by students. As a result, reading students' programs become necessary. Fortunately, many highly qualified undergraduate students who have had this course can be used as graders. Over the years, most graders expressed that they learned more in reading student programs because throughout the grading process they "know more variations in solving the same or a similar problem, and can sharpen their multithreaded programming skills."

The assignments are in three categories. Category 1, which includes assignments 1 to 4 (Section 4.1 to Section 4.4), is about the foundation of multithreaded programming. The design merit of these assignments is to provide a number of non-trivial exercises for students to familiarize and practice the use of threads and synchronization primitives. Category 2, which includes assignments 5 to 7 (Section 4.5 to Section 4.7), involves multiprocess programming, handling signals in a process-based environment, and applications of threads in operating systems design. Finally, Category 3 contains only one assignment, the mini-project (Section 4.8), and requires almost all knowledge covered in this course.

4.1 The Warm-Up Assignment

This assignment provides students with a chance to practice the most fundamental skills in multithreaded programming (*e.g.*, thread creation, exit and join). This assignment's problem has to be very simple and interesting, and yet can illustrate some important points (*e.g.*, race conditions). No mutual exclusion is required; but, we ask students to explain why mutual exclusion is not needed and why race conditions do not occur. There are many interesting PRAM (parallel random access machine) algorithms, especially those CRCW (concurrent read concurrent write) ones, that can be readily used for this assignment. The most recent assignment uses $n(n-1)/2$ threads to find the maximum of n distinct integers. Other good candidates for this assignment include matrix multiplication, quicksort, merging two sorted arrays, and parallel merge sort.

The CRCW maximum-finding algorithm is very simple. Let A and V be two global arrays. Array A contains the input and V is a working array. First, set every entry of V to 1 using n threads, one for each entry. Second, for $1 \leq i < j \leq n$, create a thread to compare A_i and A_j. It writes a 0 into V_i (*resp.*, V_j) if $A_i < A_j$ (*resp.*, $A_j < A_i$). Third, use n threads to check the values of V, one for each element, and output the value A_k if V_k is non-zero.

The maximum-finding problem, which uses $n-1$ comparisons in a sequential program, can be solved with $n(n-1)/2$ threads in three steps and each thread uses one comparison. In this way, we introduce our students to the PRAM model which is usually covered in a theory or algorithm course without a programming context. Moreover, we can revisit and contrast the complexity issues of the sequential and multithreaded versions. Most importantly, we ask students to analyze why this solution *does not* have race conditions even though many threads may write simultaneously into the same location of array V. When threads are created, they must receive array indices as parameters (*e.g.*, i and j for the thread that compares A_i and A_j). If a student simply passes the loop indices i and j, it is possible that, before the created thread runs, the indices may have already been advanced to the next

values. Consequently, race conditions occur, and he/she will learn the first lesson of subtle race conditions.

4.2 The Semaphore Assignment

Since semaphores are difficult to be used properly, we normally give students two to three weeks for this assignment. A modified version of the hungry eagles problem was used recently.

> A family of eagles has one mother eagle, n baby eagles, and m feeding pots, where $0 < m < n$. Each baby eagle must eat with a feeding pot and each feeding pot can only serve one baby eagle at any time. The mother eagle eats elsewhere. Each baby eagle repeats the cycle of playing and eating. Before a baby eagle can eat, he must find a feeding pot with food; otherwise, this baby eagle waits. After eating, the feeding pot becomes empty and has to be refilled by the mother eagle. The mother sleeps until a baby eagle wakes her up. Then, she hunts for food, fills all feeding pots, and goes back to sleep. The baby eagle who wants to eat and finds out that all feeding pots are empty wakes up the mother. Only one baby eagle is allowed to do so. More precisely, even though two or more baby eagles may discover all feeding pots being empty, only one of them is allowed to wake up the mother, and all others wait for food.

This problem is very pedagogical for a couple of reasons: **(1)** it involves all commonly seen patterns of using semaphores, **(2)** it can easily cause race conditions, **(3)** it is similar to the reader-priority version of the reader-writer problem but has sufficient differences, and **(4)** it is simple enough if it is approached correctly. Item (3) requires an elaboration. In the solution of the reader-priority reader-writer problem, students learn the way to control who is the first reader in a sequence of reader requests and who is the last reader finishing the access to a shared data. In the hungry eagles problem, students must handle a similar situation in which a baby eagle must know if s/he is the first to discover all the feeding pots being empty. Because only m eagles can eat at the same time, this problem is also similar to the bounded-buffer problem which allows a certain number of threads to access the buffer simultaneously. We always advise our students to study the solutions to classic problems carefully and not to reinvent the wheels, because classic problems are designed to illustrate commonly seen situations in an operating system.

Figure 2 is a possible straightforward solution. Semaphore `Available`, initialized to m, controls the number of baby eagles that can eat at the same time. Semaphores `Mom` and `Food` block the mother eagle and the baby eagle who waked up the mother. The lock `Mutex` is used to protect the shared variable `Emptied` that counts the number of empty feeding pots. The last baby eagle who finishes eating signals semaphore `Available` to permit one more baby eagle to enter. This newcomer will discover all feeding pots being empty, signal `Mom`, and wait for `Food`. The mother eagle waits until it is signaled by a baby eagle. Then, she resets `Emptied` to indicate that all feeding pots are full, releases the baby eagle who sent the signal, and signals the semaphore `Available` $m-1$ times to allow $m-1$ baby eagles to eat. Therefore, semaphores `Mom` and `Food` are used for notifying a thread that some events have occurred, and semaphore `Available` serves as a counting-down counter. Finally, the tests

of `Emptied` being equal to m mimic the entering and exiting protocol in the reader-priority version of the reader-writer problem.

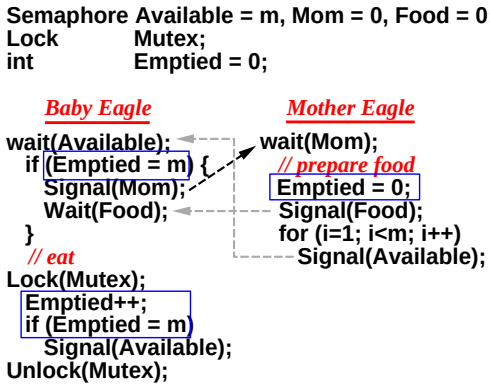

Figure 2: A possible solution to the hungry eagles problem

The solution in Figure 2 is discussed in class. Students are asked to analyze if the three accesses to global variable `Emptied` may cause race conditions. Note that the solution would be easier if we allow the baby eagle who finishes eating the last feeding pot to signal the mother. The current version is designed to make sure that the problem is complex enough for a two-week assignment.

4.3 The Monitor Assignment

Compared with semaphores, the use of monitors is easier. In the discussion of monitors, we remind students that a monitor is simply a mini-OS with the monitor procedures as system calls. We normally ask students to simulate a simple operating system feature with a *single* monitor (*e.g.*, page replacement and disk head management algorithms). We may also ask students to redo the semaphore assignment. Recently, we gave students a problem that allows them to choose a policy, implement it, and discuss how the policy and mechanism can be separated properly and how they can change policy and/or mechanism without affecting their monitors significantly. This concept is, again, pioneered by Brinch Hansen [10] and is commonly referred to as the principle of *separation of mechanism and policy* [43]. The most recent one is the following river crossing problem:

> A particular river crossing is shared by both cannibals and missionaries. A single boat is used to cross the river; but, it only seats three people and must always carry a full load. In order to guarantee the safety of the missionaries, one missionary and two cannibals are not allowed to be on the same boat; otherwise, the latter will gang up and eat the missionary. However, all other combinations are acceptable. Each missionary and cannibal is represented by a thread. When a cannibal (*resp.*, missionary) arrives the river bank, he must register to the monitor with a call to the monitor procedure `CannibalArrive()` (*resp.*, `MissionaryArrive()`). Once a safe boat-load is possible, the involved cannibals and missionaries are released and shipped to the other side. Then, the boat, cannibals and missionaries magically come back for another crossing.

Because there are a number of choices to compose a safe boat-load, we ask students to design their policy (*e.g.*, load as many cannibals as possible so that they will not gang up at the river bank and eat the waiting missionaries), justify the fairness of the chosen policy, and show that their implementation (*i.e.*, the use of condition variables and signal and wait calls) will not change significantly if the chosen policy is altered.

4.4 The Message Passing Assignment

The fourth programming assignment is about message passing. In this assignment, students learn two important topics: asynchronous message passing and the workcrew concept of organizing threads. The following is a general description of the workcrew model.

> Suppose we have one Administrator thread and some WorkCrew threads. The Administrator thread assigns work to WorkCrew threads through channel `Assignment` and collects results from WorkCrew threads through channel `Report`. Each WorkCrew thread retrieves a message from channel `Assignment`, does the required work, and either sends the unfinished work back to channel `Assignment` or sends the result to Administrator through channel `Report`. The system stops when the Administrator thread receives a complete solution to the problem.

The problem for students to solve is to do quicksort using the workcrew model. To this end, each message has a form of $[L, R]$, where L and R ($L \leq R$) give the unsorted segment of a global array. Initially, the Administrator thread sends message $[1, n]$ to channel `Assignment`, meaning that the global array of n elements is to be sorted.[1] Each WorkCrew thread retrieves a message $[L, R]$ from channel `Assignment`, partitions this segment into two $[L, M]$ and $[M, R]$, sends them back to channel `Assignment`, and retrieves the next message. If the retrieved L and R are very close to each other (*e.g.*, $R - L \leq 2$), this WorkCrew thread sorts the array segment and sends message $[L, R]$ to channel `Report` to indicate that the array segment $[L, R]$ is sorted. The Administrator thread simply collects results from channel `Report` until the whole array is sorted. Figure 3 illustrates this relationship.

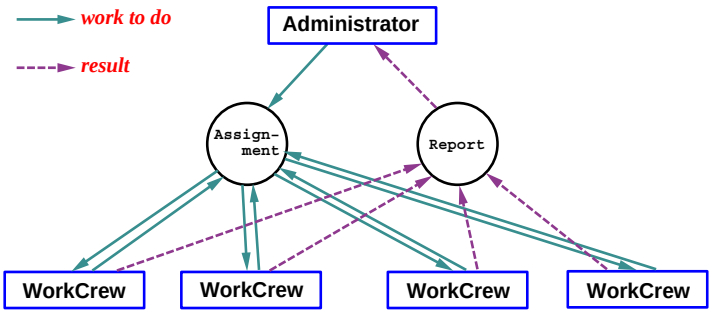

Figure 3: Quicksort based on message passing

[1]Note that in a parallel/distributed environment, this array can be distributed to a number of processors or managed by a central processor.

The easiest way of implementing an asynchronous channel is to use a bounded-buffer. This assignment can also use the message queue feature of Unix/Linux, or sockets if an instructor prefers to do it on a network. A very subtle point should be made clear. The relationship among the number of WorkCrew threads, the number of array elements, and the capacity of channel `Assignment` is important because deadlock may occur. For example, suppose channel `Assignment` is implemented using a bounded-buffer of capacity k and there are more than k WorkCrew threads. It is possible that all WorkCrew threads send their first message (*i.e.*, $[L, M]$) to channel `Assignment` about the same time. Thus, the buffer is full, and none of the threads can send their second messages. As a result, we have a deadlock. We did not mention this problem and wanted to see how many students can discover it. Interesting enough, more than one-third of the students raised their doubt and some of them even provided a complete explanation of why deadlock may occur. Then, we discussed some remedies of this problem.

4.5 The Shared Memory Programming Assignment

By the time we reach this assignment, students have learned about processes and the differences between threads and processes. The assignment requires the use of a number of Unix system calls: `fork()` for creating a child process, `exit()` for terminating a process, `wait()` for process join, and `shmget()`, `shmat()`, `shmdt()` and `shmctl()` for shared memory control. In addition, we also cover the concept of foreground and background processes and commands for managing processes and shared memory.

This assignment uses the matrix multiplication problem. Suppose we have two matrices $A = [a_{ij}]_{l \times m}$ and $B = [b_{ij}]_{m \times n}$ and want to compute their product $C = [c_{ij}]_{l \times n}$, where $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$. To this end, we can create $l \cdot n$ processes of which the (i, j)-th computes c_{ij}. Therefore, a student program simply reads in the input matrices into one or more shared memory segments, creates the necessary processes to compute the entries of matrix C, and prints out the answer when all child processes complete. Note that, similar to the Warm-Up Assignment (Section 4.1), a number of CRCW PRAM algorithms can also be used.

4.6 The Signal Handling Assignment

This assignment provides students with a chance to handle signals. It is a simple one and involves only two signals, `SIGINT` and `SIGALRM`. In fact, we use the Unix real-time alarm clock to simulate timer interrupts, and key press Ctrl-C to generate the `SIGINT` signal for activating a mini-shell serving as a console program.

Students receive a file that contains a number of functions to be run as processes. They are asked to design a program that includes a process scheduler and a mini-shell console. When a `SIGINT` signal occurs, its handler must activate the mini-shell that accepts a number of commands such as killing a process, changing the order of execution, suspending and resuming a process, resetting the clock using Unix system call `alarm()`, and other housekeeping type operations. When a `SIGALRM` signal occurs, which means the real-time alarm clock goes off, the alarm clock signal handler activates the scheduler so that the next process can run. However, the given functions are not run but *called* when it is scheduled to run.

While this is a very interesting problem, many students felt that it is not "realistic" enough because it lacks a touch of actually switching processes. Therefore, we are considering to use the more reliable POSIX type signals to test the following scheme. Each "user process" contains a handler for signal `SIGUSR1` (*resp.*, `SIGUSR2`) interpreted as a suspend (*resp.*, resume) signal. The `SIGUSR1` handler uses `sigsuspend()` to suspend the running process until a `SIGUSR2` comes. Thus, a `SIGUSR2` signal resumes the execution of the suspended process. Note that both handlers are invisible to user programs. A user process must link to a user function to be run as a process. The scheduler can create a "user process" with `vfork()`, and signal `SIGUSR1` to suspend or `SIGUSR2` to resume a user process. Therefore, students can use this mechanism to design a process scheduler with different scheduling polices.

4.7 The Disk Head Scheduling Assignment

This assignment is about the use of synchronous channels. We can implement various components of an operating system simply by treating system calls as message passing activities as shown in Brinch Hansen's RC 4000 system kernel [6, 10]. We choose the elevator algorithm for disk head scheduling; however, virtually all system components (*e.g.*, virtual memory management) can be implemented in this way.

> A system has of a disk, a disk head scheduler, and a synchronous channel (Figure 4). An I/O request is a pair (*ID*,*track*), where *ID* is the identifier of a process that makes an I/O request, and *track* is the track number on which the I/O operation will be performed. A process sends an I/O request to the synchronous channel, and waits until the I/O request is served. The scheduler repeatedly scans the channel (*i.e.*, polling), and receives and saves the messages until the channel has no message. Then, the scheduler starts the disk operation, waits for a while, sends a reply to the corresponding process to indicate the completion of the I/O request, and goes back to check the channel.

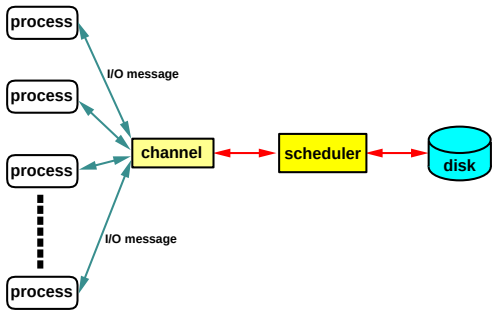

Figure 4: Message passing based disk head scheduling

There are numerous choices for this assignment. The following are some interesting ones:

1. The scheduler may be a page replacement algorithm (*e.g.*, FIFO, LRU and CLOCK). Processes randomly or based on some statistical distribution set the referenced and

changed bits, and generate and send page requests to the channel. The page replacement algorithm selects a victim, modifies the page tables, and sends a message back to the process, indicating the completion of handling a page fault. It may also collect statistics of a particular replacement algorithm. Moreover, the page reference string of each process may also be a fixed one generated before the system is run.

2. In the past years, we also used linear array of threads for this assignment. Good examples include sieve, exchange sort, and prefix computation.

3. We also used more complex schemes such as systolic matrix multiplication, Gaussian elimination, and sorting network.

4.8 The Mini-Project

The mini-project takes the last two weeks. Students are provided with a scaled down version of mtuThread, a user-level kernel that supports non-preemptive multithreaded programming [3]. It uses coroutines implemented with ANSI C functions `setjmp()` and `longjmp()` for thread context switching (Section 3.3). mtuThread supports the following functionality: (1) creating, terminating, joining, suspending and resuming a thread with `THREAD_CREATE()`, `THREAD_EXIT()`, `THREAD_JOIN()`, `THREAD_SUSPEND()` and `THREAD_RESUME()`; (2) yielding the control to other threads with `THREAD_YIELD()`; (3) creating, destroying, locking and unlocking a mutex lock with `MUTEX_INIT()`, `MUTEX_DESTROY()`, `MUTEX_LOCK()` and `MUTEX_UNLOCK()`; (4) creating, destroying, signaling and waiting on a semaphore with `SEMAPHORE_INIT()`, `SEMAPHORE_DESTROY()`, `SEMAPHORE_SIGNAL()` and `SEMAPHORE_WAIT()`; and (5) creating, destroying, sending a message to and receiving a message from an asynchronous channel of capacity one with `MSG_INIT()`, `MSG_DESTROY()`, `MSG_SEND()` and `MSG_RECEIVE()`. Therefore, mtuThread is a miniature of a typical multithreaded system.

Students receive a working version of mtuThread in object format, and a part of the working version in source format with some components replaced with empty templates. This project asks students to set their policy for each of these missing components and implement them accordingly. Possible components for students to modify and extend include (1) adding priority and using a priority type scheduler, (2) implementing semaphores, (3) implementing condition variables, (4) implementing Hoare or Mesa style monitors with semaphores and condition variables, (5) implementing synchronous channels and broadcasting, and (6) implementing thread suspension and resume. We provide a number of example programs for students to test their implementations. Moreover, we also ask students to solve some problems using their systems. These problems usually include various versions of the dining-philosophers problem, the smoker problem, the sleeping barber problem, exchange sort, a user-level mini-scheduler using thread suspend and resume, and some others. We make it very specific that a problem can only use a particular type of synchronization primitive. In this way, students will be able to understand the inner working of a kernel that supports threads and the implementation of synchronization primitives.

5 Evaluation

Materials presented in this paper have been used since the 1998 academic year. However, since we were on a quarter system until 2000, not all seven programming assignments were used. In general, we either used an asynchronous (Section 4.4) or a synchronous (Section 4.7) communication problem, and the Unix process and shared memory component were not used. Consequently, five programming assignments and one mini-project were used. Only during the 2000 academic year everything discussed in this paper was implemented. We tried our best to maintain the same level of difficulty for exam and programming problems so that comparisons would become possible. Unfortunately, many factors may inject unexpected bias into the grading process. For example, the quality of the student body may not be uniform, and graders of programming assignments may use different grading standards, especially in dealing with the issue of leniency. Furthermore, year 2000 is the first year that the semester system was in place and students and faculty were both struggling to make adjustments. Therefore, the data and analysis presented below should be considered as the results of analyzing a dynamic environment rather than a stable one.

The data to be used for this evaluation analysis consist of student scores collected in the above mentioned years (*i.e.*, 1997 to 2000). We choose to use non-parametric methods [26] rather than the popular parametric approach even though the former are less powerful, because score distributions are unlikely to be normally distributed. Table 2 shows all the basic statistics. The averages of the last three years are larger than that of 1997, which means our approach may have a positive effect on student learning. The skew statistics indicate that the score distributions all have its peak (*i.e.*, mode) toward the higher end (*i.e.*, better performance). However, the 1997 data is not as skewed (-0.09) toward the higher end as the last three years are. The positive kurtosis statistics of 1998 and 1999 indicate that the scores are concentrated, while the negative statistics of 1997 and 2000, especially that of 1997, show the scores are flatter. The low variance of 1998 is a surprise, which may indicate that the quality of students are very uniform.

Table 2: Basic Statistics

Statistics	1997	1998	1999	2000
Size	14	31	37	17
Mean	75.64	81.84	79.68	78.47
Median	75.50	84.00	82.00	79.00
Variance	97.79	67.61	118.67	108.89
Skew	-0.09	-0.89	-1.03	-0.26
Kurtosis	-1.08	0.28	0.58	-0.96

Table 3 shows the statistics for testing if the score distribution of each year is normally distributed [15]. From the table, the score distribution of 1997 is highly likely to be normal with a probability of 0.99; the score distribution of 2000 is likely to be normal with

a probability of 0.97; and the score distributions of 1998 and 1999 are unlikely to be normal. Therefore, we cannot rely on the conventional t- and F- statistics and ANOVA type techniques to compare student performance among these four years.

Table 3: Lillifor Test

Year	1997	1998	1999	2000
Statistics	0.07	0.20	0.16	0.12
Probability	0.99	0.16	0.33	0.97

The next step is to decide if the score distributions are all the same. If it is the case, there is no statistically significant improvement in student performance. To this end, we use the Kolmogorov-Smirnov test as shown in Table 4. Each entry of this table has two numbers. The first number gives the Kolmogorov-Smirnov statistic for testing if two distributions are equal, and the second is the corresponding probability. It is clear that the distributions of 1997 and 2000 are quite similar with a probability of 0.887. Since other probability values are very low, we conclude that these four score distributions are statistically different.

Table 4: Kolmogorov-Smirnov Test

1998	0.399		
	0.093		
1999	0.266	0.201	
	0.467	0.501	
2000	0.210	0.298	0.234
	0.887	0.284	0.548
	1997	1998	1999

Next we investigate if the variances are the same even though they look quite different in Table 2. Since we know that these score distributions are not normally distributed, we cannot use the ANOVA type F-tests. Instead, we choose to use the Jackknife test as shown in Table 5. With this test, the null hypothesis H_0 is $\gamma^2 = 1$ and H_1 is $\gamma^2 \neq 1$, where γ^2 is the quotient of dividing the unknown variance of the first population by the unknown variance of the second. The test statistics in Table 5 are asymptotically normal. Surprisingly enough, all test statistics suggest that the null hypothesis should not be rejected (*i.e.*, the variances of the unknown score distributions are statistically equal), although there is mild evidence showing that the variance of 1998 is smaller than those of 2000 and 2001. This is what we learned from Table 2

To the author's best knowledge, there is no more powerful non-parametric method for comparing two means without extra assumptions about the distribution or location parameter (*i.e.*, median). Thus, in order to compare the means we assume the unknown score

Table 5: Jackknife Dispersion Test

1998	0.787		
1999	-0.376	-1.240	
2000	-0.199	-1.045	0.185
	1997	1998	1999

distribution is symmetric about the unknown median. In this way, we can use Fligner-Policello test. The statistics are computed in Table 6 and are asymptotically normal. Since 2.027 is greater than $z_{0.05} = 1.64$, we conclude that the average score of 1998 is greater than that of 1997. Similarly, a mild evidence shows that the average of 1999 may also be greater than that of 1997.

Table 6: Fligner-Policello Test

1998	2.027		
1999	1.448	-0.343	
2000	0.783	-1.097	-0.599
	1997	1998	1999

Given the above evidence, we have the following conclusions. **First**, after the implementation of our approach, the mode shifts from a symmetric normal distribution of 1997 to the right in all subsequent years. This suggests to us that the majority of the students have an improved learning process. The shift of score averages is also supported by the Fligner-Policello test under a mild assumption. **Second**, the variance values are statistically equal. This means that the diversity of student learning capability does not change over years. However, the shift of average and median scores suggests that student performed better under the new approach. However, year 2001 is definitely a special case due to a change to the learning and teaching environment. Numerically, the performance in year 2000 is still better than that of 1997. More evidence is needed to justify this claim statistically. Moreover, we read student submissions and solutions to exam problems, and found out that their improvement is significant. This qualitative finding is not what a statistical analysis can answer. In summary, we believe our approach does improve student learning in a significant way and has helped students write better mutlithreaded programs.

6 Concluding Remarks

The operating systems course presented in this paper has been taught ten times in the past six years by the author. The following is a list of important findings:

1. The fundamental of threads is easy; however, the correct use of semaphores is the most challenging task for many students. This is not because students do not understand the concept of semaphores. Instead, it is because the unstructured nature of semaphores frequently causes unexpected race conditions and deadlocks. See [14] for a detailed account of this difficulty along with a set of examples that demonstrates subtle race conditions. To help students overcome this problem, we take a slow pace and discuss semaphore and its use in detail.

2. The dynamic behavior of threads makes debugging very difficult, because a bug that appears in one run may not occur in the next. Worse, a bug may never occur. To address this problem, under the support of NSF, we are developing and testing a system ThreadMentor. This system consists of a set of class libraries [13] that supports all commonly used thread related features and synchronization primitives, and a visualization subsystem [2] that can display the execution behavior of all involved threads and synchronization primitives. The visualization system can help students pinpoint potential problems visually.

3. Although we did not use a pedagogical operating system such as MINIX or Nachos for programming assignments, our user-level kernel mtuThread that supports non-preemptive multithreaded programming [3] does provide students with a realistic environment for practicing the implementation of various type of thread schedulers, synchronization primitives and other low level components. Other components such as disk head scheduling and demand paging can be done in higher level layers once the dispatcher layer becomes available.

4. This course offers a number of benefits to students. First, it gives them a comprehensive introduction to a new technology that is now very popular in high-end applications. Second, they learn a lot of other interesting material that is normally not available in a typical operating systems course. In particular, students are most interested in channels and its applications in parallel and distributed programming and in system implementation. They also like the assignments that involves the PRAM models. Third, this course prepares the students for subsequent courses such as networking, advanced operating systems, parallel programming, and distributed systems.

The most recent course syllabus, programming assignments, exams and solutions, and web-based course notes are available from

`http://www.csl.mtu.edu/cs4411.ck/www/Home.html`

ThreadMentor and its web-based tutorial are available from

`http://www.cs.mtu.edu/~shene/NSF-3`

Acknowledgments

The author's work was partially supported by the National Science Foundation under grants DUE-9653244, DUE-9752244, DUE-9952509, DUE-9952621 and DUE-0127401, and by a Michigan Research Excellence Fund 1998-1999.

References

[1] Atkin, B. and Sirer, E. G., PortOS: An Educational Operating System for the Post-PC Environment, *ACM Thirty-third Annual SIGCSE Technical Symposium*, Northern Kentucky, February 27 - March 3, 2002, pp. 116–120.

[2] Bedy, M., Carr, S., and Shene, C.-K., A Visualization System for Multithreaded Programming, *ACM Thirty-first Annual SIGCSE Technical Symposium*, Austin, Texas, March 8 – 12, 2000, pp. 1–5.

[3] Bedy, M., Carr, S., Huang, X., and Shene, C.-K., The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming, *29th ASEE/IEEE Frontiers in Education*, November 10-13, San Juan, Puerto Rico, Vol. II (1999), pp. (12b3-1)–(12b3-6).

[4] Ben-Ari, M., *Principles of Concurrent Programming*, Prentice Hall, 1982.

[5] Berk, T. S., A Simple Student Environment for Lightweight Process Concurrent Programming Using SunOS, *ACM Twenty-seven Annual SIGCSE Technical Symposium*, Philadelphia, Pennsylvania, February 15 - 18, 1996, pp. 165–169.

[6] Brinch Hansen, P., The Nucleus of a Multiprogramming System, *Communications of the ACM*, Vol. 13 (1970), No. 4 (April), pp. 238–241, 250.

[7] Brinch Hansen, P., *Operating System Principles*, Prentice Hall, 1973.

[8] Brinch Hansen, P., The Programming Language Concurrent Pascal, *IEEE Transactions on Software Engineering*, Vol. 1 (1975), No. 2 (June), pp. 199–207.

[9] Brinch Hansen, P., Java's Insecure Parallelism, *SIGPLAN Notices*, Vol. 34 (1999), No. 4 (April), pp. 38–45.

[10] Brinch Hansen, P., RC 4000 Software: Multiprogramming System, in *Classic Operating Systems*, edited by Per Brinch Hansen, Springer-Verlag, 2001.

[11] Burns, A. and Davies, G., *Concurrent Programming*, Addison-Wesley, 1993.

[12] Bynum, B. and Camp, T., After You, Alfonse: A Mutual Exclusion Toolkit, *ACM Twenty-seven Annual SIGCSE Technical Symposium*, Philadelphia, Pennsylvania, February 15 - 18, 1996, pp. 170–174.

[13] Carr, S. and Shene, C.-K., A Portable Class Library for Teaching Multithreaded Programming, *ACM 5th ITiCSE 2000 Conference*, University of Helsinki, Helsinki, Finland, July 11–13, 2000, pp. 124–127.

[14] Carr, S., Mayo, J. and Shene, C.-K., Race Conditions: A Case Study, *The Journal of Computing in Small Colleges*, Vol. 17 (2001), No. 1 (October), pp. 88–102.

[15] Conover, W. J., *Practical Nonparametric Statistics*, Third Edition, John Wiley & Sons, 1999..

[16] Dickinson, J., Operating Systems Projects Built on a Simple Hardware Simulator, *ACM Thirty-first Annual SIGCSE Technical Symposium*, Austin, Texas, March 8 – 12, 2000, pp. 320–324.

[17] Donaldson, J. L., An Architecture-Dependent Operating System Project, *ACM Thirty-second Annual SIGCSE Technical Symposium*, Charlotte, North Carolina, February 21 - 25, 2001, pp. 322–326.

[18] Downey, A. B., Teaching Experimental Design in an Operating Systems Course, *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 316–320.

[19] English, J., Multithreading in C++, *ACM SIGPLAN Notices*, Vol. 30 (1995), No. 4, pp. 21–26.

[20] Hartley, S. J., Experience with the Language SR in an Undergraduate Operating Systems Course, *ACM Twenty-third Annual SIGCSE Technical Symposium*, Kansas City, Missouri, March 5 - 6, 1992, pp. 176–180.

[21] Hartley, S. J., "Alfonse, Your Java Is Ready!" *ACM Twenty-ninth Annual SIGCSE Technical Symposium*, Atlanta, Georgia, February 26 - March 1, 1998, pp. 247–251.

[22] Hartley, S. J., "Alfonse, Wait Here for My Signal!" *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 58–62.

[23] Hartley, S. J., "Alfonse, You Have a Message!" *ACM Thirty-first Annual SIGCSE Technical Symposium*, Austin, Texas, March 8 – 12, 2000, pp. 60–64.

[24] Hartley, S. J., "Alfonse, Give Me a Call!" *ACM Thirty-second Annual SIGCSE Technical Symposium*, Charlotte, North Carolina, February 21 - 25, 2001, pp. 229–232.

[25] Holland, D. A., Lim, A. T. and Seltzer, M. I., A New Instructional Operating System, *ACM Thirty-third Annual SIGCSE Technical Symposium*, Northern Kentucky, February 27 - March 3, 2002, pp. 111–115.

[26] Hollander, M. and Wolfe, D. A., *Nonparametric Statistical Methods*, Second Edition, Wiley, 1999.

[27] Holliday, M. A., System Calls and Interrupt Vectors in an Operating Systems Course, *ACM Twenty-eight Annual SIGCSE Technical Symposium*, San Jose, California, February 27 - March 1, 1997, pp. 53–57.

[28] Hughes, L., Teaching Operating Systems Using Turbo C, *ACM Twenty-third Annual SIGCSE Technical Symposium*, Kansas City, Missouri, March 5 - 6, 1992, pp. 181–186.

[29] Hughes, L., An Applied Approach to Teaching the Fundamentals of Operating Systems, *Computer Science Education*, Vol. 10 (2000), No. 1 (April), pp. 1–23.

[30] Khuri, S. and Hsu, H.-C., Visualizing the CPU Scheduler and Page Replacement Algorithms, *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 227–231.

[31] MIX Software, *Using Multi-C: A Portable Multithreaded C Programming Library*, Prentice Hall, 1994.

[32] Moen, S., A Low-Tech Introduction to Operating Systems, *ACM Twenty-sixth Annual SIGCSE Technical Symposium*, Nashville, Tennessee, March 2 - 4, 1995, pp. 149–153.

[33] Morsiani, M. and Davoli, R., Learning Operating Systems Structure and Implementation through the MPS Computer System Simulator, *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 63–67.

[34] Netzer, R. H. B. and Miller, B. P., On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions, *International Conference on Parallel Processing*, August 1990, pp. II93-II97.

[35] Null, L., Integrating Concurrent Programming into an Introductory Operating Systems Class Using BACI, *The Journal of Computing in Small Colleges*, Vol. 14 (1999), No. 3 (March), pp. 288–298.

[36] Robbins, S. and Robbins, K. A., Empirical Exploration in Undergraduate Operating Systems, *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 311–315.

[37] Shene, C.-K., Multithreaded Programming in an Introduction to Operating Systems Course, *ACM Twenty-ninth Annual SIGCSE Technical Symposium*, Atlanta, Georgia, February 26 - March 1, 1998, pp. 242–246.

[38] Shene, C.-K. and Carr, S., The Design of a Multithreaded Programming Course and Its Accompanying Software Tools, *The Journal of Computing in Small Colleges*, Vol. 14 (1998), No. 1 (November), pp. 12–24.

[39] Silberschatz, A. and Galvin, P. B., *Operating System Concepts*, 5th edition, 1997.

[40] Stallings, W., *Operating Systems*, 4th edition, Prentice Hall, 2001.

[41] Tanenbaum, A. S., *Modern Operating Systems*, 2nd edition, Prentice Hall, 2001.

[42] Wagner, T. D. and Ressler, E. K., A Practical Approach to Reinforcing Concepts in Introductory Operating Systems, *ACM Twenty-eight Annual SIGCSE Technical Symposium*, San Jose, California, February 27 - March 1, 1997, pp. 44–47.

[43] Wulf, W. A., Cohen, E. S., Corwin, W. M., Jones, A. K., Levin, R., Pierson, C. and Pollack, J., Hydra: The Kernel of a Multiprocessor Operating System, *Communications of the ACM*, Vol. 17 (1974), No. 6 (June), pp. 337–345.

[44] Ziegler, U., Discovery Learning in Introductory Operating System Courses, *ACM Thirtieth Annual SIGCSE Technical Symposium*, New Orleans, Louisiana, March 1 - 28, 1999, pp. 321–325.