Multithreaded Programming in an Introduction to Operating Systems Course

Ching-Kuang Shene*
Department of Computer Science,
Michigan Technological University,
Houghton, MI 49931–1295
Email: `shene@mtu.edu`

Abstract

This paper presents a way of teaching multithreaded programming as a component in an introduction to operating systems course. Topics include programming assignments, term projects, and experiences. This paper also suggests future work for overcoming a bottleneck that occurs in the current version of this course.

1 Introduction

To help our students early in approaching system oriented courses and research projects, a system programming course was revised to become an introduction to operating systems course. Since there is another elective operating systems course for senior and graduate students, this course only provides a survey of important concepts and related programming skills. Students enrolled in this course normally have completed the CS1, CS2, computer organization and assembly language, and project oriented software development courses, and are sophomores and juniors. As a result, this course must be elementary and informative. Tanenbaum's *Modern Operating Systems* was selected as our text. In a 10-week quarter system, only the most important topics of the first five chapters were covered, including a brief discussion of deadlocks.

With the continuing emergence of multithreaded computation as a powerful vehicle for science and engineering, we decided to take a multithreaded programming early approach so that this skill can be used in later courses such as GUI programming, parallel programming, operating systems, computer graphics, and some other courses in which concurrency is an important element. To this end, we have to choose a programming system. We used Pascal-FC [2] previously, but was not well-received. We also seriously considered SR [6]. Since its syntax and semantics are different from that of C/C++ and since we do not have enough time to cover another programming language, SR was rejected. Eventually, we chose the SunOS lightweight process library, because it is a good user-level thread library with a simple API. To further reduce students' load, we also adopted Berk's simplified `ST_threads` [1].

The following summarizes our effort of using multithreaded programming for students' lab work. Section 2 details the programming assignments, Section 3 discusses a term project, and Section 4 describes some of our findings. Section 5 suggests some future work to improve this course. Section 6 has our conclusion.

2 Programming Assignments

After learning the process model, context switching and process scheduling and before actually designing and implementing their own systems, students must learn and appreciate the merit of multithreaded programming through several programming assignments. The purpose of these assignments is mainly to motivate students for this new programming paradigm.

There are four assignments and one term project. Each assignment takes about one week to 10 days, while the project requires approximately three weeks. The first assignment serves as a warm-up (Section 2.1). The second and the third give students chances to practice semaphores and condition variables (Section 2.2 and Section 2.3). The fourth provides an opportu-

*This work was partially supported by the National Science Foundation under grant CCR-9696084 (formerly CCR-9410707) and grant DUE-9653244.

nity for students to learn catching and handling signals (Section 2.4). The term project involves implementing a non-preemptive user-level multithreaded system that supports locks, semaphores and mailboxes (Section 3).

2.1 Warm-Up

This warm-up assignment focuses only on forking and joining threads, and resource sharing among threads. The problem was matrix multiplication.

Two matrices $A = [a_{ik}]$ and $B = [b_{kj}]$ of orders $m \times p$ and $p \times n$, respectively, are read into two global two-dimensional arrays. For each entry of matrix $C = [c_{ij}] = A \cdot B$, a thread is created for computing $c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$, where $1 \leq i \leq m$ and $1 \leq j \leq n$. Therefore, there are $m \times n$ threads running in parallel. After creating all threads, the main function uses thread join to wait until all threads are done and then displays the result of the multiplication.

Matrix multiplication is not the only choice. Problems that require the use of shared memory without synchronization except for joining threads can be very good candidates (*e.g.*, quicksort).

2.2 Semaphores

Since this is an introductory course, discussing some typical uses of semaphores first for students to incorporate into their programs could be better than only presenting classical problems. In the past, we discussed the following ways of using semaphores: locks, counters, notifications, and rendezvous. Semaphores used in this assignment are all counting semaphores implemented with condition variables.

This assignment asks students to implement a special type of rendezvous. There are two groups of threads, say A and B. To establish a rendezvous, it takes one thread from A and two threads from B. A correct solution looks like the following:

```
Thread in A          Thread in B

wait(A);             signal(B);
   wait(B);          wait(done);
   wait(B);
signal(A);
signal(done);
signal(done);
```

where `B` and `done` are initialized to `0` and `A` is initialized to `1`. Semaphore `A` makes a thread the only one waiting for a rendezvous; semaphore `B` is used to wait for a signal from a thread in B; and semaphore `done` indicates a rendezvous has occurred. Note that removing the `wait(A)-signal(A)` pair can cause deadlocks.

Students were also asked to consider a special rendezvous with message exchange. More precisely, when two threads, one from each group, come to a rendezvous point, they should exchange their IDs.

About 19% of students submitted correct solutions, while about 15% of all solutions were almost correct except that deadlocks may occur because `wait(A)-signal(A)` was missing. Other solutions were not complete because of race conditions, incorrectly formed critical sections, using unnecessary counters without mutual exclusion, and other mistakes.

2.3 Condition Variables

This assignment involves the use of condition variables and monitors. In class, the Mesa style and Hoare style of signaling a condition variable were covered. The implementation of semaphores used in the previous assignment was described to students, illustrating the differences between the Mesa style and Hoare style. Then, students were asked to write two programs using monitors. The first is the well-known readers-and-writers problem and the second is the river-crossing problem.

In the river cross-crossing problem, vehicles arrive at both ends of a bridge that can only hold three vehicles along the *same* direction at any time. Students were asked to write a monitor with entry procedures `Arrival()` and `Departure()`. `Arrival()` passes the direction of the calling vehicle to the monitor. Returning from this procedure is postponed until the situation is safe (*i.e.*, there are less than three vehicles on the bridge along the same direction). `Departure()` is called after a vehicle completes its river-crossing.

Since monitors are more structured than semaphores, more than 46% of our students submitted correct solutions. Since counters can be used in a monitor without worrying about race conditions, some students can even rewrite the second assignment using a monitor in a very short amount of time.

2.4 Handling Signals

This assignment is about catching signals `SIGINT` and `SIGALRM` by installing signal handlers and using functions `setjmp()` and `longjmp()`. This program is actually a naive scheduler in which a user function is "called" rather than resumed from where it was suspended.

Five simple functions, serving as "user programs", were provided to students to be included into their programs. The alarm clock function `alarm()` is used to set time quantum values. Once signal `SIGALRM` occurs, this program should catch and process it, and call next function. If `SIGINT` is caught due to a `Ctrl-C`, a mini-shell is activated to accept user commands such as exiting the program, exiting the mini-shell, killing a function, suspending a function, reactivating a suspended functions, and setting a new time quantum value.

The capability of jumping between a signal handler and a function gives students a sense of interrupt handlers. More importantly, one can use `setjmp()` and `longjmp()` to build coroutines, which will be used in the term project for implementing a scheduler. Without using any multithreaded library, we were able to demonstrate to students a general principle of creating a set of coroutines, which is similar to the use of function `Yield()` available in many multithreaded systems.

This is an easy problem and about 79% of students came up correct solutions.

3 Term Project

The last three weeks were reserved for a term project which involved implementing a non-preemptive user-level multithreaded system with synchronization mechanisms locks, semaphores and mailboxes. More than 17% of our students submitted correct solutions; only 10% received a score less than 75%.

3.1 Project Materials

Materials provided to students included the following: (1) a handout describing the project, (2) a copy of a correct system in object format as a reference, (3) a copy of source code of the system with all required elements removed, and (4) a set of examples illustrating the use of this system. The required elements of this project consist of implementing a non-preemptive scheduler; a queue class and its accompanying operations; a lock class with member functions `lock()` and `unlock()`; a semaphore class with member functions `wait()` and `signal()`; a mailbox class with member functions `send()` and `receive()`; and, a mechanism for managing stack space. In fact, the lock class was given to students as an implementation example.

3.2 Stack Space Management

Since an executable has only one stack and since each thread requires a separate stack for its local environment, a stack area must be found before a thread can be started. We used a scheme proposed by Kofoed [7]. A large enough block is reserved for `main()` before starting any thread. For each thread, the remaining stack space is searched for a large enough block with the first-fit algorithm. The allocated and freed blocks are chained together in address order and a freed area occupied by a completed thread is merged with its adjacent holes.

This is a challenging exercise and requires the use of `setjmp()` and `longjmp()` to jump back-and-forth between the stack management and thread management functions. Therefore, part of this stack allocation function was provided to students with the first-fit and merging adjacent holes algorithms removed. Since this term project started after memory management was covered, asking students to implement the first-fit and merging adjacent holes algorithms seemed reasonable.

3.3 The Non-preemptive Scheduler

Implementing the scheduler and various queue activities requires the use of functions `setjmp()` and `longjmp()` to mimic context switching.

The system has one ready queue and each of the synchronization mechanisms has its own waiting queue. A thread is appended to one of these queues only if it must wait according to the corresponding synchronization protocol. If a thread must be removed from a queue, the first of the queue is taken and appended to the ready queue. All threads have equal priority and all queues are FIFO. When a thread is suspended, `setjmp()` is used to save its environment into a jump buffer in that thread's control block. Moving a thread from ready to running is simply implemented by executing a `longjmp()` to its saved jump buffer.

3.4 Example Programs

Several example programs were provided to students to illustrate the use of this system. Students can run these programs with the correct implementation to get a feeling of what a correct system should look like. These programs include: (1) a program illustrating the use of semaphores to enforce an alternating execution of two threads, (2) a solution to the bounded buffer problem using mailboxes, (3) various solutions to the dining-philosophers problem using semaphores, and (4) a parallel sorting program using mailboxes.

The last problem requires some elaboration. Thread T_i sends integer messages to thread T_{i+1} with mailbox B_{i+1} ($1 \leq i < n$). A generator thread sends n integers to mailbox B_1 for thread T_1 to retrieve. Thread T_i keeps the first received number, say K_i, and waits for other integers from B_i. If the incoming one is larger than K_i, it is sent to T_{i+1}; otherwise, T_i sends out K_i and keeps the incoming one. After all integers have been sent, the generator sends the `END` message. After receiving the `END` message, K_1, K_2, …, K_n are in sorted order.

3.5 Programming Problems

Using their implementations, students are required to solve two problems: (1) the smokers problem and (2) the parallel exchange sort. The smokers problem is well-known and can be found in most operating systems textbooks. However, we only asked students to solve a restricted version in which each smoker is required to take *both* ingredients provided by the agent from the counter at the same time.

The parallel exchange sort must be solved with mailboxes. Let there be $2n$ input integers given in a shared array, say $a[\ ]$. Thread T_i ($0 \leq i < n$) retrieves $a[2i]$ and $a[2i+1]$, and iterates n times. For each iteration, T_i performs the following five steps once: (1) compares the numbers in hand, (2) sends the smaller one to T_{i-1} and the larger one to T_{i+1}, (3) waits for a number p_{i-1} from T_{i-1} and a number p_{i+1} from T_{i+1}, (4) if p_{i-1} is larger than T_i's smaller number, then uses p_{i-1} to replace the smaller number, and (5) if p_{i+1} is smaller than T_i's larger number, then uses p_{i+1} to replace its larger number. After n iterations, T_i puts the smaller and the larger numbers back to $a[2i]$ and $a[2i+1]$, respectively.

There are other good and interesting problems using mailboxes such as parallel sieve, n-queens, and fire squadrons. They may be used in the future.

3.6 Possible Improvements

This term project is not very realistic, because the scheduler is non-preemptive and is not self-scheduling (*i.e.*, the system requires a driver statement that repeatedly activates the next thread in the ready queue [7]). However, it is platform independent since it is completely written in C/C++ without any UNIX system calls. Moreover, it is simple and deterministic and students can easily trace their programs with a debugger. There are four possible improvements without adding too much burden to students.

First, making the scheduler priority driven is easy; however, starvation may become a major problem. On the other hand, students could add aging to the scheduler and practice other scheduling polices. **Second**, a more flexible stack allocation scheme is possible. All thread control blocks and stack areas can be dynamically allocated; but, some assembly language is required making the system machine dependent. This does not complicate the system very much as has been shown in REX [4] and several public domain implementations of Pthreads. REX is particularly interesting, because it is small and clean and may serve as a model if this change must be made. For a MS-DOS environment, English [5] has a simple and interesting system, with source code available. **Third**, by modifying the thread initialization part and the scheduler, this system is capable of self-scheduling and is more natural. **Fourth**, with the help of semaphores, it is possible to implement condition variables and a `MONITOR` base class for users to derive their own monitor classes. Moreover, students can try and compare the effects of Mesa style and Hoare style.

Converting the non-preemptive scheduler to a preemptive one is difficult, since context switching and signal handlers become too involved to be done in three weeks. Hence, we will keep using a non-preemptive scheduler in the near future.

4 Experiences

In our two-year experience, the most important finding is that many students just directly apply sequential programming skills to multithreaded programming. Since the behavior of a multithreaded program is dynamic and the same bug may not appear every time the program is run, students did have a hard time in learning multithreaded programming, especially those who were used to sitting in front of the computers immediately after receiving the assignment and employing the trial-and-error approach. The resulting programs were bulky and more complicated than necessary. Worse, this approach can easily introduce race conditions and deadlocks into programs. We encouraged our students to design and think the program flow and interaction among threads carefully before start coding programs. In addition to this, we also found several other problems as follows.

Student programs frequently have race conditions. It is difficult to convince them that the existence of a particular race condition may cause serious problems, since in their mind they have "tested" their programs and received correct answers. It is also difficult to pinpoint race conditions, since in many cases sharing a variable does not cause any problem. Therefore, it is believed that there remained unspotted race conditions in student programs. This definitely would have a bad impact, since students could get the impression that their programs are "correct."

To avoid race conditions, some students created large critical sections, serializing their programs. A typical example is that almost all important statements of a thread are enclosed in a large critical section, thereby forcing the threads to execute one after the other.

While the above two are beginner problems, using explicit counters rather than the built-in one of a counting semaphore could be universal. Many students in this course and some in an elective operating systems course for senior and graduate students had the same tendency. For example, in the rendezvous assignment discussed in Section 2.2, many students used a counter to count the number of threads in B arrived for a rendezvous rather than using two `wait()`s. This not only increases the complexity of the program, but also makes it inefficient. The more explicit counters are used, the more semaphores are required to establish mutual exclusion. Consequently, a program can be very inefficient due to frequently locking and unlocking counters.

Deadlocks did not occur in students' programs frequently. This is perhaps because obvious deadlocks can easily be detected and subtle deadlocks are rare in these simple programs and are difficult to uncover. In the semaphore assignment, deadlocks did appear in several programs as pointed out earlier.

5 Future Work

As pointed out in Section 4, detecting potential race conditions and deadlocks is difficult and there are no good public pedagogical tools dedicated to this purpose.

To help our students learn multithreaded programming effectively, we intend to design some pedagogical tools. This is not an easy job, since statically detecting race conditions and deadlocks are NP-complete and NP-hard, respectively [9, 8]. On-the-fly detection of deadlocks is a textbook topic; but, on-the-fly detection of race conditions is infeasible. Postmortem detection is a possibility; but, it could be too late because the user program has already been involved in some problems.

Animating the execution of a multithreaded program is very helpful. There has been some progress in the past few years [3, 10, 11]. Animation can be real-time or postmortem. The main drawback of a postmortem system is that it generates large amount of output, usually several megabytes. Moreover, since the animation system also needs some extra synchronization mechanisms (*e.g.*, locking the output file and/or other resources that are shared by the system and threads), a user program will compete with the system to gain access to these shared resources and therefore may be interfered by this extra synchronization. As a result, the behavior of the animated user program could be considerably different from that of the original.

In summary, pedagogical tools that can detect deadlocks on-the-fly, and potential race conditions and deadlocks statically are powerful aids for students. Animating the behavior of a program along with the activities of various synchronization mechanisms would also help to reducing the bottleneck of shifting from sequential programming to multithreaded programming.

6 Conclusion

In this paper, we have presented a possible way of introducing multithreaded programming in an introduction to operating systems course. We have described four programming assignments and a term project. We also discussed some findings of teaching multithreaded programming and the need of pedagogical tools. In general, this was a successful course which can be improved. We believe that combining a good textbook, reasonable and representative assignments and term projects, and pedagogical tools will improve the teaching of multithreaded programming. Based on the above discussion, we also believe that we have had a good start and are in the right direction.

References

[1] Toby S. Berk, A Simple Student Environment for Lightweight Process Concurrent Programming under SunOS, *ACM Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, February 15–18, 1996, pp. 165–169.

[2] Alan Burns and Geoff Davies, *Concurrent Programming*, Addison-Wesley, 1993.

[3] Wentong Cai, Wendy J. Milne and Stephen J. Turner, Graphical Views of the Behavior of Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 223–230.

[4] Stephen Crane, The REX Lightweight Process Library, March 7, 1996. Available by anonymous ftp from `dse.doc.ic.ac.uk` in directory `/pub/rex`.

[5] John English, Multithreading in C++, *ACM SIGPLAN Notices*, Vol. 30 (1995), No. 4, pp. 21–28.

[6] Stephen J. Hartley, *Operating Systems Programming*, Oxford University Press, 1995.

[7] Stig Kofoed, Portable Multitasking in C++, *Dr. Dobb's Journal*, No. 226 (Nov), 1995, pp. 70-78.

[8] Stephen P. Masticola, Static Infinite Wait Anomaly Detection in Polynomial Time, LCSR-TR-114, Laboratory for Computer Science Research, Rutgers University, 1990.

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

[10] John T. Stasko, The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report, Technical Report GIT-GVU-95-03, College of Computing, George Institute of Technology, 1995.

[11] Qiang A. Zhao and John T. Stasko, Visualizing the Execution of Threads-based Parallel Programs, Technical Report GIT-GVU-95-01, College of Computing, Georgia Institute of Technology, January 1995.