THE DESIGN OF A MULTITHREADED PROGRAMMING COURSE AND ITS ACCOMPANYING SOFTWARE TOOLS

Ching-Kuang Shene and Steve Carr
Department of Computer Science
Michigan Technological University
Houghton, MI 49931-1295
(*shene*|*carr*)*@mtu.edu*

ABSTRACT

With the continuing emergence of multithreaded computation as a powerful vehicle for science and engineering, the need for an introduction to multithreaded programming for scientists and engineers is high. All popular operating systems already support multithreaded programming and the popular POSIX Pthreads standard has been approved. It is the right time to teach students this new technology. This paper presents the problems and difficulties we encountered and a set of comprehensive and flexible course materials for a multithreaded programming course for sophomore and junior students. This paper also presents the design of pedagogical tools for the students to visualize and experiment with various concepts in multithreaded programming. These concepts include program behavior and execution visualization, deadlock and race condition detection, and software metrics for measuring the complexity of students' programs.

1. INTRODUCTION

John Hopcroft challenged the computer science community with the following [16]:

> The major transfer of information from universities to industry does not occur through journal articles and publications; rather it comes about through students who get degrees and then take jobs in industry. Given that students beginning their studies today will not get degrees for another four years and will not be in a position to affect the ways in which things are done in industry for possibly another 10-15 years, the time delay is on the order of 20 years. We must therefore look ahead as far as possible and try to educate students now for the rapidly changing world of 20 years hence.

Although John Hopcroft was talking about technology transfer of robotics research, the same holds true for any technology, including multithreaded programming. But, the most important point is that we must teach our students to use future technologies so that technology transfer can happen smoothly in a shorter period of time. Multithreaded programming has been around for more than 20 years. Some early systems (*e.g.,* IBM OS/MVT) and languages (*e.g.*, IBM PL/I F and Modula

2) supported multithreaded programming under different names. For example, IBM called a thread a task while Module 2 called it a coroutine. Recently, almost all operating systems, from Windows 95/NT to Unix, support multithreaded programming. Therefore, to make sure our students could lead the trend of computer science in the foreseeable future, we need to introduce them with this important skill and this is the main thrust of our project.

While teaching parallel computing can be very difficult in many institutions, multithreaded (MT) programming (MTP) is a highly accessible parallel programming technique that can improve the performance of sequential programs. The main benefits include: (1) performance gain from multiprocessing hardware, (2) increased application responsiveness and throughput, (3) enhanced process-to-process communications, (4) efficient use of system resources, (5) effective exploitation of the inherent threadedness of distributed objects, (6) increased reusability due to the fact that a MT binary can be executed on both uniprocessors and multiprocessors without modification, and (7) increased portability since a single source can be used on multiple platforms under identical standards (*e.g.*, POSIX).

Given that most popular operating systems have MT capability built-in, now is the right time for us to teach students this new technology. In so doing, students will be able to carry the skills over to their jobs. Unfortunately, MTP is not part of the *ACM Computing Curriculum 1991* and is frequently not taught in many institutions. Compared with C, C++ and Java, there is not much teaching support available for MTP.

To fill this gap, we are developing a comprehensive and flexible MT oriented course so that it can either be taught across many courses (since MT is a generic technology whose impact is not restricted to a single course) or be taught in a dedicated one. Since teaching MTP is more difficult than anticipated, we are also developing pedagogical tools that can help students to make a smooth paradigm shift, to understand synchronization mechanisms more fully, to catch a significantly higher percentage of bugs related to timing in MTP, and to reduce the complexity of their programs.

The remaining of this paper details our study of the design of a MTP course and its accompanying software tools. Section 2 surveys some problems in teaching MTP; Section 3 summarizes existing works; Section 4 presents the design merit and contents of our course and pedagogical tools; and finally Section 5 contains our conclusion.

2. SOME PROBLEMS IN TEACHING MTP

2.1 Where does MTP Fit into the Curriculum?

The most common place for teaching MTP is in an operating systems course in which approximately three to four weeks are allocated to these topics. This is inadequate since motivating multiprocess/multithread concepts and teaching synchronization mechanisms usually consume most, if not all, of these three to four weeks. Students are usually excited about the new topic in the beginning, but quickly become overwhelmed with the struggles of understanding threads, the syntax and semantics for creating/joining threads, race conditions, critical sections, semaphores, etc. Since

there is usually only very limited time for students to solve one or two simple MTP problems, they, in general, do not appreciate the merit of MTP. Furthermore, MTP requires a very difficult mindset to write correct programs. Some students give up due to frustration, making our effort worthless.

In covering MT concepts as part of an operating systems course, we have tried to teach too much material in a short period of time. A major problem is a sudden programming paradigm shift that requires a significant effort to comprehend. A few weeks cannot help students in coping with and appreciating the beauty and power of MTP. Unfortunately, even though multithreaded programming is so important, most universities find it virtually impossible to have a dedicated course due to resource constraints. Therefore, spreading the MTP concepts into several relevant courses in a careful, well-thoughout, well-planned and organized way may be a viable approach.

2.2 Where are Good Textbooks and Pedagogical Tools?

In the past year, there have been several MTP books published. However, these books do not make teaching MTP any better or easier than two or three years ago. They are either dedicated to a particular platform, or they are reference manuals in which the syntax of function calls are emphasized. Section 3.1 has the details. Worse yet, most examples in these books are too superficial or are simply rewritten versions of classical problems found in other textbooks. Some of these textbooks spend significant effort on the syntax of calling MT functions. As a result, most of these books are inappropriate for classroom use. Not only is the writing style improper, but also many fundamental concepts such as the trade-off between large critical sections with a few locks and small critical sections with many locks, are not mentioned or only touched upon superficially.

Software support for teaching MTP is virtually non-existent. During the past several years, only a few of the MTP books that have been published include software systems [2,6]. In these texts Ada, extended Pascal or SR is used to convey MTP. While these are good programming languages, it is unreasonable to use two different languages in one class, one for MT programming and another for other programming purposes. Java may be a good alternative; however, it suffers from the same problem as that of Pascal-FC and SR; operating systems are not written in these languages.

2.3 Difficulties in Teaching MT Programming

The authors have taught multiprocess, multithreaded and parallel programming for several years. Details of teaching MTP in an introduction to operating systems course and our findings can be found in Shene [30]. This is a 3-credit course with two tracks. The *lecture* track covers traditional materials of an operating systems course, while the *programming* track emphasizes multithreaded programming. Therefore, students will learn the MTP concepts and skills in a ten-week quarter by writing five MT programs and completing a user-level MT kernel. The following are some other issues we found important.

2.3.1 MTP Requires a New Mindset

Since writing MT programs requires a different mindset, especially in debugging, it is rather difficult for students to apply what they have learned and used for years. Students struggle with the

concept that they are writing multiple programs executed simultaneously. This problem is analogous to shifting from top-down structured design to object-oriented design and to shifting from procedural programming to logic or functional programming.

2.3.2 The Behavior of a MT Program is Dynamic

The behavior of a MT program is dynamic, depending on external factors such as CPU speed, system load, process/thread mix, etc. Therefore, debugging a MT program is considerably more difficult than debugging a sequential one. Adding time as a variable in program correctness leads to bugs that may not appear at the same place in every program execution. Moreover, some bugs may never appear at all. This is one of the most significant aspects of MTP that results in stumbling blocks in student comprehension of the material. If the material is pushed too fast, students lose confident quickly.

2.3.3 Proper Synchronization Is More Difficult Than Anticipated

Based on our experience in teaching operating systems, and parallel and MT programming, we have found that the concept of race conditions and use of critical sections, and mutual exclusion are far more difficult than anticipated. Without a deeper understanding of these issues, students will not be able to write decent MT programs and appreciate the power of multithreading. Students usually encounter the following problems.

First, it is very difficult to convince students that their programs contain race conditions, because these race conditions may never occur in their tests. Students write MT programs with a sequential programming mindset. As a result, they test their programs in their usual manner, never accounting for the effects of time and parallelism on their programs.

Second, deadlocks are difficult to detect by beginning students. Moreover, by only studying classical problems and examples in the textbooks, which are always correct, they do not learn enough to synchronize more complex threads correctly. A visual aid is necessary.

Third, student programs tend to have a few very large critical sections handled with a few locks and/or semaphores. The effect is to serialize their programs. It is difficult to convince students that their programs are not good since their programs work. Effective use of parallelism is not a concept that they grasp with the usual amount of time spent on MTP.

Fourth, it is common for new MT programmers to use very complex synchronization logic. For example, two semaphores would be sufficient to implement the rendezvous of two threads. But, many students use counters instead of the built-in counting mechanism of counting semaphores. As a result, complex critical sections are required to update these counters. Reasonable software metrics are required to measure the complexity of synchronization mechanisms in students' programs.

Fifth, most students indicate that they need additional and real examples in addition to the classical ones. Unfortunately, most textbooks and reference materials repeat the same set of classical problems. Thus, building a repository of problems, academic and practical, will benefit both

instructors and students.

3. PREVIOUS WORK

3.1 Textbooks and Course Materials

Most operating systems textbooks have chapters on synchronization mechanisms, which can be considered as part of MTP. Other books can be classified into three categories: (1) books using a specially designed concurrent programming language, (2) books discussing a particular system, and (3) books covering general topics and more than one platform.

The first category includes language books such as Ada [11], extended Pascal [2,6], Java [14, 18, 27] and SR [1]. Hartley also wrote a book on SR [13]; however, this is a problem solving type book. These books focus on a single language, and can be difficult to use as textbooks in a MTP course since the languages, except for Java, were designed prior to the advent of MTP. Moreover, students must learn a language that is quite different from the language they use to implement their operating system project (*e.g.*, C or C++).

The second category has those books written for a particular platform such as Windows 95 and Windows NT [4], Unix [29] and POSIX [7, 20, 26]. Except for the POSIX Pthreads standard, most systems are platform dependent. Therefore, these books are not good candidates, when a particular platform is not available.

The third category contains some interesting books, focusing on one or two popular standards with in-depth discussion [19, 17]. The former is an introduction focusing on general issues, while the latter contains an in-depth discussion of the multithread models of Sun Solaris and Pthreads. Both are good reference books.

3.2 Software Tools

In addition to major systems' multithreading support, commercial systems have started to appear recently (*e.g.*, `Threads.h++` and `Thread<ToolKit>`). These are professional development systems and even after an educational discount, they are still too expensive for classroom use. Moreover, for educational purpose, only a fraction of these systems is used.

There are free MT systems. Most of them are implementations of the POSIX Pthreads standard. Some of them are available on most UNIX platforms, while the others can only be used on recent Linux kernels (version 2.0 and up). For those places where Linux is the main platform, these Pthreads implementations are good choices. The problem, however, is that they are not very stable and like the commercial systems lack educational aids.

There are publications addressing the need of concurrent programming in undergraduate education. Berk [3] simplified SunOS lightweight process library calls, Higginbotham and Morelli [15] used the UNIX IPC, and Bynum and Camp [8] developed a language similar to Pascal-FC. The

impact of these works on a comprehensive MTP course is minimal, since their focus is narrow. While these systems implement various MT capabilities, none of them is capable of providing programming aids to students to ease learning and help visualization of the activities of a MT program. A few works do exist to make concurrent programming easier with program animation. Zimmermann et. al. [36] discussed their system for Portal which uses a special hardware, making it unportable. Hartley's approach [12,13] used a software technique. The animated program dumps its activities to a file and then uses XTANGO to playback *after* the animated program completes. Price and Baeker [28] also discussed a framework for concurrent programming animation.

3.3 Other Research Efforts

Parallel program execution visualization, debugging and performance-tuning, and race condition and deadlock detection have been studied extensively. Some of these results can be readily used for educational purpose. In the following, we shall focus on program visualization, race condition (or data race) detection, deadlock detection, and complexity measures.

3.3.1 Program Visualization

Visualizing parallel programs can be *real-time* or *post-mortem*. The former generates visuals on-the-fly, while the latter saves the events and plays back with another system. There have been many published works on these topics. Zhao and Stasko [35] used POLKA for visualizing Pthreads programs, while Cai [9] discussed a system for visualizing OCCAM programs. PARADE [31], which is based on POLKA, is a post-mortem system. The advantage of a post-mortem system is that everything relevant to the execution of a program has been saved and can be replayed at any time. However, its major disadvantages are (1) it could be too late for a programmer to catch other bugs, since it only shows one instance of the program execution, (2) a large volume of output will be generated which could be incomplete or even corrupted if the program ends abnormally, and (3) the system must also synchronize its own file writing activities, adding an extra layer of complexity that may affect the program's original behavior.

3.3.2 Race Condition Detection

Detecting race conditions is a very difficult problem. Exactly detecting races in programs that use multiple semaphores is NP-complete [25]. If the synchronization mechanism is weaker than semaphores, an exact and efficient algorithm can be found [24]; otherwise, only heuristic algorithms are known. Along this line, Lu, Klein and Netzer [21] showed that for a single semaphore, detecting exact race condition is of order $O(n^{1.5}p)$, where n is the number of semaphore operations and p is the number of processes in the execution.

3.3.3 Static and Dynamic Deadlock Detection

Deadlocks detection can be static or on-the-fly. The former can help the students to identify potential deadlocks before running their programs; however, it is inaccurate [10, 22]. On the other hand, one can monitor resource allocation so that deadlocks can be reported on-the-fly. But, by that time, it is already too late. Note that detecting deadlock cycles statically is NP-hard [23]. A static

algorithm can be very helpful, since students could receive warnings before running their programs.

3.3.4 The Complexity of a Multithreaded Program

The measure of the complexity of a MT program has not received much attention in either parallel/concurrent programming or software engineering research. Many of these works target only Ada programs [32,33,34]. Software metrics have been used in entry-level programming courses. A similar measure could also be very helpful to the students who are learning MT programming for identifying the complexity of the synchronization mechanisms.

4. THE DESIGN MERIT OF A COURSE AND LAB TOOLS

4.1 Design Merit

Unlike writing a sequential program, which has only one thread of execution, designing and writing MT programs is very challenging. With properly designed programming aids, students can indeed learn the MT concepts and skills early, practice it in later courses, and carry what they have learned to their jobs. Therefore, the design merits of our course are: (1) it should be an intermediate course for sophomores and juniors, (2) it should address most fundamentals of MT programming in a learning-by-doing way rather than an in-depth theoretical treatment, (3) it should serve as a foundation and prepare the students for other courses such as operating systems, parallel programming, user interface programming, computer graphics and so on, (4) it should be reasonably modulized so that other instructors can pick a few components for using in their classes, (5) it should provide visual aids for the students to "see" the effects and behavior of a MT program, and (6) it should provide pedagogical aids to the students for pinpointing potential problems, easing their debugging effort, and reducing the complexity of their programs.

Based on the above rationale, this course addresses the fundamentals of MTP with the help of a group of pedagogical and visual programming aids, and leaves the more complex theoretical discussion to respective courses. Emphasizing the programming fundamentals does have its merit. Most students will take jobs in the industry as programmers using some MT capable operating systems. Equipping them with MT skills will certainly enhance their career opportunity and survivability. Knowing and practicing the programming fundamentals through visualizing the program behavior and learning-by-doing will lessen students' fear of in-depth theoretical development. Process formalism, correctness proofs of synchronization mechanisms, and other topics in later courses will be less formidable, since students will already be familiar with MT technology. Consequently, paradigm shifting will be easier.

4.2 Course Contents

This is a 3-credit introduction to MTP course for sophomores and juniors with prerequisites C/C++, data structures and computer organizations. The course contents are subdivided into 12 modules to be used in a 10-week quarter. It is believed that the whole sequence should be adequate for a 15-week semester. The laboratory part involves the use of the software tools designed in this

project. The first six modules have been successfully used in an introduction to operating systems course three times [30].

Module 1 - Warm UP: This module introduces the fundamental concepts of threads, including the meaning of threads, creating, destroying and joining threads, and shared memory programming. Real examples that employ the MT technology such as Netscape and Java are mentioned.

Module 2 - Fundamentals: Typical models of MT programs (*i.e.*, client/server, pipeline, and peer) are introduced. We emphasize the dynamic behavior of MT programs and indicate when to and when not to use MTP. This is followed by simple MT programs such as an ATM server and matrix multiplication.

Module 3 - Synchronization with Locks: Race conditions, critical sections, locks and deadlocks are covered, followed by examples (*e.g.*, updating a shared counter and list insertion/delete/search). Then, we discuss lock granularity and contention, followed by the concept of thread-safe vs. thread-unsafe.

Module 4 - Synchronization with Semaphores: We emphasize commonly used semaphore programming models such as locks, counters, communications and rendezvous. Classical examples are also discussed, followed by an actual application: threading the Mandelbrot set computation program.

Module 5 - Synchronization with Condition Variables: This module focuses on different types of monitors (*i.e.*, Hoare and Mesa) using locks and condition variables. We shall build a mini-OS for managing resources such as acquiring/releasing a shared device and disk scheduler. Classical examples are also covered.

Module 6 - Synchronization with Message Passing: Message queues and mailboxes are introduced along with issues such as buffered/un-buffered, queue capacity and blocked and unblocked send and receive. The concept of rendezvous is reintroduced, followed by examples such as pipeline sort, parallel sieve and *N*-queens.

Module 7 - Back to Basics: This module leads the students into the systems level addressing thread execution environments, thread state diagram, and thread priority and scheduling. Models such as 1-to-1, *many*-to-1 and *many*-to-*many* are covered, followed by discussions of kernel vs. user threads and monolithic vs. threaded operating systems.

Module 8 - Coping with Signals: Topics covered include the concepts and meaning of signals, synchronized signals, and sending/waiting/handling signals. Two important skills will be mentioned: signal processing in a MT environment and asynchronized-signal safe thread programming. As an actual programming example, a thread dispatcher is explained.

Module 9 - Selected Topics in Pthreads: Selected topics of the POSIX Pthreads standard are introduced in this module.

Module 10 - The Concept of a Process: We step out of the thread and into the process world. Topics include the state diagram of a process and creation/termination/join of processes using UNIX system calls `fork()`, `exit()` and `wait()`. The UNIX System V IPCs (*i.e.*, shared memory, semaphore and message queue) are discussed and compared and contrasted with those of threads.

Module 11 - Threads in Parallel Computing: The techniques of using threads in a parallel computing environment are addressed in this module. In addition to topics covered in previous modules, they also learn load balancing, event synchronization, the basics of MPI, shared- and distributed-memory and data parallel programming. Parallel programming models such as work-crew, boss-workers, pipeline and master/slave are discussed.

Module 12 - Threads in Distributed Computing: Major topics covered in this module include UICI and a buffered MT communication system, remote procedure calls, distributed synchronization mechanisms and distributed solutions of several classical problems (*e.g.*, dinning philosophers, distributed sieve and *N*-queens).

4.3 Software Tools

The software tools accompanying this course consists of the following components:

Visualization
Using OO technology, threads and various synchronization mechanisms are wrapped into classes with which students can set switches to select features. These features include, but are not limited to, (1) visualizing thread creation, join, and termination, (2) visualizing the internal working of synchronization mechanisms such as the content of a semaphore queue, a monitor's boundary, a condition variable queue, rendezvous of threads, the effect of blocking and unblocking sends and receives, lock contention, starvation, etc., and (3) visualizing fundamentals of parallel and distributed computing such as reductions, broadcasting, sockets and RPCs.

We combine and take advantage of three approaches, namely static, real-time and post-mortem analysis. Static analysis (e.g., possible deadlocks and complexity of synchronization mechanisms) of a student's program can pinpoint some potential problems before it is run. Since static analysis is not powerful enough for precisely detecting all possible anomalies, on-the-fly analysis is required for detecting other problems such as deadlocks. We also believe showing synchronization activities on-the-fly would be more interesting and helpful as a student can actually see the behavior of his program on-the-fly. Since only some anomalies can be detected with static and on-the-fly approaches, a post-mortem approach is also used in our system.

Deadlock Detection
The on-the-fly detection, which can only report deadlocks that actually occur, can easily be implemented and be incorporated into the visualization module. While on-the-fly detection has been very popular in many pedagogical languages, reporting potential deadlocks before a program is run helps students in designing good MT programs without getting into the change-run-debug cycle.

Data Race Detection
This is the most challenging module. There are an unbounded number of potential data races in a MT program and this is why students always have difficulties in finding data races. Methods for detecting data races can be static, on-the-fly, and post-mortem. Since this module is a programming/debugging aid for undergraduate students whose programs are not very complicated, only static and post-mortem methods are included.

Software Metric Development
Software metrics for MT programs are virtually non-existent and extra research efforts are required. This measure should report the complexity of the thread structures and synchronization mechanisms in a student's MT program to help students writing better programs. This module may be primitive; but it will be informative and offer an important topic for further research.

Other Visualization Aids
In addition to the above programming aids, we believe some predefined animation sequences would also be very helpful. Some events may not occur as expected and require careful planning to locate and display. For example, some programming models such as client/server, pipeline and peer can easily be discussed with predefined animation sequences illustrating the flow of control and data, the communication operations among threads, and so on. We plan to compile as many predefined animation sequences as possible and make them available on the Internet.

5. CONCLUSION

In previous sections, we have presented the details of the design of a MT programming course and its accompanying software tools. We have used about 50% of the materials for a programming track in an introduction to operating systems course three times. We also reported some difficulties we have experienced and proposed software tools to address these problems. The development of these tools is being supported by National Science Foundation. We intend to release these tools to interested educators in the future.

With the continuing emergence of MT computation as a powerful vehicle for science and engineering, the need for an introduction to MTP for scientists and engineers is high. Our course is necessary not only for CS undergraduates but also for engineers as an introduction to scientific MTP methodology. It is expected that this course will be popular across campuses and become a long-term part of the CS curriculum.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation under grant DUE-9752244. The first author is also partially supported by the National Science Foundation under grant DUE-9653244.

REFERENCES

1. G. R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, 1991.

2. M. Ben Ari, *Principles of Concurrent Programming*, Prentice Hall, 1982.

3. T. 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.

4. J. E. Beveridge and R. Wiener, *Multithreading Applications in Win32*, Addison-Wesley, 1997.

5. C. Brown, *UNIX Distributed Programming*, Prentice Hall, 1994.

6. A. Burns and G. Davies, *Concurrent Programming*, Addison-Wesley, 1993.

7. D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

8. B. Bynum and T. Camp, After You, Alfonse: A Mutual Exclusion Toolkit, *ACM Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, February 15-18, 1996, pp. 170-174.

9. W. Cai, W. J. Milne and S. J. Turner, Graphical Views of the Behavior of Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol. 18 (1993), pp. 223-230.

10. D. Callahan and J. Subhlok, Static Analysis of Low-level Synchronization, *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988, pp. 100-111.

11. N. Gehani, *Ada: Concurrent Programming*, Prentice-Hall, 1984.

12. S. J. Hartley, Animating Operating Systems Algorithms with XTANGO, *ACM Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, Phoenix, March 10-11, 1994, pp. 344-348.

13. S. J. Hartley, *Operating Systems Programming*, Oxford University Press, 1995.

14. S. J. Hartley, *Concurrent Programming: The Java Programming Language*, Oxford University Press, 1998.

15. C. William Higginbotham and R. Morelli, A System for Teaching Concurrent Programming, *ACM Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, San Antonio, March 7-8, 1991, pp. 309-316.

16. J. E. Hopcroft, The Impact of Robotics on Computer Science, *Communications of the ACM*, Vol. 29 (1986), No. 6 (June), pp. 486-498.

17. S. Kleiman, D. Shah and B. Smaalders, *Programming with Threads*, Prentice Hall, 1996.

18. D. Lea, *Concurrent Programming in Java*, Addison Wesley, 1997.

19. B. Lewis and D. J. Berg, *Threads Primer*, Prentice Hall, 1996.

20. B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*, Prentice Hall, 1998.

21. H-I Lu, P. N. Klein and R. H. B. Netzer, Detecting Race Condition in Parallel Programs that Use one Semaphore, Technical Report CS-93-29, Department of Computer Science, Brown University, June 1993.

22. S. P. Masticola, A Model of Ada Programs for Static Deadlock Detection in Polynomial Time, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 20-21, Santa Cruz, California, 1991, pp. 97-109.

23. S. P. Masticola and B. G. Ryder, Static Infinite Wait Anomaly Detection in Polynomial Time, LCSR-TR-114, Laboratory for Computer Science Research, Rutgers University, 1990.

24. R. H. B. Netzer and S. Ghosh, Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization, *International Conference on Parallel Processing*, August 1992, pp. II242-II246.

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

26. B. Nichols, D. Buttlar and J. P. Farrell, *Pthreads Programming*, O'Reilly, 1996.

27. S. Oaks and H. Wong, *Java Threads*, O'Reilly, 1997.

28. B. A. Price and R. M. Baecker, The Automatic Animation of Concurrent Programs, Proceedings of *First International Workshop on Computer-Human Interface*, Moscow, August 5-9, 1991, pp. 128-137.

29. K. A. Robins and S. Robins, *Practical UNIX Programming: A Guide to Concurrency and Multithreading*, Prentice Hall, 1996.

30. C-K Shene, Multithreaded Programming in an Introduction to Operating Systems Course, *Twenty-ninth ACM Annual SIGCSE Technical Symposium*, February 26-March 1, 1998, pp 242-246. Course information, including exams and solutions, programming assignments, project and some overheads used in class, are available at

http://www.csl.mtu.edu/cs270/www/Home.html.

31. J. T. Stasko, The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report, Technical Report GIT-GVU-95-03, College of Computing, Georgia Institute of Technology, 1995.

32. R. N. Taylor, A General-Purpose Algorithm for Analyzing Concurrent Programs, *Communications of the ACM*, Vol. 26 (1983), No. 5 (May), pp. 362-376.

33. R. N. Taylor, Complexity of Analyzing the Synchronization Structure of Concurrent Programs, *Acta Informatica*, Vol. 19 (1983), pp. 57-84.

34. M. Young and R. N. Taylor, Combining Static Concurrency Analysis with Symbolic Execution, *IEEE Transactions on Software Engineering*, Vol. 14 (1988), No. 10, pp. 1499-1511.

35. Q. A. Zhao and J. 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.

36. M. Zimmermann, F. Perrenoud and A. Schiper, Understanding Concurrent Programming Through Program Animation, *ACM Nineteenth SIGCSE Technical Symposium on Computer Science Education*, Atlanta, Georgia, February 25-26, 1988, pp. 27-31.