# ACM SIGACT News Distributed Computing Column 33

*Teaching Concurrency*

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

As multi-core computer architectures are becoming mainstream, it is widely believed that the biggest challenge facing computer scientists today is learning how to exploit the parallelism that such architectures can offer. For example, Bill Dally, chair of Computer Science at Stanford, is quoted[1] as saying:

> "Parallel programming is perhaps the largest problem in computer science today and is the major obstacle to the continued scaling of computing performance that has fueled the computing industry, and several related industries, for the last 40 years."

In fact, the media buzz around multi-core programming is said to be rivaling the buzz on global warming[2].

This column looks at the multi-core programming problem from an educational perspective. How can those among us who teach, help students become comfortable with concurrency? A workshop on this very topic, organized by Nir Shavit, will take place on March 8, co-located with ASPLOS'09. I include here an announcement about the workshop, as provided by Nir.

The column then proceeds with a review, by Danny Hendler, of the book "Synchronization Algorithms and Concurrent Programming" by Gadi Taubenfeld. Danny discusses how material from the book can be used in both undergraduate and graduate courses. The next contribution, by Alan Fekete, tackles concrete questions of curriculum design: What courses in the undergraduate computer science curriculum should deal with concurrency? What should be taught about concurrency? and how? As might be expected, there are no definite answers to these questions, yet Alan provides a through discussion of the pros and cons of various approaches and what works well in different contexts. Alan also surveys related results from the Computer Education literature. The column concludes with Leslie Lamport's high-level vision of how

[1] Stanford Report, April 30, 2008

[2] Multicore Programming Blog: The Multicore Software Challenge: Catching up to Global Warming? http://www.cilk.com/multicore-blog/bid/5101/The-Multicore-Software-Challenge-Catching-up-to-Global-Warming

computer scientists should learn to think about computations, both serial and concurrent. This thought-provoking article takes a step back from the details of where, what, and how, and makes a case for the high level goal of teaching students how to think clearly. Many thanks to Danny, Alan, and Leslie for their contributions!

**Call for contributions:** I welcome suggestions for material to include in this column, including news, reviews, opinions, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

## Multicore Programming Education 2009
## Workshop on Directions in Multicore Programming Education

Sunday, March 8, 2009, Washington DC, USA. Co-located with ASPLOS 2009.

The acceptance of multiprocessors as the mainstream computing platform requires a fundamental change in the way we develop software. There are no textbooks and few reference books addressing how to program multiprocessors. Most engineers learn the tricks of the trade by asking help from more experienced friends and through a laborious trial and error process. This state of affairs must change, and the change must come through the development of an appropriate educational curriculum, one that addresses todays needs and one that will capture the rapid developments in multiprocessing hardware, operating systems, and programming languages.

A glance at the 2001 ACM computing curricula shows that net-centric computing is introduced at the intermediate level as a core topic. This emphasis reflects the advent of the web and Internet in the mid-90s. We expect that multiprocessor architectures will have an equally broad effect on everyday programming, and that similarly, the associated body of knowledge will become a core area of the curriculum within a few years.

This workshop will attempt to bring together researchers and educators, with the goal of developing, through dialog, a better understanding of the possible directions to take in order to bring multicore programming education to the masses.

The workshop will feature 8 distinguished speakers, prominent researchers and educators in the area of multiprocessor programming. The speakers will present their approach, experiences, and suggestions for educational directions, and later participate in an open panel with the workshop attendees.

Workshop Registration is free, including Lunch and Coffee breaks. *Remember to register for MPE 2009 when you register for ASPLOS!*

**Invited Speakers**

| | |
|---|---|
| Arvind | MIT |
| Guy Blelloch | CMU |
| Maurice Herlihy | Brown |
| Dan Grossman | Washington |
| Tim Mattson | Intel |
| Michael Scott | Rochester |
| Marc Snir | Illinois |
| Guy Steele | Sun Microsystems |

Book Review of

**Synchronization Algorithms and Concurrent Programming**

**Author of Book: Gadi Taubenfeld**

**Publisher: Pearson / Prentice Hall, 2006, 433 pages**

Reviewed by Danny Hendler, Ben-Gurion University, hendlerd@cs.bgu.ac.il

# 1 Introduction

Although there are quite a few books that deal with various aspects of the theory and practice of multiprocessor synchronization, only a handful of them fall into the category of 'textbooks' ([2, 13, 18] are notable examples); Gadi Taubenfeld's recent book, "Synchronization Algorithms and Concurrent Programming", is certainly one of them.

Taubenfeld's book focuses on inter-process synchronization in shared-memory systems. The book provides an in-depth discussion of algorithms and lower bounds for a wide range of synchronization problems, such as mutual exclusion, barrier synchronization, $l$-exclusion, deadlock-prevention, the dining philosophers problem, the producer-consumer problem, consensus, and many more.

As stated in its preface, the book is meant to be used as a textbook for upper-level undergraduate or graduate courses and special emphasis has been given to make it accessible to a wide audience. The book's structure and methodology reflect these goals in several ways:

- The book is mostly self-contained.
- Each section ends with a set of self review questions and their solutions.
- Each chapter ends with a section that contains numerous exercises, categorized according to their estimated difficulty level. For example, the book contains more than 160 mutual-exclusion related exercises. Many of these exercises suggest variants of the algorithms presented in the text and ask the student to analyze their properties. While teaching mutual exclusion algorithms, I found these exercises very constructive in providing intuitions and in deepening students' understanding of synchronization algorithms.
- There is a Companion Web-site that contains PowerPoint presentations covering most of the book's chapters and all of the figures that appear in it.

# 2 Summary of Contents

The first chapter demonstrates the difficulty of inter-process synchronization by two examples that appeal to the reader's intuition: the Too Much Milk problem (an allegory illustrating the mutual exclusion problem,

due to Prof. John Ousterhout) and the Coordinated Attack problem (a.k.a. *the two generals problem* [10]). It then formally introduces the mutual exclusion problem, the complexity measures used to analyze it, and some basic concepts of shared-memory multiprocessors.

Chapters 2 and 3 provide a comprehensive discussion of mutual exclusion using atomic read/write registers. Chapter 2 presents classic mutual exclusion algorithms, such as Peterson and Kessels' 2-process algorithms [14, 21], tournament algorithms, Lamport's fast mutual exclusion algorithm [16], and the bakery algorithm [15]. Correctness proofs for all these algorithms are provided, as well as the linear space lower bound on mutual exclusion using read/write registers due to Burns and Lynch [4]. Some recent results by the author (the Black-Write bakery algorithm [25] and automatic discovery of mutual-exclusion algorithms [3]) are also described. Chapter 3 deals with more advanced read/write mutual exclusion topics, such as local-spin algorithms, adaptive and fault-tolerant mutual exclusion algorithms and impossibility results, and symmetric mutual-exclusion algorithms.

Chapter 4 discusses blocking and non-blocking synchronization using strong read-modify-write primitives. Section 4.1 introduces strong synchronization primitives. Section 4.2 describes techniques for collision avoidance using test-and-set bits. Section 4.3 describes the Ticket algorithm [5]. Section 4.4 presents three queue-based local-spin mutual-exclusion algorithms: Anderson's algorithm [1], Graunke and Thakkar's algorithm [9], and the MCS algorithm [19]. Section 4.5 discusses non-blocking and wait-free implementations of concurrent objects and presents Michael and Scott's queue algorithm [20]. Chapter 4 concludes with a treatise of semaphores, monitors, and notions of concurrent object fairness.

Chapter 5 discusses the issue of barrier synchronization. Several barrier algorithms that use an atomic counter are presented, followed by test-and-set-based barriers, a combining tree barrier, a tree-based barrier [19], the dissemination barrier [11], and the See-Saw barrier (based on [7]).

Chapter 6 deals with the $l$-exclusion problem. It presents two read/write algorithms due to Peterson [22], followed by a few algorithms that use strong synchronization primitives. The chapter concludes with a discussion of the *$l$-assignment* problem, which is a stronger version of $l$-exclusion.

Chapter 7 discusses problems that involve multiple resources. After defining the concept of deadlock in such problems, a few techniques of deadlock-prevention are described, followed by a discussion of the Deadly Embrace problem. The second part of this chapter provides an in-depth discussion of the Dining Philosophers problem.

Chapter 8 presents additional classical synchronization problems, such as the Producer-Consumer problem, the Readers and Writers problem, the Sleeping Barber and the Cigarette Smoker's problems, group mutual exclusion, room synchronization, and more.

Chapter 9 deals with the consensus problem. Apart from the classical results (the impossibility of consensus with one faulty process [6, 17], Herlihy's wait-free hierarchy and the universality of consensus [12]), some less known consensus algorithms are presented, such as an algorithm of Fischer et al. for achieving consensus without memory initialization [8].

The book concludes with Chapter 10, which presents several timing-based mutual-exclusion, fast mutual-exclusion and consensus algorithms, for both the known- and the unknown-delays models.

## 3 Personal Teaching Experience

I am using "Synchronization Algorithms and Concurrent Programming" for teaching two courses: an undergraduate course on Operating Systems and a graduate course on multiprocessor synchronization algorithms.

The classic Operating Systems textbooks, such as the books by Tanenbaum [24] and Silberschatz et al. [23], do provide some material on shared-memory synchronization. In light of the recent emergence

of multi-core computing, however, I maintain that undergraduate students should gain deep understanding of shared-memory synchronization; some important synchronization topics, such as the use of strong read-modify-write primitives and local-spin algorithms, are not sufficiently covered by these classic OS textbooks. I am therefore using Taubenfeld's book, in addition to these classic textbooks, for providing a deeper treatise of mutual exclusion algorithms, semaphores, monitors, and other synchronization problems, such as the dining philosophers problem.

The graduate course I teach deals with both theoretical and practical aspects of multiprocessor synchronization algorithms. For this course, I am using Taubenfeld's book, in addition to other Distributed Computing textbooks [2, 13]. My first presentation for this course uses the Too Much Milk and the Coordinated Attack problems to provide basic intuitions into the difficulty of devising correct synchronization algorithms.

The first part of my graduate course deals with the mutual exclusion problem. Here, I am using many of the slides available in the book's companion web-site for illustrating the ideas underlying classical algorithms, such as Lamport's fast mutual exclusion algorithm [16], and the bakery algorithm [15]. Lamport's Bakery algorithm is presented in Taubenfeld's book and slides by a sequence of less and less naïve "approximations" of the original algorithm. This methodology is also used for several other mutual exclusion algorithms, and both my students and myself find it very useful. The numerous synchronization exercises provided by Taubenfeld's book make preparing home assignments and exams for both these courses much easier.

In summary, "Synchronization Algorithms and Concurrent Programming" is, in my opinion, a fine addition to the Distributed Computing bookshelf. Its structure and methodology make it accessible to a wide audience, which make it useful as a textbook. The rich variety of synchronization research results covered by it make this book valuable also as a reference for researchers in the Distributed Computing community.

## References


[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6-16, 1990.

[2] H. Attiya and J. Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2'nd Edition. John Wiley and Sons, 2004.

[3] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proc. 17th International Symp. on Distributed Computing*, LNCS 2648, Springer-Verlag, 2003, 136-150.

[4] J.E. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171-184, December 1993.

[5] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90-114, 1989.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382, 1985.

[7] M. J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld. The wakeup problem. *SIAM Journal on Computing*, 25(6):1332-1357, 1996.

[8] M. J. Fischer, S. Moran, and G. Taubenfeld. Space-efficient asynchrnous consensus without shared memory initialization. *Infomration Processing Letters*, 45(2):101-105, 1993.

[9] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 23(6):60-69, 1990.

[10] J. Gray, Notes on database operating systems. *Operating Systems, an Advanced Course*, Lecture Notes in Computer Science, 60:393-481, 1978.

[11] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1-17, 1988.

[12] M. P. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124-149, 1991.

[13] M. Herlihy and N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufman, 2008.

[14] J. L. W. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17(2):135-141, 1982.

[15] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453-455, 1974.

[16] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1-11, 1987.

[17] M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, 1987.

[18] N. Lynch. Distributed Algorithms. Morgan Kaufman, 1996.

[19] J.M Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21-65, 1991.

[20] M. M. Michael and M. L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, 267-275, 1996.

[21] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115-116, 1981.

[22] G. L. Peterson. Observations on $l$-exclusion.In *Proc. 28th Annual Allerton Conference on Communication, Control and Computing*, pages 568-577, October 1990.

[23] A. Silberschatz, P. Galvin and G Gagne. Operating System Concepts, 6'th Edition. John Wiley and Sons, 2003.

[24] A. Tanenbaum. Modern Operating Systems, 3'rd Edition. Pearson / Prentice Hall, 2008.

[25] G. Taubenfeld. The black-white bakery algorithm. In *Proc. 18th International Symp. on Distributed Computing*, LNCS 3274, Springer-Verlag, 2004, 56-70.

# Teaching about Threading: Where and What?

Alan D. Fekete
School of Information Technologies
University of Sydney, 2006 Australia
fekete@it.usyd.edu.au

**Abstract**

Trends in hardware and software are increasing the importance of concurrent programming, in the skill set expected of graduates. The curriculum is fairly full already, so teachers face complicated trade-offs in deciding how to incorporate additional material related to concurrency. In this paper we discuss some of the different ways to cover thread programming; we also survey the literature on this topic from the Computing Education community.

## 1 Introduction

The traditional Computer Science curriculum has included concurrency as a topic, but in the past, few graduates found day-to-day use for this knowledge in their professional practice. This is changing: where threading used to be a matter for five-star wizards writing systems infrastructure software, most of today's students are becoming aware of the need for experience with concurrent execution. Hardware trends are making multicore machines more and more common, and skill with concurrency is needed to achieve the performance benefits such hardware promises. Cloud computing platforms are raising awareness of the need to have software that can be effectively parallelized. Exposure to the typical shrink-wrapped application leads students to realize that their projects will also need impressive and responsive GUIs, which can't happen without the competence to manage several threads. We can point to the prevalence of the multi-tier software architectures, where arbitrary business objects (such as any developer may be writing) are executed from several concurrent threads through an application server. Languages like Java and C# have threading built-in, and this again raises the student's awareness of thread management, race conditions, deadlock and the like.

The field of Computer Science is growing rapidly, with many topics (multi-media, personalization, virtualization, visualization, gestural and haptic interfaces, semantic web technologies, social network analysis, etc) claiming increased space in the undergraduate curriculum. This is especially difficult for universities following a US-style liberal arts approach, where a major should consist of no more than 10-12 subjects among the 32 that make a 4-year degree. With constraints like this, any introduction of new material means the removal or compression of other topics, and it is becoming more common to integrate related topics into a single subject (eg multimedia and graphics taught together, or programming languages and compilers combined, or operating systems with distributed computing). The material on concurrency can be integrated in

several different ways with existing subjects of the curriculum, and we describe several of these approaches below. Alternatively, one might give concurrency a subject to itself, and make room by merging two existing subjects elsewhere in the curriculum.

In this paper, we focus on teaching students about concurrent programming in a style where a number of separate computations written as sequential code share some state variables (ie. we assume a model of threads or processes). This is the natural framework for the ideas of critical sections, data interference, as well as providing the dangers of deadlock. This is also the model that lends itself very easily to practical assignments, using languages with built-in synchronization primitives or libraries of synchronization primitives for languages like C. There are of course other models for distributed and concurrent programming, such as Actor models, explicit message-passing, or implicitly parallel functional approaches. It is a valid educational approach to teach one of these approaches first. Papers describing courses based on other models include [19] with Map-reduce for cluster computing, and [18] with a CSP-derivative called occam-$\pi$ in the context of robots. Once students have a good feel for parallelism in a simpler model, they can then be introduced to the extra complexity of shared variables, in the ways described below (but we hope with less time devoted to the material, since the essentials of concurrent thinking will already be familiar).

## 2 Concurrent Programming as a separate subject

If one has room in a curriculum to introduce an extra subject whose main focus is concurrency, then there are several textbooks which might provide the framework. These courses have quite diverse levels of sophistication, and would fit in different parts of a curriculum.

A widely-used textbook on concurrency was written by famous researchers from the Software Engineering community: Magee and Kramer from Imperial College [22]. This is aimed at students early in their study (eg first or second year in a European degree, sophomore or junior year in an American one), when they know the basics of object-oriented programming. The text is notable for placing strong focus on tool-supported formal methods. The behavior of Java constructs is clearly explained, but instead of coding directly with this understanding, students are taught a design approach in which they first produce a model of what they want, in a notation called FSP (a restricted form of CSP), and they use a tool LTSA, which allows both animation and exploration of the model, and also model-checking to find a trace with particular properties (or show that none exists). Only after the student is convinced the design is correct, are they expected to convert it into Java code with synchronize blocks etc. My experience with this approach is mixed: it helps students produce correct designs, but it leaves them unprepared to maintain, or to critique, code written in idiomatic Java style.

Readers of this column won't need urging to become excited about the recent book [14] by Maurice Herlihy and Nir Shavit (winners of the 2004 Godel Prize). This is aimed at more advanced students than Magee-Kramer (say, senior US undergraduates), in particular, it assumes basic courses in hardware and infrastructure systems (concepts like caches, processors, memory consistency). It covers theory of Lamport-style constructions that built one memory primitive from others, and the practice of fine-grained and even lock-free synchronization for basic data structures. I can't imagine this being the way we bring concurrent programming ideas to the masses of CS graduates, but it would be invaluable for the five-star wizards.

One must also mention the most respected book on concurrent programming for practioners (at least, those using Java). Brian Goetz is an acknowledged Java wizard, who presents at the main developers' conference, and Doug Lea designed the new `java.util.concurrent` library classes that bring high-performance implementations of collections, in Java 5 and 6. Their book [9] is a must-read for any coder. With some supplements from Lea's earlier book [20], it could be the basis for an excellent upper-division

course, for students who have really internalized OO design, design patterns, etc.

## 3 Concurrency within traditional subjects

Since this is a column on Distributed Computing, we should begin by considering whether to teach process/thread synchronization topics within a subject whose main focus is Distributed Computing (typically taught in the higher years). The knowledge and skills of identifying critical sections and potential interference are clearly going to be exercised extensively in such a course. Understanding thread programming is of course important for those distributed algorithms that are considered in a shared variable model, but also, expressing algorithms for a message-passing model requires care in coding the interaction between different steps each of which touches the state stored within a single node. Where the handling of a message or event is described with code that includes waiting for a condition, we need to pay attention to the concurrency between executions of individual events. My judgement is that it would be too intense, to try to introduce awareness of the problems of threading, in the same subject that has to introduce distribution, asynchrony, fault-tolerance, etc. Rather, I would like students to be aware of the nature of non-deterministic execution, data interference, basic locking, etc, before they come to the distributed computing subject, so the ideas have sunk in, and so that the distributed computing course can really build on the earlier work. Most textbooks for Distributed Computing assume that readers come with prior awareness of the ideas of critical section and data interference.

The traditional place where concurrency was introduced, is in an Operating Systems subject, as part of the discussion of process management and inter-process communication. In the widely accepted ACM/IEEE Computer Science curriculum CC2001[1], the material on concurrency is placed within the Operating Systems topic, and is described as OS3. Any OS course with practical work on implementing internals, needs students who really understand how to recognize and prevent race conditions, interference and/or deadlock. In the past, C was the language of choice for an OS course, and students learnt about coding complicated patterns of synchronization with explicit semaphore operations. This approach is immortalized in a series of bizarre puzzles for students to solve with the maximum allowed concurrency (starting with dining philosophers, but including sleepy barbers, cigarette-smokers, and cannibals). These puzzles are quite artificial, and they have no obvious connection to any real-world tasks; however, for students who don't need the practical motivation they can be excellent for building mastery of concurrency and its dangers. More recently, Java is sometimes used in the OS course, but the approach is still to teach students how to provide sophisticated synchronization, making extensive use of wait() and notifyAll() methods.

Many textbooks for Programming Languages include a chapter or two on Concurrent Programming (following say Functional programming and Logic Programming). These books usually describe the language features of several languages (usually at least one will be a language with monitors such as Java), and the texts motivate the value of these features by some examples of interference and deadlock. However, the focus is mainly on the language mechanisms, and comparisons between languages, rather than being on the programming techniques. I am not aware of any successful teachers who found that this material has really helped students become better concurrent programmers.

In some curricula, a year-long first-year subject (two successive semesters) is devoted to learning to program, starting with simple flow-control and object-oriented thinking, and proceeding to cover topics like JDBC, sockets, GUIs, etc. This approach is particularly appropriate for students who have a vocational view, for example, students in a industry-linked degree with placements after the first or second year; they

[1] See `http://www.acm.org/education/curricula-recommendations`

realize that a job-ready professional software developer needs detailed knowledge of how to code different sorts of application. There are a number of comprehensive Java programming textbooks for a course like this, such as [16], [5] or [26]. These texts typically include a chapter on threading, usually focused on the main mechanisms for thread creation and management, and the basics of data interference, and its solution with synchronization. This can provide a motivating first coverage of concurrency and threading issues. The students' understanding can then be reinforced during discussion of socket programming and server construction and/or in the GUI coding topic. However it is quite hard for students to get enough practice to become proficient, with so many diverse topics sharing their attention.

The Data Structures subject provides another place where concurrent programming can be fitted early into every student's education, so that it can be built on repeatedly in later subjects. Here one starts naturally from the design of abstract data types, explains how to implement a data type, and one can continue with the question of how to code a type in a thread-safe manner, that is, so it can be accessed concurrently from several threads. The essential content here is to teach students to recognize when methods need synchronization, and then they can learn how to synchronize (by ensuring that each piece of potentially conflicting code is enclosed in a synchronized block governed by a common lock). There are natural design choices, from coarse-grained to fine-grained synchronization, that show up in the choice of which object to lock. The author has had some success in doing this as enrichment for high-achieving students, but unfortunately the topic is not yet covered in any of the usual data structures textbooks.

We should also mention the database subject, where concepts related to concurrency have a time-honoured role. Many curricula now divide the database material into two subjects: one on using databases for knowledge management and in application development, and one on dbms internals. The first course will introduce the transaction concept, and explain the ACID properties it should give. This requires discussion of the problems of concurrency and interleaving, with examples of data interference. The second course shows the locking mechanisms that are used to provide ACID; also deadlock detection is covered. However, there is a fundamental difference between the material as taught in databases and that needed for a concurrent programmer. In a database, the application simply declares its transaction boundaries, and the infrastructure must provide the locking; in concurrent programming (except in recent proposals for transactional memory), the application programmer must themselves make decisions about what and when to lock.

## 4 Guide to the CS Education literature

The most important forums for innovative teaching are the annual conferences sponsored by ACM SIGCSE. The main SIGCSE conference is held in the US, and the ITiCSE conference is always in Europe. Each year one can expect to find several papers looking at ways to teach concurrency or related topics. Here we mention a few as starting points for interested readers.

There are several papers [23, 15, 17, 21, 19] which describe interesting instructional operating systems. Typically, these systems are used in assignments which include substantial work on concurrency issues.

A series of papers by Hartley [10, 11, 12, 13] has focussed on using Java to teach the traditional synchronization topics.

Several research groups have presented papers [1, 3, 4, 7, 24, 25] which describe tools that simulate, animate, or detect race conditions in threaded code.

I have described my own experience teaching concurrency in the context of designing thread-safe classes [8].

A recent article [6] has shown how to use Java threading across a succession of introductory subjects;

starting with examples which are carefully chosen so no synchronization is needed, and then gradually covering more and more complicated forms of parallelism.

# 5 Conclusions?

Curriculum design is an intractable problem. There is clearly more material that a graduate will be expected to know, than can be covered in the time available, and also some topics take less than a full semester to cover; thus a bin-packing task arises, choosing which topics to leave out and which to combine with one another. One also has to order the material (perhaps a partial order, since many curricula leave students some choice), and there are many cases of circular dependency, where each one of a collection of topics would be easier to teach if the other topics were done first. In any institution, one needs to balance out the best training for each of the different career paths graduates might follow, with the faculty's interests (and their desire to prepare students for research). For all these reasons, I can't give any decisive guidance on where to teach concurrency. However, there are two overall curriculum designs that are supported by good textbook choices and that I believe can work well, in different contexts where there is a clear set of career paths for which students are preparing.

Some universities can offer a traditional, high-end computer science curriculum. This would be suited to a student population who are very capable, motivated and most of whom will follow careers in developing infrastructure computing (eg working in technical roles for major vendors, or in startups). Skill at working in C and C++ (including memory management issues) is likely to be essential for this career path. This environment can support a curriculum where all (or almost all) graduates take courses on graphics algorithms, computer architecture, operating systems internals, networking protocols, etc. Here, it makes sense to cover thread management within the operating systems subject, as essential foundation which is used and reinforced during the typical assignments where OS internals are implemented (or modified). Even deeper understanding can come in a course on distributed computing, and/or one on multiprocessor systems and their software.

On the other hand, suppose the student cohort is headed mainly for careers as developers of business applications (with aspirations towards team leadership then management), working perhaps in the IT department for large corporates, or for an outsourcing supplier, or perhaps developing shrink-wrapped software for sale to business users. These graduates need a lot of experience in Java (including its many libraries) but probably won't make use in the workplace of any deep mastery of the lower-level languages. A curriculum for these graduates would center on subjects covering usability and interface issues, design patterns, systems analysis, software quality processes, business process management, etc. For a department like this, I strongly advocate the model of a year-long introductory programming course, where threading is one among the topics in the second semester.

However, many departments don't have the luxury of designing a curriculum for a uniform cohort of students, with a common style of career path; nor do they have resources to provide separate sets of courses for each type of graduate. In such a context, any curriculum will need to be a compromise between the needs of different students. My preference here is to design the main core of the curriculum for the most numerous type of student, and then provide a few electives targeted separately at each substantial (or otherwise important) minority group. For example, one could have a curriculum mainly for developers of business applications (with concurrency introduced in the second semester of a year-long programming introduction), and then add an elective on systems infrastructure (including OS internals, perhaps using [2] as textbook) where the tricky and sophisticated forms of synchronization are covered. As they say, ”your mileage may vary”.

No matter where concurrency is covered, one needs to devote enough time and academic resources to it, especially in the assignments and lab sessions. This topic is among the hardest in the whole curriculum for students to build up their experience, because of non-deterministic execution and the resulting likelihood that a student believes that their incorrect code is working properly. While there are some tools available that can help with visualization or debugging, these are limited in scope. We need to help students to learn through one-on-one feedback from an experienced teacher, pointing out incorrect possible traces of their code. This needs to happen repeatedly, and so concurrency as a topic needs plenty of space and visibility where-ever it is placed within the curriculum.

## References


[1] M. Ben-Ari. A suite of tools for teaching concurrency. In *ITiCSE '04: Proc. of 9th SIGCSE conference on Innovation and technology in computer science education*, pp. 251–251. ACM, 2004.

[2] R. Bryant and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, Inc., 2003.

[3] S. Carr, J. Mayo, and C.-K. Shene. ThreadMentor: a pedagogical tool for multithreaded programming. *J. Educ. Resour. Comput.*, 3(1):1, 2003.

[4] S.-E. Choi and E. C. Lewis. A study of common pitfalls in simple multi-threaded programs. In *SIGCSE '00: Proc. of 31st SIGCSE technical symposium on Computer science education*, pp. 325–329. ACM, 2000.

[5] H. Deitel and P. Deitel. *Java: How to Program, 7/e*. Prentice-Hall, Inc., 2007.

[6] D. Ernst and D. Stevenson. Concurrent CS: Preparing Students for a Multicore World. In *ITiCSE '08: Proc of the 13th annual conference on Innovation and technology in computer science education*, pp. 230–234. ACM, 2008.

[7] C. Exton and M. Kölling. Concurrency, objects and visualisation. In *ACSE '00: Proc. of the Australasian conference on Computing education*, pp. 109–115. ACM, 2000.

[8] A. Fekete. Teaching students to develop thread-safe java classes. In *ITiCSE '08: Proc of the 13th annual conference on Innovation and technology in computer science education*, pp. 119–123. ACM, 2008.

[9] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[10] S. J. Hartley. 'Alfonse, your java is ready!'. In *SIGCSE '98: Proc. of 29th SIGCSE technical symposium on Computer science education*, pp. 247–251. ACM, 1998.

[11] S. J. Hartley. 'Alfonse, wait here for my signal!'. In *SIGCSE '99: The proceedings of 30th SIGCSE technical symposium on Computer science education*, pp. 58–62. ACM, 1999.

[12] S. J. Hartley. 'Alfonse, you have a message!'. In *SIGCSE '00: Proc. of 31st SIGCSE technical symposium on Computer science education*, pp. 60–64. ACM, 2000.

[13] S. J. Hartley. 'Alfonse, give me a call!'. In *SIGCSE '01: Proc. of 32nd SIGCSE technical symposium on Computer Science Education*, pp. 229–232. ACM, 2001.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] D. Holland, A. Lim and M. Selzer. A new instructional operating system. In *SIGCSE '02: Proc. of 33rd SIGCSE technical symposium on Computer Science Education*, pp. 111–115. ACM, 2002.

[16] C. S. Horstmann. *Big Java, 3rd Edition*. John Wiley & Sons Ltd., 2007.

[17] D. Hovemayer, J. Hollingsworth and B. Bhattacharjee. Running on the bare metal with GeekOS. In *SIGCSE '04: Proc. of 35th SIGCSE technical symposium on Computer Science Education*, pp. 315–319. ACM, 2004.

[18] M. Jadud, J. Simpson, and C. Bisciglia. Patterns for Programming in Parallel, Pedagogically. In *SIGCSE '08: Proc. of 39th SIGCSE technical symposium on Computer Science Education*, pp. 231–235. ACM, 2008.

[19] A. Kimball, S. Michels-Slettvet, and C. Jacobsen. Cluster Computing for Web-Scale Data Processing. In *SIGCSE '08: Proc. of 39th SIGCSE technical symposium on Computer Science Education*, pp. 116–120. ACM, 2008.

[20] D. Lea. *Concurrent Programming in Java, 2nd edition*. Addison-Wesley, 1999.

[21] H. Liu, X. Chen, Y. Gong. BabyOS: a fresh start. In *SIGCSE '07: Proc. of 38th SIGCSE technical symposium on Computer Science Education*, pp. 566–570. ACM, 2007.

[22] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs, 2nd edition*. John Wiley & Sons, 2006.

[23] T. Nicholas and J. Barchanski. TOS: an educational distributed operating system in Java. In *SIGCSE '01: Proc. of 32nd SIGCSE technical symposium on Computer Science Education*, pp. 312–316. ACM, 2001.

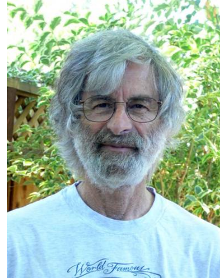
[24] R. Oechsle and K. Barzen. Checking automatically the output of concurrent threads. In *ITiCSE '07: Proc. of 12th SIGCSE conference on Innovation and technology in computer science education*, pp. 43–47. ACM, 2007.

[25] S. Robbins. Starving philosophers: experimentation with monitor synchronization. In *SIGCSE '01: Proc. of 32nd SIGCSE technical symposium on Computer Science Education*, pp. 317–321. ACM, 2001.

[26] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall, Inc., 2007.

# Teaching Concurrency

Leslie Lamport
Silicon Valley Research Center
Microsoft



I am not an academic. I have never even taken a computer science course. However, I have worked with a number of computer engineers (both hardware and software engineers), and I have seen what they knew and what they didn't know that I felt they should have. So, I have some thoughts about how concurrent computing should be taught. I am not concerned with traditional questions of curriculum—what facts should be stuffed into the student's brain. I long ago forgot most of the facts that I learned in school. What I have used throughout my career are the ways of thinking I learned when I was young, by some learning process that I have never understood. I can't claim to know the best way to teach computer engineers how to cope with concurrency. I do know that what they seem to be learning now is not helping them very much. I believe that what I am proposing here is worth a try.

I expect that the first question most computer scientists would ask about the teaching of concurrency is, what programming language should be used? This reflects the widespread syndrome in computer science of concentrating on language rather than substance. The modern field of concurrency started with Dijkstra's 1965 paper on the mutual exclusion problem [1]. For most of the 1970s, one "solved" the mutual exclusion problem by using semaphores or monitors or conditional critical regions or some other language construct. This is like solving the sorting problem by using a programming language with a *sort* command. Most of your colleagues can explain how to implement mutual exclusion using a semaphore. How many of them can answer the following question: Can one implement mutual exclusion without using lower-level constructs that, like semaphores, assume mutually exclusive access to a resource? Quite a few people who think they are experts on concurrency can't.

Teaching about concurrency requires teaching some very basic things about computer science that are not now taught. The basic subject of computer science is computation. One would expect a computer science education to answer the question, what is a computation? See how many of your colleagues can give a coherent answer to that question. While there is no single answer that is appropriate in all contexts, there is one that is most useful to engineers: A computation is a sequence of steps.

The obvious next question is, what's a step? Western languages are verb oriented. Every sentence, even one asserting that nothing happens, contains a verb. At least to westerners, the obvious answer is that a step is the acting out of a verb. An *add* step performs an addition; a *send* step sends a message. However, the obvious answer is not the most useful one. Floyd and Hoare taught us to think of a computation as a sequence of states [2, 3]. A step is then a transition from one state to the next. It is more useful to think about states than sequences of steps because what a computing device does next depends on its current state, not on what steps it took in the past. Computer engineers should learn to think about a computation in terms

of states rather than verbs.

I have found that a major problem confronting engineers when designing a system is understanding what the system is supposed to do. They are seldom faced with well-understood tasks like sorting. Most often they begin with only a vague idea of what the system should do. All too often they start implementing at that point. A problem must be understood before it can be solved. The great contribution of Dijkstra's paper on mutual exclusion was not his solution; it was stating the problem. (It is remarkable that, in this first paper on the subject, Dijkstra stated all the requirements that distinguish mutual exclusion from fundamentally simpler and less interesting problems.) Programming and hardware-design languages don't help an engineer understand what problem a system should solve. Thinking of computations as sequences of states, rather than as something described by a language, is the first step towards such understanding.

How should we describe computations? Most computer scientists would probably interpret this question to mean, what language should we use? Imagine an art historian answering "how would you describe impressionist painting?" by saying "in French". To describe a computation we need to describe a sequence of states. More often, we need to describe the set of computations that can be produced by some particular computing device, such as an algorithm. There is one method that works in practice: describing a set of computations by (1) the set of all initial initial states and (2) a next-state relation that describes, for every state, the possible next states—that is, the set of states reachable from that state by a single step. The languages used by computer engineers describe computations in this way, but how many engineers or computer scientists understand this?

Once an engineer understands what a computation is and how it is described, she can understand the most important concept in concurrency: invariance. A computing device does the correct thing only because it maintains a correct state. Correctness of the state is expressed by an invariant—a predicate that is true in every state of every computation. We prove that a predicate is an (inductive) invariant by showing that it is true in every initial state, and that the next-state relation implies that if it true in any state then it remains true in the next state. This method of reasoning, often called the inductive assertion method, was introduced by Floyd and Hoare. However, they expressed the invariant as a program annotation; most people were distracted by the language and largely ignored the essential idea behind the method.

Invariance is the key to understanding concurrent systems, but few engineers or computer scientists have learned to think in terms of invariants. Here is a simple example. Consider $N$ processes numbered from 0 through $N-1$ in which each process $i$ executes

$$\begin{aligned} x[i] &:= 1; \\ y[i] &:= x[(i-1) \bmod N] \end{aligned}$$

and stops, where each $x[i]$ initially equals 0. (The reads and writes of each $x[i]$ are assumed to be atomic.) This algorithm satisfies the following property: after every process has stopped, $y[i]$ equals 1 for at least one process $i$. It is easy to see that the algorithm satisfies this property; the last process $i$ to write $y[i]$ must set it to 1. But that process doesn't set $y[i]$ to 1 because it was the last process to write $y$. What a process does depends only on the current state, not on what processes wrote before it. The algorithm satisfies this property because it maintains an inductive invariant. Do you know what that invariant is? If not, then you do not completely understand why the algorithm satisfies this property. How can a computer engineer design a correct concurrent system without understanding it? And how can she understand it if she has not learned how to understand even a simple concurrent algorithm?

To describe a set of computations, we must describe its initial states and its next-state relation. How do we describe them? Ultimately, a description must be written in some language. An art historian must decide whether to describe impressionism in French or another language. I used some simple programming notation

and English to describe the algorithm in the preceding paragraph. However, programming notation obscures the underlying concepts, and English by itself is unsatisfactory. There is a simple language that is good for describing states and relations. It's the language used in just about every other science: mathematics. But what mathematics? Everyone who works on formalizing computation says that they use mathematics. Process algebra is algebra, which is certainly math. Category theory and temporal logic are also math. These esoteric forms of math have their place, but that place is not in the basic education of a computer engineer. The only mathematics we need to describe computations are sets, functions, and simple predicate logic.

For historical reasons, mathematicians use and teach mathematics in ways that are not well suited for computer science. For example, while it is crucial for computer engineers, an understanding of simple logic is not important for most mathematicians. Consequently, although it has been centuries since mathematicians wrote algrebraic formulas in words, they still usually obscure simple logical concepts like quantification by expressing them in prose. This is perhaps why many computer scientists feel that the standard method of formalizing ordinary mathematics used by logicians for almost a century, consisting of predicate (first-order) logic and elementary set theory, is inadequate for computer science. This is simply not true. Other, more complicated logical systems that introduce concepts such as types may be useful for some applications. However, there is no more need to use them in basic compute science education than there is to use them in teaching calculus (or arithmetic). Computer science should be based on the same standard mathematics as the rest of science.

Another problem with the way mathematicians use mathematics is its informality. Informal mathematics is fine for explaining concepts like states and relations, and for explaining invariants of simple algorithms like the one in the example above. However, a defining characteristic of computing is the need for rigor. Incorrectly writing $>$ instead of $\geq$ in the statement of a theorem is considered in mathematics to be a trivial mistake. In an algorithm, it could be a serious error. Almost no mathematicians know how to do mathematics with the degree of formal rigor needed to avoid such mistakes. They may study formal logic; they don't use it. Most think it impractical to do mathematics completely rigorously. I have often asked mathematicians and computer scientists the following question: How long would a purely formal definition of the Riemann integral (the definite integral of elementary calculus) be, assuming only the arithemtical operators on the real numbers and simple math? The answer usually ranges from 50 lines to 50 pages.

TLA$^+$ is one of those esoteric languages based on temporal logic [5]. However, if one ignores the TL (the temporal logic operators), one obtains a language for ordinary math I will call here A-Plus. For example, here is an A-Plus definition of the operator $GCD$ such that $GCD(m, n)$ equals the greatest common divisor of positive integers $m$ and $n$.

$$
\begin{array}{lll}
GCD(m, n) \triangleq & \text{LET} & DivisorsOf(p) \triangleq \{d \in 1 \,..\, p : \\
 & & \qquad\qquad \exists\, q \in 1 \,..\, p : p = d * q\} \\
 & & MaxElementOf(S) \triangleq \text{CHOOSE } s \in S : \\
 & & \qquad\qquad \forall\, t \in S : s \geq t \\
 & \text{IN} & MaxElementOf(DivisorsOf(m) \cap DivisorsOf(n))
\end{array}
$$

A-Plus is obtained by deleting from TLA$^+$ all non-constant operators except the prime operator ($'$) that is used to express next-state relations—for example, $x' = 1 + x$ is a relation that is true iff the value of $x$ in the next state equals 1 plus its value in the current state. A-Plus is a practical language for writing formal mathematics. An A-Plus definition of the Riemann integral takes about a dozen lines. One can use A-Plus to describe computations, and those descriptions can be executed by the TLC model checker to find and eliminate errors. Using A-Plus is a lot like programming, except it teaches users about math rather than about a particular set of programming language constructs.

A-Plus is not the best of all possible languages for mathematics. It has its idiosynchracies, and some people will prefer different ways of formalizing elementary mathematics—for example, Hilbert-Bernays rather than Zermelo-Fraenkel set theory. Any language for ordinary first-order logic and set theory will be fine, as long as it eschews complicated computer-science concepts like types and objects. However, it should have tools for checking descriptions of computations. Despite the inherent simplicity of mathematics, it's almost as hard to write error-free mathematical descriptions of computations as it is to write error-free programs.

Mathematics is an extremely powerful language, and it's a practical method for describing many classes of computing devices. However, programming language constructs such as assignment were developed for a reason. Although more complicated than ordinary mathematics, a simple language based on traditional programming constructs can be convenient for describing certain kinds of algorithms. Such a language is a useful tool rather than a barrier to understanding only if we realize that it is just a shorthand for writing mathematical descriptions of computations. The language must be simple enough that the user can translate an algorithm written in the language to the equivalent mathematical description. There are several rather simple languages that have this property—for example, the Unity programming language. The most powerful such language I know of is PlusCal [4]. A PlusCal algorithm is essentially translated to an A-Plus description, which can be checked using the $\text{TLA}^+$ tools. Because any A-Plus expression can be used as an expression in a PlusCal algorithm, PlusCal's expression language has the full power of ordinary math, making PlusCal enormously expressive.

Students with an understanding of computations and how to express them mathematically and of invariance are ready to learn about concurrency. What they should be taught depends on their needs. If they need to learn how to write real concurrent programs, they will need to learn a real programming language. Programming languages are much more complicated than mathematics, and it is impractical to try to explain them precisely and completely. However, their basic features can be understood with the aid of mathematics. The difference between $=$ and $.equals$ in an object-oriented programming language is easily explained in terms of sets and functions.

Teaching concurrency generally includes teaching about concurrent algorithms. Distributed algorithms are usually easy to describe directly in mathematics, using a language like A-Plus. Multithreaded algorithms are usually more convenient to describe with the aid of programming constructs such as the ones in PlusCal. In any case, an algorithm should be explained in terms of its invariant. When you use invariance, you are teaching not just an algorithm, but how to think about algorithms.

Even more important than an algorithm is the precise specification of what the algorithm is supposed to do. A specification is a set of computations—namely, the set of all computations that satisfy the specification. An engineer can find algorithms in a textbook. A textbook won't explain how to figure out what problem a system needs to solve. That requires practice. Carefully specifying a problem before presenting an algorithm that solves it can teach an engineer how to understand and describe what a system is supposed to do.

Education is not the accumulation of facts. It matters little what a student knows after taking a course. What matters is what the student is able to do after taking the course. I've seldom met engineers who were hampered by not knowing facts about concurrency. I've met quite a few who lacked the basic skills they needed to think clearly about what they were doing.

## References

[1] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[2] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[4] Leslie Lamport. The pluscal algorithm language. URL http://research.microsoft.com/users/lamport/tla/pluscal.html. The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from `uid-lamportpluscalhomepage`.

[5] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.