Teaching Java Concurrency to CS vs IT Students: A Matter of Emphasis

Charles E. Hughes
School of Electrical Eng. and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, Florida 32816-2362, USA

Marc L. Smith
Computer Science Department
Colby College
5853 Mayflower Hill
Waterville, Maine, 04901-8858, USA

Abstract*— A number of colleges and universities have recently added new degree programs in Information Technology (IT), or added IT components to existing Computer Science (CS) programs. Java language and technology are almost inescapable elements of both CS and IT programs. One of Java's more advanced features, language-level support for concurrency in the form of explicit multithreading, is important to both CS and IT students, but for different reasons. Teaching Java concurrency to CS and IT students, therefore, presents different challenges and requires emphasizing the topic in different ways. We discuss these issues, and present our experiences from CS and IT classes taught recently, in which Java concurrency was a topic.

***Index Terms*— concurrency, multithreading, Java, information technology curriculum, computer science curriculum.**

1 INTRODUCTION

For the last almost four decades, Computer Science programs have addressed issues of concurrency within their core curricula. In the early days of our discipline, concurrency was mostly presented in the context of the solutions provided in operating systems produced by CDC, Digital Equipment, IBM and Xerox. With the publication of Dijkstra's seminal article [1] in 1968, concurrency became a proper academic topic, rather than a case study. However, even with the rapid introduction of science to this complex topic, e.g., in Hoare's influential paper on monitors [2], concurrency remained an issue generally avoided by programmers, as no support existed in most programming languages. A number of early exceptions included Concurrent Pascal [3] and Modula [4].

The first author's work was supported in part by the National Science Foundation under grant EIA 9986051

During the 1970's and 1980's, the CS discipline treated concurrency in the context of operating systems and then databases, and that's where it mostly resided in the computer science curriculum until the 1990's. Of course, there were early attempts to address the topic as a language add-on, e.g., in the base classes of Smalltalk80 [5] and in the language independent library calls provided by C-Linda [6].

The late 1980's and the 1990's saw rapid changes in our understanding of concurrency and the often related topic of communication. Some of this understanding came about in the brief era of monolithic parallel computers, but mostly it is the current era of pervasive distributed computing that has made concurrency a mainstream topic in computer science and in the evolving discipline of information technology. Our goal in this paper is to discuss how the emphases in these disciplines differ and how this affects what we teach students in each. Our focus here is on the Java language, because of its built-in support for primitive concurrency and its rich set of APIs that support high-level distributed programming paradigms

2 DIFFERENTIATING NEEDS

A distributed computation course for computer scientists typically focuses on science (e.g., mathematical reasoning, algorithm design and analysis), programming paradigms (e.g., message passing, remote methods, middleware mediated) and system architectures (e.g., dedicated, shared). Computer science students study formal means of reasoning about complex systems, which results in their studying the early works of the masters (Dijkstra and Hoare) and subsequent formalisms for such reasoning, e.g., [7, 8, 9], as well as great ideas revisited and reinvigorated, e.g., [10, 11, 12]. One of the authors also adds in meta-theorems, such as the proof that "Oblivious

Comparison Exchange" sorts need only be shown correct for 0-1 data. This is an especially interesting proof for computer scientists, as it simultaneously can be used to establish an order bound on the time complexity of an algorithm in the process of its being shown correct. An appropriate text is Andrews [13].

In contrast, we typically teach information technology students about productivity, security and integration with the rest of an enterprise. These students must understand how applications interact in the enterprise setting. The focus is on assembling components in novel ways, as opposed to building individual components and assembly tools, tasks most commonly performed by computer scientists. While basic concepts such as semaphores and monitors are covered, formal proofs are rarely appropriate. An appropriate text is Deitel et al. [14]

3 CS APPROACH

3.1 Concurrency in General

Concurrency, whether multithreaded, parallel or distributed, requires communication and coordination between processes to ensure the integrity of shared resources (e.g., memory, devices). Synchronization is a special case of coordination.

One of the first topics we discuss is the meaning and differentiation of the "Safety" and "Liveness" properties. The former says that nothing bad ever happens (e.g., mutual exclusion); the latter says that something good eventually happens (e.g., termination). Using these two simple concepts as a basis, we can categorize other common properties such as partial correctness, total correctness and eventual entry to a critical section.

We then introduce students to the notions of atomic actions and levels of atomicity. We present the need for hardware/software support and the possibilities of different levels of abstraction. For example, at the instruction level, we present Test-and-Set and Fetch-and-Add instructions, plus the approaches of busy waiting and spin locks.

With these basic topics in hand, we then introduce our students to the non-determinism inherent in concurrency. We present the notion of thread interleaving, program histories, and mechanisms to limit the number of possible interleavings so that only desirable outcomes remain possible.

We then take the students from the abstract discussions above to specific concurrency control mechanisms starting with semaphores. We show how to implement semaphores with low-level instructions. We then use semaphores to implement critical sections, addressing the problem with achieving fairness (unconditional, weak and strong) and its dependence on process scheduling algorithms. This is then extended to the broader issues associated with critical sections, namely multual exclusion, absence of deadlock and livelock, absence of unnecessary delays and the need for eventual entry (a fairness criterion).

The next higher level of abstraction we introduce is monitors, a form of implicit critical section. Fairness is discussed in the context of condition variables and the algorithms for managing waiting queues (FIFO versus priority and Signal & Continue versus Signal & Wait semantics).

As with all other mechanisms, we simulate one with the other (here semaphores with monitors) to gain a better understanding of each. We also demonstrate a number of strategies such as passing the baton.

3.2 Concurrency in Java

Java and the Java Virtual Machine (JVM) support multiple threads of execution, synchronized methods and code blocks (thanks to every object providing an implicit lock), and signal/notify/notifyAll operations. Essentially, Java provides a native but somewhat limited monitor implementation. An interesting (and somewhat distressing) point is that Java's celebrated cross-platform portability and WORM (write once, run many) features do not extend to multithreaded programs, in the sense that runtime program behavior can differ from platform to platform.

Another surprising characteristic of instruction atomicity with the JVM concerns variables of type long and double. Both data types are stored in two 32-bit words. Low-level load and store operations are not required to be atomic when accessing longs and doubles. This means context

switching could occur half-way between loading or storing these variable types. To be safe, one needs to synchronize all such accesses.

Beyond these subtle, and not terribly well-documented features of Java, one must be cognizant of the consequences of the rapid evolution of the language and its JVM implementations. In fall 2001, one of us (MLS) demonstrated a multithreaded Java solution to the Dining Philosophers Problem. It had unsafe and safe methods to pick up forks. It was easy to demonstrate the deadlock. This semester (Spring 2003), running the same program, we could not get the threads to deadlock. Now, in this instance, the result may be viewed an improvement in scheduling algorithm, but notice it's just an improvement, not a change in required semantics. Theoretically, the deadlock is still possible, but much more difficult to test for and detect. Using this as an empirical example, the author was able to stress the importance of sound design principles and the need for formal reasoning. Of course, a possible consequence of a program's change in behavior over time is the possibility of undesirable properties emerging. It is as dangerous for programs to rely on particular implementations of the JVM as it was for FORTRAN programmers in the old days of computing to rely on the values of DO-loop variables after loop termination (they were undefined in the language semantics, had predictable values on most individual compilers, but had varying values across different compilers).

Beyond the basic concurrency support in Java, we look carefully at important APIs that implement theoretically sound paradigms for communication and coordination. In particular, we have chosen to study several of Java's implementation of the tuple space paradigm, along with an implementation of CSP.

The concept of tuple spaces is introduced as an example of generative communication, a paradigm distinct from shared memory and message passing. The specific implementations studied include (but are not limited to) Sun's JavaSpaces and IBM's TSpaces. With these implementations, we now have the possibility of implicit and scalable communication across JVMs and physical processors.

Our final high-level Java API is JCSP, an implementation of Hoare's CSP [7, 11]. Different from other examples of concurrency seen so far, JCSP programs are by default deterministic. Communication between processes is via channels. Composition of new CSProcesses from existing CSProcesses is elegant, as one would expect in the implementation of an elegant theoretical framework. JCSP obtains its primary advantages over other approaches due to its strong mathematical foundation for reasoning about safety and liveness properties. Alternation (nondeterminism) is supported via three operations: select(), priSelect(), and fairSelect().

Both tuple spaces and JCSP have commercial implementations. In the case of tuple spaces, the GigaSpaces Platform provides a solution that its authors claim supports large scale Enterprise applications. JCSP is commercially available in a Network Edition that permits scalable distributed computing.

3.3 Distributed Programming Paradigms

We introduce our students to three distributed computing paradigms, channels (as exemplified by TCP/IP and JCSP), distributed objects (as exemplified by Java's RMI and OMG's CORBA) and middleware-mediated (as exemplified by Jini/JavaSpaces). Typically, this stage of the course includes a few small programming assignments and one substantial project. The project can vary from an on-line bidding system, implemented in either RMI or some space-based middleware to the design and implementation of a multiplayer MUD, with simple agents playing the roles of some of the players.

4 IT APPROACH

4.1 Concurrency – an experienced-based approach

Although all students who enter our IT course on distributed computing, titled "Distributed Applications in the Enterprise," take prerequisite classes in operating systems and object-oriented programming with Java, we have found that they, like computer science students, have a limited understanding of the many consequences of concurrency. Thus, we start the course off with a

discussion of threads and processes, focusing on Java threads.

To emphasize the importance of writing thread-safe code, we give them an assignment concerning a sample program that has many threads. The program uses no synchronization, has serious race conditions, yet typically works when run on a single processor machine. The purpose here, as with the Dining Philosopher example shown in the CS course, is to give our students experience with how hard it can be to debug a multi-threaded program, and thus why getting it right is better than spending the rest of your career fixing subtle errors. In the case of CS students, this is an opportunity to talk about the benefits of formal reasoning. In the case of the IT students, it's a chance to talk about the advantages of assembling reliable components over re-inventing the wheel (and creating a square one at that).

As an example, in fall 2002 we provided our IT class with a Java program that sorts by assigning a separate thread to each interior element of an array, value[0..n-1]. The i^{th} thread ($0 < i < n-1$) in essence carries out the following algorithm:

```
forever {
        sleep for a random period of time
        up to a fifth of a second;
        if value[i] < value[i-1]
                swap (i, i-1);
        if value[i] > value[i+1]
                swap (i, i+1);
}
where swap is defined as
swap (i, j) {
        temp = value[j];
        value[j] = value[i];
        value[i] = temp;
}
```

Clearly, this has problems, e.g., if a thread loses control when it is in the middle of a swap. However, if the JVM is a single process running on a single processor, and if all threads give up control only when they sleep, each iteration of the loop acts like an atomic action, and there are no perceived race conditions.

The task we assign to our students is to find all the potential problems in this rather simple multi-threaded example, add thread yields to expose those problems, and demonstrate the results.

There are actually two very distinct types of problems here. First, losing control in the middle of the swap can cause the array to become corrupted with multiple copies of a single value replacing others. Second, losing control right after a decision is made to swap, but before swapping, can cause the sort to make wrong swaps. In that case, we may keep permuting values forever (although that is unlikely). From a theoretical point of view, the first case violates a safety condition and results in the program entering an invalid state; the second case violates a liveness condition and results in a failure of the program to converge to the correct result.

As you can see, while this is not a deep formal approach, we do expose our students to some of the science and programming paradigms that are the core of the computer science aspects.

4.2 Building a Distributed System

Enterprise systems typically include services provided by simple client-server as well as more complex multi-tiered and (in some cases) peer-to-peer systems. Our primary emphasis here is on multi-tiered systems. To present these, we introduce Servlets, and the notion of a container that provides efficient and secure execution of a Servlet's services, and the enterprise-level management of the resources shared by multiple services. We extend this notion to include Java Server Pages (JSP), typically using Apache Tomcat as the middle tiers container. However, some of our students choose to use much more functional systems such as IBM WebSphere, Borland Enterprise Server or JBoss.

Designing and building a three tiered system (web client, application server, backend database) gets our IT students familiar with the many issues involved and the concurrency challenges that arise when multiple tier 1 clients simultaneously access the tier 2 application server which in turn generates concurrent demands on the tier 3 database. The problem becomes even more interesting when the middle tier caches database entries, strategy required for efficiency and responsiveness.

Further details on this course and the IT program in general, as implemented at the University of Central Florida, can be found at [15]

4.3 Distributed Programming Paradigms

As with the CS course, we introduce our IT students to three distributed computing paradigms, channels (as exemplified by TCP/IP; we don't do JCSP here), distributed objects (as exemplified by Java's RMI and OMG's CORBA) and middleware-mediated (as exemplified by Jini/JavaSpaces). This part of the course may involve programming that is a bit more technical than IT majors typically encounter, but we feel that at least a conceptual knowledge is critical to their success. TCP/IP and its underlying UDP protocol are presented through simple examples, e.g., time and multiplayer game services. RMI and CORBA are demonstrated by a bidding service. JavaSpaces is demonstrated by investigating its use as middleware for tier 0 devices (appliances), as well as serving as the communication/ coordination layer of a shared virtual world.

In any given semester we may vary the choice of RMI, CORBA, or space-based middleware as the basis of a programming assignment. In the 2001-2002 semesters, the choice was CORBA. In fall 2002 we required students to make some small additions to an RMI bid system, adding new services. One added lesson that we hope they take from this is that one can add services to a system without disenfranchising those running older clients (ah, the beauty of interfaces).

5 CONCLUSIONS

We have presented the concurrency components of two courses: one for CS majors, the other for IT majors. Versions of the CS course were taught at the University of Central Florida (UCF) and at Colby College, the home institutions of the authors. The IT course was taught at UCF only, as Colby does not have an IT major.

This paper is not meant to lay down a template that all others should follow. Rather, our goal is to initiate a productive discussion about what are the common, core skills and concepts for both clientele (CS and IT), and what are the topics or emphases that differentiate these two groups of students.

6 REFERENCES

[1] Dijkstra, E. W. (1968) "Cooperating Sequential Processes,"in F. Genuys, ed., *Programming Languages*, Academic Press, New York, pp. 43-112.

[2] Hoare, C. A. R. (1974) "Monitors: An Operating System Structuring Concept," *Communications of the ACM* 17(10), October 1974, pp. 549-557.

[3] Brinch Hansen, P. (1975) "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* 1(2), June 1975, pp. 199-206.

[4] Wirth, N. (1977) "Modula: A Language for Modular Programming," *Software Practice and Experience* 7, pp. 3-35.

[5] Goldberg, A. J. and Robson, D. (1983) *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA.

[6] Gelernter, D. (1985) "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems* 7(1), January 1985, pp 80-112.

[7] Hoare, C. (1985). *Communicating Sequential Processes*, Prentice-Hall International, Ltd., UK.

[8] Agha, G. A. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts.

[9] Schneider, S. (1999) *Concurrent and Real-Time Systems: The CSP Approach*, John Wiley & Sons, New York.

[10] Carriero, N. and Gelernter, D. (2001) A Computational Model of Everything. *Communications of the ACM* 44(11), November 2001, pp. 77-81.

[11] Welch, P. H. and Vinter, B. (2002) in *Concurrent Systems Engineering Series* (Pascoe, Welch, Loader and Sunderam, Eds.) IOS Press, Amsterdam, pp. 203-222.

[12] Smith, M. L., Parsons, R. J. and Hughes, C. E. (2003) "View-centric Reasoning for Linda and Tuple Space Computation," *IEE Proceedings-Software* 150(2), April 2003.

[13] Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley.

[14] Deitel, H., Deitel. P. and Santry S. (2002) *Advanced Java™ 2 Platform*, Prentice-Hall.

[15] Hughes, C. E. and Marin, G. (2003). "A New Program in Information Technology," *International Conference on Information Technology (ITCC 2003)*, Las Vegas, April 28-30, 2003.