Concurrency in Practice

Threaded vs. Event-based – is that a even valid question?

Robert Staudinger
rstaudinger@cs.uni-salzburg.at
Summer 2007 Software Systems Class
Department of Computer Sciences
University of Salzburg, Austria

July 13, 2007

Abstract

This survey looks at a handful influential papers and approaches regarding concurrent programming. Threaded programming is still subject to heated discussions, it even received some high-profile criticism, as elaborated in Section 2. Rather than refuting, some multi-threading advocates argue that offhand dismissal would be premature. This has in turn led to the question whether a confrontation of both paradigms is valid at all – which has to be answered with "not really". Section 5 finally provides some details about Duff's Device, a well known way for portable implementation of cooperative threads.

1 Introduction

Run-time environments such as Java have made threaded programming look very easy by putting synchronisation primitives into the language itself. Also there seems to be a fixed idea among many programmers that multi-threaded applications would perform better, which is probably to blame on shallow literature. On the other hand advocates of event-driven programming are dismissing threads offhand, in turn citing well known difficulties such as race-conditions and timing non-determinism. This survey takes a look at a number of influential papers on the topic of threaded and event-driven programming and tries to find out whether an antagonism between both approaches really exists.

2 High-profile criticism of multi-threading

The classical multi-threading programming paradigm has been subject to much critique [10, 11], which in turn called defenders to join the discussion [13].

One of the early strong critiques was Jon Ousterhout's 1996 USENIX talk *Why Threads are a Bad Idea (for most purposes)* [11], in which he mentions a long list of unfavourable properties. His comments center around the claims that threads are hard to program because of difficulties to get synchronisation right, leading to race conditions and deadlocks. More items of critique are that threads are hard to debug and also break abstraction between modules. Ousterhout advocates an event-driven programming style instead, with a single logical (and also "physical" in terms of operating system) thread of execution. Naturally the simplicity of event-handlers comes at the price of being a less general abstraction. More complex handlers would block the event dispatcher for an unacceptably long time and thus have to be split up into parts – potentially obfuscating the code. He closes with an evaluation of respective advantages, including easier portability[1] of event-driven programs and advantages with regard to high performance requirements of threads.

In *The Problem with Threads*[10] UC Berkeley's Edward A. Lee argues in a similar direction. He points out that while threaded programming seems to be only a small step from sequential programming, it is actually much more. By discarding the basic properties of sequential computation, which are *understandability*, *predictability* and *determinism*, a threaded program becomes wildly non-deterministic. Lee recommends more aggressive pruning of threads to achieve determinism and cites careful lock acquisition and application of understood design patterns [9] as remedies. However, even when applying these techniques during software design and development, satisfactory results are hard to achieve. He continues to point out a number of programming language dialects that add support for multi-threading over their ancestors. Examples are the C dialects Split-C [7] and Cilk [3] as well as Guava [2], a Java derivative. As deterministic alternative to threads, the rendezvous approach of Ptolemy II [5], is presented and discussed in depth. He closes by pointing to a future of coordination languages [12] that are orthogonal to the actual thread functionality implementations, but admits that this is nothing new, rather just failed to take on until now.

[1] This particular concern has been alleviated by run-time environments such as Java(tm) in the meantime

3 Can issues be avoided?

Contrary to the above reservation with regard to threads Rob von Behren et al argue favourably in *Why Events are a Bad Idea (for high-concurrency servers)* [13]. However, they clearly point out that assertions made in their paper do not conflict with [11]. Their strongest claim is that desireable properties of an event-based programming style, such as low synchronisation and state management overhead, as well as better scheduling and locality, can also be achieved in a threaded system, if compiler support for threads is available. Revisiting the Lauer and Needham prove of equivalence of message-passing and process-based systems [8] they conclude that assumptions regarding cooperative scheduling and shared memory no longer hold. Thus nowadays' implementations are invalidating the strict theoretical concept of duality to some extent. For their benchmarks they used an optimised version of the GNU Pth threading package[2], removing most of the $O(n)$ operations from the scheduler. A particularly interesting point is made in the chapter about compiler support for threads, mentioning a C-derived programming language called nesC [6] that supports atomic sections. Finally an important issue is that this paper based on userland threading and coroutine techniques, which in terms of tunability are way closer to events than opaque kernel-level threading.

4 Is the "vs" valid at all?

Another interesting paper building bridges between the traditionally disjoint fields of event-driven and threaded programming approaches is *Cooperative Task Management without Manual Stack Management* [1] by Atul Adya et al. The subtitle *Event-driven Programming is Not the Opposite of Threaded Programming* makes the point pretty well and tries to clarify the underlying mostly orthogonal issues of (i) task management, (ii) stack management, (iii) I/O management, (iv) conflict management and (v) data partitioning, of which stack management is discussed in more depth. The term *stack ripping* is coined for manual stack management in languages that do not offer native support for closures. What is not made as clear are the details and implications of atomic (non-preemptible) code blocks in the context of a manually managed stack. Clearly the focus is on the hybrid approach presented using an elaborate example using *Fibers*[3], a collaborative threading API available in the WindowsNT family of operating systems. Contrary to papers discussed above, benchmarks are not provided. The focus is on

[2] http://www.gnu.org/software/pth/

[3] http://msdn2.microsoft.com/en-us/library/ms682661.aspx

software design and -programming techniques.

5 Poor man's threading

Finally, a C programming technique with roots in execution speed optimisation, called *Duff's Device*[4], is used in a number of collaborative userland threading implementations, like for example *Coroutines in C*[5]. The most elaborate and widely used incarnation of which is probably Adam Dunkels' protothreads library [4]. Duff's Device relies on the fact that `switch` statements in C do *fall through*, unless a `break` statement is inserted. The venerable Lysator site[6] also has some auxiliary historical information. What is particularly interesting is that Duff's Device offers a portable (compiler independent) way of collaborative concurrency for C, thus facilitating static analisis. In order to exploit that feature for the implementation of a userland threading package the necessary bookkeeping code is usually wrapped into preprocessor defines, to make the resulting code at least somewhat readable. Unfortunately the destruction of non-static local variables on the stack at each *yield* requires manual stack management or abstinence from using them across yield-points, which might not always be obvious.

6 Conclusion

The wildly varying presumptions and statements in the discussed papers have showed that *Threaded vs. Event-based* is indeed the wrong question. Instead, as detailed in Section 4, it is primarily a question of stack management and task management. The wide range of available solutions ranging from language-level primitives over library support to custom tailored, application specific concurrency support should be able satisfy most demands. This is not to say that there is no more room for innovation left. In terms of implementation the square peg of concurrency has not yet been fit into the round hole of determinism, at least not as a programming model that is comparably straight forward as threads in terms of being obvious and easy to understand.

[4] http://en.wikipedia.org/wiki/Duffs_device
[5] http://www.chiark.greenend.org.uk/ sgtatham/coroutines.html
[6] http://www.lysator.liu.se/c/duffs-device.html

References

[1] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management, 2002.

[2] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. *ACM SIGPLAN Notices*, 35(10):382–400, 2000.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[4] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[5] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendor, e Sonia, and S. Yuhong. Taming heterogeneity—the ptolemy approach, 2002.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems, 2003.

[7] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM Press.

[8] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[9] Douglas Lea and Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[10] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[11] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). In *USENIX Winter Technical Conference*, San Diego, CA, January 1996. USENIX.

[12] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *761*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1998.

[13] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea for high-concurrency servers, 2003.