Why Events Are A Bad Idea
(for high-concurrency servers)

Rob von Behren, Jeremy Condit and Eric Brewer

Presented by: Hisham Benotman
CS533 - Concepts of Operating Systems
Portland State University
Professor Jonathan Walpole

January 20, 2010
We are still in the debate Threads vs. Events

In this presentation, we support threads
Contents

- The three opinions
- Threads’ problems and their fixes
- Threads’ advantages
- Compiler support for threads
- Evaluation
Opinion1: Events Are Better

- For high performance concurrent applications
  - Event-based programming is the best

- Reasons
  - Inexpensive synchronization (cooperative multitasking)
  - Lower overhead for managing state (no stacks)
  - Better scheduling and locality (application-level information)
  - More flexible control flow (not just call/return)
Opinion 2: Threads Are Better

- Properties in the previous slide are not restricted to event systems
  - Many are already implemented with threads
  - The rest are possible
- Threads are more natural for high concurrency
- Compiler improvements can eliminate historical drawbacks
Opinion 3: Both Are Equal

- Lauer & Needham in 1979
- Message passing and process based systems are duals (Program structure, Performance)
- The choice depends on nature of target app.
- Both systems yields same performance if implemented properly
Opinion3: Both Are Equal

- However
  - Cooperative scheduling is used by most modern event based systems
  - Event systems use shared memory (atypical)
  - Only one exception (SEDA)
What Are Threads' Problems

- Performance
- Control Flow
- Synchronization
- State Management
- Scheduling
Performance

- For high concurrency, threads have not performed well
- Problem is an artifact of poor thread implementations
- Operations with $O(n)$ in the number of threads
- higher context switch overhead compared with events because of Preemption and kernel crossing
Performance fix

- Shortcomings are not intrinsic properties of threads
- We modified version of the GNU Pth user-level threading package
- Remove most of the O(n) operations and Compare it to SEDA
- GNU Pth matches the performance of SEDA (Scales to 100,000 threads)
Performance fix

The graph compares the performance of a Threaded Server and an Event-Based Server under different numbers of concurrent tasks. The Threaded Server shows a steady increase in requests per second, whereas the Event-Based Server experiences a peak and then a drop in performance as the number of concurrent tasks increases.
Control Flow

- Threads have restrictive control flow.
- Threads encourage the programmer to think too linearly about control flow.
- Precludes the use of more efficient control flow patterns.
Control Flow reply

- Complicated control flow patterns are rare in practice
- Event systems (SEDA, Ninja, TinyOS) have limited control flow patterns
  - Call/return, parallel calls, pipelines
  - Can be expressed more naturally with threads
- Complex patterns are difficult to use well
  - Accidental nonlinearities in event systems are hard to understand, leading to subtle races
Synchronization

- Thread synchronization mechanisms are too heavyweight
- Cooperative multitasking (i.e., no preemption) in event systems gives synchronization “for free”
  - No mutexes, handling wait queues, ...
Synchronization reply

- Free synchronization is due to cooperative multitasking not events themselves
- Cooperative thread systems can reap the same benefits
- Cooperative multitasking only provides “free” synchronization on uniprocessors
State Management

- Thread stacks are an ineffective way to manage live state.
- Threaded systems face tradeoff between stack overflow and large stacks.
- Event systems use few threads and unwind the thread stack after each event handler.
- Automatic state management allows programmers to be wasteful.
State Management fix

- Dynamic stack growth
- Will be discussed in the next slides
Scheduling

- Virtual processor model forces the runtime system to be too generic and prevents it from making optimal scheduling decisions.
- Event systems are capable of scheduling event deliveries at application level.
  - i.e. favor certain request streams
- Events allow better code locality
  - running several of same kind events in a row
Lauer-Needham duality indicates we can apply the same scheduling tricks to cooperatively scheduled threads.
Threads Are even better

More appropriate abstraction for high concurrency servers
Control Flow

- Event systems obfuscate the control flow of the application
  - “call” with an event, “return” with another event
  - Programmer must mentally match these call/return pairs
  - Often requires the programmer to manually save and restore live state
  - Can lead to subtle race conditions and logic errors
Control Flow contd.

- Thread systems are more natural
  - group calls with returns
  - much easier to understand cause/effect relationships
- the run-time call stack encapsulates all live state for a task
  - Makes debugging tools quite effective
Exception Handling and State Lifetime

- In event systems, task state is typically heap allocated
- In exceptions and normal termination, freeing this state can be extremely difficult (branches in the application’s control flow)
  - Deallocation steps can be missed
- Many event systems use garbage collection
- But Java’s general-purpose garbage collection is inappropriate for high performance systems
Exception Handling and State Lifetime Contd.

- This task is simpler in a threaded system
- Thread stack naturally tracks the live state for each task
Why don’t we just fix events?

- Build tools or languages that address the problems with event systems
  - (i.e., reply matching, live state management)
- Such tools would duplicate the syntax and run-time behavior of threads
- Fixing the problems with events is tantamount to switching to threads
Compiler Support for Threads

- Tighter integration between compilers and runtime systems is an extremely powerful concept for systems design.
- With only minor modifications to existing compilers and runtime systems, threaded systems can achieve improved safety and performance.
Dynamic Stack Growth

- Mechanism that allows the size of the stack to be adjusted at run time
- Avoids the tradeoff between potential overflow and wasted space in fixed-size stacks

- Provide an upper bound on the amount of stack space needed when calling each function
- Determine which calls may require stack growth
Live State Management

- Compilers could easily purge unnecessary state from the stack before making function calls
  - i.e. temporary variables

- Compiler could warn the programmer when large amounts of state might be held across a blocking call
  - allowing the programmer to modify the algorithms
Synchronization

- Static detection of race conditions is challenging
- However, there has been recent progress due to compiler improvements
  - i.e. nesC supports atomic sections and compiler understands the concurrency model
  - calls within an atomic section cannot yield or block
Evaluation

- Implementation of a simple cooperative threading package for Linux.
- Knot, a 700-line test web server
- Compare Knot and Haboob (SEDA’s event-driven web server)
- Two different scheduling policies for Knot
- One favors processing of active connections over accepting new ones, One does the reverse
Result
Conclusion

- Thread systems can achieve similar or even higher performance compared to event systems
- Current threads weaknesses are due to current implementations
- Better compiler analysis gives threads more advantages
- Compiler support for threads is a fruitful research area
Questions

Why Events Are A Bad Idea (for high-concurrency servers)
References

- This Presentation is based on the paper
  Why Events Are A Bad Idea
  (for high-concurrency servers)
  Final Version, Proceedings of HotOS IX
  Lihue, Kauai, Hawaii, May 2003
  Rob von Behren, Jeremy Condit and Eric Brewer
  Computer Science Division, University of California at Berkeley
  {jrvb, jcondit, brewer}@cs.berkeley.edu
  http://capriccio.cs.berkeley.edu/

- January 26, 2009 presentation of this paper for CS533 at PSU by Ryan Ledbetter.