Event-driven Programming for Robust Software

Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, Robert Morris
MIT Laboratory for Computer Science
{fdabek,kolya,kaashoek,rtm}@lcs.mit.edu, dm@scs.cs.nyu.edu

Abstract

Events are a better means of managing I/O concurrency in server software than threads: events help avoid bugs caused by the unnecessary CPU concurrency introduced by threads. Event-based programs also tend to have more stable performance under heavy load than threaded programs. We argue that our libasync non-blocking I/O library makes event-based programming convenient and evaluate extensions to the library that allow event-based programs to take advantage of multi-processors. We conclude that events provide all the benefits of threads, with substantially less complexity; the result is more robust software.

1 Introduction

The debate over whether threads or events are best suited to systems software has been raging for decades. The central question of this debate is whether threads or events should be used to manage concurrent I/O. Many prefer threads because they preserve the appearance of serial programming and can take advantage of multi-processor hardware. Programmers find programming with threads difficult, however, and as a result produce buggy software. This paper contributes to the debate by showing that event-based programming can provide a convenient programming model, that it is naturally robust, and that it can also be extended to take advantage of multi-processors. For these reasons, we argue that there is no reason to use threads for managing concurrent I/O in system programs—events provide all the benefits that threads provide, but in a more robust way.

Thread-based programs use multiple threads of control within a single program in a single address space [3]. Threaded programs achieve I/O concurrency by suspending a thread blocked on I/O and resuming execution in a different thread. Under this model, the programmer must carefully protect shared data structures with locks and use condition variables to coordinate the execution of threads.

Event-based programs are organized around the processing of events. When a program cannot complete an operation immediately because it has to wait for an *event* (e.g., the arrival of a packet or the completion of a disk transfer), it registers a *callback*—a function that will be invoked when the event occurs. Event-based programs are typically driven by a loop that polls for events and executes the appropriate callback when the event occurs. A callback executes indivisibly until it hits a blocking operation, at which point it registers a new callback and then returns.

In this position paper, we show that writing programs in the event-based model can be convenient and show how to extend the event-based model to exploit multi-processors in a way that requires programmers to make only minor changes to their code. This multi-processor event model makes it easy to capture the levels of CPU concurrency typically available to systems programs, but does not require the programmer to insert locks or condition variables. The multi-processor support is part of the *libasync* asynchronous programming library. Unlike programming with threads, *libasync* doesn't force programmers to manage thread synchronization, doesn't force programmers to guess how much space to reserve for stacks, provides high performance under high load, and doesn't have hidden performance costs. As a result, programs written with *libasync* tend to be robust. Based on our experience with the library, we believe there is no reason to use threads for managing concurrent I/O.

2 Threads lead to unreliable software

The main advantage of threads is that they allow the programmer to overlap I/O and computation while preserving the appearance of a serial programming model. The main disadvantage of threads is that they introduce concurrent execution even where it is not needed. This concurrency unnecessarily forces programmers to cope with synchronization between threads. In practice, threaded programs almost always have latent data races and deadlocks and, as a result, they are not robust. In our view, programmers must be spared the complexities of concurrency to make software more robust.

The problems with thread-based programming have long been recognized. Ousterhout argues that the convenience of threads is not worth the errors they encourage, except for multi-processor code [11]. Engler *et al.* demonstrate that synchronization errors, particularly potential deadlocks, are common in the Linux kernel [5]. Savage *et al.* [13] found races in both student code and production servers.

While it might seem that one could eliminate synchronization by moving to non-pre-emptive threads on a uniprocessor, errors will still arise if a thread blocks (yields the CPU) unexpectedly. In complex systems, it is easy for programmers of one module not to understand the exact blocking behavior of code in other modules. Thus, one can easily call a blocking function within a critical section without realizing that pre-emption may occur [1]. Deadlocks are also a possibility when using non-preemptive threads, as one still needs to hold locks across known yields. Adya *et al.* propose the use of non-preemptive threads augmented with runtime checks to detect unexpected yields in submodules [2] and describe a new taxonomy for I/O concurrency techniques. They also show how to integrate code written in the event-driven style (described in their taxonomy as cooperative task management and manual stack management) with non-preemptive threads (cooperative task management and automatic stack mangement) in the same program.

Another complication of threads is the need to predict the maximum size of a stack. Most systems conservatively use one or two pages of memory per thread stack—far more than the few hundred bytes typically required per callback. The designers of the Mach kernel found stack memory overhead so high that they restructured the kernel to use "continuations," essentially rewriting the kernel in an event-driven style [4]. Stack memory overhead is especially burdensome in embedded environments where memory is scarce. The standard practice of allocating full pages for thread stacks also causes additional TLB pressure and cache pressure, particularly with direct-mapped caches [4]. Cohort scheduling attempts to increase data and instruction locality in threaded programs by running related computations, organized as "stages," consecutively [7].

Robust software must gracefully handle overload conditions. Both Pai *et al.* [12] and Welsh *et al.* [14] explore the advantages of event-driven programs under high load. Welsh demonstrates that the throughput of a simple server using kernel-supported threads degrades rapidly as many concurrent threads are required. Pai extends the traditional event-driven architecture to overcome the lack of support for non-blocking disk I/O in most UNIX implementations by using helper processes to handle disk reads. On workloads involving many concurrent clients, Pai's event-based Web server provides higher performance than a server using kernel-based threads.

Writing programs in the event-driven style alleviates all of the problems mentioned above. Data races are not a concern since event-based programs use a single thread of execution. Event-driven programs need only allocate the memory required to hold a callback function pointer and its arguments, not a whole thread stack; this reduces overall memory usage. In addition, these pointers and arguments can be allocated roughly contiguously, reducing TLB pressure. The livelock-like behavior of threaded servers under high load is avoided by event driven programs that queue events that cannot be serviced rather than dedicating resources to them [10].

Lauer and Needham observe that programs based on message passing (which corresponds to the event-driven model) have a dual constructed in the procedure-oriented (threaded) model (and vice versa) [8]. Based on this observation, the authors conclude that neither model is inherently preferable. The model described by Lauer and Needham does not exploit the fact that coordination is considerably simpler when using non-preemptive scheduling. As a result the authors' conclusion neglects the advantages of the lower synchronization burden offered by the event-driven model.

While this paper focuses on user-level servers, the arguments apply to operating system kernels as well. In this realm, the event-driven architecture corresponds to a kernel driven by transient interrupts and traps. Ford *et al.* compare event-driven kernels with threaded kernels in the context of Fluke [6].

3 Making asynchronous programming easy

The most cited drawback of the event-driven model is programming difficulty. Threaded programs can be structured as a single flow of control, using standard linguistic constructs such as loops across blocking calls. Event-driven programs, in contrast, require a series of small callback functions, one for each blocking operation. Any stack-allocated variables disappear across callbacks. Thus, event-driven programs rely heavily on dynamic memory allocation and are more prone to memory errors in low-level languages such as C and C++. As an example, consider the following hypothetical asynchronous write function:

```
void awrite (int fd, char *buf, size_t size,
  void (*cb) (void *), void *arg);
```

`awrite` might return after arranging for the following to happen: as soon as the file descriptor becomes writeable, write the contents of `buf` to the descriptor, and then call `cb` with `arg`. `arg` is state to preserve across the callback—state that likely would be stack-allocated in a threaded program. A number of bugs arise with functions like `awrite`. For example, `awrite` probably assumes `buf` will remain intact until the callback, while a programmer might be tempted to use a stack-allocated buffer. Moreover, the cast of `arg` to void pointer and back is not type safe.

The C++ non-blocking I/O library in use by the authors, *libasync*, provides several features to eliminate such memory problems. It supplies a generic reference-counted garbage collector so as to free programmers from worrying about which function is responsible for deallocating what data.

libasync also provides a type-safe method of passing state between callbacks. A heavily overloaded template function, *wrap*, allows the programmer to pass data between callbacks with function currying: *wrap* takes as arguments a function or method pointer and one or more arguments and returns a function object accepting the original function's remaining arguments. Thus, the state of an operation can be bundled as arguments to successive callbacks; the arguments are type-checked at compile time.

Finally, the library also provides classes to help deal with the complications of short I/O operations (i.e., when kernel buffers fill up and a system call such as *writev* only writes part of the data). The suio class can hold onto a reference counted object until the "printed" data is consumed by an *output* call.

Experience with *libasync* shows that it is easy to learn and use. We use the library day-to-day when implementing network applications. Students have used the library to complete laboratory assignments including web proxies and cryptographic file systems.

4 Multi-processor event programming

We have modified the asynchronous programming library described in Section 3 to take advantage of multi-processors. The modified library (*libasync-mp*) provides a simple but effective model for running threads on multiple CPUs, but avoids most of the synchronization complexity of threaded programming models.

Programs written with *libasync-mp* take advantage of multi-processors with a simple concurrency mechanism: each callback is assigned a *color* by the programmer, and the system guarantees that no two callbacks with the same color will run concurrently. Because callbacks are assigned a default color, *libasync-mp* is backwards compatible with existing event-based applications. This allows the programmer to incrementally add parallelism to an application by focusing on just the event callbacks that are likely to benefit from parallel execution. In contrast, typical uses of threads involve parallelizing the entire computation, by (for example) creating one server thread per client; this means that all mutable non-thread-private data must be synchronized. The concurrency control model provided by *libasync-mp* also avoids deadlocks: a callback has only one color, so cycles can't arise; even if multi-color callbacks were allowed, the colors are pre-declared, so deadlock avoidance would be easy.

The *libasync-mp* model is more restrictive than many thread synchronization models; for example, there is currently no concept of a read-only color. However, our experience suggests that the model is sufficient to obtain about as much parallel speedup as could typically be obtained from threads.

libasync-mp maintains a single queue of pending callbacks. The library uses one kernel thread per CPU to execute callbacks. Each kernel thread repeatedly dequeues a callback that is eligible to run under the color constraints and executes it. An additional callback, inserted by the library, calls the `select ()` system call to add new callbacks to the queue when the events they correspond to occur.

Ideal support for multi-processor programming would make it easy to port existing programs to a multi-processor, and would make it easy to modify programs to get good parallel speedup. Thus we are interested in two metrics: performance and ease of programming. We evaluate the two criteria in an example application: the SFS file server [9].

Our performance results were obtained on a 700 MHz 4-processor Pentium III Xeon system running Linux kernel 2.4.18. Processor scaling results were obtained by completely disabling all but a certain number of processors on the server while running the benchmark.

All communication between the SFS server and clients is encrypted using a symmetric stream cipher, and authenticated with a keyed cryptographic hash. Because of its use of encryption, the SFS server is compute-bound under heavy workloads and therefore we expect that by using *libasync-mp* we can extract significant multiprocessor speedup.

The modifications to the SFS server are concentrated in the code that encrypts, decrypts, and authenticates data sent to and received from the clients. The callback responsible for sending data to the client is representative of how we parallelized this server: we split the callback into three smaller callbacks. The first and last remain synchronized with the rest of the server code (i.e. have the default color), and copy data to be transmitted into and out of a per-client buffer. The second callback encrypts the data in the client buffer, and runs in parallel with other callbacks (i.e., has a different color for each client). The amount of effort required to achieve this parallel speedup was approximately 90 lines of changed code, out of a total of roughly 12,000 in the SFS server.

We measured the total throughput of the file server to all clients, in bits per second, when multiple clients read a 200 MByte file whose contents remained in the server's disk buffer cache. We repeated this experiment for different numbers of processors.

The bars labeled "libasync-mp" in Figure 1 show the per-

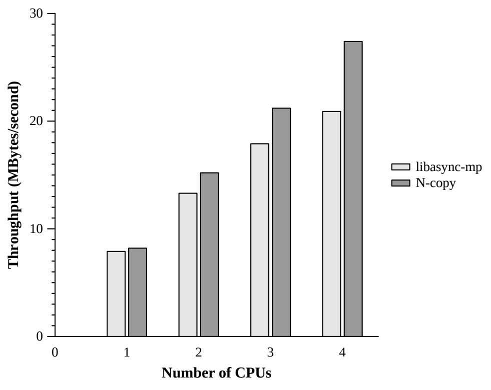

Figure 1. Performance of the SFS file server using different numbers of CPUs, relative to the performance on one CPU.

formance of the parallelized SFS server on the throughput test. On a single CPU, the parallelized server is 0.95 times as fast as the original uniprocessor server. The parallelized server is 1.62, 2.18, and 2.55 times as fast as the original uniprocessor server on two, three and four CPUs, respectively. Further parallelization of the SFS server code would allow it to incrementally take advantage of more processors.

To explore the performance limits imposed by the hardware and operating system, we also measured the total performance of multiple independent copies (as many as CPUs were available) of the original *libasync* SFS server code. In practice, such a configuration would not work unless each server were serving a distinct file system. An SFS server maintains mutable per-file-system state, such as attribute leases, that would require synchronization among the server processes. This test gives an upper bound on the performance that SFS with *libasync-mp* could achieve.

The results of this test are labeled “N-copy” in Figure 1. The SFS server implemented using *libasync-mp* closely follows the aggregate performance of multiple independent server copies for up to three CPUs. The performance difference for the 2- and 3-processor cases is due to the penalty incurred due to shared state maintained by the server, such as file lease data, user ID mapping tables, and so on.

5 Conclusions

The traditional wisdom regarding event-driven programming holds that it offers superior performance but is too difficult to program and cannot take advantage of SMP hardware. We have shown that, by using *libasync-mp*, programmers can easily write event-driven applications and take advantage of multiple processors.

References

[1] The Amoeba reference manual: Programming guide. http://www.cs.vu.nl/pub/amoeba/manuals/pro.pdf.

[2] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proc. Usenix Technical Conference* (2002).

[3] BIRRELL, A. D. An introduction to programming with threads. Tech. Rep. SRC 35, Digital SRC, 1989.

[4] DRAVES, R. P., BERSHAD, B. N., RASHID, R. F., AND DEAN, R. W. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1991), Association for Computing Machinery SIGOPS, pp. 122–136.

[5] ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. Checking system rules using system-specific, programmer-written compiler extensions. In *Usenix Symposium on Operating Systems Design and Implementation (OSDI)* (2000).

[6] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the Fluke kernel. In *Operating Systems Design and Implementation* (1999), pp. 101–115.

[7] LARUS, J. R., AND PARKES, M. Using cohort scheduling to enhance server performance. In *Proc. Usenix Technical Conference* (2002).

[8] LAUER, H. C., AND NEEDHAM, R. M. On the duality of operating system structures. In *Proc. Second International Symposium on Operating Systems, IRIA* (Oct. 1978). Reprinted in Operating Systems Review, Vol. 12, Number 2, April 1979.

[9] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP ’99)* (Kiawah Island, South Carolina, December 1999).

[10] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst. 15*, 3 (Aug. 1997), 217–252.

[11] OUSTERHOUT, J. K. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX technical conference, 1996.

[12] PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the 1999 Annual Usenix Technical Conference* (June 1999).

[13] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (1997), 391–411.

[14] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Oct. 2001), pp. 230–243.