Threads Without the Pain

From Social Computing
Vol. 3, No. 9 - November 2005
by Andreas Gustafsson, Araneus Information Systems

Who knew multithreaded programming could be so hassle-free?

Much of today's software deals with multiple concurrent tasks. Web browsers support multiple concurrent HTTP connections, graphical user interfaces deal with multiple windows and input devices, and Web and DNS servers handle concurrent connections or transactions from large numbers of clients.

The number of concurrent tasks that needs to be handled keeps increasing at the same time as the software is growing more complex. Structuring concurrent software in a way that meets the increasing scalability requirements while remaining simple, structured, and safe enough to allow mortal programmers to construct ever-more complex systems is a major engineering challenge.

Different approaches to concurrent programming offer different trade-offs in terms of performance, scalability, and programming complexity. This article examines and contrasts some of them. It is based on the author's experience writing network server and GUI applications in C and C++ in a Unix environment, but the underlying issues are universal and apply equally to other types of concurrent applications, programming languages, and environments.

What Is Multithreading, Anyway?

People often ask, "Is this program multithreaded?" This may at first seem like a straightforward question to answer with a simple yes or no, but in fact it's often not. The word multithreaded means different things to different people, and often people can't even agree on whether a particular program should be considered multithreaded or not.

Usually, the question, "Is it multithreaded?" reflects a specific concern, such as:

- Does it handle more than one user, connection, query, transaction, or other operation at a time?
- Does its user interface remain responsive while it is performing a long-running computation?
- Can it take advantage of more than one processor?
- Does it have random, nondeterministic, hard-to-find bugs?

In the case of most practical programs, these questions don't all have the same answer. Sometimes they are not even the right questions; if you have a dual-processor machine, which kind of program would you rather use? One that uses both processors, or one that uses only one of the processors but runs faster because it is based on an algorithm that can't be parallelized but is very, very fast?

When the word multithreaded is used in the rest of this article, it will not be in the sense of any of the previous questions. Instead, let's simply define multithreaded as having more than one flow of control, regardless of where those flows execute or how many of them are making progress at any given moment.

The Event-driven Approach

One popular approach to writing programs that need to deal with multiple independent sources of input is the event-driven model. In an event-driven program, a central event loop watches all external sources of data (e.g., network connections, input devices, and timers) and invokes callback functions to process each piece of data as it arrives. This model is familiar to users of GUI toolkits and is the basis of many network servers such as the BIND family of DNS servers.

Event-driven programs are highly portable since they don't require any special multithreading support from their environments, and in practice they are often successful enough in creating an illusion of concurrency that users think of them as multithreaded, even though they actually have only a single thread of control.

The downside to event-driven programming is that it is messy, tedious, unstructured, and repetitive. In an event-driven program, the event loop is in control, not the programmer. When an event-driven program wants to perform an I/O operation such as reading some data from a network connection, it can't simply stop and wait for the data. Instead, it needs to set up an I/O request and then return to the event loop. When the data is available, the event loop will invoke an I/O completion callback function to process it. This forces the programmer to ruin a perfectly fine program by chopping it up into a series of short callback functions.[1]

Algorithms that would naturally be expressed as hierarchies of function calls and loops have to be modified by splitting them into separate callbacks at each point where the algorithm needs to wait for network input or other external events. At these points, the complete state of the algorithm must be painstakingly stored away in a data structure while execution returns to the event loop awaiting invocation of the next callback.

At the points where the program is forced to return to the event loop, it will be unable to use local variables, while or for loops, conditionals, subroutines, or recursion—basically any of the fundamental tools of structured programming. Instead, the control flow of the program becomes centered on the operation of returning to the event loop to wait for a callback, which essentially is a form of (delayed) GOTO. In terms of control flow, an event-driven program tends to resemble an assembly language program, even when written in a supposedly high-level language.

Threads to the Rescue

Multithreading offers an alternative to the event-driven model. By creating a separate thread of control for each network connection, input device, user, or other appropriate entity, the programmer can express the flow of control from the perspective of that entity in a natural way. I/O operations that would have involved a return to the event loop in the event-driven model now appear as simple function calls that block—that is, cause the currently executing thread to stop until the operation is complete. While the thread is stopped, other threads may make progress. Since there is no need to return to an event loop, it is possible to do I/O in the middle of nested loops or recursive functions, and local variables are preserved across each I/O operation.

Although multithreading is often thought of as a technique for improving performance, here we are using threads for the separate goal of improving program structure, making the code smaller, more readable, and more expressive. The use of threads to express structure is not limited to situations involving I/O; even within a program, threads can be used to express producer-consumer pairs, state machines, and other structures that naturally map onto multiple threads of control.

Fear of Threads

If event-driven programming is so awkward, then why is so much software still written in the event-driven style? One reason is that threads have a reputation of being error-prone and hard to use. In a talk titled "Why Threads Are a Bad Idea (for Most Purposes)," John Ousterhout points out that threads are too hard to use for

most programmers because of the difficulty of dealing with synchronization, locking, race conditions, and deadlocks, and he argues that the event-driven model should be used instead.[2]

Synchronization issues are extremely hard to get right. A good illustration of just how hard it can be is the case of the double-checked locking pattern,[3] a technique for efficiently initializing global objects in multithreaded programs. This technique has been widely used and taught, but does not work reliably because of subtle issues related to reordering of memory accesses by optimizing compilers or the memory subsystems of multiprocessor computers.[4] If even the experts can't get synchronization right, how can we expect the average programmer to do so?

Fear Preemption, not Threads

Although programmers fear asynchrony for good reason, translating that into a fear of multithreading is a mistake. It is often assumed that multithreaded programs need to worry about synchronization, locking, and deadlocks, and that event-driven ones do not, but there are in fact event-driven programs that do have to worry about these issues, as well as multithreaded ones that do not.

The reason why most event-driven systems are safe from the dangers Ousterhout warns about is not their event-driven nature. Rather, it is that they make an unwritten guarantee that only one event handler will run at once, and that it will not be interrupted while running. If an event-driven system doesn't make these guarantees, any shared data accessed by the event handlers needs to be protected by locks, just as in a typical multithreaded system. For example, this will be the case in an event-driven system that can execute multiple event handlers simultaneously in order to use multiple processors.

Conversely, the reason why most multithreaded systems need to worry about concurrent access to data is not that they have multiple threads of control, but that those threads can execute simultaneously, or be forcibly preempted to schedule the execution of another thread. If a multithreaded system guarantees that only one thread will run at a time and that a running thread will not be preempted, there will be no opportunity for one thread to upset the invariants of another and, therefore, no need for locking.

In other words, the question of whether the control flow of a program is structured as events or threads is orthogonal to the question of whether those threads and events are preemptively scheduled;[5] this is illustrated in Table 1. The dangers Ousterhout warns against are not dangers of threads, but dangers of preemption.

TABLE 1

Control Flow	Preemptive Scheduling	Cooperative Scheduling
Multithreaded	Preemptive threads	Cooperative threads
Event-driven	Concurrent events	Serialized events

Cooperative Threading

Most multithreading systems emphasize the use of threads for performance and consider preemptive scheduling to be a feature, since it allows them to use multiple processors and to guarantee that no single thread can unfairly monopolize the processor and keep other threads from running indefinitely.

There is also a class of multithreading environments, however, that emphasize the use of threads to express structure and that consider preemptive scheduling to be a bug. Instead of guaranteeing that every thread will eventually yield control to other threads, they guarantee the opposite: that no thread will have to yield control unexpectedly. This style of multithreading goes by many different names: nonpreemptive, synchronous, or

cooperative threading, or coroutine-based programming. In a cooperatively multithreaded system, threads yield control voluntarily only at certain well-defined points, such as blocking I/O operations or explicit yield calls.

With preemptive threads, any data structure shared among the threads may be accessed by another thread at any time. This makes any access to such a data structure a candidate for race conditions and forces the programmer to introduce locking in a large number of places. On the other hand, when nonpreemption is guaranteed, a shared data structure can be accessed by other threads only at the few well-defined points where the current thread voluntarily blocks. In most cases this completely eliminates the need for locking, along with most of the dangers traditionally associated with multithreading.

Cooperative multithreading is in fact exactly as safe from race conditions as event-driven programming. Any cooperatively multithreaded program can be turned into an equivalent event-driven one by turning every blocking operation into a return to the event loop (at the expense of adding lots of bookkeeping so that the operation can be correctly resumed), or vice versa.

Threading Overhead

Another reason why programmers shun threads is that they often come with a great deal of overhead in both time and memory. Creating and destroying threads can be slow, and in the case of preemptive threads, there is locking overhead associated with each access to shared data. A lot of memory (or at least virtual address space) is wasted because a fixed-size stack is allocated for each thread.

These sacrifices may be worthwhile in applications that need more performance than a single processor can provide and that exhibit sufficiently coarse-grained parallelism so that the overhead can be amortized over a large amount of computation; but for the programmer who would like to use threads simply because the solution to his problem would be naturally expressed as thousands of small, frequently communicating threads, the overhead may be too great.

State Threads

One multithreading environment that attempts to combine the structural benefits of multithreading with the scalability and safety of event-driven programming is State Threads.[6] This is an open source, portable, lightweight, user-space thread library that was specifically designed to replace the event loop in Internet applications. The name State Threads refers to the use of threads as a way to keep track of program state that would have to be managed by hand in an event-driven program.

Unlike many other thread libraries for Unix-like systems, State Threads is not an implementation of the Posix Threads (pthreads) standard. Instead, it provides its own programming interface consisting of functions such as st_thread_create(), st_thread_join(), st_read(), st_write(), etc. It omits pthreads features that are of dubious value or cause a lot of overhead (notably, the per-thread signal mask) and adds features aimed at simplifying network programming, such as the ability to specify a timeout when doing a network I/O call.

More importantly, State Threads differs from pthreads by guaranteeing nonpreemption. Thanks to this guarantee, most State Threads applications need to do no locking at all. The programmer has to make sure only that any shared data is in a consistent state when calling a blocking function such as st_read(); this is no different from programmers of event-driven systems having to make sure that shared data is consistent when returning to the event loop.

Compared with a standard pthreads implementation, State Threads is remarkably simple and lightweight. The library itself consists of fewer than 4,000 lines of C code, and the operations of thread creation, thread destruction, and thread context switch take a fraction of a microsecond each on modest PC hardware. This is

comparable to the overhead involved in dispatching an event handler in an event-driven system. State Threads-based applications often outperform both equivalent pthreads-based ones (thanks to faster context switches and the absence of locking overhead), and event-driven ones (thanks to being smaller and simpler).

Those Pesky Fixed-Size Stacks

Although lightweight thread libraries such as State Threads are competitive with event-driven programming in terms of execution-time overhead, events still have an edge when it comes to memory. Over the past decades, most fixed-size programming structures have given way to more dynamic ones: fixed-size arrays have been replaced by dynamically allocated or even variable-size ones, fixed-size files by dynamically extended files, and so on. The stack is a notable exception: multithreaded systems still allocate a fixed-size stack to each thread. This leaves the thread programmer with the uneasy trade-off between making the stack too small and risking stack overflows, or making it larger than necessary and wasting memory and address space.

When the number of threads is small and memory is plentiful, this is usually not a problem; you can afford to make the stacks comfortably large—say, a megabyte each. It does become a problem when attempting to scale a multithreaded system to many thousands of threads or when running on memory-constrained embedded systems.

A typical default stack size in a pthreads implementation is 1 MB. The default stack size in the State Threads library is 64 KB, which facilitates scaling to a larger number of threads but requires the State Threads programmer to exercise some care to avert stack overflow, such as eliminating large stack-allocated arrays and avoiding deep recursion. On some systems, the standard C library also causes problems by allocating large buffers on the stack—for example when formatting output in printf(). Nevertheless, some State Threads-based applications can get by with as little as 4 KB of stack per thread; these will realistically scale to several hundred thousand threads.

Toward the Million-Thread Application

There are applications that could benefit from very large numbers, even millions, of threads—for example, network traffic simulators that attempt to model the behavior of a large number of independent computers. With such a large number of threads, the wasted memory resulting from the fixed-size stacks becomes a problem.

The long-term solution to this problem would be to get rid of the fixed-size stacks entirely. Programmers using the Python language already have a choice between stack-based and stackless implementations of Python,[7] where the stackless variety offers lightweight microthreads at the expense of being slightly slower at executing serial code. Similarly, Scheme programmers can choose between stack-based and stackless implementations of Scheme.

C and C++ programmers also ought to have a choice between stack-based and stackless implementations. A stackless C compiler would allocate space for function arguments and local variables on the heap instead of a stack. This would not require any changes to the C language itself, though there would be a performance hit and some challenges in interoperating with existing libraries that use stack-based calling sequences.

Putting It All Together

Different forms of multithreading are appropriate for solving different problems. There are applications involving heavy numerical computation where the performance of a single CPU just isn't enough, and full-blown preemptive threads are the only option. Other applications are light on computation and heavy on

interaction with the user, the network, and among the threads themselves. These are better served by a cooperative approach. Cooperative threads won't replace preemptive threads, but they can be an excellent replacement for the ubiquitous event loop. When different parts of the same application place different demands on threading, the best solution may be a mixed approach, such as running a lightweight, cooperative thread scheduler as one thread of a preemptive thread system.

Although most of today's programming environments provide support for events and preemptive multithreading, fully integrated support for cooperative threads—not to mention support for mixing preemptive and nonpreemptive threads—is rare. Third-party libraries such as State Threads and languages such as Stackless Python help bridge this gap for the time being. Hopefully, future systems will offer integrated and equal support for both preemptive and cooperative threads.

References

1. Fuchs, M. 1996. Escaping the event loop: An alternative control structure for multithreaded GUIs. In Engineering for Human-Computer Interaction, ed. C. Unger and L. J. Bass. Chapman & Hall.
2. Ousterhout, J. 1996. Why threads are a bad idea (for most purposes). Invited talk at the Usenix Technical Conference.
3. Schmidt, D., and Harrison, T. 1996. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. 3rd Pattern Languages of Program Design Conference; http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf.
4. Meyers, S., and Alexandrescu, A. 2004. C++ and the perils of double-checked locking. Dr. Dobbs Journal (July-August); http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.
5. 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; http://research.microsoft.com/sn/Farsite/USENIX2002.ps.
6. State Threads Library for Internet Applications; http://state-threads.sourceforge.net/.
7. Stackless Python; http://www.stackless.com/wiki/Stackless.

ANDREAS GUSTAFSSON (gson@araneus.fi) has been writing event-driven graphical user interfaces and Internet applications since 1989. He has spent the better part of the past 10 years developing DNS (Domain Name System) servers, notably as a co-author of the Internet Systems Consortium's preemptively event-driven BIND 9 server and as the architect of the cooperatively threaded Nominum CNS (Caching Name Server). He currently works for his own company, Araneus Information Systems, as a consultant and purveyor of hardware random number generators. He holds an M.Sc.Eng. from Helsinki University of Technology.

Back to Threads Without the Pain