Event-Based Programming

Hello, and Welcome to this lesson on event-based programming.

We have previously learned about threads to manage concurrency, and in this lesson we will point out a number of difficulties that thread-based programming entails. We will then explore how cooperative task management, in particular event-based programming, can provide high-performance concurrent programming as well.

We will set the stage by enumerating a number of problems with threads.

We will then review cooperative task management, and focus on event-driven programming as the most prominent example of this type of concurrency. We look at how event driven programming can be implemented, and use the POSIX select/accept calls to do this.

Events are a wonderful tool to build high-performance concurrent systems, and we will discuss why, but they also have serious drawbacks, which we will learn about as well.

At the end of this lesson you will know how event-based systems work, and you will be in a good position to make an educated choice when designing your own high-performance concurrent system.

Let's begin.

Threads have Limitations

In a famous paper, called "Why Threads are a Bad Idea (for most purposes)", John Ousterhout, argued, in a very articulate way, against the indiscriminate use of threads to provide concurrency.

He argued that it is often wrong to use threads, because threads are too hard for most programmers to use.

Even for experts, development using threads is painful.

He even went so far as to propose a tongue-in-cheek categorization of programmers, ranging from visual basic programmers on the casual side, to threads programmers on the wizards side. Of course we want to take this graph and the following statements with a grain of salt for two reasons: First, this paper was written with the intent to initiate a discussion, and was therefore written a bit more polemically than typical technical paper. Second, Ousterhout is talking about the development of sophisticated high-performance systems, where even subtle performance bugs are considered very serious. I am sure that he would not consider a casual use of threads as wizardry.

He comes to a two-fold conclusion, where he first argues that threads should be used only in multiprocessor systems when parallelism (he calls it CPU concurrency) is needed.

He generally concludes that, for most purposes where designers argue for the use of threads, events are better.

Let's follow up a bit further on these general claims.

Why Threads are "Hard"

One important point we have already discussed in the introductory lesson on concurrency and threading.

Because the scheduler controls when what part of a preemptive task (in other words, a thread) gets to execute, all inter-task invariants have to hold at all time. This must be ensured through synchronization..

If you make a mistake in the synchronization, for example, you forget a lock, one or more inter-task invariants may be violated, which in plain English means that data may get corrupted. We mark this point with a little red asterisk to point out that this is primarily a locking issue rather than an inherent threading issue. There are synchronization mechanisms, such as transaction based synchronization, which address some of these issues. We will discuss transactions and similar mechanisms in our lessons on synchronization.

Another problem with threads are deadlocks, where threads may acquire locks in an order that may give rise to circular dependencies, which in turn deadlock the threads.

Let's look at an example, where we have two threads, 1 and 2, (5) and 2 locks, A and B.

At some point, thread 1 makes a request to acquire lock A.

Since the lock is free, it can be allocated to thread 1.

Some time later, thread 2 requests Lock B, and gets it. (10)

Thread 2 now requests lock A and promptly gets blocked, because lock A is allocated already to thread 1.

A bit later, thread 1 requests lock B and gets blocked as well, because lock B is already allocated to thread 2.

None of the two threads can now make progress. We say that they are deadlocked.

Deadlocks are a very common problem in operating systems, and the use of thread and locks requires designers of one part of the operating system to have a good idea of how functions that they use in other modules are build, because they need to know how many and in which order locks are acquired. Modules therefore cannot be designed completely independently from each other. The technical term of this that the modules are not "composable".

A typical scenario is the following: Let's assume that we have two modules, which have been realized independently.

Thread 1 makes a call to a function in Module A, which acquires a lock, for some reason. This function in Module A calls a function in Module B, which makes thread 1 sleep, maybe because we the system has to wait for some external event to happen.

At some time later, another thread, call it Thread 2 comes along and tries to wakeup thread 1.

For this it has to call a function in Module A to route the request to the wakeup call. Unfortunately, thread 2 gets blocked in Module A because it fails to get the lock.

We now have a deadlock: Thread 1 cannot continue because it has to be woken up.

And Thread 2 cannot continue because it has been blocked trying to acquire a lock that is owned by thread 1.

A classical situation is when deadlocks are triggered by callbacks.

Let's uses the same scenario as before, with thread 1 making a call to Module A, which in turns intends to make a call to Module B. It acquires Module A's lock.

At the same time, thread 2, maybe through an exception handler, makes an upcall, which calls a function in Module B, which the intention to call a function in module A. It acquires the lock of Module B.

Now thread 1 cannot continue because it cannot acquire Module B's lock, and thread 2 is blocked as well because it cannot acquire Module A's lock.

As a result, the two threads are deadlocked.

In addition, Ousterhout argues that it is difficult to achieve high level of performance with threads, and this for several reasons:

- First, simple locking, for example protection of critical sections with monitors, unnecessarily reduces concurrency.
- Fine-grained locking, on the other hand, is very complicated and hampers the performance in the normal case, when we have no contention for locks.

These two points are really more about locks specifically, rather than about threading in general. So we mark them with asterisks.

In addition, the thread management additionally limits performance. Scheduling adds overhead, and context switches do that as well.

These performance limitations of threads can be easily measured. We see in this example of the SEDA web server how the throughput of the system (measured in web requests per second) grows with the number of threads up to 16 threads, and quickly collapses when the number of threads increases beyond 16 threads. As the throughput reduces, the service latency of each request increases dramatically.

Ousterhout also argues that threads are not well supported.

Unless it relies on standard thread libraries, for example POSIX pthreads, threaded code is very difficult to port.

Many common libraries have not been designed with threads in mind, and are therefore not thread safe.

Also, many kernels and window systems are single threaded, meaning that even multithreaded applications must access the kernel in a serial fashion.

Finally, analysis and debugging tool support is somewhat limited for multithreaded applications.

Cooperative Task Management

As we learned in an earlier lesson, cooperative task management is an alternative to threads.

In a cooperative task system, concurrency is provided by having tasks voluntarily give up the CPU in well-defined points of their execution. We recall that inter-task invariants now need only be ensured at these points where the execution switches from one task to the other.

Event-Driven Programming

Event-driven systems are the most well-known approach to collaborative task management.

Let's start with the task model that we are familiar with. Clients contact the server and send requests, which the server handles as tasks. Let's modify this model a bit

First, rather than sending what we may call "long-lasting" requests, such as "download this movie", clients send sequences of events. One event can be the client connecting to the server. Another event could be the client requesting one frame of the movie. In this way, the "download this movie" would become a long sequence of very small requests, each packaged as an event.

Second, the tasks would be replaced by event handlers, where each handler would "register" for an events, and therefore would be invoked whenever one of these events is processed.

Events are processed by the event loop, which waits for events and then calls the appropriate event handler when an event comes in.

We can think of as event handlers as functions, which don't get interrupted. If there is a reason for a handler to be interrupted, the handler is poorly designed and should be split up into two handlers. As a result, handlers are very short-lived. . One interesting aspect of this form of concurrency is that I/O can be easily integrated. If the application waits for a result of a READ operation from a disk, it can register an event handler that is invoked when the READ returns. Once the READ does return, an event is generated, and the appropriate event handler is eventually called.

Event-Driven Programming: Step-by-Step

Let's look how such a system would work, step-by-step. Our server, contains an event loop, which in turn manages a queue of waiting events. We call this the event queue.

There is also a set of event handlers. We assume that these event handlers are part of one task.

A client issues some sort of request, and the event gets added to the event queue.

At some point the event loop wakes up, (5) and identifies the event handler that is responsible for the next event in the queue.

It removes the event from the queue and passes the event information to the event handler, and starts the event handler. This can be as simple as a function call.

The event handler now runs to completion.

In this example, the event handler creates a new event and deposits it in the event queue.

And then returns, for example, by returning from the event handler function call.

The event handler loop takes over again, and finds an event in the queue. (10)

It identifies the next event handler for this event.

It removes the event from the queue and passes is to the event handler. The event handler is started.

The event handler returns, and the event queue is empty.

Some times later a disk becomes ready to send some data, and an event is added to the queue.

Before the event handler wakes up, another client sends a request, which is added as an event to the queue as well. (15)

The event handler wakes up, identifies the event handler to be called, removes the event, hands it over to the event handler, and starts the event handler.

The event handler returns, and this process continues with the next event.

Example: Event Programming using select/accept

Let's implement an event handler loop for a simple IP server using simple POSIX select/accept calls.

We have a server, which initializes a socket, using some function **passiveTCPSock()**, and then enters the event handler loop, which in this case is as simple as a "for loop".

Inside the event loop we use the select() function to wait for the next event to occur. In this example, we handle only events on file descriptors. It would be easy to add an explicit event queue.

In order for the **select()** function to work, we need a few lines of administrative code, which we will largely ignore in this example. (5)

Once an event has come in, we check whether this event is a new client connecting.

If it is, we accept the new connection from the client and add it to the list of file descriptors that we are monitoring for incoming events.

Now we go through all the file descriptors other than the master socket to see if they have been flagged has having an event in the select() call.

If the socket descriptor has an event, we identify the handler for this event using a function identify_handler, which we implemented somewhere else. We pass the file descriptor as a key for the event. (10)

We then invoke the event handler that was returned from identify_handler.

In this way we can build a high-performance IP server without the user of threads. At no point do we have to deal with synchronization.

Events: Advantages

In summary, the advantages of event-based concurrency that have been pointed out are:

- Ease of use, given that we don't have to deal with explicit concurrency, or preemption, or the synchronization and deadlock problem that come along with this.
- Event-driven programs are easier to debug, because we don't have to worry about interference by the system scheduler. Also, the problems are often much easier to identify and trace because of the lack of explicit synchronization, and largely of race conditions.
- Event-driven systems are very fast, because there is no context switching overhead, and no scheduling overhead in general. This is particularly important for compilers, which can now much better optimize the code, given that it is basically mostly function calls.
- Since the code is mostly function calls, except maybe for parts of the event loop mechanism, it is much easier to port.

Events: Manual Control Flow Management

The difficulties with Event-driven programming become evident the moment we try to implement a simple system.

Let's use the example of a simplified web server. Whenever a new connection comes in, the server starts a new thread, and passes the socket for the connection to the thread function **thread_main**.

The local session variable contains information needed to handle the request.

First, we start the session, which initializes the session variable.

We then read the incoming request.

And check the cache if we have the data in memory already. (5)

If we don't, we read it from a file.

The code for **check_cache()** is listed here.

Once we have the data, we write the response back, and terminate the handling of this request. We notice that this code is very straightforward.

Let's see what happens if we have to implement the same functionality in an event-based system and we have to split up the thread function into a sequence of event handlers.

Whenever a new connection comes in, the system starts the function **StartHandler**, which establishes the session variable and queues up an event to be handled by the **RequestHandler**.

The **RequestHandler** does whatever needs to be done to read the request from the connection. Once this is done, the session variable is updated accordingly, and an event is queued up for the **CacheHandler**.

When the **CacheHandler** is called, it checks if the item is in the cache.

If it is, an event is enqueued for the **WriteHandler**.

Otherwise, an event is enqueued for the **ReadFile Handler**.... and so on.

Finally, the **ExitHandler** does whatever is needed to terminate the handling of the request and then frees the session variable.

We see that the code is much more cumbersome compare to the thread based code.

Things become even more complicated for when we add typical error management.

For example, if the read fails, we want to be able to abort the entire session.

In our thread-based solution, the code to do this is very simple: if the read-request fails, return. Because the session information is on the stack, it gets freed automatically.

This is much more convoluted in the event-driven code. Inside the **RequestHandler** we need to be explicitly direct the execution to the **ExitHandler** function, because it is there where we free the session information and maybe do other things to recover from the error.

In more complex cases the programmer may not be able to keep track of where such a session variable would stop being used, and they therefore would have to rely on garbage collection to get rid of unused session variables. This adds quite a lot of complexity to the code.

Events: Manual Stack Management

One reason for all these complications is that the programmer has to manually manage the stack. Let's see what we mean by this.

In the thread-based solution, we declare the session variable to be local to the thread function. This allocates the session variable on the stack for the life time of the function. No explicit management of this variable must be done.

In an event-driven system, the handling of the client request is scattered across multiple handler functions. We can not use the stack to store session information. This information must now be managed explicitly.

It has to be explicitly allocated in the **StartHandler** and freed in the **ExitHandler**. It has to be passed to all the handlers, more importantly, it has to be passed to the event loop as well, in order to pass it to the next event handler.

In summary, what in the thread function just sits on the stack must, in an event based system, be passed manually from handler to handler.

Events: Problems - Summary

Let's summarize the difficulties of event-based programming.

We talked about the overhead of manual management of the flow control. We need to explicitly create and manage the sequence in which events must be handled. This can become quite complicated, in particular when the calling graph becomes dense, for example due to error handling.

In addition, we basically loose the stack. What would be automatically managed on the stack now has to be manually forwarded from event handler to event handler.

We have not talked about how event-driven systems perform in the presence of unexpected blocking, for example by a page fault. The level of concurrency is to a large extend controlled by the event handler functions being sufficiently short to not unduly block other functions. If now an event handler gets blocked by a page fault, the entire event loop gets blocked as well, and this means that all other event handlers have to wait until the page fault is over. In a thread based system the scheduler would detect the blocking and bring in other threads until the page fault completes. In pure event-driven processing this is not possible unless the page fault system is implemented in an event-based fashion as well.

Finally, we have also not talked about the fact that an event-driven system with one event loop can only take advantage of one processor in the system. It does therefore not support parallelism. In principle it is easy to benefit from a multiprocessor by running multiple event loops, each in a separate thread, one per processor. In practice, we are back to running multiple threads, which brings back the need for synchronization, which we tried to avoid. . In summary, the relative benefits of threading vs. events depend on many aspects of the system, ranging from the number of processor, to the specific nature of the inter-task invariants, to the exact type of synchronization primitives available in the system, the propensity for deadlocks, and many others.

Conclusion: Event-Based Programming

After this lengthy pros and cons of Events we have come to the conclusion of this lesson on event-based programming and concurrency.

We have started by summarizing a number of salient points of a famous position paper by John Ousterhout, where he strongly argued against thread-based system design in favor of an event-based approach. We noted that some of these points were more about shortcomings of locks rather than those of threads.

We then learned about event-based programming and system design. We learned how event loops and event handlers work together to provide high-performance concurrent service.

Event-based systems are very fast, don't suffer of synchronization problems, are relatively easy to trace and debug, and they provide – if designed well – a very high level of concurrency. . They are difficult to design, however, in particular for systems with rich calling graphs and much state that needs to be managed per session. In addition, they tend to work less well on top of operating systems with unexpected blocking and in multiprocessor systems.

I genuinely hope that you enjoyed this lesson on event-based programming.

With what you have learned in this lesson, you should be able to write a simple event-driven high-performance concurrent application. If you find yourself in the position to decide whether to design you system using an event- or a thread-based model, you now have the knowledge to evaluate the characteristics of your system and the application such that you can decide in an informed fashion which model to use.

Thank you for watching.