Introduction

So far, we've written about concurrency as if threads are the sole option to create concurrent applications. This is not entirely accurate, as is the case with many things in life. In particular, a new form of concurrent programming is frequently utilized in both GUI-based apps and various types of internet servers. This kind of parallelism, known as event-based concurrency, has gained popularity in modern systems, including server-side frameworks like node.js. Still, it has its roots in C/UNIX systems, which we'll examine below.

There are two issues that event-based concurrency solves. The first is that appropriately handling concurrency in multi-threaded systems can be difficult; as we've seen, missing locks, deadlock, and other unpleasant issues can occur. The second is that with a multi-threaded application, the developer has little or no influence over what is scheduled at any time; instead, the programmer merely generates threads and hopes that the underlying OS distributes them among available CPUs in a fair manner. Unfortunately, due to the difficulty of creating a general-purpose scheduler that works well in all circumstances for all workloads, the OS will occasionally schedule work in a less-than-optimal manner. As a result, we have.

THE OBJECTIVE: TO LEARN HOW TO BUILD CONCURRENT SERVERS WITHOUT THE USE OF THREADS.

How can we create a concurrent server without utilizing threads, retaining control over concurrency while avoiding some of the issues that multi-threaded apps seem to have?

The Basic Idea: An Event Loop

As previously indicated, the basic approach we'll utilize is known as event-based concurrency. The strategy is straightforward: you simply wait for something (i.e., an "event") to happen; when it does, you determine what type of event it is and perform the minimal work required (which may include making I/O requests or scheduling other events for future handling, among other things). That concludes our discussion.

Let's take a look at what a canonical event-based server looks like before diving into the details. The event loop is a simple construct used in such applications. This is how pseudocode for an event loop looks:

That's all there is to it. The main loop simply waits for something to happen (by executing getEvents() in the code above) and then processes each event one by one; the code that processes each event is referred to as an event handler. Notably, a handler's processing of an event is the system's only activity; hence, determining which event to handle next is the same as scheduling. Thus, one of the primary benefits of the event-based approach is the explicit control over scheduling.

As a result of this conversation, we've come up with a broader question: how does an event-based server identify which events happen, particularly when it comes to network and disk I/O? For example, how can an event server detect if a message has arrived for it, specifically?

An Important API: select() (or poll())

Sample content New Page

Using select()

You put this into context, consider how to use select() to determine which network descriptors have incoming messages. A simple example is shown in Figure 33.1.

The server enters an indefinite loop after some startup. First, the FD ZERO() macro is used inside the loop to clear the set of file descriptors, and then FD SET() is used to include all of the file descriptors in the set from minFD to maxFD. This set of descriptors could, for example, represent all of the network sockets that the server is monitoring. Finally, the server uses select() to determine which connections have data accessible. The event server can then use FD ISSET() in a loop to see which descriptors have data ready and process the incoming data.

Of course, a real server would be more complicated, requiring logic to be used when sending messages, issuing disk I/O, and various other aspects. For more information, see Stevens and Rago for API details, or Pai et al. or Welsh et al. for a solid introduction of event-based server flow.

Why Simpler? No Locks Needed

Concurrent programs have difficulties that are no longer present when using a single CPU and an event-based application. Because just one event is handled at a time, there is no need for locks to be acquired or released; the event-based server cannot be interrupted by another thread because it is single threaded. As a result, the basic event-based method is immune to concurrency issues that plague threaded programs.

A Problem: Blocking System Calls

So far, event-based programming appears to be a fantastic idea, right? You create a simple loop and respond to events as they occur. You don't even have to consider locking! But there's a catch: what if an event needs you to make a system call that can cause a delay?

Consider the following scenario: a client sends a request to a server to read a file from disk and deliver its contents to the client (much like a simple HTTP request). The event handler will need to write a sequence of read() calls to obtain information from the file in response to such a request. The server will most likely begin transmitting the results to the client once the file has been read into memory.

If the metadata or data needed is not in memory, open() and read() can send I/O requests to the storage system, which can be slow. This is not an issue with a thread-based server: other threads can operate while the thread initiating the I/O request is suspended (waiting for the I/O to complete), allowing the server to progress. Indeed, it is because of this natural overlap of I/O and other operations that thread-based programming feels natural and straightforward.

There are no other threads to run in an event-based method, only the main event loop. And this means that if an event handler makes a blocking call, the entire server will do the same: block until the call is completed. When the event loop becomes stuck, the system remains idle, wasting a lot of resources. As a result, there is a rule in event-based systems that must be followed: no blocking calls are permitted.

A Solution: Asynchronous I/O

Several current operating systems utilize asynchronous I/O to get around this limitation. This is a new technique for sending I/O requests to the disk system. Applications can use these interfaces to send an I/O request and swiftly return control to the caller before the I/O is completed; other applications can check if certain I/Os have been completed.

For example, let's look at the UI on a Mac (other systems have similar APIs). In layman's terms, the APIs are based on a simple structure known as the struct aiocb, or AIO control block. A simplified version of the structure looks like this (for more information, consult the manual pages):

The following information must be entered before an asynchronous read can occur: the file descriptor of the file to be read (aio fildes), the offset within the file (aio offset), the length of the request (aionbytes), and finally, the target memory location into which the read results should be copied (aio buf).

The application must then make an asynchronous call to read the file after filling up this structure; on a Mac, this API is simply the asynchronous read API:
This call attempts to perform the I/O; if it succeeds, it simply returns immediately, allowing the application (i.e., the event-based server) to continue working.
However, there is one more puzzle piece to be solved. How can we determine when an I/O is finished and the buffer (referred to by aio buf) now has the desired data?

One more API is required. On a Mac, it's known as aio error (which is a little confusing) (). This is how the API looks:

This system call determines whether the aiocbp-referenced request has been completed. If it has, the routine returns success (a zero); if it hasn't, it returns EINPROGRESS. As a result, an application can regularly poll the system via a call to aio error() to see if any outstanding asynchronous I/O has been completed.

One thing you may have observed is that checking if an I/O has been completed is uncomfortable; if a program has tens or hundreds of I/Os issued at any given time, should it simply keep rechecking each one, or should it wait a short bit beforehand, or?

Some systems use an interrupt-based technique to solve this problem. This method uses UNIX signals to notify applications when an asynchronous I/O completes, eliminating the need to query the system many times. However, as you will see (or have already seen) in the chapter on I/O devices, this polling vs. interruptions dilemma exists in devices.

The pure event-based method cannot be implemented in systems with asynchronous I/O. In their place, though, brilliant researchers have devised approaches that work reasonably well.

Another Problem: State Management

Another disadvantage of the event-based method is that it is more challenging to write than typical thread-based code. In thread-based programs, state is already on the thread stack when an asynchronous I/O is issued, so there’s no need for the event handler to package up state so the following handler can use it after the I/O has been completed. This effort is referred to as manual stack management by Adya et al., and it is essential to event-based programming.

To illustrate this notion, consider a simple scenario where a thread-based server must read from a file descriptor (fd) and then write the data read from the file to a network socket descriptor (sd). For example, the following is the code (without error checking):
As you can see, accomplishing this kind of work in a multi-threaded application is straightforward; when read() returns, the code immediately knows which socket to write to because that information is on the thread’s stack (in the variable sd).

Life isn’t easy in an event-based system. To accomplish the same action, we’d use the AIO calls to perform the read asynchronously first. Let’s say we use the aio error() function to check for read completion regularly; how does the event-based server know what to do when that call informs us that the read is complete?

The approach is to employ a continuation, which is an old programming language construct. Though it may appear complex, the concept is straightforward: simply store the necessary information to complete processing of this event in some data structure; when the event occurs (i.e., when the disk I/O completes), search up the required information and handle the event.

The answer in this scenario would be to store the socket descriptor (sd) in a data structure (e.g., a hash table) that is indexed by the file descriptor (fd). Then, when the disk I/O is finished, the event handler will look up the continuation using the file descriptor and provide the value of the socket descriptor to the caller. Finally, the server may conduct the remaining work to write the data to the socket.

What Is Still Difficult With Events

There are a few other issues with the event-based method worth mentioning. For example, when systems went from a single CPU to numerous CPUs, part of the event-based approach's simplicity vanished. The event server must run many event handlers simultaneously to use multiple CPUs; as a result, the usual synchronization difficulties (e.g., critical sections) exist, and the usual remedies (e.g., locks) must be used. As a result, basic event processing without locks is no longer practicable on newer multicore systems.

Another issue with the event-based method is that it is incompatible with some types of system activity, such as paging. For example, if an event-handler page faults, it will block, preventing the server from progressing until the page fault is resolved. Even though the server has been designed to avoid explicit blocking, implicit blocking due to page faults is difficult to avoid and can cause significant performance issues when it occurs.

A third concern is that, as the meanings of separate procedures change over time, event-based code can be challenging to manage. If a procedure goes from non-blocking to blocking, the event handler that calls it must likewise alter to fit the new nature of the routine by ripping itself in half. Because blocking is so bad for event-based servers, a programmer must always be aware of changes in the semantics of the APIs that each event utilizes.

Finally, while asynchronous disk I/O is now available on most platforms, it took a long time to get there, and it never completely combines with asynchronous network I/O in the way you might expect. While it would be nice to only utilize the choose() interface to manage all outstanding I/Os, most cases require a combination of select() for networking, and AIO calls for disk I/O.

Summary

We’ve given a quick overview of a distinct type of concurrency based on events. Event-based servers delegate scheduling control to the application, but at the cost of increased complexity and difficulty of integration with other modern systems (e.g., paging). Because of these difficulties, no single technique has emerged as the most effective; hence, threads and events will likely continue to exist as two distinct approaches to the same concurrency problem for many years.