33

Event-based Concurrency (Advanced)

Thus far, we've written about concurrency as if the only way to build concurrent applications is to use threads. Like many things in life, this is not completely true. Specifically, a different style of concurrent programming is often used in both GUI-based applications [O96] as well as some types of internet servers [PDZ99]. This style, known as **event-based concurrency**, has become popular in some modern systems, including server-side frameworks such as **node.js** [N13], but its roots are found in C/UNIX systems that we'll discuss below.

The problem that event-based concurrency addresses is two-fold. The first is that managing concurrency correctly in multi-threaded applications can be challenging; as we've discussed, missing locks, deadlock, and other nasty problems can arise. The second is that in a multi-threaded application, the developer has little or no control over what is scheduled at a given moment in time; rather, the programmer simply creates threads and then hopes that the underlying OS schedules them in a reasonable manner across available CPUs. Given the difficulty of building a general-purpose scheduler that works well in all cases for all workloads, sometimes the OS will schedule work in a manner that is less than optimal. The crux:

THE CRUX:
HOW TO BUILD CONCURRENT SERVERS WITHOUT THREADS
How can we build a concurrent server without using threads, and thus retain control over concurrency as well as avoid some of the problems that seem to plague multi-threaded applications?

33.1 The Basic Idea: An Event Loop

The basic approach we'll use, as stated above, is called **event-based concurrency**. The approach is quite simple: you simply wait for something (i.e., an "event") to occur; when it does, you check what type of

event it is and do the small amount of work it requires (which may include issuing I/O requests, or scheduling other events for future handling, etc.). That's it!

Before getting into the details, let's first examine what a canonical event-based server looks like. Such applications are based around a simple construct known as the **event loop**. Pseudocode for an event loop looks like this:

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

It's really that simple. The main loop simply waits for something to do (by calling `getEvents()` in the code above) and then, for each event returned, processes them, one at a time; the code that processes each event is known as an **event handler**. Importantly, when a handler processes an event, it is the only activity taking place in the system; thus, deciding which event to handle next is equivalent to scheduling. This explicit control over scheduling is one of the fundamental advantages of the event-based approach.

But this discussion leaves us with a bigger question: how exactly does an event-based server determine which events are taking place, in particular with regards to network and disk I/O? Specifically, how can an event server tell if a message has arrived for it?

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

With that basic event loop in mind, we next must address the question of how to receive events. In most systems, a basic API is available, via either the `select()` or `poll()` system calls.

What these interfaces enable a program to do is simple: check whether there is any incoming I/O that should be attended to. For example, imagine that a network application (such as a web server) wishes to check whether any network packets have arrived, in order to service them. These system calls let you do exactly that.

Take `select()` for example. The manual page (on Mac OS X) describes the API in this manner:

```
int select(int nfds,
           fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict errorfds,
           struct timeval *restrict timeout);
```

The actual description from the man page: *select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, are ready for writing, or have*

ASIDE: **BLOCKING VS. NON-BLOCKING INTERFACES**
Blocking (or **synchronous**) interfaces do all of their work before returning to the caller; non-blocking (or **asynchronous**) interfaces begin some work but return immediately, thus letting whatever work that needs to be done get done in the background.

The usual culprit in blocking calls is I/O of some kind. For example, if a call must read from disk in order to complete, it might block, waiting for the I/O request that has been sent to the disk to return.

Non-blocking interfaces can be used in any style of programming (e.g., with threads), but are essential in the event-based approach, as a call that blocks will halt all progress.

an exceptional condition pending, respectively. The first nfds descriptors are checked in each set, i.e., the descriptors from 0 through nfds-1 in the descriptor sets are examined. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.

A couple of points about `select()`. First, note that it lets you check whether descriptors can be *read* from as well as *written* to; the former lets a server determine that a new packet has arrived and is in need of processing, whereas the latter lets the service know when it is OK to reply (i.e., the outbound queue is not full).

Second, note the timeout argument. One common usage here is to set the timeout to NULL, which causes `select()` to block indefinitely, until some descriptor is ready. However, more robust servers will usually specify some kind of timeout; one common technique is to set the timeout to zero, and thus use the call to `select()` to return immediately.

The `poll()` system call is quite similar. See its manual page, or Stevens and Rago [SR05], for details.

Either way, these basic primitives give us a way to build a non-blocking event loop, which simply checks for incoming packets, reads from sockets with messages upon them, and replies as needed.

33.3 Using `select()`

To make this more concrete, let's examine how to use `select()` to see which network descriptors have incoming messages upon them. Figure 33.1 shows a simple example.

This code is actually fairly simple to understand. After some initialization, the server enters an infinite loop. Inside the loop, it uses the `FD_ZERO()` macro to first clear the set of file descriptors, and then uses `FD_SET()` to include all of the file descriptors from `minFD` to `maxFD` in the set. This set of descriptors might represent, for example, all of the net-

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main(void) {
8      // open and set up a bunch of sockets (not shown)
9      // main loop
10     while (1) {
11         // initialize the fd_set to all zero
12         fd_set readFDs;
13         FD_ZERO(&readFDs);
14
15         // now set the bits for the descriptors
16         // this server is interested in
17         // (for simplicity, all of them from min to max)
18         int fd;
19         for (fd = minFD; fd < maxFD; fd++)
20             FD_SET(fd, &readFDs);
21
22         // do the select
23         int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25         // check which actually have data using FD_ISSET()
26         int fd;
27         for (fd = minFD; fd < maxFD; fd++)
28             if (FD_ISSET(fd, &readFDs))
29                 processFD(fd);
30     }
31 }
```

Figure 33.1: **Simple Code Using `select()`**

work sockets to which the server is paying attention. Finally, the server calls `select()` to see which of the connections have data available upon them. By then using `FD_ISSET()` in a loop, the event server can see which of the descriptors have data ready and process the incoming data.

Of course, a real server would be more complicated than this, and require logic to use when sending messages, issuing disk I/O, and many other details. For further information, see Stevens and Rago [SR05] for API information, or Pai et. al or Welsh et al. for a good overview of the general flow of event-based servers [PDZ99, WCB01].

33.4 Why Simpler? No Locks Needed

With a single CPU and an event-based application, the problems found in concurrent programs are no longer present. Specifically, because only one event is being handled at a time, there is no need to acquire or release locks; the event-based server cannot be interrupted by another thread because it is decidedly single threaded. Thus, concurrency bugs common in threaded programs do not manifest in the basic event-based approach.

TIP: DON'T BLOCK IN EVENT-BASED SERVERS
Event-based servers enable fine-grained control over scheduling of tasks. However, to maintain such control, no call that blocks the execution the caller can ever be made; failing to obey this design tip will result in a blocked event-based server, frustrated clients, and serious questions as to whether you ever read this part of the book.

33.5 A Problem: Blocking System Calls

Thus far, event-based programming sounds great, right? You program a simple loop, and handle events as they arise. You don't even need to think about locking! But there is an issue: what if an event requires that you issue a system call that might block?

For example, imagine a request comes from a client into a server to read a file from disk and return its contents to the requesting client (much like a simple HTTP request). To service such a request, some event handler will eventually have to issue an `open()` system call to open the file, followed by a series of `read()` calls to read the file. When the file is read into memory, the server will likely start sending the results to the client.

Both the `open()` and `read()` calls may issue I/O requests to the storage system (when the needed metadata or data is not in memory already), and thus may take a long time to service. With a thread-based server, this is no issue: while the thread issuing the I/O request suspends (waiting for the I/O to complete), other threads can run, thus enabling the server to make progress. Indeed, this natural **overlap** of I/O and other computation is what makes thread-based programming quite natural and straightforward.

With an event-based approach, however, there are no other threads to run: just the main event loop. And this implies that if an event handler issues a call that blocks, the *entire* server will do just that: block until the call completes. When the event loop blocks, the system sits idle, and thus is a huge potential waste of resources. We thus have a rule that must be obeyed in event-based systems: no blocking calls are allowed.

33.6 A Solution: Asynchronous I/O

To overcome this limit, many modern operating systems have introduced new ways to issue I/O requests to the disk system, referred to generically as **asynchronous I/O**. These interfaces enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed; additional interfaces enable an application to determine whether various I/Os have completed.

For example, let us examine the interface provided on Mac OS X (other systems have similar APIs). The APIs revolve around a basic structure,

the `struct aiocb` or **AIO control block** in common terminology. A simplified version of the structure looks like this (see the manual pages for more information):

```
struct aiocb {
    int             aio_fildes;          /* File descriptor */
    off_t           aio_offset;          /* File offset */
    volatile void   *aio_buf;            /* Location of buffer */
    size_t          aio_nbytes;          /* Length of transfer */
};
```

To issue an asynchronous read to a file, an application should first fill in this structure with the relevant information: the file descriptor of the file to be read (`aio_fildes`), the offset within the file (`aio_offset`) as well as the length of the request (`aio_nbytes`), and finally the target memory location into which the results of the read should be copied (`aio_buf`).

After this structure is filled in, the application must issue the asynchronous call to read the file; on Mac OS X, this API is simply the **asynchronous read** API:

```
int aio_read(struct aiocb *aiocbp);
```

This call tries to issue the I/O; if successful, it simply returns right away and the application (i.e., the event-based server) can continue with its work.

There is one last piece of the puzzle we must solve, however. How can we tell when an I/O is complete, and thus that the buffer (pointed to by `aio_buf`) now has the requested data within it?

One last API is needed. On Mac OS X, it is referred to (somewhat confusingly) as `aio_error()`. The API looks like this:

```
int aio_error(const struct aiocb *aiocbp);
```

This system call checks whether the request referred to by `aiocbp` has completed. If it has, the routine returns success (indicated by a zero); if not, EINPROGRESS is returned. Thus, for every outstanding asynchronous I/O, an application can periodically **poll** the system via a call to `aio_error()` to determine whether said I/O has yet completed.

One thing you might have noticed is that it is painful to check whether an I/O has completed; if a program has tens or hundreds of I/Os issued at a given point in time, should it simply keep checking each of them repeatedly, or wait a little while first, or ... ?

To remedy this issue, some systems provide an approach based on the **interrupt**. This method uses UNIX **signals** to inform applications when an asynchronous I/O completes, thus removing the need to repeatedly ask the system. This polling vs. interrupts issue is seen in devices too, as you will see (or already have seen) in the chapter on I/O devices.

ASIDE: **UNIX SIGNALS**

A huge and fascinating infrastructure known as **signals** is present in all modern UNIX variants. At its simplest, signals provide a way to communicate with a process. Specifically, a signal can be delivered to an application; doing so stops the application from whatever it is doing to run a **signal handler**, i.e., some code in the application to handle that signal. When finished, the process just resumes its previous behavior.

Each signal has a name, such as **HUP** (hang up), **INT** (interrupt), **SEGV** (segmentation violation), etc; see the manual page for details. Interestingly, sometimes it is the kernel itself that does the signaling. For example, when your program encounters a segmentation violation, the OS sends it a **SIGSEGV** (prepending **SIG** to signal names is common); if your program is configured to catch that signal, you can actually run some code in response to this erroneous program behavior (which can be useful for debugging). When a signal is sent to a process not configured to handle that signal, some default behavior is enacted; for SEGV, the process is killed.

Here is a simple program that goes into an infinite loop, but has first set up a signal handler to catch SIGHUP:

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

You can send signals to it with the **kill** command line tool (yes, this is an odd and aggressive name). Doing so will interrupt the main while loop in the program and run the handler code `handle()`:

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

There is a lot more to learn about signals, so much that a single page, much less a single chapter, does not nearly suffice. As always, there is one great source: Stevens and Rago [SR05]. Read more if interested.

In systems without asynchronous I/O, the pure event-based approach cannot be implemented. However, clever researchers have derived methods that work fairly well in their place. For example, Pai et al. [PDZ99] describe a hybrid approach in which events are used to process network packets, and a thread pool is used to manage outstanding I/Os. Read their paper for details.

33.7 Another Problem: State Management

Another issue with the event-based approach is that such code is generally more complicated to write than traditional thread-based code. The reason is as follows: when an event handler issues an asynchronous I/O, it must package up some program state for the next event handler to use when the I/O finally completes; this additional work is not needed in thread-based programs, as the state the program needs is on the stack of the thread. Adya et al. call this work **manual stack management**, and it is fundamental to event-based programming [A+02].

To make this point more concrete, let's look at a simple example in which a thread-based server needs to read from a file descriptor (`fd`) and, once complete, write the data that it read from the file to a network socket descriptor (`sd`). The code (ignoring error checking) looks like this:

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

As you can see, in a multi-threaded program, doing this kind of work is trivial; when the `read()` finally returns, the code immediately knows which socket to write to because that information is on the stack of the thread (in the variable `sd`).

In an event-based system, life is not so easy. To perform the same task, we'd first issue the read asynchronously, using the AIO calls described above. Let's say we then periodically check for completion of the read using the `aio_error()` call; when that call informs us that the read is complete, how does the event-based server know what to do?

The solution, as described by Adya et al. [A+02], is to use an old programming language construct known as a **continuation** [FHK84]. Though it sounds complicated, the idea is rather simple: basically, record the needed information to finish processing this event in some data structure; when the event happens (i.e., when the disk I/O completes), look up the needed information and process the event.

In this specific case, the solution would be to record the socket descriptor (`sd`) in some kind of data structure (e.g., a hash table), indexed by the file descriptor (`fd`). When the disk I/O completes, the event handler would use the file descriptor to look up the continuation, which will return the value of the socket descriptor to the caller. At this point (finally), the server can then do the last bit of work to write the data to the socket.

33.8 What Is Still Difficult With Events

There are a few other difficulties with the event-based approach that we should mention. For example, when systems moved from a single CPU to multiple CPUs, some of the simplicity of the event-based approach disappeared. Specifically, in order to utilize more than one CPU, the event server has to run multiple event handlers in parallel; when doing so, the usual synchronization problems (e.g., critical sections) arise, and the usual solutions (e.g., locks) must be employed. Thus, on modern multicore systems, simple event handling without locks is no longer possible.

Another problem with the event-based approach is that it does not integrate well with certain kinds of systems activity, such as **paging**. For example, if an event-handler page faults, it will block, and thus the server will not make progress until the page fault completes. Even though the server has been structured to avoid *explicit* blocking, this type of *implicit* blocking due to page faults is hard to avoid and thus can lead to large performance problems when prevalent.

A third issue is that event-based code can be hard to manage over time, as the exact semantics of various routines changes [A+02]. For example, if a routine changes from non-blocking to blocking, the event handler that calls that routine must also change to accommodate its new nature, by ripping itself into two pieces. Because blocking is so disastrous for event-based servers, a programmer must always be on the lookout for such changes in the semantics of the APIs each event uses.

Finally, though asynchronous disk I/O is now possible on most platforms, it has taken a long time to get there [PDZ99], and it never quite integrates with asynchronous network I/O in as simple and uniform a manner as you might think. For example, while one would simply like to use the `select()` interface to manage all outstanding I/Os, usually some combination of `select()` for networking and the AIO calls for disk I/O are required.

33.9 Summary

We've presented a bare bones introduction to a different style of concurrency based on events. Event-based servers give control of scheduling to the application itself, but do so at some cost in complexity and difficulty of integration with other aspects of modern systems (e.g., paging). Because of these challenges, no single approach has emerged as best; thus, both threads and events are likely to persist as two different approaches to the same concurrency problem for many years to come. Read some research papers (e.g., [A+02, PDZ99, vB+03, WCB01]) or better yet, write some event-based code, to learn more.

References

[A+02] "Cooperative Task Management Without Manual Stack Management"
Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur
USENIX ATC '02, Monterey, CA, June 2002
This gem of a paper is the first to clearly articulate some of the difficulties of event-based concurrency, and suggests some simple solutions, as well explores the even crazier idea of combining the two types of concurrency management into a single application!

[FHK84] "Programming With Continuations"
Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker
In Program Transformation and Programming Environments, Springer Verlag, 1984
The classic reference to this old idea from the world of programming languages. Now increasingly popular in some modern languages.

[N13] "Node.js Documentation"
By the folks who build node.js
Available: http://nodejs.org/api/
One of the many cool new frameworks that help you readily build web services and applications. Every modern systems hacker should be proficient in frameworks such as this one (and likely, more than one). Spend the time and do some development in one of these worlds and become an expert.

[O96] "Why Threads Are A Bad Idea (for most purposes)"
John Ousterhout
Invited Talk at USENIX '96, San Diego, CA, January 1996
A great talk about how threads aren't a great match for GUI-based applications (but the ideas are more general). Ousterhout formed many of these opinions while he was developing Tcl/Tk, a cool scripting language and toolkit that made it 100x easier to develop GUI-based applications than the state of the art at the time. While the Tk GUI toolkit lives on (in Python for example), Tcl seems to be slowly dying (unfortunately).

[PDZ99] "Flash: An Efficient and Portable Web Server"
Vivek S. Pai, Peter Druschel, Willy Zwaenepoel
USENIX '99, Monterey, CA, June 1999
A pioneering paper on how to structure web servers in the then-burgeoning Internet era. Read it to understand the basics as well as to see the authors' ideas on how to build hybrids when support for asynchronous I/O is lacking.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
Once again, we refer to the classic must-have-on-your-bookshelf book of UNIX systems programming. If there is some detail you need to know, it is in here.

[vB+03] "Capriccio: Scalable Threads for Internet Services"
Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer
SOSP '03, Lake George, New York, October 2003
A paper about how to make threads work at extreme scale; a counter to all the event-based work ongoing at the time.

[WCB01] "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services"
Matt Welsh, David Culler, and Eric Brewer
SOSP '01, Banff, Canada, October 2001
A nice twist on event-based serving that combines threads, queues, and event-based hanlding into one streamlined whole. Some of these ideas have found their way into the infrastructures of companies such as Google, Amazon, and elsewhere.