**Chapter 33: Event-based Concurrency (Advanced)**

Different styles of concurrency are often used in GUI-based applications and internet server. They are **event-based concurrency**.

There are two problems. The first is managing concurrency in multi-threaded application. The second is that developers have no control or little control over what is scheduled.

* How can we build a concurrency server without using threads?

**33.1 The Basic Idea: An Event Loop**

The basic approach is **event-based concurrency**. The approach is that we simply wait for something to occur and then we check the type of event and do the small amount of work it requires.

Event-based server are based on a simple construct known as the **event loop**:

Text, letter

Description automatically generated

The piece of code that processes each event is known as **event handler**. When the handler handles an event, it is the only activity taking place in the system (explicit control).

However, how does this server determine which events take place?

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

The most basic APIs available are select() or poll() system calls.

The select() system call:

Text

Description automatically generated

It takes the I/O descriptor sets as readfds, writefds and errorfds to examines to see if their descriptors are ready for reading, writing or error. The first nfds is checked in each set. 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.

Select() allows us to check whether descriptors can be read from as well as written to. The read descriptor lets a server determine that a new package arrived and need to be processed. The write descriptor lets the service know when it is OK to reply.

We can also set the timeout argument to NULL to make select() block indefinitely until some descriptors are ready. We can set it to zero to make it return immediately.

The poll() system call is quite similar. Either way, these basic system calls give us a way to build a non-blocking event loop.

**33.3 Using select()**

Text, letter

Description automatically generated

Inside the loop, FD\_ZERO() is used to clear the set of gile descriptors. Then, FD\_SET() is used to include all of the file descriptors from minFD to maxFD. This set of descriptors represent all of the network sockets to which the server is paying attention. Finally, the server calls select() to know which of the connections have data available upon them. It also uses FD\_ISSET to see which descriptors have data ready and process the incoming data.

**33.4 Why Simpler? No Locks Needed**

Because only one event is being handled at a time, there is no need for a lock. In addition, it cannot be interrupted since it is decided by a single thread.

**33.5 A Problem: Blocking System Calls**

A problem is that what id an event requires that we issue a system call that might block?

For example, what if a client requests the server to read a file from disk. Thus, while the thread issuing the I/O request suspends, other threads can run and the server make progress. However, in our event-based approach, there are no other threads to run. Thus, this makes the system idle, creating a huge potential waste of resource.

Thus, no blocking calls are allowed.

**33.6 A Solution: Asynchronous I/O**

We will have a new way to issue I/O requests to the disk system, which is **asynchronous I/O.** This enable an application to issue an I/O request and return control immediately to the caller, before the I/O is completed.

For example, the **AIO control block**:

Text, letter

Description automatically generated

To asynchronously read to a file, we must pass the file descriptor of the file to be read (aio\_fildes), the offset in the file (aio\_offset) and the length of the request (aio\_nbytes). We also need to pass the target memory location where the results of the read should be copied into (aio\_buf).

After that, to asynchronously read a file, we call:



This simply tries to issue the I/O. If successful, it returns immediately to the application.

Lastly, we need to handle errors:



This simply checks whether the request referred to by aiocbp has completed. If it has, the routine returns success, otherwise it returns EINPROGRESS. 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.

However, checking for completion of I/O is painful. To solve this, some systems provide an approach based on the **interrupt**, which uses UNIX signals to inform applications when asynchronous I/O completes.

**33.7 Another Problem: State Management**

This approach has another problem as such code is generally more complicated to write than traditional thread-based code. 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 program. We call this **manual stack management**.

For example, 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*:

Text

Description automatically generated with medium confidence

In a thread-based system, it is very easy to complete. However, in an event-based system, we will first have to asynchronously read. Then, we check for completion of the read. But how does it know what to do?

To do this, we need to record the needed information to finish processing this event in some data structure. When the I/O completes, we simply look up the needed information and process the event.

**33.8 What Is Still Difficult With Events**

The first problem is when we have to work on multiple CPUs, some of the simplicity of the event-based approach disappeared as we have to do things in parallel, so implementing locks or critical sections is a must.

Another problem with the event-based approach is that it does not integrate well with certain kinds of systems activity, such as paging, as it will block on page fault.

The third problem is that it is hard to manage over time.

Finally, though asynchronous disk I/O is now possible on most platforms, it has taken a long time to get there, and it never quite integrates with asynchronous network I/O in as simple and uniform a manner.

**Chapter 34: Summary**

Avoid complex interactions between threads and use well-known and tried-and-true ways to manage thread interactions.

Only use concurrency when absolutely needed. Avoid it if possible.

if you really need parallelism, seek it in other simplified forms.