BrianHetrick.com
Application Note AN-8

The Event-Based Asynchronous Pattern in .NET

Contents

Goal

This Application Note describes the asynchronous event-based pattern in .NET. It discusses the pattern itself, the properties of the pattern including the circumstances when it is useful, and implementation techniques for using the pattern. The .ZIP file accompanying this Application Note includes both a sample program using the pattern and an abstract base class useful in implementing the pattern. The Application Note also describes how to use the abstract base class.

Introduction

Current computer architectures emphasize an ever-increasing number of execution units (cores) as the primary method of increasing system throughput, a design called multiprocessing. Further, many systems allow configurations that increase the number of effective cores with an overall beneficial trade-off of the effective cores' peak speeds, a technique called hyperthreading. To exploit such architectures, applications must use multiple simultaneous flows of control. This can take the form of using multiple simultaneously executing programs, each in a separate operating system process. It can also take the form of using multiple simultaneous control flows (threads) within a single program and process, a technique called multithreading. Multiprocessing typically requires inter-process communication, as data must be explicitly transferred between processes. Inter-process communication requires data to be serialized into some intermediate form, using both storage and processing capacity; transferred, typically using I/O capacity; and finally deserialized, again using both storage and processing capacity. Multithreading, in contrast, exploits the fact that the individual threads share a common address space. Only references to data (rather than the data itself) need to be transferred from one thread to another. Conversion of data to and from an external form is completely avoided.

The desire for responsive user interfaces also represents a pressure towards multithreading. User interface implementation has settled into an event-driven pattern; without multithreading, updating the user interface waits on the completion of processing prior events. The application becomes unresponsive when processing a lengthy operation (for example, copying a large file). One response to this situation is to create threads for such lengthy operations, and treat the operation completion as an event. This permits the user interface to continue handling events as the operation progresses independently.

Multithreading is hard. Race conditions, deadly embraces in resource allocation, the need to lock shared resources which must always present a consistent state, the possibility of deadlock in lock sets with cyclic dependency directed graphs, and so forth all contribute to the difficulty of achieving program correctness. Multithreaded programs have failure modes and mechanisms that simply do not arise in sequential programs, and for which our intuitions and habitual thought patterns leave us unprepared. Computer science has developed a powerful calculus of communicating processes which allows proof of program correctness. However, typical applications are developed over time, without the up-front analysis and design care needed to use this calculus. It is therefore necessary to use mechanisms which are robust in the face of even radical design changes caused by application maintenance and enhancement.

The .NET framework includes the Task-Based Asynchronous Programming model, introduced in .NET 4. The task based asynchronous programming model permits the simple expression of a number of operations used in multithreading designs. Even with simplified expression, multithreading remains a rich source of program errors. This Application Note examines an older technology, the Event-Based Asynchronous Pattern, which provides an easier and more robust (although somewhat more limited) approach to correct multithreading.

The .NET framework provides the `BackgroundWorker` class which encapsulates much of the mechanism required for a non-reentrant operation provider. This Application Note provides and describes an `AsynchronousOperationProviderBase` abstract class which encapsulates much of the mechanism required for a reentrant operation provider. The distinction between these is explained in the next section.

The Pattern and Its Properties

The event-based asynchronous pattern relies on an object which can be called the operation provider. The provider exposes possible operations through methods conventionally named with the suffix Async. A client to the operation provider provides a description of the desired operation to the operation method and receives an event notification when the operation completes. The event is conventionally named with the suffix Completed, and the event arguments is conventionally an `AsyncCompletedEventArgs` object. The client may cancel operations that have been requested but not yet completed. The completed event may reveal successful operation completion, operation failure through an unhandled exception having arisen, or operation cancellation.

Operation providers typically come in one of two forms: reentrant and non-reentrant. A non-reentrant operation provider supports only a single asynchronous operation at a time. Non-reentrant operation providers conventionally expose an `IsBusy` property, which indicates whether an asynchronous operation is currently in process. Non-reentrant operation providers reject requests to start an operation when `IsBusy` is true. A reentrant operation provider supports multiple asynchronous operations concurrently. As clients of reentrant operation providers need to distinguish the operations from one another—a completed event indicates an operation completed, but which operation?—operations are identified by a client-provided object called the *user state*. The user state is referenced in events published by the asynchronous operations. This permits the clients of the operation provider to identify which operation is being reported upon. To avoid confusion, the user state object should be distinct for each operation. Typically reentrant operation providers enforce this uniqueness constraint. The user state is conventionally the last parameter to the Async method starting the operation, coming after any parameters required to specify the operation to be performed.

Operations may optionally report their progress towards completion. This conventionally is done through an event named `ProgressChanged`, and the event arguments value is conventionally a `ProgressChangedEventArgs` object. The `ProgressChangedEventArgs` class includes a `ProgressPercentage` property which conventionally is used to indicate on a scale from 0 (not yet started) to 100 (complete) the progress of the operation.

The `AsyncCompletedEventArgs` and `ProgressChangedEventArgs` classes provided by the .NET framework do not include values possibly returned by the operation. Specialization of these classes on a per-operation basis can provide such values.

The primary, although not particularly major, complication in implementing an event-based asynchronous operation is that events are published on behalf of one thread, that of the operation, but must be received on a thread compatible with that of the client. This is ar-

ranged through a .NET framework SynchronizationContext object, typically provided on a per-client thread basis through an AsyncOperation wrapper by the AsyncOperationManager class. The WinForms and WPF frameworks provide SynchronizationContext implementations that converts the event publication into a delegate invocation event on the application event queue, where it is handled on the UI thread. In the absence of a per-thread SynchronizationContext object (as with threads that do not make other arrangements), the client must be able to handle an event delivered on a foreign thread. The WinForms Control.Invoke and WPF Dispatcher.Invoke explicit synchronization methods can also be used to convert a method invocation on a foreign thread into the corresponding event on the UI thread. These synchronization methods are not needed in the common case when the asynchronous operation provider is exposed only to UI thread objects.

The event-based asynchronous pattern is a powerful, convenient method of providing asynchronous operations to an application. It does, however, have a drawback: bi-directional communication between the operation and the operation's client is cumbersome. Event-based asynchronous operations are typically "fire and forget," rather than "start and interact" as with some other asynchronous operation schemes. Also typically, the operation must be capable of being described completely in advance, and the primary mechanism the operation has to deal with unexpected circumstances is to throw an exception (which is reported to the client in the completed event arguments).

However, event publication in fact provides bidirectional communication: the event handler can modify the event arguments object to pass information back to the event publisher. The .NET framework uses this capability in, for example, the Form.FormClosing event and the AppDomain.AssemblyResolve event. This arrangement requires the client to be responsive to the operation rather than the other way around, but is entirely usable for request/reply communications between the operation and the client. Event publication used in this way provides callback functionality in the overall request/reply communication between the client and the operation. However, asynchronous operations requiring frequent or high-transparency communications with their clients are best implemented using other models.

Implementation Techniques

A number of techniques are applicable to the implementation of the event-based asynchronous pattern. These techniques are discussed in this section.

Avoiding Race Conditions

Programmers used to event-driven frameworks are used to event handlers (such as a _Click event handler) being effectively atomic: once the event handler is entered, nothing external changes the program state until the event handler exits. In the event-driven world, things happen only between events. Carrying forward this expectation into asynchronous operations typically results in race conditions. For example, a client of an asynchronous operation provider might be written like this:

```
OperationProvider provider =
   new OperationProvider ();
provider.DoOperationAsync (null);
provider.DoOperationCompleted +=
   Provider_DoOperationCompleted;
```

The intention is to start an operation and be notified when it completes. However, this code has a race condition: there is no guarantee that the completed event will be received. The completed event will be published after the DoOperationAsync invocation, but that does not mean after the event subscription of the next line of code. The operation could complete and publish its completed event immediately (as perceived by the client) before the completed event subscriber has been added. The required order is to subscribe to the completed event, then start the operation.

Similarly, if a part of the program is responsible for doing something, it needs to ensure the something has been done rather than has been requested. For example, an asynchronous operation may transmit data to a network server. That the data transmission has been requested does not mean it has completed. Unless the completed event for an operation has been received, exiting the program or discarding the reference to the operation provider may make the operation vanish, never to be performed.

Race conditions typically exhibit as Heisenbugs: rare occurrences that cannot be reproduced. These are typically extremely difficult bugs to locate.

Scheduling on the Thread Pool

A primary requirement of the event-based asynchronous pattern is asynchrony: the operation proceeds independently of its requestor. This requirement is met through the use of a separate thread. The .NET framework maintains a pool of threads, creatively called the thread pool, on which both short- and long-duration operations can be scheduled. The `ThreadPool` static class provides access to the thread pool. The `ThreadPool.QueueUserWorkItem` method causes a thread pool thread to execute a delegate. Using the thread pool, instead of a separately created `Thread` object, has the advantage of letting the .NET framework handle lifetime management of the underlying Thread object.

Thread pool threads are background threads. The existence of background threads does not keep the application from exiting. In contrast, the existence of foreground threads does keep the application from exiting. Asynchronous operations in progress on thread pool threads do not keep the application active if the main thread exits. Typically this is a desirable property. If this is undesirable, the main thread can wait for all active operations to publish their completed events before exiting.

The delegate executed by the `QueueUserWorkItem` method accepts an object parameter, which is passed to the `QueueUserWorkItem` method along with the delegate. In the event-based asynchronous pattern, this parameter object is used to pass the description of the operation to be performed to the delegate.

Linking User State and Operation Instances

The user state object identifies a particular operation. Each event published by the operation must reference the user state object so event subscribers can determine the identity of the operation to which the event refers. Typically event subscribers are interested in only one operation or only a limited number of operations. Event subscribers therefore must first check the user state object, and then process the event only if it refers to an operation in which they are interested.

The operation provider itself must keep a record of what user states are in use. Cancellation requests provide the user state corresponding to the operation to be canceled. In order to find the corresponding operation to cancel, the operation provider must translate the user state to its corresponding operation. Typically this involves a `Dictionary` keyed by the user state. There is a minor wrinkle in using a `Dictionary` for this purpose, in that null is a perfectly acceptable user state but cannot be used as the key of a `Dictionary` entry. The operation provider must therefore create a surrogate object and use this object when the provider's clients specify null as a user state for an operation. The uniqueness constraint for user states can be enforced by referencing the `Dictionary`: if the user state for a requested operation exists as a key in the `Dictionary`, the request is rejected. Note that this implies the `Equals` method of the object used as a user state must actually work. In order for the `Dictionary` to perform well, the `GetHashCode` method of the object used as a user state must also actually work.

Operation Cancellation

Clients of the asynchronous operation provider can request that a particular operation be canceled. Cancellation is not guaranteed to have any effect: the operation is free to ignore it. Although ignoring a cancellation request may be regarded as somewhat hostile, it is a valid response. Also, the operation may have completed between the cancellation request being made and the operation provider relaying the request to the operation itself. For this reason, it is not an error for a cancellation request to reference a user object that is not in use.

A cancellation request is typically represented in the object representing the operation itself as a `bool`. During whatever iterations it performs, the operation checks the `bool` and exits if the `bool` indicates a cancellation request. Operations that use wait handles may find it more convenient to represent the cancellation request as an event, perhaps a `ManualResetEvent`. The operation can then use the `WaitHandle.WaitAny` method to wait on the intended wait handle and the cancellation wait handle simultaneously, and take appropriate action based on which handle signaled. Alternately, operations using wait handles might wait with short timeouts (for example, 250 milliseconds), and iterate this until either the cancellation flag becomes set or the wait handle signals.

Some applications may want to cancel all unfinished operations. This might occur when the operations are effectively anonymous (as when, for example, a network service provider discards references to the operations' user state objects as it is uninterested when the operations complete) or when an application must terminate quickly (as with a service ordered to stop, or as with an application presented with a shutdown event

from the operating system). Providers may therefore want to provide a method to cancel all existing operations.

Determining Progress

Some operations are by design "forever," such as an operation providing a network service. The concept of "progress toward completion" is not particularly meaningful for such operations. Other operations may have difficulty estimating progress towards completion. For example, a file copy operation might report progress based on either file count or byte count, but either requires the total amount of work to be estimated before any meaningful progress can be indicated. Operations may find it impossible to estimate progress. For example, a large number factorization operation cannot estimate when completion may occur: instead it knows only whether or not it is done. But some operations can estimate progress, and for some of those operations progress reporting may be important.

Progress is typically reported through a `ProgressChanged` event, which reports progress as a percentage of completion.

Asynchronous and Synchronous Publication

Event publication in .NET has a few subtle points. Cross-thread (asynchronous) event publication has a few more.

Event publication is based on delegates. Delegates have several methods. These methods add or remove targets, invoke the targets *en masse*, get a list of the current targets, and so forth. Declaring a delegate member `public event` in a class is a compiler trick that makes public only the methods for adding and removing targets. Because of this compiler trick, delegates declared `public event` are useful to maintain lists of event subscribers: other objects can add delegates to and remove delegates from the list, but cannot publish events as if they were the object exposing the delegate.

Conventionally, event handlers (the methods invoked through an event) accept two parameters. The first is a general object, which is the event publisher. The second is an `EventArgs` object that describes the event of which the handler is being notified. The `EventArgs` object itself has no members: events providing data in addition to notification of occurrence must define an event arguments class derived from `EventArgs` and pass that to event subscribers.

Events also have some expectations that delegates in general do not. Primary among these expectations is that when an event occurs, subscribers to the event are notified. In particular, an event-based asynchronous operation's Completed event must be delivered. When a delegate is invoked simply (by using the delegate as if it were a method), the targets are invoked sequentially and passed the event arguments. However, should one target throw an exception, delegate invocation ends: targets not yet invoked are never invoked. This breaks the expectation that subscribers are actually notified of the event.

Also, a delegate with no targets is `null`. A `null` delegate cannot be invoked: attempting to do so will throw a null reference exception. Thus one is advised to check delegates before invoking them to ensure they are not `null`. However, doing so creates a race condition: the delegate may become `null` between the check and the invocation. This can be avoided only by taking a separate reference to the delegate's current value: this reference cannot become `null` simply because the delegate itself becomes `null`.

Publishing an event therefore requires code *very* much like that of Display 1 (next page).

About the only wiggle room is what to do in the catch block when an event handler throws an exception: ignore the exception (as shown), log it, publish an event describing it, or something else. In the above method, the subscribers parameter serves as the separate reference to the delegate object.

(The code shown in Display 1 performs what is known as event synchronous publication. Event synchronous publication completes delivery to all subscribers before returning to the event publisher. Event synchronous publication also has the curious property that events published by different objects can be delivered to a subscriber in an order different than that in which they were published. A different method of publishing events, called event asynchronous publication, queues event publications and delivers them in the order in which they were published. Event asynchronous publication has the property that event argument objects typically cannot serve as an event handler to event publisher communication mechanism. Note that the phrase "event asynchronous publication" refers to the asynchronous publication of events, while the phrase "asynchronous event publication" refers to the publication of asynchronous events. The event-based asynchronous pattern deals with the latter.)

Some developers have the curious belief that throwing an exception in response to an event delivery is a way to object to whatever the event describes. For example, such developers may think throwing an exception in response to a `PropertyChanged` event is a way of aborting the property setter method and so the change. This is incorrect for two reasons: most importantly, an event delivery describes an occurrence in the past. The property has already changed, and the event subscriber is simply being notified after the fact. After all, should I mention to my friend that I bought a new computer, and my friend's response is to throw himself off a cliff, that does not somehow un-buy the new computer. Also, all throwing an exception potentially can do is to end the notification process. Again, my friend throwing himself off a cliff may make we cautious about telling other friends about my new computer, lest they too find themselves enamored of long falls; but again, it does not prevent me from telling other friends unless I wish it to. The Java platform has the concept of a `VetoableChangeListener`, where the Java equivalents to event subscribers can veto proposed changes to "constrained" properties through throwing a particular exception. Such an arrangement can be developed in .NET, of course, but the standard event semantics do not provide this functionality.

Cross-thread event publication in .NET, as noted before, involves a synchronization context object. The synchronization context provides two methods that invoke a delegate: `Post`, which does asynchronous invocation, and `Send`, which does synchronous invocation. The difference is whether the method waits for the invocation to occur and complete. Either, when invoked on a UI thread's synchronization context, inserts the delegate invocation into the UI event queue. `Post` returns immediately; `Send` waits for the invocation to occur. Cross-thread events published using `Post` cannot serve as bi-directional conduits without substantial further complexity. Cross-thread events using `Send` stall the publisher until the event is received and handled. The state of the event arguments object when `Send` returns is the state left by the event handler; this

```
public static void PublishEvent
   (object publisher, MulticastDelegate subscriberList, EventArgs eventArguments)
{
    if (publisher == null)
    {
        throw new ArgumentNullException (nameof (publisher));
    }
    if (eventArguments == null)
    {
        throw new ArgumentNullException (nameof (eventArguments));
    }

    if (subscriberList != null)
    {
        foreach (Delegate subscriber in subscriberList.GetInvocationList ())
        {
            try
            {
                subscriber.DynamicInvoke (publisher, eventArguments);
            }
            catch (Exception exc)
            {
                //  TODO: Log error in event handler
            }
        }
    }
}
```

Display 1. Event publication.

provides the subscriber-to-operation communication pathway of standard events.

The `Post` and `Send` methods each have two parameters: the `SendOrPostCallback` delegate to be invoked, and a single object to be passed to the delegate. Event handlers have two parameters: the publisher and the event arguments. The target invoked by the `Post` or `Send` methods therefore cannot be the event handler itself. Instead, the target invoked by the `Post` or `Send` methods must be a method which unpacks an event description (the single object) and publishes it. The use of a synchronization context in this way does not directly change the thread on which an event is delivered. It instead changes the thread on which the event is published, which in turn becomes the thread on which the event is delivered. Event delivery, as seen in the code above, is simply iterating over a delegate's invocation list. It does not involve threading operations.

Finally and again, in the event-based asynchronous pattern, the eventCompleted event for an operation must be published, no matter how the operation ended. Asynchronous operations might fail, but the failure must not be silent.

Synchronization Context Selection

Recall that the operation must use a `SynchronizationContext` to deliver its events, and the `SynchronizationContext` is thread-specific. The operation has a choice of synchronization context objects. The synchronization context can be that of the thread that created the operation provider object, or the thread that requested the operation. Although in many applications these are the same thread, they need not be.

If the operation provider is created by a UI component, using the provider creator's synchronization context for all operations guarantees that all events will be delivered on the UI thread. This implies event subscribers manipulating the UI need not be careful about thread safety: UI components receive event delivery on the UI thread. Non-UI components receive event delivery on an arbitrary thread just as they otherwise would; as it happens, the arbitrary thread is the UI thread. However this arrangement also has the disadvantage that non-UI thread subscribers must wait for the UI thread to be idle before they get their event deliveries.

Alternately, the operation can use the synchronization context of the thread that requested the operation. This is encouraged by the `AsyncEventManager` framework class, which requires a user context to create an `AsyncEvent` object (which is a wrapper for a `SynchronizationContext` object). The `AsyncEvent` object is described as tracking the lifetime of a particular operation, which further supports its per-operation nature. This arrangement permits non-UI thread subscribers to immediately receive event delivery from operations started by non-UI threads. However, this arrangement also requires the UI as a whole to be thread aware. The `Control.Invoke` or `Dispatcher.Invoke` method can be used by UI thread subscribers to ensure the event delivery itself is handled on the correct thread. However recall that one responsibility of the event handler is to determine whether the event is from an operation in which it is interested. This implies the data structure maintaining the list of user objects corresponding to interesting operations must be accessed in a thread-safe manner both at the event handler and everywhere else.

The Microsoft guidance on this selection is mixed. The example and tutorial Microsoft gives on the Event-Based Asynchronous Pattern uses a non-reentrant operation provider defined by and used only by the UI thread, so the user state object identifying the operation is superfluous, and the thread on which the `AsyncEvent` is created is irrelevant. As it happens, in the Microsoft example the `AsyncEvent` is created by the provider's constructor rather than the provider's start operation method. Because the provider is non-reentrant, only one `AsyncEvent` is needed; because the provider creation and operation requests all occur only on a single thread, where the single `AsyncEvent` is created makes no difference. Now, it might in some circumstances be wisest to use the synchronization context of the thread that created the provider, rather than that of the thread that created the operation. But this is not the pattern encouraged by the `AsyncEventManager` and `AsyncEvent` framework classes. Deviation from the convention of publishing completed and progress changed events on a thread compatible with that of the operation's requestor should be thoroughly justified and documented.

The complications for the UI arising from using the synchronization context of the operation requestor, rather than the operation provider creator, arise only if a non-UI thread requests operations on an operation provider also used by the UI thread. In the typical case, the operation provider object is created by the UI thread, operations are requested by the UI thread, and the operation provider is never exposed to other

threads. In this typical case, the UI thread event handlers are guaranteed events will be presented on the UI thread, and cross-thread event reception is not an issue.

Load Limiting and Operation Queuing

The point of a reentrant asynchronous operation provider is to allow multiple operation requests to be serviced simultaneously. However, multiple is only rarely the same as an unlimited number. Database management systems, for example, allow only up to a certain number of connections: should a program attempt the operation of "copy the table contents to a CSV file" for all tables simultaneously, the database management system may be unable to handle the number of connections required. This would cause some operations to fail. Similarly, should operations require substantial computational resources, attempting more simultaneous operations than the number of cores in the machine is relatively fruitless. It is frequently desirable to limit the number of simultaneous active operations based on the nature of the operation.

Such situations can be addressed by queuing operation requests, then servicing the requests in a rate-controlled manner.

This Application Note has already drawn a distinction between an operation provider and an operation. The need to limit the load imposed by a large number of operations requires a further distinction: one between the operation and the operation executor. As operation executor is a thread that performs operations.

To provide load control, the operation provider need not maintain any specific objects representing operation executors, but it must maintain at least a count. When an operation is requested, the provider queues the operation description and checks the current executor count against the limit. If the current provider count is less than the limit, it starts an executor and increments the current provider count.

Each executor removes an operation description from the queue and performs it, publishing completed events as described previously. The executor then performs this action again, repeatedly, as long as the operation queue has elements. When the operation queue is observed to be empty, the executor decrements the current provider count and returns to its caller. This in turn returns the thread to the thread pool.

Synchronization and operation ordering are important here. Incrementing the executor count is a promise by the operation provider that an executor will be started; decrementing the executor count is a promise by the executor that it will cease operation. The queue of operations must be accessed only after locking; similarly the `Dictionary` of user states and their corresponding operations; and similarly the executor count value. Executors must continue execution despite exceptions thrown by operations.

One might think that using the `Interlocked` class to manipulate the executor count would avoid the need for locking for at least this one value. Unfortunately this is not the case. The decision to create an executor, for example, is taken based on an observation of the executor count. The increment must occur atomically with respect to the observation. (Creating an executor need not be part or the atomic operation, but must occur once the increment has occurred.) Otherwise there is a race condition: `Interlocked.Increment` will indeed increment the counter value atomically, but the counter value incremented is not necessarily the counter value observed. An arbitrary number of executors could be created via this race condition. Decrementing the counter similarly depends on observing the state of the operation queue, and so similarly must be atomic with respect to the observation. An arbitrary number of executors could exit via this race condition. In particular, the operation provider could find itself with queued operations and no executors if this is not the case.

Sharing Resources Among Operation Instances

As mentioned in the previous section, the queue of operations, the `Dictionary` mapping user states to operations, and the executor count constitute state which must be manipulated atomically not only individually but as a whole. Some operations may require other state to be shared among executors: for example, an operation provider may offer a single HTTP connection which the operations use to fetch or store data as required, or a single database connection on which the operations create and use commands as required. In such circumstances, a synchronization object must be used to ensure only one entity at a time uses the shared resource.

The .NET framework provides several classes that may be useful in managing such resource sharing: `Monitor`, `Mutex`, `ReaderWriterLock`, and `Semaphore` are useful here. The C# `lock` statement is

implemented with `Monitor.Enter` and `Monitor.Exit`; these involve only the current process. `Mutex` and `Semaphore` objects are managed representations of kernel objects; named instances of these are visible system-wide and are typically used for inter-process (not inter-thread) synchronization. `Monitor` and `ReaderWriterLock` are .NET objects rather than kernel objects and so much lighter weight. `ReaderWriterLock` is to be preferred over `Monitor` (and so the `lock` statement) when most accesses do not change the value protected. Any use of these synchronization primitives must ensure correct behavior in the presence of exceptions. Typically acquiring the synchronization resource is immediately followed by a `try-finally` block: the `try` section performs the action of interest and the `finally` releases the synchronization resource. The C# `lock` statement generates such code automatically.

Waiting for Asynchronous Operations

The event-based asynchronous pattern is designed for use with event-driven clients. A temptation when using the event-based asynchronous pattern is to use a wait-based synchronization object, such as an `AutoResetEvent`, to indicate the completed event has been received; or to use a `bool` and loop using `Thread.Sleep()` until the `bool` becomes set. Then, when a client has reached a point where it needs the results of the asynchronous operation to continue, it waits on the event. This is typically poor practice, and can be disastrous in two specific circumstances.

The first disastrous circumstance is when the waiter is a UI thread. Should the UI thread wait in this way for an operation it has itself scheduled, the application will deadlock. The completed event must be delivered to the UI thread via a method invocation event on the UI event queue. The method invocation event will be dispatched only when the UI event loop reaches the queued invocation event. The UI event loop will reach the queued invocation event only when all preceding events in the queue have been dispatched and returned. The currently executing UI event has just issued a wait and will not return until the completed event has been delivered. This is a classic circular dependency and a classic deadlock. The UI thread must never wait on a wait handle set by the UI thread, and should in fact never wait at all.

There is a workaround if you absolutely must wait in the UI thread on a wait handle set by the completed event of an operation scheduled by the UI thread: request the desired asynchronous operation through an intermediate asynchronous operation. Define a second asynchronous operation for which the operation is to request the actual desired operation. The UI thread then subscribes to the completed event on the actual desired operation, but requests the secondary operation to be performed. The secondary operation will execute on a `ThreadPool` thread and schedule the desired operation; the completed event of secondary operation is ignored. When the desired operation completes, its completed event will be delivered on a thread compatible with the thread of the operation requestor, that is, the `ThreadPool` thread of the secondary operation. This will be an arbitrary thread—the way `SynchronizationContext` objects work, it will in fact be the `ThreadPool` thread that performed the desired operation. The completed event handler, operating on the thread that performed the desired operation, must then set the wait handle without touching anything else and in particular without using `Invoke` or `BeginInvoke`. This in turn will release the UI thread. This arrangement should be regarded as an odious hack and avoided if at all possible.

Note that in the odious hack described above, the second operation is defined as requesting the actually desires operation, not as performing it—or equivalently requesting it and waiting for its completed event. This is required because of the second disastrous circumstance, described next.

The second disastrous circumstance is when the waiter is itself another asynchronous operation on the same provider. The waiter, while waiting, consumes an executor thread, making it unavailable for other operations. The operation on which the waiter is waiting must therefore be run by a different executor. If the maximum executor count is capped, as with the load limiting approach described above, a free executor on which to execute the waited-for operation may never arise: all the executors are waiting for other operations to complete, and the other operations cannot be started. Strictly speaking this is not a deadlock, but a deadly embrace, as the resource required to continue is not a lock; unfortunately the results are the same. This situation also generalizes to cases where operations can wait on operations provided by another provider, whose operations wait on operations provided by still another provider, whose operations wait on operations provided by the original provider; and so forth.

The point of the event-based asynchronous pattern is to be event-based, not wait-based. Waiting for completed event of an independent operation, rather than simply handling it when it occurs, is best handled with other asynchronous models.

Instrumentation

Finally, it is often useful to provide performance instrumentation on asynchronous operations. Typically desired performance counters include a count of operations completed, the maximum and current number of executor threads, the total elapsed time taken by completed operations, and the total CPU time taken by completed operations.

Of these, all but the last are simple and obvious in their acquisition. Typically each executor can observe its thread's accumulated CPU time at the start and end of each operation, and add the difference to the total CPU time counter. On Windows, per-thread CPU time can be obtained from the Kernel32 `GetThreadTimes` function. The time values provided by this function are `FILETIME` structures, which are 4-byte aligned 8-byte counts of 100 nanosecond intervals. This is conveniently compatible with a .NET `TimeSpan` structure. Kernel date-times are relative to 1 January 1601 00:00:00 UTC (proleptic Gregorian calendar) and so can be easily converted to and from .NET `DateTime` structures which are relative to 1 January 1 00:00:00 (local or UTC depending on `DateTimeKind`) (proleptic Gregorian calendar). On Linux, thread CPU times can be obtained from the `getrusage` and `clock_gettime` kernel calls with a precision of up to microseconds.

However, 100 nanosecond or microsecond precision does not mean 100 nanosecond or microsecond accuracy. With the exception of the `Stopwatch` class, date-times and durations on most systems are accurate to only a scheduler quantum. The scheduler quantum is typically 10 to 20 milliseconds, but can vary depending on the system and even from time to time on the same system.

Further, CPU time measurement is complicated by threads waiting (and so releasing their core) before their quantum is up, and by threads being dispatched in the middle of a quantum when a core becomes available because some other thread waited. The accounting function has no way of knowing how much time was actually used by either thread in these cases. Finally, modern systems will adjust the CPU clock speed based on system load and CPU temperature. For power efficiency, the lighter the load on the system the slower the CPUs are clocked. Similarly, the high frequency CPU clock used for a peak load can only be sustained for a short time. A high CPU clock rate will increase CPU heat production, which will increase the CPU temperature, to which the system will respond by slowing the CPU clock. Because of this, a unit of CPU time is elastic: it does not measure any particular amount of work.

In summary, CPU time measurements on typical systems are both inaccurate and largely meaningless. For this reason, attempting to measure CPU time is not recommended. CPU time measurements, much like buggy whips, belong to a simpler, bygone age.

A Base Class for Operation Providers

The code provided with this Application Note includes an abstract base class for event-based asynchronous pattern operation providers. This class is called `AsyncOperationProviderBase`. This class provides a framework supporting the execution and management of asynchronous operations. The asynchronous operations themselves are implemented as descendants of a nested class `OperationBase`.

The Operation Provider Base

Public elements of the `AsyncOperationProviderBase` class are few: there is a `GetProviderCounters()` method to retrieve performance counters, a `Cancel(object userState)` method to cancel a single operation, a `CancelAll()` method to cancel all operations, and a nested public `ProviderCounters` class defining the performance counters returned. The `AsyncOperationProviderBase` base class does not define any operation methods. Instead descendant classes are responsible for defining operations and the interfaces to the operations.

Descendant classes must define an operationAsync method to start an operation and an operationCompleted event through which to publish completed events. The operation itself is implemented as a descendant of the nested protected abstract `OperationBase` class.

A typical operationAsync method consists of two lines of code. It creates and populates an appropriate descendant of the `OperationBase` class, and passes that descendant to the `StartOperation` method. For example, the complete text of the `SplitFileAsync`

method in the sample operation provider is given in Display 2 (below). The method simply constructs an instance of the `OperationBase` descendant—in this case, `SplitFileOperation`—and starts it.

Note that the `OperationBase` object requires a reference to the `AsyncOperationProviderBase` object. This is as the `OperationBase` object must be able to get the completed event subscriber list, and possibly the progress changed event subscriber list. These subscriber lists are held by the `AsyncOperationProviderBase` object.

The `StartOperation` method is shown in Display 3 (below). This method does four things: checks that the user state object is not already in use (and throws an exception if it is), inserts the user state object-`OperationBase` descendant object pair into the `Dictio-`

```
public void SplitFileAsync (string sourceFileName, string targetFileNameBase,
    long segmentSize, object userState)
{
    OperationBase splitFileOperation = new SplitFileOperation
       (sourceFileName, targetFileNameBase, segmentSize, this, userState);
    StartOperation (splitFileOperation);
}
```

Display 2. The SplitFileAsync method.

```
protected void StartOperation (OperationBase operation)
{
    bool startExecutor;
    lock (_providerState)
    {
        object dictionaryKey = operation.UserState ?? _nullSurrogate;
        if (_providerState.CurrentOperations.ContainsKey (dictionaryKey))
        {
            operation.Dispose ();
            throw new ArgumentException ("userState object already in use.");
        }

        _providerState.CurrentOperations.Add (dictionaryKey, operation);
        _providerState.QueuedOperations.Enqueue (operation);
        _providerState.MaximumQueueLength = Math.Max
            (_providerState.MaximumQueueLength,
             _providerState.QueuedOperations.Count);
        startExecutor = (_providerState.CurrentExecutorCount < _executorCountLimit);
        if (startExecutor)
        {
            _providerState.CurrentExecutorCount ++;
            _providerState.MaximumExecutorCount = Math.Max
               (_providerState.MaximumExecutorCount,
                _providerState.CurrentExecutorCount);
        }
    }
    if (startExecutor)
    {
        ThreadPool.QueueUserWorkItem (Executor, null);
    }
}
```

Display 3. The StartOperation method.

nary of current operations, enqueues the OperationBase descendant object for execution by an executor, and, if necessary, starts an executor.

The private method Executor of the AsyncOperationProviderBase class is run on a thread pool thread to supervise the execution of the operation. This method removes operations from the operation queue, executes the PerformOperation method of the OperationBase descendant class to perform the operation, catches exceptions thrown by the PerformOperation method, creates the completed event arguments, publishes the operationCompleted event, removes the operation from the Dictionary, and decrements the executor count and exits if appropriate.

The actual execution of the operation occurs in the lines:

```
if (! operation.Canceled)
{
    operation.PerformOperation ();
}
completedEventArgs =
    operation.GetCompletedEventArgs
        (null);
```

These lines direct the operation to perform itself, and then to construct its completed event arguments for successful or canceled operation completion. Should either of these lines throw an exception, the exception is caught and the line:

```
completedEventArgs =
    operation.GetCompletedEventArgs
        (exc);
```

is executed. This line directs the operation to construct its completed event arguments for operation termination by an exception.

The remainder of the Executor method maintains the operation provider's state, including updating the performance counters; and exits when there are no more operations to perform.

The AsyncOperationProviderBase class also provides operation cancellation facilities. The Cancel method cancels a single operation, while the CancelAll method cancels all currently known operations. Operations are not removed from the Dictionary or Queue when canceled. Instead the Executor method notices they are canceled and publishes their completed events without attempting to execute the operation itself. This ensures the operations' completed events are always published. The completed event indicates whether the operation was canceled.

The Operation Base

The asynchronous operation itself takes place in the OperationBase descendant object. The OperationBase class has only two methods which must be overridden, and one which may be overridden. The GetOperationCompletedSubscribers method is responsible for reaching out to the provider class through the Parent property reference and retrieving the operationCompleted subscribers. The GetOperationCompletedEventArgs method is responsible for obtaining the completed event arguments object, if the operation is one that returns a value. There is a default implementation of GetOperationCompletedEventArgs adequate for the cases where the operation does not return a value. The operation itself is implemented in the void-valued PerformOperation method.

If the operation provides results, the operation must store any result in properties or variables of the OperationBase descendant class itself. The AsyncOperationProviderBase descendant class should define a public class descending from AsyncCompletedEventArgs containing operation results, and define the operation's completed event as an EventHandler<> of that class. The GetOperationCompletedEventArgs method must create an instance of the completed event argument class containing the operation results.

Should the operation produce progress events, the operation provider must define a ProgressChanged event, and the operation must create ProgressChanged event arguments, acquire the subscriber list to the ProgressChanged event through the Parent reference, and use the AsynchronousPublishEvent method of the OperationBase class to publish the event. Two of the operations provided by the sample operation do this.

The major resources provided to the operation by the OperationBase class are the ability to both asynchronously and synchronously publish events, and to access the members of the operation provider through the Parent reference.

The AsynchronousPublishEvent method of OperationBase provides asynchronous publication (that is, publication that does not wait for event delivery) of an asynchronous event (that is, an event published by an asynchronous operation). The SynchronousPub-

lishEvent method provides synchronous publication of an asynchronous event. These methods use Post or Send on the AnyncOperation created as the OperationBase was constructed to invoke a private DeliverEvent method. The DeliverEvent method is, due to invocation through the appropriate SynchronizationContext, executed on a thread compatible with that of the operation requestor.

Using these base classes to construct an event-oriented asynchronous operation provider is straightforward. Create a class that inherits from AsyncOperationProviderBase. In that class, create a class that inherits from OperationBase. In the OperationBase descendant class, write the PerformOperation method to perform the operation desired as if it were synchronous. Use the OperationBase class to hold all context needed for the operation and any return value: input and output file names, or result file lengths, for example. If the operation is to return results, define a completed event arguments class appropriately. Write the GetCompletedEventSubscribers method to return the operationCompleted delegate from the provider class. In the provider class, create an operationAsync method and an operationCompleted event. The operationAsync method must create an instance of the OperationBase descendant and pass it to the StartOperation method. The AsyncOperationProviderBase and OperationBase classes take care of the various boilerplate details.

The Sample Operation Provider

The code provided with this Application Note includes a sample event-based asynchronous pattern operation provider called FileStoreProvider. This sample provider exhibits the use of the AsyncOperationProviderBase class.

The sample operation provider creates copies of files or streams, possibly separating the copy into fragments of a specified size. Fragmentation of files is necessary when, for example, storing large files on FAT32 volumes such as USB Flash media. Only files of less than 4 GB can be stored on FAT32 volumes.

The operation provider provides four operations:

- Copy a specified file to a specified file. This operation is invoked through the CopyFileAsync method of FileStoreProvider and is implemented through the CopyFileOperation object.
- Fragment a specified file to a set of fragment files with a specified path prefix. This operation is invoked through the FragmentFileAsync method of FileStoreProvider and is implemented through the FragmentFileOperation object.
- Copy a specified stream to a specified file. This operation is invoked through the CopyStreamAsync method of FileStoreProvider and is implemented through the CopyStreamOperation object.
- Fragment a specified stream to a set of fragment files with a specified path prefix. This operation is invoked through the FragmentStreamAsync method of FileStoreProvider and is implemented through the FragmentStreamOperation object.

When a file or stream is fragmented, each fragment but the last is of the specified size. Additionally, if a fragmentation operation produces a single fragment, the fragment is named exactly the prefix. Otherwise, the resulting files are named the prefix with a suffix of . 00, .01, and so forth.

Because the operations are so similar to one another, an abstract StoreOperationBase class containing common functionality is used as an intermediate class between the OperationBase class and the operation implementation classes. The primary logic for copying a specified amount of data from a source to a target is in this class as the CopyStreamSegment method. This method is shown in Display 4 (next page). Note how the overall iteration is controlled by three factors: the Canceled property, the amount of data copied in this invocation, and the end of data from the source. The overall iteration also determines the progress of the overall operation and, if possible and needed, publishes a ProgressChanged events when the progress has actually changed.

The provider exposes both an operationAsync method and an operationCompleted event for each operation. In addition, the provider exposes a ProgressChanged event. The progress changed event is not published for the stream copy and fragment operations, as there is no way to determine the amount of data to be transferred and so no way to estimate the fraction of the operation that is complete. Completed events are of course published for each operation.

Note that while each of the four available operations has its own operationCompleted event, the Pro-

```
protected long CopyStreamSegment
   (Stream source,
    long previouslyCopiedLength,
    Stream target,
    long maximumLength)
{
    byte [] buffer = new byte [_bufferSize];
    long bytesCopied = 0;

    while (! Canceled)
    {
        long bytesRemaining = maximumLength - bytesCopied;
        int readLength = (int) Math.Min (_bufferSize, bytesRemaining);
        if (readLength <= 0)
        {
            break;
        }
        int writeLength = source.Read (buffer, 0, readLength);
        if (writeLength <= 0)
        {
            break;
        }
        target.Write (buffer, 0, writeLength);
        bytesCopied += writeLength;

        if (_sourceFileLength >= 0)
        {
            float percentCopiedFloat =
                (100.0f * (float) (previouslyCopiedLength + bytesCopied) /
                    (float) _sourceFileLength);
            int percentCopied =
                (int) Math.Min (Math.Max (0.0f, percentCopiedFloat), 100.0f);
            if (percentCopied != _previousProgressPublication)
            {
                ProgressChangedEventArgs eventArgs =
                    new ProgressChangedEventArgs (percentCopied, UserState);
                EventHelper.EventDescription eventDescription =
                    new EventHelper.EventDescription
                       (Parent,
                        ((FileStoreProvider) Parent).ProgressChanged,
                        eventArgs);
                AsynchronousPublishEvent (eventDescription);
                _previousProgressPublication = percentCopied;
            }
        }
    }

    return bytesCopied;
}
```

Display 4. The CopyStreamSegment method.

gressChanged event is shared among all operations: the user state object included in the event arguments permits identifying the operation for which progress is reported. This is arguably a flaw—if operations should have separate Completed events why should they not also have separate ProgressChanged events?—but it is idiomatic for the event-based asynchronous operation model.

The provider by default limits the number of simultaneous operations to two, in an attempt to avoid disk positioning contention. The constructor argument can override this limit.

The Sample Application

The code provided with this Application Note includes a sample application providing access to the file split operation of the sample operation provider.

The sample program implements a thoroughly typical WinForms GUI exposing the file fragment operation. It indicates operation progress as the selected files are fragmented. The program contains a FileStatusDisplay class that links operation state to display state. The FileStatusDisplay instance corresponding to an operation is used as the user state object for the object. The program maintains a count of the number of operations requested but not yet completed; this count informs the enabling and disabling of controls and the role of the start/cancel UI button.

File operation creation and cancellation is performed by the code shown in Display 5 (below). Reception of the ProgressChanged event is shown in Display 6 (next page). Reception of the completed event for the file operations is shown in Display 7 (next page).

The GUI program uses a technique of possible further utility. The overall form uses a TableLayoutPanel for its controls in a typical fashion. The control where operation status is displayed is also a TableLayoutPanel with rows dynamically inserted: these rows represent the operations desired.

The FileStatusDisplay class is not itself a control, but contains three controls. The FileStatusDisplay class can be instructed to insert or remove these controls from a container control, which in this case is the interior TableLayoutPanel. The AddStatusDis-

```
private void ExecuteButton_Click (object sender, EventArgs e)
{
    if (_inProgressCount != 0)
    {
        _fileStoreProvider.CancelAll ();
    }
    else
    {
        string targetDirectoryName =
            Path.GetFullPath (TargetDirectoryTextBox.Text.Trim ());
        foreach (FileStatusDisplay fileStatusDisplay in _currentFileStatusDisplays)
        {
            string sourceFilePath = Path.GetFullPath (fileStatusDisplay.FileName);
            string sourceFileBaseName = Path.GetFileName (sourceFilePath);
            string targetFileBasePath =
                Path.Combine (targetDirectoryName, sourceFileBaseName);
            _fileStoreProvider.SplitFileAsync
               (sourceFilePath, targetFileBasePath, _fragmentSize.Value,
                fileStatusDisplay);
            _inProgressCount++;
        }
    }

    EnableControls ();
}
```

Display 5. Creation of asynchronous file fragment operations.

play method of the overall Form has the FileStatusDisplay instance insert its controls into the interior TableLayoutPanel, then arranges the controls into a new row of the TableLayoutPanel. The new row is always the next-to-last row: an empty row follows it. Combined with all rows having row styles of size type AutoSize, this permits the visible contents of the file status area to be at the top of its display area, without a visible expansion of the last row.

Further Information

A previous version of this Application Note was published under the GPL 3 at CodeProject as The Event-Based Asynchronous Pattern in .NET.

This Application Note and a Visual Studio solution containing all source code for the sample application are contained in a .ZIP file at: http://brianhetrick.com/an/AN-8.ZIP.

```
private void FileStoreProvider_ProgressChanged
    (object publisher, ProgressChangedEventArgs eventArgs)
{
    FileStatusDisplay fileStatusDisplay = eventArgs.UserState as FileStatusDisplay;
    if (fileStatusDisplay != null)
    {
        fileStatusDisplay.ProgressPercentage = eventArgs.ProgressPercentage;
    }
}
```

Display 6. Reception of operation ProgressChanged events.

```
private void FileStoreProvider_SplitFileCompleted
    (object publisher, AsyncCompletedEventArgs eventArgs)
{
    FileStatusDisplay fileStatusDisplay = eventArgs.UserState as FileStatusDisplay;
    if (fileStatusDisplay != null)
    {
        if (eventArgs.Error != null)
        {
            fileStatusDisplay.Message = "Aborted.";
            fileStatusDisplay.Successful = false;
        }
        else if (eventArgs.Cancelled)
        {
            fileStatusDisplay.Message = "Canceled.";
            fileStatusDisplay.Successful = false;
        }
        else
        {
            fileStatusDisplay.Message = "Successful.";
            fileStatusDisplay.Successful = true;
        }
        _inProgressCount--;
    }

    EnableControls ();
}
```

Display 7. Reception of operation Completed events.

The MonoDevelop software used to produce most of the code described in this Application Note is available for no charge at http://www.monodevelop.com/. The Raspbian operating system on which this software was produced and an which this Application Note was produced is available for no charge at https://www.raspberrypi.org/.

The Visual Studio Community Edition software used to produce the WinForms code described in this Application Note is available for no charge at https://www.visualstudio.com/downloads/.

The LibreOffice office suite used to create this Application Note itself is available for no charge at http://www.libreoffice.org.

This Application Note uses the Book Antiqua font for body text, the Source Sans Pro fonts for headings, headers, and footers, and the Source Code Pro font for computer code. The Book Antiqua font can be licensed from Monotype at http://catalog.monotype.com/family/monotype/book-antiqua and is included in several of Microsoft's products. The Source Sans Pro font is available at no charge at https://adobe-fonts.github.io/source-sans-pro/. The Source Code Pro font is available at no charge at https://github.com/adobe-fonts/source-code-pro.

Disclaimer

The author believes all content of this Application Note and any software and design it describes is a straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes, the designs it describes, or any portion of any or all, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License