Expert F#

Don Syme, Adam Granicz, and
Antonio Cisternino

Apress®

Expert F#

ISBN-13: 978-1-59059-850-4

ISBN-10: 1-59059-850-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Jim Huddleston, Jonathan Hassell
Technical Reviewer: Tomáš Petrícek
Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jason Gilmore, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Susan Glinert
Proofreader: April Eddy
Indexer: Present Day Indexing
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.expert-fsharp.com.

CHAPTER 13

Reactive, Asynchronous, and Concurrent Programming

So far in this book you've seen functions and objects that process their inputs immediately using a single "thread" of execution where the code runs to completion and produces useful results or state changes. In this chapter, you'll turn your attention to *concurrent*, *parallel*, *asynchronous*, and *reactive* programs. These each represent substantially different approaches to programming from those you've seen so far. Some of the reasons for turning to these techniques are as follows:

- To achieve better responsiveness in a graphical user interface (GUI)
- To report progress results during a long-running computation and to support cancellation of these computations
- To achieve greater throughput in a reactive application or service
- To achieve faster processing rates on a multiprocessor machine or cluster
- To take advantage of the I/O parallelism available in modern disk drives or network connections
- To sustain processing while network and disk I/O operations are in process

In this chapter, we cover some of the techniques that can help achieve these outcomes:

- Using .NET threads and the `BackgroundWorker` class for background computations
- Using events and messages to report results back to a GUI
- Using F# asynchronous workflows and the .NET thread pool to handle network requests and other asynchronous I/O operations
- Using F# pattern matching to process message queues
- Using low-level .NET shared-memory primitives to implement new concurrency techniques and control access to mutable data structures

In Chapter 11 we looked at the most common type of reactive program: GUI programs that respond to events raised on the GUI thread. The inner loop of such an application (contained

in the Windows Forms library) spends most of its time blocked waiting for the underlying operating system to notify it of a relevant event, such as a click from the user or a timer event from the operating system. This notification is received as an event in a message queue. Many GUI programs have only a single thread of execution, so all computation happens on the GUI thread. This can lead to problems such as nonresponsive user interfaces. This is one of many reasons it is important to master some of the techniques of concurrent and asynchronous programming.

Introducing Some Terminology

Before we begin, let's look more closely at some terminology:

- *Processes* are, in the context of this chapter, standard operating system (OS) processes. Each instance of the .NET CLR runs in its own process, and multiple instances of the .NET CLR will often be running on the same machine.
- *Threads* are, in the context of this chapter, standard .NET threads. On most implementations of .NET these correspond to operating system threads. Each .NET process has many threads running within the one process.
- *Concurrent programs* are ones with multiple threads of execution, each typically executing different code, or are at different execution points within the same code. Simultaneous execution may be simulated by scheduling and descheduling the threads, which is done by the OS. For example, most operating system services and GUI applications are concurrent.
- *Parallel programs* are one or more processes or threads executing simultaneously. For example, many modern microprocessors have two or more physical CPUs capable of executing processes and threads in parallel. Parallel programs can also be *data parallel*. For example, a massively parallel device such as a graphics processor unit (GPU) can process arrays and images in parallel. Parallel programs can also be a cluster of computers on a network, communicating via message passing. Historically, some parallel scientific programs have even used e-mail for communication!
- *Asynchronous programs* perform requests that do not complete immediately but are fulfilled at a later time and where the program issuing the request has to do meaningful work in the meantime. For example, most network I/O is inherently asynchronous. A web crawler is also a highly asynchronous program, managing hundreds or thousands of simultaneous network requests.
- *Reactive programs* are ones whose normal mode of operation is to be in a state waiting for some kind of input, such as waiting for user input or for input from a message queue via a network socket. For example, GUI applications and web servers are reactive programs.

Parallel, asynchronous, concurrent, and reactive programs bring many interesting challenges. For example, these programs are nearly always *nondeterministic*. This makes debugging more challenging since it is difficult to "step" through a program, and even pausing a running program with outstanding asynchronous requests may cause timeouts. Most dramatically, incorrect concurrent programs may *deadlock*, which means all threads are waiting on results from some other thread and no thread can make progress. Programs may also *livelock*, where processing is occurring and messages are being sent between threads but no useful work is being performed.

Using and Designing Background Workers

One of the easiest ways to get going with concurrency and parallelism is to use the `System.ComponentModel.BackgroundWorker` class of the .NET Framework. A `BackgroundWorker` class runs on its own dedicated operating system thread. These objects can be used in many situations but are especially useful for "coarse-grained" concurrency and parallelism such as checking the spelling of a document in the background. In this section we show some simple uses of `BackgroundWorker` and how to build similar objects that use a `BackgroundWorker` internally.

Listing 13-1 shows a simple use of `BackgroundWorker` that computes the Fibonacci numbers on the worker thread.

Listing 13-1. *A Simple `BackgroundWorker`*

```
open System.ComponentModel
open System.Windows.Forms

let worker = new BackgroundWorker()
let numIterations = 1000

worker.DoWork.Add(fun args ->

    let rec computeFibonacci resPrevPrev resPrev i =
        // Compute the next result
        let res = resPrevPrev + resPrev

        // At the end of the computation and write the result into the mutable state
        if i = numIterations then
            args.Result <- box res
        else
            // Compute the next result
            computeFibonacci resPrev res (i+1)

    computeFibonacci 1 1 2)

worker.RunWorkerCompleted.Add(fun args ->
    MessageBox.Show(sprintf "Result = %A" args.Result) |> ignore)

// Execute the worker
worker.RunWorkerAsync()
```

Table 13-1 shows the primary members of a `BackgroundWorker` object. The execution sequence of the code in Listing 13-1 is as follows:

1. The main application thread creates and configures a `BackgroundWorker` object.
2. Once configuration is complete, the main application thread calls the `RunWorkerAsync` method on the `BackgroundWorker` object. This causes the `DoWork` event to be raised on the worker thread.

3. The `DoWork` event handler is executed in the worker thread and computes the 1000th Fibonacci number. At the end of the computation, the result is written into `args.Result`, a mutable storage location in the event arguments for the `DoWork` event. The `DoWork` event handler then completes.

4. At some point after the `DoWork` event handler completes, the `RunWorkerCompleted` event is automatically raised on the main application thread. This displays a message box with the result of the computation, retrieved from the `args` field of the event arguments.

Table 13-1. *Primary Members of the `BackgroundWorker` Class*

Member and Type	Description
`RunWorkerAsync: unit -> unit`	Starts the process on a separate thread asynchronously. Called from the main thread.
`CancelAsync: unit -> unit`	Set the `CancellationPending` flag of the background task. Called from the main thread.
`CancellationPending: bool`	Set to `true` by raising `CancelAsync`. Used by the worker thread.
`WorkerReportsProgress: bool`	Set to `true` if the worker can support progress updates. Used by the main thread.
`WorkerSupportsCancellation: bool`	Set to `true` if the worker can support cancellation of the current task in progress. Used by the main thread.
`ReportProgress: int -> unit`	Indicate the progress of the operation. Used by the worker thread.
`DoWork: IEvent<DoWorkEventArgs>`	Fires in response to a call to `RunWorkerAsync`. Invoked on the worker thread.
`RunWorkerCompleted: IEvent<RunWorkerCompletedEventArgs>`	Fires when the background operation is canceled, when the operation is completed, or when an exception is thrown. Invoked on the main thread.
`ProgressChanged: IEvent<ProgressChangedEventArgs>`	Fires whenever the `ReportProgress` property is set. Invoked on the main thread.

■ **Note** Objects such as a `BackgroundWorker` are "two-faced": they have some methods and events that are for use from the main thread and some that are for use on the worker thread. This is common in concurrent programming. In particular, be careful to understand which thread an event is raised on. For `BackgroundWorker`, the `RunWorkerAsync` and `CancelAsync` methods are for use from the GUI thread, and the `ProgressChanged` and `RunWorkerCompleted` events are raised on the GUI thread. The `DoWork` event is raised on the worker thread, and the `ReportProgress` method and the `CancellationPending` property are for use from the worker thread when handling this event.

The members in Table 13-1 show two additional facets of BackgroundWorker objects: they can optionally support protocols for cancellation and reporting progress. To report progress percentages, a worker must simply call the ReportProgress method, which raises the ProgressChanged event in the GUI thread. For cancellation, a worker computation need only check the CancellationPending property at regular intervals, exiting the computation as a result.

Building a Simpler Iterative Worker

Capturing common control patterns such as cancellation and progress reporting is an absolutely essential part of mastering concurrent programming. However, one of the problems with .NET classes such as BackgroundWorker is that they are often more imperative than an F# programmer may want, and they force other common patterns to be captured by using mutable data structures shared between threads. This leads to the more difficult topic of shared-memory concurrency, which we discuss later in the chapter. Furthermore, the way BackgroundWorker handles cancellation means that flag-checks and early-exit paths have to be inserted in the executing background process. Finally, BackgroundWorker is not useful for background threads that perform asynchronous operations, since the background thread will exit "too early," before the callbacks for the asynchronous operations have executed.

For this reason, it can often be useful to build abstractions that are similar to BackgroundWorker but that capture richer or different control patterns, preferably in a way that does not rely on the use of mutable state and that interferes less with the structure of the overall computation. Much of the rest of this chapter will look at various techniques to build these control structures.

We start with a case study where we build a type IterativeBackgroundWorker that represents a variation on the BackgroundWorker design pattern. Listing 13-2 shows the code.

Listing 13-2. *A Variation on the BackgroundWorker Design Pattern for Iterative Computations*

```
open System.ComponentModel
open System.Windows.Forms

/// An IterativeBackgroudWorker follows the BackgroundWorker design pattern
/// but instead of running an arbitrary computation it iterates a function
/// a fixed number of times and reports intermediate and final results.
/// The worker is paramaterized by its internal state type.
///
/// Percentage progress is based on the iteration number. Cancellation checks
/// are made at each iteration. Implemented via an internal BackgroundWorker.
type IterativeBackgroundWorker<'a>(oneStep:('a -> 'a),
                                   initialState:'a,
                                   numIterations:int) =

    let worker =
        new BackgroundWorker(WorkerReportsProgress=true,
                             WorkerSupportsCancellation=true)
```

```
// Create the events that we will later trigger
let triggerCompleted,completed = IEvent.create()
let triggerError    ,error     = IEvent.create()
let triggerCancelled,cancelled = IEvent.create()
let triggerProgress ,progress  = IEvent.create()

do worker.DoWork.Add(fun args ->
    // This recursive function represents the computation loop.
    // It runs at "maximum speed", i.e. is an active rather than
    // a reactive process, and can only be controlled by a
    // cancellation signal.
    let rec iterate state i =
        // At the end of the computation terminate the recursive loop
        if worker.CancellationPending then
           args.Cancel <- true
        elif i < numIterations then
            // Compute the next result
            let state' = oneStep state

            // Report the percentage computation and the internal state
            let percent = int ((float (i+1)/float numIterations) * 100.0)
            do worker.ReportProgress(percent, box state);

            // Compute the next result
            iterate state' (i+1)
        else
            args.Result <- box state

    iterate initialState 0)

do worker.RunWorkerCompleted.Add(fun args ->
    if args.Cancelled       then triggerCancelled()
    elif args.Error <> null then triggerError args.Error
    else triggerCompleted (args.Result :?> 'a))

do worker.ProgressChanged.Add(fun args ->
    triggerProgress (args.ProgressPercentage,(args.UserState :?> 'a)))

member x.WorkerCompleted  = completed
member x.WorkerCancelled  = cancelled
member x.WorkerError      = error
member x.ProgressChanged  = progress

// Delegate the remaining members to the underlying worker
member x.RunWorkerAsync()    = worker.RunWorkerAsync()
member x.CancelAsync()       = worker.CancelAsync()
```

The types inferred for the code in Listing 13-2 are as follows:

```
type IterativeBackgroundWorker<'state> =
    new : ('state -> 'state) * 'state * int -> IterativeBackgroundWorker<'state>
    member RunWorkerAsync : unit -> unit
    member CancelAsync : unit -> unit

    member ProgressChanged : IEvent<int * 'state>
    member WorkerCompleted : IEvent<'state>
    member WorkerCancelled : IEvent<unit>
    member WorkerError     : IEvent<exn>
```

Let's take a look at this signature first, because it represents the design of the type. The worker constructor is given a function of type `'state -> 'state` to compute successive iterations of the computation, plus an initial state and the number of iterations to compute. For example, you can compute the Fibonacci numbers using the following iteration function:

```
let fibOneStep (fibPrevPrev:bigint,fibPrev) = (fibPrev, fibPrevPrev+fibPrev);;
```

The type of this function is as follows:

```
val fibOneStep : bigint * bigint -> bigint * bigint
```

The `RunWorkerAsync` and `CancelAsync` members follow the `BackgroundWorker` design pattern, as do the events, except that we have expanded the `RunWorkerCompleted` event into three events to correspond to the three termination conditions and modified the `ProgressChanged` to include the state. You can instantiate the type as follows:

```
> let worker = new IterativeBackgroundWorker<_>( fibOneStep,(1I,1I),100);;
val worker : IterativeBackgroundWorker<bigint * bigint>

> worker.WorkerCompleted.Add(fun result ->
      MessageBox.Show(sprintf "Result = %A" result) |> ignore);;
val it : unit = ()

> worker.ProgressChanged.Add(fun (percentage, state) ->
    printfn "%d%% complete, state = %A" percentage state);;
val it : unit = ()

> worker.RunWorkerAsync();;
1% complete, state = (1I, 1I)
2% complete, state = (1I, 2I)
3% complete, state = (2I, 3I)
4% complete, state = (3I, 5I)
...
```

```
98% complete, state = (135301852344706746049I, 2189229958345551690261)
99% complete, state = (218922995834555169026I, 3542248481792619150751)
100% complete, state = (354224848179261915075I, 5731478440138170841011)
val it : unit = ()
```

One difference here is that cancellation and percentage progress reporting are handled automatically based on the iterations of the computation. This is assuming each iteration takes roughly the same amount of time. Other variations on the `BackgroundWorker` design pattern are possible. For example, reporting percentage completion of fixed tasks such as installation is often performed by timing sample executions of the tasks and adjusting the percentage reports appropriately.

■ **Note** We implemented `IterativeBackgroundWorker` via delegation rather than inheritance. This is because its external members are different from those of `BackgroundWorker`. The .NET documentation recommends you use implementation inheritance for this, but we disagree. Implementation inheritance can only *add* complexity to the signature of an abstraction and never makes things simpler, whereas an `IterativeBackgroundWorker` is in many ways simpler than using a `BackgroundWorker`, despite that it uses an instance of the latter internally. Powerful, compositional, simple abstractions are the primary building blocks of functional programming.

Raising Additional Events from Background Workers

Often you will need to raise additional events from objects that follow the `BackgroundWorker` design pattern. For example, let's say you want to augment `IterativeBackgroundWorker` to raise an event when the worker starts its work and for this event to pass the exact time that the worker thread started as an event argument. Listing 13-3 shows the extra code you need to add to the `IterativeBackgroundWorker` type to make this happen. We use this extra code in the next section.

Listing 13-3. *Code to Raise GUI-Thread Events from an* `IterativeBackgroundWorker` *Object*

```
open System
open System.Threading

type IterativeBackgroundWorker<'a>(...) =

    let worker = ...

    // The constructor captures the synchronization context. This allows us to post
    // messages back to the GUI thread where the BackgroundWorker was created.
    let syncContext = SynchronizationContext.Current
    do if syncContext = null then failwith "no synchronization context found"

    let triggerStarted,started = IEvent.create()
```

```
    // Raise the event when the worker starts. This is done by posting a message
    // to the captured synchronization context.
    do worker.DoWork.Add(fun args ->
        syncContext.Post(SendOrPostCallback(fun _ -> triggerStarted(DateTime.Now)),
                         state=null)
        ...

    /// The Started event gets raised when the worker starts. It is
    /// raised on the GUI thread (i.e. in the synchronization context of
    /// the thread where the worker object was created).
    // It has type IEvent<DateTime>
    member x.Started             = started
```

The simple way to raise additional events is often wrong. For example, it is tempting to simply create an event, arrange for it to be triggered, and publish it, as you would do for a GUI control. However, if you do that, you will end up triggering the event on the background worker thread, and its event handlers will run on that thread. This is dangerous, because most GUI objects (and many other objects) can be accessed only from the thread they were created on; this is a restriction enforced by most GUI systems.

One of the nice features of the `BackgroundWorker` class is that it automatically arranges to raise the `RunWorkerCompleted` and `ProgressChanged` events on the GUI thread. We have shown how to achieve this in Listing 13-3. Technically speaking, the extended `IterativeBackgroundWorker` object has captured the *synchronization context* of the thread where it was created and posts an operation back to that synchronization context. A synchronization context is just an object that lets you post operations back to another thread. For threads such as a GUI thread, this means posting an operation will post a message through the GUI event loop.

Connecting a Background Worker to a GUI

To round off this section on the `BackgroundWorker` design pattern, we show the full code required to build a small application with a background worker task that supports cancellation and reports progress. Listing 13-4 shows the full code.

Listing 13-4. *Connecting an `IterativeBackgroundWorker` to a GUI*

```
open System.Drawing
open System.Windows.Forms

let form = new Form(Visible=true,TopMost=true)

let panel = new FlowLayoutPanel(Visible=true,
                                Height = 20,
                                Dock=DockStyle.Bottom,
                                BorderStyle=BorderStyle.FixedSingle)

let progress = new ProgressBar(Visible=false,
                               Anchor=(AnchorStyles.Bottom ||| AnchorStyles.Top),
                               Value=0)
```

```
let text = new Label(Text="Paused",
                     Anchor=AnchorStyles.Left,
                     Height=20,
                     TextAlign= ContentAlignment.MiddleLeft)

panel.Controls.Add(progress)
panel.Controls.Add(text)
form.Controls.Add(panel)

let fibOneStep (fibPrevPrev:bigint,fibPrev) = (fibPrev, fibPrevPrev+fibPrev)

// Run the iterative algorithm 500 times before reporting intermediate results
// Burn some additional cycles to make sure it runs slowly enough
let rec RepeatN n f s = if n <= 0 then s else RepeatN (n-1) f (f s)
let rec BurnN n f s = if n <= 0 then f s else ignore (f s); BurnN (n-1) f s
let step = (RepeatN 500 (BurnN 1000 fibOneStep))

// Create the iterative worker.
let worker = new IterativeBackgroundWorker<_>(step,(1I,1I),100)

worker.ProgressChanged.Add(fun (progressPercentage,state)->
    progress.Value <- progressPercentage)

worker.WorkerCompleted.Add(fun (_,result) ->
    progress.Visible <- false;
    text.Text <- "Paused";
    MessageBox.Show(sprintf "Result = %A" result) |> ignore)

worker.WorkerCancelled.Add(fun () ->
    progress.Visible <- false;
    text.Text <- "Paused";
    MessageBox.Show(sprintf "Cancelled OK!") |> ignore)

worker.WorkerError.Add(fun exn ->
    text.Text <- "Paused";
    MessageBox.Show(sprintf "Error: %A" exn) |> ignore)

form.Menu <- new MainMenu()
let workerMenu = form.Menu.MenuItems.Add("&Worker")

workerMenu.MenuItems.Add(new MenuItem("Run",onClick=(fun _ args ->
    text.Text <- "Running";
    progress.Visible <- true;
    worker.RunWorkerAsync())))
```

```
workerMenu.MenuItems.Add(new MenuItem("Cancel",onClick=(fun _ args ->
    text.Text <- "Cancelling";
    worker.CancelAsync())))

form.Closed.Add(fun _ -> worker.CancelAsync())
```

When run in F# Interactive, a window appears as in Figure 13-1.

Figure 13-1. *A GUI window with a* `BackgroundWorker` *reporting progress percentage*

■ **Note** Forcibly aborting computations uncooperatively is not recommended in .NET programming. You can attempt to do this using `System.Threading.Thread.Abort()`, but the use of this method may have many unintended consequences, discussed later in this chapter.

Introducing Asynchronous Computations

The two background worker samples we've shown so far run at "full throttle." In other words, the computations run on the background threads as active loops, and their reactive behavior is limited to flags that check for cancellation. In reality, background threads often have to do different kinds of work, either by responding to completing asynchronous I/O requests, by processing messages, by sleeping, or by waiting to acquire shared resources. Fortunately, F# comes with a powerful set of techniques for structuring asynchronous programs in a natural way. These are called *asynchronous workflows*. In the next three sections, we cover how to use asynchronous workflows to structure asynchronous and message-processing tasks in ways that preserve the essential logical structure of your code.

Fetching Multiple Web Pages Asynchronously

One of the most intuitive asynchronous tasks is fetching a web page; we all use web browsers that can fetch multiple pages simultaneously. In the samples in Chapter 2 we showed how to

fetch pages synchronously. This is useful for many purposes, but browsers and high-performance web crawlers will have tens or thousands of connections "in flight" at once.

The type `Microsoft.FSharp.Control.Async<'a>` lies at the heart of F# asynchronous workflows. A value of type `Async<'a>` represents a program fragment that will generate a value of type `'a` "at some point in the future." Listing 13-5 shows how to use asynchronous workflows to fetch several web pages simultaneously.

Listing 13-5. *Fetching Three Web Pages Simultaneously*

```
open System.Net
open System.IO
open Microsoft.FSharp.Control.CommonExtensions

let museums = ["MOMA",           "http://moma.org/";
               "British Museum", "http://www.thebritishmuseum.ac.uk/";
               "Prado",          "http://museoprado.mcu.es"]

let fetchAsync(nm,url:string) =
    async { do printfn "Creating request for %s..." nm
            let req  = WebRequest.Create(url)

            let! resp  = req.GetResponseAsync()

            do printfn "Getting response stream for %s..." nm
            let stream = resp.GetResponseStream()

            do printfn "Reading response for %s..." nm
            let reader = new StreamReader(stream)
            let! html = reader.ReadToEndAsync()

            do printfn "Read %d characters for %s..." html.Length nm }
for nm,url in museums do
    Async.Spawn (fetchAsync(nm,url))
```

The types of these functions and values are as follows:

```
val museums : (string * string) list
val fetchAsync : string * string -> Async<unit>
```

When run on one of our machines via F# Interactive, the output of the code from Listing 13-5 is as follows:

```
Creating request for MOMA...
Creating request for British Museum...
Creating request for Prado...
Getting response for MOMA...
Reading response for MOMA...
```

```
Getting response for Prado...
Reading response for Prado...
Read 188 characters for Prado...
Read 41635 characters for MOMA...
Getting response for British Museum...
Reading response for British Museum...
Read 24341 characters for British Museum...
```

The heart of the code in Listing 13-5 is the construct introduced by `async { ... }`. This is an application of the workflow syntax introduced in Chapter 9. Now let's take a closer look at Listing 13-5. The key operations are the two `let!` operations within the workflow expression:

```
async { do ...
        let! resp  = req.GetResponseAsync()
        do ...
        ...
        let! html = reader.ReadToEndAsync()
        do ... }
```

Within asynchronous workflow expressions, the language construct `let!` *var* `=` *expr* `in` *body* simply means "perform the asynchronous operation *expr* and bind the result to *var* when the operation completes. Then continue by executing the rest of the computation *body*."

With this in mind, you can now see what `fetchAsync` does:

- It synchronously requests a web page.
- It asynchronously awaits a response to the request.
- It gets the response `Stream` and `StreamReader` synchronously after the asynchronous request completes.
- It reads to the end of the stream asynchronously.
- After the read completes, it prints the total number of characters read synchronously.

Finally, the method `Async.Spawn` is used to initiate the execution of a number of asynchronous computations. This works by queuing the computations in the .NET thread pool. The .NET thread pool is explained in more detail in the following section.

Understanding Thread Hopping

Asynchronous computations are different from normal, synchronous computations: an asynchronous computation tends to "hop" between different underlying .NET threads. To see this, let's augment the asynchronous computation with diagnostics that show the ID of the underlying .NET thread at each point of active execution. You can do this by replacing uses of `printfn` in the function `fetchAsync` with uses of the following function:

```
let tprintfn fmt =
    printf "[.NET Thread %d]" System.Threading.Thread.CurrentThread.ManagedThreadId;
    printfn fmt
```

After doing this, the output changes to the following:

```
[.NET Thread 12]Creating request for MOMA...
[.NET Thread 13]Creating request for British Museum...
[.NET Thread 12]Creating request for Prado...
[.NET Thread 8]Getting response for MOMA...
[.NET Thread 8]Reading response for MOMA...
[.NET Thread 9]Getting response for Prado...
[.NET Thread 9]Reading response for Prado...
[.NET Thread 9]Read 188 characters for Prado...
[.NET Thread 8]Read 41635 characters for MOMA...
[.NET Thread 8]Getting response for British Museum...
[.NET Thread 8]Reading response for British Museum...
[.NET Thread 8]Read 24341 characters for British Museum...
```

Note how each individual `Async` program "hops" between threads; the MOMA request started on .NET thread 12 and finished life on .NET thread 8. Each asynchronous computation in Listing 13-5 executes in the following way:

- Each asynchronous computation starts life as a work item in the .NET thread pool. (The .NET thread pool is explained in the "What Is the .NET Thread Pool?" sidebar.) These are processed by a number of .NET threads.
- When the asynchronous computations reach the `GetResponseAsync` and `ReadToEndAsync` calls, the requests are made and the continuations are registered as "I/O completion actions" in the .NET thread pool. No thread is used while the request is in progress.
- When the requests complete, they trigger a callback in the .NET thread pool. These may be serviced by different threads than those that originated the calls.

WHAT IS THE .NET THREAD POOL?

.NET objects such as `BackgroundWorker` use a single .NET background thread, which corresponds to a single Windows or other OS thread. OS threads have supporting resources such as an execution stack that consume memory and are relatively expensive resources to create and run.

However, many concurrent processing tasks require only the ability to schedule short-lived tasks that then suspend, waiting for further input. To simplify the process of creating and managing these tasks, the .NET Framework provides the `System.Threading.ThreadPool` class. The thread pool consists of two main sets of suspended tasks: a queue containing user work items and a pool of "I/O completion" callbacks, each waiting for a signal from the operating system. The number of threads in the thread pool is automatically tuned, and items can be either queued asynchronously or registered against a .NET `WaitHandle` synchronization object (for example, a lock, a semaphore, or an I/O request). This is how to queue a work item in the .NET thread pool:

```
open System.Threading

ThreadPool.QueueUserWorkItem(fun _ -> printf "Hello!") |> ignore
```

Under the Hood: What Are Asynchronous Computations?

`Async<'a>` values are essentially a way of writing "continuation-passing" or "callback" programs explicitly. Continuations themselves were described in Chapter 8 along with techniques to pass them explicitly. `Async<'a>` are computations that call a *success continuation* when the asynchronous computation completes and an *exception continuation* if it fails. They provide a form of *managed asynchronous computation*, where "managed" means that several aspects of asynchronous programming are handled automatically:

- *Exception propagation is added "for free"*: If an exception is raised during an asynchronous step, then the exception terminates the entire asynchronous computation and cleans up any resources declared using `use`, and the exception value is then handed to a continuation. Exceptions may also be caught and managed within the asynchronous workflow by using `try/with/finally`.
- *Cancellation checking is added "for free"*: The execution of an `Async<'a>` workflow automatically checks a cancellation flag at each asynchronous operation. Cancellation is controlled through the use of asynchronous groups, a topic covered at `http://www.expert-fsharp.com/Topics/Cancellation`.
- *Resource lifetime management is fairly simple*: You can protect resources across parts of an asynchronous computation by using `use` inside the workflow syntax.

If we put aside the question of cancellation, values of type `Async<'a>` are effectively identical to the following type:

```
type Async<'a> = Async of ('a -> unit) * (exn -> unit) -> unit
```

Here the functions are the success continuation and exception continuations, respectively. Each value of type `Async<'a>` should eventually call one of these two continuations. The `async` object is of type `AsyncBuilder` and supports the following methods, among others:

```
type AsyncBuilder with
    member Return : 'a -> Async<'a>
    member Delay : (unit -> Async<'a>) -> Async<'a>
    member Using: 'a * ('a -> Async<'b>) -> Async<'b> when 'a :> System.IDisposable
    member Let: 'a * ('a -> Async<'b>) -> Async<'b>
    member Bind: Async<'a> * ('a -> Async<'b>) -> Async<'b>
```

The full definition of `Async<'a>` values and the implementations of these methods for the `async` object are given in the F# library source code. As you saw in Chapter 9, builder objects such as `async` containing methods like those shown previously mean that the syntax `async { ... }` can be used as a way of building `Async<'a>` values.

Table 13-2 shows the common constructs used in asynchronous workflow expressions. For example, the following asynchronous workflow:

```
async { let req  = WebRequest.Create("http://moma.org/")
        let! resp  = req.GetResponseAsync()
        let stream = resp.GetResponseStream()
        let reader = new StreamReader(stream)
        let! html = reader.ReadToEndAsync()
        html }
```

is shorthand for the following code:

```
async.Delay(fun () ->
   async.Let(WebRequest.Async("http://moma.org/"), (fun req ->
      async.Bind(req.GetResponseAsync(), (fun resp ->
         async.Let(resp.GetResponseStream(), (fun stream ->
            async.Let(new StreamReader(stream), (fun reader ->
               async.Bind(reader.ReadToEndAsync(), (fun html ->
                  async.Return(html)))))))))))
```

As you saw in Chapter 9, the key to understanding the F# workflow syntax is always to understand the meaning of `let!`. In the case of `async` workflows, `let!` executes one asynchronous computation and schedules the next computation for execution once the first asynchronous computation completes. This is syntactic sugar for the `Bind` operation on the `async` object.

Table 13-2. *Common Constructs Used in `async { ... }` Workflow Expressions*

Construct	Description
`let!` *pat* `=` *expr*	Execute the asynchronous computation `expr` and bind its result to *pat* when it completes. If *expr* has type `Async<'a>`, then *pat* has type `'a`. Equivalent to `async.Bind(expr,(fun pat -> ...))`.
`let pat = expr`	Execute an expression synchronously and bind its result to *pat* immediately. If *expr* has type `'a`, then *pat* has type `'a`. Equivalent to `async.Let(expr,(fun pat -> ...))`.
`do!` *expr*	Equivalent to `let! () = expr`.
`do` *expr*	Equivalent to `let () = expr`.
`return` *expr*	Evaluate the expression, and return its value as the result of the containing asynchronous workflow. Equivalent to `async.Return(expr)`.
`return!` *expr*	Execute the expression as an asynchronous computation, and return its result as the overall result of the containing asynchronous workflow. Equivalent to `expr`.
`use` *pat* `=` *expr*	Execute the expression immediately, and bind its result immediately. Call the `Dispose` method on each variable bound in the pattern when the subsequent asynchronous workflow terminates, regardless of whether it terminates normally or by an exception. Equivalent to `async.Using(expr,(fun pat -> ...))`.

File Processing Using Asynchronous Workflows

We now show a slightly longer sample of asynchronous I/O processing. Our running sample is an application that must read a large number of image files and perform some processing on them. This kind of application may be *compute bound* (if the processing takes a long time and the file system is fast) or *I/O bound* (if the processing is quick and the file system is slow). Using asynchronous techniques tends to give good overall performance gains when an application is I/O bound and can also give performance improvements for compute-bound applications if asynchronous operations are executed in parallel on multicore machines.

Listing 13-6 shows a synchronous implementation of our image transformation program.

Listing 13-6. *A Synchronous Image Processor*

```
open System.IO
let numImages = 200
let size = 512
let numPixels = size * size

let MakeImageFiles() =
    printfn "making %d %dx%d images... " numImages size size
    let pixels = Array.init numPixels (fun i -> byte i)
    for i = 1 to numImages  do
        System.IO.File.WriteAllBytes(sprintf "Image%d.tmp" i, pixels)
    printfn "done."

let processImageRepeats = 20

let TransformImage(pixels, imageNum) =
    printfn "TransformImage %d" imageNum;
    // Perform a CPU-intensive operation on the image.
    pixels |> Func.repeatN processImageRepeats (Array.map (fun b -> b + 1uy))

let ProcessImageSync(i) =
    use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
    let pixels = Array.zero_create numPixels
    let nPixels = inStream.Read(pixels,0,numPixels);
    let pixels' = TransformImage(pixels,i)
    use outStream =  File.OpenWrite(sprintf "Image%d.done" i)
    outStream.Write(pixels',0,numPixels)

let ProcessImagesSync() =
    printfn "ProcessImagesSync...";
    for i in 1 .. numImages do
        ProcessImageSync(i)
```

We assume the image files are already created using the following code:

```
> System.Environment.CurrentDirectory <- __SOURCE_DIRECTORY__;;
val it : unit = ()

> MakeImageFiles();;
val it : unit = ()
```

We have left the transformation on the image largely unspecified, such as the function `TransformImage`. By changing the value of `processImageRepeats`, you can adjust the computation from compute bound to I/O bound.

The problem with this implementation is that each image is read and processed sequentially, when in practice multiple images can be read and transformed simultaneously, giving much greater throughput. Listing 13-7 shows the implementation of the image processor using an asynchronous workflow.

Listing 13-7. *The Asynchronous Image Processor*

```
open Microsoft.FSharp.Control
open Microsoft.FSharp.Control.CommonExtensions

let ProcessImageAsync(i) =
    async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
            let! pixels = inStream.ReadAsync(numPixels)
            let  pixels' = TransformImage(pixels,i)
            use outStream = File.OpenWrite(sprintf "Image%d.done" i)
            do! outStream.WriteAsync(pixels')  }

let ProcessImagesAsync() =
    printfn "ProcessImagesAsync...";
    let tasks = [ for i in 1 .. numImages -> ProcessImageAsync(i) ]
    Async.Run (Async.Parallel tasks)  |> ignore
    printfn "ProcessImagesAsync finished!";
```

On the one of our machines, the asynchronous version of the code ran up to three times as fast as the synchronous version (in total elapsed time), when `processImageRepeats` is 20 and `numImages` is 200. A factor of 2 was achieved consistently for any number of `processImageRepeats` since this machine had two CPUs.

Let's take a closer look at this code. The call `Async.Run (Async.Parallel ...)` executes a set of asynchronous operations in the thread pool, collects their results (or their exceptions), and returns the overall array of results to the original code. The core asynchronous workflow is introduced by the `async { ... }` construct. Let's look at the inner workflow line by line:

```
    async { use inStream = File.OpenRead(sprintf "Image%d.tmp" i)
            ... }
```

This line opened the input stream synchronously using `File.OpenRead`. Although this is a synchronous operation, the use of `use` indicates that the lifetime of the stream is managed over the remainder of the workflow. The stream will be closed when the variable is no longer in scope,

that is, at the end of the workflow, even if asynchronous activations occur in between. If any step in the workflow raises an uncaught exception, then the stream will also be closed while handling the exception.

The next line reads the input stream asynchronously using `inStream.ReadAsync`:

```
async { use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
        let! pixels = inStream.ReadAsync(numPixels)
        ... }
```

`Stream.ReadAsync` is an extension method added to the .NET `System.IO.Stream` class by opening the F# namespace `Microsoft.FSharp.Control.CommonExtensions`, and it generates a value of type `Async<byte[]>`. The use of `let!` executes this operation asynchronously and registers a callback. When the callback is invoked, the value `pixels` is bound to the result of the operation, and the remainder of the asynchronous workflow is executed. The next line transforms the image synchronously using `TransformImage`:

```
async { use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
        let! pixels = inStream.ReadAsync(numPixels)
        let  pixels' = TransformImage(pixels,i)
        ... }
```

Like the first line, the next line opens the output stream. Using `use` guarantees that the stream is closed by the end of the workflow regardless of whether exceptions are thrown in the remainder of the workflow.

```
async { use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
        let! pixels = inStream.ReadAsync(numPixels)
        let  pixels' = TransformImage(pixels,i)
        use outStream =  File.OpenWrite(sprintf "Image%d.done" i)
        ... }
```

The final line of the workflow performs an asynchronous write of the image. Once again, `WriteAsync` is an extension method added to the .NET `System.IO.Stream` class by opening the F# namespace `Microsoft.FSharp.Control.CommonExtensions`.

```
async { use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
        let! pixels = inStream.ReadAsync(numPixels)
        let  pixels' = TransformImage(pixels,i)
        use outStream =  File.OpenWrite(sprintf "Image%d.done" i)
        do! outStream.WriteAsync(pixels')  }
```

If you now return to the first part of the function, you can see that the overall operation of the function is to create `numImages` individual asynchronous operations, using a sequence expression that generates a list:

```
let tasks = [ for i in 1 .. numImages -> ProcessImageAsync(i) ]
```

You can then compose these tasks in parallel using `Async.Parallel` and then run the resulting process using `Async.Run`. This waits for the overall operation to complete and returns the result.

```
Async.Run (Async.Parallel tasks)
```

Table 13-3 shows some of the primitives and combinators commonly used to build asynchronous workflows. Take the time to compare Listings 13-7 and 13-6. Notice the following:

- The overall structure and flow of the core of Listing 13-7 is quite similar to Listing 13-6, that is, the synchronous algorithm, even though it includes steps executed asynchronously.
- The performance characteristics of Listing 13-7 are the same as those of Listing 13-6. Any overhead involved in executing the asynchronous workflow is easily dominated by the overall cost of I/O and image processing. It is also much easier to experiment with modifications such as making the write operation synchronous.

Table 13-3. *Some Common Primitives Used to Build* `Async<'a>` *Values*

Member/Type	Description
`Async.Catch: Async<'a> -> Async<Choice<'a,exn>>`	Catches any errors from an asynchronous computation and returns a `Choice` result indicating success or failure.
`Async.Primitive: ('a -> unit) * (exn -> unit) -> Async<'a>`	Builds a single primitive asynchronous step of an asynchronous computation. The function that implements the step is passed continuations to call once the step is complete or if the step fails.
`Async.Parallel: Async<#seq<'a>> -> Async<'a[]>`	Builds a single asynchronous computation that runs the given asynchronous computations in parallel and waits for results from all to be returned. Each may either terminate with a value or return an exception. If any raise an exception, then the others are cancelled, and the overall asynchronous computation also raises the same exception.

Running Asynchronous Computations

Values of type `Async<'a>` are usually run using the functions listed in Table 13-4. `Async<'a>` values can be built by using functions and members in the F# libraries.

Table 13-4. *Common Methods in the* `Async` *Type Used to Run* `Async<'a>` *Values*

Member/Type	Description
`Async.Run: Async<'a> -> 'a`	Runs an operation in the thread pool and waits for its result.
`Async.Spawn: Async<unit> -> unit`	Queues the asynchronous computation as an operation in the thread pool.
`Async.SpawnChild: Async<unit> -> Async<unit>`	Queues the asynchronous computation, initially as a work item in the thread pool, but inherits the cancellation handle from the current asynchronous computation.

Table 13-4. *Common Methods in the* `Async` *Type Used to Run* `Async<'a>` *Values*

Member/Type	Description
`Async.SpawnThenPostBack: Async<'a> * ('a -> unit) -> unit`	Queues the asynchronous computation, initially as a work item in the thread pool. When its result is available, executes the given callback by posting a message to the synchronization context of the thread that called `SpawnThenPostBack`. Useful for returning the results of asynchronous computations to a GUI application.
`Async.Future: Async<'a> -> Future<'a>`	Queues the asynchronous computation as an operation in the thread pool and returns an object that can be used to later rendezvous with its result.

Common I/O Operations in Asynchronous Workflows

Asynchronous programming is becoming more widespread because of the use of multicore machines and networks in applications, and many .NET APIs now come with both synchronous and asynchronous versions of their functionality. For example, all web service.APIs generated by .NET tools have asynchronous versions of their requests. A quick scan of the .NET API documentation on the Microsoft website reveals the asynchronous operations listed in Table 13-5. These all have equivalent `Async<'a>` operations defined in the F# libraries as extensions to the corresponding .NET types.

Table 13-5. *Some Asynchronous Operations in the .NET Libraries and Corresponding F# Extensions*

.NET Asynchronous Operation	F# Extension	Description
`Stream.Begin/EndRead`	`ReadAsync`	Read a stream of bytes asynchronously. See also `FileStream`, `NetworkStream`, `DeflateStream`, `IsolatedStorageFileStream`, and `SslStream`.
`Stream.Begin/EndWrite`	`WriteAsync`	Write a stream of bytes asynchronously. See also `FileStream`.
`Socket.BeginAccept/EndAccept`	`AcceptAsync`	Accept an incoming network socket request asynchronously.
`Socket.BeginReceive/EndRecevie`	`ReceiveAsync`	Receive data on a network socket asynchronously.
`Socket.BeginSend/EndSend`	`SendAsync`	Send data on a network socket asynchronously.
`WebRequest.Begin/EndGetResponse`	`GetResponseAsync`	Make an asynchronous web request. See also `FtpWebRequest`, `SoapWebRequest`, and `HttpWebRequest`.

Table 13-5. *Some Asynchronous Operations in the .NET Libraries and Corresponding F# Extensions*

.NET Asynchronous Operation	F# Extension	Description
`SqlCommand.Begin/EndExecuteReader`	`ExecuteReaderAsync`	Execute an `SqlCommand` asynchronously.
`SqlCommand.Begin/EndExecuteXmlReader`	`ExecuteXmlReaderAsync`	Execute a read of XML asynchronously.
`SqlCommand.Begin/EndExecuteNonQuery`	`ExecuteNonQueryAsync`	Execute a nonreading `SqlCommand` asynchronously.

Sometimes you may need to write a few primitives to map .NET asynchronous operations into the F# asynchronous framework. We give some examples later in this section and in Chapter 14.

Under the Hood: Implementing a Primitive Asynchronous Step

Let's take a moment to look at how to implement one of the primitive `Async<'a>` actions we've been using earlier in the chapter. Listing 13-8 shows the essence of the implementation of `Stream.ReadAsync`, which is a primitive asynchronous action that wraps a pair of `Stream.BeginRead` and `Stream.EndRead` calls using `Async.Primitive`. We implement this as an extension to the `System.IO.Stream` type to ensure it is easy to find the asynchronous version of this functionality alongside existing functions (extension members were described in Chapter 6).

Listing 13-8. *An Implementation of an* `Async.Primitive`

```
open System

let trylet f x = (try Choice2_1 (f x) with exn -> Choice2_2(exn))

let protect cont econt f x =
    match trylet f x with
    | Choice2_1 v -> cont v
    | Choice2_2 exn -> econt exn

type System.IO.Stream with
    member stream.ReadAsync (buffer,offset,count) =
       Async.Primitive (fun (cont,econt) ->
          stream.BeginRead
              (buffer=buffer,
               offset=offset,
               count=count,
               state=null,
               callback=AsyncCallback(protect cont econt stream.EndRead))
            |> ignore)
```

The type of `Async.Primitive` is as follows:

```
val Async.Primitive : (('a -> unit) * (exn -> unit) -> unit) -> Async<'a>
```

The inferred type of the extension to the `System.IO.Stream` type in Listing 13-8 is as follows:

```
type System.IO.Stream with
    member ReadAsync: buffer:byte[] * offset:int * count:int -> Async<int>
```

In Listing 13-8, `Async.Primitive` builds an `Async<int>` value, where the integer result indicates the number of bytes read from the stream. But what are all these function values? As you saw earlier, asynchronous computations work via *continuations*. This means a primitive step is given two continuation functions, `cont` and `econt`, which must be called upon success and/or exceptional failure of the operation. The previous implementation calls `BeginRead` and passes it a callback that will be invoked when the asynchronous operation returns. Note that the call to `BeginRead` uses named arguments, covered in Chapter 6. The callback calls `EndRead` to retrieve the result and passes this result to the success continuation `cont`; the call to `EndRead` is protected by an exception handler that calls the exception continuation `econt` should something go wrong.

The simple wrapper shown in Listing 13-8 now allows us to use `ReadAsync` in workflows, such as in the following line of our asynchronous image processor:

```
async { use inStream =  File.OpenRead(sprintf "Image%d.tmp" i)
        let! pixels = inStream.ReadAsync(numPixels)
        ... }
```

Note that the `econt` continuation of a primitive step should be called if an exception occurs. The example includes the try/catch handlers required to catch exceptions from `EndRead`. For more details, see the full implementation of `ReadAsync` and other similar wrappers in the F# library source code.

Under the Hood: Implementing Async.Parallel

`Async.Parallel` can appear magical. Computation tasks are created, executed, and resynchronized almost without effort. However, Listing 13-9 shows that a basic implementation of this operator is simple and again helps you see how `Async<'a>` values work "under the hood."

Listing 13-9. *A Basic Implementation of* `Async.Parallel`

```
let Parallel(taskSeq) =
    Async.Primitive (fun (cont,econt) ->
        let tasks = Seq.to_array taskSeq
        let count = ref tasks.Length
        let results = Array.zero_create tasks.Length
        tasks |> Array.iteri (fun i p ->
            Async.Spawn
               (async { let! res = p
                        do results.[i] <- res;
                        let n = System.Threading.Interlocked.Decrement(count)
                        do if n=0 then cont results })))
```

This basic implementation `Parallel` first converts the input task sequence to an array and then creates mutable state `count` and `results` to record the progress of the parallel computations. It then iterates through the tasks and queues each for execution in the .NET thread pool. Upon completion, each writes its result and decrements the counter using an atomic `Interlocked.Decrement` operator, discussed further in the section "Understanding Shared-Memory Concurrency" at the end of this chapter. The last process to finish calls the continuation with the collected results.

In practice, `Parallel` is implemented slightly differently to take into account exceptions and cancellation; once again, see the F# library code for full details.

Understanding Exceptions and Cancellation

Two recurring topics in asynchronous programming are exceptions and cancellation. Let's first explore some of the behavior of asynchronous programs with regard to exceptions.

```
> let failingTask = async { do failwith "fail" };;
val failingTask: Async<unit>

> Async.Run failingTask;;
Microsoft.FSharp.Core.FailureException: fail
stopped due to error

> let failingTasks = [ async { do failwith "fail A" };
                       async { do failwith "fail B" }; ];;
val failingTasks: Async<unit>

> Async.Run (Async.Parallel failingTasks);;
Microsoft.FSharp.Core.FailureException: fail A
stopped due to error

> Async.Run (Async.Parallel failingTasks);;
Microsoft.FSharp.Core.FailureException: fail B
stopped due to error
```

From this you can see the following:

- Tasks fail only when they are actually executed. The construction of a task using the `async { ... }` syntax will never fail.
- Tasks run using `Async.Run` report any failure back to the controlling thread as an exception.
- It is nondeterministic which task will fail first.
- Tasks composed using `Async.Parallel` report the first failure from amongst the collected set of tasks. An attempt is made to cancel other tasks by setting the cancellation flag for the group of tasks, and any further failures are ignored.

You can wrap a task using the `Async.Catch` combinator. This has the following type:

```
static member Catch : Async<'a> -> Async<Choice<'a,exn>>
```

For example:

```
> Async.Run (Async.Catch failingTask);;
val it : Choice<unit,exn> = Choice2_2 (FailureException ())
```

You can also handle errors by using `try/finally` in an `async { ... }` workflow.

■ **Note** You can find further information and examples of asynchronous workflows at `http://www.expert-fsharp.net/topics/AsyncWorkflows`.

Passing and Processing Messages

A distinction is often made between *shared-memory* concurrency and *message passing* concurrency. The former is often more efficient on local machines and is covered in the section "Using Shared-Memory Concurrency" later in this chapter. The latter scales to systems where there is no shared memory, for example, distributed systems, and can also be used to avoid performance problems associated with shared memory. Asynchronous message passing and processing is a common foundation for concurrent programming, and in this section we look at some simple examples of message-passing programs.

Introducing Message Processing

In a sense you have already seen a good deal of message passing in this chapter. For example:

- In the `BackgroundWorker` design pattern, the `CancelAsync` method is a simple kind of message.
- Whenever you raise events on a GUI thread from a background thread, you are, under the hood, posting a message to the GUI's event queue. On Windows this event queue is managed by the operating system, and the processing of the events on the GUI thread is called the Windows Event Loop.

In this section we cover a simple kind of message processing called *mailbox processing*. This is popular in languages such as Erlang. A *mailbox* is a message queue that you can scan for a message particularly relevant to the message-processing agent you are defining. Listing 13-10 shows a concurrent agent that implements a simple counter by processing a mailbox as messages arrive. The type `MailboxProcessor` is defined in the F# library module `Microsoft.FSharp.Control.Mailboxes`.

Listing 13-10. *Implementing a Counter Using a* `MailboxProcessor`

```
open Microsoft.FSharp.Control.Mailboxes

let counter =
    MailboxProcessor.Create(fun inbox ->
        let rec loop(n) =
            async { do printfn "n = %d, waiting..." n
                    let! msg = inbox.Receive()
                    return! loop(n+msg) }
        loop(0))
```

The type of counter is `MailboxProcessor<int>`, where the type argument indicates that this object expects to be sent messages of type `int`.

```
val counter : MailboxProcessor<int>
```

The "The Message Processing and State Machines" sidebar describes the general pattern of Listing 13-10 and the other `MailboxProcessor` examples in this chapter, all of which can be thought of as *state machines*. With this in mind, let's take a closer look at Listing 13-10. First let's use `counter` on some simple inputs:

```
> counter.Start();;
n = 0, waiting...

> counter.Post(1);;
n = 1, waiting...

> counter.Post(2);;
n = 3, waiting...

> counter.Post(1);;
n = 4, waiting...
```

Looking at Listing 13-10, note calling the `MailboxProcessor.Start` method causes the processing agent to enter `loop` with `n = 0`. The agent then performs an asynchronous `Receive` request on the `inbox` for the `MailboxProcessor`; that is, the agent waits asynchronously until a message has been received. When the message `msg` is received, the program calls `loop(n+msg)`. As additional messages are received, the internal "counter" (actually an argument) is incremented further.

We post messages to the agent using `mailbox.Post`. The type of `mailbox.Receive` is as follows:

```
member Mailbox<'msg>.Receive: unit -> Async<'msg>
```

Using an asynchronous receive ensures no "real" threads are blocked for the duration of the wait. This means the previous techniques scale to many thousands of concurrent agents.

MESSAGE PROCESSING AND STATE MACHINES

Listing 13-10 shares a common structure with many of the other message-processing components you'll be looking at in this chapter, all of which are *state machines*. This general structure is as follows:

```
let agent =
    MailboxProcessor.Start(fun inbox ->

        // The states of the state machine
        let rec state1(args) =  async { ... }
        and     state2(args) =  async { ... }
        ...
        and     stateN(args) =  async { ... }

        // Enter the initial state
        state1(initialArgs))
```

That is, message-processing components typically use sets of recursive functions, each defining an asynchronous computation. Each of these functions can be thought of as a state, and one of these states is identified as the initial state. Arguments may be passed between these states just as you pass them between any other set of recursive functions.

Creating Objects That React to Messages

Often it is wise to hide the internals of an asynchronous computation behind an object, since the use of message passing can be seen as an implementation detail. Furthermore, Listing 13-10 hasn't shown you how to retrieve information from the counter, except by printing it to the standard output. Furthermore, it hasn't shown how to ask the processing agent to exit. Listing 13-11 shows how to implement an object wrapping an agent that supports `Increment`, `Stop`, and `Fetch` messages.

Listing 13-11. *Hiding a Mailbox and Supporting a Fetch Method*

```
open Microsoft.FSharp.Control.Mailboxes

/// The internal type of messages for the agent
type internal msg = Increment of int | Fetch of IChannel<int> | Stop
```

```
type CountingAgent() =
    let counter = MailboxProcessor.Start(fun inbox ->
             // The states of the message-processing state machine...
             let rec loop(n) =
                async { let! msg = inbox.Receive()
                        match msg with
                        | Increment m ->
                            // increment and continue...
                            return! loop(n+m)
                        | Stop ->
                            // exit
                            return ()
                        | Fetch  replyChannel  ->
                            // post response to reply channel and continue
                            do replyChannel.Post(n)
                            return! loop(n) }

             // The initial state of the message-processing state machine...
             loop(0))

    member a.Increment(n) = counter.Post(Increment(n))
    member a.Stop() = counter.Post(Stop)
    member a.Fetch() = counter.PostSync(fun replyChannel -> Fetch(replyChannel))
```

The inferred public types indicate how the presence of a concurrent agent is successfully hidden by the use of an object:

```
type CountingAgent =
     new : unit -> CountingAgent
     member Fetch : unit -> int
     member Increment : n:int -> unit
     member Stop : unit -> unit
```

Here you can see an instance of this object in action:

```
> let counter = new CountingAgent();;
val counter : CountingAgent

> counter.Increment(1);;
val it : unit = ()

> counter.Fetch();;
val it : int = 1
```

```
> counter.Increment(2);;
val it : unit = ()

> counter.Fetch();;
val it : int = 3

> counter.Stop();;
val it : unit = ()
```

Listing 13-11 shows several important aspects of message passing and processing using the mailbox-processing model:

- Internal messages protocols are often represented using discriminated unions. Here the type `msg` has cases `Increment`, `Fetch`, and `Stop` corresponding to the three methods accepted by the object that wraps the overall agent implementation.
- Pattern matching over discriminated unions gives a succinct way to process messages. A common pattern is a call to `inbox.Receive()` or `inbox.TryReceive()` followed by a match on the message contents.
- The `PostSync` on the `MailboxProcessor` type gives a way to post a message and wait for a reply. A temporary *reply channel* is created and should form part of the message. A reply channel is simply an object of type `Microsoft.FSharp.Control.IChannel<'reply>`, which in turn simply supports a `Post` method. This can be used by the `MailboxProcessor` to post a reply to the waiting caller. In Listing 13-11 the channel is sent to the underlying message-processing agent `counter` as part of the `Fetch` message.

Table 13-6 summarizes the most important members available on the `MailboxProcessor` type.

Table 13-6. *Some Members of the `MailboxProcessor<'msg>` Type*

Member/Type	Description
`Post: 'msg -> unit`	Posts a message to a mailbox queue.
`Receive: ?timeout:int -> Async<'msg>`	Returns the next message in the mailbox queue. If no messages are present, performs an asynchronous wait until the message arrives. If a timeout occurs, then raises a `TimeoutException`.
`Scan: ('msg -> Async<'a> option) * ?timeout:int -> Async<'a>`	Scans the mailbox for a message where the function returns a `Some(_)` value. Returns the chosen result. If no messages are present, performs an asynchronous wait until more messages arrive. If a timeout occurs, then raises a `TimeoutException`.
`TryReceive : ?timeout:int -> Async<'msg option>`	Like `Receive`, but if a timeout occurs, then returns `None`.
`TryScan : ('msg -> Async<'a> option) * ?timeout:int -> Async<'a option>`	Like `Scan`, but if a timeout occurs, then returns `None`.

Scanning Mailboxes for Relevant Messages

It is common for a message-processing agent to end up in a state where it's not interested in all messages that might appear in a mailbox but only a subset of them. For example, you may be awaiting a reply from another agent and aren't interested in serving new requests. In this case, it is essential you use `MailboxProcessor.Scan` rather than `MailboxProcessor.Receive`. Table 13-6 shows the signatures of both of these. The former lets you choose between available messages by processing them in order, while the latter forces you to process every message. Listing 13-12 shows an example of using `Mailbox.Scan`.

Listing 13-12. *Scanning a Mailbox for Relevant Messages*

```
open Microsoft.FSharp.Control.Mailboxes

type msg =
    | Message1
    | Message2 of int
    | Message3 of string

let agent =
    MailboxProcessor.Start(fun inbox ->
        let rec loop() =
            inbox.Scan(function
                | Message1 ->
                   Some (async { do printfn "message 1!"
                                 return! loop() })
                | Message2 n ->
                   Some (async { do printfn "message 2!"
                                 return! loop() })
                | Message3 _ ->
                   None)
        loop())
```

We can now post these agent messages, including messages of the ignored kind `Message3`:

```
> agent.Post(Message1) ;;
message 1!
val it : unit = ()

> agent.Post(Message2(100));;
message 2!
val it : unit = ()

> agent.Post(Message3("abc"));;
val it : unit = ()
```

```
> agent.Post(Message2(100));;
message 2!
val it : unit = ()

> agent.UnsafeMessageQueueContents;;
val it : seq<msg> = seq [Message3("abc")]
```

When we sent `Message3` to the message processor, the message was ignored. However, the last line shows that the unprocessed `Message3` is still in the message queue, which we have examined by using the "backdoor" property `UnsafeMessageQueueContents`.

Note You can find further examples of asynchronous message processing with F# at `http://www.expert-fsharp.net/topics/MessageProcessing`.

Example: Asynchronous Web Crawling

At the start of this chapter we mentioned that the rise of the Web and other forms of networks is a major reason for the increasing importance of concurrent and asynchronous programming. Listing 13-13 shows an implementation of a web crawler using asynchronous programming and mailbox-processing techniques.

Listing 13-13. *A Scalable, Controlled Asynchronous Web Crawler*

```
open System.Collections.Generic
open System.Net
open System.IO
open System.Threading
open System.Text.RegularExpressions
open Microsoft.FSharp.Control
open Microsoft.FSharp.Control.Mailboxes
open Microsoft.FSharp.Control.CommonExtensions

let limit = 50
let linkPat = "href=\s*\"[^\"h]*(http://[^&\"]*)\""
let getLinks (txt:string) =
    [ for m in Regex.Matches(txt,linkPat)  -> m.Groups.Item(1).Value ]

let (<--) (mp: #IChannel<_>) x = mp.Post(x)

// A type that helps limit the number of active web requests
type RequestGate(n:int) =
    let semaphore = new Semaphore(initialCount=n,maximumCount=n)
    member x.AcquireAsync(?timeout) =
        async { let! ok = semaphore.WaitOneAsync(?millisecondsTimeout=timeout)
```

```
                if ok then
                   return
                     { new System.IDisposable with
                         member x.Dispose() =
                             semaphore.Release() |> ignore }
                else
                   return! failwith "couldn't acquire a semaphore" }

// Gate the number of active web requests
let webRequestGate = RequestGate(5)

// Fetch the URL, and post the results to the urlCollector.
let collectLinks (url:string) =
    async { // An Async web request with a global gate
            let! html =
                async { // Acquire an entry in the webRequestGate. Release
                        // it when 'holder' goes out of scope
                        use! holder = webRequestGate.AcquireAsync()

                        let req = WebRequest.Create(url,Timeout=5)

                        // Wait for the WebResponse
                        use! response = req.GetResponseAsync()

                        // Get the response stream
                        use reader = new StreamReader(response.GetResponseStream())

                        // Read the response stream
                        return! reader.ReadToEndAsync()  }

            // Compute the links, synchronously
            let links = getLinks html

            // Report, synchronously
            do printfn "finished reading %s, got %d links" url (List.length links)

            // We're done
            return links }

/// 'urlCollector' is a single agent that receives URLs as messages. It creates new
/// asynchronous tasks that post messages back to this object.
let urlCollector =
    MailboxProcessor.Start(fun self ->

        // This is the main state of the urlCollector
        let rec waitForUrl (visited : Set<string>) =
```

```
        async { // Check the limit
                if visited.Count < limit then

                    // Wait for a URL...
                    let! url = self.Receive()
                    if not (visited.Contains(url)) then
                        // Spawn off a new task for the new url. Each collects
                        // links and posts them back to the urlCollector.
                        do! Async.SpawnChild
                                (async { let! links = collectLinks url
                                         for link in links do
                                         do self <-- link })

                    // Recurse into the waiting state
                    return! waitForUrl(visited.Add(url)) }

    // This is the initial state.
    waitForUrl(Set.empty))
```

We can initiate a web crawl from a particular URL as follows:

```
> urlCollector <-- "http://news.google.com";;
finished reading http://news.google.com, got 191 links
finished reading http://news.google.com/?output=rss, got 0 links
finished reading http://www.ktvu.com/politics/13732578/detail.html, got 14 links
finished reading http://www.washingtonpost.com/wp-dyn/content/art..., got 218 links
finished reading http://www.newsobserver.com/politics/story/646..., got 56 links
finished reading http://www.foxnews.com/story/0,2933,290307,0...l, got 22 links
...
```

The key techniques shown in Listing 13-13 are as follows:

- The type `RequestGate` encapsulates the logic needed to ensure that we place a global limit on the number of active web requests occurring at any one point in time. This is instantiated to the particular instance `webRequestGate` with limit 5. This uses a `System.Threading.Semaphore` object to coordinate access to this "shared resource." Semaphores are discussed in more detail in the section "Using Shared-Memory Concurrency."
- The `RequestGate` type ensures that web requests sitting in the request queue do not block threads but rather wait asynchronously as callback items in the thread pool until a slot in the `webRequestGate` becomes available.
- The `collectLinks` function is a regular asynchronous computation. It first enters the `RequestGate` (that is, acquires one of the available entries in the `Semaphore`). Once a response has been received, it reads off the HTML from the resulting reader, scrapes the HTML for links using regular expressions, and returns the generated set of links.

- The `urlCollector` is the only message-processing program. It is written using a `MailboxProcessor`. In its main state it waits for a fresh URL and spawns a new asynchronous computation to call `collectLinks` once one is received. For each collected link a new message is sent back to the `urlCollector`'s mailbox. Finally, we recurse to the waiting state, having added the fresh URL to the overall set of URLs we have traversed so far.
- The operator `<--` is used as shorthand for posting a message to an agent. This is a recommended abbreviation in F# asynchronous programming.
- The `AcquireAsync` method of the `RequestGate` type uses a design pattern called a *holder*. The object returned by this method is an `IDisposable` object that represents the acquisition of a resource. This "holder" object is bound using `use`, and this ensures the resource is released when the computation completes or when the computation ends with an exception.

Listing 13-13 shows that it is relatively easy to create sophisticated, scalable asynchronous programs using a mix of message passing and asynchronous I/O techniques. Modern web crawlers have thousands of outstanding open connections, indicating the importance of using asynchronous techniques in modern scalable web-based programming.

Using Shared-Memory Concurrency

The final topics we cover in this chapter are the various "primitive" mechanisms used for threads, shared-memory concurrency, and signaling. In many ways, these are the "assembly language" of concurrency.

In this chapter we've concentrated mostly on techniques that work well with immutable data structures. That is not to say you should *always* use immutable data structures. It is, for example, perfectly valid to use mutable data structures as long as they are accessed from only one particular thread. Furthermore, private mutable data structures can often be safely passed through an asynchronous workflow, because at each point the mutable data structure will be accessed by only one thread, even if different parts of the asynchronous workflow are executed by different threads. This does not apply to workflows that use operators such as `Async.Parallel` or `Async.SpawnChild` that start additional threads of computation.

This means that we've largely avoided covering shared-memory primitives so far, because F# provides powerful declarative constructs such as asynchronous workflows and message passing that often subsume the need to resort to shared-memory concurrency. However, a working knowledge of thread primitives and shared-memory concurrency is still very useful, especially if you want to implement your own basic constructs or highly efficient concurrent algorithms on shared-memory hardware.

Creating Threads Explicitly

In this chapter we've avoided showing how to work with threads directly, instead relying on abstractions such as `BackgroundWorker` and the .NET thread pool. If you do want to create threads directly, here is a short sample:

```
open System.Threading
let t = new Thread(ThreadStart(fun _ ->
                printfn "Thread %d: Hello" Thread.CurrentThread.ManagedThreadId));
t.Start();
printfn "Thread %d: Waiting!" Thread.CurrentThread.ManagedThreadId
t.Join();
printfn "Done!"
```

When run, this gives the following:

```
val t : Thread

Thread 1: Waiting!
Thread 10: Hello
Done!
```

■ **Caution** Always avoid using `Thread.Suspend`, `Thread.Resume`, and `Thread.Abort`. These are a guaranteed way to put obscure concurrency bugs in your program! The MSDN website has a good description of why `Thread.Abort` may not even succeed. One of the only compelling uses for `Thread.Abort` is to implement Ctrl+C in an interactive development environment for a general-purpose language such as F# Interactive.

Shared Memory, Race Conditions, and the .NET Memory Model

Many multithreaded applications use mutable data structures shared between multiple threads. Without synchronization, these data structures will almost certainly become corrupt, because threads may read data that has been only partially updated (because not all mutations are *atomic*), or two threads may write to the same data simultaneously (a *race condition*). Mutable data structures are usually protected by *locks*, though lock-free mutable data structures are also possible.

Shared-memory concurrency is a difficult and complicated topic, and a considerable amount of good material on .NET shared-memory concurrency is available on the Web. All this material applies to F# when programming with mutable data structures such as reference cells, arrays, and hash tables when the data structures can be accessed from multiple threads simultaneously. F# mutable data structures map to .NET memory in fairly predictable ways; for example, mutable references become mutable fields in a .NET class, and mutable fields of word size can be assigned atomically.

On modern microprocessors multiple threads can see views of memory that are not consistent; that is, not all writes are propagated to all threads immediately. The guarantees given are called a *memory model* and are usually expressed in terms of the ordering dependencies between instructions that read/write memory locations. This is, of course, deeply troubling, because you have to think about a huge number of possible reorderings of your code, and it is one of

the main reasons why shared mutable data structures are difficult to work with. You can find further details on the .NET memory model at http://www.expert-fsharp.net/topics/MemoryModel.

Using Locks to Avoid Race Conditions

Locks are the simplest way to enforce mutual exclusion between two threads attempting to read or write the same mutable memory location. Listing 13-14 shows an example of code with a race condition.

Listing 13-14. *Shared-Memory Code with a Race Condition*

```
type MutablePair<'a,'b>(x:'a,y:'b) =
    let mutable currentX = x
    let mutable currentY = y
    member p.Value = (currentX,currentY)
    member p.Update(x,y) =
        // Race condition: This pair of updates is not atomic
        currentX <- x;
        currentY <- y

let p = new MutablePair<_,_>(1,2)
do Async.Spawn (async { do (while true do p.Update(10,10)) })
do Async.Spawn (async { do (while true do p.Update(20,20)) })
```

Here is the definition of the F# `lock` function:

```
open System.Threading
let lock (lockobj :> obj) f  =
    Monitor.Enter(lockobj);
    try
        f()
    finally
        Monitor.Exit(lockobj)
```

The pair of mutations in the `Update` method is not atomic; that is, one thread may have written to `currentX`, another then writes to both `currentX` and `currentY`, and the final thread then writes to `currentY`, leaving the pair holding the value `(10,20)` or `(20,10)`. Mutable data structures are inherently prone to this kind of problem if shared between multiple threads. Luckily, of course, F# code tends to have fewer mutations than imperative languages, because functions normally take immutable values and return a calculated value. However, when you do use mutable data structures, they should not be shared between threads, or you should design them carefully and document their properties with respect to multithreaded access.

Here is one way to use the F# `lock` function to ensure that updates to the data structure are atomic. Locks would also be required on uses of the property `p.Value`.

```
do Async.Spawn (async { do (while true do lock p (fun () -> p.Update(10,10))) })
do Async.Spawn (async { do (while true do lock p (fun () -> p.Update(20,20))) })
```

■ **Caution** If you use locks inside data structures, then do so only in a simple way that uses them to enforce just the concurrency properties you have documented. Don't lock "just for the sake of it," and don't hold locks longer than necessary. In particular, beware of making indirect calls to externally supplied function values, interfaces, or abstract members while a lock is held. The code providing the implementation may not be expecting to be called when a lock is held and may attempt to acquire further locks in an inconsistent fashion.

Using ReaderWriterLock

It is common that mutable data structures get read more than they are written. Indeed, mutation is often used only to initialize a mutable data structure. In this case, you can use a .NET `ReaderWriterLock` to protect access to a resource. The following two functions are provided in the F# library module `Microsoft.FSharp.Control.SharedMemory.Helpers`:

```
open System.Threading

let readLock (rwlock : ReaderWriterLock) f  =
  rwlock.AcquireReaderLock(Timeout.Infinite)
  try
      f()
  finally
      rwlock.ReleaseReaderLock()

let writeLock (rwlock : ReaderWriterLock) f  =
  rwlock.AcquireWriterLock(Timeout.Infinite)
  try
      f();
      Thread.MemoryBarrier()
  finally
      rwlock.ReleaseWriterLock()
```

Listing 13-15 shows how to use these functions to protect the `MutablePair` class.

Listing 13-15. *Shared-Memory Code with a Race Condition*

```
type MutablePair<'a,'b>(x:'a,y:'b) =
    let mutable currentX = x
    let mutable currentY = y
    let rwlock = new ReaderWriterLock()
    member p.Value =
        readLock rwlock (fun () ->
            (currentX,currentY))
    member p.Update(x,y) =
        writeLock rwlock (fun () ->
            currentX <- x;
            currentY <- y)
```

Some Other Concurrency Primitives

Table 13-7 shows some of the other shared-memory concurrency primitives available in the .NET Framework.

Table 13-7. *.NET Shared-Memory Concurrency Primitives*

Type	Description
`System.Threading.WaitHandle`	A synchronization object for signaling the control of threads.
`System.Threading.AutoResetEvent`	A two-state (on/off) `WaitHandle` that resets itself to "off" automatically after the signal is read. Similar to a two-state traffic light.
`System.Threading.ManualResetEvent`	A two-state (on/off) `WaitHandle` that requires a call to `ManualResetEvent.Reset()` to set it "off."
`System.Threading.Mutex`	A lock-like object that can be shared between operating system processes.
`System.Threading.Semaphore`	Used to limit the number of threads simultaneously accessing a resource. However, use a mutex or lock if at most one thread can access a resource at a time.
`System.Threading.Interlocked`	Atomic operations on memory locations. Especially useful for atomic operations on F# reference cells.

Summary

In this chapter, we covered concurrent, reactive, and asynchronous programming, which is a set of topics of growing importance in modern programming because of the widespread adoption of multicore microprocessors, network-aware applications, and asynchronous I/O channels. We've covered in depth background processing and a powerful F# construct called asynchronous workflows. Finally, we covered applications of asynchronous workflows to message-processing agents and web crawling, and we covered some of the shared-memory primitives for concurrent programming on the .NET platform. In the next chapter, we'll look at web programming, from serving web pages to delivering applications via web browsers.