Threading enables your Visual Basic program to perform concurrent processing so that you can do more than one operation at a time. For example, you can use threading to monitor input from the user, perform background tasks, and handle simultaneous streams of input.

Threads have the following properties:

- Threads enable your program to perform concurrent processing.
- The .NET Framework `System.Threading` namespace makes using threads easier.
- Threads share the application's resources. For more information, see Using Threads and Threading.

By default, a Visual Basic program has one thread. However, auxiliary threads can be created and used to execute code in parallel with the primary thread. These threads are often called *worker threads*.

Worker threads can be used to perform time-consuming or time-critical tasks without tying up the primary thread. For example, worker threads are often used in server applications to fulfill incoming requests without waiting for the previous request to be completed. Worker threads are also used to perform "background" tasks in desktop applications so that the main thread—which drives user interface elements—remains responsive to user actions.

Threading solves problems with throughput and responsiveness, but it can also introduce resource-sharing issues such as deadlocks and race conditions. Multiple threads are best for tasks that require different resources such as file handles and network connections. Assigning multiple threads to a single resource is likely to cause synchronization issues, and having threads frequently blocked when waiting for other threads defeats the purpose of using multiple threads.

A common strategy is to use worker threads to perform time-consuming or time-critical tasks that do not require many of the resources used by other threads. Naturally, some resources in your program must be accessed by multiple threads. For these cases, the `System.Threading` namespace provides classes for synchronizing threads. These classes include `Mutex`, `Monitor`, `Interlocked`, `AutoResetEvent`, and `ManualResetEvent`.

You can use some or all these classes to synchronize the activities of multiple threads, but some support for threading is supported by the Visual Basic language. For example, the `SyncLock Statement` provides synchronization features through implicit use of `Monitor`.

**Note**

Beginning with the .NET Framework 4, multithreaded programming is greatly simplified with the `System.Threading.Tasks.Parallel` and `System.Threading.Tasks.Task` classes, Parallel LINQ (PLINQ), new concurrent collection classes in the `System.Collections.Concurrent` namespace, and a new programming model that is based on the concept of tasks rather than threads. For more information, see Parallel Programming in the .NET Framework.
## Related Topics

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreaded Applications (Visual Basic)</td>
<td>Describes how to create and use threads.</td>
</tr>
<tr>
<td>Parameters and Return Values for Multithreaded Procedures</td>
<td>Describes how to pass and return parameters with multithreaded applications.</td>
</tr>
<tr>
<td>(Visual Basic)</td>
<td></td>
</tr>
<tr>
<td>Walkthrough: Multithreading with the BackgroundWorker</td>
<td>Shows how to create a simple multithreaded application.</td>
</tr>
<tr>
<td>Component (Visual Basic)</td>
<td></td>
</tr>
<tr>
<td>Thread Synchronization (Visual Basic)</td>
<td>Describes how to control the interactions of threads.</td>
</tr>
<tr>
<td>Thread Timers (Visual Basic)</td>
<td>Describes how to run procedures on separate threads at fixed intervals.</td>
</tr>
<tr>
<td>Thread Pooling (Visual Basic)</td>
<td>Describes how to use a pool of worker threads that are managed by the system.</td>
</tr>
<tr>
<td>How to: Use a Thread Pool (Visual Basic)</td>
<td>Demonstrates synchronized use of multiple threads in the thread pool.</td>
</tr>
<tr>
<td>Managed Threading</td>
<td>Describes how to implement threading in the .NET Framework.</td>
</tr>
</tbody>
</table>

© 2016 Microsoft
Multithreaded Applications (Visual Basic)

With Visual Basic, you can write applications that perform multiple tasks at the same time. Tasks with the potential of holding up other tasks can execute on separate threads, a process known as multithreading or free threading.

Applications that use multithreading are more responsive to user input because the user interface stays active as processor-intensive tasks execute on separate threads. Multithreading is also useful when you create scalable applications, because you can add threads as the workload increases.

Creating and Using Threads

If you need more control over the behavior of your application's threads, you can manage the threads yourself. However, realize that writing correct multithreaded applications can be difficult: Your application may stop responding or experience transient errors caused by race conditions. For more information, see Thread-Safe Components.

You create a new thread by declaring a variable of type `Thread` and calling the constructor, providing the name of the procedure or method that you want to execute on the new thread. The following code provides an example.

```vbnet
Dim newThread As New System.Threading.Thread(AddressOf AMethod)
```

Starting and Stopping Threads

To start the execution of a new thread, use the `Start` method, as shown in the following code.

```vbnet
newThread.Start()
```

To stop the execution of a thread, use the `Abort` method, as shown in the following code.

```vbnet
newThread.Abort()
```

Besides starting and stopping threads, you can also pause threads by calling the `Sleep` or `Suspend` method, resume a suspended thread by using the `Resume` method, and destroy a thread by using the `Abort` method.

Thread Methods

The following table shows some of the methods that you can use to control individual threads.
<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Causes a thread to start to run.</td>
</tr>
<tr>
<td>Sleep</td>
<td>Pauses a thread for a specified time.</td>
</tr>
<tr>
<td>Suspend</td>
<td>Pauses a thread when it reaches a safe point.</td>
</tr>
<tr>
<td>Abort</td>
<td>Stops a thread when it reaches a safe point.</td>
</tr>
<tr>
<td>Resume</td>
<td>Restarts a suspended thread</td>
</tr>
<tr>
<td>Join</td>
<td>Causes the current thread to wait for another thread to finish. If used with a time-out value, this method returns <strong>True</strong> if the thread finishes in the allocated time.</td>
</tr>
</tbody>
</table>

**Safe Points**

Most of these methods are self-explanatory, but the concept of *safe points* may be new to you. Safe points are locations in code where it is safe for the common language runtime to perform automatic *garbage collection*, the process of releasing unused variables and reclaiming memory. When you call the *Abort* or *Suspend* method of a thread, the common language runtime analyzes the code and determines the location of an appropriate location for the thread to stop running.

**Thread Properties**
Threads also contain several useful properties, as shown in the following table:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsAlive</td>
<td>Contains the value <strong>True</strong> if a thread is active.</td>
</tr>
<tr>
<td>IsBackground</td>
<td>Gets or sets a Boolean that indicates if a thread is or should be a background thread. Background threads are like foreground threads, but a background thread does not prevent a process from stopping. Once all foreground threads that belong to a process have stopped, the common language runtime ends the process by calling the <strong>Abort</strong> method on background threads that are still alive.</td>
</tr>
<tr>
<td>Name</td>
<td>Gets or sets the name of a thread. Most frequently used to discover individual threads when you debug.</td>
</tr>
<tr>
<td>Priority</td>
<td>Gets or sets a value that is used by the operating system to prioritize thread scheduling.</td>
</tr>
<tr>
<td>ThreadState</td>
<td>Contains a value that describes a thread's state or states.</td>
</tr>
</tbody>
</table>

**Thread Priorities**

Every thread has a priority property that determines how big or small a slice of processor time it has to execute. The operating system allocates longer time slices to high-priority threads and shorter time slices to low-priority threads. New threads are created with the value of **Normal**, but you can change the **Priority** property to any value in the **ThreadPriority** enumeration.

See **ThreadPriority** for a detailed description of the various thread priorities.

**Foreground and Background Threads**

A **foreground thread** runs indefinitely, whereas a **background thread** stops as soon as the last foreground thread has stopped. You can use the **IsBackground** property to determine or change the background status of a thread.

**See Also**

- Thread
- Thread Synchronization (Visual Basic)
- Parameters and Return Values for Multithreaded Procedures (Visual Basic)
- Threading (Visual Basic)
Parameters and Return Values for Multithreaded Procedures (Visual Basic)

Supplying and returning values in a multithreaded application is complicated because the constructor for the thread class must be passed a reference to a procedure that takes no arguments and returns no value. The following sections show some simple ways to supply parameters and return values from procedures on separate threads.

Supplying Parameters for Multithreaded Procedures

The best way to supply parameters for a multithreaded method call is to wrap the target method in a class and define fields for that class that will serve as parameters for the new thread. The advantage of this approach is that you can create a new instance of the class, with its own parameters, every time you want to start a new thread. For example, suppose you have a function that calculates the area of a triangle, as in the following code:

```vbnet
Function CalcArea(ByVal Base As Double, ByVal Height As Double) As Double
    CalcArea = 0.5 * Base * Height
End Function
```

You can write a class that wraps the `CalcArea` function and creates fields to store input parameters, as follows:

```vbnet
Class AreaClass
    Public Base As Double
    Public Height As Double
    Public Area As Double
    Sub CalcArea()
        Area = 0.5 * Base * Height
        MessageBox.Show("The area is: " & Area.ToString)
    End Sub
End Class
```

To use the `AreaClass`, you can create an `AreaClass` object, and set the `Base` and `Height` properties as shown in the following code:

```vbnet
Protected Sub TestArea()
    Dim AreaObject As New AreaClass
    Dim Thread As New System.Threading.Thread(
        AddressOf AreaObject.CalcArea)
```
Notice that the `TestArea` procedure does not check the value of the `Area` field after calling the `CalcArea` method. Because `CalcArea` runs on a separate thread, the `Area` field is not guaranteed to be set if you check it immediately after calling `Thread.Start`. The next section discusses a better way to return values from multithreaded procedures.

### Returning Values from Multithreaded Procedures

Returning values from procedures that run on separate threads is complicated by the fact that the procedures cannot be functions and cannot use `ByRef` arguments. The easiest way to return values is to use the `BackgroundWorker` component to manage your threads and raise an event when the task is done, and process the results with an event handler.

The following example returns a value by raising an event from a procedure running on a separate thread:

```vb
Private Class AreaClass2
    Public Base As Double
    Public Height As Double

    Function CalcArea() As Double
        ' Calculate the area of a triangle.
        Return 0.5 * Base * Height
    End Function
End Class

Private WithEvents BackgroundWorker1 As New System.ComponentModel.BackgroundWorker

Private Sub TestArea2()
    Dim AreaObject2 As New AreaClass2
    AreaObject2.Base = 30
    AreaObject2.Height = 40

    ' Start the asynchronous operation.
    BackgroundWorker1.RunWorkerAsync(AreaObject2)
End Sub

' This method runs on the background thread when it starts.
Private Sub BackgroundWorker1_DoWork(ByVal sender As Object, ByVal e As System.ComponentModel.DoWorkEventArgs) Handles BackgroundWorker1.DoWork
    Dim AreaObject2 As AreaClass2 = CType(e.Argument, AreaClass2)
    ' Return the value through the Result property.
    e.Result = AreaObject2.CalcArea()
End Sub

' This method runs on the main thread when the background thread finishes.
Private Sub BackgroundWorker1_DoWork_Completed(ByVal sender As Object, ByVal e As System.ComponentModel.DoWorkCompletedEventArgs) Handles BackgroundWorker1.DoWork_Completed

    ' Process results.
End Sub
```
You can provide parameters and return values to thread-pool threads by using the optional `ByVal` state-object variable of the `QueueUserWorkItem` method. Thread-timer threads also support a state object for this purpose. For information on thread pooling and thread timers, see Thread Pooling (Visual Basic) and Thread Timers (Visual Basic).

**See Also**

- Walkthrough: Multithreading with the BackgroundWorker Component (Visual Basic)
- Thread Pooling (Visual Basic)
- Thread Synchronization (Visual Basic)
- Events (Visual Basic)
- Multithreaded Applications (Visual Basic)
- Delegates (Visual Basic)
- Multithreading in Components

© 2016 Microsoft
Walkthrough: Multithreading with the BackgroundWorker Component (Visual Basic)

Visual Studio 2015

This walkthrough demonstrates how to create a multithreaded Windows Forms application that searches a text file for occurrences of a word. It demonstrates:

- Defining a class with a method that can be called by the BackgroundWorker component.
- Handling events raised by the BackgroundWorker component.
- Starting a BackgroundWorker component to run a method.
- Implementing a Cancel button that stops the BackgroundWorker component.

To create the user interface

1. Open a new Visual Basic Windows Forms Application project, and create a form named Form1.
2. Add two buttons and four text boxes to Form1.
3. Name the objects as shown in the following table.

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>First button</td>
<td>Name, Text</td>
<td>Start, Start</td>
</tr>
<tr>
<td>Second button</td>
<td>Name, Text</td>
<td>Cancel, Cancel</td>
</tr>
<tr>
<td>First text box</td>
<td>Name, Text</td>
<td>SourceFile, &quot;&quot;</td>
</tr>
<tr>
<td>Second text box</td>
<td>Name, Text</td>
<td>CompareString, &quot;&quot;</td>
</tr>
<tr>
<td>Third text box</td>
<td>Name, Text</td>
<td>WordsCounted, &quot;0&quot;</td>
</tr>
<tr>
<td>Fourth text box</td>
<td>Name, Text</td>
<td>LinesCounted, &quot;0&quot;</td>
</tr>
</tbody>
</table>

4. Add a label next to each text box. Set the Text property for each label as shown in the following table.
Object | Property | Setting
---|---|---
First label | Text | Source File
Second label | Text | Compare String
Third label | Text | Matching Words
Fourth label | Text | Lines Counted

To create a BackgroundWorker component and subscribe to its events

1. Add a **BackgroundWorker** component from the **Components** section of the **ToolBox** to the form. It will appear in the form’s component tray.

2. Set the following properties for the BackgroundWorker1 object.

<table>
<thead>
<tr>
<th>Property Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WorkerReportsProgress</strong></td>
</tr>
<tr>
<td><strong>WorkerSupportsCancellation</strong></td>
</tr>
</tbody>
</table>

To define the method that will run on a separate thread

1. From the **Project** menu, choose **Add Class** to add a class to the project. The **Add New Item** dialog box is displayed.
2. Select **Class** from the templates window and enter **Words.vb** in the name field.
3. Click **Add**. The **Words** class is displayed.
4. Add the following code to the **Words** class:

```vb
Public Class Words
    ' Object to store the current state, for passing to the caller.
    Public Class CurrentState
        Public LinesCounted As Integer
        Public WordsMatched As Integer
    End Class

    Public SourceFile As String
    Public CompareString As String
```
Private WordCount As Integer = 0
Private LinesCounted As Integer = 0

Public Sub CountWords(
    ByVal worker As System.ComponentModel.BackgroundWorker,
    ByVal e As System.ComponentModel.DoWorkEventArgs
)
    ' Initialize the variables.
    Dim state As New CurrentState
    Dim line As String
    Dim elapsedTime As Integer = 20
    Dim lastReportDateTime As Date = Now

    If CompareString Is Nothing OrElse CompareString = System.String.Empty Then
        Throw New Exception("CompareString not specified.")
    End If

    Using myStream As New System.IO.StreamReader(SourceFile)
        ' Process lines while there are lines remaining in the file.
        Do While Not myStream.EndOfStream
            If worker.CancellationPending Then
                e.Cancel = True
                Exit Do
            Else
                line = myStream.ReadLine
                WordCount += CountInString(line, CompareString)
                LinesCounted += 1
            End If
            ' Raise an event so the form can monitor progress.
            If Now > lastReportDateTime.AddMilliseconds(elapsedTime) Then
                state.LinesCounted = LinesCounted
                state.WordsMatched = WordCount
                worker.ReportProgress(0, state)
                lastReportDateTime = Now
            End If
            ' Uncomment for testing.
            ' System.Threading.Thread.Sleep(5)
        Loop
        ' Report the final count values.
        state.LinesCounted = LinesCounted
        state.WordsMatched = WordCount
        worker.ReportProgress(0, state)
    End Using
End Sub

Private Function CountInString(
    ByVal SourceString As String,
    ByVal CompareString As String
)
To handle events from the thread

- Add the following event handlers to your main form:

```vb
Private Sub BackgroundWorker1_RunWorkerCompleted(
    ByVal sender As Object,
    ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs
) Handles BackgroundWorker1.RunWorkerCompleted

    ' This event handler is called when the background thread finishes.
    ' This method runs on the main thread.
    If e.Error IsNot Nothing Then
        MessageBox.Show("Error: " & e.Error.Message)
    ElseIf e.Canceled Then
        MessageBox.Show("Word counting canceled.")
    Else
        MessageBox.Show("Finished counting words.")
    End If
End Sub

Private Sub BackgroundWorker1_ProgressChanged(
    ByVal sender As Object,
    ByVal e As System.ComponentModel.ProgressChangedEventArgs
) Handles BackgroundWorker1.ProgressChanged

    ' This method runs on the main thread.
    Dim state As Words.CurrentState = CType(e.UserState, Words.CurrentState)
```
To start and call a new thread that runs the WordCount method

1. Add the following procedures to your program:

```
Private Sub BackgroundWorker1_DoWork(
    ByVal sender As Object,
    ByVal e As System.ComponentModel.DoWorkEventArgs
) Handles BackgroundWorker1.DoWork
    ' This event handler is where the actual work is done.
    ' This method runs on the background thread.
    Dim worker As System.ComponentModel.BackgroundWorker = CType(sender, System.ComponentModel.BackgroundWorker)
    Dim WC As Words = CType(e.Argument, Words)
    WC.CountWords(worker, e)
End Sub

Sub StartThread()
    ' This method runs on the main thread.
    Me.WordsCounted.Text = "0"
    ' Initialize the object that the background worker calls.
    Dim WC As New Words
    WC.CompareString = Me.CompareString.Text
    WC.SourceFile = Me.SourceFile.Text
    ' Start the asynchronous operation.
    BackgroundWorker1.RunWorkerAsync(WC)
End Sub
```

2. Call the StartThread method from the Start button on your form:

```
Private Sub Start_Click() Handles Start.Click
    StartThread()
End Sub
```

To implement a Cancel button that stops the thread:
• Call the StopThread procedure from the Click event handler for the Cancel button.

```
Private Sub Cancel_Click() Handles Cancel.Click
    ' Cancel the asynchronous operation.
    Me.BackgroundWorker1.CancelAsync()
End Sub
```

### Testing
You can now test the application to make sure it works correctly.

**To test the application**

1. Press F5 to run the application.

2. When the form is displayed, enter the file path for the file you want to test in the sourceFile box. For example, assuming your test file is named Test.txt, enter C:\Test.txt.

3. In the second text box, enter a word or phrase for the application to search for in the text file.

4. Click the Start button. The LinesCounted button should begin incrementing immediately. The application displays the message “Finished Counting” when it is done.

**To test the Cancel button**

1. Press F5 to start the application, and enter the file name and search word as described in the previous procedure. Make sure that the file you choose is large enough to ensure you will have time to cancel the procedure before it is finished.

2. Click the Start button to start the application.

3. Click the Cancel button. The application should stop counting immediately.

### Next Steps
This application contains some basic error handling. It detects blank search strings. You can make this program more robust by handling other errors, such as exceeding the maximum number of words or lines that can be counted.

### See Also
- Threading (Visual Basic)
- Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic
- How to: Subscribe to and Unsubscribe from Events (C# Programming Guide)
Thread Synchronization (Visual Basic)

Visual Studio 2015

The following sections describe features and classes that can be used to synchronize access to resources in multithreaded applications.

One of the benefits of using multiple threads in an application is that each thread executes asynchronously. For Windows applications, this allows time-consuming tasks to be performed in the background while the application window and controls remain responsive. For server applications, multithreading provides the ability to handle each incoming request with a different thread. Otherwise, each new request would not get serviced until the previous request had been fully satisfied.

However, the asynchronous nature of threads means that access to resources such as file handles, network connections, and memory must be coordinated. Otherwise, two or more threads could access the same resource at the same time, each unaware of the other’s actions. The result is unpredictable data corruption.

For simple operations on integral numeric data types, synchronizing threads can be accomplished with members of the Interlocked class. For all other data types and non thread-safe resources, multithreading can only be safely performed using the constructs in this topic.

For background information on multithreaded programming, see:

- Managed Threading Basics
- Using Threads and Threading
- Managed Threading Best Practices

The lock and SyncLock Keywords

The Visual Basic SyncLock statement can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block.

A SyncLock statement is given an object as an argument, and is followed by a code block that is to be executed by only one thread at a time. For example:

```vb
Public Class TestThreading
    Dim lockThis As New Object

    Public Sub Process()
        SyncLock lockThis
            ' Access thread-sensitive resources.
        End SyncLock
    End Sub
```

The argument provided to the **SyncLock** keyword must be an object based on a reference type, and is used to define the scope of the lock. In the example above, the lock scope is limited to this function because no references to the object `lockThis` exist outside the function. If such a reference did exist, lock scope would extend to that object. Strictly speaking, the object provided is used solely to uniquely identify the resource being shared among multiple threads, so it can be an arbitrary class instance. In practice, however, this object usually represents the resource for which thread synchronization is necessary. For example, if a container object is to be used by multiple threads, then the container can be passed to lock, and the synchronized code block following the lock would access the container. As long as other threads locks on the same contain before accessing it, then access to the object is safely synchronized.

Generally, it is best to avoid locking on a **public** type, or on object instances beyond the control of your application. For example, `lockThis` can be problematic if the instance can be accessed publicly, because code beyond your control may lock on the object as well. This could create deadlock situations where two or more threads wait for the release of the same object. Locking on a public data type, as opposed to an object, can cause problems for the same reason. Locking on literal strings is especially risky because literal strings are *interned* by the common language runtime (CLR). This means that there is one instance of any given string literal for the entire program, the exact same object represents the literal in all running application domains, on all threads. As a result, a lock placed on a string with the same contents anywhere in the application process locks all instances of that string in the application. As a result, it is best to lock a private or protected member that is not interned. Some classes provide members specifically for locking. The **Array** type, for example, provides `SyncRoot`. Many collection types provide a `SyncRoot` member as well.

For more information about the **SyncLock** statement, see the following topics:

- **SyncLock Statement**
- **Monitors**

### Monitors

Like the **SyncLock** keyword, monitors prevent blocks of code from simultaneous execution by multiple threads. The **Enter** method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls **Exit**. This is just like using the **SyncLock** keyword. For example:

```vb
SyncLock x
    DoSomething()
End SyncLock
```

This is equivalent to:

```vb
Dim obj As Object = CType(x, Object)
System.Threading.Monitor.Enter(obj)
Try
    DoSomething()
Finally
```

...
Using the **SyncLock** keyword is generally preferred over using the **Monitor** class directly, both because **SyncLock** is more concise, and because **SyncLock** insures that the underlying monitor is released, even if the protected code throws an exception. This is accomplished with the **Finally** keyword, which executes its associated code block regardless of whether an exception is thrown.

---

### Synchronization Events and Wait Handles

Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another. This requires **synchronization events**, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled. If a thread attempts to wait on an event that is already signaled, then the thread continues to execute without delay.

There are two kinds of synchronization events: **AutoResetEvent**, and **ManualResetEvent**. They differ only in that **AutoResetEvent** changes from signaled to unsignaled automatically any time it activates a thread. Conversely, a **ManualResetEvent** allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its **Reset** method is called.

Threads can be made to wait on events by calling one of the wait methods, such as **WaitOne**, **WaitAny**, or **WaitAll**. **WaitHandle.WaitOne** causes the thread to wait until a single event becomes signaled, **WaitHandle.WaitAny** blocks a thread until one or more indicated events become signaled, and **WaitHandle.WaitAll** blocks the thread until all of the indicated events become signaled. An event becomes signaled when its **Set** method is called.

In the following example, a thread is created and started by the **Main** function. The new thread waits on an event using the **WaitOne** method. The thread is suspended until the event becomes signaled by the primary thread that is executing the **Main** function. Once the event becomes signaled, the auxiliary thread returns. In this case, because the event is only used for one thread activation, either the **AutoResetEvent** or **ManualResetEvent** classes could be used.

```vb
Imports System.Threading
Module Module1
    Dim autoEvent As AutoResetEvent

    Sub DoWork()
        Console.WriteLine("worker thread started, now waiting on event...")
        autoEvent.WaitOne()
        Console.WriteLine("worker thread reactivated, now exiting...")
    End Sub

    Sub Main()
        autoEvent = New AutoResetEvent(False)
        Console.WriteLine("main thread starting worker thread...")
        Dim t As New Thread(AddressOf DoWork)
        t.Start()
    End Sub
```

---


3 of 5  03.09.2016 17:48
Mutex Object

A mutex is similar to a monitor; it prevents the simultaneous execution of a block of code by more than one thread at a time. In fact, the name "mutex" is a shortened form of the term "mutually exclusive." Unlike monitors, however, a mutex can be used to synchronize threads across processes. A mutex is represented by the Mutex class.

When used for inter-process synchronization, a mutex is called a named mutex because it is to be used in another application, and therefore it cannot be shared by means of a global or static variable. It must be given a name so that both applications can access the same mutex object.

Although a mutex can be used for intra-process thread synchronization, using Monitor is generally preferred, because monitors were designed specifically for the .NET Framework and therefore make better use of resources. In contrast, the Mutex class is a wrapper to a Win32 construct. While it is more powerful than a monitor, a mutex requires interop transitions that are more computationally expensive than those required by the Monitor class. For an example of using a mutex, see Mutexes.

Interlocked Class

You can use the methods of the Interlocked class to prevent problems that can occur when multiple threads attempt to simultaneously update or compare the same value. The methods of this class let you safely increment, decrement, exchange, and compare values from any thread.

ReaderWriter Locks

In some cases, you may want to lock a resource only when data is being written and permit multiple clients to simultaneously read data when data is not being updated. The ReaderWriterLock class enforces exclusive access to a resource while a thread is modifying the resource, but it allows non-exclusive access when reading the resource. ReaderWriter locks are a useful alternative to exclusive locks, which cause other threads to wait, even when those threads do not need to update data.

Deadlocks

Thread synchronization is invaluable in multithreaded applications, but there is always the danger of creating a deadlock, where multiple threads are waiting for each other and the application comes to a halt. A deadlock is analogous to a situation in which cars are stopped at a four-way stop and each person is waiting for the other to go. Avoiding deadlocks
is important; the key is careful planning. You can often predict deadlock situations by diagramming multithreaded applications before you start coding.

See Also

Thread
WaitOne
WaitAny
WaitAll
Join
Start
Sleep
Monitor
Mutex
AutoResetEvent
ManualResetEvent
Interlocked
WaitHandle
EventWaitHandle
System.Threading
Set
Multithreaded Applications (Visual Basic)
SyncLock Statement
Mutexes
Monitors
Interlocked Operations
AutoResetEvent
Synchronizing Data for Multithreading

© 2016 Microsoft
Thread Timers (Visual Basic)

Visual Studio 2015

The `System.Threading.Timer` class is useful for periodically running a task on a separate thread. For example, you could use a thread timer to check the status and integrity of a database or to back up critical files.

Thread Timer Example

The following example starts a task every two seconds and uses a flag to initiate the `Dispose` method that stops the timer. This example posts status to the output window.

```vb
Private Class StateObjClass
    ' Used to hold parameters for calls to TimerTask.
    Public SomeValue As Integer
    Public TimerReference As System.Threading.Timer
    Public TimerCanceled As Boolean
End Class

Public Sub RunTimer()
    Dim StateObj As New StateObjClass
    StateObj.TimerCanceled = False
    StateObj.SomeValue = 1
    Dim TimerDelegate As New System.Threading.TimerCallback(AddressOf TimerTask)
    ' Create a timer that calls a procedure every 2 seconds.
    ' Note: There is no Start method; the timer starts running as soon as
    ' the instance is created.
    Dim TimerItem As New System.Threading.Timer(TimerDelegate, StateObj, 2000, 2000)
    ' Save a reference for Dispose.
    StateObj.TimerReference = TimerItem

    ' Run for ten loops.
    While StateObj.SomeValue < 10
        ' Wait one second.
        System.Threading.Thread.Sleep(1000)
    End While

    ' Request Dispose of the timer object.
    StateObj.TimerCanceled = True
End Sub

Private Sub TimerTask(ByVal StateObj As Object)
    Dim State As StateObjClass = CType(StateObj, StateObjClass)
    ' Use the interlocked class to increment the counter variable.
    System.Threading.Interlocked.Increment(State.SomeValue)
End Sub
```
Thread timers are particularly useful when the `System.Windows.Forms.Timer` object is unavailable, such as when you are developing console applications.

```vbnet
If State.TimerCanceled Then
    ' Dispose Requested.
    State.TimerReference.Dispose()
    System.Diagnostics.Debug.WriteLine("Done " & Now)
End If
End Sub
```

See Also

- `System.Threading`
- `Multithreaded Applications (Visual Basic)`

© 2016 Microsoft
Thread Pooling (Visual Basic)

**Visual Studio 2015**

A *thread pool* is a collection of threads that can be used to perform several tasks in the background. (See Threading (Visual Basic) for background information.) This leaves the primary thread free to perform other tasks asynchronously.

Thread pools are often employed in server applications. Each incoming request is assigned to a thread from the thread pool, so that the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.

Once a thread in the pool completes its task, it is returned to a queue of waiting threads, where it can be reused. This reuse enables applications to avoid the cost of creating a new thread for each task.

Thread pools typically have a maximum number of threads. If all the threads are busy, additional tasks are put in queue until they can be serviced as threads become available.

You can implement your own thread pool, but it is easier to use the thread pool provided by the .NET Framework through the **ThreadPool** class.

With thread pooling, you call the **ThreadPool.QueueUserWorkItem** method with a delegate for the procedure you want to run, and Visual Basic creates the thread and runs your procedure.

**Thread Pooling Example**

The following example shows how you can use thread pooling to start several tasks.

```vb
Public Sub DoWork()
    ' Queue a task.
    System.Threading.ThreadPool.QueueUserWorkItem(
        New System.Threading.WaitCallback(AddressOf SomeLongTask))
    ' Queue another task.
    System.Threading.ThreadPool.QueueUserWorkItem(
        New System.Threading.WaitCallback(AddressOf AnotherLongTask))
End Sub
Private Sub SomeLongTask(ByVal state As Object)
    ' Insert code to perform a long task.
End Sub
Private Sub AnotherLongTask(ByVal state As Object)
    ' Insert code to perform another long task.
End Sub
```

One advantage of thread pooling is that you can pass arguments in a state object to the task procedure. If the procedure you are calling requires more than one argument, you can cast a structure or an instance of a class into an **Object** data type.
Thread Pool Parameters and Return Values

Returning values from a thread-pool thread is not straightforward. The standard way of returning values from a function call is not allowed because Sub procedures are the only type of procedure that can be queued to a thread pool. One way you can provide parameters and return values is by wrapping the parameters, return values, and methods in a wrapper class as described in Parameters and Return Values for Multithreaded Procedures (Visual Basic).

An easier way to provide parameters and return values is by using the optional ByVal state object variable of the QueueUserWorkItem method. If you use this variable to pass a reference to an instance of a class, the members of the instance can be modified by the thread-pool thread and used as return values.

At first it may not be obvious that you can modify an object referred to by a variable that is passed by value. You can do this because only the object reference is passed by value. When you make changes to members of the object referred to by the object reference, the changes apply to the actual class instance.

Structures cannot be used to return values inside state objects. Because structures are value types, changes that the asynchronous process makes do not change the members of the original structure. Use structures to provide parameters when return values are not needed.

See Also

QueueUserWorkItem
System.Threading
ThreadPool
How to: Use a Thread Pool (Visual Basic)
Threading (Visual Basic)
Multithreaded Applications (Visual Basic)
Thread Synchronization (Visual Basic)
Thread pooling is a form of multithreading in which tasks are added to a queue and automatically started when threads are created. For more information, see Thread Pooling (Visual Basic).

The following example uses the .NET Framework thread pool to calculate the Fibonacci result for ten numbers between 20 and 40. Each Fibonacci result is represented by the Fibonacci class, which provides a method named ThreadPoolCallback that performs the calculation. An object that represents each Fibonacci value is created, and the ThreadPoolCallback method is passed to QueueUserWorkItem, which assigns an available thread in the pool to execute the method.

Because each Fibonacci object is given a semi-random value to compute, and because each thread will be competing for processor time, you cannot know in advance how long it will take for all ten results to be calculated. That is why each Fibonacci object is passed an instance of the ManualResetEvent class during construction. Each object signals the provided event object when its calculation is complete, which allows the primary thread to block execution with WaitAll until all ten Fibonacci objects have calculated a result. The Main method then displays each Fibonacci result.

Example

```vb
Imports System.Threading

Module Module1

Public Class Fibonacci
    Private _n As Integer
    Private _fibOfN
    Private _doneEvent As ManualResetEvent

    Public ReadOnly Property N() As Integer
        Get
            Return _n
        End Get
    End Property

    Public ReadOnly Property FibOfN() As Integer
        Get
            Return _fibOfN
        End Get
    End Property

    Sub New(ByVal n As Integer, ByVal doneEvent As ManualResetEvent)
        _n = n
        _doneEvent = doneEvent
    End Sub

```

'Wrapper method for use with the thread pool.

Public Sub ThreadPoolCallBack(ByVal threadContext As Object)
    Dim threadIndex As Integer = CType(threadContext, Integer)
    _fibOfN = Calculate(_n)
    Console.WriteLine("thread {0} result calculated...", threadIndex)
    _doneEvent.Set()
End Sub

Public Function Calculate(ByVal n As Integer) As Integer
    If n <= 1 Then
        Return n
    End If
    Return Calculate(n - 1) + Calculate(n - 2)
End Function

End Class

<MTAThread()>
Sub Main()
    Const FibonacciCalculations As Integer = 9 ' 0 to 9

    ' One event is used for each Fibonacci object
    Dim doneEvents(FibonacciCalculations) As ManualResetEvent
    Dim fibArray(FibonacciCalculations) As Fibonacci
    Dim r As New Random()

    ' Configure and start threads using ThreadPool.
    Console.WriteLine("launching {0} tasks...", FibonacciCalculations)
    For i As Integer = 0 To FibonacciCalculations
        doneEvents(i) = New ManualResetEvent(False)
        Dim f = New Fibonacci(r.Next(20, 40), doneEvents(i))
        fibArray(i) = f
        ThreadPool.QueueUserWorkItem(AddressOf f.ThreadPoolCallBack, i)
    Next

    ' Wait for all threads in pool to calculate.
    WaitHandle.WaitAll(doneEvents)
    Console.WriteLine("All calculations are complete.")

    ' Display the results.
    For i As Integer = 0 To FibonacciCalculations
        Dim f As Fibonacci = fibArray(i)
        Console.WriteLine("Fibonacci({0}) = {1}", i, f.FibOfN)
    Next
End Sub

End Module

Following is an example of the output.
launching 10 tasks...
thread 0 started...
thread 1 started...
thread 1 result calculated...
thread 2 started...
thread 2 result calculated...
thread 3 started...
thread 3 result calculated...
thread 4 started...
thread 0 result calculated...
thread 5 started...
thread 5 result calculated...
thread 6 started...
thread 4 result calculated...
thread 7 started...
thread 6 result calculated...
thread 8 started...
thread 8 result calculated...
thread 9 started...
thread 9 result calculated...
thread 7 result calculated...
All calculations are complete.
Fibonacci(38) = 39088169
Fibonacci(29) = 514229
Fibonacci(25) = 75025
Fibonacci(22) = 17711
Fibonacci(38) = 39088169
Fibonacci(29) = 514229
Fibonacci(29) = 514229
Fibonacci(38) = 39088169
Fibonacci(21) = 10946
Fibonacci(27) = 196418

See Also

Mutex
WaitAll
ManualResetEvent
Set
ThreadPool
QueueUserWorkItem
ManualResetEvent
Thread Pooling (Visual Basic)
Threading (Visual Basic)
Monitors
Security in the .NET Framework

© 2016 Microsoft