Managed Threading

.NET Framework (current version)

Whether you are developing for computers with one processor or several, you want your application to provide the most responsive interaction with the user, even if the application is currently doing other work. Using multiple threads of execution is one of the most powerful ways to keep your application responsive to the user and at the same time make use of the processor in between or even during user events. While this section introduces the basic concepts of threading, it focuses on managed threading concepts and using managed threading.

Note

Starting 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.

In This Section

Managed Threading Basics
: Provides an overview of managed threading and discusses when to use multiple threads.

Using Threads and Threading
: Explains how to create, start, pause, resume, and abort threads.

Managed Threading Best Practices
: Discusses levels of synchronization, how to avoid deadlocks and race conditions, single-processor and multiprocessor computers, and other threading issues.

Threading Objects and Features
: Describes the managed classes you can use to synchronize the activities of threads and the data of objects accessed on different threads, and provides an overview of thread pool threads.

Reference

System.Threading
: Contains classes for using and synchronizing managed threads.

System.Collections.Concurrent
: Contains collection classes that are safe for use with multiple threads.

System.Threading.Tasks
: Contains classes for creating and scheduling concurrent processing tasks.

Related Sections

Application Domains
: Provides an overview of application domains and their use by the Common Language Infrastructure.

Asynchronous File I/O
: Describes the performance advantages and basic operation of asynchronous I/O.

Event-based Asynchronous Pattern (EAP)
: Provides an overview of asynchronous programming.

Calling Synchronous Methods Asynchronously
: Explains how to call methods on thread pool threads using built-in features of delegates.

Parallel Programming in the .NET Framework
: Describes the parallel programming libraries, which simplify the use of multiple threads in applications.

Parallel LINQ (PLINQ)
: Describes a system for running queries in parallel, to take advantage of multiple processors.

Managed Threading Basics

.NET Framework (current version)

The first five topics of this section are designed to help you determine when to use managed threading, and to explain some basic features. For information on classes that provide additional features, see Threading Objects and Features and Overview of Synchronization Primitives.

The rest of the topics in this section cover advanced topics, including the interaction of managed threading with the Windows operating system.

Note

In the .NET Framework 4, the Task Parallel Library and PLINQ provide APIs for task and data parallelism in multi-threaded programs. For more information, see Parallel Programming in the .NET Framework.

In This Section

Threads and Threading
: Discusses the advantages and drawbacks of multiple threads, and outlines the scenarios in which you might create threads or use thread pool threads.

Exceptions in Managed Threads
: Describes the behavior of unhandled exceptions in threads for different versions of the .NET Framework, in particular the situations in which they result in termination of the application.

Synchronizing Data for Multithreading
: Describes strategies for synchronizing data in classes that will be used with multiple threads.

Managed Thread States
: Describes the basic thread states, and explains how to detect whether a thread is running.

Foreground and Background Threads
: Explains the differences between foreground and background threads.

Managed and Unmanaged Threading in Windows
: Discusses the relationship between managed and unmanaged threading, lists managed equivalents for Windows threading APIs, and discusses the interaction of COM apartments and managed threads.

Thread.Suspend, Garbage Collection, and Safe Points
: Describes thread suspension and garbage collection.

Thread Local Storage: Thread-Relative Static Fields and Data Slots
: Describes thread-relative storage mechanisms.

Cancellation in Managed Threads

Describes how asynchronous or long-running synchronous operations can be canceled by using a cancellation token.

Reference

Thread

Provides reference documentation for the **Thread** class, which represents a managed thread, whether it came from unmanaged code or was created in a managed application.

BackgroundWorker

Provides a safe way to implement multithreading in conjunction with user-interface objects.

Related Sections

Overview of Synchronization Primitives

Describes the managed classes used to synchronize the activities of multiple threads.

Managed Threading Best Practices

Describes common problems with multithreading and strategies for avoiding problems.

Parallel Programming in the .NET Framework

Describes the Task Parallel Library and PLINQ, which greatly simplify the work of creating asynchronous and multi-threaded .NET Framework applications.

Threads and Threading

.NET Framework (current version)

Operating systems use processes to separate the different applications that they are executing. Threads are the basic unit to which an operating system allocates processor time, and more than one thread can be executing code inside that process. Each thread maintains exception handlers, a scheduling priority, and a set of structures the system uses to save the thread context until it is scheduled. The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack, in the address space of the thread's host process.

The .NET Framework further subdivides an operating system process into lightweight managed subprocesses, called application domains, represented by System.AppDomain. One or more managed threads (represented by System.Threading.Thread) can run in one or any number of application domains within the same managed process. Although each application domain is started with a single thread, code in that application domain can create additional application domains and additional threads. The result is that a managed thread can move freely between application domains inside the same managed process; you might have only one thread moving among several application domains.

An operating system that supports preemptive multitasking creates the effect of simultaneous execution of multiple threads from multiple processes. It does this by dividing the available processor time among the threads that need it, allocating a processor time slice to each thread one after another. The currently executing thread is suspended when its time slice elapses, and another thread resumes running. When the system switches from one thread to another, it saves the thread context of the preempted thread and reloads the saved thread context of the next thread in the thread queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small, multiple threads appear to be executing at the same time, even if there is only one processor. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors.

When To Use Multiple Threads

Software that requires user interaction must react to the user's activities as rapidly as possible to provide a rich user experience. At the same time, however, it must do the calculations necessary to present data to the user as fast as possible. If your application uses only one thread of execution, you can combine asynchronous programming with .NET Framework remoting or XML Web services created using ASP.NET to use the processing time of other computers in addition to that of your own to increase responsiveness to the user and decrease the data processing time of your application. If you are doing intensive input/output work, you can also use I/O completion ports to increase your application's responsiveness.

Advantages of Multiple Threads

Using more than one thread, however, is the most powerful technique available to increase responsiveness to the user and process the data necessary to get the job done at almost the same time. On a computer with one processor, multiple threads can create this effect, taking advantage of the small periods of time in between user events to process the data in the background. For example, a user can edit a spreadsheet while another thread is recalculating other parts of the spreadsheet within the same application.

Without modification, the same application would dramatically increase user satisfaction when run on a computer with more than one processor. Your single application domain could use multiple threads to accomplish the following tasks:

- Communicate over a network, to a Web server, and to a database.
- Perform operations that take a large amount of time.
- Distinguish tasks of varying priority. For example, a high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.
- Allow the user interface to remain responsive, while allocating time to background tasks.

Disadvantages of Multiple Threads

It is recommended that you use as few threads as possible, thereby minimizing the use of operating-system resources and improving performance. Threading also has resource requirements and potential conflicts to be considered when designing your application. The resource requirements are as follows:

- The system consumes memory for the context information required by processes, **AppDomain** objects, and threads. Therefore, the number of processes, **AppDomain** objects, and threads that can be created is limited by available memory.
- Keeping track of a large number of threads consumes significant processor time. If there are too many threads, most of them will not make significant progress. If most of the current threads are in one process, threads in other processes are scheduled less frequently.
- Controlling code execution with many threads is complex, and can be a source of many bugs.
- Destroying threads requires knowing what could happen and handling those issues.

Providing shared access to resources can create conflicts. To avoid conflicts, you must synchronize, or control the access to, shared resources. Failure to synchronize access properly (in the same or different application domains) can lead to problems such as deadlocks (in which two threads stop responding while each waits for the other to complete) and race conditions (when an anomalous result occurs due to an unexpected critical dependence on the timing of two events). The system provides synchronization objects that can be used to coordinate resource sharing among multiple threads. Reducing the number of threads makes it easier to synchronize resources.

Resources that require synchronization include:

- System resources (such as communications ports).
- Resources shared by multiple processes (such as file handles).
- The resources of a single application domain (such as global, static, and instance fields) accessed by multiple threads.

Threading and Application Design

In general, using the ThreadPool class is the easiest way to handle multiple threads for relatively short tasks that will not block other threads and when you do not expect any particular scheduling of the tasks. However, there are a number of

reasons to create your own threads:

- If you need a task to have a particular priority.
- If you have a task that might run a long time (and therefore block other tasks).
- If you need to place threads into a single-threaded apartment (all **ThreadPool** threads are in the multithreaded apartment).
- If you need a stable identity associated with the thread. For example, you should use a dedicated thread to abort that thread, suspend it, or discover it by name.
- If you need to run background threads that interact with the user interface, the .NET Framework version 2.0 provides a BackgroundWorker component that communicates using events, with cross-thread marshaling to the user-interface thread.

Threading and Exceptions

Do handle exceptions in threads. Unhandled exceptions in threads, even background threads, generally terminate the process. There are three exceptions to this rule:

- A ThreadAbortException is thrown in a thread because Abort was called.
- An AppDomainUnloadedException is thrown in a thread because the application domain is being unloaded.
- The common language runtime or a host process terminates the thread.

For more information, see Exceptions in Managed Threads.

Note

In the .NET Framework versions 1.0 and 1.1, the common language runtime silently traps some exceptions, for example in thread pool threads. This may corrupt application state and eventually cause applications to hang, which might be very difficult to debug.

See Also

ThreadPool
BackgroundWorker
Synchronizing Data for Multithreading
The Managed Thread Pool

Synchronizing Data for Multithreading

.NET Framework (current version)

When multiple threads can make calls to the properties and methods of a single object, it is critical that those calls be synchronized. Otherwise one thread might interrupt what another thread is doing, and the object could be left in an invalid state. A class whose members are protected from such interruptions is called thread-safe.

The Common Language Infrastructure provides several strategies to synchronize access to instance and static members:

- Synchronized code regions. You can use the Monitor class or compiler support for this class to synchronize only the code block that needs it, improving performance.
- Manual synchronization. You can use the synchronization objects provided by the .NET Framework class library. See Overview of Synchronization Primitives, which includes a discussion of the Monitor class.
- Synchronized contexts. You can use the SynchronizationAttribute to enable simple, automatic synchronization for ContextBoundObject objects.
- Collection classes in the System.Collections.Concurrent namespace. These classes provide built-in synchronized add and remove operations. For more information, see Thread-Safe Collections.

The common language runtime provides a thread model in which classes fall into a number of categories that can be synchronized in a variety of different ways depending on the requirements. The following table shows what synchronization support is provided for fields and methods with a given synchronization category.

Category	Global fields	Static fields	Static methods	Instance fields	Instance methods	Specific code blocks
No Synchronization	No	No	No	No	No	No
Synchronized Context	No	No	No	Yes	Yes	No
Synchronized Code Regions	No	No	Only if marked	No	Only if marked	Only if marked
Manual Synchronization	Manual	Manual	Manual	Manual	Manual	Manual

No Synchronization

This is the default for objects. Any thread can access any method or field at any time. Only one thread at a time should

access these objects.

Manual Synchronization

The .NET Framework class library provides a number of classes for synchronizing threads. See Overview of Synchronization Primitives.

Synchronized Code Regions

You can use the Monitor class or a compiler keyword to synchronize blocks of code, instance methods, and static methods. There is no support for synchronized static fields.

Both Visual Basic and C# support the marking of blocks of code with a particular language keyword, the **lock** statement in C# or the **SyncLock** statement in Visual Basic. When the code is executed by a thread, an attempt is made to acquire the lock. If the lock has already been acquired by another thread, the thread blocks until the lock becomes available. When the thread exits the synchronized block of code, the lock is released, no matter how the thread exits the block.

Note

The **lock** and **SyncLock** statements are implemented using Monitor.Enter and Monitor.Exit, so other methods of Monitor can be used in conjunction with them within the synchronized region.

You can also decorate a method with a **MethodImplAttribute** and **MethodImplOptions.Synchronized**, which has the same effect as using **Monitor** or one of the compiler keywords to lock the entire body of the method.

Thread.Interrupt can be used to break a thread out of blocking operations such as waiting for access to a synchronized region of code. **Thread.Interrupt** is also used to break threads out of operations like Thread.Sleep.

Important

Do not lock the type — that is, `typeof(MyType)` in C#, `GetType(MyType)` in Visual Basic, or `MyType::typeid` in C++ — in order to protect **static** methods (**Shared** methods in Visual Basic). Use a private static object instead. Similarly, do not use **this** in C# (**Me** in Visual Basic) to lock instance methods. Use a private object instead. A class or instance can be locked by code other than your own, potentially causing deadlocks or performance problems.

Compiler Support

Both Visual Basic and C# support a language keyword that uses Monitor.Enter and Monitor.Exit to lock the object. Visual Basic supports the SyncLock statement; C# supports the lock statement.

In both cases, if an exception is thrown in the code block, the lock acquired by the **lock** or **SyncLock** is released automatically. The C# and Visual Basic compilers emit a **try**/**finally** block with **Monitor.Enter** at the beginning of the try, and **Monitor.Exit** in the **finally** block. If an exception is thrown inside the **lock** or **SyncLock** block, the **finally**

handler runs to allow you to do any clean-up work.

Synchronized Context

You can use the **SynchronizationAttribute** on any **ContextBoundObject** to synchronize all instance methods and fields. All objects in the same context domain share the same lock. Multiple threads are allowed to access the methods and fields, but only a single thread is allowed at any one time.

See Also

SynchronizationAttribute
Threads and Threading
Overview of Synchronization Primitives
SyncLock Statement
lock Statement (C# Reference)

Foreground and Background Threads

.NET Framework (current version)

A managed thread is either a background thread or a foreground thread. Background threads are identical to foreground threads with one exception: a background thread does not keep the managed execution environment running. Once all foreground threads have been stopped in a managed process (where the .exe file is a managed assembly), the system stops all background threads and shuts down.

Note

When the runtime stops a background thread because the process is shutting down, no exception is thrown in the thread. However, when threads are stopped because the AppDomain.Unload method unloads the application domain, a ThreadAbortException is thrown in both foreground and background threads.

Use the Thread.IsBackground property to determine whether a thread is a background or a foreground thread, or to change its status. A thread can be changed to a background thread at any time by setting its IsBackground property to **true**.

Important

The foreground or background status of a thread does not affect the outcome of an unhandled exception in the thread. In the .NET Framework version 2.0, an unhandled exception in either foreground or background threads results in termination of the application. See Exceptions in Managed Threads.

Threads that belong to the managed thread pool (that is, threads whose IsThreadPoolThread property is **true**) are background threads. All threads that enter the managed execution environment from unmanaged code are marked as background threads. All threads generated by creating and starting a new Thread object are by default foreground threads.

If you use a thread to monitor an activity, such as a socket connection, set its IsBackground property to **true** so that the thread does not prevent your process from terminating.

See Also

Thread.IsBackground
Thread
ThreadAbortException

Managed and Unmanaged Threading in Windows

.NET Framework (current version)

Management of all threads is done through the Thread class, including threads created by the common language runtime and those created outside the runtime that enter the managed environment to execute code. The runtime monitors all the threads in its process that have ever executed code within the managed execution environment. It does not track any other threads. Threads can enter the managed execution environment through COM interop (because the runtime exposes managed objects as COM objects to the unmanaged world), the COM DllGetClassObject function, and platform invoke.

When an unmanaged thread enters the runtime through, for example, a COM callable wrapper, the system checks the thread-local store of that thread to look for an internal managed Thread object. If one is found, the runtime is already aware of this thread. If it cannot find one, however, the runtime builds a new Thread object and installs it in the thread-local store of that thread.

In managed threading, Thread.GetHashCode is the stable managed thread identification. For the lifetime of your thread, it will not collide with the value from any other thread, regardless of the application domain from which you obtain this value.

Note

An operating-system **ThreadId** has no fixed relationship to a managed thread, because an unmanaged host can control the relationship between managed and unmanaged threads. Specifically, a sophisticated host can use the Fiber API to schedule many managed threads against the same operating system thread, or to move a managed thread among different operating system threads.

Mapping from Win32 Threading to Managed Threading

The following table maps Win32 threading elements to their approximate runtime equivalent. Note that this mapping does not represent identical functionality. For example, **TerminateThread** does not execute **finally** clauses or free up resources, and cannot be prevented. However, Thread.Abort executes all your rollback code, reclaims all the resources, and can be denied using ResetAbort. Be sure to read the documentation closely before making assumptions about functionality.

In Win32	In the common language runtime
CreateThread	Combination of **Thread** and ThreadStart
TerminateThread	Thread.Abort
SuspendThread	Thread.Suspend

ResumeThread	Thread.Resume
Sleep	Thread.Sleep
WaitForSingleObject on the thread handle	Thread.Join
ExitThread	No equivalent
GetCurrentThread	Thread.CurrentThread
SetThreadPriority	Thread.Priority
No equivalent	Thread.Name
No equivalent	Thread.IsBackground
Close to **CoInitializeEx** (OLE32.DLL)	Thread.ApartmentState

Managed Threads and COM Apartments

A managed thread can be marked to indicate that it will host a single-threaded or multithreaded apartment. (For more information on the COM threading architecture, see Processes, threads, and Apartments.) The GetApartmentState, SetApartmentState, and TrySetApartmentState methods of the Thread class return and assign the apartment state of a thread. If the state has not been set, GetApartmentState returns ApartmentState.Unknown.

The property can be set only when the thread is in the ThreadState.Unstarted state; it can be set only once for a thread.

If the apartment state is not set before the thread is started, the thread is initialized as a multithreaded apartment (MTA). The finalizer thread and all threads controlled by ThreadPool are MTA.

Important

For application startup code, the only way to control apartment state is to apply the MTAThreadAttribute or the STAThreadAttribute to the entry point procedure. In the .NET Framework 1.0 and 1.1, the ApartmentState property can be set as the first line of code. This is not permitted in the .NET Framework 2.0.

Managed objects that are exposed to COM behave as if they had aggregated the free-threaded marshaler. In other words, they can be called from any COM apartment in a free-threaded manner. The only managed objects that do not exhibit this free-threaded behavior are those objects that derive from ServicedComponent or StandardOleMarshalObject.

In the managed world, there is no support for the SynchronizationAttribute unless you use contexts and context-bound managed instances. If you are using Enterprise Services, then your object must derive from ServicedComponent (which is itself derived from ContextBoundObject).

When managed code calls out to COM objects, it always follows COM rules. In other words, it calls through COM apartment proxies and COM+ 1.0 context wrappers as dictated by OLE32.

Blocking Issues

If a thread makes an unmanaged call into the operating system that has blocked the thread in unmanaged code, the runtime will not take control of it for Thread.Interrupt or Thread.Abort. In the case of Thread.Abort, the runtime marks the thread for **Abort** and takes control of it when it re-enters managed code. It is preferable for you to use managed blocking rather than unmanaged blocking. WaitHandle.WaitOne, WaitHandle.WaitAny, WaitHandle.WaitAll, Monitor.Enter, Monitor.TryEnter, Thread.Join, GC.WaitForPendingFinalizers, and so on are all responsive to Thread.Interrupt and to Thread.Abort. Also, if your thread is in a single-threaded apartment, all these managed blocking operations will correctly pump messages in your apartment while your thread is blocked.

See Also

Thread.ApartmentState
ThreadState
ServicedComponent
Thread
Monitor

Cancellation in Managed Threads

.NET Framework (current version)

Starting with the .NET Framework 4, the .NET Framework uses a unified model for cooperative cancellation of asynchronous or long-running synchronous operations. This model is based on a lightweight object called a cancellation token. The object that invokes one or more cancelable operations, for example by creating new threads or tasks, passes the token to each operation. Individual operations can in turn pass copies of the token to other operations. At some later time, the object that created the token can use it to request that the operations stop what they are doing. Only the requesting object can issue the cancellation request, and each listener is responsible for noticing the request and responding to it in an appropriate and timely manner.

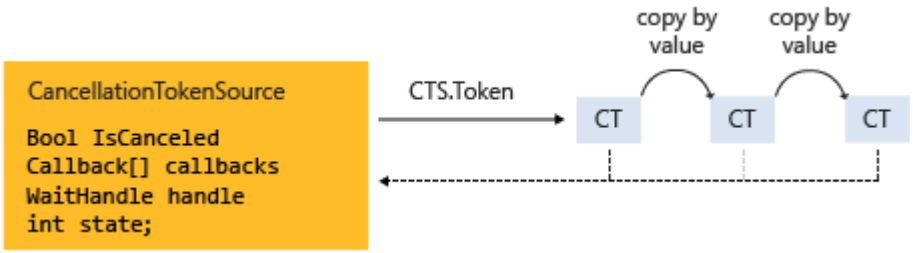
The general pattern for implementing the cooperative cancellation model is:

- Instantiate a CancellationTokenSource object, which manages and sends cancellation notification to the individual cancellation tokens.
- Pass the token returned by the CancellationTokenSource.Token property to each task or thread that listens for cancellation.
- Provide a mechanism for each task or thread to respond to cancellation.
- Call the CancellationTokenSource.Cancel method to provide notification of cancellation.

Important

The CancellationTokenSource class implements the IDisposable interface. You should be sure to call the CancellationTokenSource.Dispose method when you have finished using the cancellation token source to free any unmanaged resources it holds.

The following illustration shows the relationship between a token source and all the copies of its token.



The new cancellation model makes it easier to create cancellation-aware applications and libraries, and it supports the following features:

- Cancellation is cooperative and is not forced on the listener. The listener determines how to gracefully terminate in response to a cancellation request.
- Requesting is distinct from listening. An object that invokes a cancelable operation can control when (if ever)

cancellation is requested.

- The requesting object issues the cancellation request to all copies of the token by using just one method call.
- A listener can listen to multiple tokens simultaneously by joining them into one *linked token*.
- User code can notice and respond to cancellation requests from library code, and library code can notice and respond to cancellation requests from user code.
- Listeners can be notified of cancellation requests by polling, callback registration, or waiting on wait handles.

Cancellation Types

The cancellation framework is implemented as a set of related types, which are listed in the following table.

Type name	Description
CancellationTokenSource	Object that creates a cancellation token, and also issues the cancellation request for all copies of that token.
CancellationToken	Lightweight value type passed to one or more listeners, typically as a method parameter. Listeners monitor the value of the **IsCancellationRequested** property of the token by polling, callback, or wait handle.
OperationCanceledException	Overloads of this exception's constructor accept a CancellationToken as a parameter. Listeners can optionally throw this exception to verify the source of the cancellation and notify others that it has responded to a cancellation request.

The new cancellation model is integrated into the .NET Framework in several types. The most important ones are System.Threading.Tasks.Parallel, System.Threading.Tasks.Task, System.Threading.Tasks.Task(Of TResult) and System.Linq.ParallelEnumerable. We recommend that you use this new cancellation model for all new library and application code.

Code Example

In the following example, the requesting object creates a CancellationTokenSource object, and then passes its Token property to the cancelable operation. The operation that receives the request monitors the value of the IsCancellationRequested property of the token by polling. When the value becomes **true**, the listener can terminate in whatever manner is appropriate. In this example, the method just exits, which is all that is required in many cases.

Note

The example uses the QueueUserWorkItem method to demonstrate that the new cancellation framework is compatible with legacy APIs. For an example that uses the new, preferred System.Threading.Tasks.Task type, see How to: Cancel a Task and Its Children.

VB

```
Imports System.Threading

Module Example
   Public Sub Main()
      ' Create the token source.
      Dim cts As New CancellationTokenSource()

      ' Pass the token to the cancelable operation.
      ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf DoSomeWork), cts.Token)
      Thread.Sleep(2500)

      ' Request cancellation by setting a flag on the token.
      cts.Cancel()
      Console.WriteLine("Cancellation set in token source...")
      Thread.Sleep(2500)
      ' Cancellation should have happened, so call Dispose.
      cts.Dispose()
   End Sub

   ' Thread 2: The listener
   Sub DoSomeWork(ByVal obj As Object)
      Dim token As CancellationToken = CType(obj, CancellationToken)

      For i As Integer = 0 To 1000000
         If token.IsCancellationRequested Then
            Console.WriteLine("In iteration {0}, cancellation has been requested...",
                              i + 1)
            ' Perform cleanup if necessary.
            '...
            ' Terminate the operation.
            Exit For
         End If

         ' Simulate some work.
         Thread.SpinWait(500000)
      Next
   End Sub
End Module
' The example displays output like the following:
'       Cancellation set in token source...
'       In iteration 1430, cancellation has been requested...
```

Operation Cancellation Versus Object Cancellation

In the new cancellation framework, cancellation refers to operations, not objects. The cancellation request means that the operation should stop as soon as possible after any required cleanup is performed. One cancellation token should refer to one "cancelable operation," however that operation may be implemented in your program. After the IsCancellationRequested property of the token has been set to **true**, it cannot be reset to **false**. Therefore, cancellation

tokens cannot be reused after they have been canceled.

If you require an object cancellation mechanism, you can base it on the operation cancellation mechanism by calling the CancellationToken.Register method, as shown in the following example.

VB

```
Imports System.Threading

Class CancelableObject
   Public id As String

   Public Sub New(id As String)
      Me.id = id
   End Sub

   Public Sub Cancel()
      Console.WriteLine("Object {0} Cancel callback", id)
      ' Perform object cancellation here.
   End Sub
End Class

Module Example
   Public Sub Main()
        Dim cts As New CancellationTokenSource()
        Dim token As CancellationToken = cts.Token

        ' User defined Class with its own method for cancellation
        Dim obj1 As New CancelableObject("1")
        Dim obj2 As New CancelableObject("2")
        Dim obj3 As New CancelableObject("3")

        ' Register the object's cancel method with the token's
        ' cancellation request.
        token.Register(Sub() obj1.Cancel())
        token.Register(Sub() obj2.Cancel())
        token.Register(Sub() obj3.Cancel())

        ' Request cancellation on the token.
        cts.Cancel()
        ' Call Dispose when we're done with the CancellationTokenSource.
        cts.Dispose()
   End Sub
End Module
' The example displays output like the following:
'       Object 3 Cancel callback
'       Object 2 Cancel callback
'       Object 1 Cancel callback
```

If an object supports more than one concurrent cancelable operation, pass a separate token as input to each distinct cancelable operation. That way, one operation can be cancelled without affecting the others.

Listening and Responding to Cancellation Requests

In the user delegate, the implementer of a cancelable operation determines how to terminate the operation in response to a cancellation request. In many cases, the user delegate can just perform any required cleanup and then return immediately.

However, in more complex cases, it might be necessary for the user delegate to notify library code that cancellation has occurred. In such cases, the correct way to terminate the operation is for the delegate to call the ThrowIfCancellationRequested, method, which will cause an OperationCanceledException to be thrown. Library code can catch this exception on the user delegate thread and examine the exception's token to determine whether the exception indicates cooperative cancellation or some other exceptional situation.

The Task class handles OperationCanceledException in this way. For more information, see Task Cancellation.

Listening by Polling

For long-running computations that loop or recurse, you can listen for a cancellation request by periodically polling the value of the CancellationToken.IsCancellationRequested property. If its value is **true**, the method should clean up and terminate as quickly as possible. The optimal frequency of polling depends on the type of application. It is up to the developer to determine the best polling frequency for any given program. Polling itself does not significantly impact performance. The following example shows one possible way to poll.

VB

```
Shared Sub NestedLoops(ByVal rect As Rectangle, ByVal token As CancellationToken)
    For x As Integer = 0 To rect.columns
        For y As Integer = 0 To rect.rows
            ' Simulating work.
            Thread.SpinWait(5000)
            Console.Write("0' end block,1' end block ", x, y)
        Next

        ' Assume that we know that the inner loop is very fast.
        ' Therefore, checking once per row is sufficient.
        If token.IsCancellationRequested = True Then
            ' Cleanup or undo here if necessary...
            Console.WriteLine(vbCrLf + "Cancelling after row 0' end block.", x)
            Console.WriteLine("Press any key to exit.")
            ' then...
            Exit For
            ' ...or, if using Task:
            ' token.ThrowIfCancellationRequested()
        End If
    Next
End Sub
```

For a more complete example, see How to: Listen for Cancellation Requests by Polling.

Listening by Registering a Callback

Some operations can become blocked in such a way that they cannot check the value of the cancellation token in a timely manner. For these cases, you can register a callback method that unblocks the method when a cancellation request is received.

The Register method returns a CancellationTokenRegistration object that is used specifically for this purpose. The following example shows how to use the Register method to cancel an asynchronous Web request.

VB

```
Imports System
Imports System.Net
Imports System.Threading
Imports System.Threading.Tasks

Class Example
    Public Shared Event externalEvent(ByVal sender As Object, ByVal obj As Object)

    Public Sub Example1()
        Dim cts As New CancellationTokenSource()
        AddHandler externalEvent, Sub(sender, obj) cts.Cancel()

        Try
            Dim val As Integer = LongRunningFunc(cts.Token)
        Catch oce As OperationCanceledException
            'cleanup after cancellation if required...
            Console.WriteLine("Operation was canceled as expected.")
        Finally
           cts.Dispose()
        End Try
    End Sub

    Private Shared Function LongRunningFunc(ByVal token As CancellationToken) As
Integer
        Console.WriteLine("Long running method")
        Dim total As Integer = 0
        For i As Integer = 0 To 1000000
            For j As Integer = 0 To 1000000
                total = total + total
            Next
            If token.IsCancellationRequested Then
                ' observe cancellation
                Console.WriteLine("Cancellation observed.")
                Throw New OperationCanceledException(token) ' acknowledge cancellation
            End If
        Next
        Console.WriteLine("Done looping")
        Return total
    End Function

    Shared Sub Main()
        Dim ex As New Example()
        Dim t As Thread
        t = New Thread(AddressOf ex.Example1)
        t.Start()
```

```
        Console.WriteLine("Press 'c' to cancel.")
        If Console.ReadKey(True).KeyChar = "c"c Then
            RaiseEvent externalEvent(ex, New EventArgs())

            Console.WriteLine("Press enter to exit.")
            Console.ReadLine()
        End If
    End Sub
End Class

Class MyCancelableObject

    Public Sub Cancel()
    End Sub

    Shared Sub ObjectCancellationMiniSnippetForOverview()


    End Sub
End Class
Class CancelWaitHandleMiniSnippetsForOverviewTopic

    Shared Sub CancelByCallback()
        Dim cts As New CancellationTokenSource()
        Dim token As CancellationToken = cts.Token
        Dim wc As New WebClient()

        ' To request cancellation on the token
        ' will call CancelAsync on the WebClient.
        token.Register(Sub() wc.CancelAsync())

        Console.WriteLine("Starting request")
        wc.DownloadStringAsync(New Uri("http://www.contoso.com"))
    End Sub
End Class
```

The CancellationTokenRegistration object manages thread synchronization and ensures that the callback will stop executing at a precise point in time.

In order to ensure system responsiveness and to avoid deadlocks, the following guidelines must be followed when registering callbacks:

- The callback method should be fast because it is called synchronously and therefore the call to Cancel does not return until the callback returns.
- If you call Dispose while the callback is running, and you hold a lock that the callback is waiting on, your program can deadlock. After **Dispose** returns, you can free any resources required by the callback.
- Callbacks should not perform any manual thread or SynchronizationContext usage in a callback. If a callback must run on a particular thread, use the System.Threading.CancellationTokenRegistration constructor that enables you to specify that the target syncContext is the active SynchronizationContext.Current. Performing

manual threading in a callback can cause deadlock.

For a more complete example, see How to: Register Callbacks for Cancellation Requests.

Listening by Using a Wait Handle

When a cancelable operation can block while it waits on a synchronization primitive such as a System.Threading.ManualResetEvent or System.Threading.Semaphore, you can use the CancellationToken.WaitHandle property to enable the operation to wait on both the event and the cancellation request. The wait handle of the cancellation token will become signaled in response to a cancellation request, and the method can use the return value of the WaitAny method to determine whether it was the cancellation token that signaled. The operation can then just exit, or throw a OperationCanceledException, as appropriate.

VB

```
' Wait on the event if it is not signaled.
Dim waitHandles() As WaitHandle = { mre, token.WaitHandle }
Dim eventThatSignaledIndex =
    WaitHandle.WaitAny(waitHandles, _
                       New TimeSpan(0, 0, 20))
```

In new code that targets the .NET Framework 4, System.Threading.ManualResetEventSlim and System.Threading.SemaphoreSlim both support the new cancellation framework in their **Wait** methods. You can pass the CancellationToken to the method, and when the cancellation is requested, the event wakes up and throws an OperationCanceledException.

VB

```
Try
    ' mres is a ManualResetEventSlim
    mres.Wait(token)
Catch e As OperationCanceledException
    ' Throw immediately to be responsive. The
    ' alternative is to do one more item of work,
    ' and throw on next iteration, because
    ' IsCancellationRequested will be true.
    Console.WriteLine("Canceled while waiting.")
    Throw
End Try

 ' Simulating work.
Console.Write("Working...")
Thread.SpinWait(500000)
```

For a more complete example, see How to: Listen for Cancellation Requests That Have Wait Handles.

Listening to Multiple Tokens Simultaneously

In some cases, a listener may have to listen to multiple cancellation tokens simultaneously. For example, a cancelable

operation may have to monitor an internal cancellation token in addition to a token passed in externally as an argument to a method parameter. To accomplish this, create a linked token source that can join two or more tokens into one token, as shown in the following example.

VB

```
Public Sub DoWork(ByVal externalToken As CancellationToken)
   ' Create a new token that combines the internal and external tokens.
   Dim internalToken As CancellationToken = internalTokenSource.Token
   Dim linkedCts As CancellationTokenSource =
   CancellationTokenSource.CreateLinkedTokenSource(internalToken, externalToken)
   Using (linkedCts)
      Try
         DoWorkInternal(linkedCts.Token)
      Catch e As OperationCanceledException
         If e.CancellationToken = internalToken Then
            Console.WriteLine("Operation timed out.")
         ElseIf e.CancellationToken = externalToken Then
            Console.WriteLine("Canceled by external token.")
            externalToken.ThrowIfCancellationRequested()
         End If
      End Try
   End Using
End Sub
```

Notice that you must call **Dispose** on the linked token source when you are done with it. For a more complete example, see How to: Listen for Multiple Cancellation Requests.

Cooperation Between Library Code and User Code

The unified cancellation framework makes it possible for library code to cancel user code, and for user code to cancel library code in a cooperative manner. Smooth cooperation depends on each side following these guidelines:

- If library code provides cancelable operations, it should also provide public methods that accept an external cancellation token so that user code can request cancellation.
- If library code calls into user code, the library code should interpret an OperationCanceledException(externalToken) as *cooperative cancellation*, and not necessarily as a failure exception.
- User-delegates should attempt to respond to cancellation requests from library code in a timely manner.

System.Threading.Tasks.Task and System.Linq.ParallelEnumerable are examples of classes that follows these guidelines. For more information, see Task Cancellationand How to: Cancel a PLINQ Query.

See Also

Managed Threading Basics

Using Threads and Threading

.NET Framework (current version)

The topics in this section discuss the creation and management of managed threads, how to pass data to managed threads and get results back, and how to destroy threads and handle a ThreadAbortException.

In This Section

Creating Threads and Passing Data at Start Time
: Discusses and demonstrates the creation of managed threads, including how to pass data to new threads and how to get data back.

Pausing and Resuming Threads
: Discusses the ramifications of pausing and resuming managed threads.

Destroying Threads
: Discusses the ramifications of destroying managed threads, and how to handle a ThreadAbortException.

Scheduling Threads
: Discusses thread priorities and how they affect thread scheduling.

Reference

Thread
: Provides reference documentation for the Thread class, which represents a managed thread, whether it came from unmanaged code or was created in a managed application.

ThreadStart
: Provides reference documentation for the ThreadStart delegate that represents parameterless thread procedures.

ParameterizedThreadStart
: Provides an easy way to pass data to a thread procedure, although without strong typing.

Related Sections

Threads and Threading
: Provides an introduction to managed threading.

Creating Threads and Passing Data at Start Time

.NET Framework (current version)

When an operating-system process is created, the operating system injects a thread to execute code in that process, including any original application domain. From that point on, application domains can be created and destroyed without any operating system threads necessarily being created or destroyed. If the code being executed is managed code, then a Thread object for the thread executing in the current application domain can be obtained by retrieving the static CurrentThread property of type Thread. This topic describes thread creation and discusses alternatives for passing data to the thread procedure.

Creating a Thread

Creating a new Thread object creates a new managed thread. The Thread class has constructors that take a ThreadStart delegate or a ParameterizedThreadStart delegate; the delegate wraps the method that is invoked by the new thread when you call the Start method. Calling Start more than once causes a ThreadStateException to be thrown.

The Start method returns immediately, often before the new thread has actually started. You can use the ThreadState and IsAlive properties to determine the state of the thread at any one moment, but these properties should never be used for synchronizing the activities of threads.

Note

Once a thread is started, it is not necessary to retain a reference to the Thread object. The thread continues to execute until the thread procedure ends.

The following code example creates two new threads to call instance and static methods on another object.

VB

```
Imports System.Threading

Public class ServerClass
    ' The method that will be called when the thread is started.
    Public Sub InstanceMethod()
        Console.WriteLine(
            "ServerClass.InstanceMethod is running on another thread.")

        ' Pause for a moment to provide a delay to make
        ' threads more apparent.
        Thread.Sleep(3000)
        Console.WriteLine(
            "The instance method called by the worker thread has ended.")
    End Sub
```

```
    Public Shared Sub SharedMethod()
        Console.WriteLine(
            "ServerClass.SharedMethod is running on another thread.")

        ' Pause for a moment to provide a delay to make
        ' threads more apparent.
        Thread.Sleep(5000)
        Console.WriteLine(
            "The Shared method called by the worker thread has ended.")
    End Sub
End Class

Public class Simple
    Public Shared Sub Main()
        Dim serverObject As New ServerClass()

        ' Create the thread object, passing in the
        ' serverObject.InstanceMethod method using a
        ' ThreadStart delegate.
        Dim InstanceCaller As New Thread(AddressOf serverObject.InstanceMethod)

        ' Start the thread.
        InstanceCaller.Start()

        Console.WriteLine("The Main() thread calls this after " _
            + "starting the new InstanceCaller thread.")

        ' Create the thread object, passing in the
        ' serverObject.SharedMethod method using a
        ' ThreadStart delegate.
        Dim SharedCaller As New Thread( _
            New ThreadStart(AddressOf ServerClass.SharedMethod))

        ' Start the thread.
        SharedCaller.Start()

        Console.WriteLine("The Main() thread calls this after " _
            + "starting the new SharedCaller thread.")
    End Sub
End Class
' The example displays output like the following:
'    The Main() thread calls this after starting the new InstanceCaller thread.
'    The Main() thread calls this after starting the new StaticCaller thread.
'    ServerClass.StaticMethod is running on another thread.
'    ServerClass.InstanceMethod is running on another thread.
'    The instance method called by the worker thread has ended.
'    The static method called by the worker thread has ended.
```

Passing Data to Threads and Retrieving Data from Threads

In the .NET Framework version 2.0, the ParameterizedThreadStart delegate provides an easy way to pass an object containing data to a thread when you call the Thread.Start method overload. See ParameterizedThreadStart for a code example.

Using the ParameterizedThreadStart delegate is not a type-safe way to pass data, because the Thread.Start method overload accepts any object. An alternative is to encapsulate the thread procedure and the data in a helper class and use the ThreadStart delegate to execute the thread procedure. This technique is shown in the two code examples that follow.

Neither of these delegates has a return value, because there is no place to return the data from an asynchronous call. To retrieve the results of a thread method, you can use a callback method, as demonstrated in the second code example.

VB

```
Imports System.Threading

' The ThreadWithState class contains the information needed for
' a task, and the method that executes the task.
Public Class ThreadWithState
    ' State information used in the task.
    Private boilerplate As String
    Private value As Integer

    ' The constructor obtains the state information.
    Public Sub New(text As String, number As Integer)
        boilerplate = text
        value = number
    End Sub

    ' The thread procedure performs the task, such as formatting
    ' and printing a document.
    Public Sub ThreadProc()
        Console.WriteLine(boilerplate, value)
    End Sub
End Class

' Entry point for the example.
'
Public Class Example
    Public Shared Sub Main()
        ' Supply the state information required by the task.
        Dim tws As New ThreadWithState( _
            "This report displays the number {0}.", 42)

        ' Create a thread to execute the task, and then
        ' start the thread.
        Dim t As New Thread(New ThreadStart(AddressOf tws.ThreadProc))
        t.Start()
        Console.WriteLine("Main thread does some work, then waits.")
        t.Join()
        Console.WriteLine( _
            "Independent task has completed main thread ends.")
    End Sub
End Class
' The example displays the following output:
```

```
'       Main thread does some work, then waits.
'       This report displays the number 42.
'       Independent task has completed; main thread ends.
```

Retrieving Data with Callback Methods

The following example demonstrates a callback method that retrieves data from a thread. The constructor for the class that contains the data and the thread method also accepts a delegate representing the callback method; before the thread method ends, it invokes the callback delegate.

VB

```
Imports System.Threading

' The ThreadWithState class contains the information needed for
' a task, the method that executes the task, and a delegate
' to call when the task is complete.
Public Class ThreadWithState
    ' State information used in the task.
    Private boilerplate As String
    Private value As Integer

    ' Delegate used to execute the callback method when the
    ' task is complete.
    Private callback As ExampleCallback

    ' The constructor obtains the state information and the
    ' callback delegate.
    Public Sub New(text As String, number As Integer, _
        callbackDelegate As ExampleCallback)
        boilerplate = text
        value = number
        callback = callbackDelegate
    End Sub

    ' The thread procedure performs the task, such as
    ' formatting and printing a document, and then invokes
    ' the callback delegate with the number of lines printed.
    Public Sub ThreadProc()
        Console.WriteLine(boilerplate, value)
        If Not (callback Is Nothing) Then
            callback(1)
        End If
    End Sub
End Class

' Delegate that defines the signature for the callback method.
'
Public Delegate Sub ExampleCallback(lineCount As Integer)

Public Class Example
    Public Shared Sub Main()
        ' Supply the state information required by the task.
```

```
        Dim tws As New ThreadWithState( _
            "This report displays the number {0}.", _
            42, _
            AddressOf ResultCallback)

        Dim t As New Thread(AddressOf tws.ThreadProc)
        t.Start()
        Console.WriteLine("Main thread does some work, then waits.")
        t.Join()
        Console.WriteLine( _
            "Independent task has completed; main thread ends.")
    End Sub

    Public Shared Sub ResultCallback(lineCount As Integer)
        Console.WriteLine( _
            "Independent task printed {0} lines.", lineCount)
    End Sub
End Class
' The example displays the following output:
'       Main thread does some work, then waits.
'       This report displays the number 42.
'       Independent task printed 1 lines.
'       Independent task has completed; main thread ends.
```

See Also

- Thread
- ThreadStart
- ParameterizedThreadStart
- Thread.Start
- Managed Threading
- Using Threads and Threading

Pausing and Resuming Threads

.NET Framework (current version)

The most common ways to synchronize the activities of threads are to block and release threads, or to lock objects or regions of code. For more information on these locking and blocking mechanisms, see Overview of Synchronization Primitives.

You can also have threads put themselves to sleep. When threads are blocked or sleeping, you can use a ThreadInterruptedException to break them out of their wait states.

The Thread.Sleep Method

Calling the Thread.Sleep method causes the current thread to immediately block for the number of milliseconds or the time interval you pass to the method, and yields the remainder of its time slice to another thread. Once that interval elapses, the sleeping thread resumes execution.

One thread cannot call Thread.Sleep on another thread. Thread.Sleep is a static method that always causes the current thread to sleep.

Calling Thread.Sleep with a value of Timeout.Infinite causes a thread to sleep until it is interrupted by another thread that calls the Thread.Interrupt method on the sleeping thread, or until it is terminated by a call to its Thread.Abort method. The following example illustrates both methods of interrupting a sleeping thread.

VB

```
Imports System.Threading

Module Example
   Public Sub Main()
      ' Interrupt a sleeping thread.
      Dim sleepingThread = New Thread(AddressOf Example.SleepIndefinitely)
      sleepingThread.Name = "Sleeping"
      sleepingThread.Start()
      Thread.Sleep(2000)
      sleepingThread.Interrupt()

      Thread.Sleep(1000)

      sleepingThread = New Thread(AddressOf Example.SleepIndefinitely)
      sleepingThread.Name = "Sleeping2"
      sleepingThread.Start()
      Thread.Sleep(2000)
      sleepingThread.Abort()
   End Sub

   Private Sub SleepIndefinitely()
      Console.WriteLine("Thread '{0}' about to sleep indefinitely.",
                        Thread.CurrentThread.Name)
```

```
        Try
            Thread.Sleep(Timeout.Infinite)
        Catch ex As ThreadInterruptedException
            Console.WriteLine("Thread '{0}' awoken.",
                              Thread.CurrentThread.Name)
        Catch ex As ThreadAbortException
            Console.WriteLine("Thread '{0}' aborted.",
                              Thread.CurrentThread.Name)
        Finally
            Console.WriteLine("Thread '{0}' executing finally block.",
                              Thread.CurrentThread.Name)
        End Try
        Console.WriteLine("Thread '{0} finishing normal execution.",
                          Thread.CurrentThread.Name)
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'       Thread 'Sleeping' about to sleep indefinitely.
'       Thread 'Sleeping' awoken.
'       Thread 'Sleeping' executing finally block.
'       Thread 'Sleeping finishing normal execution.
'
'       Thread 'Sleeping2' about to sleep indefinitely.
'       Thread 'Sleeping2' aborted.
'       Thread 'Sleeping2' executing finally block.
```

Interrupting Threads

You can interrupt a waiting thread by calling the Thread.Interrupt method on the blocked thread to throw a ThreadInterruptedException, which breaks the thread out of the blocking call. The thread should catch the ThreadInterruptedException and do whatever is appropriate to continue working. If the thread ignores the exception, the runtime catches the exception and stops the thread.

Note

If the target thread is not blocked when Thread.Interrupt is called, the thread is not interrupted until it blocks. If the thread never blocks, it could complete without ever being interrupted.

If a wait is a managed wait, then Thread.Interrupt and Thread.Abort both wake the thread immediately. If a wait is an unmanaged wait (for example, a platform invoke call to the Win32 WaitForSingleObject function), neither Thread.Interrupt nor Thread.Abort can take control of the thread until it returns to or calls into managed code. In managed code, the behavior is as follows:

- Thread.Interrupt wakes a thread out of any wait it might be in and causes a ThreadInterruptedException to be thrown in the destination thread.
- Thread.Abort wakes a thread out of any wait it might be in and causes a ThreadAbortException to be thrown on

the thread. For details, see Destroying Threads.

See Also

- Thread
- ThreadInterruptedException
- ThreadAbortException
- Managed Threading
- Using Threads and Threading
- Overview of Synchronization Primitives

Destroying Threads

.NET Framework (current version)

The Abort method is used to stop a managed thread permanently. When you call Abort, the common language runtime throws a ThreadAbortException in the target thread, which the target thread can catch. For more information, see Thread.Abort.

Note

If a thread is executing unmanaged code when its Abort method is called, the runtime marks it ThreadState.AbortRequested. The exception is thrown when the thread returns to managed code.

Once a thread is aborted, it cannot be restarted.

The Abort method does not cause the thread to abort immediately, because the target thread can catch the ThreadAbortException and execute arbitrary amounts of code in a **finally** block. You can call Thread.Join if you need to wait until the thread has ended. Thread.Join is a blocking call that does not return until the thread has actually stopped executing or an optional timeout interval has elapsed. The aborted thread could call the ResetAbort method or perform unbounded processing in a **finally** block, so if you do not specify a timeout, the wait is not guaranteed to end.

Threads that are waiting on a call to the Thread.Join method can be interrupted by other threads that call Thread.Interrupt.

Handling ThreadAbortException

If you expect your thread to be aborted, either as a result of calling Abort from your own code or as a result of unloading an application domain in which the thread is running (AppDomain.Unload uses Thread.Abort to terminate threads), your thread must handle the ThreadAbortException and perform any final processing in a **finally** clause, as shown in the following code.

VB

```
Try
    ' Code that is executing when the thread is aborted.
Catch ex As ThreadAbortException
    ' Clean-up code can go here.
    ' If there is no Finally clause, ThreadAbortException is
    ' re-thrown by the system at the end of the Catch clause.
Finally
    ' Clean-up code can go here.
End Try
' Do not put clean-up code here, because the exception
' is rethrown at the end of the Finally clause.
```

Your clean-up code must be in the **catch** clause or the **finally** clause, because a ThreadAbortException is rethrown by the

system at the end of the **finally** clause, or at the end of the **catch** clause if there is no **finally** clause.

You can prevent the system from rethrowing the exception by calling the Thread.ResetAbort method. However, you should do this only if your own code caused the ThreadAbortException.

See Also

ThreadAbortException
Thread
Using Threads and Threading

Scheduling Threads

.NET Framework (current version)

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of **ThreadPriority.Normal**. Threads created outside the runtime retain the priority they had before they entered the managed environment. You can get or set the priority of any thread with the **Thread.Priority** property.

Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. The details of the scheduling algorithm used to determine the order in which threads are executed varies with each operating system. Under some operating systems, the thread with the highest priority (of those threads that can be executed) is always scheduled to run first. If multiple threads with the same priority are all available, the scheduler cycles through the threads at that priority, giving each thread a fixed time slice in which to execute. As long as a thread with a higher priority is available to run, lower priority threads do not get to execute. When there are no more runnable threads at a given priority, the scheduler moves to the next lower priority and schedules the threads at that priority for execution. If a higher priority thread becomes runnable, the lower priority thread is preempted and the higher priority thread is allowed to execute once again. On top of all that, the operating system can also adjust thread priorities dynamically as an application's user interface is moved between foreground and background. Other operating systems might choose to use a different scheduling algorithm.

See Also

Using Threads and Threading
Managed and Unmanaged Threading in Windows

Canceling Threads Cooperatively

.NET Framework (current version)

Prior to the .NET Framework 4, the .NET Framework provided no built-in way to cancel a thread cooperatively after it was started. However, in .NET Framework 4, you can use cancellation tokens to cancel threads, just as you can use them to cancel System.Threading.Tasks.Task objects or PLINQ queries. Although the System.Threading.Thread class does not offer built-in support for cancellation tokens, you can pass a token to a thread procedure by using the Thread constructor that takes a ParameterizedThreadStart delegate. The following example demonstrates how to do this.

VB

```
Imports System.Threading

Public Class ServerClass
   Public Shared Sub StaticMethod(obj As Object)
      Dim ct AS CancellationToken = CType(obj, CancellationToken)
      Console.WriteLine("ServerClass.StaticMethod is running on another thread.")

      ' Simulate work that can be canceled.
      While Not ct.IsCancellationRequested
         Thread.SpinWait(50000)
      End While
      Console.WriteLine("The worker thread has been canceled. Press any key to exit.")
      Console.ReadKey(True)
   End Sub
End Class

Public Class Simple
   Public Shared Sub Main()
      ' The Simple class controls access to the token source.
      Dim cts As New CancellationTokenSource()

      Console.WriteLine("Press 'C' to terminate the application..." + vbCrLf)
      ' Allow the UI thread to capture the token source, so that it
      ' can issue the cancel command.
      Dim t1 As New Thread( Sub()
                               If
Console.ReadKey(true).KeyChar.ToString().ToUpperInvariant() = "C" Then
                                  cts.Cancel()
                               End If
                            End Sub)

      ' ServerClass sees only the token, not the token source.
      Dim t2 As New Thread(New ParameterizedThreadStart(AddressOf
ServerClass.StaticMethod))

      ' Start the UI thread.
      t1.Start()
```

```
        ' Start the worker thread and pass it the token.
        t2.Start(cts.Token)

        t2.Join()
        cts.Dispose()
    End Sub
End Class
' The example displays the following output:
'       Press 'C' to terminate the application...
'
'       ServerClass.StaticMethod is running on another thread.
'       The worker thread has been canceled. Press any key to exit.
```

See Also

Using Threads and Threading

Managed Threading Best Practices

.NET Framework (current version)

Multithreading requires careful programming. For most tasks, you can reduce complexity by queuing requests for execution by thread pool threads. This topic addresses more difficult situations, such as coordinating the work of multiple threads, or handling threads that block.

Note

In the .NET Framework 4, the Task Parallel Library and PLINQ provide APIs that reduce some of the complexity and risks of multi-threaded programming. For more information, see Parallel Programming in the .NET Framework.

Deadlocks and Race Conditions

Multithreading solves problems with throughput and responsiveness, but in doing so it introduces new problems: deadlocks and race conditions.

Deadlocks

A deadlock occurs when each of two threads tries to lock a resource the other has already locked. Neither thread can make any further progress.

Many methods of the managed threading classes provide time-outs to help you detect deadlocks. For example, the following code attempts to acquire a lock on the current instance. If the lock is not obtained in 300 milliseconds, Monitor.TryEnter returns **false**.

VB

```
If Monitor.TryEnter(lockObject, 300) Then
    Try
        ' Place code protected by the Monitor here.
    Finally
        Monitor.Exit(Me)
    End Try
Else
    ' Code to execute if the attempt times out.
End If
```

Race Conditions

A race condition is a bug that occurs when the outcome of a program depends on which of two or more threads

reaches a particular block of code first. Running the program many times produces different results, and the result of any given run cannot be predicted.

A simple example of a race condition is incrementing a field. Suppose a class has a private **static** field (**Shared** in Visual Basic) that is incremented every time an instance of the class is created, using code such as `objCt++;` (C#) or `objCt += 1` (Visual Basic). This operation requires loading the value from `objCt` into a register, incrementing the value, and storing it in `objCt`.

In a multithreaded application, a thread that has loaded and incremented the value might be preempted by another thread which performs all three steps; when the first thread resumes execution and stores its value, it overwrites `objCt` without taking into account the fact that the value has changed in the interim.

This particular race condition is easily avoided by using methods of the Interlocked class, such as Interlocked.Increment. To read about other techniques for synchronizing data among multiple threads, see Synchronizing Data for Multithreading.

Race conditions can also occur when you synchronize the activities of multiple threads. Whenever you write a line of code, you must consider what might happen if a thread were preempted before executing the line (or before any of the individual machine instructions that make up the line), and another thread overtook it.

Number of Processors

Most computers now have multiple processors (also called cores), even small devices such as tablets and phones. If you know you're developing software that will also run on single-processor computers, you should be aware that multithreading solves different problems for single-processor computers and multiprocessor computers.

Multiprocessor Computers

Multithreading provides greater throughput. Ten processors can do ten times the work of one, but only if the work is divided so that all ten can be working at once; threads provide an easy way to divide the work and exploit the extra processing power. If you use multithreading on a multiprocessor computer:

- The number of threads that can execute concurrently is limited by the number of processors.
- A background thread executes only when the number of foreground threads executing is smaller than the number of processors.
- When you call the Thread.Start method on a thread, that thread might or might not start executing immediately, depending on the number of processors and the number of threads currently waiting to execute.
- Race conditions can occur not only because threads are preempted unexpectedly, but because two threads executing on different processors might be racing to reach the same code block.

Single-Processor Computers

Multithreading provides greater responsiveness to the computer user, and uses idle time for background tasks. If you use multithreading on a single-processor computer:

- Only one thread runs at any instant.
- A background thread executes only when the main user thread is idle. A foreground thread that executes constantly starves background threads of processor time.
- When you call the Thread.Start method on a thread, that thread does not start executing until the current thread yields or is preempted by the operating system.
- Race conditions typically occur because the programmer did not anticipate the fact that a thread can be preempted at an awkward moment, sometimes allowing another thread to reach a code block first.

Static Members and Static Constructors

A class is not initialized until its class constructor (**static** constructor in C#, **Shared Sub New** in Visual Basic) has finished running. To prevent the execution of code on a type that is not initialized, the common language runtime blocks all calls from other threads to **static** members of the class (**Shared** members in Visual Basic) until the class constructor has finished running.

For example, if a class constructor starts a new thread, and the thread procedure calls a **static** member of the class, the new thread blocks until the class constructor completes.

This applies to any type that can have a **static** constructor.

General Recommendations

Consider the following guidelines when using multiple threads:

- Don't use Thread.Abort to terminate other threads. Calling **Abort** on another thread is akin to throwing an exception on that thread, without knowing what point that thread has reached in its processing.
- Don't use Thread.Suspend and Thread.Resume to synchronize the activities of multiple threads. Do use Mutex, ManualResetEvent, AutoResetEvent, and Monitor.
- Don't control the execution of worker threads from your main program (using events, for example). Instead, design your program so that worker threads are responsible for waiting until work is available, executing it, and notifying other parts of your program when finished. If your worker threads do not block, consider using thread pool threads. Monitor.PulseAll is useful in situations where worker threads block.
- Don't use types as lock objects. That is, avoid code such as `lock(typeof(X))` in C# or `SyncLock(GetType(X))` in Visual Basic, or the use of Monitor.Enter with Type objects. For a given type, there is only one instance of System.Type per application domain. If the type you take a lock on is public, code other than your own can take locks on it, leading to deadlocks. For additional issues, see Reliability Best Practices.
- Use caution when locking on instances, for example `lock(this)` in C# or `SyncLock(Me)` in Visual Basic. If other code in your application, external to the type, takes a lock on the object, deadlocks could occur.
- Do ensure that a thread that has entered a monitor always leaves that monitor, even if an exception occurs while

the thread is in the monitor. The C# lock statement and the Visual Basic SyncLock statement provide this behavior automatically, employing a **finally** block to ensure that Monitor.Exit is called. If you cannot ensure that **Exit** will be called, consider changing your design to use **Mutex**. A mutex is automatically released when the thread that currently owns it terminates.

- Do use multiple threads for tasks that require different resources, and avoid assigning multiple threads to a single resource. For example, any task involving I/O benefits from having its own thread, because that thread will block during I/O operations and thus allow other threads to execute. User input is another resource that benefits from a dedicated thread. On a single-processor computer, a task that involves intensive computation coexists with user input and with tasks that involve I/O, but multiple computation-intensive tasks contend with each other.

- Consider using methods of the Interlocked class for simple state changes, instead of using the **lock** statement (**SyncLock** in Visual Basic). The **lock** statement is a good general-purpose tool, but the Interlocked class provides better performance for updates that must be atomic. Internally, it executes a single lock prefix if there is no contention. In code reviews, watch for code like that shown in the following examples. In the first example, a state variable is incremented:

VB

```
SyncLock lockObject
    myField += 1
End SyncLock
```

You can improve performance by using the Increment method instead of the **lock** statement, as follows:

VB

```
System.Threading.Interlocked.Increment(myField)
```

Note

In the .NET Framework version 2.0, the Add method provides atomic updates in increments larger than 1.

In the second example, a reference type variable is updated only if it is a null reference (**Nothing** in Visual Basic).

VB

```
If x Is Nothing Then
    SyncLock lockObject
        If x Is Nothing Then
            x = y
        End If
    End SyncLock
End If
```

Performance can be improved by using the CompareExchange method instead, as follows:

VB

```
System.Threading.Interlocked.CompareExchange(x, y, Nothing)
```

Note

In the .NET Framework version 2.0, the CompareExchange method has a generic overload that can be used for type-safe replacement of any reference type.

Recommendations for Class Libraries

Consider the following guidelines when designing class libraries for multithreading:

- Avoid the need for synchronization, if possible. This is especially true for heavily used code. For example, an algorithm might be adjusted to tolerate a race condition rather than eliminate it. Unnecessary synchronization decreases performance and creates the possibility of deadlocks and race conditions.
- Make static data (**Shared** in Visual Basic) thread safe by default.
- Do not make instance data thread safe by default. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlocks to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework class libraries are not thread safe by default.
- Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility of threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests. Furthermore, if static data are synchronized, calls between static methods that alter state can result in deadlocks or redundant synchronization, adversely affecting performance.

See Also

Managed Threading
Threads and Threading