Contents

- Threading
  - Managed Threading Basics
    - Threads and Threading
    - Exceptions in Managed Threads
    - Synchronizing Data for Multithreading
    - Foreground and Background Threads
    - Managed and Unmanaged Threading in Windows
    - Thread Local Storage: Thread-Relative Static Fields and Data Slots
  - Using threads and threading
    - Creating threads and passing data at start time
    - Pausing and interrupting threads
    - Destroying threads
    - Scheduling threads
    - Cancellation in managed threads
      - Canceling threads cooperatively
      - How to: Listen for Cancellation Requests by Polling
      - How to: Register Callbacks for Cancellation Requests
      - How to: Listen for Cancellation Requests That Have Wait Handles
      - How to: Listen for Multiple Cancellation Requests
  - Managed Threading Best Practices
  - Threading objects and features
    - The managed thread pool
    - Timers
    - Overview of synchronization primitives
    - EventWaitHandle
    - CountdownEvent
    - Mutexes
    - Semaphore and SemaphoreSlim
    - Barrier

- How to: Synchronize concurrent operations with a Barrier

SpinLock
- How to: Use SpinLock for low-level synchronization
- How to: Enable thread-tracking mode in SpinLock

SpinWait
- How to: Use SpinWait to implement a two-phase wait operation

Managed Threading

8/22/2019 • 2 minutes to read • Edit Online

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 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, 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.

Task-based Asynchronous Pattern (TAP)
Provides an overview of the recommended pattern for asynchronous programming in .NET.

Calling Synchronous Methods Asynchronously

Explains how to call methods on thread pool threads using built-in features of delegates.

Parallel Programming

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

8/22/2019 • 2 minutes to read • Edit Online

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 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.

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

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

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

Threads and threading

5/15/2019 • 2 minutes to read • Edit Online

Multithreading allows you to increase the responsiveness of your application and, if your application runs on a multiprocessor or multi-core system, increase its throughput.

Processes and threads

A *process* is an executing program. An operating system uses processes to separate the applications that are being executed. A *thread* is the basic unit to which an operating system allocates processor time. Each thread has a scheduling priority and maintains a set of structures the system uses to save the thread context when the thread's execution is paused. The thread context includes all the information the thread needs to seamlessly resume execution, including the thread's set of CPU registers and stack. Multiple threads can run in the context of a process. All threads of a process share its virtual address space. A thread can execute any part of the program code, including parts currently being executed by another thread.

NOTE

The .NET Framework provides a way to isolate applications within a process with the use of *application domains*. (Application domains are not available on .NET Core.) For more information, see the Application domains and threads section of the Application domains article.

By default, a .NET program is started with a single thread, often called the *primary* thread. However, it can create additional threads to execute code in parallel or concurrently with the primary thread. These threads are often called *worker* threads.

When to use multiple threads

You use multiple threads to increase the responsiveness of your application and to take advantage of a multiprocessor or multi-core system to increase the application's throughput.

Consider a desktop application, in which the primary thread is responsible for user interface elements and responds to user actions. Use worker threads to perform time-consuming operations that, otherwise, would occupy the primary thread and make the user interface non-responsive. You also can use a dedicated thread for network or device communication to be more responsive to incoming messages or events.

If your program performs operations that can be done in parallel, the total execution time can be decreased by performing those operations in separate threads and running the program on a multiprocessor or multi-core system. On such a system, use of multithreading might increase throughput along with the increased responsiveness.

How to use multithreading in .NET

Starting with the .NET Framework 4, the recommended way to utilize multithreading is to use Task Parallel Library (TPL) and Parallel LINQ (PLINQ). For more information, see Parallel programming.

Both TPL and PLINQ rely on the ThreadPool threads. The System.Threading.ThreadPool class provides a .NET application with a pool of worker threads. You also can use thread pool threads. For more information, see The managed thread pool.

At last, you can use the System.Threading.Thread class that represents a managed thread. For more information,

see Using threads and threading.

Multiple threads might need to access a shared resource. To keep the resource in a uncorrupted state and avoid race conditions, you must synchronize the thread access to it. You also might want to coordinate the interaction of multiple threads. .NET provides a range of types that you can use to synchronize access to a shared resource or coordinate thread interaction. For more information, see Overview of synchronization primitives.

Do handle exceptions in threads. Unhandled exceptions in threads generally terminate the process. For more information, see Exceptions in managed threads.

See also

- Threading objects and features
- Managed threading best practices
- Parallel Processing, Concurrency, and Async Programming in .NET
- About Processes and Threads

Exceptions in Managed Threads

8/22/2019 • 3 minutes to read • Edit Online

Starting with the .NET Framework version 2.0, the common language runtime allows most unhandled exceptions in threads to proceed naturally. In most cases this means that the unhandled exception causes the application to terminate.

> **NOTE**
>
> This is a significant change from the .NET Framework versions 1.0 and 1.1, which provide a backstop for many unhandled exceptions — for example, unhandled exceptions in thread pool threads. See Change from Previous Versions later in this topic.

The common language runtime provides a backstop for certain unhandled exceptions that are used for controlling program flow:

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

If any of these exceptions are unhandled in threads created by the common language runtime, the exception terminates the thread, but the common language runtime does not allow the exception to proceed further.

If these exceptions are unhandled in the main thread, or in threads that entered the runtime from unmanaged code, they proceed normally, resulting in termination of the application.

> **NOTE**
>
> It is possible for the runtime to throw an unhandled exception before any managed code has had a chance to install an exception handler. Even though managed code had no chance to handle such an exception, the exception is allowed to proceed naturally.

Exposing Threading Problems During Development

When threads are allowed to fail silently, without terminating the application, serious programming problems can go undetected. This is a particular problem for services and other applications which run for extended periods. As threads fail, program state gradually becomes corrupted. Application performance may degrade, or the application might become unresponsive.

Allowing unhandled exceptions in threads to proceed naturally, until the operating system terminates the program, exposes such problems during development and testing. Error reports on program terminations support debugging.

Change from Previous Versions

The most significant change pertains to managed threads. In the .NET Framework versions 1.0 and 1.1, the common language runtime provides a backstop for unhandled exceptions in the following situations:

- There is no such thing as an unhandled exception on a thread pool thread. When a task throws an exception that it does not handle, the runtime prints the exception stack trace to the console and then returns the thread to the thread pool.
- There is no such thing as an unhandled exception on a thread created with the Start method of the Thread class. When code running on such a thread throws an exception that it does not handle, the runtime prints the exception stack trace to the console and then gracefully terminates the thread.
- There is no such thing as an unhandled exception on the finalizer thread. When a finalizer throws an exception that it does not handle, the runtime prints the exception stack trace to the console and then allows the finalizer thread to resume running finalizers.

The foreground or background status of a managed thread does not affect this behavior.

For unhandled exceptions on threads originating in unmanaged code, the difference is more subtle. The runtime JIT-attach dialog preempts the operating system dialog for managed exceptions or native exceptions on threads that have passed through native code. The process terminates in all cases.

Migrating Code

In general, the change will expose previously unrecognized programming problems so that they can be fixed. In some cases, however, programmers might have taken advantage of the runtime backstop, for example to terminate threads. Depending on the situation, they should consider one of the following migration strategies:

- Restructure the code so the thread exits gracefully when a signal is received.
- Use the Thread.Abort method to abort the thread.
- If a thread must be stopped so that process termination can proceed, make the thread a background thread so that it is automatically terminated on process exit.

In all cases, the strategy should follow the design guidelines for exceptions. See Design Guidelines for Exceptions.

Application Compatibility Flag

As a temporary compatibility measure, administrators can place a compatibility flag in the `<runtime>` section of the application configuration file. This causes the common language runtime to revert to the behavior of versions 1.0 and 1.1.

```
<legacyUnhandledExceptionPolicy enabled="1"/>
```

Host Override

In the .NET Framework version 2.0, an unmanaged host can use the ICLRPolicyManager interface in the Hosting API to override the default unhandled exception policy of the common language runtime. The ICLRPolicyManager::SetUnhandledExceptionPolicy function is used to set the policy for unhandled exceptions.

See also

- Managed Threading Basics

Synchronizing data for multithreading

8/22/2019 • 3 minutes to read • Edit Online

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.

.NET 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 class library. See Overview of Synchronization Primitives, which includes a discussion of the Monitor class.
- Synchronized contexts. For .NET Framework and Xamarin applications, 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 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 with a value of 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

In .NET Framework and Xamarin applications only, 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

Foreground and Background Threads

8/22/2019 • 2 minutes to read • Edit Online

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

8/22/2019 • 3 minutes to read • Edit Online

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

IN WIN32	IN THE COMMON LANGUAGE RUNTIME
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.

Threads and fibers

The .NET threading model does not support fibers. You should not call into any unmanaged function that is implemented by using fibers. Such calls may result in a crash of the .NET runtime.

See also

- Thread.ApartmentState
- ThreadState
- ServicedComponent
- Thread
- Monitor

Thread Local Storage: Thread-Relative Static Fields and Data Slots

6/4/2019 • 3 minutes to read • Edit Online

You can use managed thread local storage (TLS) to store data that is unique to a thread and application domain. The .NET Framework provides two ways to use managed TLS: thread-relative static fields and data slots.

- Use thread-relative static fields (thread-relative `Shared` fields in Visual Basic) if you can anticipate your exact needs at compile time. Thread-relative static fields provide the best performance. They also give you the benefits of compile-time type checking.
- Use data slots when your actual requirements might be discovered only at run time. Data slots are slower and more awkward to use than thread-relative static fields, and data is stored as type Object, so you must cast it to the correct type before you use it.

In unmanaged C++, you use `TlsAlloc` to allocate slots dynamically and `__declspec(thread)` to declare that a variable should be allocated in thread-relative storage. Thread-relative static fields and data slots provide the managed version of this behavior.

In the .NET Framework 4, you can use the System.Threading.ThreadLocal<T> class to create thread-local objects that are initialized lazily when the object is first consumed. For more information, see Lazy Initialization.

Uniqueness of Data in Managed TLS

Whether you use thread-relative static fields or data slots, data in managed TLS is unique to the combination of thread and application domain.

- Within an application domain, one thread cannot modify data from another thread, even when both threads use the same field or slot.
- When a thread accesses the same field or slot from multiple application domains, a separate value is maintained in each application domain.

For example, if a thread sets the value of a thread-relative static field, enters another application domain, and then retrieves the value of the field, the value retrieved in the second application domain differs from the value in the first application domain. Setting a new value for the field in the second application domain does not affect the field's value in the first application domain.

Similarly, when a thread gets the same named data slot in two different application domains, the data in the first application domain remains independent of the data in the second application domain.

Thread-Relative Static Fields

If you know that a piece of data is always unique to a thread and application-domain combination, apply the ThreadStaticAttribute attribute to the static field. Use the field as you would use any other static field. The data in the field is unique to each thread that uses it.

Thread-relative static fields provide better performance than data slots and have the benefit of compile-time type checking.

Be aware that any class constructor code will run on the first thread in the first context that accesses the field. In all other threads or contexts in the same application domain, the fields will be initialized to `null` (`Nothing` in Visual

Basic) if they are reference types, or to their default values if they are value types. Therefore, you should not rely on class constructors to initialize thread-relative static fields. Instead, avoid initializing thread-relative static fields and assume that they are initialized to `null` (`Nothing`) or to their default values.

Data Slots

The .NET Framework provides dynamic data slots that are unique to a combination of thread and application-domain. There are two types of data slots: named slots and unnamed slots. Both are implemented by using the LocalDataStoreSlot structure.

- To create a named data slot, use the Thread.AllocateNamedDataSlot or Thread.GetNamedDataSlot method. To get a reference to an existing named slot, pass its name to the GetNamedDataSlot method.
- To create an unnamed data slot, use the Thread.AllocateDataSlot method.

For both named and unnamed slots, use the Thread.SetData and Thread.GetData methods to set and retrieve the information in the slot. These are static methods that always act on the data for the thread that is currently executing them.

Named slots can be convenient, because you can retrieve the slot when you need it by passing its name to the GetNamedDataSlot method, instead of maintaining a reference to an unnamed slot. However, if another component uses the same name for its thread-relative storage and a thread executes code from both your component and the other component, the two components might corrupt each other's data. (This scenario assumes that both components are running in the same application domain, and that they are not designed to share the same data.)

See also

- ContextStaticAttribute
- Thread.GetNamedDataSlot
- ThreadStaticAttribute
- CallContext
- Threading

Using threads and threading

5/21/2019 • 2 minutes to read • Edit Online

With .NET, you can write applications that perform multiple operations at the same time. Operations with the potential of holding up other operations 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.

NOTE

If you need more control over the behavior of the application's threads, you can manage the threads yourself. However, 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 and Task Parallel Library (TPL).

How to: Create and start a new thread

You create a new thread by creating a new instance of the System.Threading.Thread class and providing the name of the method that you want to execute on a new thread to the constructor. To start a created thread, call the Thread.Start method. For more information and examples, see the Creating threads and passing data at start time article and the Thread API reference.

How to: Stop a thread

To terminate the execution of a thread, use the Thread.Abort method. That method raises a ThreadAbortException on the thread on which it's invoked. For more information, see Destroying threads.

Beginning with the .NET Framework 4, you can use the System.Threading.CancellationToken to cancel a thread cooperatively. For more information, see Cancellation in managed threads.

Use the Thread.Join method to make the calling thread wait for the termination of the thread on which the method is invoked.

How to: Pause or interrupt a thread

You use the Thread.Sleep method to pause the current thread for a specified amount of time. You can interrupt a blocked thread by calling the Thread.Interrupt method. For more information, see Pausing and interrupting threads.

Thread properties

The following table presents some of the Thread properties:

PROPERTY	DESCRIPTION

PROPERTY	DESCRIPTION
IsAlive	Returns `true` if a thread has been started and has not yet terminated normally or aborted.
IsBackground	Gets or sets a Boolean that indicates if a thread is a background thread. Background threads are like foreground threads, but a background thread doesn't 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 Abort method on background threads that are still alive. For more information, see Foreground and Background Threads.
Name	Gets or sets the name of a thread. Most frequently used to discover individual threads when you debug.
Priority	Gets or sets a ThreadPriority value that is used by the operating system to prioritize thread scheduling. For more information, see Scheduling threads and the ThreadPriority reference.
ThreadState	Gets a ThreadState value containing the current states of a thread.

See also

- System.Threading.Thread
- Threads and Threading
- Parallel Programming

Creating threads and passing data at start time

1/23/2019 • 13 minutes to read • Edit Online

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.

```
using namespace System;
using namespace System::Threading;

public ref class ServerClass
{
public:
    // The method that will be called when the thread is started.
    void 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.");
    }

    static void StaticMethod()
    {
        Console::WriteLine(
            "ServerClass.StaticMethod is running on another thread.");

        // Pause for a moment to provide a delay to make
        // threads more apparent.
        Thread::Sleep(5000);
        Console::WriteLine(
```

```
        Console::WriteLine(
            "The static method called by the worker thread has ended.");
    }
};

public ref class Simple
{
public:
    static void Main()
    {
        ServerClass^ serverObject = gcnew ServerClass();

        // Create the thread object, passing in the
        // serverObject.InstanceMethod method using a
        // ThreadStart delegate.
        Thread^ InstanceCaller = gcnew Thread(
            gcnew ThreadStart(serverObject, &ServerClass::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.StaticMethod method using a
        // ThreadStart delegate.
        Thread^ StaticCaller = gcnew Thread(
            gcnew ThreadStart(&ServerClass::StaticMethod));

        // Start the thread.
        StaticCaller->Start();

        Console::WriteLine("The Main() thread calls this after "
            + "starting the new StaticCaller thread.");
    }
};

int main()
{
    Simple::Main();
}
// 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.
```

```
using System;
using System.Threading;

public class ServerClass
{
    // The method that will be called when the thread is started.
    public void 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.");
    }
```

```
    public static void StaticMethod()
    {
        Console.WriteLine(
            "ServerClass.StaticMethod is running on another thread.");

        // Pause for a moment to provide a delay to make
        // threads more apparent.
        Thread.Sleep(5000);
        Console.WriteLine(
            "The static method called by the worker thread has ended.");
    }
}

public class Simple
{
    public static void Main()
    {
        ServerClass serverObject = new ServerClass();

        // Create the thread object, passing in the
        // serverObject.InstanceMethod method using a
        // ThreadStart delegate.
        Thread InstanceCaller = new Thread(
            new ThreadStart(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.StaticMethod method using a
        // ThreadStart delegate.
        Thread StaticCaller = new Thread(
            new ThreadStart(ServerClass.StaticMethod));

        // Start the thread.
        StaticCaller.Start();

        Console.WriteLine("The Main() thread calls this after "
            + "starting the new StaticCaller thread.");
    }
}
// The example displays the 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.
```

```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

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. The following example demonstrates this technique:

```
using namespace System;
using namespace System::Threading;

// The ThreadWithState class contains the information needed for
// a task, and the method that executes the task.
//
public ref class ThreadWithState
{
private:
    // State information used in the task.
    String^ boilerplate;
    int numberValue;

    // The constructor obtains the state information.
public:
    ThreadWithState(String^ text, int number)
    {
        boilerplate = text;
        numberValue = number;
    }

    // The thread procedure performs the task, such as formatting
    // and printing a document.
    void ThreadProc()
    {
        Console::WriteLine(boilerplate, numberValue);
    }
};

// Entry point for the example.
//
public ref class Example
{
public:
    static void Main()
    {
        // Supply the state information required by the task.
        ThreadWithState^ tws = gcnew ThreadWithState(
            "This report displays the number {0}.", 42);

        // Create a thread to execute the task, and then
        // start the thread.
        Thread^ t = gcnew Thread(gcnew ThreadStart(tws, &ThreadWithState::ThreadProc));
        t->Start();
        Console::WriteLine("Main thread does some work, then waits.");
        t->Join();
        Console::WriteLine(
            "Independent task has completed; main thread ends.");
    }
};

int main()
{
    Example::Main();
}
// This 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.
```

```
using System;
using 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 string boilerplate;
    private int numberValue;

    // The constructor obtains the state information.
    public ThreadWithState(string text, int number)
    {
        boilerplate = text;
        numberValue = number;
    }

    // The thread procedure performs the task, such as formatting
    // and printing a document.
    public void ThreadProc()
    {
        Console.WriteLine(boilerplate, numberValue);
    }
}

// Entry point for the example.
//
public class Example
{
    public static void Main()
    {
        // Supply the state information required by the task.
        ThreadWithState tws = new ThreadWithState(
            "This report displays the number {0}.", 42);

        // Create a thread to execute the task, and then
        // start the thread.
        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
        Console.WriteLine("Main thread does some work, then waits.");
        t.Join();
        Console.WriteLine(
            "Independent task has completed; main thread ends.");
    }
}
// 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.
```

```
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 numberValue As Integer

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

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

Neither ThreadStart nor ParameterizedThreadStart delegate 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 shown in the next section.

Retrieving data from threads 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.

```
using namespace System;
using namespace System::Threading;

// Delegate that defines the signature for the callback method.
//
public delegate void ExampleCallback(int lineCount);

// 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 ref class ThreadWithState
{
private:
    // State information used in the task.
    String^ boilerplate;
    int numberValue;

    // Delegate used to execute the callback method when the
    // task is complete.
    ExampleCallback^ callback;

public:
    // The constructor obtains the state information and the
    // callback delegate.
    ThreadWithState(String^ text, int number,
        ExampleCallback^ callbackDelegate)
    {
        boilerplate = text;
        numberValue = number;
        callback = callbackDelegate;
    }

    // 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.
    void ThreadProc()
    {
        Console::WriteLine(boilerplate, numberValue);
        if (callback != nullptr)
        {
            callback(1);
        }
    }
};

// Entry point for the example.
//
public ref class Example
{
public:
    static void Main()
    {
        // Supply the state information required by the task.
        ThreadWithState^ tws = gcnew ThreadWithState(
            "This report displays the number {0}.",
            42,
            gcnew ExampleCallback(&Example::ResultCallback)
        );

        Thread^ t = gcnew Thread(gcnew ThreadStart(tws, &ThreadWithState::ThreadProc));
        t->Start();
        Console::WriteLine("Main thread does some work, then waits.");
        t->Join();
        Console::WriteLine(
            "Independent task has completed; main thread ends.");
    }

    // The callback method must match the signature of the
    // callback delegate.
    //
    static void ResultCallback(int lineCount)
    {
        Console::WriteLine(
            "Independent task printed {0} lines.", lineCount);
    }
};
```

```
int main()
{
    Example::Main();
}
// 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.
```

```
using System;
using 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 string boilerplate;
    private int numberValue;

    // Delegate used to execute the callback method when the
    // task is complete.
    private ExampleCallback callback;

    // The constructor obtains the state information and the
    // callback delegate.
    public ThreadWithState(string text, int number,
        ExampleCallback callbackDelegate)
    {
        boilerplate = text;
        numberValue = number;
        callback = callbackDelegate;
    }

    // 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 void ThreadProc()
    {
        Console.WriteLine(boilerplate, numberValue);
        if (callback != null)
            callback(1);
    }
}

// Delegate that defines the signature for the callback method.
//
public delegate void ExampleCallback(int lineCount);

// Entry point for the example.
//
public class Example
{
    public static void Main()
    {
        // Supply the state information required by the task.
        ThreadWithState tws = new ThreadWithState(
            "This report displays the number {0}.",
            42,
            new ExampleCallback(ResultCallback)
        );

        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
```

```
        t.Start();
        Console.WriteLine("Main thread does some work, then waits.");
        t.Join();
        Console.WriteLine(
            "Independent task has completed; main thread ends.");
    }

    // The callback method must match the signature of the
    // callback delegate.
    //
    public static void ResultCallback(int lineCount)
    {
        Console.WriteLine(
            "Independent task printed {0} lines.", lineCount);
    }
}
// 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.
```

```
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 numberValue 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
        numberValue = 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, numberValue)
        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
- Threading
- Using Threads and Threading

Pausing and interrupting threads

8/22/2019 • 3 minutes to read • Edit Online

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.

```
using System;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Interrupt a sleeping thread.
      var sleepingThread = new Thread(Example.SleepIndefinitely);
      sleepingThread.Name = "Sleeping";
      sleepingThread.Start();
      Thread.Sleep(2000);
      sleepingThread.Interrupt();

      Thread.Sleep(1000);

      sleepingThread = new Thread(Example.SleepIndefinitely);
      sleepingThread.Name = "Sleeping2";
      sleepingThread.Start();
      Thread.Sleep(2000);
      sleepingThread.Abort();
   }

   private static void SleepIndefinitely()
   {
      Console.WriteLine("Thread '{0}' about to sleep indefinitely.",
                        Thread.CurrentThread.Name);
      try {
         Thread.Sleep(Timeout.Infinite);
      }
      catch (ThreadInterruptedException) {
         Console.WriteLine("Thread '{0}' awoken.",
                           Thread.CurrentThread.Name);
      }
      catch (ThreadAbortException) {
         Console.WriteLine("Thread '{0}' aborted.",
                           Thread.CurrentThread.Name);
      }
      finally
      {
         Console.WriteLine("Thread '{0}' executing finally block.",
                           Thread.CurrentThread.Name);
      }
      Console.WriteLine("Thread '{0} finishing normal execution.",
                        Thread.CurrentThread.Name);
      Console.WriteLine();
   }
}
// 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.
```

```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
- Threading
- Using Threads and Threading
- Overview of Synchronization Primitives

Destroying threads

8/22/2019 • 2 minutes to read • Edit Online

The Thread.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.

```
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.
```

```
try
{
    // Code that is executing when the thread is aborted.
}
catch (ThreadAbortException ex)
{
    // 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.
}
// 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

1/23/2019 • 2 minutes to read • Edit Online

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

Cancellation in Managed Threads

8/22/2019 • 13 minutes to read • Edit Online

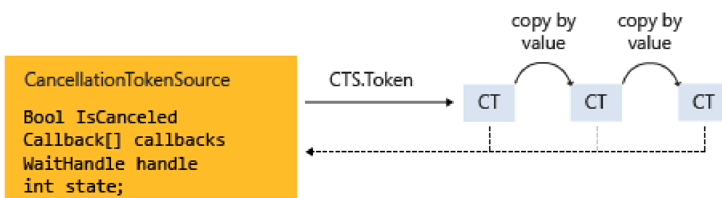
Starting with the .NET Framework 4, the .NET Framework uses a unified model for cooperative cancellation of asynchronous or long-running synchronous operations. This model is based on a lightweight object called a cancellation token. The object that invokes one or more cancelable operations, for example 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<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.

```
using System;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Create the token source.
      CancellationTokenSource cts = new CancellationTokenSource();

      // Pass the token to the cancelable operation.
      ThreadPool.QueueUserWorkItem(new WaitCallback(DoSomeWork), cts.Token);
      Thread.Sleep(2500);

      // Request cancellation.
      cts.Cancel();
      Console.WriteLine("Cancellation set in token source...");
      Thread.Sleep(2500);
      // Cancellation should have happened, so call Dispose.
      cts.Dispose();
   }

   // Thread 2: The listener
   static void DoSomeWork(object obj)
   {
      CancellationToken token = (CancellationToken)obj;

      for (int i = 0; i < 100000; i++) {
         if (token.IsCancellationRequested)
         {
            Console.WriteLine("In iteration {0}, cancellation has been requested...",
                              i + 1);
            // Perform cleanup if necessary.
            //...
            // Terminate the operation.
            break;
         }
         // Simulate some work.
         Thread.SpinWait(500000);
      }
   }
}
// The example displays output like the following:
//       Cancellation set in token source...
//       In iteration 1430, cancellation has been requested...
```

```
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.

```
using System;
using System.Threading;

class CancelableObject
{
   public string id;

   public CancelableObject(string id)
   {
      this.id = id;
   }

   public void Cancel()
   {
      Console.WriteLine("Object {0} Cancel callback", id);
      // Perform object cancellation here.
   }
}

public class Example
{
   public static void Main()
   {
      CancellationTokenSource cts = new CancellationTokenSource();
      CancellationToken token = cts.Token;

      // User defined Class with its own method for cancellation
      var obj1 = new CancelableObject("1");
      var obj2 = new CancelableObject("2");
      var obj3 = new CancelableObject("3");

      // Register the object's cancel method with the token's
      // cancellation request.
      token.Register(() => obj1.Cancel());
      token.Register(() => obj2.Cancel());
      token.Register(() => obj3.Cancel());

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

```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.

```
static void NestedLoops(Rectangle rect, CancellationToken token)
{
   for (int x = 0; x < rect.columns && !token.IsCancellationRequested; x++) {
      for (int y = 0; y < rect.rows; y++) {
         // Simulating work.
         Thread.SpinWait(5000);
         Console.Write("{0},{1} ", x, y);
      }

      // Assume that we know that the inner loop is very fast.
      // Therefore, checking once per row is sufficient.
      if (token.IsCancellationRequested) {
         // Cleanup or undo here if necessary...
         Console.WriteLine("\r\nCancelling after row {0}.", x);
         Console.WriteLine("Press any key to exit.");
         // then...
         break;
         // ...or, if using Task:
         // token.ThrowIfCancellationRequested();
      }
   }
}
```

```
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.

```csharp
using System;
using System.Net;
using System.Threading;

class Example
{
    static void Main()
    {
        CancellationTokenSource cts = new CancellationTokenSource();

        StartWebRequest(cts.Token);

        // cancellation will cause the web
        // request to be cancelled
        cts.Cancel();
    }

    static void StartWebRequest(CancellationToken token)
    {
        WebClient wc = new WebClient();
        wc.DownloadStringCompleted += (s, e) => Console.WriteLine("Request completed.");

        // Cancellation on the token will
        // call CancelAsync on the WebClient.
        token.Register(() =>
        {
            wc.CancelAsync();
            Console.WriteLine("Request cancelled!");
        });

        Console.WriteLine("Starting request.");
        wc.DownloadStringAsync(new Uri("http://www.contoso.com"));
    }
}
```

```vb
Imports System.Net
Imports System.Threading

Class Example
    Private Shared Sub Main()
        Dim cts As New CancellationTokenSource()

        StartWebRequest(cts.Token)

        ' cancellation will cause the web
        ' request to be cancelled
        cts.Cancel()
    End Sub

    Private Shared Sub StartWebRequest(token As CancellationToken)
        Dim wc As New WebClient()
        wc.DownloadStringCompleted += Function(s, e) Console.WriteLine("Request completed.")

        ' Cancellation on the token will
        ' call CancelAsync on the WebClient.
        token.Register(Function()
        wc.CancelAsync()
        Console.WriteLine("Request cancelled!")

End Function)

        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.

```
// Wait on the event if it is not signaled.
int eventThatSignaledIndex =
       WaitHandle.WaitAny(new WaitHandle[] { mre, token.WaitHandle },
                          new TimeSpan(0, 0, 20));
```

```
' 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.

```csharp
try
{
    // mres is a ManualResetEventSlim
    mres.Wait(token);
}
catch (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("The wait operation was canceled.");
    throw;
}

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

```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.

```
public void DoWork(CancellationToken externalToken)
{
   // Create a new token that combines the internal and external tokens.
   this.internalToken = internalTokenSource.Token;
   this.externalToken = externalToken;

   using (CancellationTokenSource linkedCts =
           CancellationTokenSource.CreateLinkedTokenSource(internalToken, externalToken))
   {
       try {
           DoWorkInternal(linkedCts.Token);
       }
       catch (OperationCanceledException) {
           if (internalToken.IsCancellationRequested) {
               Console.WriteLine("Operation timed out.");
           }
           else if (externalToken.IsCancellationRequested) {
               Console.WriteLine("Cancelling per user request.");
               externalToken.ThrowIfCancellationRequested();
           }
       }
   }
}
```

```
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 follow these

guidelines. For more information, see Task Cancellation and How to: Cancel a PLINQ Query.

See also

- Managed Threading Basics

Canceling threads cooperatively

9/6/2018 • 2 minutes to read • Edit Online

Prior to the .NET Framework 4, the .NET Framework provided no built-in way to cancel a thread cooperatively after it was started. However, starting with the .NET Framework 4, you can use a System.Threading.CancellationToken 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.

```
using System;
using System.Threading;

public class ServerClass
{
   public static void StaticMethod(object obj)
   {
      CancellationToken ct = (CancellationToken)obj;
      Console.WriteLine("ServerClass.StaticMethod is running on another thread.");

      // Simulate work that can be canceled.
      while (!ct.IsCancellationRequested) {
         Thread.SpinWait(50000);
      }
      Console.WriteLine("The worker thread has been canceled. Press any key to exit.");
      Console.ReadKey(true);
   }
}

public class Simple
{
   public static void Main()
   {
      // The Simple class controls access to the token source.
      CancellationTokenSource cts = new CancellationTokenSource();

      Console.WriteLine("Press 'C' to terminate the application...\n");
      // Allow the UI thread to capture the token source, so that it
      // can issue the cancel command.
      Thread t1 = new Thread(() => { if (Console.ReadKey(true).KeyChar.ToString().ToUpperInvariant() == "C")
                                     cts.Cancel(); } );

      // ServerClass sees only the token, not the token source.
      Thread t2 = new Thread(new ParameterizedThreadStart(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();
   }
}
// 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.
```

```
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

How to: Listen for Cancellation Requests by Polling

9/6/2018 • 3 minutes to read • Edit Online

The following example shows one way that user code can poll a cancellation token at regular intervals to see whether cancellation has been requested from the calling thread. This example uses the System.Threading.Tasks.Task type, but the same pattern applies to asynchronous operations created directly by the System.Threading.ThreadPool type or the System.Threading.Thread type.

Example

Polling requires some kind of loop or recursive code that can periodically read the value of the Boolean IsCancellationRequested property. If you are using the System.Threading.Tasks.Task type and you are waiting for the task to complete on the calling thread, you can use the ThrowIfCancellationRequested method to check the property and throw the exception. By using this method, you ensure that the correct exception is thrown in response to a request. If you are using a Task, then calling this method is better than manually throwing an OperationCanceledException. If you do not have to throw the exception, then you can just check the property and return from the method if the property is `true`.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public struct Rectangle
{
   public int columns;
   public int rows;
}

class CancelByPolling
{
   static void Main()
   {
      var tokenSource = new CancellationTokenSource();
      // Toy object for demo purposes
      Rectangle rect = new Rectangle() { columns = 1000, rows = 500 };

      // Simple cancellation scenario #1. Calling thread does not wait
      // on the task to complete, and the user delegate simply returns
      // on cancellation request without throwing.
      Task.Run(() => NestedLoops(rect, tokenSource.Token), tokenSource.Token);

      // Simple cancellation scenario #2. Calling thread does not wait
      // on the task to complete, and the user delegate throws
      // OperationCanceledException to shut down task and transition its state.
      // Task.Run(() => PollByTimeSpan(tokenSource.Token), tokenSource.Token);

      Console.WriteLine("Press 'c' to cancel");
      if (Console.ReadKey(true).KeyChar == 'c') {
          tokenSource.Cancel();
          Console.WriteLine("Press any key to exit.");
      }

      Console.ReadKey();
      tokenSource.Dispose();
  }

   static void NestedLoops(Rectangle rect, CancellationToken token)
   {
      for (int x = 0; x < rect.columns && !token.IsCancellationRequested; x++) {
         for (int y = 0; y < rect.rows; y++) {
            // Simulating work.
            Thread.SpinWait(5000);
            Console.Write("{0},{1} ", x, y);
         }

         // Assume that we know that the inner loop is very fast.
         // Therefore, checking once per row is sufficient.
         if (token.IsCancellationRequested) {
            // Cleanup or undo here if necessary...
            Console.WriteLine("\r\nCancelling after row {0}.", x);
            Console.WriteLine("Press any key to exit.");
            // then...
            break;
            // ...or, if using Task:
            // token.ThrowIfCancellationRequested();
         }
      }
   }
}
```

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

Public Structure Rectangle
    Public columns As Integer
    Public rows As Integer
End Structure

Class CancelByPolling
    Shared Sub Main()
        Dim tokenSource As New CancellationTokenSource()
        ' Toy object for demo purposes
        Dim rect As New Rectangle()
        rect.columns = 1000
        rect.rows = 500

        ' Simple cancellation scenario #1. Calling thread does not wait
        ' on the task to complete, and the user delegate simply returns
        ' on cancellation request without throwing.
        Task.Run(Sub() NestedLoops(rect, tokenSource.Token), tokenSource.Token)

        ' Simple cancellation scenario #2. Calling thread does not wait
        ' on the task to complete, and the user delegate throws
        ' OperationCanceledException to shut down task and transition its state.
        ' Task.Run(Sub() PollByTimeSpan(tokenSource.Token), tokenSource.Token)

        Console.WriteLine("Press 'c' to cancel")
        If Console.ReadKey(True).KeyChar = "c"c Then

            tokenSource.Cancel()
            Console.WriteLine("Press any key to exit.")
        End If

        Console.ReadKey()
        tokenSource.Dispose()
    End Sub

    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
End Class
```

Calling ThrowIfCancellationRequested is extremely fast and does not introduce significant overhead in loops.

If you are calling ThrowIfCancellationRequested, you only have to explicitly check the IsCancellationRequested property if you have other work to do in response to the cancellation besides throwing the exception. In this example, you can see that the code actually accesses the property twice: once in the explicit access and again in the

ThrowIfCancellationRequested method. But because the act of reading the IsCancellationRequested property involves only one volatile read instruction per access, the double access is not significant from a performance perspective. It is still preferable to call the method rather than manually throw the OperationCanceledException.

See also

- Cancellation in Managed Threads

How to: Register Callbacks for Cancellation Requests

8/22/2019 • 2 minutes to read • Edit Online

The following example shows how to register a delegate that will be invoked when a IsCancellationRequested property becomes true due to a call to Cancel on the object that created the token. Use this technique for cancelling asynchronous operations that do not natively support the unified cancellation framework, and for unblocking methods that might be waiting for an asynchronous operation to finish.

> **NOTE**
>
> When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the examples below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

Example

In the following example, the CancelAsync method is registered as the method to be invoked when cancellation is requested through the cancellation token.

```
using System;
using System.Net;
using System.Threading;
using System.Threading.Tasks;

class CancelWithCallback
{
   static void Main()
   {
      var cts = new CancellationTokenSource();
      var token = cts.Token;

      // Start cancelable task.
      Task t = Task.Run( () => {
                    WebClient wc = new WebClient();

                    // Create an event handler to receive the result.
                    wc.DownloadStringCompleted += (obj, e) => {
                               // Check status of WebClient, not external token.
                               if (!e.Cancelled) {
                                  Console.WriteLine("The download has completed:\n");
                                  Console.WriteLine(e.Result + "\n\nPress any key.");
                               }
                               else {
                                  Console.WriteLine("The download was canceled.");
                               }
                    };

                    // Do not initiate download if the external token
                    // has already been canceled.
                    if (!token.IsCancellationRequested) {
                       // Register the callback to a method that can unblock.
                       using (CancellationTokenRegistration ctr = token.Register(() => wc.CancelAsync()))
                       {
                          Console.WriteLine("Starting request\n");
                          wc.DownloadStringAsync(new Uri("http://www.contoso.com"));
                       }
                    }
               }, token);

      Console.WriteLine("Press 'c' to cancel.\n");
      char ch = Console.ReadKey().KeyChar;
      Console.WriteLine();
      if (ch == 'c')
         cts.Cancel();

      Console.WriteLine("Press any key to exit.");
      Console.ReadKey();
      cts.Dispose();
   }
}
```

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

Class CancelWithCallback
    Shared Sub Main()
        Dim cts As New CancellationTokenSource()
        Dim token As CancellationToken = cts.Token

        ' Start cancelable task.
        Dim t As Task = Task.Run(
           Sub()
              Dim wc As New WebClient()

              ' Create an event handler to receive the result.
              AddHandler wc.DownloadStringCompleted,
                         Sub(obj, e)
                            ' Check status of WebClient, not external token.
                            If Not e.Cancelled Then
                               Console.WriteLine("The download has completed:" + vbCrLf)
                               Console.WriteLine(e.Result + vbCrLf + vbCrLf + "Press any key.")
                            Else
                               Console.WriteLine("Download was canceled.")
                            End If
                          End Sub
              Using ctr As CancellationTokenRegistration = token.Register(Sub() wc.CancelAsync())
                 Console.WriteLine("Starting request..." + vbCrLf)
                 wc.DownloadStringAsync(New Uri("http://www.contoso.com"))
              End Using
           End Sub, token)

        Console.WriteLine("Press 'c' to cancel." + vbCrLf)
        Dim ch As Char = Console.ReadKey().KeyChar
        Console.WriteLine()

        If ch = "c"c Then
            cts.Cancel()
        End If
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
        cts.Dispose()
    End Sub
End Class
```

If cancellation has already been requested when the callback is registered, the callback is still guaranteed to be called. In this particular case, the CancelAsync method will do nothing if no asynchronous operation is in progress, so it is always safe to call the method.

See also

- Cancellation in Managed Threads

How to: Listen for Cancellation Requests That Have Wait Handles

8/22/2019 • 5 minutes to read • Edit Online

If a method is blocked while it is waiting for an event to be signaled, it cannot check the value of the cancellation token and respond in a timely manner. The first example shows how to solve this problem when you are working with events such as System.Threading.ManualResetEvent that do not natively support the unified cancellation framework. The second example shows a more streamlined approach that uses System.Threading.ManualResetEventSlim, which does support unified cancellation.

> **NOTE**
>
> When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the examples below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

Example

The following example uses a ManualResetEvent to demonstrate how to unblock wait handles that do not support unified cancellation.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CancelOldStyleEvents
{
   // Old-style MRE that doesn't support unified cancellation.
   static ManualResetEvent mre = new ManualResetEvent(false);

   static void Main()
   {
      var cts = new CancellationTokenSource();

      // Pass the same token source to the delegate and to the task instance.
      Task.Run(() => DoWork(cts.Token), cts.Token);
      Console.WriteLine("Press s to start/restart, p to pause, or c to cancel.");
      Console.WriteLine("Or any other key to exit.");

      // Old-style UI thread.
      bool goAgain = true;
      while (goAgain) {
         char ch = Console.ReadKey(true).KeyChar;

         switch (ch) {
            case 'c':
               cts.Cancel();
               break;
            case 'p':
               mre.Reset();
               break;
            case 's':
               mre.Set();
               break;
            default:
```

```
               goAgain = false;
               break;
         }

         Thread.Sleep(100);
      }
      cts.Dispose();
   }

   static void DoWork(CancellationToken token)
   {
      while (true)
      {
         // Wait on the event if it is not signaled.
         int eventThatSignaledIndex =
                WaitHandle.WaitAny(new WaitHandle[] { mre, token.WaitHandle },
                                   new TimeSpan(0, 0, 20));

         // Were we canceled while waiting?
         if (eventThatSignaledIndex == 1) {
            Console.WriteLine("The wait operation was canceled.");
            throw new OperationCanceledException(token);
         }
         // Were we canceled while running?
         else if (token.IsCancellationRequested) {
            Console.WriteLine("I was canceled while running.");
            token.ThrowIfCancellationRequested();
         }
         // Did we time out?
         else if (eventThatSignaledIndex == WaitHandle.WaitTimeout) {
            Console.WriteLine("I timed out.");
            break;
         }
         else {
            Console.Write("Working... ");
            // Simulating work.
            Thread.SpinWait(5000000);
         }
      }
   }
}
```

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

Class CancelOldStyleEvents
   ' Old-style MRE that doesn't support unified cancellation.
   Shared mre As New ManualResetEvent(False)

   Shared Sub Main()
      Dim cts As New CancellationTokenSource()

      ' Pass the same token source to the delegate and to the task instance.
      Task.Run(Sub() DoWork(cts.Token), cts.Token)
        Console.WriteLine("Press c to cancel, p to pause, or s to start/restart.")
        Console.WriteLine("Or any other key to exit.")

        ' Old-style UI thread.
        Dim goAgain As Boolean = True
        While goAgain
            Dim ch As Char = Console.ReadKey(True).KeyChar
            Select Case ch
                Case "c"c
                    cts.Cancel()
                Case "p"c
                    mre.Reset()
                Case "s"c
                    mre.Set()
                Case Else
                    goAgain = False
         End Select

         Thread.Sleep(100)
      End While
      cts.Dispose()
   End Sub

    Shared Sub DoWork(ByVal token As CancellationToken)
        While True
            ' 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))

            ' Were we canceled while waiting?
            ' The first If statement is equivalent to
            ' token.ThrowIfCancellationRequested()
            If eventThatSignaledIndex = 1 Then
                Console.WriteLine("The wait operation was canceled.")
                Throw New OperationCanceledException(token)
            ' Were we canceled while running?
            ElseIf token.IsCancellationRequested = True Then
                Console.WriteLine("Cancelling per user request.")
                token.ThrowIfCancellationRequested()
            ' Did we time out?
            ElseIf eventThatSignaledIndex = WaitHandle.WaitTimeout Then
                Console.WriteLine("The wait operation timed out.")
                Exit While
            Else
                ' Simulating work.
                Console.Write("Working... ")
                Thread.SpinWait(5000000)
            End If
        End While
    End Sub
End Class
```

Example

The following example uses a ManualResetEventSlim to demonstrate how to unblock coordination primitives that do support unified cancellation. The same approach can be used with other lightweight coordination primitives, such as Semaphore `Slim` and CountdownEvent.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CancelNewStyleEvents
{
   // New-style MRESlim that supports unified cancellation
   // in its Wait methods.
   static ManualResetEventSlim mres = new ManualResetEventSlim(false);

   static void Main()
   {
      var cts = new CancellationTokenSource();

      // Pass the same token source to the delegate and to the task instance.
      Task.Run(() => DoWork(cts.Token), cts.Token);
      Console.WriteLine("Press c to cancel, p to pause, or s to start/restart,");
      Console.WriteLine("or any other key to exit.");

      // New-style UI thread.
         bool goAgain = true;
         while (goAgain)
         {
             char ch = Console.ReadKey(true).KeyChar;

             switch (ch)
             {
                 case 'c':
                     // Token can only be canceled once.
                     cts.Cancel();
                     break;
                 case 'p':
                     mres.Reset();
                     break;
                 case 's':
                     mres.Set();
                     break;
                 default:
                     goAgain = false;
                     break;
             }

             Thread.Sleep(100);
         }
         cts.Dispose();
     }

     static void DoWork(CancellationToken token)
     {

         while (true)
         {
             if (token.IsCancellationRequested)
             {
                 Console.WriteLine("Canceled while running.");
                 token.ThrowIfCancellationRequested();
             }

             // Wait on the event to be signaled
             // or the token to be canceled,
             // whichever comes first. The token
```

```
            // whichever comes first. The token
            // will throw an exception if it is canceled
            // while the thread is waiting on the event.
            try
            {
                // mres is a ManualResetEventSlim
                mres.Wait(token);
            }
            catch (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("The wait operation was canceled.");
                throw;
            }

            Console.Write("Working...");
            // Simulating work.
            Thread.SpinWait(500000);
        }
    }
}
```

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

Class CancelNewStyleEvents

    ' New-style MRESlim that supports unified cancellation
    ' in its Wait methods.
    Shared mres As ManualResetEventSlim = New ManualResetEventSlim(False)

    Shared Sub Main()

        Dim cts As New CancellationTokenSource()

        ' Pass the same token source to the delegate and to the task instance.
      Task.Run(Sub() DoWork(cts.Token), cts.Token)
        Console.WriteLine("Press c to cancel, p to pause, or s to start/restart,")
        Console.WriteLine("or any other key to exit.")

        ' New-style UI thread.
        Dim goAgain As Boolean = True
        While goAgain = True

            Dim ch As Char = Console.ReadKey(True).KeyChar

            Select Case ch
                Case "c"c
                    ' Token can only be canceled once.
                    cts.Cancel()
                Case "p"c
                    mres.Reset()
                Case "s"c
                    mres.Set()
                Case Else
                    goAgain = False
            End Select

            Thread.Sleep(100)
        End While
        cts.Dispose()
    End Sub

    Shared Sub DoWork(ByVal token As CancellationToken)
        While True
```

```vb
            If token.IsCancellationRequested Then
                Console.WriteLine("Canceled while running.")
                token.ThrowIfCancellationRequested()
            End If

            ' Wait on the event to be signaled
            ' or the token to be canceled,
            ' whichever comes first. The token
            ' will throw an exception if it is canceled
            ' while the thread is waiting on the event.
            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)
        End While
    End Sub
End Class
```

See also

- Cancellation in Managed Threads

How to: Listen for Multiple Cancellation Requests

8/22/2019 • 4 minutes to read • Edit Online

This example shows how to listen to two cancellation tokens simultaneously so that you can cancel an operation if either token requests it.

> **NOTE**
>
> When "Just My Code" is enabled, Visual Studio in some cases will break on the line that throws the exception and display an error message that says "exception not handled by user code." This error is benign. You can press F5 to continue from it, and see the exception-handling behavior that is demonstrated in the examples below. To prevent Visual Studio from breaking on the first error, just uncheck the "Just My Code" checkbox under **Tools, Options, Debugging, General**.

Example

In the following example, the CreateLinkedTokenSource method is used to join two tokens into one token. This enables the token to be passed to methods that take just one cancellation token as an argument. The example demonstrates a common scenario in which a method must observe both a token passed in from outside the class, and a token generated inside the class.

```
using System;
using System.Threading;
using System.Threading.Tasks;

class LinkedTokenSourceDemo
{
  static void Main()
  {
      WorkerWithTimer worker = new WorkerWithTimer();
      CancellationTokenSource cts = new CancellationTokenSource();

      // Task for UI thread, so we can call Task.Wait wait on the main thread.
      Task.Run(() =>
      {
          Console.WriteLine("Press 'c' to cancel within 3 seconds after work begins.");
          Console.WriteLine("Or let the task time out by doing nothing.");
          if (Console.ReadKey(true).KeyChar == 'c')
              cts.Cancel();
      });

      // Let the user read the UI message.
      Thread.Sleep(1000);

      // Start the worker task.
      Task task = Task.Run(() => worker.DoWork(cts.Token), cts.Token);

      try {
          task.Wait(cts.Token);
      }
      catch (OperationCanceledException e) {
          if (e.CancellationToken == cts.Token)
              Console.WriteLine("Canceled from UI thread throwing OCE.");
      }
      catch (AggregateException ae) {
          Console.WriteLine("AggregateException caught: " + ae.InnerException);
          foreach (var inner in ae.InnerExceptions) {
              Console.WriteLine(inner.Message + inner.Source);
```

```
        }
    }

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
    cts.Dispose();
  }
}

class WorkerWithTimer
{
  CancellationTokenSource internalTokenSource = new CancellationTokenSource();
  CancellationToken internalToken;
  CancellationToken externalToken;
  Timer timer;

  public WorkerWithTimer()
  {
      internalTokenSource = new CancellationTokenSource();
      internalToken = internalTokenSource.Token;

      // A toy cancellation trigger that times out after 3 seconds
      // if the user does not press 'c'.
      timer = new Timer(new TimerCallback(CancelAfterTimeout), null, 3000, 3000);
  }

   public void DoWork(CancellationToken externalToken)
   {
      // Create a new token that combines the internal and external tokens.
      this.internalToken = internalTokenSource.Token;
      this.externalToken = externalToken;

      using (CancellationTokenSource linkedCts =
              CancellationTokenSource.CreateLinkedTokenSource(internalToken, externalToken))
      {
          try {
              DoWorkInternal(linkedCts.Token);
          }
          catch (OperationCanceledException) {
              if (internalToken.IsCancellationRequested) {
                  Console.WriteLine("Operation timed out.");
              }
              else if (externalToken.IsCancellationRequested) {
                  Console.WriteLine("Cancelling per user request.");
                  externalToken.ThrowIfCancellationRequested();
              }
          }
      }
   }

   private void DoWorkInternal(CancellationToken token)
   {
      for (int i = 0; i < 1000; i++)
      {
          if (token.IsCancellationRequested)
          {
              // We need to dispose the timer if cancellation
              // was requested by the external token.
              timer.Dispose();

              // Throw the exception.
              token.ThrowIfCancellationRequested();
          }

           // Simulating work.
          Thread.SpinWait(7500000);
          Console.Write("working... ");
      }
   }
```

```
    public void CancelAfterTimeout(object state)
    {
       Console.WriteLine("\r\nTimer fired.");
       internalTokenSource.Cancel();
       timer.Dispose();
    }
}
```

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

Class LinkedTokenSourceDemo
   Shared Sub Main()
      Dim worker As New WorkerWithTimer()
      Dim cts As New CancellationTokenSource()

      ' Task for UI thread, so we can call Task.Wait wait on the main thread.
      Task.Run(Sub()
                  Console.WriteLine("Press 'c' to cancel within 3 seconds after work begins.")
                  Console.WriteLine("Or let the task time out by doing nothing.")
                  If Console.ReadKey(True).KeyChar = "c"c Then
                     cts.Cancel()
                  End If
               End Sub)
      ' Let the user read the UI message.
      Thread.Sleep(1000)

      ' Start the worker task.
      Dim t As Task = Task.Run(Sub() worker.DoWork(cts.Token), cts.Token)
      Try
         t.Wait()
      Catch ae As AggregateException
         For Each inner In ae.InnerExceptions
            Console.WriteLine(inner.Message)
         Next
      End Try

      Console.WriteLine("Press any key to exit.")
      Console.ReadKey()
      cts.Dispose()
    End Sub
End Class

Class WorkerWithTimer
   Dim internalTokenSource As CancellationTokenSource
   Dim token As CancellationToken
   Dim timer As Timer

   Public Sub New()
      internalTokenSource = New CancellationTokenSource()
      token = internalTokenSource.Token

      ' A toy cancellation trigger that times out after 3 seconds
      ' if the user does not press 'c'.
      timer = New Timer(New TimerCallback(AddressOf CancelAfterTimeout), Nothing, 3000, 3000)
   End Sub

   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

   Private Sub DoWorkInternal(ByVal token As CancellationToken)
      For i As Integer = 0 To 1000
         If token.IsCancellationRequested Then
            ' We need to dispose the timer if cancellation
            ' was requested by the external token.
           timer.Dispose()

            ' Output for demonstration purposes.
            Console.WriteLine(vbCrLf + "Cancelling per request.")

            ' Throw the exception.
            token.ThrowIfCancellationRequested()
         End If

         ' Simulating work.
         Thread.SpinWait(7500000)
         Console.Write("working... ")
      Next
   End Sub

   Public Sub CancelAfterTimeout(ByVal state As Object)
      Console.WriteLine(vbCrLf + "Timer fired.")
      internalTokenSource.Cancel()
      timer.Dispose()
   End Sub
End Class
```

When the linked token throws an OperationCanceledException, the token that is passed to the exception is the linked token, not either of the predecessor tokens. To determine which of the tokens was canceled, check the status of the predecessor tokens directly.

In this example, AggregateException should never be thrown, but it is caught here because in real-world scenarios any other exceptions besides OperationCanceledException that are thrown from the task delegate are wrapped in a AggregateException.

See also

- Cancellation in Managed Threads

Managed threading best practices

8/20/2019 • 7 minutes to read • Edit Online

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**
>
> Starting with 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 .NET.

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 an object named `lockObject`. If the lock is not obtained in 300 milliseconds, Monitor.TryEnter returns `false`.

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

```
if (Monitor.TryEnter(lockObject, 300)) {
    try {
        // Place code protected by the Monitor here.
    }
    finally {
        Monitor.Exit(lockObject);
    }
}
else {
    // Code to execute if the attempt times out.
}
```

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.

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.

Number of processors

Whether there are multiple processors or only one processor available on a system can influence multithreaded architecture. For more information, see Number of Processors.

Use the Environment.ProcessorCount property to determine the number of processors available at runtime.

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:

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

```
lock(lockObject)
{
    myField++;
}
```

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

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

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

NOTE

In the .NET Framework 2.0 and later, use the Add method for atomic 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).

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

```
if (x == null)
{
    lock (lockObject)
    {
        if (x == null)
        {
            x = y;
        }
    }
}
```

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

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

```
System.Threading.Interlocked.CompareExchange(ref x, y, null);
```

> **NOTE**
>
> Beginning with .NET Framework 2.0, the CompareExchange<T>(T, T, T) method overload provides a type-safe alternative for reference types.

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

- Threading
- Threads and Threading

Threading objects and features

2/6/2019 • 2 minutes to read • Edit Online

Along with the System.Threading.Thread class, .NET provides a number of classes that help you develop multithreaded applications. The following articles provide overview of those classes:

TITLE	DESCRIPTION
The managed thread pool	Describes the System.Threading.ThreadPool class, which provides a pool of worker threads that are managed by .NET.
Timers	Describes .NET timers that can be used in a multithreaded environment.
Overview of synchronization primitives	Describes types that can be used to synchronize access to a shared resource or control thread interaction.
EventWaitHandle	Describes the System.Threading.EventWaitHandle class, which represents a thread synchronization event.
CountdownEvent	Describes the System.Threading.CountdownEvent class, which represents a thread synchronization event that becomes set when its count is zero.
Mutexes	Describes the System.Threading.Mutex class, which grants exclusive access to a shared resource.
Semaphore and SemaphoreSlim	Describes the System.Threading.Semaphore class, which limits number of threads that can access a shared resource or a pool of resources concurrently.
Barrier	Describes the System.Threading.Barrier class, which implements the barrier pattern for coordination of threads in phased operations.
SpinLock	Describes the System.Threading.SpinLock structure, which is a lightweight alternative to the System.Threading.Monitor class for certain low-level locking scenarios.
SpinWait	Describes the System.Threading.SpinWait structure, which provides support for spin-based waiting.

See also

- System.Threading.Monitor
- System.Threading.WaitHandle
- System.ComponentModel.BackgroundWorker
- System.Threading.Tasks.Parallel
- System.Threading.Tasks.Task
- Using threads and threading
- Asynchronous File I/O

- Parallel Programming
- Task Parallel Library (TPL)

The managed thread pool

1/23/2019 • 4 minutes to read • Edit Online

The System.Threading.ThreadPool class provides your application with a pool of worker threads that are managed by the system, allowing you to concentrate on application tasks rather than thread management. If you have short tasks that require background processing, the managed thread pool is an easy way to take advantage of multiple threads. Use of the thread pool is significantly easier in Framework 4 and later, since you can create Task and Task<TResult> objects that perform asynchronous tasks on thread pool threads.

.NET uses thread pool threads for many purposes, including Task Parallel Library (TPL) operations, asynchronous I/O completion, timer callbacks, registered wait operations, asynchronous method calls using delegates, and System.Net socket connections.

Thread pool characteristics

Thread pool threads are background threads. Each thread uses the default stack size, runs at the default priority, and is in the multithreaded apartment. Once a thread in the thread pool completes its task, it's returned to a queue of waiting threads. From this moment it can be reused. This reuse enables applications to avoid the cost of creating a new thread for each task.

There is only one thread pool per process.

Exceptions in thread pool threads

Unhandled exceptions in thread pool threads terminate the process. There are three exceptions to this rule:

- A System.Threading.ThreadAbortException is thrown in a thread pool thread because Thread.Abort was called.
- A System.AppDomainUnloadedException is thrown in a thread pool 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.

Maximum number of thread pool threads

The number of operations that can be queued to the thread pool is limited only by available memory. However, the thread pool limits the number of threads that can be active in the process simultaneously. If all thread pool threads are busy, additional work items are queued until threads to execute them become available. Beginning with the .NET Framework 4, the default size of the thread pool for a process depends on several factors, such as the size of the virtual address space. A process can call the ThreadPool.GetMaxThreads method to determine the number of threads.

You can control the maximum number of threads by using the ThreadPool.GetMaxThreads and ThreadPool.SetMaxThreads methods.

> **NOTE**
>
> Code that hosts the common language runtime can set the size using the `ICorThreadpool::CorSetMaxThreads` method.

Thread pool minimums

The thread pool provides new worker threads or I/O completion threads on demand until it reaches a specified minimum for each category. You can use the ThreadPool.GetMinThreads method to obtain these minimum values.

NOTE

When demand is low, the actual number of thread pool threads can fall below the minimum values.

When a minimum is reached, the thread pool can create additional threads or wait until some tasks complete. Beginning with the .NET Framework 4, the thread pool creates and destroys worker threads in order to optimize throughput, which is defined as the number of tasks that complete per unit of time. Too few threads might not make optimal use of available resources, whereas too many threads could increase resource contention.

Caution

You can use the ThreadPool.SetMinThreads method to increase the minimum number of idle threads. However, unnecessarily increasing these values can cause performance problems. If too many tasks start at the same time, all of them might appear to be slow. In most cases the thread pool will perform better with its own algorithm for allocating threads.

Using the thread pool

Beginning with the .NET Framework 4, the easiest way to use the thread pool is to use the Task Parallel Library (TPL). By default, TPL types like Task and Task<TResult> use thread pool threads to run tasks.

You can also use the thread pool by calling ThreadPool.QueueUserWorkItem from managed code (or `ICorThreadpool::CorQueueUserWorkItem` from unmanaged code) and passing a System.Threading.WaitCallback delegate representing the method that performs the task.

Another way to use the thread pool is to queue work items that are related to a wait operation by using the ThreadPool.RegisterWaitForSingleObject method and passing a System.Threading.WaitHandle that, when signaled or when timed out, calls the method represented by the System.Threading.WaitOrTimerCallback delegate. Thread pool threads are used to invoke callback methods.

For the examples, check the referenced API pages.

Skipping security checks

The thread pool also provides the ThreadPool.UnsafeQueueUserWorkItem and ThreadPool.UnsafeRegisterWaitForSingleObject methods. Use these methods only when you are certain that the caller's stack is irrelevant to any security checks performed during the execution of the queued task. ThreadPool.QueueUserWorkItem and ThreadPool.RegisterWaitForSingleObject both capture the caller's stack, which is merged into the stack of the thread pool thread when the thread begins to execute a task. If a security check is required, the entire stack must be checked. Although the check provides safety, it also has a performance cost.

When not to use thread pool threads

There are several scenarios in which it's appropriate to create and manage your own threads instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

See also

- System.Threading.ThreadPool
- System.Threading.Tasks.Task
- System.Threading.Tasks.Task<TResult>
- Task Parallel Library (TPL)
- How to: Return a Value from a Task
- Threading Objects and Features
- Threads and Threading
- Asynchronous File I/O
- Timers

Timers

1/23/2019 • 3 minutes to read • Edit Online

.NET provides two timers to use in a multithreaded environment:

- System.Threading.Timer, which executes a single callback method on a ThreadPool thread at regular intervals.
- System.Timers.Timer, which by default raises an event on a ThreadPool thread at regular intervals.

> **NOTE**
>
> Some .NET implementations may include additional timers:
>
> - System.Windows.Forms.Timer: a Windows Forms component that fires an event at regular intervals. The component has no user interface and is designed for use in a single-threaded environment.
> - System.Web.UI.Timer: an ASP.NET component that performs asynchronous or synchronous web page postbacks at a regular interval.
> - System.Windows.Threading.DispatcherTimer: a timer that is integrated into the Dispatcher queue which is processed at a specified interval of time and at a specified priority.

The System.Threading.Timer class

The System.Threading.Timer class enables you to continuously call a delegate at specified time intervals. You also can use this class to schedule a single call to a delegate in a specified time interval. The delegate is executed on a ThreadPool thread.

When you create a System.Threading.Timer object, you specify a TimerCallback delegate that defines the callback method, an optional state object that is passed to the callback, the amount of time to delay before the first invocation of the callback, and the time interval between callback invocations. To cancel a pending timer, call the Timer.Dispose method.

The following example creates a timer that calls the provided delegate for the first time after one second (1000 milliseconds) and then calls it every two seconds. The state object in the example is used to count how many times the delegate is called. The timer is stopped when the delegate has been called at least 10 times.

```
using namespace System;
using namespace System::Threading;

ref class TimerState
{
public:
    int counter;
};

ref class Example
{
private:
    static Timer^ timer;

public:
    static void TimerTask(Object^ state)
    {
        Console::WriteLine("{0:HH:mm:ss.fff}: starting a new callback.", DateTime::Now);

        TimerState^ timerState = dynamic_cast<TimerState^>(state);
        Interlocked::Increment(timerState->counter);
    }

    static void Main()
    {
        TimerCallback^ tcb = gcnew TimerCallback(&TimerTask);
        TimerState^ state = gcnew TimerState();
        state->counter = 0;
        timer = gcnew Timer(tcb, state, 1000, 2000);

        while (state->counter <= 10)
        {
            Thread::Sleep(1000);
        }

        timer->~Timer();
        Console::WriteLine("{0:HH:mm:ss.fff}: done.", DateTime::Now);
    }
};

int main()
{
    Example::Main();
}
```

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    private static Timer timer;

    static void Main(string[] args)
    {
        var timerState = new TimerState { Counter = 0 };

        timer = new Timer(
            callback: new TimerCallback(TimerTask),
            state: timerState,
            dueTime: 1000,
            period: 2000);

        while (timerState.Counter <= 10)
        {
            Task.Delay(1000).Wait();
        }

        timer.Dispose();
        Console.WriteLine($"{DateTime.Now:HH:mm:ss.fff}: done.");
    }

    private static void TimerTask(object timerState)
    {
        Console.WriteLine($"{DateTime.Now:HH:mm:ss.fff}: starting a new callback.");
        var state = timerState as TimerState;
        Interlocked.Increment(ref state.Counter);
    }

    class TimerState
    {
        public int Counter;
    }
}
```

```vb
Imports System.Threading

Module Program

    Private Timer As Timer

    Sub Main(args As String())

        Dim StateObj As New TimerState
        StateObj.Counter = 0

        Timer = New Timer(New TimerCallback(AddressOf TimerTask), StateObj, 1000, 2000)

        While StateObj.Counter <= 10
            Task.Delay(1000).Wait()
        End While

        Timer.Dispose()
        Console.WriteLine($"{DateTime.Now:HH:mm:ss.fff}: done.")
    End Sub

    Private Sub TimerTask(ByVal StateObj As Object)

        Console.WriteLine($"{DateTime.Now:HH:mm:ss.fff}: starting a new callback.")

        Dim State As TimerState = CType(StateObj, TimerState)
        Interlocked.Increment(State.Counter)
    End Sub

    Private Class TimerState
        Public Counter As Integer
    End Class
End Module
```

For more information and examples, see System.Threading.Timer.

The System.Timers.Timer class

Another timer that can be used in a multithreaded environment is System.Timers.Timer that by default raises an event on a ThreadPool thread.

When you create a System.Timers.Timer object, you may specify the time interval in which to raise an Elapsed event. Use the Enabled property to indicate if a timer should raise an Elapsed event. If you need an Elapsed event to be raised only once after the specified interval has elapsed, set the AutoReset to `false`. The default value of the AutoReset property is `true`, which means that an Elapsed event is raised regularly at the interval defined by the Interval property.

For more information and examples, see System.Timers.Timer.

See also

- System.Threading.Timer
- System.Timers.Timer
- Threading Objects and Features

Overview of synchronization primitives

8/21/2019 • 7 minutes to read • Edit Online

.NET provides a range of types that you can use to synchronize access to a shared resource or coordinate thread interaction.

> **IMPORTANT**
> Use the same synchronization primitive instance to protect access of a shared resource. If you use different synchronization primitive instances to protect the same resource, you'll circumvent the protection provided by a synchronization primitive.

WaitHandle class and lightweight synchronization types

Multiple .NET synchronization primitives derive from the System.Threading.WaitHandle class, which encapsulates a native operating system synchronization handle and uses a signaling mechanism for thread interaction. Those classes include:

- System.Threading.Mutex, which grants exclusive access to a shared resource. The state of a mutex is signaled if no thread owns it.
- System.Threading.Semaphore, which limits the number of threads that can access a shared resource or a pool of resources concurrently. The state of a semaphore is set to signaled when its count is greater than zero, and nonsignaled when its count is zero.
- System.Threading.EventWaitHandle, which represents a thread synchronization event and can be either in a signaled or unsignaled state.
- System.Threading.AutoResetEvent, which derives from EventWaitHandle and, when signaled, resets automatically to an unsignaled state after releasing a single waiting thread.
- System.Threading.ManualResetEvent, which derives from EventWaitHandle and, when signaled, stays in a signaled state until the Reset method is called.

In the .NET Framework, because WaitHandle derives from System.MarshalByRefObject, these types can be used to synchronize the activities of threads across application domain boundaries.

In the .NET Framework and .NET Core, some of these types can represent named system synchronization handles, which are visible throughout the operating system and can be used for the inter-process synchronization:

- Mutex (.NET Framework and .NET Core),
- Semaphore (.NET Framework and .NET Core on Windows),
- EventWaitHandle (.NET Framework and .NET Core on Windows).

For more information, see the WaitHandle API reference.

Lightweight synchronization types don't rely on underlying operating system handles and typically provide better performance. However, they cannot be used for the inter-process synchronization. Use those types for thread synchronization within one application.

Some of those types are alternatives to the types derived from WaitHandle. For example, SemaphoreSlim is a lightweight alternative to Semaphore.

Synchronization of access to a shared resource

.NET provides a range of synchronization primitives to control access to a shared resource by multiple threads.

Monitor class

The System.Threading.Monitor class grants mutually exclusive access to a shared resource by acquiring or releasing a lock on the object that identifies the resource. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and the Monitor.Enter method waits until the lock is released. The Enter method acquires a released lock. You can also use the Monitor.TryEnter method to specify the amount of time during which a thread attempts to acquire a lock. Because the Monitor class has thread affinity, the thread that acquired a lock must release the lock by calling the Monitor.Exit method.

You can coordinate the interaction of threads that acquire a lock on the same object by using the Monitor.Wait, Monitor.Pulse, and Monitor.PulseAll methods.

For more information, see the Monitor API reference.

> **NOTE**
>
> Use the lock statement in C# and the SyncLock statement in Visual Basic to synchronize access to a shared resource instead of using the Monitor class directly. Those statements are implemented by using the Enter and Exit methods and a `try…finally` block to ensure that the acquired lock is always released.

Mutex class

The System.Threading.Mutex class, like Monitor, grants exclusive access to a shared resource. Use one of the Mutex.WaitOne method overloads to request the ownership of a mutex. Like Monitor, Mutex has thread affinity and the thread that acquired a mutex must release it by calling the Mutex.ReleaseMutex method.

Unlike Monitor, the Mutex class can be used for inter-process synchronization. To do that, use a named mutex, which is visible throughout the operating system. To create a named mutex instance, use a Mutex constructor that specifies a name. You also can call the Mutex.OpenExisting method to open an existing named system mutex.

For more information, see the Mutexes article and the Mutex API reference.

SpinLock structure

The System.Threading.SpinLock structure, like Monitor, grants exclusive access to a shared resource based on the availability of a lock. When SpinLock attempts to acquire a lock that is unavailable, it waits in a loop, repeatedly checking until the lock becomes available.

For more information about the benefits and drawbacks of using spin lock, see the SpinLock article and the SpinLock API reference.

ReaderWriterLockSlim class

The System.Threading.ReaderWriterLockSlim class grants exclusive access to a shared resource for writing and allows multiple threads to access the resource simultaneously for reading. You might want to use ReaderWriterLockSlim to synchronize access to a shared data structure that supports thread-safe read operations, but requires exclusive access to perform write operation. When a thread requests exclusive access (for example, by calling the ReaderWriterLockSlim.EnterWriteLock method), subsequent reader and writer requests block until all existing readers have exited the lock, and the writer has entered and exited the lock.

For more information, see the ReaderWriterLockSlim API reference.

Semaphore and SemaphoreSlim classes

The System.Threading.Semaphore and System.Threading.SemaphoreSlim classes limit the number of threads that can access a shared resource or a pool of resources concurrently. Additional threads that request the resource wait until any thread releases the semaphore. Because the semaphore doesn't have thread affinity, a thread can acquire the semaphore and another one can release it.

SemaphoreSlim is a lightweight alternative to Semaphore and can be used only for synchronization within a single process boundary.

On Windows, you can use Semaphore for the inter-process synchronization. To do that, create a Semaphore instance that represents a named system semaphore by using one of the Semaphore constructors that specifies a name or the Semaphore.OpenExisting method. SemaphoreSlim doesn't support named system semaphores.

For more information, see the Semaphore and SemaphoreSlim article and the Semaphore or SemaphoreSlim API reference.

Thread interaction, or signaling

Thread interaction (or thread signaling) means that a thread must wait for notification, or a signal, from one or more threads in order to proceed. For example, if thread A calls the Thread.Join method of thread B, thread A is blocked until thread B completes. The synchronization primitives described in the preceding section provide a different mechanism for signaling: by releasing a lock, a thread notifies another thread that it can proceed by acquiring the lock.

This section describes additional signaling constructs provided by .NET.

EventWaitHandle, AutoResetEvent, ManualResetEvent, and ManualResetEventSlim classes

The System.Threading.EventWaitHandle class represents a thread synchronization event.

A synchronization event can be either in an unsignaled or signaled state. When the state of an event is unsignaled, a thread that calls the event's WaitOne overload is blocked until an event is signaled. The EventWaitHandle.Set method sets the state of an event to signaled.

The behavior of an EventWaitHandle that has been signaled depends on its reset mode:

- An EventWaitHandle created with the EventResetMode.AutoReset flag resets automatically after releasing a single waiting thread. It's like a turnstile that allows only one thread through each time it's signaled. The System.Threading.AutoResetEvent class, which derives from EventWaitHandle, represents that behavior.
- An EventWaitHandle created with the EventResetMode.ManualReset flag remains signaled until its Reset method is called. It's like a gate that is closed until signaled and then stays open until someone closes it. The System.Threading.ManualResetEvent class, which derives from EventWaitHandle, represents that behavior. The System.Threading.ManualResetEventSlim class is a lightweight alternative to ManualResetEvent.

On Windows, you can use EventWaitHandle for the inter-process synchronization. To do that, create a EventWaitHandle instance that represents a named system synchronization event by using one of the EventWaitHandle constructors that specifies a name or the EventWaitHandle.OpenExisting method.

For more information, see the EventWaitHandle article. For the API reference, see EventWaitHandle, AutoResetEvent, ManualResetEvent, and ManualResetEventSlim.

CountdownEvent class

The System.Threading.CountdownEvent class represents an event that becomes set when its count is zero. While CountdownEvent.CurrentCount is greater than zero, a thread that calls CountdownEvent.Wait is blocked. Call CountdownEvent.Signal to decrement an event's count.

In contrast to ManualResetEvent or ManualResetEventSlim, which you can use to unblock multiple threads with a signal from one thread, you can use CountdownEvent to unblock one or more threads with signals from multiple threads.

For more information, see the CountdownEvent article and the CountdownEvent API reference.

Barrier class

The System.Threading.Barrier class represents a thread execution barrier. A thread that calls the Barrier.SignalAndWait method signals that it reached the barrier and waits until other participant threads reach the barrier. When all participant threads reach the barrier, they proceed and the barrier is reset and can be used again.

You might use Barrier when one or more threads require the results of other threads before proceeding to the next computation phase.

For more information, see the Barrier article and the Barrier API reference.

Interlocked class

The System.Threading.Interlocked class provides static methods that perform simple atomic operations on a variable. Those atomic operations include addition, increment and decrement, exchange and conditional exchange that depends on a comparison, and read operation of a 64-bit integer value.

For more information, see the Interlocked API reference.

SpinWait structure

The System.Threading.SpinWait structure provides support for spin-based waiting. You might want to use it when a thread has to wait for an event to be signaled or a condition to be met, but when the actual wait time is expected to be less than the waiting time required by using a wait handle or by otherwise blocking the thread. By using SpinWait, you can specify a short period of time to spin while waiting, and then yield (for example, by waiting or sleeping) only if the condition was not met in the specified time.

For more information, see the SpinWait article and the SpinWait API reference.

See also

- System.Collections.Concurrent
- Thread-safe collections
- Threading objects and features

EventWaitHandle

8/22/2019 • 4 minutes to read • Edit Online

The EventWaitHandle class allows threads to communicate with each other by signaling and by waiting for signals. Event wait handles (also referred to simply as events) are wait handles that can be signaled in order to release one or more waiting threads. After it is signaled, an event wait handle is reset either manually or automatically. The EventWaitHandle class can represent either a local event wait handle (local event) or a named system event wait handle (named event or system event, visible to all processes).

> **NOTE**
>
> Event wait handles are not .NET events. There are no delegates or event handlers involved. The word "event" is used to describe them because they have traditionally been referred to as operating-system events, and because the act of signaling the wait handle indicates to waiting threads that an event has occurred.

Both local and named event wait handles use system synchronization objects, which are protected by SafeWaitHandle wrappers to ensure that the resources are released. You can use the Dispose method to free the resources immediately when you have finished using the object.

Event Wait Handles That Reset Automatically

You create an automatic reset event by specifying EventResetMode.AutoReset when you create the EventWaitHandle object. As its name implies, this synchronization event resets automatically when signaled, after releasing a single waiting thread. Signal the event by calling its Set method.

Automatic reset events are usually used to provide exclusive access to a resource for a single thread at a time. A thread requests the resource by calling the WaitOne method. If no other thread is holding the wait handle, the method returns `true` and the calling thread has control of the resource.

> **IMPORTANT**
>
> As with all synchronization mechanisms, you must ensure that all code paths wait on the appropriate wait handle before accessing a protected resource. Thread synchronization is cooperative.

If an automatic reset event is signaled when no threads are waiting, it remains signaled until a thread attempts to wait on it. The event releases the thread and immediately resets, blocking subsequent threads.

Event Wait Handles That Reset Manually

You create a manual reset event by specifying EventResetMode.ManualReset when you create the EventWaitHandle object. As its name implies, this synchronization event must be reset manually after it has been signaled. Until it is reset, by calling its Reset method, threads that wait on the event handle proceed immediately without blocking.

A manual reset event acts like the gate of a corral. When the event is not signaled, threads that wait on it block, like horses in a corral. When the event is signaled, by calling its Set method, all waiting threads are free to proceed. The event remains signaled until its Reset method is called. This makes the manual reset event an ideal way to hold up threads that need to wait until one thread finishes a task.

Like horses leaving a corral, it takes time for the released threads to be scheduled by the operating system and to

resume execution. If the Reset method is called before all the threads have resumed execution, the remaining threads once again block. Which threads resume and which threads block depends on random factors like the load on the system, the number of threads waiting for the scheduler, and so on. This is not a problem if the thread that signals the event ends after signaling, which is the most common usage pattern. If you want the thread that signaled the event to begin a new task after all the waiting threads have resumed, you must block it until all the waiting threads have resumed. Otherwise, you have a race condition, and the behavior of your code is unpredictable.

Features Common to Automatic and Manual Events

Typically, one or more threads block on an EventWaitHandle until an unblocked thread calls the Set method, which releases one of the waiting threads (in the case of automatic reset events) or all of them (in the case of manual reset events). A thread can signal an EventWaitHandle and then block on it, as an atomic operation, by calling the static WaitHandle.SignalAndWait method.

EventWaitHandle objects can be used with the static WaitHandle.WaitAll and WaitHandle.WaitAny methods. Because the EventWaitHandle and Mutex classes both derive from WaitHandle, you can use both classes with these methods.

Named Events

The Windows operating system allows event wait handles to have names. A named event is system wide. That is, once the named event is created, it is visible to all threads in all processes. Thus, named events can be used to synchronize the activities of processes as well as threads.

You can create an EventWaitHandle object that represents a named system event by using one of the constructors that specifies an event name.

> **NOTE**
>
> Because named events are system wide, it is possible to have multiple EventWaitHandle objects that represent the same named event. Each time you call a constructor, or the OpenExisting method, a new EventWaitHandle object is created. Specifying the same name repeatedly creates multiple objects that represent the same named event.

Caution is advised in using named events. Because they are system wide, another process that uses the same name can block your threads unexpectedly. Malicious code executing on the same computer could use this as the basis of a denial-of-service attack.

Use access control security to protect an EventWaitHandle object that represents a named event, preferably by using a constructor that specifies an EventWaitHandleSecurity object. You can also apply access control security using the SetAccessControl method, but this leaves a window of vulnerability between the time the event wait handle is created and the time it is protected. Protecting events with access control security helps prevent malicious attacks, but it does not solve the problem of unintentional name collisions.

> **NOTE**
>
> Unlike the EventWaitHandle class, the derived classes AutoResetEvent and ManualResetEvent can represent only local wait handles. They cannot represent named system events.

See also

- EventWaitHandle
- WaitHandle
- AutoResetEvent

- ManualResetEvent

CountdownEvent

8/22/2019 • 5 minutes to read • Edit Online

System.Threading.CountdownEvent is a synchronization primitive that unblocks its waiting threads after it has been signaled a certain number of times. CountdownEvent is designed for scenarios in which you would otherwise have to use a ManualResetEvent or ManualResetEventSlim and manually decrement a variable before signaling the event. For example, in a fork/join scenario, you can just create a CountdownEvent that has a signal count of 5, and then start five work items on the thread pool and have each work item call Signal when it completes. Each call to Signal decrements the signal count by 1. On the main thread, the call to Wait will block until the signal count is zero.

> **NOTE**
> For code that does not have to interact with legacy .NET Framework synchronization APIs, consider using System.Threading.Tasks.Task objects or the Invoke method for an even easier approach to expressing fork-join parallelism.

CountdownEvent has these additional features:

- The wait operation can be canceled by using cancellation tokens.
- Its signal count can be incremented after the instance is created.
- Instances can be reused after Wait has returned by calling the Reset method.
- Instances expose a WaitHandle for integration with other .NET Framework synchronization APIs such as WaitAll.

Basic Usage

The following example demonstrates how to use a CountdownEvent with ThreadPool work items.

```csharp
IEnumerable<Data> source = GetData();
using (CountdownEvent e = new CountdownEvent(1))
{
    // fork work:
    foreach (Data element in source)
    {
        // Dynamically increment signal count.
        e.AddCount();
        ThreadPool.QueueUserWorkItem(delegate(object state)
         {
             try
             {
                 ProcessData(state);
             }
             finally
             {
                 e.Signal();
             }
         },
         element);
    }
    e.Signal();

    // The first element could be run on this thread.

    // Join with work.
    e.Wait();
}
// .,.
```

```vb
Dim source As IEnumerable(Of Data) = GetData()
Dim e = New CountdownEvent(1)

' Fork work:
For Each element As Data In source
    ' Dynamically increment signal count.
    e.AddCount()

    ThreadPool.QueueUserWorkItem(Sub(state)
                                     Try
                                         ProcessData(state)
                                     Finally
                                         e.Signal()
                                     End Try
                                 End Sub,
                                  element)
Next
' Decrement the signal count by the one we added
' in the constructor.
e.Signal()

' The first element could also be run on this thread.
' ProcessData(New Data(0))

' Join with work:
e.Wait()
```

CountdownEvent With Cancellation

The following example shows how to cancel the wait operation on CountdownEvent by using a cancellation token. The basic pattern follows the model for unified cancellation, which is introduced in .NET Framework 4. For more information, see Cancellation in Managed Threads.

```
class CancelableCountdownEvent
{
    class Data
    {
        public int Num { get; set; }
        public Data(int i) { Num = i; }
        public Data() { }
    }

    class DataWithToken
    {
        public CancellationToken Token { get; set; }
        public Data Data { get; private set; }
        public DataWithToken(Data data, CancellationToken ct)
        {
            this.Data = data;
            this.Token = ct;
        }
    }
    static IEnumerable<Data> GetData()
    {
        return new List<Data>() { new Data(1), new Data(2), new Data(3), new Data(4), new Data(5) };
    }
    static void ProcessData(object obj)
    {
        DataWithToken dataWithToken = (DataWithToken)obj;
        if (dataWithToken.Token.IsCancellationRequested)
        {
            Console.WriteLine("Canceled before starting {0}", dataWithToken.Data.Num);
            return;
        }

        for (int i = 0; i < 10000; i++)
        {
            if (dataWithToken.Token.IsCancellationRequested)
            {
                Console.WriteLine("Cancelling while executing {0}", dataWithToken.Data.Num);
                return;
            }
            // Increase this value to slow down the program.
            Thread.SpinWait(100000);
        }
        Console.WriteLine("Processed {0}", dataWithToken.Data.Num);
    }

    static void Main(string[] args)
    {
        EventWithCancel();

        Console.WriteLine("Press enter to exit.");
        Console.ReadLine();
    }

    static void EventWithCancel()
    {
        IEnumerable<Data> source = GetData();
        CancellationTokenSource cts = new CancellationTokenSource();

        //Enable cancellation request from a simple UI thread.
        Task.Factory.StartNew(() =>
             {
                 if (Console.ReadKey().KeyChar == 'c')
                     cts.Cancel();
             });

        // Event must have a count of at least 1
        CountdownEvent e = new CountdownEvent(1);
```

```
        // fork work:
        foreach (Data element in source)
        {
            DataWithToken item = new DataWithToken(element, cts.Token);
            // Dynamically increment signal count.
            e.AddCount();
            ThreadPool.QueueUserWorkItem(delegate(object state)
             {
                 ProcessData(state);
                 if (!cts.Token.IsCancellationRequested)
                     e.Signal();
             },
             item);
        }
        // Decrement the signal count by the one we added
        // in the constructor.
        e.Signal();

        // The first element could be run on this thread.

        // Join with work or catch cancellation.
        try
        {
            e.Wait(cts.Token);
        }
        catch (OperationCanceledException oce)
        {
            if (oce.CancellationToken == cts.Token)
            {
                Console.WriteLine("User canceled.");
            }
            else throw; //We don't know who canceled us!
        }
        finally {
            e.Dispose();
            cts.Dispose();
        }
        //...
    } //end method
} //end class
```

```
Option Strict On
Option Explicit On

Imports System.Collections
Imports System.Collections.Generic
Imports System.Linq
Imports System.Threading
Imports System.Threading.Tasks

Module CancelEventWait

    Class Data
        Public Num As Integer
        Public Sub New(ByVal i As Integer)
            Num = i
        End Sub
        Public Sub New()

        End Sub
    End Class

    Class DataWithToken
        Public Token As CancellationToken
        Public _data As Data
        Public Sub New(ByVal d As Data, ByVal ct As CancellationToken)
```

```
        Me._data = d
        Me.Token = ct
    End Sub
End Class

Class Program
    Shared Function GetData() As IEnumerable(Of Data)
        Dim nums = New List(Of Data)
        For i As Integer = 1 To 5
            nums.Add(New Data(i))
        Next
        Return nums
    End Function

    Shared Sub ProcessData(ByVal obj As Object)
        Dim dataItem As DataWithToken = CType(obj, DataWithToken)
        If dataItem.Token.IsCancellationRequested = True Then
            Console.WriteLine("Canceled before starting {0}", dataItem._data.Num)
            Exit Sub
        End If

        ' Increase this value to slow down the program.
        For i As Integer = 0 To 10000

            If dataItem.Token.IsCancellationRequested = True Then
                Console.WriteLine("Cancelling while executing {0}", dataItem._data.Num)
                Exit Sub
            End If
            Thread.SpinWait(100000)
        Next
        Console.WriteLine("Processed {0}", dataItem._data.Num)


    End Sub

    Shared Sub Main()
        DoEventWithCancel()
        Console.WriteLine("Press the enter key to exit.")
        Console.ReadLine()
    End Sub

    Shared Sub DoEventWithCancel()
        Dim source As IEnumerable(Of Data) = GetData()
        Dim cts As CancellationTokenSource = New CancellationTokenSource()

        ' Enable cancellation request from a simple UI thread.
        Task.Factory.StartNew(Sub()
                                  If Console.ReadKey().KeyChar = "c"c Then
                                      cts.Cancel()
                                  End If
                              End Sub)

        ' Must have a count of at least 1 or else it is signaled.
        Dim e As CountdownEvent = New CountdownEvent(1)

        For Each element As Data In source
            Dim item As DataWithToken = New DataWithToken(element, cts.Token)

            ' Dynamically increment signal count.
            e.AddCount()

            ThreadPool.QueueUserWorkItem(Sub(state)
                                             ProcessData(state)
                                             If cts.Token.IsCancellationRequested = False Then
                                                 e.Signal()
                                             End If
                                         End Sub,
                                        item)
        Next
```

```
            ' Decrement the signal count by the one we added
            ' in the constructor.
            e.Signal()
            ' The first element could be run on this thread.
            ' ProcessData(source(0))

            ' Join with work or catch cancellation exception
            Try
                e.Wait(cts.Token)
            Catch ex As OperationCanceledException
                If ex.CancellationToken = cts.Token Then
                    Console.WriteLine("User canceled.")
                Else : Throw ' we don't know who canceled us.

                End If
            Finally
                e.Dispose()
                cts.Dispose()
            End Try
        End Sub
    End Class
End Module
```

Note that the wait operation does not cancel the threads that are signaling it. Typically, cancellation is applied to a logical operation, and that can include waiting on the event as well as all the work items that the wait is synchronizing. In this example, each work item is passed a copy of the same cancellation token so that it can respond to the cancellation request.

See also

- System.Threading.Semaphore

Mutexes

8/22/2019 • 2 minutes to read • Edit Online

You can use a Mutex object to provide exclusive access to a resource. The Mutex class uses more system resources than the Monitor class, but it can be marshaled across application domain boundaries, it can be used with multiple waits, and it can be used to synchronize threads in different processes. For a comparison of managed synchronization mechanisms, see Overview of Synchronization Primitives.

For code examples, see the reference documentation for the Mutex constructors.

Using Mutexes

A thread calls the WaitOne method of a mutex to request ownership. The call blocks until the mutex is available, or until the optional timeout interval elapses. The state of a mutex is signaled if no thread owns it.

A thread releases a mutex by calling its ReleaseMutex method. Mutexes have thread affinity; that is, the mutex can be released only by the thread that owns it. If a thread releases a mutex it does not own, an ApplicationException is thrown in the thread.

Because the Mutex class derives from WaitHandle, you can also call the static WaitAll or WaitAny methods of WaitHandle to request ownership of a Mutex in combination with other wait handles.

If a thread owns a Mutex, that thread can specify the same Mutex in repeated wait-request calls without blocking its execution; however, it must release the Mutex as many times to release ownership.

Abandoned Mutexes

If a thread terminates without releasing a Mutex, the mutex is said to be abandoned. This often indicates a serious programming error because the resource the mutex is protecting might be left in an inconsistent state. In the .NET Framework version 2.0, an AbandonedMutexException is thrown in the next thread that acquires the mutex.

> **NOTE**
>
> In the .NET Framework versions 1.0 and 1.1, an abandoned Mutex is set to the signaled state and the next waiting thread gets ownership. If no thread is waiting, the Mutex remains in a signaled state. No exception is thrown.

In the case of a system-wide mutex, an abandoned mutex might indicate that an application has been terminated abruptly (for example, by using Windows Task Manager).

Local and System Mutexes

Mutexes are of two types: local mutexes and named system mutexes. If you create a Mutex object using a constructor that accepts a name, it is associated with an operating-system object of that name. Named system mutexes are visible throughout the operating system and can be used to synchronize the activities of processes. You can create multiple Mutex objects that represent the same named system mutex, and you can use the OpenExisting method to open an existing named system mutex.

A local mutex exists only within your process. It can be used by any thread in your process that has a reference to the local Mutex object. Each Mutex object is a separate local mutex.

Access Control Security for System Mutexes

The .NET Framework version 2.0 provides the ability to query and set Windows access control security for named system objects. Protecting system mutexes from the moment of creation is recommended because system objects are global and therefore can be locked by code other than your own.

For information on access control security for mutexes, see the MutexSecurity and MutexAccessRule classes, the MutexRights enumeration, the GetAccessControl, SetAccessControl, and OpenExisting methods of the Mutex class, and the Mutex(Boolean, String, Boolean, MutexSecurity) constructor.

See also

- System.Threading.Mutex
- Mutex.Mutex
- System.Security.AccessControl.MutexSecurity
- System.Security.AccessControl.MutexAccessRule
- System.Threading.Monitor
- Threading objects and features
- Threads and threading
- Threading

Semaphore and SemaphoreSlim

8/22/2019 • 3 minutes to read • Edit Online

The System.Threading.Semaphore class represents a named (systemwide) or local semaphore. It is a thin wrapper around the Win32 semaphore object. Win32 semaphores are counting semaphores, which can be used to control access to a pool of resources.

The SemaphoreSlim class represents a lightweight, fast semaphore that can be used for waiting within a single process when wait times are expected to be very short. SemaphoreSlim relies as much as possible on synchronization primitives provided by the common language runtime (CLR). However, it also provides lazily initialized, kernel-based wait handles as necessary to support waiting on multiple semaphores. SemaphoreSlim also supports the use of cancellation tokens, but it does not support named semaphores or the use of a wait handle for synchronization.

Managing a Limited Resource

Threads enter the semaphore by calling the WaitOne method, which is inherited from the WaitHandle class, in the case of a System.Threading.Semaphore object, or the SemaphoreSlim.Wait or SemaphoreSlim.WaitAsync method, in the case of a SemaphoreSlim object. When the call returns, the count on the semaphore is decremented. When a thread requests entry and the count is zero, the thread blocks. As threads release the semaphore by calling the Semaphore.Release or SemaphoreSlim.Release method, blocked threads are allowed to enter. There is no guaranteed order, such as first-in, first-out (FIFO) or last-in, first-out (LIFO), for blocked threads to enter the semaphore.

A thread can enter the semaphore multiple times by calling the System.Threading.Semaphore object's WaitOne method or the SemaphoreSlim object's Wait method repeatedly. To release the semaphore, the thread can either call the Semaphore.Release() or SemaphoreSlim.Release() method overload the same number of times, or call the Semaphore.Release(Int32) or SemaphoreSlim.Release(Int32) method overload and specify the number of entries to be released.

Semaphores and Thread Identity

The two semaphore types do not enforce thread identity on calls to the WaitOne, Wait, Release, and SemaphoreSlim.Release methods. For example, a common usage scenario for semaphores involves a producer thread and a consumer thread, with one thread always incrementing the semaphore count and the other always decrementing it.

It is the programmer's responsibility to ensure that a thread does not release the semaphore too many times. For example, suppose a semaphore has a maximum count of two, and that thread A and thread B both enter the semaphore. If a programming error in thread B causes it to call `Release` twice, both calls succeed. The count on the semaphore is full, and when thread A eventually calls `Release`, a SemaphoreFullException is thrown.

Named Semaphores

The Windows operating system allows semaphores to have names. A named semaphore is system wide. That is, once the named semaphore is created, it is visible to all threads in all processes. Thus, named semaphore can be used to synchronize the activities of processes as well as threads.

You can create a Semaphore object that represents a named system semaphore by using one of the constructors that specifies a name.

NOTE

Because named semaphores are system wide, it is possible to have multiple Semaphore objects that represent the same named semaphore. Each time you call a constructor or the Semaphore.OpenExisting method, a new Semaphore object is created. Specifying the same name repeatedly creates multiple objects that represent the same named semaphore.

Be careful when you use named semaphores. Because they are system wide, another process that uses the same name can enter your semaphore unexpectedly. Malicious code executing on the same computer could use this as the basis of a denial-of-service attack.

Use access control security to protect a Semaphore object that represents a named semaphore, preferably by using a constructor that specifies a System.Security.AccessControl.SemaphoreSecurity object. You can also apply access control security using the Semaphore.SetAccessControl method, but this leaves a window of vulnerability between the time the semaphore is created and the time it is protected. Protecting semaphores with access control security helps prevent malicious attacks, but does not solve the problem of unintentional name collisions.

See also

- Semaphore
- SemaphoreSlim
- Threading Objects and Features

Barrier

4/28/2019 • 3 minutes to read • Edit Online

A System.Threading.Barrier is a synchronization primitive that enables multiple threads (known as *participants*) to work concurrently on an algorithm in phases. Each participant executes until it reaches the barrier point in the code. The barrier represents the end of one phase of work. When a participant reaches the barrier, it blocks until all participants have reached the same barrier. After all participants have reached the barrier, you can optionally invoke a post-phase action. This post-phase action can be used to perform actions by a single thread while all other threads are still blocked. After the action has been executed, the participants are all unblocked.

The following code snippet shows a basic barrier pattern.

```
// Create the Barrier object, and supply a post-phase delegate
// to be invoked at the end of each phase.
Barrier barrier = new Barrier(2, (bar) =>
    {
        // Examine results from all threads, determine
        // whether to continue, create inputs for next phase, etc.
        if (someCondition)
            success = true;
    });


// Define the work that each thread will perform. (Threads do not
// have to all execute the same method.)
void CrunchNumbers(int partitionNum)
{
    // Up to System.Int64.MaxValue phases are supported. We assume
    // in this code that the problem will be solved before that.
    while (success == false)
    {
        // Begin phase:
        // Process data here on each thread, and optionally
        // store results, for example:
        results[partitionNum] = ProcessData(data[partitionNum]);

        // End phase:
        // After all threads arrive,post-phase delegate
        // is invoked, then threads are unblocked. Overloads
        // accept a timeout value and/or CancellationToken.
        barrier.SignalAndWait();
    }
}

// Perform n tasks to run in in parallel. For simplicity
// all threads execute the same method in this example.
static void Main()
{
    var app = new BarrierDemo();
    Thread t1 = new Thread(() => app.CrunchNumbers(0));
    Thread t2 = new Thread(() => app.CrunchNumbers(1));
    t1.Start();
    t2.Start();

}
```

```
' Create the Barrier object, and supply a post-phase delegate
' to be invoked at the end of each phase.
Dim barrier = New Barrier(2, Sub(bar)
                                 ' Examine results from all threads, determine
                                 ' whether to continue, create inputs for next phase, etc.
                                 If (someCondition) Then
                                     success = True
                                 End If
                             End Sub)



' Define the work that each thread will perform. (Threads do not
' have to all execute the same method.)
Sub CrunchNumbers(ByVal partitionNum As Integer)

    ' Up to System.Int64.MaxValue phases are supported. We assume
    ' in this code that the problem will be solved before that.
    While (success = False)

        ' Begin phase:
        ' Process data here on each thread, and optionally
        ' store results, for example:
        results(partitionNum) = ProcessData(myData(partitionNum))

        ' End phase:
        ' After all threads arrive,post-phase delegate
        ' is invoked, then threads are unblocked. Overloads
        ' accept a timeout value and/or CancellationToken.
        barrier.SignalAndWait()
    End While
End Sub

' Perform n tasks to run in in parallel. For simplicity
' all threads execute the same method in this example.
Shared Sub Main()

    Dim app = New BarrierDemo()
    Dim t1 = New Thread(Sub() app.CrunchNumbers(0))
    Dim t2 = New Thread(Sub() app.CrunchNumbers(1))
    t1.Start()
    t2.Start()
End Sub
```

For a complete example, see How to: synchronize concurrent operations with a Barrier.

Adding and removing participants

When you create a Barrier instance, specify the number of participants. You can also add or remove participants dynamically at any time. For example, if one participant solves its part of the problem, you can store the result, stop execution on that thread, and call Barrier.RemoveParticipant to decrement the number of participants in the barrier. When you add a participant by calling Barrier.AddParticipant, the return value specifies the current phase number, which may be useful in order to initialize the work of the new participant.

Broken barriers

Deadlocks can occur if one participant fails to reach the barrier. To avoid these deadlocks, use the overloads of the Barrier.SignalAndWait method to specify a time-out period and a cancellation token. These overloads return a Boolean value that every participant can check before it continues to the next phase.

Post-phase exceptions

If the post-phase delegate throws an exception, it is wrapped in a BarrierPostPhaseException object which is then propagated to all participants.

Barrier versus ContinueWhenAll

Barriers are especially useful when the threads are performing multiple phases in loops. If your code requires only one or two phases of work, consider whether to use System.Threading.Tasks.Task objects with any kind of implicit join, including:

- TaskFactory.ContinueWhenAll
- Parallel.Invoke
- Parallel.ForEach
- Parallel.For

For more information, see Chaining Tasks by Using Continuation Tasks.

See also

- Threading objects and features
- How to: synchronize concurrent operations with a Barrier

How to: Synchronize Concurrent Operations with a Barrier

9/6/2018 • 3 minutes to read • Edit Online

The following example shows how to synchronize concurrent tasks with a Barrier.

Example

The purpose of the following program is to count how many iterations (or phases) are required for two threads to each find their half of the solution on the same phase by using a randomizing algorithm to reshuffle the words. After each thread has shuffled its words, the barrier post-phase operation compares the two results to see if the complete sentence has been rendered in correct word order.

```
//#define TRACE
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace BarrierSimple
{
    class Program
    {
        static string[] words1 = new string[] { "brown",  "jumps", "the", "fox", "quick"};
        static string[] words2 = new string[] { "dog", "lazy","the","over"};
        static string solution = "the quick brown fox jumps over the lazy dog.";

        static bool success = false;
        static Barrier barrier = new Barrier(2, (b) =>
        {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < words1.Length; i++)
            {
                sb.Append(words1[i]);
                sb.Append(" ");
            }
            for (int i = 0; i < words2.Length; i++)
            {
                sb.Append(words2[i]);

                if(i < words2.Length - 1)
                    sb.Append(" ");
            }
            sb.Append(".");
#if TRACE
            System.Diagnostics.Trace.WriteLine(sb.ToString());
#endif
            Console.CursorLeft = 0;
            Console.Write("Current phase: {0}", barrier.CurrentPhaseNumber);
            if (String.CompareOrdinal(solution, sb.ToString()) == 0)
            {
                success = true;
                Console.WriteLine("\r\nThe solution was found in {0} attempts", barrier.CurrentPhaseNumber);
            }
        });

        static void Main(string[] args)
```

```
        {

            Thread t1 = new Thread(() => Solve(words1));
            Thread t2 = new Thread(() => Solve(words2));
            t1.Start();
            t2.Start();

            // Keep the console window open.
            Console.ReadLine();
        }

        // Use Knuth-Fisher-Yates shuffle to randomly reorder each array.
        // For simplicity, we require that both wordArrays be solved in the same phase.
        // Success of right or left side only is not stored and does not count.
        static void Solve(string[] wordArray)
        {
            while(success == false)
            {
                Random random = new Random();
                for (int i = wordArray.Length - 1; i > 0; i--)
                {
                    int swapIndex = random.Next(i + 1);
                    string temp = wordArray[i];
                    wordArray[i] = wordArray[swapIndex];
                    wordArray[swapIndex] = temp;
                }

                // We need to stop here to examine results
                // of all thread activity. This is done in the post-phase
                // delegate that is defined in the Barrier constructor.
                barrier.SignalAndWait();
            }
        }
    }
}
```

```
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports System.Threading
Imports System.Threading.Tasks


    Class Program
        Shared words1() = New String() {"brown", "jumps", "the", "fox", "quick"}
        Shared words2() = New String() {"dog", "lazy", "the", "over"}
        Shared solution = "the quick brown fox jumps over the lazy dog."

        Shared success = False
        Shared barrier = New Barrier(2, Sub(b)
                                            Dim sb = New StringBuilder()
                                            For i As Integer = 0 To words1.Length - 1
                                                sb.Append(words1(i))
                                                sb.Append(" ")
                                            Next
                                            For i As Integer = 0 To words2.Length - 1

                                                sb.Append(words2(i))

                                                If (i < words2.Length - 1) Then
                                                    sb.Append(" ")
                                                End If
                                            Next
                                            sb.Append(".")
                                            System.Diagnostics.Trace.WriteLine(sb.ToString())

                                            Console.CursorLeft = 0
```

```
                                               Console.Write("Current phase: {0}", barrier.CurrentPhaseNumber)
                                               If (String.CompareOrdinal(solution, sb.ToString()) = 0) Then
                                                   success = True
                                                   Console.WriteLine()
                                                   Console.WriteLine("The solution was found in {0} attempts",
barrier.CurrentPhaseNumber)
                                               End If
                                           End Sub)

        Shared Sub Main()
            Dim t1 = New Thread(Sub() Solve(words1))
            Dim t2 = New Thread(Sub() Solve(words2))
            t1.Start()
            t2.Start()

            ' Keep the console window open.
            Console.ReadLine()
        End Sub

        ' Use Knuth-Fisher-Yates shuffle to randomly reorder each array.
        ' For simplicity, we require that both wordArrays be solved in the same phase.
        ' Success of right or left side only is not stored and does not count.
        Shared Sub Solve(ByVal wordArray As String())
            While success = False
                Dim rand = New Random()
                For i As Integer = 0 To wordArray.Length - 1
                    Dim swapIndex As Integer = rand.Next(i + 1)
                    Dim temp As String = wordArray(i)
                    wordArray(i) = wordArray(swapIndex)
                    wordArray(swapIndex) = temp
                Next

                ' We need to stop here to examine results
                ' of all thread activity. This is done in the post-phase
                ' delegate that is defined in the Barrier constructor.
                barrier.SignalAndWait()
            End While
        End Sub
    End Class
```

A Barrier is an object that prevents individual tasks in a parallel operation from continuing until all tasks reach the barrier. It is useful when a parallel operation occurs in phases, and each phase requires synchronization between tasks. In this example, there are two phases to the operation. In the first phase, each task fills its section of the buffer with data. When each task finishes filling its section, the task signals the barrier that it is ready to continue, and then waits. When all tasks have signaled the barrier, they are unblocked and the second phase starts. The barrier is necessary because the second phase requires that each task have access to all the data that has been generated to this point. Without the barrier, the first tasks to complete might try to read from buffers that have not been filled in yet by other tasks. You can synchronize any number of phases in this manner.

See also

- Data Structures for Parallel Programming

SpinLock

9/6/2018 • 2 minutes to read • Edit Online

The SpinLock structure is a low-level, mutual-exclusion synchronization primitive that spins while it waits to acquire a lock. On multicore computers, when wait times are expected to be short and when contention is minimal, SpinLock can perform better than other kinds of locks. However, we recommend that you use SpinLock only when you determine by profiling that the System.Threading.Monitor method or the Interlocked methods are significantly slowing the performance of your program.

SpinLock may yield the time slice of the thread even if it has not yet acquired the lock. It does this to avoid thread-priority inversion, and to enable the garbage collector to make progress. When you use a SpinLock, ensure that no thread can hold the lock for more than a very brief time span, and that no thread can block while it holds the lock.

Because SpinLock is a value type, you must explicitly pass it by reference if you intend the two copies to refer to the same lock.

For more information about how to use this type, see System.Threading.SpinLock. For an example, see How to: Use SpinLock for Low-Level Synchronization.

SpinLock supports a *thread-tracking* mode that you can use during the development phase to help track the thread that is holding the lock at a specific time. Thread-tracking mode is very useful for debugging, but we recommend that you turn it off in the release version of your program because it may slow performance. For more information, see How to: Enable Thread-Tracking Mode in SpinLock.

See also

- Threading Objects and Features

How to: use SpinLock for low-level synchronization

9/17/2018 • 3 minutes to read • Edit Online

The following example demonstrates how to use a SpinLock. In this example, the critical section performs a minimal amount of work, which makes it a good candidate for a SpinLock. Increasing the work a small amount increases the performance of the SpinLock compared to a standard lock. However, there is a point at which a SpinLock becomes more expensive than a standard lock. You can use the concurrency profiling functionality in the profiling tools to see which type of lock provides better performance in your program. For more information, see Concurrency Visualizer.

```
class SpinLockDemo2
{
    const int N = 100000;
    static Queue<Data> _queue = new Queue<Data>();
    static object _lock = new Object();
    static SpinLock _spinlock = new SpinLock();

    class Data
    {
        public string Name { get; set; }
        public double Number { get; set; }
    }
    static void Main(string[] args)
    {

        // First use a standard lock for comparison purposes.
        UseLock();
        _queue.Clear();
        UseSpinLock();

        Console.WriteLine("Press a key");
        Console.ReadKey();

    }

    private static void UpdateWithSpinLock(Data d, int i)
    {
        bool lockTaken = false;
        try
        {
            _spinlock.Enter(ref lockTaken);
            _queue.Enqueue( d );
        }
        finally
        {
            if (lockTaken) _spinlock.Exit(false);
        }
    }

    private static void UseSpinLock()
    {

          Stopwatch sw = Stopwatch.StartNew();

          Parallel.Invoke(
                  () => {
                      for (int i = 0; i < N; i++)
                      {
                          UpdateWithSpinLock(new Data() { Name = i.ToString(), Number = i }, i);
                      }
```

```
                }
            },
            () => {
                for (int i = 0; i < N; i++)
                {
                    UpdateWithSpinLock(new Data() { Name = i.ToString(), Number = i }, i);
                }
            }
        );
        sw.Stop();
        Console.WriteLine("elapsed ms with spinlock: {0}", sw.ElapsedMilliseconds);
    }

    static void UpdateWithLock(Data d, int i)
    {
        lock (_lock)
        {
            _queue.Enqueue(d);
        }
    }

    private static void UseLock()
    {
        Stopwatch sw = Stopwatch.StartNew();

        Parallel.Invoke(
                () => {
                    for (int i = 0; i < N; i++)
                    {
                        UpdateWithLock(new Data() { Name = i.ToString(), Number = i }, i);
                    }
                },
                () => {
                    for (int i = 0; i < N; i++)
                    {
                        UpdateWithLock(new Data() { Name = i.ToString(), Number = i }, i);
                    }
                }
            );
        sw.Stop();
        Console.WriteLine("elapsed ms with lock: {0}", sw.ElapsedMilliseconds);
    }
}
```

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

Class SpinLockDemo2

    Const N As Integer = 100000
    Shared _queue = New Queue(Of Data)()
    Shared _lock = New Object()
    Shared _spinlock = New SpinLock()

    Class Data
        Public Name As String
        Public Number As Double
    End Class
    Shared Sub Main()

        ' First use a standard lock for comparison purposes.
        UseLock()
        _queue.Clear()
        UseSpinLock()

        Console.WriteLine("Press a key")
        Console.ReadKey()
```

```
    End Sub

    Private Shared Sub UpdateWithSpinLock(ByVal d As Data, ByVal i As Integer)

        Dim lockTaken As Boolean = False
        Try
            _spinlock.Enter(lockTaken)
            _queue.Enqueue(d)
        Finally

            If lockTaken Then
                _spinlock.Exit(False)
            End If
        End Try
    End Sub

    Private Shared Sub UseSpinLock()


        Dim sw = Stopwatch.StartNew()

        Parallel.Invoke(
               Sub()
                   For i As Integer = 0 To N - 1
                       UpdateWithSpinLock(New Data() With {.Name = i.ToString(), .Number = i}, i)
                   Next
               End Sub,
                Sub()
                    For i As Integer = 0 To N - 1
                        UpdateWithSpinLock(New Data() With {.Name = i.ToString(), .Number = i}, i)
                    Next
                End Sub
            )
        sw.Stop()
        Console.WriteLine("elapsed ms with spinlock: {0}", sw.ElapsedMilliseconds)
    End Sub

    Shared Sub UpdateWithLock(ByVal d As Data, ByVal i As Integer)

        SyncLock (_lock)
            _queue.Enqueue(d)
        End SyncLock
    End Sub

    Private Shared Sub UseLock()

        Dim sw = Stopwatch.StartNew()

        Parallel.Invoke(
                Sub()
                    For i As Integer = 0 To N - 1
                        UpdateWithLock(New Data() With {.Name = i.ToString(), .Number = i}, i)
                    Next
                End Sub,
               Sub()
                   For i As Integer = 0 To N - 1
                       UpdateWithLock(New Data() With {.Name = i.ToString(), .Number = i}, i)
                   Next
               End Sub
                )
        sw.Stop()
        Console.WriteLine("elapsed ms with lock: {0}", sw.ElapsedMilliseconds)
    End Sub
End Class
```

SpinLock might be useful when a lock on a shared resource is not going to be held for very long. In such cases, on multi-core computers it can be efficient for the blocked thread to spin for a few cycles until the lock is released. By

spinning, the thread does not become blocked, which is a CPU-intensive process. SpinLock will stop spinning under certain conditions to prevent starvation of logical processors or priority inversion on systems with Hyper-Threading.

This example uses the System.Collections.Generic.Queue<T> class, which requires user synchronization for multi-threaded access. In applications that target the .NET Framework version 4, another option is to use the System.Collections.Concurrent.ConcurrentQueue<T>, which does not require any user locks.

Note the use of `false` (`False` in Visual Basic) in the call to SpinLock.Exit. This provides the best performance. Specify `true` (`True` in Visual Basic) on IA64 architectures to use the memory fence, which flushes the write buffers to ensure that the lock is now available for other threads to exit.

See also

- Threading objects and features
- lock statement (C#)
- SyncLock statement (Visual Basic)

How to: Enable Thread-Tracking Mode in SpinLock

4/28/2019 • 3 minutes to read • Edit Online

System.Threading.SpinLock is a low-level mutual exclusion lock that you can use for scenarios that have very short wait times. SpinLock is not re-entrant. After a thread enters the lock, it must exit the lock correctly before it can enter again. Typically, any attempt to re-enter the lock would cause deadlock, and deadlocks can be very difficult to debug. As an aid to development, System.Threading.SpinLock supports a thread-tracking mode that causes an exception to be thrown when a thread attempts to re-enter a lock that it already holds. This lets you more easily locate the point at which the lock was not exited correctly. You can turn on thread-tracking mode by using the SpinLock constructor that takes a Boolean input parameter, and passing in an argument of `true`. After you complete the development and testing phases, turn off thread-tracking mode for better performance.

Example

The following example demonstrates thread-tracking mode. The lines that correctly exit the lock are commented out to simulate a coding error that causes one of the following results:

- An exception is thrown if the SpinLock was created by using an argument of `true` (`True` in Visual Basic).
- Deadlock if the SpinLock was created by using an argument of `false` (`False` in Visual Basic).

```
using System;
using System.Collections.Generic;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace SpinLockDemo
{
    // C#
    public class SpinLockTest
    {
        // Specify true to enable thread tracking. This will cause
        // exception to be thrown when the first thread attempts to reenter the lock.
        // Specify false to cause deadlock due to coding error below.
        private static SpinLock _spinLock = new SpinLock(true);

        static void Main()
        {
            Parallel.Invoke(
                () => DoWork(),
                () => DoWork(),
                () => DoWork(),
                () => DoWork()
                );

            Console.WriteLine("Press any key.");
            Console.ReadKey();
        }

        public static void DoWork()
        {
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < 100; i++)
            {
```

```
                bool lockTaken = false;

                try
                {
                    _spinLock.Enter(ref lockTaken);

                    // do work here protected by the lock
                    Thread.SpinWait(50000);
                    sb.Append(Thread.CurrentThread.ManagedThreadId);
                    sb.Append(" Entered-");
                }
                catch (LockRecursionException ex)
                {
                    Console.WriteLine("Thread {0} attempted to reenter the lock",
                                       Thread.CurrentThread.ManagedThreadId);
                    throw;
                }
                finally
                {
                    // INTENTIONAL CODING ERROR TO DEMONSTRATE THREAD TRACKING!
                    // UNCOMMENT THE LINES FOR CORRECT SPINLOCK BEHAVIOR
                    // Commenting out these lines causes the same thread
                    // to attempt to reenter the lock. If the SpinLock was
                    // created with thread tracking enabled, the exception
                    // is thrown. Otherwise the spinlock deadlocks.
                    if (lockTaken)
                    {
                       // _spinLock.Exit(false);
                       // sb.Append("Exited ");
                    }
                }

                // Output for diagnostic display.
                if(i % 4 != 0)
                    Console.Write(sb.ToString());
                else
                    Console.WriteLine(sb.ToString());
                sb.Clear();

            }
        }
    }
}
```

```
Imports System.Text
Imports System.Threading
Imports System.Threading.Tasks

Module Module1

    Public Class SpinTest

        ' True means "enable thread tracking." This will cause an
        ' exception to be thrown when the first thread attempts to reenter the lock.
        ' Specify False to cause deadlock due to coding error below.
        Private Shared _spinLock = New SpinLock(True)

        Public Shared Sub Main()

            Parallel.Invoke(
                Sub() DoWork(),
                Sub() DoWork(),
                Sub() DoWork(),
                Sub() DoWork()
                )
```

```
            Console.WriteLine("Press any key.")
            Console.ReadKey()
        End Sub

        Public Shared Sub DoWork()

            Dim sb = New StringBuilder()

            For i As Integer = 1 To 9999

                Dim lockTaken As Boolean = False

                Try
                    _spinLock.Enter(lockTaken)

                    ' do work here protected by the lock
                    Thread.SpinWait(50000)
                    sb.Append(Thread.CurrentThread.ManagedThreadId)
                    sb.Append(" Entered-")

                Catch ex As LockRecursionException
                    Console.WriteLine("Thread {0} attempted to reenter the lock",
                                       Thread.CurrentThread.ManagedThreadId)
                    Throw

                Finally

                    ' INTENTIONAL CODING ERROR TO DEMONSTRATE THREAD TRACKING!
                    ' UNCOMMENT THE LINES FOR CORRECT SPINLOCK BEHAVIOR
                    ' Commenting out these lines causes the same thread
                    ' to attempt to reenter the lock. If the SpinLock was
                    ' created with thread tracking enabled, the exception
                    ' is thrown. Otherwise, if the SpinLock was created with a
                    ' parameter of false, and these lines are left commented, the spinlock deadlocks.
                    If (lockTaken) Then

                        '  _spinLock.Exit()
                        '  sb.Append("Exited ")
                    End If
                End Try

                ' Output for diagnostic display.
                If (i Mod 4 <> 0) Then
                    Console.Write(sb.ToString())
                Else
                    Console.WriteLine(sb.ToString())
                End If
                sb.Clear()
            Next
        End Sub
    End Class
End Module
```

See also

- SpinLock

SpinWait

1/23/2019 • 2 minutes to read • Edit Online

System.Threading.SpinWait is a lightweight synchronization type that you can use in low-level scenarios to avoid the expensive context switches and kernel transitions that are required for kernel events. On multicore computers, when a resource is not expected to be held for long periods of time, it can be more efficient for a waiting thread to spin in user mode for a few dozen or a few hundred cycles, and then retry to acquire the resource. If the resource is available after spinning, then you have saved several thousand cycles. If the resource is still not available, then you have spent only a few cycles and can still enter a kernel-based wait. This spinning-then-waiting combination is sometimes referred to as a *two-phase wait operation*.

SpinWait is designed to be used in conjunction with the .NET Framework types that wrap kernel events such as ManualResetEvent. SpinWait can also be used by itself for basic spinning functionality in just one program.

SpinWait is more than just an empty loop. It is carefully implemented to provide correct spinning behavior for the general case, and will itself initiate context switches if it spins long enough (roughly the length of time required for a kernel transition). For example, on single-core computers, SpinWait yields the time slice of the thread immediately because spinning blocks forward progress on all threads. SpinWait also yields even on multi-core machines to prevent the waiting thread from blocking higher-priority threads or the garbage collector. Therefore, if you are using a SpinWait in a two-phase wait operation, we recommend that you invoke the kernel wait before the SpinWait itself initiates a context switch. SpinWait provides the NextSpinWillYield property, which you can check before every call to SpinOnce. When the property returns `true`, initiate your own Wait operation. For an example, see How to: Use SpinWait to Implement a Two-Phase Wait Operation.

If you are not performing a two-phase wait operation but are just spinning until some condition is true, you can enable SpinWait to perform its context switches so that it is a good citizen in the Windows operating system environment. The following basic example shows a SpinWait in a lock-free stack. If you require a high-performance, thread-safe stack, consider using System.Collections.Concurrent.ConcurrentStack<T>.

```
public class LockFreeStack<T>
{
    private volatile Node m_head;

    private class Node { public Node Next; public T Value; }

    public void Push(T item)
    {
        var spin = new SpinWait();
        Node node = new Node { Value = item }, head;
        while (true)
        {
            head = m_head;
            node.Next = head;
            if (Interlocked.CompareExchange(ref m_head, node, head) == head) break;
            spin.SpinOnce();
        }
    }

    public bool TryPop(out T result)
    {
        result = default(T);
        var spin = new SpinWait();

        Node head;
        while (true)
        {
            head = m_head;
            if (head == null) return false;
            if (Interlocked.CompareExchange(ref m_head, head.Next, head) == head)
            {
                result = head.Value;
                return true;
            }
            spin.SpinOnce();
        }
    }
}
```

```vb
Imports System.Threading

Module SpinWaitDemo


    Public Class LockFreeStack(Of T)
        Private m_head As Node

        Private Class Node
            Public [Next] As Node
            Public Value As T
        End Class

        Public Sub Push(ByVal item As T)
            Dim spin As New SpinWait()
            Dim head As Node, node As New Node With {.Value = item}

            While True
                Thread.MemoryBarrier()
                head = m_head
                node.Next = head
                If Interlocked.CompareExchange(m_head, node, head) Is head Then Exit While
                spin.SpinOnce()
            End While
        End Sub

        Public Function TryPop(ByRef result As T) As Boolean
            result = CType(Nothing, T)
            Dim spin As New SpinWait()

            Dim head As Node
            While True
                Thread.MemoryBarrier()
                head = m_head
                If head Is Nothing Then Return False
                If Interlocked.CompareExchange(m_head, head.Next, head) Is head Then
                    result = head.Value
                    Return True
                End If
                spin.SpinOnce()
            End While
        End Function
    End Class


End Module
```

See also

- SpinWait
- Threading Objects and Features

How to: Use SpinWait to Implement a Two-Phase Wait Operation

1/23/2019 • 5 minutes to read • Edit Online

The following example shows how to use a System.Threading.SpinWait object to implement a two-phase wait operation. In the first phase, the synchronization object, a `Latch`, spins for a few cycles while it checks whether the lock has become available. In the second phase, if the lock becomes available, then the `Wait` method returns without using the System.Threading.ManualResetEvent to perform its wait; otherwise, `Wait` performs the wait.

Example

This example shows a very basic implementation of a Latch synchronization primitive. You can use this data structure when wait times are expected to be very short. This example is for demonstration purposes only. If you require latch-type functionality in your program, consider using System.Threading.ManualResetEventSlim.

```
#define LOGGING

using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

class Latch
{
   private object latchLock = new object();
   // 0 = unset, 1 = set.
   private int m_state = 0;
   private volatile int totalKernelWaits = 0;

   // Block threads waiting for ManualResetEvent.
   private ManualResetEvent m_ev = new ManualResetEvent(false);
#if LOGGING
   // For fast logging with minimal impact on latch behavior.
   // Spin counts greater than 20 might be encountered depending on machine config.
   private long[] spinCountLog = new long[20];

   public void DisplayLog()
   {
      for (int i = 0; i < spinCountLog.Length; i++)
      {
          Console.WriteLine("Wait succeeded with spin count of {0} on {1:N0} attempts",
                            i, spinCountLog[i]);
      }
      Console.WriteLine("Wait used the kernel event on {0:N0} attempts.", totalKernelWaits);
      Console.WriteLine("Logging complete");
   }
#endif

   public void Set()
   {
      lock(latchLock) {
         m_state = 1;
         m_ev.Set();
      }
   }

   public void Wait()
   {
```

```
      Trace.WriteLine("Wait timeout infinite");
      Wait(Timeout.Infinite);
   }

   public bool Wait(int timeout)
   {
      SpinWait spinner = new SpinWait();
      Stopwatch watch;

      while (m_state == 0)
      {
          // Lazily allocate and start stopwatch to track timeout.
          watch = Stopwatch.StartNew();

          // Spin only until the SpinWait is ready
          // to initiate its own context switch.
          if (!spinner.NextSpinWillYield)
          {
              spinner.SpinOnce();
          }
          // Rather than let SpinWait do a context switch now,
          //  we initiate the kernel Wait operation, because
          // we plan on doing this anyway.
          else
          {
              Interlocked.Increment(ref totalKernelWaits);
              // Account for elapsed time.
              long realTimeout = timeout - watch.ElapsedMilliseconds;

              // Do the wait.
              if (realTimeout <= 0 || !m_ev.WaitOne((int)realTimeout))
              {
                  Trace.WriteLine("wait timed out.");
                  return false;
              }
          }
      }

#if LOGGING
      Interlocked.Increment(ref spinCountLog[spinner.Count]);
#endif
      // Take the latch.
      Interlocked.Exchange(ref m_state, 0);

      return true;
   }
}

class Example
{
   static Latch latch = new Latch();
   static int count = 2;
   static CancellationTokenSource cts = new CancellationTokenSource();

   static void TestMethod()
   {
      while (!cts.IsCancellationRequested)
      {
         // Obtain the latch.
         if (latch.Wait(50))
         {
            // Do the work. Here we vary the workload a slight amount
            // to help cause varying spin counts in latch.
            double d = 0;
            if (count % 2 != 0) {
               d = Math.Sqrt(count);
            }
            Interlocked.Increment(ref count);
```

```
            // Release the latch.
            latch.Set();
         }
      }
   }

   static void Main()
   {
      // Demonstrate latch with a simple scenario: multiple
      // threads updating a shared integer. Both operations
      // are relatively fast, which enables the latch to
      // demonstrate successful waits by spinning only.
      latch.Set();

      // UI thread. Press 'c' to cancel the loop.
      Task.Factory.StartNew(() =>
      {
         Console.WriteLine("Press 'c' to cancel.");
         if (Console.ReadKey(true).KeyChar == 'c') {
            cts.Cancel();
         }
      });

      Parallel.Invoke( () => TestMethod(),
                       () => TestMethod(),
                       () => TestMethod() );

#if LOGGING
      latch.DisplayLog();
      if (cts != null) cts.Dispose();
#endif
   }
}
```

```
#Const LOGGING = 1

Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks

Class Latch
   Private latchLock As New Object()
   ' 0 = unset, 1 = set.
   Private m_state As Integer = 0
   Private totalKernelWaits As Integer = 0

   ' Block threads waiting for ManualResetEvent.
   Private m_ev = New ManualResetEvent(False)

#If LOGGING Then
   ' For fast logging with minimal impact on latch behavior.
   ' Spin counts greater than 20 might be encountered depending on machine config.
   Dim spinCountLog(19) As Long

   Public Sub DisplayLog()
      For i As Integer = 0 To spinCountLog.Length - 1
         Console.WriteLine("Wait succeeded with spin count of {0} on {1:N0} attempts",
                           i, spinCountLog(i))
      Next
      Console.WriteLine("Wait used the kernel event on {0:N0} attempts.",
                        totalKernelWaits)
      Console.WriteLine("Logging complete")
   End Sub
#End If

   Public Sub SetLatch()
      SyncLock(latchLock)
```

```
            m_state = 1
            m_ev.Set()
        End SyncLock
    End Sub

    Public Sub Wait()
        Trace.WriteLine("Wait timeout infinite")
        Wait(Timeout.Infinite)
    End Sub

    Public Function Wait(ByVal timeout As Integer) As Boolean
        ' Allocated on the stack.
        Dim spinner = New SpinWait()
        Dim watch As Stopwatch

        While (m_state = 0)
            ' Lazily allocate and start stopwatch to track timeout.
            watch = Stopwatch.StartNew()

            ' Spin only until the SpinWait is ready
            ' to initiate its own context switch.
            If Not spinner.NextSpinWillYield Then
                spinner.SpinOnce()

                ' Rather than let SpinWait do a context switch now,
                '  we initiate the kernel Wait operation, because
                ' we plan on doing this anyway.
            Else
                Interlocked.Increment(totalKernelWaits)
                ' Account for elapsed time.
                Dim realTimeout As Long = timeout - watch.ElapsedMilliseconds

                ' Do the wait.
                If realTimeout <= 0 OrElse Not m_ev.WaitOne(realTimeout) Then
                    Trace.WriteLine("wait timed out.")
                    Return False
                End If
            End If
        End While

#If LOGGING Then
        Interlocked.Increment(spinCountLog(spinner.Count))
#End If
        ' Take the latch.
        Interlocked.Exchange(m_state, 0)

        Return True
    End Function
End Class

Class Program
    Shared latch = New Latch()
    Shared count As Integer = 2
    Shared cts = New CancellationTokenSource()
    Shared lockObj As New Object()

    Shared Sub TestMethod()
        While (Not cts.IsCancellationRequested)
            ' Obtain the latch.
            If (latch.Wait(50)) Then
                ' Do the work. Here we vary the workload a slight amount
                ' to help cause varying spin counts in latch.
                Dim d As Double = 0
                If (count Mod 2 <> 0) Then
                    d = Math.Sqrt(count)
                End If

                SyncLock(lockObj)
                    If count = Int32.MaxValue Then count = 0
```

```
                count += 1
            End SyncLock

            ' Release the latch.
            latch.SetLatch()
         End If
      End While
   End Sub

   Shared Sub Main()
      ' Demonstrate latch with a simple scenario:
      ' two threads updating a shared integer and
      ' accessing a shared StringBuilder. Both operations
      ' are relatively fast, which enables the latch to
      ' demonstrate successful waits by spinning only.
      latch.SetLatch()

      ' UI thread. Press 'c' to cancel the loop.
      Task.Factory.StartNew(Sub()
                               Console.WriteLine("Press 'c' to cancel.")
                               If (Console.ReadKey(True).KeyChar = "c"c) Then
                                  cts.Cancel()
                               End If
                            End Sub)
      Parallel.Invoke(
             Sub() TestMethod(),
             Sub() TestMethod(),
             Sub() TestMethod()
             )

#If LOGGING Then
         latch.DisplayLog()
#End If
          If cts IsNot Nothing Then cts.Dispose()
   End Sub
End Class
```

The latch uses the SpinWait object to spin in place only until the next call to `SpinOnce` causes the SpinWait to yield the time slice of the thread. At that point, the latch causes its own context switch by calling WaitOne on the ManualResetEvent and passing in the remainder of the time-out value.

The logging output shows how often the Latch was able to increase performance by acquiring the lock without using the ManualResetEvent.

See also

- SpinWait
- Threading Objects and Features