**Threads and Threading**

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

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

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

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

http://i.msdn.microsoft.com/Global/Images/clear.gif When To Use Multiple Threads

Software that requires user interaction must react to the user's activities as rapidly as possible to provide a rich user experience. At the same time, however, it must do the calculations necessary to present data to the user as fast as possible. If your application uses only one thread of execution, you can combine [asynchronous programming](http://msdn.microsoft.com/en-us/library/2e08f6yc.aspx) with [.NET Framework remoting](http://msdn.microsoft.com/en-us/library/kwdt6w2k.aspx) or [XML Web services](http://msdn.microsoft.com/en-us/library/7bkzywba.aspx) created using ASP.NET to use the processing time of other computers in addition to that of your own to increase responsiveness to the user and decrease the data processing time of your application. If you are doing intensive input/output work, you can also use I/O completion ports to increase your application's responsiveness.

**Advantages of Multiple Threads**

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

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

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

**Disadvantages of Multiple Threads**

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

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

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

Resources that require synchronization include:

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

**Threading and Application Design**

In general, using the [ThreadPool](http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx) class is the easiest way to handle multiple threads for relatively short tasks that will not block other threads and when you do not expect any particular scheduling of the tasks. However, there are a number of reasons to create your own threads:

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

**Threading and Exceptions**

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

* A [ThreadAbortException](http://msdn.microsoft.com/en-us/library/system.threading.threadabortexception.aspx) is thrown in a thread because [Abort](http://msdn.microsoft.com/en-us/library/system.threading.thread.abort.aspx) was called.
* An [AppDomainUnloadedException](http://msdn.microsoft.com/en-us/library/system.appdomainunloadedexception.aspx) is thrown in a thread because the application domain is being unloaded.
* The common language runtime or a host process terminates the thread.

For more information, see [Exceptions in Managed Threads](http://msdn.microsoft.com/en-us/library/ms228965.aspx).

|  |
| --- |
| **6kac2kdh.alert_note(en-us,VS.90).gifNote:** |
| In the .NET Framework versions 1.0 and 1.1, the common language runtime silently traps some exceptions, for example in thread pool threads. This may corrupt application state and eventually cause applications to hang, which might be very difficult to debug. |

**Managed Threading Best Practices**

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.

 Deadlocks and Race Conditions

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

**Deadlocks**

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

Many methods of the managed threading classes provide time-outs to help you detect deadlocks. For example, the following code attempts to acquire a lock on the current instance. If the lock is not obtained in 300 milliseconds, [Monitor..::.TryEnter](http://msdn.microsoft.com/en-us/library/system.threading.monitor.tryenter.aspx) returns **false**.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl13VisualBasic');)

If Monitor.TryEnter(Me, 300) Then

Try

' Place code protected by the Monitor here.

Finally

Monitor.Exit(Me)

End Try

Else

' Code to execute if the attempt times out.

End If

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl14CSharp');)

if (Monitor.TryEnter(this, 300)) {

try {

// Place code protected by the Monitor here.

}

finally {

Monitor.Exit(this);

}

}

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](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.aspx) class, such as [Interlocked..::.Increment](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.increment.aspx). To read about other techniques for synchronizing data among multiple threads, see [Synchronizing Data for Multithreading](http://msdn.microsoft.com/en-us/library/z8chs7ft.aspx).

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

 Number of Processors

Multithreading solves different problems for the single-processor computers that run most end-user software, and the multiprocessor computers typically used as servers.

**Single-Processor Computers**

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

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

**Multiprocessor Computers**

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

* The number of threads that can execute concurrently is limited by the number of processors.
* A background thread executes only when the number of foreground threads executing is smaller than the number of processors.
* When you call the [Thread..::.Start](http://msdn.microsoft.com/en-us/library/system.threading.thread.start.aspx) method on a thread, that thread might or might not start executing immediately, depending on the number of processors and the number of threads currently waiting to execute.
* Race conditions can occur not only because threads are preempted unexpectedly, but because two threads executing on different processors might be racing to reach the same code block.

 Static Members and Static Constructors

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

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

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

 General Recommendations

Consider the following guidelines when using multiple threads:

* Don't use [Thread..::.Abort](http://msdn.microsoft.com/en-us/library/system.threading.thread.abort.aspx) 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](http://msdn.microsoft.com/en-us/library/system.threading.thread.suspend.aspx) and [Thread..::.Resume](http://msdn.microsoft.com/en-us/library/system.threading.thread.resume.aspx) to synchronize the activities of multiple threads. Do use [Mutex](http://msdn.microsoft.com/en-us/library/system.threading.mutex.aspx), [ManualResetEvent](http://msdn.microsoft.com/en-us/library/system.threading.manualresetevent.aspx), [AutoResetEvent](http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent.aspx), and [Monitor](http://msdn.microsoft.com/en-us/library/system.threading.monitor.aspx).
* 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](http://msdn.microsoft.com/en-us/library/system.threading.monitor.pulseall.aspx) 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](http://msdn.microsoft.com/en-us/library/de0542zz.aspx) with [Type](http://msdn.microsoft.com/en-us/library/system.type.aspx) objects. For a given type, there is only one instance of [System..::.Type](http://msdn.microsoft.com/en-us/library/system.type.aspx) 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](http://msdn.microsoft.com/en-us/library/ms228970.aspx).
* 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](http://msdn.microsoft.com/en-us/library/c5kehkcz.aspx) statement and the Visual Basic [SyncLock](http://msdn.microsoft.com/en-us/library/3a86s51t.aspx) statement provide this behavior automatically, employing a **finally** block to ensure that [Monitor..::.Exit](http://msdn.microsoft.com/en-us/library/system.threading.monitor.exit.aspx) 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](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.aspx) 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](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.aspx) 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:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl95VisualBasic');)

SyncLock lockObject

myField += 1

End SyncLock

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl96CSharp');)

lock(lockObject)

{

myField++;

}

You can improve performance by using the [Increment](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.increment.aspx) method instead of the **lock** statement, as follows:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl98VisualBasic');)

System.Threading.Interlocked.Increment(myField)

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl99CSharp');)

System.Threading.Interlocked.Increment(myField);

|  |
| --- |
| **Note:** |
| In the .NET Framework version 2.0, the [Add](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.add.aspx) method provides atomic updates in increments larger than 1. |

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

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl102VisualBasic');)

If x Is Nothing Then

SyncLock lockObject

If x Is Nothing Then

x = y

End If

End SyncLock

End If

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl103CSharp');)

if (x == null)

{

lock (lockObject)

{

if (x == null)

{

x = y;

}

}

}

Performance can be improved by using the [CompareExchange](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.compareexchange.aspx) method instead, as follows:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl105VisualBasic');)

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

C#

[Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl106CSharp');)

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

|  |
| --- |
| **Note:** |
| In the .NET Framework version 2.0, the [CompareExchange](http://msdn.microsoft.com/en-us/library/system.threading.interlocked.compareexchange.aspx) method has a generic overload that can be used for type-safe replacement of any reference type. |

 Recommendations for Class Libraries

Consider the following guidelines when designing class libraries for multithreading:

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