**Threading in C#**

Multithreading or free-threading is the ability of an operating system to concurrently run programs that have been divided into subcomponents, or threads.

Technically, multithreaded programming requires a multitasking/multithreading operating system, such as GNU/Linux, Windows NT/2000 or OS/2; capable of running many programs concurrently, and of course, programs have to be written in a special way in order to take advantage of these multitasking operating systems which appear to function as multiple processors. In reality, the user's sense of time is much slower than the processing speed of a computer, and multitasking appears to be simultaneous, even though only one task at a time can use a computer processing cycle.

Features and Benefits of Threads

Mutually exclusive tasks, such as gathering user input and background processing can be managed with the use of threads. Threads can also be used as a convenient way to structure a program that performs several similar or identical tasks concurrently.

One of the advantages of using the threads is that you can have multiple activities happening simultaneously. Another advantage is that a developer can make use of threads to achieve faster computations by doing two different computations in two threads instead of serially one after the other.

Threading Concepts in C#

In .NET, threads run in AppDomains. An AppDomain is a runtime representation of a logical process within a physical process. And a thread is the basic unit to which the OS allocates processor time. To start with, each AppDomain is started with a single thread. But it is capable of creating other threads from the single thread and from any created thread as well.

A multitasking operation system divides the available processor time among the processes and threads that need it. A thread is executed in the given time slice, and then it is suspended and execution starts for next thread/process in the queue. When the OS switches from one thread to another, it saves thread context for preempted thread and loads the thread context for the thread to execute.

The length of time slice that is allocated for a thread depends on the OS, the processor, as also on the priority of the task itself.

In .NET framework, System.Threading namespace provides classes and interfaces that enable multi-threaded programming. This namespace provides:

* ThreadPool class for managing group of threads,
* Timer class to enable calling of delegates after a certain amount of time,
* A Mutex class for synchronizing mutually exclusive threads, along with classes for scheduling the threads, sending wait notifications and deadlock resolutions.

Information on this namespace is available in the help documentations in the Framework SDK.

#### Defining and Calling threads

To get a feel of how Threading works, run the below code:

using System;

using System.Threading;

public class ServerClass

{

// The method that will be called when the thread is started.

public void Instance Method()

{

Console.WriteLine("You are in InstranceMethod.Running on Thread A”);

Console.WriteLine("Thread A Going to Sleep Zzzzzzzz”);

// Pause for a moment to provide a delay to make threads more apparent.

Thread. Sleep(3000);

Console.WriteLine ("You are Back in InstanceMethod.Running on Thread A");

}

public static void StaticMethod()

{

Console.WriteLine("You are in StaticMethod. Running on Thread B.");

// Pause for a moment to provide a delay to make threads more apparent.

Console.WriteLine("Thread B Going to Sleep Zzzzzzzz");

Thread.Sleep(5000);

Console.WriteLine("You are back in static method. Running on Thread B");

}

}

public class Simple

{

public static int Main(String[] args)

{

Console.WriteLine ("Thread Simple Sample");

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 threads.");

return 0;

}

}

If the code in this example is compiled and executed, you would notice how processor time is allocated between the two method calls. If not for threading, you would have to wait till the first method slept for 3000 secs for the next method to be called. Try disabling threading in the above code and notice how they work. Nevertheless, execution time for both would be the same.

An important property of this class (which is also settable) is Priority.

### Scheduling Threads

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 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 run able 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 run able, 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.

### Pausing and Resuming threads

After you have started a thread, you often want to pause that thread for a fixed period of time. Calling [Thread.Sleep](http://www.codeproject.com/Articles/8694/_target) causes the current thread to immediately block for the number of milliseconds you pass to Sleep, yielding the remainder of its time slice to another thread. One thread cannot call Sleep on another thread. Calling Thread.Sleep(Timeout.Infinite) causes a thread to sleep until it is interrupted by another thread that calls [Thread.Interrupt](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassInterruptTopic.asp) or is aborted by [Thread.Abort](http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemThreadingThreadClassAbortTopic.asp).

### Thread Safety

When we are working in a multi threaded environment, we need to maintain that no thread leaves the object in an invalid state when it gets suspended. Thread safety basically means the members of an object always maintain a valid state when used concurrently by multiple threads.

There are multiple ways of achieving this – The Mutex class or the Monitor classes of the Framework enable this, and more information on both is available in the Framework SDK documentation. What we are going to look at here is the use of locks.

You put a lock on a block of code – which means that that block has to be executed at one go and that at any given time, only one thread could be executing that block.

The syntax for the lock would be as follows:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/8694/Multithreading-Concepts-in-C)

using System;

using System.Threading;

//define the namespace, class etc.

...

public somemethod(...)

{

...

lock(this)

{

Console.WriteLine(“Inside the lock now”);

...

}

}

In the above code sample, the code block following the lock statement will be executed as one unit of execution, and only one thread would be able to execute it at any given time. So, once a thread enters that block, no other thread can enter the block till the first thread has exited it.

This becomes necessary in the kind of database transactions required in banking applications and reservations systems etc.