

A Look into C# Threads
CS384: Design of Operating Systems
By: Dave Pederson

Submitted To: Dr. Chris Taylor
Date: 2/3/2003

<u>Table of Contents</u>	2
I. Introduction.....	3
A. Purpose.....	3
B. Scope.....	3
C. The .NET Framework.....	4
D. The C# Language.....	5
II. Multi-threading.....	6
A. What is a Thread.....	6
B. Multi-Threading and C#.....	7
C. When to Use Multiple Threads.....	10
D. Advantages of Multiple Threads.....	10
E. Disadvantages of Multiple Threads.....	11
F. Thread Interaction and Synchronization.....	11
III. Threading Objects and Features.....	13
A. Thread Pooling.....	13
B. Mutex.....	13
C. Interlocked.....	14
IV. Conclusion.....	14
V. Bibliography.....	15

Introduction:

Purpose:

The main purpose of this document is to provide a look into the usage of threads in the C# programming language. The second purpose is to give a general background of the Microsoft .NET Framework. This report will cover many aspects and features of C# threads. First, a background of the C# language and the .NET Framework will be established. The report will then lay out the definition of a thread and how to create and execute a thread with a simple code example.

After a basic foundation of a C# thread is shown, an investigation into using multiple threads will be developed. This will include such topics as when to use multiple threads, advantages and disadvantages of multiple threads, and thread synchronization and interaction.

When using C# threads, there are many objects and features available as well. Finally, this report will cover some of these important features. These features include thread pooling, Mutual Exclusion (mutex), and interlocked threads. Overall, this report shall paint a general, but useful picture of C# threads within the .NET Framework.

Scope:

As mentioned above, this document will give a broad look into C# threads and establish their usage within the .NET Framework. This document is intended to provide information about a fairly new and technical topic for the Design of Operating Systems class at the Milwaukee School of Engineering. The intended audience is the current professor and future students who will take Design of Operating Systems within the next two to three years. Also, as reports of students are posted on the Internet upon submission, any other people who might find this report helpful or informative (provided they find it at all) are also considered in the scope of this document.

The .NET Framework

The .NET Framework is an integral Windows component that supports building and running the next generation of applications and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. The runtime is an agent that manages code at execution time, providing core services such as memory management, and thread management, while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services. The following applications can all be developed in the .NET Framework:

- Console applications.

- Scripted or hosted applications.
- Windows GUI applications (Windows Forms).
- ASP.NET applications.
- XML Web services.
- Windows services.

As expected in any object-oriented language, a wide variety of programming tasks can be performed. Through the above types of applications, one can program tasks such as string management, data collection, database connectivity and file access along with a variety of specialized development scenarios. For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If one writes a Web Form application, the Windows Forms classes can be used [Visual Studio].

The C# Language:

For the past twenty years or so, C and C++ have been the most widely used languages for developing commercial and business software. While both languages provide the programmer with a tremendous amount of fine-grained control, this flexibility can come with some cost to productivity. Compared with a language such as Microsoft Visual Basic® or even Java, equivalent C and C++ applications often take longer to develop. Due to the potential complexity and long cycle times associated with these languages, Microsoft decided to create a language that offers a better balance between power and productivity.

An ideal solution for C and C++ programmers would be rapid development combined with the power to access all the functionality of the underlying platform. This would be done with an environment that is completely in sync with emerging Web standards and one that provides easy integration with existing Windows applications. However, the solution should also allow the ability to code at a low level when the need arises.

The Microsoft solution to this problem is a language called C# (pronounced "C sharp"). C# is a modern, object-oriented language that enables programmers to quickly build a wide range of applications for the new Microsoft .NET development environment, which provides tools and services that fully exploit both computing and communications. Using simple C# language constructs, these components can be converted into XML Web services, allowing them to be invoked across the Internet, from any language running on any operating system.

More than anything else, C# was designed to bring rapid development to the C++ programmer without sacrificing the power and control that have been a hallmark of C and C++. Because of this heritage, C# has a high degree of fidelity with C and C++. Microsoft's hope is that developers familiar with these languages can quickly become productive in C# [Visual Studio].

Multi-Threading:

What is a Thread?

A Thread is a flow of control within a single process and is often referred to as a lightweight process. When computers were first invented, they were capable of executing one program at a time. Once one program was completely executed, a second one was chosen, and so on. With time, the concept of timesharing was developed whereby each program was given a specific amount of processor time (a quantum) and when its time got over the second program standing in queue was called upon. Each running program (called a process) had its own memory space, stack, heap and set of variables. One process could spawn another process, but once that occurred the two behaved independently of each other.

The above can all be said to be heavyweight processes. Threads are similar to these heavyweight processes in many ways, but they are very different in others. A thread will have a memory space, program counter, stack, heap and a set of variables just as a heavyweight process. The difference is that there can be multiple threads associated with one process, allowing shared access to resources such as code sections and data sections. This allows the process to perform multiple tasks at the same time [Silberschatz, Galvin, Gagne].

Multi-Threading and C#:

Applications written on the .NET platform are naturally threaded. In C#, the runtime environment starts execution of the program with the Main() method in one thread. There is an automatic garbage collection happening in the background, which takes place in a second thread. All of this happens so naturally that it usually goes unnoticed. There arise situations, however, when we need to add threads to our application.

In C#, the classes and interfaces in the System.Threading namespace provide the multi-threading support in the .NET platform. This namespace consists of a number of classes.

`System.Threading.Thread` is the main class for creating threads and controlling them. The `Thread` class has a number of methods. A few interesting methods are listed below:

- `Start()`: starts the execution of the thread.
- `Suspend()`: suspends the thread, if the thread is already suspended, nothing happens.
- `Resume()` : resumes a thread that has been suspended.
- `Interrupt()`: interrupts a thread that is in the wait, sleep or join stage.
- `Join()`: blocks a calling thread until the thread terminates.
- `Sleep(int x)` : suspends the thread for specified amount of time (in milliseconds).
- `Abort()`: Begins the process of terminating the thread. Once the thread terminates, it cannot be restarted by calling the function `Start()` again.

A thread can be paused/blocked by calling `Thread.Sleep` or `Thread.Suspend` or `Thread.Join`. Calling the method `Sleep()` or `Suspend()` on a thread means, the thread does not get any processor time. There is a difference between these two ways of pausing a thread. `Thread.Sleep` causes a thread to stop immediately but the common language runtime waits until the thread has reached some safe point before calling the `Suspend()` method on the thread. One thread cannot call `Sleep()` on another thread but one thread can call `Suspend()` on the other thread and it causes the other thread to pause. Calling `Resume()` on the suspended thread breaks the thread out of the suspended state and allows it to continue execution. A single call to `Resume()` is sufficient to activate a thread regardless of the number of times `Suspend()` was called to block it. A thread that has already been terminated or has not yet started functioning cannot be suspended. `Thread.Sleep(int.Infinite)` causes a thread to sleep indefinitely. The thread can only wake up when it is interrupted by another thread that calls `Thread.Interrupt` or is aborted by `Thread.Abort`. The `Thread.Interrupt` can be used to break a thread out of its blocking state but it throws a `ThreadInterruptedException`. This can either catch the exception, do whatever the user wants to do with the thread, or ignore the exception and let the run-time stop the thread. For managed wait, `Thread.Interrupt` and `Thread.Abort` both wake up the thread immediately.

It may be desirable at times to terminate a thread from some other thread. In such situations the `Thread.Abort` method can be used to stop a thread permanently and using this function throws a `ThreadAbortException`. The terminating thread can catch the exception but it is difficult to suppress it. The only way that it can be suppressed is by calling `Thread.ResetAbort`

method but it can only be called if this thread had been the one that had provoked the exception. Since, Thread.Abort is normally called by some thread A on some other thread B, B therefore, cannot invoke the method Thread.ResetAbort to suppress it from terminating. The Thread.Abort method lets the system quietly stop the thread without informing the user. Once aborted, a thread cannot be restarted. As this method does not say that the thread will abort immediately, hence to be sure that the thread has terminated, Thread.Join can be called to wait on the thread. Join is a blocking call that does not return until the thread has actually stopped executing. But, a thread may call Thread.Interrupt to interrupt another thread that is waiting on a call to Thread.Join.

Suspend() should be avoided when trying to block a thread whenever possible, as it could lead to serious problems like deadlocks. Imagine, what would happen if we suspend a thread that holds a resource that another thread would need. Rather, different threads should be given different priorities based on their importance. Thread.Priority should be used rather than Thread.Suspend.

The Thread class also has a number of interesting properties as listed below:

- IsAlive: (if true, signifies that thread has been started and has not yet been terminated or aborted)
- Name (gets/sets the name of the thread)
- Priority (gets/sets the scheduling priority of a thread)
- ThreadState (gets a value containing the state of the current thread).

The example, shown below, is a very simple one that discusses how to apply a limited number of these thread properties. To create a thread, we need to instantiate the Thread class, passing a ThreadStart delegate (System.Threading.ThreadStart) in its constructor. This delegate contains the method where the thread will begin execution, when started. The Start() method of the Thread class then starts the execution of a new thread [Dhar, Krikorian].

```
using System;
using System.Threading ;
namespace LearnThreads
{
class Thread_App
{
    public static void First_Thread()
    {
```

```

        Console.WriteLine("First thread created");
        Thread current_thread = Thread.CurrentThread;
        string thread_details = "Thread Name: " + current_thread.Name +
        "\r\nThread State: " + current_thread.ThreadState.ToString()+
        "\r\n Thread Priority level:"+current_thread.Priority.ToString();
        Console.WriteLine("The details of the thread are :"+/
thread_details);
        Console.WriteLine ("first thread terminated");
    }
    public static void Main()
    {
        ThreadStart thr_start_func = new ThreadStart (First_Thread);
        Console.WriteLine ("Creating the first thread ");
        Thread fThread = new Thread (thr_start_func);
        fThread.Name = "first_thread";
        fThread.Start (); //starting the thread
    }
}
}

```

When To Use Multiple Threads:

Programmers use multiple threads of control for a variety of reasons: to build responsive servers that interact with multiple clients, to run computations in parallel on a multiprocessor for performance, and as a structuring mechanism for creating rich user interfaces. In general, threads are useful whenever the software needs to manage a set of tasks with varying interaction latencies, exploit multiple physical resources or execute largely independent tasks in response to multiple external events. To put it more generally, any operation that is time-critical, or needs constant attention or is time-consuming should be placed in a thread of its own [Rinard].

Advantages of Multiple Threads:

Using more than one thread 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. A 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.

[Boyapati, Rinard]

Disadvantages of Multiple Threads:

There are certain issues that arise when dealing with multiple threads. Some of these include data flow (specifically data races), control flow, deadlocks and granularity. There are certain types of data that are potentially sensitive to a data racing condition. These are private, inherited, migrating, mutex, published and reader/writer data. Private data is accessed by only one thread. Inherited data is initialized by a parent thread and passed as a parameter to its child thread. Once the child starts execution, the parent no longer has access to the data. Migrating data is passed between parallel threads and is usually used in a producer/consumer relationship. Although multiple threads access migrating data at each point in time, there is a single thread that has conceptual ownership of the data and no other threads access the data until ownership changes. Published data is initialized by a single thread and then distributed to multiple reader threads for read only access. Mutex data is potentially accessed and updated by multiple parallel threads with the updates kept consistent with mutual exclusion synchronization [Boyapati, Rinard]. If any of these conditions apply, careful analysis is recommended before multiple threads of control are implemented.

Thread Interaction and Synchronization:

A multithreaded application usually has resources that can be accessed from multiple threads; for example, a global variable that is incremented or decremented by multiple threads. It is sometimes desirable to prevent multiple threads from concurrently altering the state of a resource. The .NET Framework includes several classes and data types that we can use to synchronize actions performed by two threads.

The simplest case is if we have a shared variable that we need to update from different threads. To do this, we can use the `System.Threading.Interlocked` class. For example, to

increment or decrement the shared variable called num, we'd write `Interlocked.Increment(num)` or `Interlocked.Decrement(num)`. we can also use `Interlocked` to set the variables to a specific value or to check the equality of two variables.

If there's a section of code in an object's method that should not be accessed concurrently by multiple threads, we can use the `Monitor` class to acquire a lock on that object by calling `Monitor.Enter(object)`. Any other thread wanting to execute the same code would need to acquire the same lock and will be paused until the first thread releases the lock by calling `Monitor.Exit(object)`.

For more control over thread synchronization or for cross-process synchronization, use the `Mutex` class, which is a named synchronization object that can be obtained from any thread in any process. Once we create or obtain the mutex, we use its `GetHandle` method to (as we'd expect) get a handle that we can use with the `WaitHandle.WaitAny` or `WaitHandle.WaitAll` methods. These two methods are blocking and will return only if the specified handle is signaled (that is, the mutex is not being used by another thread) or if the specified timeout expires. After we obtain the mutex, we perform the necessary synchronized processing and then call `Mutex.ReleaseMutex` to release it.

Sometimes we need a mechanism for one thread to notify other threads of some interesting event that occurred. In those cases we can use the .NET synchronization event classes, `ManualResetEvent` and `AutoResetEvent`. In the world of thread synchronization, an event is an object that has two states: signaled and nonsignaled. An event has a handle and can be used with `WaitHandle` just like a mutex. A thread that waits for an event will be blocked until another thread signals the event by calling `ManualResetEvent.Set` or `AutoResetEvent.Set`. If we are using a `ManualResetEvent`, we must call its `Reset` method to put it back to the nonsignaled state. An `AutoResetEvent` will automatically go back to nonsignaled as soon as a waiting thread is notified that the event became signaled [Dhar, Microsoft MSDN].

Threading Objects and Features:

Thread Pooling:

Thread pooling is one way to increase the efficiency of using multiple threads, depending

on the application. Many applications use multiple threads, but often those threads spend a great deal of time in the sleeping state waiting for an event to occur. Other threads might enter a sleeping state and be awakened only periodically to poll for a change or update status information before going to sleep again. Using thread pooling provides an application with a pool of worker threads that are managed by the system, allowing programmers the ability to concentrate on application tasks rather than thread management. In fact, if there exist a number of short tasks that require more than one thread, using the ThreadPool class is the easiest and best way to take advantage of multiple threads. Using a thread pool enables the system to optimize this for better throughput not only for this process but also with respect to other processes on the computer, something the application will know nothing about. Using a thread pool enables the system to optimize thread time slices taking into account all the current processes on a computer [Ewing].

Mutex:

A Mutex object can be used to synchronize between threads and across processes. Although Mutex doesn't have all of the wait and pulse functionality of other threading classes, it does offer the creation of named mutexes that can be used between processes. WaitOne, WaitAll, or WaitAny may be called to request ownership of the Mutex. The state of the Mutex is signaled if no thread owns it. 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. If a thread terminates normally while owning a Mutex, the state of the Mutex is set to signaled and the next waiting thread gets ownership [Ewing].

Interlocked:

The Interlocked methods CompareExchange, Decrement, Exchange, and Increment provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory.

The Increment and Decrement functions combine the operations of incrementing or decrementing the variable and checking the resulting value. This atomic operation is useful in a multitasking operating system, in which the system can interrupt one thread's execution to grant

a slice of processor time to another thread. Without such synchronization, one thread could increment a variable but be interrupted by the system before it could check the resulting value of the variable. A second thread could then increment the same variable. When the first thread receives its next time slice, it will check the value of the variable, which has now been incremented not once but twice. The Interlocked variable access functions protect against this kind of error.

The Exchange function atomically exchanges the values of the specified variables. The CompareExchange function combines two operations: comparing two values and storing a third value in one of the variables, based on the outcome of the comparison [Ewing].

Conclusion:

The .NET Framework is an integral Windows component that supports building and running the next generation of applications. C# is a modern, object-oriented language for creating applications on the .NET Framework. There are times during the development of applications with C#, where multiple threads of control are needed. Hopefully this report has given a good insight into how and when to create, use and manage multiple threads of control for the various types of applications supported on the .NET Framework.

Bibliography

Boyapati, Rinard. "A Parameterized System for Race Free Java Programs." ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), October, 2001.

Dhar, Ashish. c-sharpcenter.com. "Multithreaded Applications with C# - Part I". 09/17/2001 Tutorial, 1/6/2003. <http://www.c-sharpcenter.com/Tutorial/Thread_01.htm>

Dhar, Ashish. c-sharpcenter.com. "Multithreaded Programming Part II (Synchronization)". 09/17/2002. Tutorial. 1/6/2003.
<http://www.csharpcenter.com/Tutorial/MultiThread_02.htm>

Ewing, Greg. Microsoft MSDN. "Using Threads". March 2002
Using Threads. 1/6/2003
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbtchUsingThreads.asp>

Krikorian, Raffi. O'Reilly Network. "Multithreading with C#". 08/06/2002.
Multithreading. 1/6/2003
<<http://www.ondotnet.com/pub/a/dotnet/2001/08/06/csharp.html>>

Microsoft MSDN. C# Programmers Reference. "Threading Tutorial". 1/6/2003
<<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcwlkthreadingtutorial.asp>>

Rinard, Martin. "Analysis of Multithreaded Programs." Massachusetts Institute of Technology.
<<http://www.cag.lcs.mit.edu/~rinard>>

Visual Studio .NET Technical Resources. "C# Introduction and Overview". CSharpIntro.
1/6/2003
<<http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpintro.asp>>