<http://www.codeproject.com/Articles/43994/The-Practical-Guide-to-Multithreading-Part-1>

<http://www.codeproject.com/Articles/65158/The-Practical-Guide-to-Multithreading-Part-2>

<http://msdn.microsoft.com/en-us/library/aa645740(v=vs.71).aspx>

<http://www.albahari.com/threading/part2.aspx>

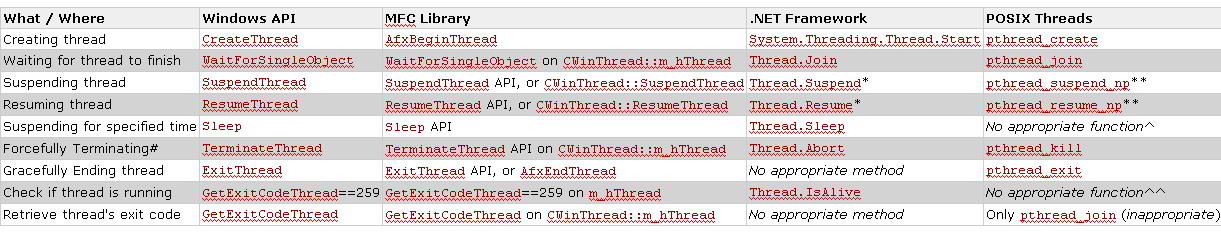
<http://msdn.microsoft.com/en-us/library/system.threading.mutex.aspx>

Cannot use DoEvents because it sometime crashes applications and not a perfect way of managing things.

**Multithreading**

To start a new thread, define a new process, then do Thread.Start(Procedure name).

Below is the list of actions involved in managing Multithread applications:



**Join :** This method makes the calling main thread to wait for the thread on which this method is called.

**Thread cannot be restarted** after it is stopped (aborted).

**Thread Pool:**<http://msdn.microsoft.com/en-US/library/3dasc8as(v=vs.80).aspx>

* A thread pool is a collection of threads that can be used to perform a number of tasks in the background. (See [Using Threading](http://msdn.microsoft.com/en-US/library/5xt1dysy(v=vs.80).aspx) for background information.) This leaves the primary thread free to perform other tasks asynchronously.
* Thread pools are often employed in server applications. Each incoming request is assigned to a thread from the thread pool, so the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.
* Once a thread in the pool completes its task, it is returned to a queue of waiting threads, where it can be reused. This reuse enables applications to avoid the cost of creating a new thread for each task.
* You can implement your own thread pool, but it is easier to use the thread pool provided by the .NET Framework through the [ThreadPool](http://msdn.microsoft.com/en-US/library/system.threading.threadpool(v=vs.80).aspx) class.

Use the call below to pick up a thread from thread pool:  
 Use an object ManualResetEvent for signaling purpose , set it false (saying work in continuation) so that the main thread knows that the thread is in continuation, by using WaitHandle.WaitAll

doneEvents[i] = new ManualResetEvent(false);  
ThreadPool.QueueUserWorkItem(“Name of a method”, i)

\_doneEvent.Set();--resets the Event saying works done

**Thread Synchronization:** This means controlling simultaneous access to any special part of code by multiple threads or also restrict access till it has to wait till some other thread is completed.

<http://www.yoda.arachsys.com/csharp/threads/waithandles.shtml>

|  |
| --- |
| * Use the events when you've got a thread that is waiting on one of or all of a number of events to do something. * Use the monitor if you want to restrict access to a data structure by limiting how many threads can access it. * Monitors usually protect a resource, whereas events tell you something's happening, like the application shutting down. * Mutex and Monitor or (lock) are same, mutex is kernel object hence costly should be used if synchronization is needed accor application executable boundary. * AutoResetEvent will only allow one single waiting thread to continue, it automatically resets when it passes event.WaitOne() (this keeps the thread halted till it gets the signal). A ManualResetEvent on the other hand will keep allowing threads, several at the same time even, to continue until you tell it to stop (Reset it). * Semaphore is same as Mutex, only thing it gives option to allow particular no of threads which can access the resource (associated code) simultaneously. |

**Simple blocking methods**

These wait for another thread to finish or for a period of time to elapse. Sleep, Join, and Task.Wait are simple blocking methods.

**Locking constructs**

These limit the number of threads that can perform some activity or execute a section of code at a time. Exclusive locking constructs are most common — these allow just one thread in at a time, and allow competing threads to access common data without interfering with each other. The standard exclusive locking constructs are [**lock**](http://www.albahari.com/threading/part2.aspx#_Locking) (Monitor.Enter/Monitor.Exit), [**Mutex**](http://www.albahari.com/threading/part2.aspx#_Mutex), and [SpinLock](http://www.albahari.com/threading/part5.aspx#_SpinLock_and_SpinWait). The nonexclusive locking constructs are [**Semaphore**](http://www.albahari.com/threading/part2.aspx#_Semaphore), [**SemaphoreSlim**](http://www.albahari.com/threading/part2.aspx#_Semaphore), and the [reader/writer locks](http://www.albahari.com/threading/part4.aspx#_Reader_Writer_Locks).

**Signaling constructs**

These allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling. There are two commonly used signaling devices: [event wait handles](http://www.albahari.com/threading/part2.aspx#_Signaling_with_Event_Wait_Handles) and Monitor’s [**Wait/Pulse**](http://www.albahari.com/threading/part4.aspx#_Signaling_with_Wait_and_Pulse) methods. Framework 4.0 introduces the [**CountdownEvent**](http://www.albahari.com/threading/part2.aspx#_CountdownEvent) and [**Barrier**](http://www.albahari.com/threading/part4.aspx#_The_Barrier_Class) classes.

**Nonblocking synchronization constructs**

These protect access to a common field by calling upon processor primitives. The CLR and C# provide the following nonblocking constructs: [**Thread.MemoryBarrier, Thread.VolatileRead, Thread.VolatileWrite**](http://www.albahari.com/threading/part4.aspx#_Memory_Barriers_and_Volatility), the [**volatile**](http://www.albahari.com/threading/part4.aspx#_The_volatile_keyword) keyword, and the [**Interlocked**](http://www.albahari.com/threading/part4.aspx#_Interlocked) class.

**Monitor Class:** The **Monitor** class controls access to objects by granting a lock for an object to a single thread.

* First use the “Lock” or “Monitor.Enter” keyword to lock access to ANY method of the class, then use Monitor.Pulse method to release the lock. Then in any of the methods of the class, which should be accessed at the same time as this method, use Monitor.Wait at the beginning, this will make the other thread wait till it gets signal from Monitor.Pulse.

**Mutex**When two or more threads need to access a shared resource at the same time, the system needs a synchronization mechanism to ensure that only one thread at a time uses the resource. Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread. If a thread acquires a mutex, the second thread that wants to acquire that mutex is suspended until the first thread releases the mutex. One can specify the timeinterval the thread has to wait. The instance **declaring** the mutex owns the mutex.

A **Mutex** is like a C# **lock**, but it can work across multiple processes. In other words, **Mutex** can be *computer-wide* as well as *application-wide*. It’s a kernel object hence expensive and should be used only when required across applications instances.

**Semaphores:** This also provides a locking mechanism for a particular piece of code, only difference is to restrict only Particualr no. of threads, say 5 threads from executing a piece of code, Any thread can call **Release** on a **Semaphore**, whereas with **Mutex** and **lock**, only the thread that obtained the lock can release it.

**Events** are used to For Signaling from one thread to other thread. For eg, if a long process is going in the background and it has to be cancelled, at the click of the cancel button, an event (Signal) can be raised the long process checks for the event (signal) if its raised it cancels or exits the long process.

An event is also a synchronization object. The difference with the mutex object is that you can explicitly set or reset the event with SetEvent() or ResetEvent(). Note that resetting the event may be automatic depending on the way you created the event. It is often used with thread to send a signal to a thread. Here is an implementation of the Event class:

The base class is “EventWaitHandle”, below are the two classes derived from this class.

Different types or events are:

* AutoReset Event
* ManualResetEvent

The most important difference is that an AutoResetEvent will only allow one single waiting thread to continue. A ManualResetEvent on the other hand will keep allowing threads, several at the same time even, to continue until you tell it to stop

Use the events when you've got a thread that is waiting on one of or all of a number of events to do something.

Use the monitor if you want to restrict access to a data structure by limiting how many threads can access it.

Monitors usually protect a resource, whereas events tell you something's happening, like the application shutting down.

**Mutex:** It is used to ensure that a resource at any time is accesed by only one thread.

For eg. Before accessing a file (which is to be accessed by only one thread at a time), create a mutex object

**What is a Critical Section?**

A critical section is very similar to mutex - it allows only one thread to access some protected data. However, these are the differences:

* Critical Section is a *User Object,* whereas Mutex is a *Kernel Object.* A user-object need no interference with the kernel (OS Core); a Kernel-object requires a round-trip to kernel-mode.
* Critical Sections are faster than Mutexes. This rule holds true for any User v/s Kernel object.
* Critical Sections **cannot be named** and **cannot be shared** among processes.
* User objects cannot have security attributes attached to them.
* You **cannot specify the timeout** while waiting on a critical section. That means *infinite* wait time-out!
* Some languages provide a direct and simple mechanism to use critical sections. For example, C# facilitates the lock keyword to *implicitly* use a critical section, which is more intuitive than using the Monitor class.

Static [Monitor](http://msdn.microsoft.com/en-us/library/system.threading.monitor.aspx) class, or [lock](http://msdn.microsoft.com/en-us/library/c5kehkcz(VS.80).aspx) keyword

**lock** ensures that one thread does not enter a critical section of code while another thread is in the critical section. If another thread attempts to enter a locked code, it will wait, block, until the object is released.

For eg: If there are two different threads accessing some crucial data such as Account Balance in case of banking software, using lock we can ensure that at any time only one thread accesses the or checks and debits the account balance, in short a Global variable which at any time should be accessed by one particular thread only.

**BackGround Worker: This is a control provided in windows components, used in Windows and WPF desktop appplications. The main thread in this case is refered to as UI thread. Using this control is not recommended if there is a constant communication required between various threads.**

[**http://www.dotnetperls.com/backgroundworker**](http://www.dotnetperls.com/backgroundworker)

[**http://elegantcode.com/2009/07/03/wpf-multithreading-using-the-backgroundworker-and-reporting-the-progress-to-the-ui/**](http://elegantcode.com/2009/07/03/wpf-multithreading-using-the-backgroundworker-and-reporting-the-progress-to-the-ui/)

[**http://www.nerdparadise.com/tech/csharp/backgroundworker/**](http://www.nerdparadise.com/tech/csharp/backgroundworker/)

**RunWorkerAsync** make the BackgroundWorker fire the DoWork event in a new thread, however, it's best to check the IsBusy property otherwise you'll get a crash if the worker as already been started.

**First, call RunWorkerAsync with an argument.** You can pass any argument to this method on BackgroundWorker, including null. It simply must inherit from object, which everything does.  
  
**Second, custom processing is run.** Your expensive code is executed in the DoWork method. Insert pause here as your program does its calculations.  
  
**Third, it finishes.** When your processing is done, RunWorkerCompleted is called. In this method, you receive the result. In this way, your BackgroundWorker object modifies an object on another thread, and you receive it when it is done.