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.

A Mutex is a synchronization primitive that can also be used for interprocess synchronization. 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.  
  
In short, A mutual exclusion ("Mutex") is a mechanism that acts as a flag to prevent two threads from performing one or more actions simultaneously. The entire action that you want to run exclusively is called a critical section or protected section.

A critical section is a piece of code that accesses a shared resource (data structure or device) but the condition is that only one thread can enter in this section at a time.

Modern programming languages support this natively. In C#, it's as simple as:

* Instantiating a new static Mutex object that's accessible from each thread.
* Wrapping whatever code you want to be executed in the critical section with that object's WaitOne() and ReleaseMutex() methods in each thread

With a Mutex class, you call the **WaitHandle.WaitOne**method to lock and ReleaseMutex to unlock. Closing or disposing a Mutex automatically releases it. Just as with the lock statement, a Mutex can be released only from the same thread that obtained it.

This example shows how a local Mutex object is used to synchronize access to a protected resource.