<https://stackoverflow.com/questions/30283262/task-vs-barrier>

Lock

Преобразуется в

var lockFlag = new object();

bool lockTaken = false;

try

{

System.Threading.Monitor.Enter(lockFlag, ref lockTaken);

//logic

}

finally

{

if (lockTaken)

System.Threading.Monitor.Exit(lockFlag);

}

# Lock vs Monitor vs Mutext vs Semaphore

**lock(obj)**

* is a CLR construct that for (intra-object?) thread synchronization. Ensures that only one thread can take ownership of the object's lock & enter the locked block of code. Other threads must wait till the current owner relinquishes the lock by exiting the block of code. Also it is recommended that you lock on a private member object of your class.

**Monitors**

* lock(obj) is implemented internally using a Monitor. You should prefer lock(obj) because it prevents you from goofing up like forgetting the cleanup procedure. It 'idiot-proof's the Monitor construct if you will.  
  Using Monitor is generally preferred over mutexes, because monitors were designed specifically for the .NET Framework and therefore make better use of resources.

Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but **these constructs do not allow one thread to communicate an event to another. This requires synchronization events**, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Mutex, Semaphores are OS-level concepts. e.g with a named mutex you could synchronize across multiple (managed) exes (ensuring that only one instance of your application is running on the machine.)

**Mutex:**

* Unlike monitors, however, **a mutex can be used to synchronize threads across processes.** When used for inter-process synchronization, a mutex is called a **named mutex** because it is to be used in another application, and therefore it cannot be shared by means of a global or static variable. It must be given a name so that both applications can access the same mutex object. In contrast, **the Mutex class is a wrapper to a Win32 construct. While it is more powerful than a monitor, a mutex requires interop transitions that are more computationally expensive than those required by the Monitor class.**

[**Semaphores**](http://msdn.microsoft.com/en-us/library/system.threading.semaphore.aspx) (hurt my brain).

* Use the Semaphore class to control access to a pool of resources. Threads enter the semaphore by calling the WaitOne method, which is inherited from the WaitHandle class, and release the semaphore by calling the Release method. The count on a semaphore is decremented each time a thread enters the semaphore, and incremented when a thread releases the semaphore. When the count is zero, subsequent requests block until other threads release the semaphore. When all threads have released the semaphore, the count is at the maximum value specified when the semaphore was created. **A thread can enter the semaphore multiple times..The Semaphore class does not enforce thread identity on WaitOne or Release.. programmers responsibility to not muck up.** Semaphores are of two types: local semaphores and named **system semaphores. If you create a Semaphore object using a constructor that accepts a name, it is associated with an operating-system semaphore of that name.** Named system semaphores are visible throughout the operating system, and can be used to synchronize the activities of processes. **A local semaphore exists only within your process. It can be used by any thread in your process that has a reference to the local Semaphore object. Each Semaphore object is a separate local semaphore.**

[THE PAGE TO READ - Thread Synchronization (C#)](http://msdn.microsoft.com/en-us/library/ms173179(VS.80).aspx)