mutex.cm

```
/*
    ================================================================

    Copyright (c) 2012-2016 Seppo Laakko
    http://sourceforge.net/projects/cmajor/

    Distributed under the GNU General Public License, version 3 (GPLv3).
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html
        )

    ================================================================

    */

// Copyright (c) 1994
// Hewlett-Packard Company
// Copyright (c) 1996
// Silicon Graphics Computer Systems, Inc.
// Copyright (c) 2009 Alexander Stepanov and Paul McJones

using System;

namespace System.Threading
{
    public class Mutex
    {
        public Mutex(): handle(allocate_mutex_handle())
        {
            int result = init_mutex(handle);
            if (result != 0)
            {
                string reason = strerror(result);
                throw ThreadingException("could not initialize a mutex",
                    reason);
            }
        }
        internal Mutex(bool): handle(allocate_mutex_handle()), recursive(
            true)
        {
            int result = init_recursive_mutex(handle);
            if (result != 0)
            {
                string reason = strerror(result);
                throw ThreadingException("could not initialize a
                    recursive mutex", reason);
            }
        }
        public ~Mutex()
        {
            int result = destroy_mutex(handle);
            free_mutex_handle(handle);
        }
```

```
        public mutex_t* Handle() const
        {
            return handle;
        }
        suppress Mutex(const Mutex&);
        suppress void operator=(const Mutex&);
        suppress Mutex(Mutex&&);
        suppress void operator=(Mutex&&);
        public void Lock()
        {
            int result = lock_mutex(handle);
            if (result != 0)
            {
                string reason = strerror(result);
                if (recursive)
                {
                    throw ThreadingException("could not lock a recursive
                        mutex", reason);
                }
                else
                {
                    throw ThreadingException("could not lock a mutex",
                        reason);
                }
            }
        }
        public void Unlock()
        {
            int result = unlock_mutex(handle);
            if (result != 0)
            {
                string reason = strerror(result);
                if (recursive)
                {
                    throw ThreadingException("could not unlock a
                        recursive mutex", reason);
                }
                else
                {
                    throw ThreadingException("could not unlock a mutex",
                        reason);
                }
            }
        }
        public bool TryLock()
        {
            bool already_locked = false;
            int result = trylock_mutex(handle, already_locked);
            if (result != 0)
            {
                string reason = strerror(result);
                if (recursive)
                {
```

```
                    throw ThreadingException("could not trylock a
                        recursive mutex", reason);
                }
                else
                {
                    throw ThreadingException("could not trylock a mutex",
                        reason);
                }
            }
            return already_locked;
        }
        private mutex_t* handle;
        private bool recursive;
    }

    public class RecursiveMutex: Mutex
    {
        public RecursiveMutex(): base(true)
        {
        }
        suppress RecursiveMutex(const RecursiveMutex&);
        suppress void operator=(const RecursiveMutex&);
        suppress RecursiveMutex(RecursiveMutex&&);
        suppress void operator=(RecursiveMutex&&);
    }

    public concept Lockable<M>
    {
        void M.Lock();
        void M.Unlock();
    }

    public class LockGuard<M> where M is Lockable
    {
        public LockGuard(M& m_): m(m_)
        {
            m.Lock();
        }
        public ~LockGuard()
        {
            m.Unlock();
        }
        public nothrow M& GetLock()
        {
            return m;
        }
        private M& m;
    }
}
```