The C++ RecMutex Class

A non-recursive mutex cannot be locked more than once, even by the thread that holds the lock. This frequently becomes a problem if a program contains a number of functions, each of which must acquire a mutex, and you want to call one function as part of the implementation of another function:

C++

```
IceUtil::Mutex _mutex;

void
f1()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                               // Deadlock!

    // More code here...
}
```

`f1` and `f2` each correctly lock the mutex before manipulating data but, as part of its implementation, `f2` calls `f1`. At that point, the program deadlocks because `f2` already holds the lock that `f1` is trying to acquire. For this simple example, the problem is obvious. However, in complex systems with many functions that acquire and release locks, it can get very difficult to track down this kind of situation: the locking conventions are not manifest anywhere but in the source code and each caller must know which locks to acquire (or not to acquire) before calling a function. The resulting complexity can quickly get out of hand.

Ice provides a recursive mutex class `RecMutex` (defined in `IceUtil/RecMutex.h`) that avoids this problem:

C++

```
namespace IceUtil {

    class RecMutex {
    public:
        RecMutex();
        RecMutex(MutexProtocol p);
        ~RecMutex();

        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<RecMutex> Lock;
        typedef TryLockT<RecMutex> TryLock;
    };
}
```

Note that the signatures of the operations are the same as for `IceUtil::Mutex`. However, `RecMutex` implements a recursive mutex:

- `RecMutex()`
  `RecMutex(MutexProtocol p)`
  You can optionally specify a mutex protocol when you construct a mutex. The mutex protocol controls how the mutex behaves with respect to thread priorities. Default-constructed mutexes use a system-wide default.

- `lock`
  The `lock` function attempts to acquire the mutex. If the mutex is already locked by another thread, it suspends the calling thread until the mutex becomes available. If the mutex is available or is already locked by the calling thread, the call returns immediately with the mutex locked.

- `tryLock`
  The `tryLock` function works like `lock`, but, instead of blocking the caller, it returns false if the mutex is locked by another thread. Otherwise, the return value is true.

- `unlock`
  The `unlock` function unlocks the mutex.

As for non-recursive mutexes, you must adhere to a few simple rules for recursive mutexes:

- Do not call `unlock` on a mutex unless the calling thread holds the lock.
- You must call `unlock` as many times as you called `lock` for the mutex to become available to another thread. (Internally, a recursive mutex is implemented with a counter that is initialized to zero. Each call to `lock` increments the counter and each call to `unlock` decrements the counter; the mutex is made available to another thread when the counter returns to zero.)

Using recursive mutexes, the code fragment shown earlier works correctly:

C++

```
#include <IceUtil/RecMutex.h>
// ...

IceUtil::RecMutex _mutex;       // Recursive mutex

void
f1()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                               // Fine

    // More code here...
}
```

Note that the type of the mutex is now `RecMutex` instead of `Mutex`, and that we are using the `Lock` type definition provided by the `RecMutex` class, not the one provided by the `Mutex` class.

See Also

- The C++ Mutex Class
- Priority Inversion in C++