The C++ Mutex Class

This page describes how to use mutexes — one of the available synchronization primitives.

On this page:

Mutex Member Functions

The class `IceUtil::Mutex` (defined in `IceUtil/Mutex.h`) provides a simple non-recursive mutual exclusion mechanism:

C++

```
namespace IceUtil {
    enum MutexProtocol { PrioInherit, PrioNone };

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

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

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

The member functions of this class work as follows:

- `Mutex()`
  `Mutex(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, it suspends the calling thread until the mutex becomes available. The call returns once the calling thread has acquired the mutex.

- `tryLock`
  The `tryLock` function attempts to acquire the mutex. If the mutex is available, the call returns true with the mutex locked. Otherwise, if the mutex is locked by another thread, the call returns false.

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

Note that `IceUtil::Mutex` is a non-recursive mutex implementation. This means that you must adhere to the following rules:

- Do not call `lock` on the same mutex more than once from a thread. The mutex is not recursive so, if the owner of a mutex attempts to lock it a second time, the behavior is undefined.

- Do not call `unlock` on a mutex unless the calling thread holds the lock. Calling `unlock` on a mutex that is not currently held by any thread, or calling `unlock` on a mutex that is held by a different thread, results in undefined behavior.

Use the `IceUtil::RecMutex` class if you need recursive semantics.

Adding Thread Safety to the File System Application in C++

Recall that the implementation of the `read` and `write` operations for our file system server is not thread safe:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    return _lines;      // Not thread safe!
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
    _lines = text;      // Not thread safe!
}
```

The problem here is that, if we receive concurrent invocations of `read` and `write`, one thread will be assigning to the `_lines` vector while another thread is reading that same vector. The outcome of such concurrent data access is undefined; to avoid the problem, we need to serialize access to the `_lines` member with a mutex. We can make the mutex a data member of the `FileI` class and lock and unlock it in the `read` and `write` operations:

C++

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

namespace Filesystem {
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::Mutex _fileMutex;
    };
    // ...
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    _fileMutex.lock();
    Lines l = _lines;
    _fileMutex.unlock();
    return l;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice::Current&)
{
    _fileMutex.lock();
    _lines = text;
    _fileMutex.unlock();
}
```

The `FileI` class here is identical to the original implementation, except that we have added the `_fileMutex` data member. The `read` and `write` operations lock and unlock the mutex to ensure that only one thread can read or write the file at a time. Note that, by using a separate mutex for each `FileI` instance, it is still possible for multiple threads to concurrently read or write files, as long as they each access a *different* file. Only concurrent accesses to the *same* file are serialized.

The implementation of `read` is somewhat awkward here: we must make a local copy of the file contents while we are holding the lock and return that copy. Doing so is necessary because we must unlock the mutex before we can return from the function. However, as we will see in the next section, the copy can be avoided by using a helper class that unlocks the mutex automatically when the function returns.

Guaranteed Unlocking of Mutexes in C++

Using the raw `lock` and `unlock` operations on mutexes has an inherent problem: if you forget to unlock a mutex, your program will deadlock. Forgetting to unlock a mutex is easier than you might suspect, for example:

C++

```
Filesystem::Lines
Filesystem::File::read(const Ice::Current&) const
{
    _fileMutex.lock();                  // Lock the mutex
    Lines l = readFileContents();       // Read from database
    _fileMutex.unlock();                // Unlock the mutex
    return l;
}
```

Assume that we are keeping the contents of the file on secondary storage, such as a database, and that the `readFileContents` function accesses the file. The code is almost identical to the previous example but now contains a latent bug: if `readFileContents` throws an exception, the `read` function terminates without ever unlocking the mutex. In other words, this implementation of `read` is not exception-safe.

The same problem can easily arise if you have a larger function with multiple return paths. For example:

C++

```
void
SomeClass::someFunction(/* params here... */)
{
    _mutex.lock();                      // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                         // Oops!!!
    }

    // More code here...

    _mutex.unlock();                    // Unlock the mutex
}
```

In this example, the early return from the middle of the function leaves the mutex locked. Even though this example makes the problem quite obvious, in large and complex pieces of code, both exceptions and early returns can cause hard-to-track deadlock problems. To avoid this, the `Mutex` class contains two type definitions for helper classes, called `Lock` and `TryLock`:

C++

```
namespace IceUtil {

    class Mutex {
        // ...

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

`LockT` and `TryLockT` are simple templates that primarily consist of a constructor and a destructor; the `LockT` constructor calls `lock` on its argument, and the `TryLockT` constructor calls `tryLock` on its argument. The destructors call `unlock` if the mutex is locked when the template goes out of scope. By instantiating a local variable of type `Lock` or `TryLock`, we can avoid the deadlock problem entirely:

C++

```
void
SomeClass::someFunction(/* params here... */)
{
    IceUtil::Mutex::Lock lock(_mutex);  // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                         // No problem
    }

    // More code here...

}   // Destructor of lock unlocks the mutex
```

This is an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [1].

On entry to `someFunction`, we instantiate a local variable `lock`, of type `IceUtil::Mutex::Lock`. The constructor of `lock` calls `lock` on the mutex so the remainder of the function is inside a critical region. Eventually, `someFunction` returns, either via an ordinary return (in the middle of the function or at the end) or because an exception was thrown somewhere in the function body. Regardless of how the function terminates, the C++ run time unwinds the stack and calls the destructor of `lock`, which unlocks the mutex, so we cannot get trapped by the deadlock problem we had previously.

Both the `Lock` and `TryLock` templates have a few member functions:

- `void acquire() const`
  This function attempts to acquire the lock and blocks the calling thread until the lock becomes available. If the caller calls `acquire` on a mutex it has locked previously, the function throws `ThreadLockedException`.
- `bool tryAcquire() const`
  This function attempts to acquire the mutex. If the mutex can be acquired, it returns true with the mutex locked; if the mutex cannot be acquired, it returns false. If the caller calls `tryAcquire` on a mutex it has locked previously, the function throws `ThreadLockedException`
- `void release() const`
  This function releases a previously locked mutex. If the caller calls release on a mutex it has unlocked previously, the function throws `ThreadLockedException`.
- `bool acquired() const`
  This function returns true if the caller has locked the mutex previously, otherwise it returns false. If you use the `TryLock` template, you must call `acquired` after instantiating the template to test whether the lock actually was acquired.

These functions are useful if you want to use the `Lock` and `TryLock` templates for guaranteed unlocking, but need to temporarily release the lock:

C++

```
{
    IceUtil::Mutex::TryLock m(someMutex);

    if (m.acquired())
    {

        // Got the lock, do processing here...

        if (release_condition) {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.
        // ...

        m.acquire(); // Block until mutex becomes available.

        // ...

        if (release_condition) {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.

        // ...

        // Spin on the mutex until it becomes available.
        while (!m.tryLock()) {
            // Do some other processing here...
        }

        // Mutex locked again at this point.

        // ...
    }

} // Close scope, m is unlocked by its destructor.
```

Tip

You should make it a habit to always use the `Lock` and `TryLock` helpers instead of calling `lock` and `unlock` directly. Doing so results in code that is easier to understand and maintain.

Using the `Lock` helper, we can rewrite the implementation of our `read` and `write` operations as follows:

C++

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    _lines = text;
}
```

Note that this also eliminates the need to make a copy of the `_lines` data member: the return value is initialized under protection of the mutex and cannot be modified by another thread once the destructor of `lock` unlocks the mutex.

See Also

- Example of a File System Server in C++
- Priority Inversion in C++
- The C++ RecMutex Class

References

1. Stroustrup, B. 1997. *The C++ Programming Language*. Reading, MA: Addison-Wesley.