The C++ Monitor Class

The recursive and non-recursive mutex classes implement a simple mutual exclusion mechanism that allows only a single thread to be active in a critical region at a time. In particular, for a thread to enter the critical region, another thread must leave it. This means that, with mutexes, it is impossible to suspend a thread inside a critical region and have that thread wake up again at a later time, for example, when a condition becomes true.

To address this problem, Ice provides a monitor. Briefly, a monitor is a synchronization mechanism that protects a critical region: as for a mutex, only one thread may be active at a time inside the critical region. However, a monitor allows you to suspend a thread inside the critical region; doing so allows another thread to enter the critical region. The second thread can either leave the monitor (thereby unlocking the monitor), or it can suspend itself inside the monitor; either way, the original thread is woken up and continues execution inside the monitor. This extends to any number of threads, so several threads can be suspended inside a monitor.

The monitors provided by Ice have *Mesa* semantics, so called because they were first implemented by the Mesa programming language [1] . Mesa monitors are provided by a number of languages, including Java and Ada. With Mesa semantics, the signalling thread continues to run and another thread gets to run only once the signalling thread suspends itself or leaves the monitor.

Monitors provide a more flexible mutual exclusion mechanism than mutexes because they allow a thread to check a condition and, if the condition is false, put itself to sleep; the thread is woken up by some other thread that has changed the condition.

On this page:

Monitor Member Functions

Ice provides monitors with the `IceUtil::Monitor` class (defined in `IceUtil/Monitor.h`):

C++

```
namespace IceUtil {

    template <class T>
    class Monitor {
    public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;

        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();

        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

Note that `Monitor` is a template class that requires either `Mutex` or `RecMutex` as its template parameter. (Instantiating a `Monitor` with a `RecMutex` makes the monitor recursive.)

The member functions behave as follows:

- `lock`
  This function attempts to lock the monitor. If the monitor is currently locked by another thread, the calling thread is suspended until the monitor becomes available. The call returns with the monitor locked.

- `tryLock`
  This function attempts to lock a monitor. If the monitor is available, the call returns true with the monitor locked. If the monitor is locked by another thread, the call returns false.

- `unlock`
  This function unlocks a monitor. If other threads are waiting to enter the monitor (are blocked inside a call to `lock`), one of the threads is woken up and locks the monitor.

- `wait`
  This function suspends the calling thread and, at the same time, releases the lock on the monitor. A thread suspended inside a call to `wait` can be woken up by another thread that calls `notify` or `notifyAll`. When the call returns, the suspended thread resumes execution with the monitor locked.

- `timedWait`
  This function suspends the calling thread for up to the specified timeout. If another thread calls `notify` or `notifyAll` and wakes up the suspended thread before the timeout expires, the call returns true and the suspended thread resumes execution with the monitor locked. Otherwise, if the timeout expires, the function returns false. Wait intervals are represented by instances of the `Time` class.

- `notify`
  This function wakes up a single thread that is currently suspended in a call to `wait` or `timedWait`. If no thread is suspended in a call to `wait` or `timedWait` at the time `notify` is called, the notification is lost (that is, calls to `notify` are *not* remembered if there is no thread to be woken up). Note that notifying does not run another thread immediately. Another thread gets to run only once the notifying thread either calls `wait` or `timedWait` or unlocks the monitor (Mesa semantics).

- `notifyAll`
  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `notify`, calls to `notifyAll` are lost if no threads are suspended at the time. Also as for `notify`, `notifyAll` causes other threads to run only once the notifying thread has either called `wait` or `timedWait` or unlocked the monitor (Mesa semantics).

You must adhere to a few rules for monitors to work correctly:

- Do not call `unlock` unless you hold the lock. If you instantiate a monitor with a recursive mutex, you get recursive semantics, that is, you must call `unlock` as many times as you have called `lock` (or `tryLock`) for the monitor to become available.
- Do not call `wait` or `timedWait` unless you hold the lock.
- Do not call `notify` or `notifyAll` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding (as shown below).

Using Monitors in C++

To illustrate how to use a monitor, consider a simple unbounded queue of items. A number of producer threads add items to the queue, and a number of consumer threads remove items from the queue. If the queue becomes empty, consumers must wait until a producer puts a new item on the queue. The queue itself is a critical region, that is, we cannot allow a producer to put an item on the queue while a consumer is removing an item. Here is a very simple implementation of a such a queue:

C++

```
template<class T> class Queue {
public:
    void put(const T& item) {
        _q.push_back(item);
    }

    T get() {
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

As you can see, producers call the `put` method to enqueue an item, and consumers call the `get` method to dequeue an item. Obviously, this implementation of the queue is not thread-safe and there is nothing to stop a consumer from attempting to dequeue an item from an empty queue.

Here is a version of the queue that uses a monitor to suspend a consumer if the queue is empty:

C++

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

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

Note that the `Queue` class now inherits from `IceUtil::Monitor<IceUtil::Mutex>`, that is, `Queue` *is-a* monitor.

Both the `put` and `get` methods lock the monitor when they are called. As for mutexes, instead of calling `lock` and `unlock` directly, we are using the `Lock` helper which automatically locks the monitor when it is instantiated and unlocks the monitor again when it is destroyed.

The `put` method first locks the monitor and then, now being in sole possession of the critical region, enqueues an item. Before returning (thereby unlocking the monitor), `put` calls `notify`. The call to `notify` will wake up any consumer thread that may be asleep in a `wait` call to inform the consumer that an item is available.

The `get` method also locks the monitor and then, before attempting to dequeue an item, tests whether the queue is empty. If so, the consumer calls `wait`. This suspends the consumer inside the `wait` call and unlocks the monitor, so a producer can enter the monitor to enqueue an item. Once that happens, the producer calls `notify`, which causes the consumer's `wait` call to complete, with the monitor again locked for the consumer. The consumer now dequeues an item and returns (thereby unlocking the monitor).

For this machinery to work correctly, the implementation of `get` does two things:

- `get` tests whether the queue is empty *after* acquiring the lock.
- `get` re-tests the condition in a loop around the call to `wait`; if the queue is still empty after `wait` returns, the `wait` call is re-entered.

You *must* always write your code to follow the same pattern:

- *Never* test a condition unless you hold the lock.
- *Always* re-test the condition in a loop around `wait`. If the test still shows the wrong outcome, call `wait` again.

Not adhering to these conditions will eventually result in a thread accessing shared data when it is not in its expected state, for the following reasons:

1. If you test a condition without holding the lock, there is nothing to prevent another thread from entering the monitor and changing its state before you can acquire the lock. This means that, by the time you get around to locking the monitor, the state of the monitor may no longer be in agreement with the result of the test.
2. Some thread implementations suffer from a problem known as *spurious wake-up*: occasionally, more than one thread may wake up in response to a call to `notify`, or a thread may wake up without any call to `notify` at all. As a result, each thread that returns from a call to `wait` must re-test the condition to ensure that the monitor is in its expected state: the fact that `wait` returns does *not* indicate that the condition has changed.

Efficient Notification using Monitors in C++

The previous implementation of our thread-safe queue unconditionally notifies a waiting reader whenever a writer deposits an item into the queue. If no reader is waiting, the notification is lost and does no harm. However, unless there is only a single reader and writer, many notifications will be sent unnecessarily, causing unwanted overhead.

Here is one way to fix the problem:

C++

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

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() == 1)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

The only difference between this code and the implementation shown earlier is that a writer calls `notify` only if the queue length has just changed from empty to non-empty. That way, unnecessary `notify` calls are never made. However, this approach works only for a single reader thread. To see why, consider the following scenario:

1. Assume that the queue currently contains a number of items and that we have five reader threads.
2. The five reader threads continue to call `get` until the queue becomes empty and all five readers are waiting in `get`.
3. The scheduler schedules a writer thread. The writer finds the queue empty, deposits an item, and wakes up a single reader thread.
4. The awakened reader thread dequeues the single item on the queue.
5. The reader calls `get` a second time, finds the queue empty, and goes to sleep again.

The net effect of this is that there is a good chance that only one reader thread will ever be active; the other four reader threads end up being permanently asleep inside the `get` method.

One way around this problem is call `notifyAll` instead of `notify` once the queue length exceeds a certain amount, for example:

C++

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

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() >= _wakeupThreshold)
            notifyAll();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    const int _wakeupThreshold = 100;
};
```

Here, we have added a private data member `_wakeupThreshold`; a writer wakes up *all* waiting readers once the queue length exceeds the threshold, in the expectation that all the readers will consume items more quickly than they are produced, thereby reducing the queue length below the threshold again.

This approach works, but has drawbacks as well:

- The appropriate value of `_wakeupThreshold` is difficult to determine and sensitive to things such as speed and number of processors and I/O bandwidth.
- If multiple readers are asleep, they are all made runnable by the thread scheduler once a writer calls `notifyAll`. On a multiprocessor machine, this may result in all readers running at once (one per CPU). However, as soon as the readers are made runnable, each of them attempts to reacquire the mutex that protects the monitor before returning from `wait`. Of course, only one of the readers actually succeeds and the remaining readers are suspended again, waiting for the mutex to become available. The net result is a large number of thread context switches as well as repeated and unnecessary locking of the system bus.

A better option than calling `notifyAll` is to wake up waiting readers one at a time. To do this, we keep track of the number of waiting readers and call `notify` only if a reader needs to be woken up:

C++

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

template<class T> class Queue : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    Queue() : _waitingReaders(0) {}

    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_waitingReaders)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0) {
            try {
                ++_waitingReaders;
                wait();
                --_waitingReaders;
            } catch (...) {
                --_waitingReaders;
                throw;
            }
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    short _waitingReaders;
};
```

This implementation uses a member variable `_waitingReaders` to keep track of the number of readers that are suspended. The constructor initializes the variable to zero and the implementation of `get` increments and decrements the variable around the call to `wait`. Note that these statements are enclosed in a `try-catch` block; this ensures that the count of waiting readers remains accurate even if `wait` throws an exception. Finally, `put` calls `notify` only if there is a waiting reader.

The advantage of this implementation is that it minimizes contention on the monitor mutex: a writer wakes up only a single reader at a time, so we do not end up with multiple readers simultaneously trying to lock the mutex. Moreover, the monitor `notify` implementation signals a waiting thread only *after* it has unlocked the mutex. This means that, when a thread wakes up from its call to `wait` and tries to reacquire the mutex, the mutex is likely to be unlocked. This results in more efficient operation because acquiring an unlocked mutex is typically very efficient, whereas forcefully putting a thread to sleep on a locked mutex is expensive (because it forces a thread context switch).

See Also

- The C++ Mutex Class
- The C++ RecMutex Class
- The C++ Time Class

References

1. Mitchell, J. G., et al. 1979. *Mesa Language Manual*. CSL-793. Palo Alto, CA: Xerox PARC.