The C++ Cond Class

Condition variables are similar to monitors in that they allow a thread to enter a critical region, test a condition, and sleep inside the critical region while releasing its lock. Another thread then is free to enter the critical region, change the condition, and eventually signal the sleeping thread, which resumes at the point where it went to sleep and with the critical region once again locked.

Note that condition variables provide a subset of the functionality of monitors, so a monitor can always be used instead of a condition variable. However, condition variables are smaller, which may be important if you are seriously constrained with respect to memory.

Condition variables are provided by the `IceUtil::Cond` class. Here is its interface:

C++

```
class Cond : private noncopyable {
public:

    Cond();
    ~Cond();

    void signal();
    void broadcast();

    template<typename Lock>
    void wait(const Lock& lock) const;

    template<typename Lock>
    bool timedWait(const Lock& lock, const Time& timeout) const;
};
```

Using a condition variable is very similar to using a monitor. The main difference in the `Cond` interface is that the `wait` and `timedWait` member functions are template functions, instead of the entire class being a template. The member functions behave as follows:

- `wait`
  This function suspends the calling thread and, at the same time, releases the lock of the condition variable. A thread suspended inside a call to `wait` can be woken up by another thread that calls `signal` or `broadcast`. When `wait` completes, the suspended thread resumes execution with the lock held.

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

- `signal`
  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 `signal` is called, the signal is lost (that is, calls to `signal` are *not* remembered if there is no thread to be woken up). Note that signalling does not necessarily run another thread immediately; the thread calling `signal` may continue to run. However, depending on the underlying thread library, `signal` may also cause an immediate context switch to another thread.

- `broadcast`
  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `signal`, calls to `broadcast` are lost if no threads are suspended at the time.

You must adhere to a few rules for condition variables to work correctly:

- Do not call `wait` or `timedWait` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding, just as for a monitor.

In contrast to monitors, which require you to call `notify` and `notifyAll` with the lock held, condition variables permit you to call `signal` and `broadcast` without holding the lock. Here is a code example that changes a condition and signals on a condition variable:

C++

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

     c.signal();

     // ...
} // m is unlocked here
```

This code is correct and will work as intended, but it is potentially inefficient. Consider the code executed by the waiting thread:

C++

```
{
    Mutex::Lock sync(m);

    while(!condition) {
        c.wait(sync);
    }

    // Condition is now true, do some processing...

} // m is unlocked here
```

Again, this code is correct and will work as intended. However, consider what can happen once the first thread calls `signal`. It is possible that the call to `signal` will cause an immediate context switch to the waiting thread. But, even if the thread implementation does not cause such an immediate context switch, it is possible for the signalling thread to be suspended after it has called `signal`, but before it unlocks the mutex `m`. If this happens, the following sequence of events occurs:

1. The waiting thread is still suspended inside the implementation of `wait` and is now woken up by the call to `signal`.
2. The now-awake thread tries to acquire the mutex `m` but, because the signalling thread has not yet released the mutex, is suspended again waiting for the mutex to be unlocked.
3. The signalling thread is scheduled again and leaves the scope enclosing `sync`, which unlocks the mutex, making the thread waiting for the mutex runnable.
4. The thread waiting for the mutex acquires the mutex and retests its condition.

While the preceding scenario is functionally correct, it is inefficient because it incurs two extra context switches between the signalling thread and the waiting thread. Because context switches are expensive, this can have quite a large impact on run-time performance, especially if the critical region is small and the condition changes frequently.

You can avoid the inefficiency by unlocking the mutex *before* calling signal:

C++

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

} // m is unlocked here

c.signal(); // Signal with the lock available
```

By arranging the code as shown, you avoid the additional context switches because, when the waiting thread is woken up by the call to `signal`, it succeeds in acquiring the mutex before returning from `wait` without being suspended and woken up again first.

As for monitors, you should exercise caution in using `broadcast`, particularly if you have many threads waiting on a condition. Condition variables suffer from the same potential problem as monitors with respect to `broadcast`, namely, that all threads that are currently suspended inside `wait` can immediately attempt to acquire the mutex, but only one of them can succeed and all other threads are suspended again. If your application is sensitive to this condition, you may want to consider waking threads in a more controlled manner.

See Also

- The C++ Monitor Class
- The C++ Time Class