ACE Readers/Writer Lock

We have used the readers/writer lock mechanism, and believe me, it wasn't fun at all. We have wasted a week of our lives investigating into this issue, eventually learning two things:

1. The ACE guys implemented it right.
2. The documentation, well, could be improved (mildly put).

Just to make sure we did not miss any important piece of documentation, here is what we have read:

- http://www.cs.wustl.edu/~schmidt/PDF/dispatching.pdf - a general paper about dispatching mechanisms, that describes the context, problem, forces, solution and consequences of readers/writer lock. Does not get too sepecific about the implementation and usage of the readers/writer lock in ACE.
- http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf - describes the Double-Checked Locking pattern, which you will probably need to use.
- http://www.cs.wustl.edu/~schmidt/PDF/locking-patterns.pdf - does not deal with the readers/writer lock directly. Still worth reading.
- http://www.cs.wustl.edu/~schmidt/ACE/book1/ - the C++NPv1 book includes a short chapter about the ACE Synchronization Wrapper Facades, which contains a 3-page section about the readers/writer lock, out of which two pages are dedicated to an example. The remaining page does not include a thorough description of this mechanism.
- http://doc.ece.uci.edu/Doxygen/Stable/html/ace/class_ACE_RW_Mutex.html - the Doxygen documentation assumes you understand the rationale of the rw-lock, and it has two major flaws. First - the section that deals with `tryacquire_write_upgrade()` is ambiguous at best, and outright erroneous otherwise. Second, there is no mention of the consqeunces of any non-trivial state transition.

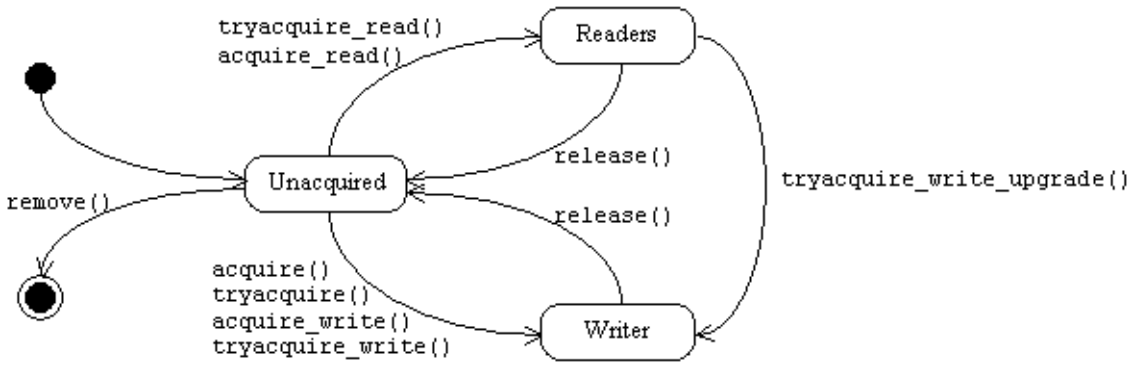
So here is what we would have liked to find in the documentation, a short how-to on readers/writer lock usage.

Applicability

We have used the ACE 5.2 RW-lock under Win32. Since this mechanism is not natively supported under this OS, ACE emulates the RW behavior by using other synchronization mechanisms (such as semaphore). The following discussion applies to the ACE emulation implementation, and may not apply to other systems where it probably wraps an existing mechanism, thus inheriting its semantics.

Readers/Writer Lock States

From a client's point of view, the RW-lock has the following states and actions.



Notes:

- Transition between states can occur through calls to the methods accompanying the arrows. However, a call to a method does not always cause a state transition, specifically in the

`tryacquire_*` methods.

- You must be careful to follow only the state transition paths that are depicted above. Not using the lock in this manner will eventually cause some problem. Some examples of bad usage:
  - Calling `relase()` different amount of times than acquiring the lock - The ACE emulation has an internal ref_count, which is incremented each time the lock is acquired, and decremented each time `release()` is called. It has no way of knowing which thread is calling `release()`, or to validate that this thread actually acquired the lock beforehand (and did not already release it).
  - Calling `tryacquire_write_upgrade()` without having a read lock first.
  - Calling any method after `remove()`.
  - Calling `acquire_write()` after `acquire_read()` by the same thread.
  - Calling `acquire_read()` after `acquire_write()` by the same thread.

Use of `tryacquire_write_upgrade()`

The `tryacquire_write_upgrade()` method's name is somewhat misleading. Although it suggests a non-blocking operation, the method sometimes blocks while trying to acquire the write upgrade. Why is that? Let's investigate a hypothetical `acquire_write_upgrade()` function – a blocking method much like `acquire_write()`. But there is a fundemental difference between these two methods: `acquire_write_upgrade()` may fail, while `acquire_write()` can only succeed (possibly after being blocked).

This becomes clear when you think about two threads holding a read lock, that are now calling `acquire_write_upgrade()` concurrently: One of them should be upgraded, but only after the other one releases its read lock - since a write lock allows no readers. For that to happen, the "winning" thread should block, while the "losing" thread's request should fail without blocking, and let the "loser" politely release its read lock. If both threads block, a deadlock is created, since nothing would cause any of them to release its lock.

So the `acquire_write_upgrade()` method must return with a fail value at least to one of the threads. More so, it may do this immediately, as it already knows that there is yet another thread waiting for an upgrade. Therefore, this hypothetical method is actually implemented with its name preceded by *try*: `tryacquire_write_upgrade()`.

Now what about the "winning" thread? It is waiting for all other read-lock holders to `release()` their locks, and when it is the only one left, it is promoted to an exclusive write lock, just as if it has called `acquire_write()`. Still, this is performed with one difference: the upgrading thread becomes an **important writer** - i.e., it has a higher priority than any other threads that are awaiting to acquire any lock at this point in time. This is again due to the nature of the upgrade state transition: The requesting thread must already own a read lock, so there is no theoretical possibility of giving any other thread a write lock first.

So what happens to new calls to `acquire_write()` or `acquire_read()` while there is a thread waiting for an upgrade? They will block until the upgrading thread accepts and releases its lock.

Another point worth mentioning is that the ACE rw lock mechanism was designed in such a manner that it gives preference to writers (and upgraders too). This means that threads calling `acquire_read()` will get their lock only after all threads that called `acquire_write()` and `tryacquire_write_upgrade()` have accepted and released their locks. (This is useful especailly when the ratio of writers/readers is small.) This is true even in the presence of an upgrading thread, although it may create an awkward situation: Thread X requests an upgrade while there are other threads owning a read lock, so it gets blocked; Another thread Y calls `acquire_read()` and is also blocked until thread X releases the write lock it did not yet get; All other threads call `release()`, which eventually yields the write upgrade to X; Then X calls `release()`, and Y finally gets its chance. Strange, but reasoanble.

A short summary:

- The thread calling `tryacquire_write_upgrade()` must already own a read lock.
- Only one thread can succeed in an upgrade. This thread becomes blocked until it acquires the write lock, while other threads trying to upgrade fail the request immediately.
- All failing threads must `release()` their read lock to avoid deadlock.
- While a thread is blocked waiting for an upgrade, any request (acquire, not tryacquire) by another thread for a read lock or write lock would also block, and would be unblocked only after the upgrader has released its lock.

A Suggestion

We found the following suggestion quite useful: Develop an Owned RW Lock. This flavor of the RW lock records the lock's state as it is viewed by a specific client (hence "owned"), and validates the entry conditions so it can prevent harmful operations.

```
class Owned_Thread_RW_Mutex : public ACE_RW_Thread_Mutex
{
public:
    Owned_Thread_RW_Mutex () :
        ACE_RW_Thread_Mutex (), _status (RW_LOCK_UNACQUIRED) {}
    virtual ~Owned_Thread_RW_Mutex () { this->remove (); }

    virtual int remove () {
        if (_status == RW_LOCK_REMOVED)     return -1;
        this->release ();
        _status = RW_LOCK_REMOVED;
        return ACE_RW_Thread_Mutex::remove ();
    }
    virtual int release () {
        if (_status == RW_LOCK_UNACQUIRED)  return  0;
        if (_status == RW_LOCK_REMOVED)     return -1;
        _status = RW_LOCK_UNACQUIRED;
        return ACE_RW_Thread_Mutex::release ();
    }
    virtual int acquire () { return this->acquire_write (); }
    virtual int acquire_write () {
        if (_status == RW_LOCK_WRITE)       return  0;
        if (_status != RW_LOCK_UNACQUIRED)  return -1;
        int result = ACE_RW_Thread_Mutex::acquire_write ();
        if (result == 0) _status = RW_LOCK_WRITE;
        return result;
    }
    virtual int acquire_read () {
        if (_status == RW_LOCK_READ)        return  0;
        if (_status != RW_LOCK_UNACQUIRED)  return -1;
        int result = ACE_RW_Thread_Mutex::acquire_read ();
        if (result == 0) _status = RW_LOCK_READ;
        return result;
    }
    virtual int tryacquire () { return this->tryacquire_write (); }
    virtual int tryacquire_read () {
        if (_status == RW_LOCK_READ)        return  0;
        if (_status != RW_LOCK_UNACQUIRED)  return -1;
        int result = ACE_RW_Thread_Mutex::tryacquire_read ();
        if (result == 0) _status = RW_LOCK_READ;
        return result;
    }
    virtual int tryacquire_write () {
        if (_status == RW_LOCK_WRITE)       return  0;
        if (_status != RW_LOCK_UNACQUIRED)  return -1;
        int result = ACE_RW_Thread_Mutex::tryacquire_write ();
        if (result == 0) _status = RW_LOCK_WRITE;
        return result;
    }
    virtual int tryacquire_write_upgrade () {
        if (_status == RW_LOCK_WRITE)       return  0;
        if (_status != RW_LOCK_READ)        return -1;
        int result = ACE_RW_Thread_Mutex::tryacquire_write_upgrade ();
        if (result == 0) _status = RW_LOCK_WRITE;
        else this->release ();  // auto-release to prevent deadlocks - may
                                // be inappropriate for your program.
        return result;
    }
```

```
private:
    enum
    {
        RW_LOCK_UNACQUIRED,
        RW_LOCK_READ,
        RW_LOCK_WRITE,
        RW_LOCK_REMOVED
    } _status;
};
```

Comments

Mar 4, 2002