INTRODUCTION
TO THE
MULTIPLE READ / SINGLE WRITE LOCKING

Francisco Jose Tapia
fjtapia@gmail.com

1.- INTRODUCTION

This is a classic problem, not well resolved in the C++11 specification. Several solutions are proposed, specially the upgrade_mutex as described in the document
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3427.html#OverviewUpgrade

or in the Boost Thread library http://www.boost.org/doc/libs/1_54_0/doc/html/thread.html

This solution showed here an alternative to these solutions, with the idea of to be easy to understand and easy to use.

The standard library have two kind of elements :

- **mutex**, which are the elements for to block a thread. There are several types of mutex
- **elements for to handle the mutex** (*std::unique_lock, std::shared_lock, std::lock_guard*). These elements provide an easy and robust way for to manage the mutex. Several functions running on several threads create shared_lock , unique_lock or lock_guard over the same mutex.

The idea of this implementation is the same.

- **mutex_data** which are the elements for to lock the threads
- **elements for to handle the mutex_data** . They provide an easy and robust way for to manage the mutex_data. When manage only one mutex_data (*mutex_read, mutex_write*) or when manage a pair of mutex_data (*mutex_read_read, mutex_write_write, mutex_write_read*). Several functions running on several threads create these elements over the same mutex_data.

2.- BASIC CONCEPTS

mutex_data

Is the data structure which have all the internal information of the mutex. All the data structures to be shared between several threads must include an mutex_data object.

This lock model for multiple read and single write, run over a mutex_data structure. The mutex_data structure can be implemented in several ways. In the countertree library it's implemented over a spin-lock mutex with yield functions.

mutex_read

This is the handle of a mutex_data for operations which don't modify the the data structure. Several threads can have simultaneously mutex_read locks over the same mutex_data

```
template <class mutex_data>
class mutex_read
{public :
    mutex_read( mutex_data & mdc, bool lock_now = true);
    ~mutex_read ( void);

    bool try_lock  ( void );
    void lock      ( void );
    void unlock    ( void ) ;
};
```

The mutex_read receives a mutex_data in the constructor. It receives too a bool variable lock_now. If true lock the mutex_data in the constructor. If it can't lock wait until can do it.

The destructor unlock the mutex_read if locked. This mutex_read have the classical operations try_lock , lock and unlock.

mutex_write

This is the handle for operations which modify the data structure (insert, delete or modify). Only 1 thread can have a mutex_write lock over a mutex_data. If other threads try to lock a mutex_write or lock a muted_read over the same data structure, they are stopped and must wait until the unlock of the mutex_write lock owner.

```
template <class mutex_data>
class mutex_write
{public :
    mutex_write (mutex_data &mm, bool lock_now =true);
    ~mutex_write ( void);

    bool  try_lock ( void);
    void  lock     ( void);
    void  unlock   ( void);

    void wait_no_readers ();
};
```

The mutex_read receives a mutex_data in the constructor. It receives too a bool variable lock_now. If true lock the mutex_data in the constructor. If it can't lock , wait until can do it. The destructor unlock the mutex_write if locked.

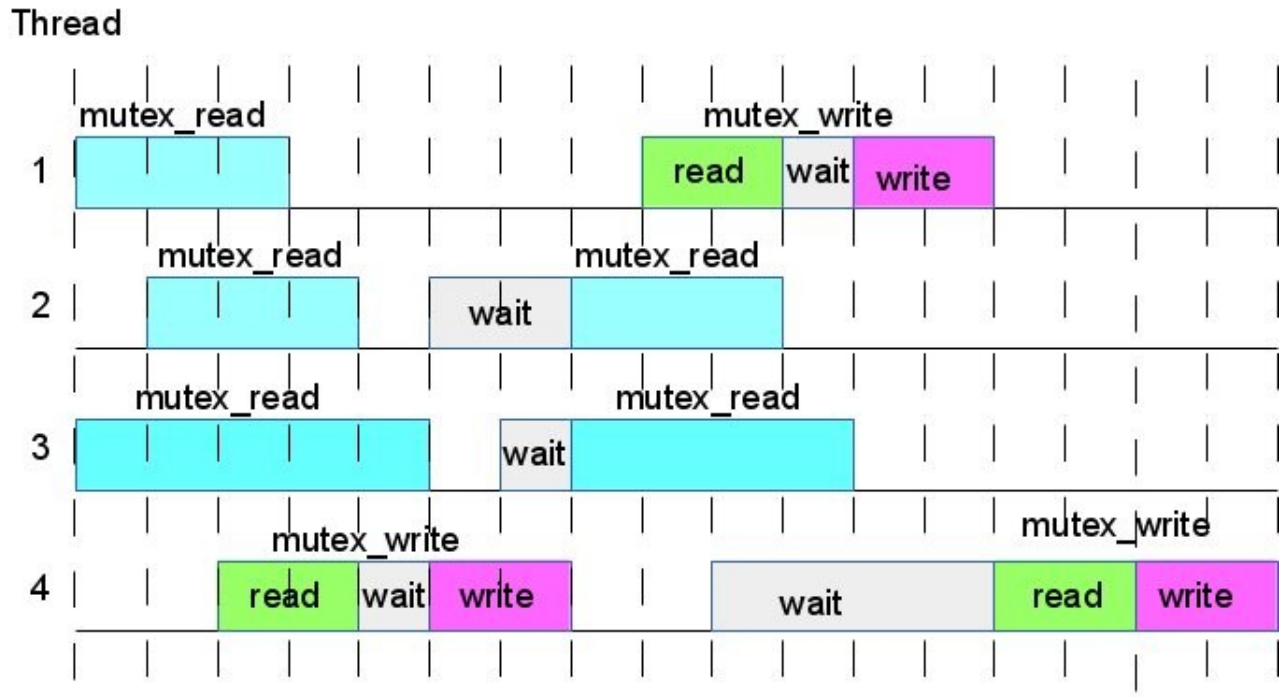
The operation of this mutex have 3 parts. For to easy understand imagine the operation of delete an element from a key in a map.

Part 1 Find the pointer to the node to delete from the key . This part don't modify the map, due this can be concurrent with operations with mutex_read locking over the map which lock before the lock of the mutex_write

Part 2 Wait until the unlock of all the operations with mutex_read locking over the map locked before the mutex_write lock . This part don't exist if no wait is needed . This wait is done with the function ***wait_no_readers***

Part 3 In this part realize the modification of the data structure (in the example the deletion of the node from the pointer to the node). The thread owner of the mutex_write lock is the only one which can do something over the map.

You can see an example in this diagram



3.- ADDITIONAL CLASSES

In order to simplify the use when you must lock two mutex_data , the implementation provide handler for to manage two simultaneously (*mutex_read_read, mutex_write_read and mutex_write_write*). The two mutex_data are locked or unlocked in 1 operation. If the lock process is not OK , the two mutex_data are unlocked and the thread is wait until the two mutex_data can be locked.

In the constructor , they receive the two mutex_data and a bool parameter lock_now, if this parameter is true try to lock the two mutex_data in the constructor. The destructor unlock it if locked.

mutex_write_read

```
template <class mutex_data1 , class mutex_data2>
class mutex_write_read
{public :
    mutex_write_read ( mutex_data1 &mw,
                       mutex_data2 &mr,
                       bool lock_now=true);
    ~mutex_write_read ( void);

    bool try_lock ( void);
    void lock     ( void);
    void unlock   ( void);
    void wait_no_readers ();
};
```

Receives two mutex data in the constructor. The first is for to build an internal mutex_write and the second is for to build an internal mutex_read. It receives too a bool variable lock_now. If true lock the mutex_data in the constructor. If it can't lock wait until can do it. The destructor unlock it if locked ;

In this class the function wait_no readers is the wait of the mutex_write until all the operations with mutex_read over the data structure unlock their mutex.

mutex_write_write

Receives two mutex data in the constructor. The first is for to build the first internal mutex_write and the second is for to build the second It receives too a bool variable lock_now. If true lock the mutex_data in the constructor. If it can't lock wait until can do it. The destructor unlock it if locked ;

```
template <class mutex_data1 ,class mutex_data2>
class mutex_write_write
{public :
    mutex_write_write( mutex_data1 &mw1 ,
                       mutex_data2 &mw2,
                       bool lock_now=true );
    ~mutex_write_write (void);

    bool try_lock ( void );
    void lock     ( void );
    void unlock   ( void );

    void wait_no_readers_first  ();
    void wait_no_readers_second ();
};
```

mutex_read_read

Receives two mutex data in the constructor. The first is for to build the first internal mutex_read and the second is for to build the second It receives too a bool variable lock_now. If true lock the mutex_data in the constructor. If it can't lock wait until can do it.
The destructor unlock it if locked ;

```
template <class mutex_data1 , class mutex_data2>
class mutex_read_read
{public :
    mutex_read_read( mutex_data1 &mr1 ,
                     mutex_data2 &mr2,
                     bool lock_now=true );
    ~mutex_read_read( void);

    bool try_lock ( void);
    void lock     ( void);
    void unlock   ( void);
};
```

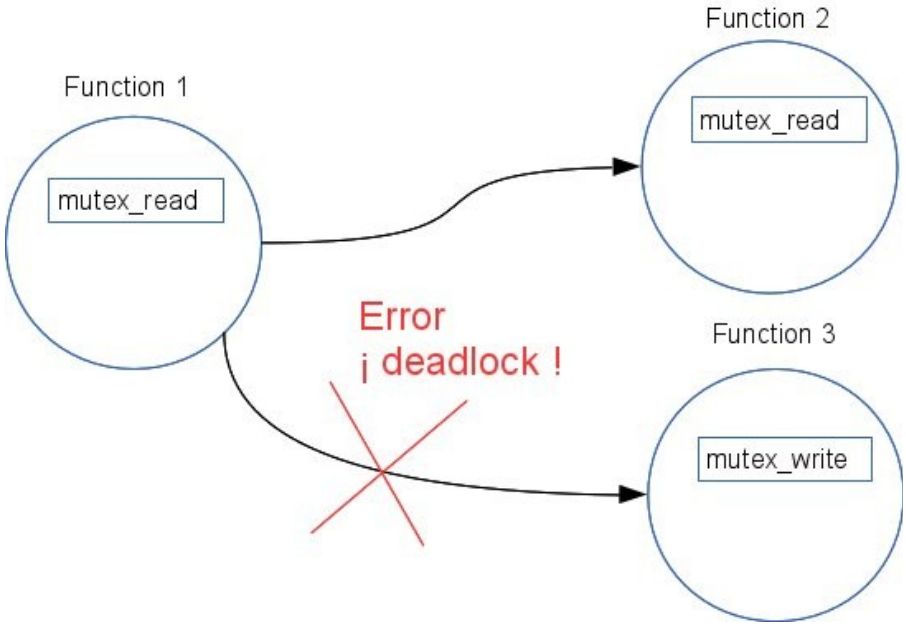
4.- RECURSIVE

This mutex is recursive. The thread which lock the mutex_write, can call to other function which lock other mutex_read or mutex_write over the same data structure.

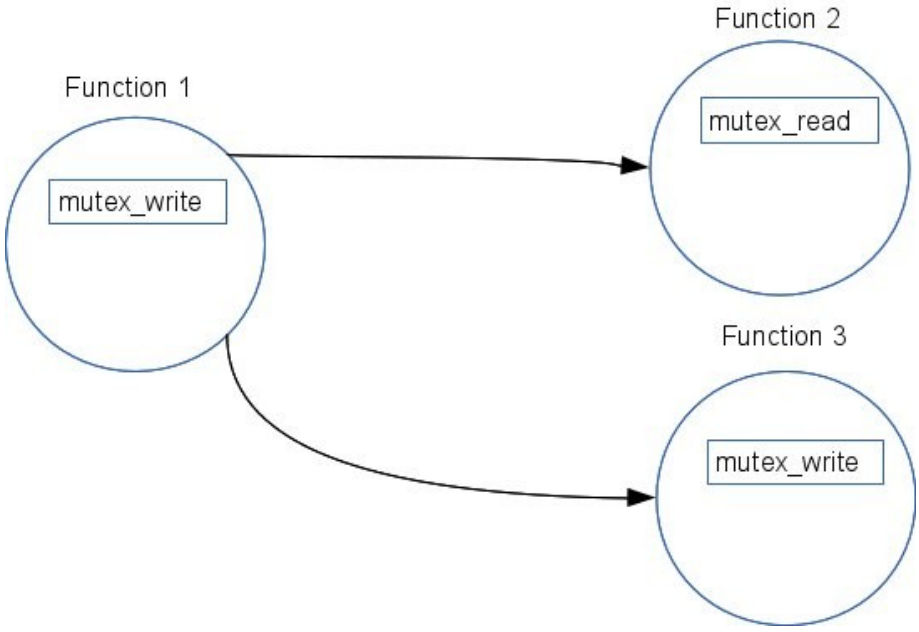
The thread which lock the mutex_read, can call to other function which lock other mutex_read over the same data structure, but don't lock a mutex_write over the same data structure because this produce a deadlock.

It's easy to remember. The mutex_read is for functions which don't modify the object, as the const functions. The mutex_write is for functions which modify the object as the non const functions.

The const functions (mutex_read) can call only to const functions. If try to call a non const functions (mutex_write) produce an error. In the const functions is detected by the compiler, with the mutex , you produce a deadlock.



The non const functions (mutex_write) can call any function, const (mutex_read) and non const (mutex_write) without problems.



5.- EXAMPLES

The code of these examples is obtained from the class `countertree::cntree_set.` This class have the next public definitions and defined a field BD of the type mtx_data.

```
//*************************************************************
//              P U B L I C      D E F I N I T I O N S
//*************************************************************
typedef typename config_fastmutex<cnc>::fastmutex_data   mtx_data ;
typedef mutex_read <mtx_data>                            mtx_read ;
typedef mutex_write<mtx_data>                            mtx_write ;

mutable mtx_data BD ;
```

MUTEX_READ

```
//-------------------------------------------------------------
//  function : size
/// @brief return the number of elements in the cntree_map
/// @return number of elements in the cntree_map
//-------------------------------------------------------------
size_type size (void) const
{   mtx_read BR ( BD);
    return st.size() ;
};
```

MUTEX_WRITE

```
//-------------------------------------------------------------
//  function : erase
/// @brief Erase all the elements with a key
/// @param [in] x : key of all the elements to erase
/// @return number of elements erased
//-------------------------------------------------------------
size_type erase ( const key_type& x )
{   //------------ begin ---------------
    mtx_write BM ( BD);
    iterator I = st.find_norep(x) ;
    if ( I == end()) return 0 ;
    BM.wait_no_readers() ;
    st.erase( I);
    return 1 ;
} ;
```

MUTEX_WRITE_READ

```
//-----------------------------------------------------------------
//  function : operator =
/// @brief Asignation operator
/// @param [in] m : cntree_map from where copy the data
/// @return Reference to the cntree_map after the copy
//-----------------------------------------------------------------
cntree_map  & operator= ( const cntree_map &m)
{   //---------------------- begin ----------------------
    if ( this == &m) return *this ;
    mutex_write_read <mtx_data, mtx_data> BM ( BD, m.BD);
    BM.wait_no_readers() ;
    st = m.st;
    return *this ;
};
```

MUTEX_WRITE_WRITE

```
//-----------------------------------------------------------------------
//  function : swap
/// @brief swap the data of the cntree_map st with the actual cntree_map
/// @param [in] mp : cntree_map to swap
/// @return none
//-----------------------------------------------------------------------
void swap ( cntree_map & mp )
{   //--------------- begin -------------
    if ( this == &mp) return ;
    mutex_write_write <mtx_data, mtx_data> BM ( BD, mp.BD);
    BM.wait_no_readers_first() ;
    BM.wait_no_readers_second() ;
    st.swap ( mp.st);
};
```

MUTEX_READ_READ

```
//-------------------------------------------------------------------------
//  function : operator==
/// @brief swap the data of the cntree_map st with the actual cntree_map
/// @param [in] m1 : cntree_map to compare
/// @param [in] m2 : cntree_map to compare
/// @return true : equals  false : not equals
//-------------------------------------------------------------------------
static bool operator == (const cntree_map & m1, const cntree_map & m2 )
{   //--------------- begin -------------
    if ( this == &mp) return true;
    mutex_read_read <mtx_data, mtx_data> BM ( BD, mp.BD);
    return ( m1.st == m2.st);
};
```