Boost.Synchro 0.3.3

Vicente J. Botet Escriba

Table of Contents

Warning

Synchro is not a part of the Boost libraries.

Overview

Description

To date, C++ multi threaded programs that need to be efficient use the same mutexes, semaphores, and events that Dijkstra described 40 years ago. This unfortunate state of affairs makes multi threaded programs difficult to design, debug, ensure correct, optimize, maintain, and analyze formally. Consequently, building tools that automate detection of race conditions and deadlocks is highly desirable.

I believe that most of the synchronization mechanisms applied to multi-threaded programs works as well for multi-process programs. This can not be done if the basic facilities mutexes and condition variables are not based on common concepts and take a different form depending of the library provider. We need to states these bases through concepts.

The main sources of inspiration of this library were

- the papers of Kevlin Henney about asynchronous C++ More C++ Threading - From Procedural to Generic, by Example and its C++0x proposal [N1833 - Preliminary Threading Library Proposal for TR2].
- the papers of Andrei Alexandrescu on multi-threading programming volatile - Multithreaded Programmer's Best Friend
- the ACE framework of Douglas C. Schmidt An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit.

This library is a compilation of what I have found in the literature not yet present in Boost. My main concen has been to boostify all these ideas in a coherent way.

Boost.Synchro provides:

- A uniform usage of Boost.Thread and Boost.Interprocess synchronization mechanisms based on lockables(mutexes) concepts and locker(guards) concepts.
  - lockables traits and lock generators,
  - generic free functions on lockables as: `lock`, `try_lock`, ...
  - locker adapters of the Boost.Thread and Boost.Interprocess lockers models,
  - complete them with the corresponding models for single-threaded programms: `null_mutex` and `null_condition` classes,
  - locking families,
  - `semaphore` and `binary_semaphore`,
  - `condition_lockable` lock which put toghether a lock and its associated conditions.
- A coherent way exception based timed lock approach for functions and constructors,
- A rich palete of lockers as
  - `strict_locker`, `nested_strict_locker`,
  - `condition_locker`,
  - `reverse_locker`, `nested_reverse_locker`,
  - `locking_ptr`, `on_derreference_locking_ptr`,
  - `externally_locked`,
- `array_unique_locker` on multiple lockables.
- Generic free functions on multiple lockables `lock`, `try_lock`, `lock_until`, `lock_for`, `try_lock_until`, `try_lock_for`, `unlock` * lock adapters of the Boost.Thread and Boost.Interprocess lockable models,
  - `lock_until`, `lock_for`, `try_lock_until`, `try_lock_for`
- A polymorphic lockable hierarchy.
- High-level abstractions for handling more complicated synchronization problems, including
  - `monitor` for guaranteeing exclusive access to an object.

- Language-like Synchronized Block Macros

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of Synchro files
#include <boost/synchro/synchro.hpp>

// Create a namespace aliases
namespace bsynchro = boost::synchro;
```

Introduction

Using Thread, Interprocess and Null synchronization mechanisms uniformly

One of the problems when doing multi threaded application with Boost.Thread and Boost.Interprocess is that the synchronization mechanism of these two libraries even if they are very close since the release 1.35, there are some minor differences that make quite difficult to design a class that can work independently with synchronization mechanisms of both libraries.

This library proposes some classes that allows to write code that can be used indistinguishably with thread or interprocess synchronization mechanisms. This section is inspired on the work from **C++ Threading - A Generic-Programming Approach** - Kevlin Henney.

Lock substitutability

The Boost (C++0x) mutexes have associated a category which form a sub-typing hierarchy:

```
ExclusiveLockable <- SharedLockable <- UpgradeLockable
```

```
struct exclusive_lock_tag {};
struct sharable_lock_tag : exclusive_lock_tag {};
struct upgradable_lock_tag : sharable_lock_tag  {};
```

Locking behavior can be further categorized as:

- Re-entrancy: recursive or not

```
non_recursive <- recursive
```

- Scope: whether the lock is usable with a mono-threaded, multi-threaded or multi-process context

```
mono_threaded <- multi_threaded <- multi_process
```

- Lifetime: The lifetime of a lock could be associated to the process, the kernel or the file-system

```
process_lifetime <- kernel_lifetime <- filesystem_lifetime
```

- Timed interface: has or not a timed interfaces

```
hasnt_timed_interface <- has_timed_interface
```

Substitutability applies both to the degree of syntactic support and to the locking semantics

- A recursive mutex and binary semaphore are substitutable in code written against a exclusive mutex
- A null mutex is substitutable for all others in a single-threaded environment

We can see these axes of variation expressed against some Boost synchronization mechanisms (from now bip stands for boost::interprocess):

- boost::mutex: ExclusiveLock, non-recursive, has-not-timed-interface, multi-threaded
- boost::shared_mutex: UpgradableLock, non-recursive, has-timed-interface, multi-threaded
- bip::synchro::null_mutex: UpgradableLock, recursive, has-timed-interface, mono-threaded
- bip::synchro::interprocess_recursive_mutex ExclusiveLock, recursive, has-timed-interface, multi_process.

Lock traits

The Boost.Synchro library contains a set of very specific traits classes, some of them encapsulate a single trait for a Lockable type; for example, is a lock recursive (is_recursive), is useful in a multi threaded context (is_multi_threaded).

The Boost.Synchro lock-traits classes share a unified design that mimic the one of Boost.TypeTraits: each class inherits from a the type true_type if the type has the specified property and inherits from false_type otherwise.

Boost.Synchro also contains a set of classes that perform a specific transformation on a type; for example, they can remove a top-level const or volatile qualifier from a type. Each class that performs a transformation defines a single typedef-member type that is the result of the transformation.

Finding the best lock

Inverse traits can match a lockable type based on specific traits, for a given family of lock types.

It is also possible to specify characteristics to perform a reverse lookup to find a primitive lock type, either by exact match or by substitutable match.

Synchronization family

A class that will do internal locking can be parameterized by the type of synchronization family needed to achieve the desired level of concurrency control. This depends of the usage scope of this class, and this can be mono_threaded, multi_threaded, multi_process.

For example the thread_synchronization_family can be used to instantiate a message_queue class in a multi_threaded environment, all public methods will be thread-safe, with the corresponding overhead that implies. In contrast, if a null_synchronization_policy class is used to instantiate message_queue, all public methods will not be thread-safe, and there will be no additional overhead.

Syntactic lock traits

The Boost.Synchro library also contains classes that try to remove the syntactic differences between the synchronization mechanisms of the Boost.Thread and Boost::Interprocess libraries. The differences identified up to now are:

- The scoped locks live in a different namespace and some have different names with the same semantic. IMO these should be shared.
- The exception thrown lives in a different name space and different names with the same semantic.
- This exception should be common.
- The move semantics (&&) are expressed with a class named differently. This class could be a good candidate of Boost library by itself.
- The scoped locks can be initialized with static const variables in order to overload the constructor for lock adoption, lock deferral or try to lock. Even if the name of these variables is the same, these variables live in different namespace. It would be nice if these both libraries use the same type and the same variables

I hope that these two Boost libraries will merge their synchronization mechanisms in a near future. Waiting for this merge this could serve as a temporary solution.

Mapping the current mutexes (Boost.Thread and Boost/Interprocess) to the common concepts

The mapping from the current mutexes and scoped guards (Boost.Thread and Boost/Interprocess) to the common concepts has been done adding a indirection level. Instead of requiring a given member function, the lockable concepts reside in fre functions. Neither the Boost.Thread nor Boost/Interprocess mutexes and locks are based on functions, but can see them as models of the common lockable and loker concepts by specializing these generic free functions. In order to make easier the mapping these functions call by default to a member function with the equivalent signature.

For example thread_timed_mutex is viwed as a lockable by specializing the folloxing functions:

```
namespace partial_specialization_workaround {
    template <class Clock, class Duration >
    struct lock_until<boost::timed_mutex,Clock, Duration>   {
        static void
        apply( boost::timed_mutex& lockable, const chrono::time_point<Clock, Dura↵
tion>& abs_time ) {
            if(!lockable.timed_lock(boost::con↵
vert_to<posix_time::ptime>(abs_time))) throw timeout_exception();
        }
    };
    template <class Rep, class Period >
    struct lock_for<boost::timed_mutex,Rep, Period> {
        static void
        apply( boost::timed_mutex& lockable, const chrono::duration<Rep, Period>& rel_time ) {
            if(!lockable.timed_lock(boost::convert_to<posix_time::time_dura↵
tion>(rel_time))) throw timeout_exception();
        }
    };
    template <class Clock, class Duration >
    struct try_lock_until<boost::timed_mutex,Clock, Duration> {
        static typename result_of::template try_lock_until<boost::timed_mutex,Clock, Dura↵
tion>::type
        apply( boost::timed_mutex& lockable, const chrono::time_point<Clock, Dura↵
tion>& abs_time ) {
            return lockable.timed_lock(boost::convert_to<posix_time::ptime>(abs_time));
        }
    };
    template <class Rep, class Period >
    struct try_lock_for<boost::timed_mutex,Rep, Period> {
        static typename result_of::template try_lock_for<boost::timed_mutex,Rep, Period>::type
        apply( boost::timed_mutex& lockable, const chrono::duration<Rep, Period>& rel_time ) {
            return lockable.timed_lock(boost::convert_to<posix_time::time_duration>(rel_time));
        }
    };
}
```

Note that only the functions for which the equivalent signature differ are defined. For the others the default works as expected.

Condition lockable

Based on the idead of Kevlin Henney, the library provides condition lockable, which allows a condition variable to be associated with a Lockable.

Treating condition locking as a property of Lockable rather than viceversa has the benefit of making clear how something is locked and accessed, as it were emphasising it in the first person.

```
class product_queue {
public:
    ...
    product *pull() {
        mtx.lock();
        while(queue.empty())
            guard.relock_on(not_empty);
        product *pulled = queue.front();
        queue.pop();
        mtx.unlock();
        return pulled;
    }
    ...
};
```

Requiring the user of a condition variable to implement a while loop to verify a condition's predicate is potentially error prone. It can be better encapsulated by passing the predicate as a function object to the locking function.

```
class product_queue {
public:
    ...
    product *pull() {
        mtx.lock_when(not_empty, has_products(queue));
        product *pulled = queue.front();
        queue.pop();
        mtx.unlock();
        return pulled;
    }
    ...
};
```

Exception-based Timed Locks

Based on the idead of Kevlin Henney, the library supports timeout exception for all the locking functions having a time or duration parameter.

- A lock with a timeout parameter, i.e. a time or a duration, throws a `timeout_exception` on expiry
- A `try_lock` with a timeout simply returns false on expiry
- Any of the conditional locks throw a `timeout_exception` on expiry
- all the locker constructors with the first parameter a timeout.

Use of timeouts can create more robust programs, by not blocking forever, but at the same time one needs to avoid annoyingly arbitrary limits.

Lockers

Typically, object-oriented programs use object-level locking by associating a synchronization object (mutex) with each object that is susceptible to be shared between threads. Then, code that manipulates the state of the object can synchronize by locking that object. Inside a synchronized section, the mutex associated with the object is locked, and consequently that object's fields can be accessed safely.

In C++, this fundamental idiom is typically implemented with a helper Locker object or lock guard.

A locker is any object or function responsible for coordinating the use of lockable objects.

- Lockers depend on lockable objects - which need not be locking primitives - and not vice-versa. This avoids cycles in the dependency graph.
- Lockers are applications of lockable objects and, as such, form a potentially unbounded family. Most common role of lockers is for exception safety and programming convenience
- Lockers execute-around the lock-unlock pairing.

A locker defines an execution strategy for locking and unlocking that is automated by construction and destruction. It simplifies common use of locking, and does so in an exception-safe fashion. As such, lockers depend on the interface of lockables -e.g. lock and unlock - but lockables do not depend on lockers. The relationship is strictly layered, open and extensible: lockable types may be whole, externally locked objects against which existing lockers can be used; new lockers can be defined that work against existing lockable types.

Substitutability between lockables and lockers does not make sense, so the constructor is always explicit. Implicit copyability is also disabled.

Boost.Thread and Boost.Interprocess defines already a good starting point with these lockers:

- `boost::lock_guard`,
- `boost::unique_lock`, `boost::interprocess::unique_lock` and `boost::interprocess::scoped_lock`
- `boost::share_lock` and `boost::interprocess::sharable_lock`
- `boost::upgrade_lock` and `boost::interprocess::upgradable_lock`.

The problem is that even if these locker models the same model, there is no a single syntax.

The library defines some locker adapters which take care of naming differences and that can be used like

```
bsynchro::unique_locker<boost::mutex> scoped(guard);
```

or

```
bsynchro::unique_locker<boost::interprocess::interprocess_mutex> scoped(guard);
```

Strict lockers

Strict lockers were first introduced by Andrei Alexandrescu. A strict locker is a scoped lock guard ensuring the mutex is locked on the scope of the lock, by locking the mutex on construction and unlocking it on destruction.

`boost::lock_guard` could be seen as a strict_locker if the following constructor didn't exists

```
lock_guard(Lockable &  m, boost::adopt_lock_t)
```

We can say that lock_guard is a strict locker "sur parolle".

There is a const function that is very useful when working with strict lockers and external locking which check is the strict locker is locking an instace of a lockable.

```
bool is_locking(lockable_type* l) const;
```

The library provides three strict lockers

- `strict_locker`: is the basic strict locker, special use when doing external locking.
- `neested_strict_locker`: is a strict_locker of another locker as a unique_lock.
- `conditional_unique_locker` and `conditional_shared_locker` : are strict lockers with the condition_lockable interface. These are the synchronizer of the monitor class.

```
class product_queue {
public:
    ...
    product *pull() {
        conditional_unique_locker<> _(mtx, not_empty, has_products(queue));
        product *pulled = queue.front();
        queue.pop();
        mtx.unlock();
        return pulled;
    }
    ...
};
```

and a meta function `is_strict_locker` which states if a locker is a strict locker.

So as strict lockers do not provide lock/unlock functions they are not models of Lockable.

Try lockers

A Try Locker is a Locker that initialize it in a such way that instead of locking on the constructor with `lock()` they can try to lock with `try_lock()`. Most of the lockers defined in Boost.Thread and Boost.Interprocess could be cosidered as `TryLockers`, i.e. them initialize in this way when `boost::try_to_lock` is given as parameter.

The following code shows one way to use the TryLocker:

```
product *product_queue::try_pull() {
    product *pulled = 0;
    boost::unique_lock<boost::mutex> locker(mtx, boost::try_to_lock);
    if(locker && !queue.empty()) {
        pulled = queue.front();
        queue.pop();
    }
    return pulled;
}
```

All of them use a safe strategy for a boolean conversion which use a member pointer rather than a `bool`, which is typically too permissive:

```
typedef bool try_locker::*is_locked;
operator is_locked() const {
    return locked ? &try_locker::locked : 0;
}
```

If we use interprocess mutexes we need to replace the following line

```
boost::unique_lock<boost::mutex> scoped(guard, boost::try_to_lock);
```

by

```
boost::interprocess::scoped_lock<boost::interprocess::interprocess_mutex> scoped(guard, boost::in↵
terprocess::try_to_lock);
```

There are other `TryLockers` in Boost.Thread defined as a member typedef `scoped_try_lock`. The semantics of each constructor and member function are identical to those of `boost::unique_lock<Lockable>` for the same Lockable, except that the constructor that takes a single reference to a mutex will call `m.try_lock()` rather than `m.lock()`.

```
boost::mutex::scoped_try_lock locker(mtx);
```

The library defines in a generic way a try_unique_locker adapter which takes care of naming differences and that can be used like

```
bsynchro::unique_try_locker<Lockable> locker(mtx);
```

for any model of Lockable.

Exception-based Timed Lockers

In addition to supporting timeout exception for Lock, the library supports them also for ExceptionBaseTimedLockers. The semantics of each constructor and member function are identical to those of boost::unique_locker<Lockable> for the same Lockable, except that the constructor that takes a time or a duration as first parameter in a addition to the reference to a mutex will call `m.lock_un-`

`til(t)` or `m.lock_for(d)` rather than `m.try_lock_until(t)` or `m.try_lock_for(d)` and so a `timeout_exception` is possible on the constructor.

Let me start with an example of an application needing to lock several locks at the same time. Once all the locks are locked something must be done. Otherwise the application do something else and reiterate the lock requests. The natural and exception safe way to do that is

```
while (polling) {
    t=now()+100;
    boost::unique_lock<boost::mutex> l1(m1, t);
    boost::unique_lock<boost::mutex> l2(m2, t);
    boost::unique_lock<boost::mutex> l3(m3, t);
    if (l1.has_lock() && l2.has_lock() && l3.has_lock() {
        foo();
        polling = false;
    } else execute_on_failed();
}
```

The problem with this code is that it locks `m2` even if `l1` do not owns the lock `m1`. The advertised reader could argument that if the lock m1 has not been locked by a timeout, as all share the same time constraint the failing lock of m2 will not be expensive. Well the problem is that the locking of m1 can fail because m1 is already locked. When we try to optimize this

```
while (polling) {
    t=now()+100;
    boost::unique_lock<boost::mutex> l1(m1, t);
    if (l1.has_lock() {
        boost::unique_lock<boost::mutex> l2(m2, t);
        if (l2.has_lock() {
            boost::unique_lock<boost::mutex> l3(m3, t);
            if (l2.has_lock() {
                foo();
                polling = false;
            } else execute_on_failed();
        } else execute_on_failed();
    } else execute_on_failed();
}
```

we found that this starts to be unmaintenable. What is event wrost is that as the preceding one is subject to deadlock if another thread acquire the locks in a different order.

try_lock_until and try_lock_for free functions

To avoid this we can request the acquisition of all the locks toghether (letting the function to try several orders), as it does the function try_lock of Boost.Threads, but adding this time a expiration period parameter

```
while (polling) {
    if (bsynchro::try_lock_for(100, m1, m2, m3)) {
        foo();
        polling = false;
    } else execute_on_failed();
}
```

While this solve the deadlock problem, this code is not exception safe. With exception based lockers we can do the following (note that the time is given as first aregument to the locker constructor)

```
while (polling)
    try {
        t=now()+100;
        bsynchro::unique_locker<boost::mutex> l1(t, m1);
        bsynchro::unique_locker<boost::mutex> l2(t, m2);
        bsynchro::unique_locker<boost::mutex> l3(t, m3);
        foo();
        polling = false;
    } catch (bsynchro::timeout_exception& ex) {execute_on_failed(); }
```

`locker_tuples` or `locker_array` of Locker containers

While this code is exception safe and do not locks `m2` if `m1` is not acquired, it is subject to deadlock. We can go a step ahead and mix the advantage of taking all the locks at once and making the acquisition block scoped. In order to do that we need either a array_locker or a tuple_locker depending on whether the locks are homogeneus or not. The library provides both of them. These locker containers follows the same rules as the element wise lockers. If the time comes after the locks no exception is thrown on timeout and if given as the first parameter a exception will be thown when the time will expire.

So the preceding code becomes without timeout exceptions

```
while (polling) {
    bsynchro::array_unique_locker<boost::mutex, 3> lk(m1, m2, m3, 100);
    if (lk.owns_lock()) {
        foo();
        polling = false;
    } else execute_on_failed();
}
```

which is exception safe or with exception based timed locks (Note that the time is given before the locks)

```
while (polling)
try { bsynchro::array_locker<boost::mutex, 3> lk(100, m1, m2, m3);
    foo();
    polling = false;
} catch (bsynchro::timeout_exception& ex) { execute_on_failed(); }
```

When the Locks locked by an `array_unique_locker` are not homogeneus we need some kind of tuple.

```
while (polling)
try { bsynchro::tuple_unique_locker<T1, T2, T1> lk(100, m1, m2, m3);
    foo();
    polling = false;
} catch (bsynchro::timed_exception& ex) { execute_on_failed(); }
```

`lock_until` and `lock_for` free functions

For completion the exception based timed multi lock functions `unlock`, `lock_until` and `lock_for` are also provided.

```
while (polling)
    try {
        bsynchro::lock_for(100, m1, m2, m3);
        foo();
        polling = false;
    } catch (bsynchro::timeout_exception& ex) {execute_on_failed(); }
```

External lockers

An alternative or complementary approach to internal locking is to support external locking for an object - Multiple calls may be grouped within the same externally defined critical region.

External locking has some associated risks for high-level objects. Incorrect usage can be too easy: a forgotten call to lock or unlock is more likely than with synchronisation primitives because the focus of using the object is on the rest of its non-Lockable interface, so it becomes easy to forget that to use the interface correctly also requires participation in a locking scheme.

To some extent lockers can help, but such a co-operative scheme should only be employed when internal locking is too restricted for a given use, e.g. multiple operations must be performed together. Ideally, if such operations are common they should be defined internally locked and defined in the interface of the object as Combined Methods.

Assuming that locks are re-entrant, external locking can be provided to complement the more encapsulated internal locking, i.e. by default if you want to call a single function you just call it and it automatically locks, but if you want to call multiple functions together you first apply an external lock.

The library provides a `externally_locked` class that allows to access a externally locked class in a thread safe mode through strict lockers.

Where only external locking is used, a safe approach is needed for calling single functions easily. The library provides two classes

- `locking_ptr` and
- `on_dereference_locking_ptr`
- `externally_locked`

Polymorphic lockable

The locks classes introduced previously are a non-polymorphic classes. Clearly, many of the synchronisation primitives support common operations, and hence interfaces. In some cases a more general interface is useful.

The synchronised interface class may be used explicitly as a base class for a class supporting synchronisation.

```
struct exclusive_lock {
    virtual ~exclusive_lock()=0;
    virtual void lock()=0;
    virtual void unlock()=0;
    virtual bool try_lock()=0;
};
```

More usefully for primitives, which are best left as non-polymorphic, an adaptor class is used to provide the interface -- run-time polymorphism -- on behalf of anything supporting the correctly named functions - compile time polymorphism. It easier to take a nonpolymorphic class and adapt it to be polymorphic, than it is do it the other way around: the overhead and semantics of polymorphism can only be introduced to a class, not removed.

```
template <typename Lockable>
class exclusive_lock_adapter : public virtual exclusive_lock
{
    exclusive_lock_adapter(): lock_() {}
    virtual ~exclusive_lock_adapter() {}

    virtual void lock() {lock_.lock();}
    virtual void unlock() {lock_.unlock();}
    virtual bool try_lock() { return lock_.try_lock();}
protected:
    Lockable lock_;
};
```

Language-like Synchronized Block

Nest follows an example of mutual exclusion with automatic objects.

```
{
    scoped_guard<boost_mutex> lock(l);
    foo();
    return bar(); // lock released
}
```

With language-like mutual exclusion this results in:

```
synchronize(l)
{
    foo();
    return bar();
} // lock released
```

This is achieved through macros. If the user wants to use synchronized this way he needs to include a specific file which defines

```
#define synchronized(MUTEX) BOOST_SYNCHRONIZED(MUTEX)
```

The library do not provides this directly because this can broke some user code. The library provideds other safer macros, using the BOOST prefix.

```
#define BOOST_SYNCHRONIZED_VAR(VARS_DECLARATION) \
    if (bool stop_ = false) {} else \
    for (VARS_DECLARATION; !stop_; stop_ = true)

#define BOOST_SYNCHRONIZED(MUTEX) \
    BOOST_SYNCHRONIZED_VAR(boost::scoped_guard<boost::mutex> __lock(MUTEX))
```

Monitors

Concurrent components may interact in different ways: they may access the same objects by, for example, executing functions of these objects; or they may communicate directly by executing functions of each other.

Concurrent execution of objects requires a mechanism for synchronizing the access to shared objects, just as direct communication between objects may require synchronization. The basic mechanism for synchronization in Boost.Threads and Boost.Interprocess are the well known mutex and condition_variables. Mutexes and condition variables are, however, only useful for very simple synchronization problems. The Synchro Library therefore introduce high-level abstractions for handling more complicated synchronization problems, including monitor for guaranteeing exclusive access to an object.

[/

Users'Guide

[/

Getting Started

Installing Synchro

Getting Boost.Synchro

You can get the last stable release of Boost.Synchro by downloading `synchro.zip` from the Boost Vault

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.Synchro

There is no need to compile **Boost.Synchro**, since it's a header only library. Just include your Boost header directory in your compiler include path.

Requirements

Boost.Synchro depends on Boost. You must use either Boost version 1.38.x or the version in SVN trunk (even if Boost version 1.35.x should works also). In particular, **Boost.Synchro** depends on:

Boost.Interprocess	interprocess synchronization primitives
Boost.MPL	for all the meta programing task
Boost.Thread	thread synchronization primitives

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Currently, **Boost.Synchro** has been tested in the following compilers/platforms:

- GCC 3.4.4 Cygwin
- GCC 3.4.6 Linux
- GCC 4.1.2 Linux

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Hello World!

Tutorial

[info The contents of this tutorial is an adaptation of the papers of Andrei Alexandrescu, Kevlin Henney and the concurrent part of the BETA Programming Language.]

Concurrent components may interact in different ways: they may access the same shared objects by, for example, executing functions of these objects; or they may communicate directly by executing functions of each other.

Concurrent execution of objects requires a mechanism for synchronizing the access to shared objects, just as direct communication between objects may require synchronization. The basic mechanism for synchronization in Boost.Threads and Boost.Interprocess are the well known mutex and condition_variables. Mutexes and condition variables are, however, only useful for very simple synchronization problems. The Synchro Library therefore introduce high-level abstractions for handling more complicated synchronization problems, including monitor for guaranteeing exclusive access to an object, controlling external locking and last a so-called rendezvous mechanism for handling direct communication between objects. All the concurrency abstractions being introduced are defined by means of mutexes an conditions.

Lockables

Note

The following is an adaptation of the article "C++ Threading - A Generic-Programming Approach" by Kevin Henney.

Lock substitutability

The Boost (C++0x) mutexes have associated a category which form a sub-typing hierarchy: ExclusiveLock <- SharableLock <- UpgradableLock

```
exclusive_lock <- sharable_lock <- upgradable_lock
```

```
struct exclusive_lock_tag {};
struct sharable_lock_tag : exclusive_lock_tag {};
struct upgradable_lock_tag : sharable_lock_tag  {};
```

Locking behavior can be further categorized as:

- Re-entrancy: recursive or not

```
non_recursive <- recursive
```

```
struct non_recursive_tag {};
struct recursive_tag : non_recursive_tag {};
```

- Scope: whether the lock is usable with a mono-threaded, multi-threaded or multi-process context

```
mono_threaded <- multi_threaded <- multi_process
```

```
struct mono_threaded_tag {};
struct multi_threaded_tag : mono_threaded_tag {};
struct multi_process_tag : multi_threaded_tag  {};
```

- Lifetime: The lifetime of a lock could be associated to the process, the kernel or the file-system

```
process_lifetime <- kernel_lifetime <- filesystem_lifetime
```

```
struct process_lifetime_tag {};
struct kernel_lifetime_tag : process_lifetime_tag {};
struct filesystem_lifetime_tag : kernel_lifetime_tag {};
```

- Timed interface: has or not a timed interfaces

```
hasnt_timed_interface <- has_timed_interface
```

```
struct hasnt_timed_interface_tag {};
struct has_timed_interface_tag : hasnt_timed_interface_tag {};
```

Substitutability applies both to the degree of syntactic support and to the locking semantics

- A recursive mutex and binary semaphore are substitutable in code written against a exclusive mutex
- A null mutex is substitutable for all others in a single-threaded environment

We can see these axes of variation expressed against some Boost synchronization mechanisms (from now bip stands for boost::interprocess):

- boost::mutex: ExclusiveLock, non-recursive, has-not-timed-interface, multi-threaded
- boost::shared_mutex: UpgradableLock, non-recursive, has-timed-interface, multi-threaded
- bip::sync::null_mutex: UpgradableLock, recursive, has-timed-interface, mono-threaded
- bip::sync::interprocess_recursive_mutex ExclusiveLock, recursive, has-timed-interface, multi_process.

Lock traits

Generic programming (writing code which works with any data type meeting a set of requirements) has become the method of choice for providing reusable code.

However, there are times in generic programming when "generic" just isn't good enough - sometimes the differences between types are too large for an efficient generic implementation. This is when the traits technique becomes important - by encapsulating those properties that need to be considered on a type by type basis inside a traits class, we can minimize the amount of code that has to differ from one type to another, and maximize the amount of generic code.

Consider an example:

Boost.Synchro follow the design of Boost.TypeTraits, and contains a set of very specific traits classes, each of which encapsulate a single trait from the Lockable or Locker concepts; for example, is the lock recursive? Or does the lock have a timed interface?

The Boost.Synchro traits classes share a unified design: each class inherits from a the type true_type if the type has the specified property and inherits from false_type otherwise. As we will show, these classes can be used in generic programming to determine the properties of a given lock type and introduce optimizations that are appropriate for that case.

The type-traits library also contains a set of classes that perform a specific transformation on a type; for example, they can remove a top-level const or volatile qualifier from a type. Each class that performs a transformation defines a single typedef-member type that is the result of the transformation. All of the type-traits classes are defined inside namespace boost; for brevity, namespace-qualification is omitted in most of the code samples given.

Implementation

Most of the implementation is fairly repetitive anyway, so here we will just give you a flavor for how some of the classes are implemented. See the reference section for the full details.

A lockable implementer must specialize the scope_tag template class. By default the scope_tag forward to a nested type scope.

```
template <typename Lockable>
struct scope_tag {
    typedef typename Lockable::scope type;
};
```

So the implementer can either have a nested type scope or inherit from the helper lock_traits_base.

```
template<
    typename Scope,
    typename Category,
    typename Reentrancy,
    typename TimedInterface,
    typename Lifetime,
    typename Naming,
    typename Base
>
struct lock_traits_base : Base {
    // TODO add constraints on typenames
    typedef Scope scope;
    typedef Category category;
    typedef Reentrancy reentrancy;
    typedef TimedInterface timed_interface;
    typedef Lifetime lifetime;
    typedef Naming naming;
};
```

which defines the correct types. The lock_traits_base has a lot of parameters, and the defaults are the ones from boost::mutex. So Boost.Thread could use it as follows

```
class mutex : public lock_traits_base<> {
    // ...
};
```

Waiting for that Boost.Synchro specialize the scope_tag in the synchro/thread/mutex.hpp file.

```
template<>
struct scope_tag<boost::mutex> {
    typedef multi_threaded_tag type;
};
```

So the user must include this file to make boost::mutex a model of Lockable for Boost.Synchro.

For example the trait is_multi_threaded is defined as : If Lockable has a scope_tag that inherits from multi_threaded_tag then inherits from true_type, otherwise inherits from false_type.

```
template <typename Lockable>
struct is_multi_threaded
    : is_same_or_is_base_and_derived<
        multi_threaded_tag,
        typename scope_tag<Lockable>::type
    >
{};
```

Finding the best lock

Inverse traits can match a lockable type based on specific traits, for a given family of lock types.

It is also possible to specify characteristics to perform a reverse lookup to find a primitive lock type, either by exact match or by substitutable match.

```
find_best_lock<>::type == boost::mutex
find_best_lock<mono_threaded_tag>::type == bsync::null_mutex
find_best_lock<multi_threaded_tag>::type == bsync::thread_mutex
find_best_lock<multi_process_tag>::type == bsync::interprocess_mutex
```

The user can also find a lock using mpl constraints as follows

```
typedef find_best_lock_between<Lockables,
        mpl::and<is_multi_threaded<_>, is_recursive<_> > >::type best;
```

Synchronization family

A class that will do internal locking can be parameterized by the type of synchronization familly needed to achieve the desired level of concurrency control. This could depends of the usage scope of this class, and this can be mono_threaded, multi_threaded, multi_process.

For example the thread_synchronization_family can be used to instantiate a message_queue class in a multi_threaded environment, all public methods will be thread-safe, with the corresponding overhead that implies. In contrast, if a null_synchronization_policy class is used to instantiate message_queue, all public methods will not be thread-safe, and there will be no additional overhead. A synchronization family must define typedef as for example

```
template <>
struct synchronization_family<multi_threaded_tag> {
   typedef thread_mutex                      mutex_type;
   typedef thread_recursive_mutex            recursive_mutex_type;
   typedef thread_timed_mutex                timed_mutex_type;
   typedef thread_recursive_timed_mutex      recursive_timed_mutex_type;
   typedef thread_shared_mutex               shared_mutex_type;
   typedef boost::condition_variable         condition_type;
   typedef boost::condition_variable_any     condition_type_any;
};
```

Lockable concept

For the main category clasification, the library provides concept classes that can be used with Boost.ConceptCheck. For example LockableConcept object supports the basic features required to delimit a critical region. Supports the basic lock, unlock and try_lock functions and defines the lock traits.

```
template <typename Lockable>
struct LockableConcept {
    typedef typename category_tag<Lockable>::type category;
    typedef typename timed_interface_tag<Lockable>::type timed_interface;
    typedef typename reentrancy_tag<Lockable>::type reentrancy;
    typedef typename scope_tag<Lockable>::type  scope;
    typedef typename lifetime_tag<Lockable>::type  lifetime;
    typedef typename naming_tag<Lockable>::type  naming;

    BOOST_CONCEPT_USAGE(LockableConcept) {
        lockable::lock(l);
        lockable::unlock(l);
        lockable::try_lock(l);
    }
    Lockable& l;
};
```

The user can now check staticaly that the template parameter is a model of Locable as follows

```
#include "boost/synchro/lockable_concepts.hpp"
template <typename Lockable>
class my_class {
  BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
    // ...
};
```

The same can be done for TimedLockableConcept, ShareLockableConcept and UpgradeLockableConcept (See the reference section for more details).

Syntactic lock traits

The locks on Boost.Thread and Boost::Interprocess do not follow the same interface. Most of the differences can be handled through traits, but other are better handled by adapting the interface.

The * The scoped locks live in a different namespace and some have different names with the same semantic.

```
template <typename Lockable>
struct scoped_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::scoped_lock type;
};

template <typename Lockable>
struct unique_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::unique_lock type;
};

template <typename Lockable>
struct shared_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::shared_lock type;
};

template <typename Lockable>
struct upgrade_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::upgrade_lock type;
};

template <typename Lockable>
struct upgrade_to_unique_locker_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::upgrade_to_unique_locker type;
};
```

So instead of using directly the locker of the respective libraries, use these traits.

```
bsync::shared_lock_type<Lockable>::type lock(mutex_);
```

- The exception thrown lives in a different name space and different names with the same semantic.

```
template <typename Lockable>
struct lock_error_type {
    typedef typename lockable_scope_traits<
        typename scope_tag<Lockable>::type, Lockable
    >::lock_error type;
};
```

So instead of using directly the exception type of the respective libraries, use these traits.

```
try {
    // ...
} catch (bsync::lock_error_type<Lockable>::type& ex) {
    // ...
}
```

- The move semantics (&&) are expressed with a class named differently.

```
template <typename Lockable>
struct move_object_type {
    template <typename T>
    struct moved_object : lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::template moved_object<T> {
        moved_object(T& t_): lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::template moved_object<T>(t_) {}
    };
};
```

- The scoped locks can be initialized with static const variables in order to overload the constructor for lock adoption, lock deferral or try to lock. Even if the name of these variables is the same, these variables live in different namespace.

```
template <typename Lockable>
struct defer_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::defer_lock_t type;
    static const type& value() {return lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::defer_lock();}
};

template <typename Lockable>
struct adopt_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::adopt_lock_t type;
    static const type& value(){return lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::adopt_lock();}
};

template <typename Lockable>
struct try_to_lock_type {
    typedef typename lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::try_to_lock_t type;
    static const type& value(){return lockable_scope_traits<
                typename scope_tag<Lockable>::type, Lockable>::try_to_lock();}
};
```

So instead of using directly the variables of the respective libraries, use these traits.

```
bsync::shared_lock_type<Lockable>::type lock(mutex_, bsync::try_to_lock_type::value());
```

Internal Locking--Monitors

Note

This tutorial is an adaptation of chapter Concurrency of the Object-Oriented Programming in the BETA Programming Language and of the paper of Andrei Alexandrescu "Multithreading and the C++ Type System" to the Boost library.

Concurrent threads of execution

Consider, for example, modeling a bank account class that supports simultaneous deposits and withdrawals from multiple locations (arguably the "Hello, World" of multithreaded programming).

From here a component is a model of the `Callable` concept.

On C++0X (Boost) concurrent execution of a component is obtained by means of the `std::thread`(`boost::thread`):

```
boost::thread thread1(S);
```

where `S` is a model of `Callable`. The meaning of this expression is that execution of `S()` will take place concurrently with the current thread of execution executing the expression.

The following example includes a bank account of a person (Joe) and two components, one corresponding to a bank agent depositing money in Joe's account, and one representing Joe. Joe will only be withdrawing money from the account:

```
class BankAccount;

BankAccount JoesAccount;

void bankAgent()
{
    for (int i =10; i>0; --i) {
        //...
        JoesAccount.Deposit(500);
        //...
    }
}

void Joe() {
    for (int i =10; i>0; --i) {
        //...
        int myPocket = JoesAccount.Withdraw(100);
        std::cout << myPocket << std::endl;
        //...
    }
}

int main() {
    //...
    boost::thread thread1(bankAgent); // start concurrent execution of bankAgent
    boost::thread thread2(Joe); // start concurrent execution of Joe
    thread1.join();
    thread2.join();
    return 0;
}
```

From time to time, the `bankAgent` will deposit $500 in `JoesAccount`. Joe will similarly withdraw $100 from his account. These sentences describe that the bankAgent and Joe are executed concurrently.

The above example works well as long as the bankAgent and Joe do not access JoesAccount at the same time. There is, however, no guarantee that this will not happen. We may use a mutex to guarantee exclusive access to each bank.

```
class BankAccount {
    boost::mutex mtx_;
    int balance_;
public:
    void Deposit(int amount) {
        mtx_.lock();
        balance_ += amount;
        mtx_.unlock();
    }
    void Withdraw(int amount) {
        mtx_.lock();
        balance_ -= amount;
        mtx_.unlock();
    }
    int GetBalance() {
        mtx_.lock();
        int b = balance_;
        mtx_.unlock();
        return balance_;
    }
};
```

Execution of the Deposit and Withdraw operations will no longer be able to make simultaneous access to balance.

Mutex is a simple and basic mechanism for obtaining synchronization. In the above example it is relatively easy to be convinced that the synchronization works correctly (in the absence of exception). In a system with several concurrent objects and several shared objects, it may be difficult to describe synchronization by means of mutexes. Programs that make heavy use of mutexes may be difficult to read and write. Instead, we shall introduce a number of generic classes for handling more complicated forms of synchronization and communication.

With the RAII idiom we can simplify a lot this using the scoped locks. In the code below, guard's constructor locks the passed-in object this, and guard's destructor unlocks this.

```
class BankAccount {
    boost::mutex mtx_; ❶
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ -= amount;
    }
    int GetBalance() {
        boost::lock_guard<boost::mutex> guard(mtx_);
        return balance_;
    }
};
```

❶ explicit mutex declaration

The object-level locking idiom doesn't cover the entire richness of a threading model. For example, the model above is quite deadlock-prone when you try to coordinate multi-object transactions. Nonetheless, object-level locking is useful in many cases, and in combination with other mechanisms can provide a satisfactory solution to many threaded access problems in object-oriented programs.

The BankAccount class above uses internal locking. Basically, a class that uses internal locking guarantees that any concurrent calls to its public member functions don't corrupt an instance of that class. This is typically ensured by having each public member function

acquire a lock on the object upon entry. This way, for any given object of that class, there can be only one member function call active at any moment, so the operations are nicely serialized.

This approach is reasonably easy to implement and has an attractive simplicity. Unfortunately, "simple" might sometimes morph into "simplistic."

Internal locking is insufficient for many real-world synchronization tasks. Imagine that you want to implement an ATM withdrawal transaction with the BankAccount class. The requirements are simple. The ATM transaction consists of two withdrawals-one for the actual money and one for the $2 commission. The two withdrawals must appear in strict sequence; that is, no other transaction can exist between them.

The obvious implementation is erratic:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    acct.Withdraw(sum);
    ❶
    acct.Withdraw(2);
}
```

❶ preemption possible

The problem is that between the two calls above, another thread can perform another operation on the account, thus breaking the second design requirement.

In an attempt to solve this problem, let's lock the account from the outside during the two operations:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    boost::lock_guard<boost::mutex> guard(acct.mtx_); ❶
    acct.Withdraw(sum);
    acct.Withdraw(2);
```

❶ mtx_ field is private

Notice that the code above do not compiles, the `mtx_` field is private. We have two possibilities:

- make `mtx_` public which seams odd
- make the `BankAccount` lockable by adding the lock/unlock functions

We can add these functions explicitly

```
class BankAccount {
    boost::mutex mtx_;
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ -= amount;
    }
    void lock() {
        mtx_.lock();
    }
    void unlock() {
        mtx_.unlock();
    }
};
```

or inheriting from a class which add these lockable functions.

The `exclusive_lockable_adapter` class

```
template <typename Lockable>
class exclusive_lockable_adapter
{
public:
    typedef Lockable lockable_type;
    typedef typename scope_tag<Lockable>::type scope;
    typedef typename category_tag<Lockable>::type category;
    typedef typename reentrancy_tag<Lockable>::type reentrancy;
    typedef typename timed_interface_tag<Lockable>::type timed_interface;
    typedef typename lifetime_tag<Lockable>::type lifetime;
    typedef typename naming_tag<Lockable>::type naming;

    BOOST_COPY_CONSTRUCTOR_DELETE(exclusive_lockable_adapter) ❶
    BOOST_COPY_ASSIGNEMENT_DELETE(exclusive_lockable_adapter) ❷
    exclusive_lockable_adapter() {}
    void lock() {lock_.lock();}
    void unlock() {lock_.unlock();}
    bool try_lock() { return lock_.try_lock();}

protected:
    lockable_type* mutex() const { return &lock_; }
    mutable Lockable lock_;
};
```

❶ disable copy construction
❷ disable copy asignement

The `BankAccount` class result now in

```
class BankAccount
: public exclusive_lockable_adapter<thread_mutex>
{
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<BankAccount> guard(*this);
//        boost::lock_guard<boost::mutex> guard(*this->mutex());
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<BankAccount> guard(*this);
//        boost::lock_guard<boost::mutex> guard(*this->mutex());
        balance_ -= amount;
    }
    int GetBalance() {
        boost::lock_guard<BankAccount> guard(*this);
//        boost::lock_guard<boost::mutex> guard(*this->mutex());
        return balance_;
    }
};
```

and the code that do not comiles becomes

```
void ATMWithdrawal(BankAccount& acct, int sum) {
//    boost::lock_guard<boost::mutex> guard(*acct.mutex());
    boost::lock_guard<BankAccount> guard(acct);
    acct.Withdraw(sum);
    acct.Withdraw(2);
}
```

Notice that now acct is being locked by Withdraw after it has already been locked by guard. When running such code, one of two things happens.

- Your mutex implementation might support the so-called recursive mutex semantics. This means that the same thread can lock the same mutex several times successfully. In this case, the implementation works but has a performance overhead due to unnecessary locking. (The locking/unlocking sequence in the two Withdraw calls is not needed but performed anyway-and that costs time.)
- Your mutex implementation might not support recursive locking, which means that as soon as you try to acquire it the second time, it blocks-so the ATMWithdrawal function enters the dreaded deadlock.

As `boost::mutex` is not recursive, we need to use its recursive version `boost::recursive_mutex`.

```
class BankAccount
: public exclusive_lockable_adapter<thread_recursive_mutex>
{

    // ...
};
```

Monitors

The use of `mutex` and `lockers`, as in `BankAccount`, is a common way of defining objects shared by two or more concurrent components. The exclusive_lockable_adapter class was a first step. We shall therefore introduce an abstraction that makes it easier to define such objects. The following class describes a so-called monitor pattern.

```
template <
    typename Lockable=thread_mutex
>
class exclusive_monitor : protected exclusive_lockable_adapter<Lockable> { ❶
protected:
    typedef unspecified synchronizer; ❷
};
```

❶ behaves like an ExclusiveLockable for the derived classes
❷ is an strict lock guard

A monitor object behaves like a `ExclusiveLockable` object but only for the inheriting classes. Protected inheritance from exclusive_lockable_adapter provide to all the derived classes all ExclusiveLockable operations. In addition has a protected nested class, synchronizer, used when defining the monitor operations to synchronize the access critical regions. The BankAccount may be described using Monitor in the following way:

```
class BankAccount : protected exclusive_monitor<>
{
protected:
    int balance_;
public:
    BankAccount() : balance_(0) {}
    BankAccount(const BankAccount &rhs) {
        synchronizer _(*rhs.mutex());
        balance_=rhs.balance_;
    }

    BankAccount& operator=(BankAccount &rhs)
    {
        if(&rhs == this) return *this;

        int balance=0;
        {
            synchronizer _(*rhs.mutex());
            balance=rhs.balance_;
        }
        synchronizer _(*this->mutex());
        balance_=balance;
        return *this;
    }
#if 0
    BankAccount& operator=(BankAccount &rhs)
    {
        if(&rhs == this) return *this;
        int balance=0;
        synchronize (*rhs.mutex()) balance=rhs.balance_;
        synchronize (*this->mutex()) balance_=balance;
        return *this;
    }
#endif

    void Deposit(int amount) {
        synchronizer _(*this->mutex());
        balance_ += amount;
    }
    int Withdraw(int amount) {
        synchronizer _(*this->mutex());
        balance_ -= amount;
        return amount;
```

```
    }
    int GetBalance() {
        synchronizer _(*this->mutex());
        return balance_;
    }
};
```

In the following, a monitor means some sub-class of monitor. A synchroronized operation means an operation using the synchronizer guard defined within some monitor. Monitor is one example of a high-level concurrency abstraction that can be defined by means of mutexes.

Monitor Conditions

It may happen that a component executing an entry operation of a monitor is unable to continue execution due to some condition not being fulfilled. Consider, for instance, a bounded buffer of characters. Such a buffer may be implemented as a monitor with two operations Push and Pull: the Puss operation cannot be executed if the buffer is full, and the Pull operation cannot be executed if the buffer is empty. A sketch of such a buffer monitor may look as follows:

```
class sync_buffer {
    boost::mutex mtx_; ❶
public:
    ...
    bool full() { return in_==out_; }
    bool empty() { return in_==(out_%size)+1; }
    void push(T& v) {
        // wait if buffer is full
        data_[in_]=v;
        in_ = (in_% size)+1;
    }
    T pull() {
        // wait if buffer is empty
        out_ = (out_% size)+1;
        return data_[out_];
    }
};
```

❶ explicit mutex declaration

The meaning of a wait is that the calling component is delayed until the condition becomes true. We can do that using Boost.Thread condition variables like:

```
template <typename T, unsigned size>
class sync_buffer
{
    typedef boost::mutex mutex_type;
    typedef boost::condition_variable condition_type;
    typedef boost::unique_lock<mutex_type> unique_lock_type;
    mutex_type mtx_;
    condition_type not_full_;
    condition_type not_empty_;

    T data_[size+1];
    unsigned in_, out_;

public:
    sync_buffer():in_(0), out_(0) {}

    bool full() { return out_==(in_+1)%(size+1); }
    bool empty() { return out_==in_; }

    unsigned get_in() {return in_;}
    unsigned get_out() {return out_;}
    void push(T v) {
        unique_lock_type guard(mtx_); ❶
        while (full()) { ❷
            not_full_.wait(guard);
        }
        data_[in_]=v;
        in_ = (in_+1)% (size+1);
        not_empty_.notify_one(); ❸
    }

    T pull() {
        unique_lock_type guard(mtx_); ❹
        while (empty()) { ❺
            not_empty_.wait(guard);
        }
        unsigned idx = out_;
        out_ = (out_+1)% (size+1);
        not_full_.notify_one(); ❻
        return data_[idx];
    }
};
```

❶ ensure the mutex is locked!!!
❷ loop until not full!!!
❸ notifies a not_empty condition
❹ ensure the mutex is locked!!!
❺ loop until not full!!!
❻ notifies a not_full condition

The Monitor class replace the nested synchronizer unique_lock with the class `condition_locker` for this purpose:

```
template <
    typename Lockable,
    class Condition=condition_safe<typename best_condition<Lockable>::type >
    , typename ScopeTag=typename scope_tag<Lockable>::type
>
class condition_unique_locker
    : protected unique_locker<Lockable,ScopeTag>
{
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
    typedef unique_locker<Lockable, ScopeTag> super_type;
public:
    typedef Lockable lockable_type;
    typedef Condition condition;

    explicit condition_unique_locker(lockable_type& obj); ❶
    condition_unique_locker(lockable_type& obj, condition &cond); ❷
    template <typename Predicate>
    condition_unique_locker(lockable_type& obj, condition &cond, Predicate pred); ❸
    ~condition_unique_locker() ❹

    typedef bool (condition_unique_locker::*bool_type)() const; ❺
    operator bool_type() const; ❻
    bool operator!() const { return false; } ❼
    bool owns_lock() const { return true; } ❽
    bool is_locking(lockable_type* l) const ❾

    void relock_on(condition & cond);
    template<typename Clock, typename Duration>
    void relock_until(condition & cond, chrono::time_point<Clock, Duration> const& abs_time);
    template<typename duration_type>
    void relock_on_for(condition & cond, duration_type const& rel_time);

    template<typename Predicate>
    void relock_when(condition &cond, Predicate pred);
    template<typename Predicate>
    template<typename Clock, typename Duration>
    void relock_when_until(condition &cond, Predicate pred,
            chrono::time_point<Clock, Duration> const& abs_time);
    template<typename Predicate, typename duration_type>
    void relock_when_for(condition &cond, Predicate pred,
            duration_type const& rel_time);

    ❿
};
```

❶ locks on construction
❷ relock on condition
❸ relock condition when predicate satisfaied
❹ unlocks on destruction
❺ safe bool idiom
❻ always owned
❼ always owned
❽ always owned
❾ strict lockers specific function
❿ no possibility to unlock without blocking on wait...

We may now give the complete version of the buffer class. The content of the buffer is: `data_[out_+1], data_[out_+2], ... data_R[in_-1]` where all the indexes are modulo size. The buffer is full if `in_=out_` and it is empty if `in_=(out_+1)%size`.

```
template <typename T, unsigned size>
class sync_buffer : protected exclusive_monitor<>
{
    condition not_full_;
    condition not_empty_;

    T data_[size+1];
    unsigned in_, out_;

    struct  not_full {
        explicit not_full(sync_buffer &b):that_(b){};
        bool operator()() const { return !that_.full(); }
        sync_buffer &that_;
    };
    struct  not_empty {
        explicit not_empty(sync_buffer &b):that_(b){};
        bool operator()() const { return !that_.empty(); }
        sync_buffer &that_;
    };
public:
    BOOST_COPY_CONSTRUCTOR_DELETE(sync_buffer) ❶
    BOOST_COPY_ASSIGNEMENT_DELETE(sync_buffer) ❷
    sync_buffer():in_(0), out_(0) {}

    bool full() { return out_==(in_+1)%(size+1); }
    bool empty() { return out_==in_; }

    unsigned get_in() {return in_;}
    unsigned get_out() {return out_;}

    void push(T v) {
        synchronizer _(*this->mutex(), not_full_, not_full(*this)); ❸
        data_[in_]=v;
        in_ = (in_+1)% (size+1);
        not_empty_.notify_one(); ❹
    }

    T pull() {
        synchronizer _(*this->mutex(), not_empty_, not_empty(*this)); ❺
        unsigned idx = out_;
        out_ = (out_+1)% (size+1);
        not_full_.notify_one(); ❻
        return data_[idx];
    }
};
```

❶ disable copy construction
❷ disable copy asignement
❸ waits until the condition not_full is satisfyed
❹ notifies a not_empty condition
❺ waits until the condition not_empty is satisfyed
❻ notifies a not_full condition

Monitors and conditions are useful for describing simple cases of shared objects (by simple we mean a limited use of conditions). If the conditions for delaying a calling component become complicated, the monitor may similarly become difficult to program and read.

`volatile` and `locking_ptr`

Note

This tutorial is an adaptation of the article of Andrei Alexandrescu "`volatile` - Multithreaded Programmer's Best Friend" to the Boost library.

Just a Little Keyword

Although both C and C++ Standards are conspicuously silent when it comes to threads, they do make a little concession to multi-threading, in the form of the volatile keyword.

Just like its better-known counterpart const, volatile is a type modifier. It's intended to be used in conjunction with variables that are accessed and modified in different threads. Basically, without volatile, either writing multi-threaded programs becomes impossible, or the compiler wastes vast optimization opportunities. An explanation is in order.

Consider the following code:

```
class Gadget
{
public:
    void Wait()
    {
        while (!flag_)
        {
            sleep(1000); // sleeps for 1000 milliseconds
        }
    }
    void Wakeup()
    {
        flag_ = true;
    }
    ...
private:
    bool flag_;
};
```

The purpose of `Gadget::Wait` above is to check the `flag_` member variable every second and return when that variable has been set to true by another thread. At least that's what its programmer intended, but, alas, Wait is incorrect. Suppose the compiler figures out that `sleep(1000)` is a call into an external library that cannot possibly modify the member variable `flag_`. Then the compiler concludes that it can cache `flag_` in a register and use that register instead of accessing the slower on-board memory. This is an excellent optimization for single-threaded code, but in this case, it harms correctness: after you call Wait for some `Gadget` object, although another thread calls Wakeup, Wait will loop forever. This is because the change of `flag_` will not be reflected in the register that caches `flag_`. The optimization is too ... optimistic. Caching variables in registers is a very valuable optimization that applies most of the time, so it would be a pity to waste it. C and C++ give you the chance to explicitly disable such caching. If you use the volatile modifier on a variable, the compiler won't cache that variable in registers -- each access will hit the actual memory location of that variable. So all you have to do to make Gadget's Wait/Wakeup combo work is to qualify `flag_` appropriately:

```
class Gadget
{
public:
    ... as above ...
private:
    volatile bool flag_;
};
```

Most explanations of the rationale and usage of `volatile` stop here and advise you to volatile-qualify the primitive types that you use in multiple threads. However, there is much more you can do with `volatile`, because it is part of C++'s wonderful type system.

Using `volatile` with User-Defined Types

You can volatile-qualify not only primitive types, but also user-defined types. In that case, `volatile` modifies the type in a way similar to const. (You can also apply const and `volatile` to the same type simultaneously.) Unlike `const`, `volatile` discriminates between primitive types and user-defined types. Namely, unlike classes, primitive types still support all of their operations (addition, multiplication, assignment, etc.) when volatile-qualified. For example, you can assign a non-volatile `int` to a `volatile int`, but you cannot assign a non-volatile object to a `volatile` object. Let's illustrate how `volatile` works on user-defined types on an example.

```
class Gadget
{
public:
    void Foo() volatile;
    void Bar();
    ...
private:
    std::tring name_;
    int state_;
};
...
Gadget regularGadget;
volatile Gadget volatileGadget;
```

If you think `volatile` is not that useful with objects, prepare for some surprise.

```
volatileGadget.Foo();   // ok, volatile fun called for
                        // volatile object


regularGadget.Foo();    // ok, volatile fun called for
                        // non-volatile object
volatileGadget.Bar();   // error! Non-volatile function called for
                        // volatile object!
```

The conversion from a non-qualified type to its `volatile` counterpart is trivial. However, just as with const, you cannot make the trip back from `volatile` to non-qualified. You must use a cast:

```
Gadget& ref = const_cast<Gadget&>(volatileGadget);
ref.Bar(); // ok
```

A volatile-qualified class gives access only to a subset of its interface, a subset that is under the control of the class implementer. Users can gain full access to that type's interface only by using a `const_cast`. In addition, just like constness, volatileness propagates from the class to its members (for example, `volatileGadget.name_` and `volatileGadget.state_` are `volatile` variables).

`volatile`, Critical Sections, and Race Conditions

The simplest and the most often-used synchronization device in multi-threaded programs is the mutex.

Mutexes are used to protect data against race conditions. By definition, a race condition occurs when the effect of more threads on data depends on how threads are scheduled. Race conditions appear when two or more threads compete for using the same data. Because threads can interrupt each other at arbitrary moments in time, data can be corrupted or misinterpreted. Consequently, changes and sometimes accesses to data must be carefully protected with critical sections. In object-oriented programming, this usually means that you store a mutex in a class as a member variable and use it whenever you access that class' state. Experienced multi-threaded programmers might have yawned reading the two paragraphs above, but their purpose is to provide an intellectual workout, because now we will link with the `volatile` connection. We do this by drawing a parallel between the C++ types' world and the threading semantics world.

- Outside a critical section, any thread might interrupt any other at any time; there is no control, so consequently variables accessible from multiple threads are `volatile`. This is in keeping with the original intent of `volatile` -- that of preventing the compiler from unwittingly caching values used by multiple threads at once.
- Inside a critical section defined by a mutex, only one thread has access. Consequently, inside a critical section, the executing code has single-threaded semantics. The controlled variable is not `volatile` anymore -- you can remove the `volatile` qualifier.

In short, data shared between threads is conceptually `volatile` outside a critical section, and non-volatile inside a critical section. You enter a critical section by locking a mutex. You remove the `volatile` qualifier from a type by applying a `const_cast`. If we manage to put these two operations together, we create a connection between C++'s type system and an application's threading semantics. We can make the compiler check race conditions for us.

`locking_ptr`

We need a tool that collects a mutex acquisition and a `const_cast`. Let's develop a `locking_ptr` class template that you initialize with a volatile object obj and a mutex mtx. During its lifetime, a `locking_ptr` keeps `mtx` acquired. Also, `locking_ptr` offers access to the volatile-stripped obj. The access is offered in a smart pointer fashion, through operator-> and operator*. The `const_cast` is performed inside `locking_ptr`. The cast is semantically valid because `locking_ptr` keeps the mutex acquired for its lifetime. First, let's define the skeleton of a class `mutex` with which `locking_ptr` will work:

```
class mutex
{
public:
    void lock();
    void unlock();
    ...
};
```

`locking_ptr` is templated with the type of the controlled variable and the exclusive lockable type. For example, if you want to control a Widget, you use a `locking_ptr<Widget>` that you initialize with a variable of type volatile `Widget`. locking_ptr is very simple. locking_ptr implements an unsophisticated smart pointer. It focuses solely on collecting a const_cast` and a critical section.

```
template <typename T>
class locking_ptr {
public:
    // Constructors/destructors
    locking_ptr(volatile T& obj, mutex& mtx)
        : obj_(*const_cast<T*>(&obj)),
          mtx_(mtx)
    {    mtx_.lock();    }
    ~locking_ptr()
    {    mtx_->unlock();    }
    // Pointer behavior
    T& operator*()
    {    return obj_;    }
    T* operator->()
    {   return &obj_;   }
private:
    T& obj_;
    mutex& mtx_;
    locking_ptr(const locking_ptr&);
    locking_ptr& operator=(const locking_ptr&);
};
```

In spite of its simplicity, `locking_ptr` is a very useful aid in writing correct multi-threaded code. You should define objects that are shared between threads as volatile and never use `const_cast` with them -- always use `locking_ptr` automatic objects. Let's illustrate this with an example. Say you have two threads that share a vector<char> object:

```
class SynchroBuf {
public:
    void Thread1();
    void Thread2();
private:
    typedef vector<char> BufT;
    volatile BufT buffer_;
    mutex mtx_; // controls access to buffer_
};
```

Inside a thread function, you simply use a `locking_ptr<BufT>` to get controlled access to the `buffer_` member variable:

```
void SynchroBuf::Thread1() {
    locking_ptr<BufT> lpBuf(buffer_, mtx_);
    BufT::iterator i = lpBuf->begin();
    for (; i != lpBuf->end(); ++i) {
        ... use *i ...
    }
}
```

The code is very easy to write and understand -- whenever you need to use `buffer_`, you must create a `locking_ptr<BufT>` pointing to it. Once you do that, you have access to vector's entire interface. The nice part is that if you make a mistake, the compiler will point it out:

```
void SynchroBuf::Thread2() {
    // Error! Cannot access 'begin' for a volatile object
    BufT::iterator i = buffer_.begin();
    // Error! Cannot access 'end' for a volatile object
    for (; i != lpBuf->end(); ++i) {
        ... use *i ...
    }
}
```

You cannot access any function of `buffer_` until you either apply a `const_cast` or use `locking_ptr`. The difference is that `locking_ptr` offers an ordered way of applying `const_cast` to volatile variables. `locking_ptr` is remarkably expressive. If you only need to call one function, you can create an unnamed temporary `locking_ptr` object and use it directly:

```
unsigned int SynchroBuf::Size() {
    return locking_ptr<BufT>(buffer_, mtx_)->size();
}
```

Back to Primitive Types

We saw how nicely `volatile` protects objects against uncontrolled access and how `locking_ptr` provides a simple and effective way of writing thread-safe code. Let's now return to primitive types, which are treated differently by `volatile`. Let's consider an example where multiple threads share a variable of type int.

```
class Counter
{
public:
    ...
    void Increment() { ++ctr_; }
    void Decrement() { --ctr_; }
private:
    int ctr_;
};
```

If Increment and Decrement are to be called from different threads, the fragment above is buggy. First, `ctr_` must be volatile. Second, even a seemingly atomic operation such as `++ctr_` is actually a three-stage operation. Memory itself has no arithmetic capabilities. When incrementing a variable, the processor:

- Reads that variable in a register
- Increments the value in the register
- Writes the result back to memory

This three-step operation is called RMW (Read-Modify-Write). During the Modify part of an RMW operation, most processors free the memory bus in order to give other processors access to the memory. If at that time another processor performs a RMW operation on the same variable, we have a race condition: the second write overwrites the effect of the first. To avoid that, you can rely, again, on `locking_ptr`:

```
class Counter
{
public:
    ...
    void Increment() { ++*locking_ptr<int, boost::mutex>(ctr_, mtx_); }
    void Decrement() { --*locking_ptr<int, boost::mutex>(ctr_, mtx_); }
private:
    volatile int ctr_;
    boost::mutex mtx_;
};
```

Now the code is correct, but its quality is inferior when compared to SynchroBuf's code. Why? Because with Counter, the compiler will not warn you if you mistakenly access `ctr_` directly (without locking it). The compiler compiles `++ctr_` if `ctr_` is volatile, although the generated code is simply incorrect. The compiler is not your ally anymore, and only your attention can help you avoid race conditions. What should you do then? Simply encapsulate the primitive data that you use in higher-level structures and use `volatile` with those structures. Paradoxically, it's worse to use `volatile` directly with built-ins, in spite of the fact that initially this was the usage intent of `volatile`!

`volatile` Member Functions

So far, we've had classes that aggregate `volatile` data members; now let's think of designing classes that in turn will be part of larger objects and shared between threads. Here is where `volatile` member functions can be of great help. When designing your class, you volatile-qualify only those member functions that are thread safe. You must assume that code from the outside will call the volatile functions from any code at any time. Don't forget: `volatile` equals free multi-threaded code and no critical section; non-volatile equals single-threaded scenario or inside a critical section. For example, you define a class Widget that implements an operation in two variants -- a thread-safe one and a fast, unprotected one.

```
class Widget
{
public:
    void Operation() volatile;
    void Operation();
    ...
private:
    boost::mutex mtx_;
};
```

Notice the use of overloading. Now Widget's user can invoke Operation using a uniform syntax either for volatile objects and get thread safety, or for regular objects and get speed. The user must be careful about defining the shared Widget objects as `volatile`. When implementing a `volatile` member function, the first operation is usually to lock this with a `locking_ptr`. Then the work is done by using the non-volatile sibling:

```
void Widget::Operation() volatile
{
    locking_ptr<Widget,boost::mutex> lpThis(*this, mtx_);
    lpThis->Operation(); // invokes the non-volatile function
}
```

Generic `locking_ptr`

The `locking_ptr` works with a mutex class. How to use it with other mutexes? We can make a more generic `locking_ptr` adding a Lockable template parameter.

```
template <typename T, typename Lockable=boost::mutex>
class locking_ptr  {
public:
    typedef T value_type;
    typedef Lockable lockable_type;

    locking_ptr(volatile value_type& obj, lockable_type& mtx) ❶
        : ptr_(const_cast<value_type*>(&obj)) ❷
        , mtx_(mtx)
    {    mtx_.lock();    } ❸
    ~locking_ptr()
    {    mtx_.unlock();    } ❹

    ❺
    value_type& operator*()
    {    return *ptr_;    }
    value_type* operator->()
    {   return ptr_;   }

    BOOST_DEFAULT_CONSTRUCTOR_DELETE(locking_ptr) ❻
    BOOST_COPY_CONSTRUCTOR_DELETE(locking_ptr) ❼
    BOOST_COPY_ASSIGNEMENT_DELETE(locking_ptr) ❽

private:
    value_type* ptr_;
    lockable_type& mtx_;
};
```

❶ volatile
❷ const_cast
❸ locks on construction
❹ unlocks on destruction
❺ smart pointer related operations
❻ disable default construction
❼ disable copy construction
❽ disable copy asignement

Every model of the ExclusiveLockable concept can be used as parameter.

Specific `locking_ptr` for lockable value types

When the value type is itself lockable we can simplify the `locking_ptr` as follows:

```
template <typename Lockable>
class locking_ptr_1  {
public:
    typedef Lockable value_type;
    typedef Lockable mutex_type;

    locking_ptr_1(volatile value_type& obj)
       : ptr_(const_cast<value_type*>(&obj))
    {    ptr_->lock();    }
    ~locking_ptr_1()
    {    ptr_->unlock();    }

    value_type& operator*()
    {    return *ptr_;    }
    value_type* operator->()
    {   return ptr_;   }

    BOOST_DEFAULT_CONSTRUCTOR_DELETE(locking_ptr_1) ❶
    BOOST_COPY_CONSTRUCTOR_DELETE(locking_ptr_1) ❷
    BOOST_COPY_ASSIGNEMENT_DELETE(locking_ptr_1) ❸

private:
    value_type* ptr_; ❹
};
```

❶ disable default construction
❷ disable copy construction
❸ disable copy asignement
❹ only one pointer needed

External Locking -- `strict_locker` and `externally_locked` classes

Note

This tutorial is an adaptation of the paper of Andrei Alexandrescu "Multithreading and the C++ Type System" to the Boost library.

Locks as Permits

So what to do? Ideally, the BankAccount class should do the following:

- Support both locking models (internal and external).
- Be efficient; that is, use no unnecessary locking.
- Be safe; that is, BankAccount objects cannot be manipulated without appropriate locking.

Let's make a worthwhile observation: Whenever you lock a BankAccount, you do so by using a `lock_guard<BankAccount>` object. Turning this statement around, wherever there's a `lock_guard<BankAccount>`, there's also a locked `BankAccount` somewhere. Thus, you can think of-and use-a `lock_guard<BankAccount>` object as a permit. Owning a `lock_guard<BankAccount>` gives you rights to do certain things. The `lock_guard<BankAccount>` object should not be copied or aliased (it's not a transmissible permit).

1. As long as a permit is still alive, the `BankAccount` object stays locked.
2. When the `lock_guard<BankAccount>` is destroyed, the `BankAccount`'s mutex is released.

The net effect is that at any point in your code, having access to a `lock_guard<BankAccount>` object guarantees that a `BankAccount` is locked. (You don't know exactly which `BankAccount` is locked, however-an issue that we'll address soon.)

For now, let's make a couple of enhancements to the `lock_guard` class template defined in Boost.Thread. We'll call the enhanced version `strict_locker`. Essentially, a `strict_locker`'s role is only to live on the stack as an automatic variable. `strict_locker` must adhere to a non-copy and non-alias policy. `strict_locker` disables copying by making the copy constructor and the assignment operator private. While we're at it, let's disable operator new and operator delete; `strict_locker` are not intended to be allocated on the heap. `strict_locker` avoids aliasing by using a slightly less orthodox and less well-known technique: disable address taking.

```
template <typename Lockable>
class strict_locker
{

      BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
public:
    typedef Lockable lockable_type;
    explicit strict_locker(lockable_type& obj)
        : obj_(obj) { obj.lock(); } ❶
    ~strict_locker() { obj_.unlock(); } ❷

    typedef bool (strict_locker::*bool_type)() const; ❸
    operator bool_type() const { return &strict_locker::owns_lock;  }
    bool operator!() const { return false; } ❹
    bool owns_lock() const { return true; }
    const lockable_type* mutex() const { return &obj_; }
    bool is_locking(lockable_type* l) const { return l==mutex(); } ❺


    BOOST_ADRESS_OF_DELETE(strict_locker) ❻
    BOOST_HEAP_ALLOCATEION_DELETE(strict_locker) ❼
    BOOST_DEFAULT_CONSTRUCTOR_DELETE(strict_locker) ❽
    BOOST_COPY_CONSTRUCTOR_DELETE(strict_locker) ❾
    BOOST_COPY_ASSIGNEMENT_DELETE(strict_locker) ❿

    ⓫

private:
    lockable_type& obj_;
};
```

❶ locks on construction
❷ unlocks on destruction
❸ safe bool idiom
❹ always owned
❺ strict lockers specific function
❻ disable aliasing
❼ disable heap allocation
❽ disable default construction
❾ disable copy construction
❿ disable copy asignement
⓫ no possibility to unlock

Silence can be sometimes louder than words-what's forbidden to do with a `strict_locker` is as important as what you can do. Let's see what you can and what you cannot do with a `strict_locker` instantiation:

- You can create a `strict_locker<T>` only starting from a valid T object. Notice that there is no other way you can create a `strict_locker<T>`.

```
BankAccount myAccount("John Doe", "123-45-6789");
strict_locerk<BankAccount> myLock(myAccount); // ok
```

- You cannot copy `strict_lockers` to one another. In particular, you cannot pass `strict_lockers` by value to functions or have them returned by functions:

```
extern strict_locker<BankAccount> Foo(); // compile-time error
extern void Bar(strict_locker<BankAccount>); // compile-time error
```

- However, you still can pass `strict_lockers` by reference to and from functions:

```
// ok, Foo returns a reference to strict_locker<BankAccount>
extern strict_locker<BankAccount>& Foo();
// ok, Bar takes a reference to strict_locker<BankAccount>
extern void Bar(strict_locker<BankAccount>&);
```

- You cannot allocate a `strict_locker` on the heap. However, you still can put `strict_lockers` on the heap if they're members of a class.

```
strict_locker<BankAccount>* pL =
    new strict_locker<BankAccount>(myAcount); //error!
    // operator new is not accessible
class Wrapper {
    strict_locker memberLock_;
    ...
};
Wrapper* pW = new Wrapper; // ok
```

(Making `strict_locker` a member variable of a class is not recommended. Fortunately, disabling copying and default construction makes `strict_locker` quite an unfriendly member variable.)

- You cannot take the address of a `strict_locker` object. This interesting feature, implemented by disabling unary operator&, makes it very unlikely to alias a `strict_locker` object. Aliasing is still possible by taking references to a `strict_locker`:

```
strict_locker<BankAccount> myLock(myAccount); // ok
strict_locker<BankAccount>* pAlias = &myLock; // error!
    // strict_locker<BankAccount>::operator& is not accessible
strict_locker<BankAccount>& rAlias = myLock; // ok
```

Fortunately, references don't engender as bad aliasing as pointers because they're much less versatile (references cannot be copied or reseated).

- You can even make `strict_locker` final; that is, impossible to derive from. This task is left in the form of an exercise to the reader.

All these rules were put in place with one purpose-enforcing that owning a `strict_locker<T>` is a reasonably strong guarantee that

1. you locked a T object, and

2. that object will be unlocked at a later point.

Now that we have such a strict `strict_locker`, how do we harness its power in defining a safe, flexible interface for BankAccount? The idea is as follows:

- Each of BankAccount's interface functions (in our case, Deposit and Withdraw) comes in two overloaded variants.

- One version keeps the same signature as before, and the other takes an additional argument of type `strict_locker<BankAccount>`. The first version is internally locked; the second one requires external locking. External locking is enforced at compile time by requiring client code to create a `strict_locker<BankAccount>` object.
- BankAccount avoids code bloating by having the internal locked functions forward to the external locked functions, which do the actual job.

A little code is worth 1,000 words, a (hacked into) saying goes, so here's the new BankAccount class:

```
class BankAccount
: public exclusive_lockable_adapter<boost:recursive_mutex>
{
    int balance_;
public:
    void Deposit(int amount, strict_locker<BankAccount>&) {
        // Externally locked
        balance_ += amount;
    }
    void Deposit(int amount) {
        strict_locker<boost:mutex> guard(*this); // Internally locked
        Deposit(amount, guard);
    }
    void Withdraw(int amount, strict_locker<BankAccount>&) {
        // Externally locked
        balance_ -= amount;
    }
    void Withdraw(int amount) {
        strict_locker<boost:mutex> guard(*this); // Internally locked
        Withdraw(amount, guard);
    }
};
```

Now, if you want the benefit of internal locking, you simply call Deposit(int) and Withdraw(int). If you want to use external locking, you lock the object by constructing a `strict_locker<BankAccount>` and then you call `Deposit(int, strict_locker<BankAccount>&)` and `Withdraw(int, strict_locker<BankAccount>&)`. For example, here's the `ATMWithdrawal` function implemented correctly:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    strict_locker<BankAccount> guard(acct);
    acct.Withdraw(sum, guard);
    acct.Withdraw(2, guard);
}
```

This function has the best of both worlds-it's reasonably safe and efficient at the same time.

It's worth noting that `strict_locker` being a template gives extra safety compared to a straight polymorphic approach. In such a design, BankAccount would derive from a Lockable interface. `strict_locker` would manipulate Lockable references so there's no need for templates. This approach is sound; however, it provides fewer compile-time guarantees. Having a `strict_locker` object would only tell that some object derived from Lockable is currently locked. In the templated approach, having a `strict_locker<BankAccount>` gives a stronger guarantee-it's a `BankAccount` that stays locked.

There's a weasel word in there-I mentioned that ATMWithdrawal is reasonably safe. It's not really safe because there's no enforcement that the `strict_locker<BankAccount>` object locks the appropriate BankAccount object. The type system only ensures that some BankAccount object is locked. For example, consider the following phony implementation of ATMWithdrawal:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    BankAccount fakeAcct("John Doe", "123-45-6789");
    strict_locker<BankAccount> guard(fakeAcct);
    acct.Withdraw(sum, guard);
    acct.Withdraw(2, guard);
}
```

This code compiles warning-free but obviously doesn't do the right thing-it locks one account and uses another.

It's important to understand what can be enforced within the realm of the C++ type system and what needs to be enforced at runtime. The mechanism we've put in place so far ensures that some BankAccount object is locked during the call to `BankAccount::Withdraw(int, strict_locker<BankAccount>&)`. We must enforce at runtime exactly what object is locked.

If our scheme still needs runtime checks, how is it useful? An unwary or malicious programmer can easily lock the wrong object and manipulate any BankAccount without actually locking it.

First, let's get the malice issue out of the way. C is a language that requires a lot of attention and discipline from the programmer. C++ made some progress by asking a little less of those, while still fundamentally trusting the programmer. These languages are not concerned with malice (as Java is, for example). After all, you can break any C/C++ design simply by using casts "appropriately" (if appropriately is an, er, appropriate word in this context).

The scheme is useful because the likelihood of a programmer forgetting about any locking whatsoever is much greater than the likelihood of a programmer who does remember about locking, but locks the wrong object.

Using `strict_locker` permits compile-time checking of the most common source of errors, and runtime checking of the less frequent problem.

Let's see how to enforce that the appropriate BankAccount object is locked. First, we need to add a member function to the `strict_locker` class template. The `strict_locker<T>::get_lockable` function returns a reference to the locked object.

```
template <class T> class strict_locker {
    ... as before ...
public:
    T* get_lockable() const { return &obj_; }
};
```

Second, BankAccount needs to compare the locked object against this:

```
class BankAccount {
: public exclusive_lockable_adapter<boost::recursive_mutex>
    int balance_;
public:
    void Deposit(int amount, strict_locker<BankAccount>& guard) {
        // Externally locked
        if (!guard.is_locking(*this))
            throw "Locking Error: Wrong Object Locked";
        balance_ += amount;
    }
...
};
```

The overhead incurred by the test above is much lower than locking a recursive mutex for the second time.

Improving External Locking

Now let's assume that BankAccount doesn't use its own locking at all, and has only a thread-neutral implementation:

```
class BankAccount {
    int balance_;
public:
    void Deposit(int amount) {
        balance_ += amount;
    }
    void Withdraw(int amount) {
        balance_ -= amount;
    }
};
```

Now you can use BankAccount in single-threaded and multi-threaded applications alike, but you need to provide your own synchronization in the latter case.

Say we have an AccountManager class that holds and manipulates a BankAccount object:

```
class AccountManager
: public exclusive_lockable_adapter<boost::mutex>
{
    BankAccount checkingAcct_;
    BankAccount savingsAcct_;
    ...
};
```

Let's also assume that, by design, AccountManager must stay locked while accessing its BankAccount members. The question is, how can we express this design constraint using the C++ type system? How can we state "You have access to this BankAccount object only after locking its parent AccountManager object"?

The solution is to use a little bridge template `externally_locked` that controls access to a BankAccount.

```
template <typename  T, typename Lockable>
class externally_locked {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));

//
    typedef typename syntactic_lock_traits<Lockable>::lock_error lock_error; ❶
public:
    externally_locked(T& obj, Lockable& lockable)
        : obj_(obj)
        , lockable_(lockable)
    {}

    externally_locked(Lockable& lockable)
        : obj_()
        , lockable_(lockable)
    {}

    T& get(strict_locker<Lockable>& locker) {

#ifndef BOOST_SYNCHRO_EXTERNALLY_LOCKED_DONT_CHECK_SAME  ❷
        if (!locker.is_locking(&lockable_)) throw lock_error(); ❸
#endif
        return obj_;
    }
    void set(const T& obj, Lockable& lockable) {
        obj_ = obj;
        lockable_=lockable;
    }
private:
    T obj_;
    Lockable& lockable_;
};
```

❶ needed until Boost Thread and Interprocess unify the exceptions

❷ define BOOST_SYNCHRO_EXTERNALLY_LOCKED_DONT_CHECK_SAME if you don't want to check locker check the same lockable

❸ run time check throw if not locks the same

`externally_locked` cloaks an object of type T, and actually provides full access to that object through the get and set member functions, provided you pass a reference to a `strict_locker<Owner>` object.

Instead of making `checkingAcct_` and `savingsAcct_` of type `BankAccount`, `AccountManager` holds objects of type `externally_locked<BankAccount, AccountManager>`:

```
class AccountManager
    : public exclusive_lockable_adapter<thread_mutex>
{
public:
    typedef exclusive_lockable_adapter<thread_mutex> lockable_base_type;
    AccountManager()
        : checkingAcct_(*this)
        , savingsAcct_(*this)
    {}
    void Checking2Savings(int amount);
    void AccountManager::AMoreComplicatedChecking2Savings(int amount);
private:

    externally_locked_any<BankAccount, AccountManager> checkingAcct_;
    externally_locked_any<BankAccount, AccountManager> savingsAcct_;
};
```

The pattern is the same as before-to access the BankAccount object cloaked by `checkingAcct_`, you need to call `get`. To call `get`, you need to pass it a `strict_locker<AccountManager>`. The one thing you have to take care of is to not hold pointers or references you obtained by calling `get`. If you do that, make sure that you don't use them after the strict_locker has been destroyed. That is, if you alias the cloaked objects, you're back from "the compiler takes care of that" mode to "you must pay attention" mode.

Typically, you use `externally_locked` as shown below. Suppose you want to execute an atomic transfer from your checking account to your savings account:

```
void AccountManager::Checking2Savings(int amount) {
    strict_locker<AccountManager> guard(*this);
    checkingAcct_.get(guard).Withdraw(amount);
    savingsAcct_.get(guard).Deposit(amount);
}
```

We achieved two important goals. First, the declaration of `checkingAcct_` and `savingsAcct_` makes it clear to the code reader that that variable is protected by a lock on an AccountManager. Second, the design makes it impossible to manipulate the two accounts without actually locking a BankAccount. `externally_locked` is what could be called active documentation.

Allowing other strict lockers

Now imagine that the AccountManager function needs to take a `unique_lock` in order to reduce the critical regions. And at some time it needs to access to the `checkingAcct_`. As `unique_lock` is not a strict lock the following code do not compiles:

```
void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard(*this);
    if (some_condition()) {
        guard.lock();
    }
    checkingAcct_.get(guard).Withdraw(amount);
    savingsAcct_.get(guard).Deposit(amount);
    guard1.unlock();
}
```

We need a way to transfer the ownership from the `unique_lock` to a `strict_locker` the time we are working with `savingsAcct_` and then restore the ownership on `unique_lock`.

```
void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard1(*this);
    if (some_condition()) {
        guard1.lock();
    }
    {
        strict_locker<AccountManager> guard(guard1);
        checkingAcct_.get(guard).Withdraw(amount);
        savingsAcct_.get(guard).Deposit(amount);
    }
    guard1.unlock();
}
```

In order to make this code compilable we need to store either a Lockable or a `unique_lock<Lockable>` reference depending on the constructor. Store which kind of reference we have stored,and in the destructor call either to the Lockable `unlock` or restore the ownership.

This seams too complicated to me. Another possibility is to define a nested strict locker class. The drawback is that instead of having only one strict locker we have two and we need either to duplicate every function taking a `strict\_lock` or make these function templates functions. The problem with template functions is that we don't profit anymore of the C++ type system. We must add some static metafunction that check that the Locker parameter is a strict locker. The problem is that we can not really check this or can we?. The `is_strict_locker` metafunction must be specialized by the strict locker developer. We need to belive it "sur parolle". The advantage is that now we can manage with more than two strict lockers without changing our code. Ths is really nice.

Now we need to state that both classes are `strict_lockers`.

```
template <typename Locker>
struct is_strict_locker : mpl::false_ {};

template <typename Lockable>
struct is_strict_locker<strict_locker<Lockable> > : mpl::true_ {}

template <typename Locker>
struct is_strict_locker<nested_strict_locker<Locker> > : mpl::true_ {}
```

Well let me show how this `nested_strict_locker` class looks like and the impacts on the `externally_locked` class and the `AccountManager::AMoreComplicatedFunction` function.

First `nested_strict_locker` class will store on a temporary lock the `Locker`, and transfer the lock ownership on the constructor. On destruction he will restore the ownership. Note also that the Locker needs to have already a reference to the mutex otherwise an exception is thrown and the use of the `locker_traits`.

```
template <typename Locker >
class nested_strict_locker
    {
      BOOST_CONCEPT_ASSERT((MovableLockerConcept<Locker>));
public:
    typedef typename lockable_type<Locker>::type lockable_type; ❶
    typedef typename syntactic_lock_traits<lockable_type>::lock_error lock_error;

    nested_strict_locker(Locker& locker)
        : locker_(locker) ❷
        , tmp_locker_(locker.move()) ❸
    {
        #ifndef BOOST_SYNCHRO_STRCIT_LOCKER_DONT_CHECK_OWNERSHIP  ❹
        if (tmp_locker_.mutex()==0) {
            locker_=tmp_locker_.move(); ❺
            throw lock_error();
        }
        #endif
        if (!tmp_locker_) tmp_locker_.lock(); ❻
    }
    ~nested_strict_locker() {
        locker_=tmp_locker_.move(); ❼
    }
    typedef bool (nested_strict_locker::*bool_type)() const;
    operator bool_type() const { return &nested_strict_locker::owns_lock; }
    bool operator!() const { return false; }
    bool owns_lock() const { return true; }
    const lockable_type* mutex() const { return tmp_locker_.mutex(); }
    bool is_locking(lockable_type* l) const { return l==mutex(); }

    BOOST_ADRESS_OF_DELETE(nested_strict_locker)
    BOOST_HEAP_ALLOCATEION_DELETE(nested_strict_locker)
    BOOST_DEFAULT_CONSTRUCTOR_DELETE(nested_strict_locker) ❽
    BOOST_COPY_CONSTRUCTOR_DELETE(nested_strict_locker) ❾
    BOOST_COPY_ASSIGNEMENT_DELETE(nested_strict_locker) ❿

private:
    Locker& locker_;
    Locker tmp_locker_;
};
```

❶ Name the lockable type locked by Locker

❷ Store reference to locker
❸ Move ownership to temporaty locker
❹ Define BOOST_SYNCHRO_EXTERNALLY_LOCKED_DONT_CHECK_OWNERSHIP if you don't want to check locker ownership
❺ Rollback for coherency purposes
❻ ensures it is locked
❼ Move ownership to nesting locker
❽ disable default construction
❾ disable copy construction
❿ disable copy asignement

The `externally_locked` get function is now a template function taking a Locker as parameters instead of a `strict_locker`. We can add test in debug mode that ensure that the Lockable object is locked.

```
template <typename  T, typename Lockable>
class externally_locked_any {

//
public:

    // ... as before
    template <class Locker>
    T& get(Locker& locker) {
        BOOST_CONCEPT_ASSERT((StrictLockerConcept<Locker>));

        BOOST_STATIC_ASSERT((is_strict_locker<Locker>::value)); ❶
        BOOST_STATIC_ASSERT((is_same<Lockable,
                typename lockable_type<Locker>::type>::value)); ❷
#ifndef BOOST_SYNCHRO_EXTERNALLY_LOCKED_DONT_CHECK_OWNERSHIP  ❸
        if (! locker ) throw lock_error(); ❹
#endif
#ifndef BOOST_SYNCHRO_EXTERNALLY_LOCKED_DONT_CHECK_SAME
        if (!locker.is_locking(&lockable_)) throw lock_error();
#endif
        return obj_;
    }

};
```

❶ locker is a strict locker "sur parolle"
❷ that locks the same type
❸ define BOOST_SYNCHRO_EXTERNALLY_LOCKED_NO_CHECK_OWNERSHIP if you don't want to check locker ownership
❹ run time check throw if no locked

The `AccountManager::AMoreComplicatedFunction` function needs only to replace the `strict_locker` by a `nested_strict_locker`.

```
void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard1(*this);
    if (some_condition()) {
        guard1.lock();
    }
    {
        nested_strict_locker<unique_lock<AccountManager> > guard(guard1);
        checkingAcct_.get(guard).Withdraw(amount);
        savingsAcct_.get(guard).Deposit(amount);
    }
    guard1.unlock();
}
```

References

Toward Simplified Parallel Support in C++	Justin E. Gottschlich & Paul J. Rogers, 2009 - Not yet published
N1833 - Preliminary Threading Library Proposal for TR2	Kevlin Henney, 2005
More C++ Threading - From Procedural to Generic, by Example	Kevlin Henney
C++ Threading - A Generic-Programming Approach	Kevlin Henney, April 16, 2004
Multithreading and the C++ Type System	Andrei Alexandrescu, Febraury 8, 2002
volatile - Multithreaded Programmer's Best Friend	Andrei Alexandrescu, Febraury 1, 2001
Asynchronous C++	Kevlin Henney, September 9, 1996.
An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit	Douglas C. Schmidt
ACE	Douglas C. Schmidt

Glossary

lockable .

reentrancy .

lock .

locker .

lifetime .

scope .

Reference

Lockables

Header <boost/synchro/lockable_traits.hpp>

```
namespace boost { namespace synchro {

    struct mono_threaded_tag;
    struct multi_threaded_tag;
    struct multi_process_tag;
    template <typename Lockable> struct scope_tag;

    template <typename Lockable> struct is_mono_threaded;
    template <typename Lockable> struct is_multi_threaded;
    template <typename Lockable> struct is_multi_process;

    struct process_lifetime_tag;
    struct kernel_lifetime_tag;
    struct filesystem_lifetime_tag;
    template <typename Lockable> struct lifetime_tag;

    struct anonymous_tag;
    struct named_tag;
    template <typename Lockable> struct naming_tag;

    struct exclusive_lock_tag;
    struct sharable_lock_tag;
    struct upgradable_lock_tag;
    template <typename Lockable> struct category_tag;

    template <typename Lockable> struct is_exclusive_lock;
    template <typename Lockable> struct is_sharable_lock;
    template <typename Lockable> struct is_upgradable_lock;

    struct non_recursive_tag;
    struct recursive_tag;
    template <typename Lockable> struct reentrancy_tag;

    template <typename Lockable> struct is_recursive_lock;

    struct hasnt_timed_interface_tag;
    struct has_timed_interface_tag;
    template <typename Lockable> struct timed_interface_tag;

    template <typename Lockable> struct has_timed_interface;

    template <typename Locker> struct lockable_type;

    template <typename Lockable> struct best_condition;

    template <typename Lockable> struct best_condition_any;

    template <typename Lockable> struct scoped_lock_type;
    template <typename Lockable> struct unique_lock_type;
    template <typename Lockable> struct shared_lock_type;
    template <typename Lockable> struct upgrade_lock_type;

    template <typename Lockable> struct lock_error_type;
```

```
    template <typename Lockable> struct move_object_type;

    template <typename Lockable> struct defer_lock_type;
    template <typename Lockable> struct adopt_lock_type;
    template <typename Lockable> struct try_to_lock_type;

    template<typename Scope> struct default_lifetime;

    template<
        typename Scope=multi_threaded_tag,
        typename Cathegory=exclusive_lock_tag,
        typename Reentrancy=non_recursive_tag,
        typename TimedInterface=has_timed_interface_tag,
        typename Lifetime=typename default_lifetime<Scope>,
        typename Naming=anonymous_tag,
        typename Base=void
    > struct lock_traits_base;

}}
```

Template Class `has_timed_interface`

```
template <typename Lockable>
struct has_timed_interface
    : is_same_or_is_base_and_derived<
        has_timed_interface_tag,
        typename timed_interface_tag<Lockable>::type
    >
{};
```

Synopsis Description

Class `is_exclusive`

Synopsis Description

Class `is_shared`

Synopsis Description

Class `is_recursive`

Synopsis Description

Class `is_mono_threaded`

Synopsis Description

Class `is_multi_threaded`

Synopsis Description

Class `is_multi_process`

Synopsis Description

Class `mutex_type`

Synopsis Description

Class `scoped_lock`

Synopsis Description

Class `unique_lock`

Synopsis Description

Class `shared_lock`

Synopsis Description

Class `upgrade_lock`

Synopsis Description

Class `lock_error`

Synopsis Description

Class `moved_object`

Synopsis Description

Class `lock_error2`

Synopsis Description

Class `lock_error3`

Synopsis Description

Class `lock_error4`

Synopsis Description

Class `lock_traits`

Synopsis

```
template<typename Lockable>
struct lock_traits;
    typedef Lockable           mutex_type;
    typedef unspecified           scoped_lock;
    typedef unspecified           unique_lock;
    typedef unspecified           shared_lock;
    typedef unspecified           upgrade_lock;
    typedef unspecified              lock_error;
    typedef unspecified         moved_object;
    static const unspecified     defer_lock();
    static const unspecified     adopt_lock();
    static const unspecified     try_to_lock();
};
```

Description Lock Traits characterise lockable types.

`nested_strict_locker` public member types

1. ;

Header <boost/synchro/lockable_concepts.hpp>

Lockable concepts.

```
template <typename Lockable> struct LockableConcept;
template <typename Lockable> struct TimedLockableConcept;
template <typename Lockable> struct ShareLockableConcept;
template <typename Lockable> struct UpgradeLockableConcept;
```

Template Class LockableConcept<>

The boost::mutex and boost:interprocess mutex family classes are a non-polymorphic classes that encapsulates a system primitive and portion of C API. Clearly, many of the synchronisation primitives support common operations, and hence a Concept. The ExclusiveLockable class can be used with the Boost.ConceptCheck in templates that work with a exclusive synchronisation.

LockableConcept object supports the basic features required to delimit a critical region. Supports the basic lock, unlock and try_lock functions and defines the lock traits

```
template <typename Lockable>
struct LockableConcept {
    typedef typename category_tag<Lockable>::type category;
    typedef typename timed_interface_tag<Lockable>::type timed_interface;
    typedef typename reentrancy_tag<Lockable>::type reentrancy;
    typedef typename scope_tag<Lockable>::type  scope;
    typedef typename lifetime_tag<Lockable>::type  lifetime;
    typedef typename naming_tag<Lockable>::type  naming;

    BOOST_CONCEPT_USAGE(LockableConcept) {
        lockable::lock(l);
        lockable::unlock(l);
        lockable::try_lock(l);
    }
    Lockable& l;
};
```

Template Class TimedLockableConcept<>

TimedLockableConcept object extends ExclusiveLockConcept with the timed_lock function

```
template <typename Lockable>
struct TimedLockableConcept {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));

    BOOST_CONCEPT_USAGE(TimedLockableConcept) {
        lockable::lock_until(l, t);
        lockable::lock_for(l, boost::chrono::seconds(1));
        lockable::try_lock_until(l, t);
        lockable::try_lock_for(l, boost::chrono::seconds(1));
    }
    Lockable& l;
    boost::chrono::system_clock::time_point t;
};
```

Template Class ShareLockableConcept<>

ShareLockableConcept object extends ExclusiveTimedLockConcept with the lock_shared, timed_lock_shared, try_lock_shared and unlock_shared functions

```
template <typename Lockable>
struct ShareLockableConcept {
    BOOST_CONCEPT_ASSERT((TimedLockableConcept<Lockable>));

    BOOST_CONCEPT_USAGE(ShareLockableConcept) {
        lockable::lock_shared(l);
        lockable::lock_shared_until(l, t);
        lockable::lock_shared_for(l, boost::chrono::seconds(1));
        lockable::try_lock_shared(l);
        lockable::try_lock_shared_until(l, t);
        lockable::try_lock_shared_for(l, boost::chrono::seconds(1));
        lockable::unlock_shared(l);
    }
    Lockable& l;
    boost::chrono::system_clock::time_point t;
};
```

Template Class `UpgradeLockableConcept<>`

UpgradeLockableConcept object extends SharableLockableConcept with the lock_upgrade, timed_lock_upgrade, unlock_upgrade_and_lock, unlock_and_lock_shared and unlock_upgrade_and_lock_shared functions.

```
template <typename Lockable>
struct UpgradeLockableConcept {
    BOOST_CONCEPT_ASSERT((ShareLockableConcept<Lockable>));

    BOOST_CONCEPT_USAGE(UpgradeLockableConcept) {
        lockable::lock_upgrade(l);
        //lockable::lock_upgrade_until(l, t);
        //lockable::lock_upgrade_for(l, boost::chrono::seconds(1));
        lockable::try_lock_upgrade(l);
        //lockable::try_lock_upgrade_until(l, t);
        //lockable::try_lock_upgrade_for(l,boost::chrono::seconds(1));
        lockable::unlock_upgrade_and_lock(l);
        lockable::unlock_and_lock_upgrade(l);
        lockable::unlock_and_lock_shared(l);
        lockable::unlock_upgrade_and_lock_shared(l);
    }
    Lockable& l;
    boost::chrono::system_clock::time_point t;
};
```

Header <boost/synchro/lock_generator.hpp>

```
namespace boost { namespace synchro {
    template<typename Scope>
    struct default_lifetime {
        typedef see_below type;
    }
    template<
        typename Scope=multi_threaded_tag,
        typename Category=exclusive_lock_tag,
        typename Reentrancy=non_recursive_tag,
        typename TimedInterface=has_timed_interface_tag,
        typename Lifetime=typename default_lifetime<Scope>,
        typename Naming=anonymous_tag
    >
    struct find_best_lock{
        typedef see_below type;
    }
}}
```

Metafunction default_lifetime<>

Metafunction that returns the default lifetime depending on the Scope.

```
template<typename Scope> struct default_lifetime;
template<> struct default_lifetime<multi_threaded_tag> {
    typedef process_lifetime_tag type;
};
template<> struct default_lifetime<multi_process_tag> {
    typedef kernel_lifetime_tag type;
};
```

Expression: default_lifetime<Scope>::type

Return type: A lifetime tag.

Complexity: constant.

Table 1. default_lifetime relationship

Scope	default_lifetime<Scope>::type
multi_threaded_tag	process_lifetime_tag
multi_process_tag	kernel_lifetime_tag

Metafunction `find_best_lock<>`

```
template<
    typename Scope=multi_threaded_tag,
    typename Category=exclusive_lock_tag,
    typename Reentrancy=non_recursive_tag,
    typename TimedInterface=has_timed_interface_tag,
    typename Lifetime=typename default_lifetime<Scope>,
    typename Naming=anonymous_tag
>
struct find_best_lock{
    typedef see_below type;
}
```

Expression: find_best_lock<...>::type

Return type: A model of `Lockable`.

Complexity: constant.

The library defines already the following matchings:

Table 2. find_best_lock relationship

Scope	Category	Reentrancy	TimedInterface	Lifetime	Naming	find_best_lock<...>::type
multi_threaded_tag	exclusive_lock_tag	non_recursive_tag	has_timed_interface_tag	process_lifetime_tag	anonymous_tag	boost::synchro::thread_mutex
multi_threaded_tag	exclusive_lock_tag	recursive_tag	has_timed_interface_tag	process_lifetime_tag	anonymous_tag	boost::synchro::thread_recursive_mutex
multi_threaded_tag	exclusive_lock_tag	non_recursive_tag	has_timed_interface_tag	process_lifetime_tag	anonymous_tag	boost::synchro::thread_timed_mutex
multi_threaded_tag	*	non_recursive_tag	*	process_lifetime_tag	anonymous_tag	boost::synchro::thread_shared_mutex
multi_process_tag	exclusive_lock_tag	non_recursive_tag	*	kernel_lifetime_tag	anonymous_tag	boost::synchro::interprocess_mutex
multi_process_tag	exclusive_lock_tag	recursive_tag	*	kernel_lifetime_tag	anonymous_tag	boost::synchro::interprocess_recursive_mutex
multi_process_tag	*	non_recursive_tag	*	process_lifetime_tag	anonymous_tag	boost::synchro::interprocess_upgradable_mutex

Header <boost/synchro/lockable_adapter.hpp>

```
namespace boost { namespace synchro {
    template <typename Lockable> class exclusive_lockable_adapter;
    template <typename TimedLock> class timed_lockable_adapter;
    template <typename SharableLock> class shared_lockable_adapter;
    template <typename UpgradableLock> class upgrade_lockable_adapter;
    template <
        typename Lockable,
        typename category,
        typename timed_interface
    > struct lockable_adapter;
}}
```

Template Class exclusive_lockable_adapter<>

```
template <typename Lockable>
class exclusive_lockable_adapter {
public:
    BOOST_COPY_CONSTRUCTOR_DELETE(exclusive_lockable_adapter)
    BOOST_COPY_ASSIGNEMENT_DELETE(exclusive_lockable_adapter)

    typedef Lockable lockable_type;
    typedef typename scope_tag<Lockable>::type scope;
    typedef typename category_tag<Lockable>::type category;
    typedef typename reentrancy_tag<Lockable>::type reentrancy;
    typedef typename timed_interface_tag<Lockable>::type timed_interface;
    typedef typename lifetime_tag<Lockable>::type lifetime;
    typedef typename naming_tag<Lockable>::type naming;

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

Template Class timed_lockable_adapter<>

```
template <typename TimedLock>
class timed_lockable_adapter : public exclusive_lockable_adapter<TimedLock> {
public:
    timed_lockable_adapter() {}

    bool try_lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    bool try_lock_for(TimeDuration const & relative_time);

    void lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    void lock_for(TimeDuration const & relative_time);
};
```

Template Class `shared_lockable_adapter<>`

```
template <typename SharableLock>
class shared_lockable_adapter : public timed_lockable_adapter<SharableLock>     {
public:
    shared_lockable_adapter() {}
    void lock_shared();
    bool try_lock_shared();
    void unlock_shared();

    bool try_lock_shared_until(system_time const& t);
    template<typename TimeDuration>
    bool try_lock_shared_for(TimeDuration const& t);

    template<typename TimeDuration>
    void lock_shared_for(TimeDuration const& t);
    void lock_shared_until(system_time const& t);
};
```

Template Class `upgrade_lockable_adapter<>`

```
template <typename UpgradableLock>
class upgrade_lockable_adapter : public shared_lockable_adapter<UpgradableLock>{
public:
    upgrade_lockable_adapter();

    void lock_upgrade();
    bool try_lock_upgrade();
    void unlock_upgrade();

    void unlock_upgrade_and_lock();
    void unlock_and_lock_upgrade();
    void unlock_and_lock_shared();
    void unlock_upgrade_and_lock_shared();

    bool try_lock_upgrade_until(system_time const&t);
    template<typename TimeDuration>
    bool try_lock_upgrade_for(TimeDuration const&t);
    void lock_upgrade_until(system_time const&t);
    template<typename TimeDuration>
    void lock_upgrade_for(TimeDuration const&t);
};
```

Generic Free Functions on Lockable

Header `<boost/sychro/lockable/lock.hpp>`

Defines a free function `lock` which locks the `Lockable` passed as parameter. The default implementation applies the `lock` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock` free function if the `Lockable` do not provides a `lock` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock` member function, `lock` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct lock {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template<typename Lockable> struct lock {
           static typename result_of::template lock<Lockable>::type apply( Lockable& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template lock<Lockable>::type
    lock(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock.hpp>

Defines a free function `unlock` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock` free function if the `Lockable` do not provides a `unlock` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock` member function, `unlock` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock {
           static typename result_of::template unlock<Lockable>::type apply( Lockable& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock<Lockable>::type
    unlock(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock.hpp>

Defines a free function `try_lock` which try to lock the `Lockable` passed as parameter and return false if unsucessful. The default implementation applies the `try_lock` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock` free function if the `Lockable` do not provides a `try_lock` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock` member function, `try_lock` calls to the static operation apply on a class with the same name in the namespace __partial_specializationwork-around _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct try_lock {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct try_lock {
            static typename result_of::template try_lock<Lockable>::type apply( Lockable& lock↵
able );
        };
    }

    template <typename Lockable>
    typename result_of::template try_lock<Lockable>::type
    try_lock(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable_ do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace unlock_upgrade_and_lock_until. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct try_lock_until {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct try_lock_until {
            static typename result_of::template try_lock_until<Lockable,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template try_lock_until<Lockable,Clock,Duration>::type
    try_lock_until(Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time);
}}}
```

Header <boost/sychro/lockable/try_lock_for.hpp>

Defines a free function `try_lock_for` which tries to locks the `Lockable` passed as parameter until a given time is elapsed. The default implementation applies the `try_lock_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock_for` free function if the `Lockable` do not provides a `try_lock_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock_for` member function, `try_lock_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct try_lock_for {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct try_lock_for {
            static typename result_of::template try_lock_for<Lockable,Rep,Period>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template try_lock_for<Lockable,Rep,Period>::type
    try_lock_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_until.hpp>

Defines a free function `lock_until` which lock the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_until` free function if the __lockuntilable_ do not provides a `lock_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_until` member function, `lock_until` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct lock_until {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct lock_until {
            static typename result_of::template lock_until<Lockable,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template lock_until<Lockable,Clock,Duration>::type
    lock_until(Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_for.hpp>

Defines a free function `lock_for` which lock the `Lockable` passed as parameter until a given time is elapsed. The default implementation applies the `lock_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_for` free function if the `Lockable` do not provides a `lock_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_for` member function, `lock_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationwork-around _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct lock_for {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct lock_for {
            static typename result_of::template lock_for<Lockable>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template lock_for<Lockable>::type
    lock_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_shared.hpp>

Defines a free function `lock_shared` which lock shared the `Lockable` passed as parameter. The default implementation applies the `lock_shared` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared` free function if the `Lockable` do not provides a `lock_shared` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared` member function, `lock_shared` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct lock_shared {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct lock_shared {
           static typename result_of::template lock_shared<Lockable>::type apply( Lockable& lock↵
able );
        };
    }

    template <typename Lockable>
    typename result_of::template lock_shared<Lockable>::type
    lock_shared(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_shared.hpp>

Defines a free function `unlock_shared` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_shared` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_shared` free function if the `Lockable` do not provides a `unlock_shared` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_shared` member function, `unlock_shared` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_shared {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_shared {
          static typename result_of::template unlock_shared<Lockable>::type apply( Lockable& lock↵
able );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_shared<Lockable>::type
    unlock_shared(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock_shared.hpp>

Defines a free function `try_lock_shared` which try to lock the `Lockable` passed as parameter and return false if unsucessful. The default implementation applies the `try_lock_shared` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock_shared` free function if the `Lockable` do not provides a `try_lock_shared` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock_shared` member function, `try_lock_shared` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct try_lock_shared_shared {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct try_lock_shared {
            static typename result_of::template try_lock_shared<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template try_lock_shared<Lockable>::type
    try_lock_shared(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock_shared_for.hpp>

Defines a free function `try_lock_shared_for` which tries to locks the `Lockable` passed as parameter until a given time is elapsed. The default implementation applies the `try_lock_shared_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock_shared_for` free function if the `Lockable` do not provides a `try_lock_shared_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock_shared_for` member function, `try_lock_shared_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround . So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct try_lock_shared_for {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct try_lock_shared_for {
            static typename result_of::template try_lock_shared_for<Lockable,Rep,Period>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template try_lock_shared_for<Lockable,Rep,Period>::type
    try_lock_shared_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_shared_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround . So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct lock_shared_until {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct lock_shared_until {
          static typename result_of::template lock_shared_until<Lockable,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template lock_shared_until<Lockable,Clock,Duration>::type
    lock_shared_until(Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_shared_for.hpp>

Defines a free function `lock_shared_for` which locks shared the `Lockable` passed as parameter until a given time is elepsed. The default implementation applies the `lock_shared_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_for` free function if the `Lockable` do not provides a `lock_shared_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_for` member function, `lock_shared_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround . So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct lock_shared_for {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct lock_shared_for {
            static typename result_of::template lock_shared_for<Lockable,Rep,Period>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template lock_shared_for<Lockable,Rep,Period>::type
    lock_shared_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_upgrade.hpp>

Defines a free function `lock_upgrade` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `lock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_upgrade` free function if the `Lockable` do not provides a `lock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_upgrade` member function, `lock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct lock_upgrade {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct lock_upgrade {
          static typename result_of::template lock_upgrade<Lockable>::type apply( Lockable& lock↵
able );
        };
    }

    template <typename Lockable>
    typename result_of::template lock_upgrade<Lockable>::type
    lock_upgrade(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_upgrade.hpp>

Defines a free function `unlock_upgrade` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_upgrade` free function if the `Lockable` do not provides a `unlock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_upgrade` member function, `unlock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_upgrade {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_upgrade {
            static typename result_of::template unlock_upgrade<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_upgrade<Lockable>::type
    unlock_upgrade(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock_upgrade.hpp>

Defines a free function `try_lock_upgrade` which try to lock the `Lockable` passed as parameter and return false if unsucessful. The default implementation applies the `try_lock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock_upgrade` free function if the `Lockable` do not provides a `try_lock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock_upgrade` member function, `try_lock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct try_lock_upgrade_shared {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct try_lock_upgrade {
            static typename result_of::template try_lock_upgrade<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template try_lock_upgrade<Lockable>::type
    try_lock_upgrade(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_lock_upgrade_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable_ do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace unlock_upgrade_and_lock_until. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct try_lock_upgrade_until {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct try_lock_upgrade_until {
            static typename result_of::template try_lock_upgrade_until<Lockable,Clock,Dura↵
tion>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template try_lock_upgrade_until<Lockable,Clock,Duration>::type
    try_lock_upgrade_until(Lockable& lockable, const chrono::time_point<Clock, Dura↵
tion>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_upgrade_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable_ do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct lock_upgrade_until {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct lock_upgrade_until {
          static typename result_of::template lock_upgrade_until<Lockable,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template lock_upgrade_until<Lockable,Clock,Duration>::type
   lock_upgrade_until(Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time);
}}}
```

Header <boost/sychro/lockable/lock_upgrade_for.hpp>

Defines a free function `lock_upgrade_for` which locks shared the `Lockable` passed as parameter until a given time is elepsed. The default implementation applies the `lock_upgrade_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_upgrade_for` free function if the `Lockable` do not provides a `lock_upgrade_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_upgrade_for` member function, `lock_upgrade_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct lock_upgrade_for {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct lock_upgrade_for {
            static typename result_of::template lock_upgrade_for<Lockable,Rep,Period>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template lock_upgrade_for<Lockable,Rep,Period>::type
    lock_upgrade_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/unlock_and_lock_upgrade.hpp>

Defines a free function `unlock_and_lock_upgrade` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_and_lock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_and_lock_upgrade` free function if the `Lockable` do not provides a `unlock_and_lock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_and_lock_upgrade` member function, `unlock_and_lock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround . So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_and_lock_upgrade {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_and_lock_upgrade {
          static typename result_of::template unlock_and_lock_upgrade<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_and_lock_upgrade<Lockable>::type
    unlock_and_lock_upgrade(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_and_lock_shared.hpp>

Defines a free function `unlock_and_lock_shared` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_and_lock_shared` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_and_lock_shared` free function if the `Lockable` do not provides a `unlock_and_lock_shared` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_and_lock_shared` member function, `unlock_and_lock_shared` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_and_lock_shared {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_and_lock_shared {
          static typename result_of::template unlock_and_lock_shared<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_and_lock_shared<Lockable>::type
    unlock_and_lock_shared(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_upgrade_and_lock_shared.hpp>

Defines a free function `unlock_upgrade_and_lock_shared` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_upgrade_and_lock_shared` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_upgrade_and_lock_shared` free function if the `Lockable` do not provides a `unlock_upgrade_and_lock_shared` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_upgrade_and_lock_shared` member function, `unlock_upgrade_and_lock_shared` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_upgrade_and_lock_shared {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_upgrade_and_lock_shared {
          static typename result_of::template unlock_upgrade_and_lock_shared<Lockable>::type ap↵
ply( Lockable& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_upgrade_and_lock_shared<Lockable>::type
    unlock_upgrade_and_lock_shared(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_upgrade_and_lock.hpp>

Defines a free function `unlock_upgrade_and_lock` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_upgrade_and_lock` member function to the `Lockable`. A user adapting another `Lockable` could

need to specialize the `unlock_upgrade_and_lock` free function if the `Lockable` do not provides a `unlock_upgrade_and_lock` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_upgrade_and_lock` member function, `unlock_upgrade_and_lock` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_upgrade_and_lock {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_upgrade_and_lock {
          static typename result_of::template unlock_upgrade_and_lock<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_upgrade_and_lock<Lockable>::type
    unlock_upgrade_and_lock(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/unlock_upgrade_and_lock_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable_ do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace unlock_upgrade_and_lock_until. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct lock_upgrade_until {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct lock_upgrade_until {
          static typename result_of::template lock_upgrade_until<Lockable,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template lock_upgrade_until<Lockable,Clock,Duration>::type
   lock_upgrade_until(Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time);
}}}
```

Header <boost/sychro/lockable/unlock_upgrade_and_lock_for.hpp>

Defines a free function `unlock_upgrade_and_lock_for` which locks shared the `Lockable` passed as parameter until a given time is elepsed. The default implementation applies the `unlock_upgrade_and_lock_for` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_upgrade_and_lock_for` free function if the `Lockable` do not provides a `unlock_upgrade_and_lock_for` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_upgrade_and_lock_for` member function, `unlock_upgrade_and_lock_for` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround . So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct unlock_upgrade_and_lock_for {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct unlock_upgrade_and_lock_for {
            static typename result_of::template unlock_upgrade_and_lock_for<Lockable,Rep,Peri↵
od>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template unlock_upgrade_and_lock_for<Lockable,Rep,Period>::type
   unlock_upgrade_and_lock_for(Lockable& lockable, const chrono::duration<Rep, Period>& abs_time);
}}}
```

Header <boost/sychro/lockable/try_unlock_upgrade_and_lock_until.hpp>

Defines a free function `lock_shared_until` which locks shared the `Lockable` passed as parameter until a given time is reached. The default implementation applies the `lock_shared_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `lock_shared_until` free function if the __lock_shareduntilable do not provides a `lock_shared_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `lock_shared_until` member function, `lock_shared_until` calls to the static operation apply on a class with the same name in the namespace unlock_upgrade_and_lock_until. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Clock, class Duration >
        struct try_unlock_upgrade_and_lock_until {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Clock, class Duration >
        struct try_unlock_upgrade_and_lock_until {
            static typename result_of::template try_unlock_upgrade_and_lock_until<Lock↵
able,Clock,Duration>::type
            apply( Lockable& lockable, const chrono::time_point<Clock, Duration>& abs_time );
        };
    }

    template <typename Lockable, class Clock, class Duration >
    typename result_of::template try_unlock_upgrade_and_lock_until<Lockable,Clock,Duration>::type
    try_unlock_upgrade_and_lock_until(Lockable& lockable, const chrono::time_point<Clock, Dura↵
tion>& abs_time);
}}}
```

Header <boost/sychro/lockable/try_unlock_upgrade_and_lock_for.hpp>

Defines a free function `try_unlock_upgrade_and_lock_until` which tries to locks the `Lockable` passed as parameter until a given time is elapsed. The default implementation applies the `try_unlock_upgrade_and_lock_until` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_unlock_upgrade_and_lock_until` free function if the `Lockable` do not provides a `try_unlock_upgrade_and_lock_until` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_unlock_upgrade_and_lock_until` member function, `try_unlock_upgrade_and_lock_until` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround__. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable, class Rep, class Period >
        struct try_unlock_upgrade_and_lock_until {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template <typename Lockable, class Rep, class Period >
        struct try_unlock_upgrade_and_lock_until {
          static typename result_of::template try_unlock_upgrade_and_lock_until<Lockable,Rep,Peri↵
od>::type
            apply( Lockable& lockable, const chrono::duration<Rep, Period>& rel_time );
        };
    }

    template <typename Lockable, class Rep, class Period >
    typename result_of::template try_unlock_upgrade_and_lock_until<Lockable,Rep,Period>::type
    try_unlock_upgrade_and_lock_until(Lockable& lockable, const chrono::duration<Rep, Peri↵
od>& abs_time);
}}}
```

Header <boost/sychro/lockable/unlock_shared_and_lock.hpp>

Defines a free function `unlock_shared_and_lock` which lock upgrade the `Lockable` passed as parameter. The default implementation applies the `unlock_shared_and_lock` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `unlock_shared_and_lock` free function if the `Lockable` do not provides a `unlock_shared_and_lock` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `unlock_shared_and_lock` member function, `unlock_shared_and_lock` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct unlock_shared_and_lock {
            typedef void type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct unlock_shared_and_lock {
          static typename result_of::template unlock_shared_and_lock<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template unlock_shared_and_lock<Lockable>::type
    unlock_shared_and_lock(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_unlock_shared_and_lock_hpp.hpp>

Defines a free function `try_lock_upgrade` which try to lock the `Lockable` passed as parameter and return false if unsucessful. The default implementation applies the `try_lock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_lock_upgrade` free function if the `Lockable` do not provides a `try_lock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_lock_upgrade` member function, `try_lock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct try_unlock_shared_and_lock {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct try_lock_upgrade {
            static typename result_of::template try_lock_upgrade<Lockable>::type apply( Lock↵
able& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template try_lock_upgrade<Lockable>::type
    try_lock_upgrade(Lockable& lockable);
}}}
```

Header <boost/sychro/lockable/try_unlock_shared_and_lock_upgrade.hpp>

Defines a free function `try_unlock_shared_and_lock_upgrade` which try to lock the `Lockable` passed as parameter and return false if unsucessful. The default implementation applies the `try_unlock_shared_and_lock_upgrade` member function to the `Lockable`. A user adapting another `Lockable` could need to specialize the `try_unlock_shared_and_lock_upgrade` free function if the `Lockable` do not provides a `try_unlock_shared_and_lock_upgrade` member function with the same prototype.

As for the moment we can not partially specialize a function a trick is used: instead of calling directly to the `try_unlock_shared_and_lock_upgrade` member function, `try_unlock_shared_and_lock_upgrade` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround_. So the user can specialize partially this class.

```
namespace boost { namespace synchro { namespace lockable {
    namespace result_of {
        template <typename Lockable> struct try_unlock_shared_and_lock_upgrade_shared {
            typedef bool type;
        };
    }

    namespace partial_specialization_workaround {
        template< typename Lockable >
        struct try_unlock_shared_and_lock_upgrade {
            static typename result_of::template try_unlock_shared_and_lock_upgrade<Lock↵
able>::type apply( Lockable& lockable );
        };
    }

    template <typename Lockable>
    typename result_of::template try_unlock_shared_and_lock_upgrade<Lockable>::type
    try_unlock_shared_and_lock_upgrade(Lockable& lockable);
}}}
```

Generic Free Functions on Multiple Lockables

Condition Lockables

Header <boost/synchro/locker/condition_safe.hpp>

Wraps a condition variable in order make public only the safe functions, and let the others accesible via a backdoor.

```
template <typename Condition> class condition_safe;
```

Template Class condition_safe

```
template <class Condition>
class condition_safe {
public:
 typedef Condition condition;
 typedef condition_backdoor<Condition> backdoor;
    void notify_one();
    void notify_all();
};
```

Header <boost/synchro/condition_backdoor.hpp>

```
template <typename Condition> struct condition_backdoor;
```

Template Class condition_backdoor

Condition Backdoor template. Used by safe lockers as condition_locker.

```
template <class Condition>
struct condition_backdoor {
 condition_backdoor(condition_safe<Condition>&cnd);
    template <typename Locker>
    void wait(Locker& lock);
    template <typename Locker>
    bool wait_until(Locker& lock, boost::system_time  const&  abs_time);
    template<typename Locker, typename duration_type>
    bool wait_for(Locker& lock, duration_type const& rel_time);

    template <typename Locker, typename Predicate>
    void wait_when(Locker& lock, Predicate pred);
    template<typename Locker, typename predicate_type>
   bool wait_when_until(Locker& lock, predicate_type pred, boost::system_time const& abs_time);
    template<typename Locker, typename predicate_type, typename duration_type>
    bool wait_when_for(Locker& lock, predicate_type pred, duration_type const& rel_time);

    template <typename Locker>
    void notify_one(Locker& lock);
    template <typename Locker>
    void notify_all(Locker& lock);
};
```

Header <boost/synchro/condition_lockable.hpp>

```
namespace boost { namespace synchro {
    template <typename Lockable, typename Condition>
    class condition_lockable
    typedef condition_lockable<thread_mutex> thread_condition_mutex;
    typedef condition_lockable<interprocess_mutex> interprocess_condition_mutex;
}}
```

Template Class condition_lockable

Allows a condition variable to be associated with a Lockable. Treating condition locking as a property of Lockable rather than viceversa has the benefit of making clear how something is locked and accessed, as it were emphasising it in the first person.

Requiring the user of a condition variable to implement a while loop to verify a condition's predicate is potentially error prone. It can be better encapsulated by passing the predicate as a function object to the locking function.

```
template <
    typename Lockable=thread_mutex,
    class Condition=condition_safe<typename best_condition<Lockable>::type >
>
class condition_lockable {
    : public Lockable {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
public:
    typedef Lockable lockable_type;
    typedef Condition condition;

    condition_lockable();
    ~condition_lockable();

    void relock_on(condition & cond);
    void relock_on_until(condition & cond, boost::system_time const& abs_time);
    template<typename duration_type>
    void relock_on_for(condition & cond, duration_type const& rel_time);

    template<typename Predicate>
    void relock_when(condition &cond, Predicate pred);
    template<typename Predicate>
    void relock_when_until(condition &cond, Predicate pred,
            boost::system_time const& abs_time);
    template<typename Predicate, typename duration_type>
    void relock_when_for(condition &cond, Predicate pred,
            duration_type const& rel_time);

private:
    friend class boost::condition_variable;
    friend class boost::condition_variable_any;
    friend class boost::interprocess::condition;
};
```

Lockers

Header <boost/synchro/locker_concepts.hpp>

Locker concepts.

```
namespace boost {  namespace synchro {

    template <typename Locker> struct BasicLockerConcept;
    template <typename Locker> struct LockerConcept;
    template <typename Locker> struct TimedLockerConcept;
    template <typename Locker> struct UniqueLockerConcept;
    template <typename Locker> struct SharedLockerConcept;
    template <typename Locker> struct UpgradeLockerConcept;
    template <typename Locker> struct MovableLockerConcept;

}}
```

Template Class `BasicLockerConcept<>`

```
template <typename Locker>
struct BasicLockerConcept {
    typedef typename lockable_type<Locker>::type lockable_type;

    BOOST_CONCEPT_USAGE(BasicLockerConcept) {
        const Locker l1(mtx_);
        if (l1.is_locking(mtx_)) return;
        if (l1.owns_lock()) return;
        if (l1) return;
        if (!l1) return;
    }
    lockable_type mtx_;
    system_time t;
};
```

Template Class `LockerConcept<>`

```
template <typename Locker>
struct LockerConcept {
    BOOST_CONCEPT_ASSERT((BasicLockerConcept<Locker>));
    typedef typename lockable_type<Locker>::type lockable_type;

    BOOST_CONCEPT_USAGE(LockerConcept) {
        Locker l2(mtx_, defer_lock);
        Locker l3(mtx_, adopt_lock);
        Locker l4(mtx_, try_to_lock);
        l2.lock();
        if (l2.try_lock()) return;
        l2.unlock();
        l2.release();
    }
    lockable_type mtx_;
    system_time t;
};
```

Template Class `TimedLockerConcept<>`

```
template <typename Locker>
struct TimedLockerConcept {
    BOOST_CONCEPT_ASSERT((LockerConcept<Locker>));
    typedef typename lockable_type<Locker>::type lockable_type;

    BOOST_CONCEPT_USAGE(TimedLockerConcept) {
        const Locker l1(mtx_);
        Locker l5(mtx_, t);
        Locker l6(mtx_, boost::posix_time::seconds(1));
        Locker l7(t, mtx_);
        Locker l8(boost::posix_time::seconds(1), mtx_);
        l5.lock_until(t);
        l5.lock_for(boost::posix_time::seconds(1));
        if (l5.try_lock_until(t)) return;
        if (l5.try_lock_for(boost::posix_time::seconds(1))) return;
    }
    lockable_type mtx_;
    system_time t;
};
```

Template Class `UniqueLockerConcept<>`

```
template <typename Locker>
struct UniqueLockerConcept {
    BOOST_CONCEPT_ASSERT((TimedLockerConcept<Locker>));

    BOOST_CONCEPT_USAGE(UniqueLockerConcept) {
    }
};
```

Template Class `SharedLockerConcept<>`

```
template <typename Locker>
struct SharedLockerConcept {
    BOOST_CONCEPT_ASSERT((TimedLockerConcept<Locker>));

    BOOST_CONCEPT_USAGE(SharedLockerConcept) {
    }
};
```

Template Class `UpgradeLockerConcept<>`

```
template <typename Locker>
struct UpgradeLockerConcept {
    BOOST_CONCEPT_ASSERT((TimedLockerConcept<Locker>));

    BOOST_CONCEPT_USAGE(UpgradeLockerConcept) {
    }
};
```

Template Class `MovableLockerConcept<>`

```
template <typename Locker>
struct MovableLockerConcept {
    typedef typename lockable_type<Locker>::type lockable_type;
    BOOST_CONCEPT_ASSERT((LockerConcept<lockable_type>));

    BOOST_CONCEPT_USAGE(MovableLockerConcept) {
        Locker l1(mtx_);
        Locker& l2(l1);
        Locker l3(l1.move());
        BOOST_ASSERT((l2.mutex()!=&mtx_));
        l3.lock();
        l2 = l3.move();
    }
    lockable_type mtx_;
};
```

Header `<boost/synchro/locker/is_strict_locker.hpp>`

Metafunction `is_strict_locker`

```
template <typename Locker>
struct is_strict_locker {
    typedef unspecified value;
};
```

Header `<boost/synchro/locker/strict_locker.hpp>`

Class `strict_locker`

Synopsis

namespace boost { namespace synchro {

```
template <typename Locker>
struct StrictLockerConcept {
    typedef typename lockable_type<Locker>::type lockable_type;
    BOOST_STATIC_ASSERT((is_strict_locker<Locker>::value));

    void f(Locker& l ) {
        BOOST_ASSERT((l.is_locking(lock)));
    }

    BOOST_CONCEPT_USAGE(StrictLockerConcept) {
        {
//            Locker l(lock);
//            BOOST_ASSERT((l));
        }
    }
    lockable_type lock;
};

template <typename Lockable>
class strict_locker {
public:
    typedef Lockable lockable_type;
    typedef unspecified bool_type;

    explicit strict_locker(lockable_type& obj);
    ~strict_locker();

    operator bool_type() const;
    bool operator!() const;
    bool owns_lock() const;
    lockable_type* mutex() const;

    BOOST_ADRESS_OF_DELETE(strict_locker)
    BOOST_HEAP_ALLOCATION_DELETE()
    BOOST_DEFAULT_CONSTRUCTOR_DELETE(strict_locker) /*< disable default construction >*/
    BOOST_COPY_CONSTRUCTOR_DELETE(strict_locker) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(strict_locker) /*< disable copy asignement >*/
};

template <typename Lockable>
struct is_strict_locker<strict_locker<Lockable> > : mpl::true_ {};

template <typename Locker > class nested_strict_locker {
      BOOST_CONCEPT_ASSERT((MovableLockerConcept<Locker>));
public:
    typedef typename lockable_type<Locker>::type lockable_type;
    typedef typename syntactic_lock_traits<lockable_type>::lock_error lock_error;

    nested_strict_locker(Locker& locker);
    ~nested_strict_locker();
    typedef bool (nested_strict_locker::*bool_type)() const;
    operator bool_type() const;
    bool operator!() const;
    bool owns_lock() const;
    const lockable_type* mutex() const;
    bool is_locking(lockable_type* l) const;

    BOOST_ADRESS_OF_DELETE(nested_strict_locker)
    BOOST_HEAP_ALLOCATEION_DELETE(nested_strict_locker)
    BOOST_DEFAULT_CONSTRUCTOR_DELETE(strict_locker) /*< disable default construction >*/
```

```
    BOOST_COPY_CONSTRUCTOR_DELETE(strict_locker) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(strict_locker) /*< disable copy asignement >*/
};

template <typename Locker>
struct is_strict_locker<nested_strict_locker<Locker> > : mpl::true_ {} ;
```

}}

Description

[Note strict_locker is not a model of Lockable concept.]

`strict_locker` template parameters

- `Lockable` : The exclusive lockable type used to synchronize exclusive access

`strict_locker` public types

- `lockable_type` : The exclusive lockable type used to synchronize exclusive access
- `lock_error` : The exception type throw incase of errors
- `bool_type` : The bool_type of the safe_bool idiom

`nested_strict_locker` public member functions

- `explicit strict_locker(lockable_type& obj);`
- `~strict_locker();`
- `operator bool_type() const;`
- `bool operator!() const;`
- `operator bool_type() const;`
- `lockable_type* mutex() const;`
- `lockable_type* get_lockable() const;`

`nested_strict_locker` private and not defined member functions

- `strict_locker()`
- `strict_locker(strict_locker&);`
- `operator=(strict_locker&);`
- `operator&();`
- `void* operator new(std::size_t)`
- `void* operator new[](std::size_t)`
- `void operator delete(void*)`
- `void operator delete[](void*)`

Class `nested_strict_locker`

Synopsis

```
template <typename Locker>
class nested_strict_locker : private boost::noncopyable {
public:
    typedef typename locker_traits<Locker>::bad_lock bad_lock;

    nested_strict_locker(Locker& lock);
    ~nested_strict_locker();

    typedef unspecified bool_type;
    operator bool_type() const;

    bool operator!() const
    bool owns_lock() const
    Mutex* mutex() const
private:
    strict_locker();
    BOOST_NON_ALIAS(strict_locker);
    BOOST_NON_HEAP_ALLOCATED();
};
```

Description

Template Class `nested_strict_locker`

A reverse (or anti) locker.

Synopsis

Description

Header `<boost/synchro/locker/reverse_locker.hpp>`

```
template <typename Lockable>
class reverse_locker;
```

Class `reverse_lock`

This is an interesting adapter class that changes a Lockable into a reverse lockable, i.e.,`lock` on this class calls `unlock` on the lockable, and `unlock` on this class calls `lock` on the lock. One motivation for this class is when we temporarily want to unlock a lock (which we have already locked) but then re-lock it soon after.

Synopsis

```
template <typename Lockable>
class reverse_locker
{
    reverse_locker(Lockable& mtx);
    ~reverse_locker();

protected:
    Lockable& mtx_;
};
```

Header `<boost/synchro/locker/nested_reverse_locker.hpp>`

```
template <typename Locker>
class nested_reverse_locker;
```

Class `nested_reverse_locker`

This is an interesting adapter class that changes a locker into a reverse locker, i.e., `unlock` on construction and `lock` on destruction. One motivation for this class is when we temporarily want to unlock a lock (locked by another locker) but then re-lock it soon after.

Synopsis

template <typename Locker> class nested_reverse_locker : boost::noncopyable { BOOST_CONCEPT_ASSERT((MovableLocker-Concept<Locker>)); public: explicit nested_reverse_locker(Locker& locker); ~nested_reverse_locker(); };

Header `<boost/synchro/locker/condition_locker.hpp>`

```
template <typename Condition> struct condition_backdoor;
template <typename Condition> class condition_safe;
template <typename Lockable, typename Condition>
class condition_unique_locker
template <typename Lockable, typename Condition>
class condition_shared_locker

template <typename Lockable, typename Condition>
class condition_unique_lockable
template <typename Lockable, typename Condition>
class condition_shared_lockable
```

Template Class `condition_backdoor`

```
template <class Condition>
struct condition_backdoor {
 condition_backdoor(condition_safe<Condition>&cnd);
    template <typename Locker>
    void wait(Locker& lock);
    template <typename Locker>
    bool wait_until(Locker& lock, boost::system_time  const&  abs_time);
    template<typename Locker, typename duration_type>
    bool wait_for(Locker& lock, duration_type const& rel_time);

    template <typename Locker, typename Predicate>
    void wait_when(Locker& lock, Predicate pred);
    template<typename Locker, typename predicate_type>
   bool wait_when_until(Locker& lock, predicate_type pred, boost::system_time const& abs_time);
    template<typename Locker, typename predicate_type, typename duration_type>
    bool wait_when_for(Locker& lock, predicate_type pred, duration_type const& rel_time);

    template <typename Locker>
    void notify_one(Locker& lock);
    template <typename Locker>
    void notify_all(Locker& lock);
};
```

Template Class `condition_safe`

```
template <class Condition>
class condition_safe {
public:
 typedef Condition condition;
 typedef condition_backdoor<Condition> backdoor;
    void notify_one();
    void notify_all();
};
```

Template Class `condition_unique_locker`

template < typename Lockable, class Condition=condition_safe<typename best_condition<Lockable>::type > > class condition_unique_locker : protected unique_lock<Lockable> { BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>)); public: typedef Lockable lockable_type; typedef Condition condition;

```
explicit condition_unique_locker(lockable_type& obj);
condition_unique_locker(lockable_type& obj, condition &cond);
template <typename Predicate>
condition_unique_locker(lockable_type& obj, condition &cond, Predicate pred);
~condition_unique_locker();

typedef bool (condition_unique_locker::*bool_type)() const;
operator bool_type() const;
bool operator!() const;
bool owns_lock() const;
bool is_locking(lockable_type* l) const;

void relock_on(condition & cond);
void relock_until(condition & cond, boost::system_time const& abs_time);
template<typename duration_type>
void relock_on_for(condition & cond, duration_type const& rel_time);

template<typename Predicate>
void relock_when(condition &cond, Predicate pred);
template<typename Predicate>
void relock_when_until(condition &cond, Predicate pred,
  boost::system_time const& abs_time);
template<typename Predicate, typename duration_type>
void relock_when_for(condition &cond, Predicate pred,
  duration_type const& rel_time);

/*< no possibility to unlock without blocking on wait... >*/
```

};

Header `<boost/synchro/locker/externally_locked.hpp>`

Template Class `externally_locked`

Synopsis

```
template <class T, class Lockable>
class externally_locked {
public:
    externally_locked(Lockable& owner);
    externally_locked(const T& obj, Lockable& own);

    template <typename Locker>
    T& get(Locker& locker);
    void set(const T& obj, Lockable& owner);
};
```

Description `externally_locked` cloaks an object of type T, and actually provides full access to that object through the get and set member functions, provided you pass a reference to a strict_locker<Lockable> object.

`externally_locked` template parameters

- `T` : the type locked externally

- `Lockable` : The lockable type used to synchronize the access to a T instance

`externally_locked` public member functions

- `template <typename Locker> T& get(Locker& locker);`

Requires: mpl:and_<is_strict_locker<Locker>, is_same<lockable_type_trait<Locker>, Lockable>.

Returns: a reference to the type locked externally.

Throws: lock_error when the locker do not owns the lockable instance

- `void set(const T& obj, Lockable& owner);`

Effect: reinit the type and lockable references with the given values.

Example: See

Header `<boost/synchro/locker/locking_ptr.hpp>`

Class `locking_ptr`

Synopsis

```
template <typename T, typename Lockable>
class locking_ptr : private boost::noncopyable {
public:
    typedef T value_type;
    typedef Lockable mutex_type;

    locking_ptr(volatile value_type& obj, mutex_type& mtx);
    ~locking_ptr();

    value_type& operator*();
    const value_type& operator*() const;
    value_type* operator->();
    const value_type* operator->() const;
};
```

Description

The `locking_ptr` overloads `operator->` to return a temporary object that will perform the locking. This too provides an `operator->`. Calls to `operator->` are automatically chained by the compiler until a raw pointer type is returned. In pointer's `operator->` the lock is applied and in its destructor, called at the end of a full expression, it is released.

Warning

Programmers should be careful about attempting to access the same object twice in a statement using `locking_ptr`: this will cause deadlock if the synchronisation strategy is not re-entrant.

Example Code

`locking_ptr` constructors:destructors # `locking_ptr(volatile value_type& obj, mutex_type& mtx);` # `~locking_ptr();`

`locking_ptr` public member functions

1. `value_type& operator*();`

2. `const value_type& operator*() const;`

3. `value_type* operator->();`

4. `const value_type* operator->() const;`

Class `sharable_locking_ptr`

Synopsis

```
template <typename T, typename SharableLockable>
class sharable_locking_ptr
    : private boost::noncopyable {
public:
    typedef T value_type;
    typedef SharableLockable mutex_type;

    sharable_locking_ptr(volatile value_type& obj, mutex_type& mtx);
    ~sharable_locking_ptr();

    value_type& operator*();
    const value_type& operator*() const;
    value_type* operator->();
    const value_type* operator->() const;
};

template <typename T, typename SharableLockable>
class sharable_locking_ptr<const T, SharableLockable>
    : private boost::noncopyable {
public:
    typedef T value_type;
    typedef SharableLockable mutex_type;

    sharable_locking_ptr(
        volatile const value_type& obj,
        mutex_type& mtx);
    ~sharable_locking_ptr();

    value_type& operator*();
    const value_type& operator*() const;
    value_type* operator->();
    const value_type* operator->() const;
};
```

Description

`nested_strict_locker` public member functions

1. `;`

Header `<boost/synchro/locker/on_derreference_locking_ptr.hpp>`

```
namespace boost { namespace synchro {
    template<typename T, typename Lockable>
    class on_derreference_locking_ptr;
}}
```

Class `on_derreference_locking_ptr`

Synopsis

```
template<typename T, typename Lockable>
class on_derreference_locking_ptr {
public:
    class pointer {
    public:
        explicit pointer(T* target, Lockable* mutex);
        ~pointer();
        T *operator->();
    };

    explicit on_derreference_locking_ptr(T &target, Lockable& mutex);
    pointer operator->() const;
};
```

Description

Header <boost/synchro/locker/array_unique_locker.hpp>

```
namespace boost {  namespace synchro {

    template <typename Lockable, unsigned N>
    class unique_array_locker;
    template <typename Lockable, unsigned N>
    class try_unique_array_locker;

}}
```

Template Class unique_array_locker<>

Synopsis

```
template <typename Lockable, unsigned N>
class unique_array_locker {
public:
    typedef Lockable lockable_type;
    BOOST_NON_CONST_COPY_CONSTRUCTOR_DELETE(unique_array_locker) /*< disable copy construction >*/
    BOOST_NON_CONST_COPY_ASSIGNEMENT_DELETE(unique_array_locker) /*< disable copy asignement >*/

    unique_array_locker(Lockable& m0, ..., Lockable& m_n);
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, adopt_lock_t);
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, defer_lock);
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, try_to_lock_t);

    // try_lock_until constructor
    template<class Clock, class Duration >
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, chrono::time_point<Clock, Dura↵
tion> const& target_time);
    template<class Rep, class Period >
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, chrono::duration<Rep, Period> const& tar↵
get_time)

    template<class Clock, class Duration >
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, chrono::time_point<Clock, Dura↵
tion> const& target_time, throw_timeout_t);
    template<class Rep, class Period >
    unique_array_locker(Lockable& m0, ..., Lockable& m_n, chrono::duration<Rep, Period> const& tar↵
get_time, throw_timeout_t)

    template<class Clock, class Duration >
    unique_array_locker(chrono::time_point<Clock, Duration> const& target_time, Lock↵
able& m0, ..., Lockable& m_n);
    template<class Rep, class Period >
    unique_array_locker(chrono::duration<Rep, Period> const& target_time, Lockable& m0, ..., Lock↵
able& m_n, )

    template<class Clock, class Duration >
    unique_array_locker(nothrow_timeout_t, chrono::time_point<Clock, Duration> const& tar↵
get_time, Lockable& m0, ..., Lockable& m_n);
    template<class Rep, class Period >
    unique_array_locker(nothrow_timeout_t, chrono::duration<Rep, Period> const& target_time, Lock↵
able& m0, ..., Lockable& m2)

    ~unique_array_locker();

    bool is_locking(lockable_type* l) const;
    bool owns_lock() const;
    typedef unspecified-type bool_type)() const; /*< safe bool idiom >*/
    operator bool_type() const;
    bool operator!() const;

    void lock();
    template<class Clock, class Duration >
    void lock_until(chrono::time_point<Clock, Duration> const& absolute_time);
    template<class Rep, class Period >
    void lock_for(chrono::duration<Rep, Period> const& relative_time);

    void unlock();

    bool try_lock();
```

```
    template<class Clock, class Duration >
    bool try_lock_until(chrono::time_point<Clock, Duration> const& absolute_time);
    template<class Rep, class Period >
    bool try_lock_for(chrono::duration<Rep, Period> const& relative_time);

};
```

Description

`unique_array_locker` template parameters

- `Lockable` : The exclusive lockable type used to synchronize exclusive access
- `N` : The number of lockables in the array

`unique_array_locker` public types

- `lockable_type` : The exclusive lockable type used to synchronize exclusive access
- `lock_error` : The exception type throw in case of errors
- `bool_type` : The bool_type of the safe_bool idiom

`unique_array_locker` private and not defined member functions

- `unique_array_locker()`
- `unique_array_locker(unique_array_locker&);`
- `operator=(unique_array_locker&);`

Template Class `try_unique_array_locker<>`

Synopsis

Single-threaded

Header <boost/synchro/null_mutex.hpp>

```
namespace boost { namespace synchro {
    class null_condition;
    class null_mutex : public lock_traits_base<
        mono_threaded_tag,
        upgradable_lock_tag,
        recursive_tag,
        has_timed_interface_tag,
        kernel_lifetime_tag,
        anonymous_tag,
        void
    >
    {
       null_mutex(const null_mutex&);
       null_mutex &operator= (const null_mutex&);
    public:
        typedef null_condition condition_type;
        typedef null_condition condition_any_type;

       null_mutex();
       ~null_mutex();

       void lock();
       bool try_lock();
       bool try_lock_until(const boost::posix_time::ptime &);
       template<typename TimeDuration>
       bool try_lock_for(TimeDuration const & relative_time);

       void unlock();

       void lock_shared(){};
       bool try_lock_shared();
       bool try_lock_shared_until(const boost::posix_time::ptime &);
       template<typename DurationType>
       bool try_lock_shared_for(DurationType const& rel_time)

       void unlock_shared();

       void lock_upgrade();
       bool try_lock_upgrade();
       bool timed_lock_upgrade(boost::posix_time::ptime const &);

        void unlock_upgrade();

       void unlock_and_lock_upgrade();

       void unlock_and_lock_shared();

       void unlock_upgrade_and_lock_shared();

       void unlock_upgrade_and_lock();

       bool try_unlock_upgrade_and_lock();

       bool timed_unlock_upgrade_and_lock(const boost::posix_time::ptime &);
```

```
        bool try_unlock_share_and_lock();

        bool try_unlock_share_and_lock_upgrade();
    };

}}
```

Header <boost/synchro/null_condition.hpp>

```
namespace boost { namespace synchro {

    class null_condition {
    private:
       null_condition(const null_condition &);
       null_condition &operator=(const null_condition &);
    public:
       null_condition();
       ~null_condition();

       void notify_one();

       void notify_all();

       template <typename L>
       void wait(L& lock);

       template <typename L, typename Pr>
       void wait(L& lock, Pr pred);

       template <typename L>
       bool timed_wait(L& lock, const boost::posix_time::ptime &abs_time);

       template <typename L, typename Pr>
       bool timed_wait(L& lock, const boost::posix_time::ptime &abs_time, Pr pred);

    };
    template <>
    struct best_condition<null_mutex> {
        typedef null_condition type;
    };
}}
```

Header <boost/synchro/null_synchronization_family.hpp>

```
namespace boost { namespace synchro {
    struct null_synchronization_family;
}}
```

Class null_synchronization_policy

Synopsis

```
struct null_synchronization_family
{
   typedef boost::synchro::null_mutex  mutex_type;
   typedef boost::synchro::null_mutex  recursive_mutex_type;
   typedef boost::synchro::null_mutex    timed_mutex_type;
   typedef boost::synchro::null_mutex    recursive_timed_mutex_type;
   typedef boost::synchro::null_mutex    shared_mutex_type;
   typedef boost::synchro::null_condition   condition_type;
   typedef boost::synchro::null_condition   condition_any_type;
};
```

Multi-threaded

Header <boost/synchro/thread/lockable_scope_traits.hpp>

```
namespace boost { namespace synchro {
    template<> struct scope_traits<multi_threaded_tag>;
    template<typename Lockable>
    struct lockable_scope_traits<multi_threaded_tag, Lockable>;
}}
```

Template Class Specialization scope_traits<multi_threaded_tag>

Synopsis

```
template<> struct scope_traits<multi_threaded_tag> {
    typedef boost::lock_error lock_error;

    template <typename T>
    struct moved_object : boost::detail::thread_move_t<T> {
        moved_object(T& t_): boost::detail::thread_move_t<T>(t_) {}
    };

    typedef boost::defer_lock_t defer_lock_t;
    typedef boost::adopt_lock_t adopt_lock_t;
    typedef boost::try_to_lock_t try_to_lock_t;

    static const defer_lock_t& defer_lock() {return boost::defer_lock;}
    static const adopt_lock_t& adopt_lock() {return boost::adopt_lock;}
    static const try_to_lock_t& try_to_lock() {return boost::try_to_lock;}
};
```

Template Class Specialization lockable_scope_traits<multi_threaded_tag, Lockable>

```
template<typename Lockable>
struct lockable_scope_traits<multi_threaded_tag, Lockable> : scope_traits<multi_threaded_tag> {
    typedef Lockable                            lockable_type;
    typedef boost::unique_lock<lockable_type>   scoped_lock;
    typedef boost::unique_lock<lockable_type>   unique_lock;
    typedef boost::shared_lock<lockable_type>   shared_lock;
    typedef boost::upgrade_lock<lockable_type>  upgrade_lock;
};
```

Header <boost/synchro/thread/mutex.hpp>

```
namespace boost { namespace synchro {
    class thread_mutex;
    template <> struct unique_lock_type<thread_mutex>;
    template <> struct shared_lock_type<thread_mutex>;
    template <> struct upgrade_lock_type<thread_mutex>;
    template <> struct upgrade_to_unique_locker_type<thread_mutex>;
    template <> struct lock_error_type<boost::mutex>;

    class thread_timed_mutex;
}}
```

Class thread_mutex

Synopsis

```
class thread_mutex : public lock_traits_base<
        multi_threaded_tag,
        exclusive_lock_tag,
        non_recursive_tag,
        hasnt_timed_interface_tag,
        process_lifetime_tag,
        anonymous_tag,
        mutex
    > {
public:
    // public types
    typedef boost::condition_variable  best_condition_type;
    typedef boost::condition_variable_any  best_condition_any_type;

    //Non-copyable
    BOOST_COPY_CONSTRUCTOR_DELETE(thread_mutex) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(thread_mutex) /*< disable copy asignement >*/

    thread_mutex() {}
};

template <>
struct unique_lock_type<thread_mutex> {
    typedef boost::unique_lock<boost::mutex> type;
};

template <>
struct shared_lock_type<thread_mutex> {
    typedef boost::shared_lock<boost::mutex> type;
};

template <>
struct upgrade_lock_type<thread_mutex> {
    typedef boost::upgrade_lock<boost::mutex> type;
};

template <>
struct upgrade_to_unique_locker_type<thread_mutex> {
    typedef boost::upgrade_to_unique_lock<boost::mutex> type;
};

template <>
struct lock_error_type<boost::mutex> {
    typedef boost::lock_error type;
};
```

Class `thread_timed_mutex`

```
class thread_timed_mutex : public lock_traits_base<
        multi_threaded_tag,
        exclusive_lock_tag,
        non_recursive_tag,
        has_timed_interface_tag,
        process_lifetime_tag,
        anonymous_tag,
        timed_mutex
    > {
public:
    typedef boost::condition_variable_any  best_condition_type;
    typedef boost::condition_variable_any  best_condition_any_type;

    //Non-copyable
    BOOST_COPY_CONSTRUCTOR_DELETE(thread_timed_mutex) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(thread_timed_mutex) /*< disable copy asignement >*/
    thread_timed_mutex ();

    bool try_lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    bool try_lock_for(TimeDuration const & relative_time);

    void lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    void lock_for(TimeDuration const & relative_time);
};
```

Header `<boost/synchro/thread_recursive_mutex.hpp>`

```
namespace boost { namespace synchro {
    class thread_recursive_mutex;

    class thread_timed_mutex;
}}
```

Class `thread_recursive_mutex`

```
class thread_recursive_mutex : public lock_traits_base<
        multi_threaded_tag,
        exclusive_lock_tag,
        recursive_tag,
        hasnt_timed_interface_tag,
        process_lifetime_tag,
        anonymous_tag,
        recursive_mutex
    > {
public:
    typedef boost::condition_variable_any  best_condition_type;
    typedef boost::condition_variable_any  best_condition_any_type;

    //Non-copyable
    BOOST_COPY_CONSTRUCTOR_DELETE(thread_recursive_mutex) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(thread_recursive_mutex) /*< disable copy asignement >*/
    thread_recursive_mutex() {}
};
```

Class `thread_recursive_timed_mutex`

```
class thread_recursive_timed_mutex : public lock_traits_base<
        multi_threaded_tag,
        exclusive_lock_tag,
        recursive_tag,
        has_timed_interface_tag,
        process_lifetime_tag,
        anonymous_tag,
        recursive_timed_mutex
    > {
public:
    typedef boost::condition_variable_any  best_condition_type;
    typedef boost::condition_variable_any  best_condition_any_type;

    //Non-copyable
    BOOST_COPY_CONSTRUCTOR_DELETE(thread_recursive_timed_mutex) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(thread_recursive_timed_mutex) /*< disable copy asignement >*/
    thread_recursive_timed_mutex();

    bool try_lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    bool try_lock_for(TimeDuration const & relative_time);

    void lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    void lock_for(TimeDuration const & relative_time);

};
```

Header `<boost/synchro/thread/shared_mutex.hpp>`

```
namespace boost { namespace synchro {
    class thread_recursive_mutex;

    class thread_timed_mutex;
}}
```

Class `thread_shared_mutex`

Synopsis

```
class thread_shared_mutex : public lock_traits_base<
        multi_threaded_tag,
        upgradable_lock_tag,
        non_recursive_tag,
        has_timed_interface_tag,
        process_lifetime_tag,
        anonymous_tag,
        shared_mutex
    > {
public:
    typedef boost::condition_variable_any  best_condition_type;
    typedef boost::condition_variable_any  best_condition_any_type;

    //Non-copyable
    BOOST_COPY_CONSTRUCTOR_DELETE(thread_shared_mutex) /*< disable copy construction >*/
    BOOST_COPY_ASSIGNEMENT_DELETE(thread_shared_mutex) /*< disable copy asignement >*/
    thread_shared_mutex();

    bool try_lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    bool try_lock_for(TimeDuration const & relative_time);

    void lock_until(system_time const & abs_time);
    template<typename TimeDuration>
    void lock_for(TimeDuration const & relative_time);

    bool try_lock_shared_until(system_time const& abs_time);
    template<typename TimeDuration>
    bool try_lock_shared_for(TimeDuration const& rel_time);

    void lock_shared_until(system_time const& abs_time);
    template<typename TimeDuration>
    void lock_shared_for(TimeDuration const& abs_time);

};
```

Header <boost/synchro/thread/locks.hpp>

```
namespace boost { namespace synchro {
    class thread_recursive_mutex;

    class thread_timed_mutex;
}}
```

Template Class `unique_locker<>`

```
template <typename T>
struct lockable_type<boost::unique_lock<T> > {
    typedef T type;
};
template <typename T>
struct lockable_type<boost::shared_lock<T> > {
    typedef T type;
};
template <typename T>
struct lockable_type<boost::upgrade_lock<T> > {
    typedef T type;
};

template <typename T>
struct lockable_type<boost::upgrade_to_unique_lock<T> > {
    typedef T type;
};

template<typename Mutex>
class unique_locker<Mutex,multi_threaded_tag>:  public unique_lock_type<Mutex>::type {
    //typename scope_tag<Mutex>::type == multi_threaded_tag
private:
    typedef Mutex lockable_type;
    typedef multi_threaded_tag scope_tag_type;
    typedef typename unique_lock_type<Mutex>::type base_type;

public:
    BOOST_NON_CONST_COPY_CONSTRUCTOR_DELETE(unique_locker) /*< disable copy construction >*/
    BOOST_NON_CONST_COPY_ASSIGNEMENT_DELETE(unique_locker) /*< disable copy asignement >*/
    unique_locker();

    explicit unique_locker(Mutex& m_);
    unique_locker(Mutex& m_,adopt_lock_t);
    unique_locker(Mutex& m_,defer_lock_t);
    unique_locker(Mutex& m_,try_to_lock_t);
    template<typename TimeDuration>
    unique_locker(Mutex& m_,TimeDuration const& target_time);
    unique_locker(Mutex& m_,system_time const& target_time);
    template<typename TimeDuration>
    unique_locker(Mutex& m_,TimeDuration const& target_time, throw_lock_t);
    unique_locker(Mutex& m_,system_time const& target_time, throw_lock_t);
    template<typename TimeDuration>
    unique_locker(TimeDuration const& target_time, Mutex& m_);
    unique_locker(system_time const& target_time, Mutex& m_);

#ifdef BOOST_HAS_RVALUE_REFS
    unique_locker(unique_locker&& other);
    explicit unique_locker(upgrade_locker<Mutex,scope_tag_type>&& other);

    unique_locker<Mutex, scope_tag_type>&& move();


    unique_locker& operator=(unique_locker<Mutex, scope_tag_type>&& other);
    unique_locker& operator=(upgrade_locker<Mutex, multi_threaded_tag>&& other);

    void swap(unique_locker&& other);
#else
    unique_locker(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);
    unique_locker(detail::thread_move_t<upgrade_locker<Mutex,scope_tag_type> > other);

    operator detail::thread_move_t<unique_locker<Mutex, scope_tag_type> >();
```

```
    detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > move();

    unique_locker& operator=(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);
    unique_locker& operator=(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > other);
    void swap(unique_locker& other);
    void swap(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);
#endif

    ~unique_locker() {}

    Mutex* mutex() const;
    bool is_locking(lockable_type* l) const;

    template<typename TimeDuration>
    bool try_lock_for(TimeDuration const& relative_time);
    bool try_lock_until(::boost::system_time const& absolute_time);

    template<typename TimeDuration>
    void lock_for(TimeDuration const& relative_time);
    void lock_until(::boost::system_time const& absolute_time);
};
```

Template Class `try_unique_locker<>`

```
template<typename Mutex>
class try_unique_locker<Mutex,multi_threaded_tag>:  public unique_locker<Mu↵
tex,multi_threaded_tag> {
    //typename scope_tag<Mutex>::type == multi_threaded_tag
private:
    typedef Mutex lockable_type;
    typedef multi_threaded_tag scope_tag_type;
    typedef unique_locker<Mutex,multi_threaded_tag> base_type;
public:
    try_unique_locker();

    explicit try_unique_locker(Mutex& m_);
    try_unique_locker(Mutex& m_,force_lock_t);
    try_unique_locker(Mutex& m_,adopt_lock_t);
    try_unique_locker(Mutex& m_,defer_lock_t);
    try_unique_locker(Mutex& m_,try_to_lock_t);
    template<typename TimeDuration>
    try_unique_locker(Mutex& m_,TimeDuration const& target_time);
    try_unique_locker(Mutex& m_,system_time const& target_time);
    template<typename TimeDuration>
    try_unique_locker(Mutex& m_,TimeDuration const& target_time, throw_lock_t);
    try_unique_locker(Mutex& m_,system_time const& target_time, throw_lock_t);
    template<typename TimeDuration>
    try_unique_locker(TimeDuration const& target_time, Mutex& m_);
    try_unique_locker(system_time const& target_time, Mutex& m_);

#ifdef BOOST_HAS_RVALUE_REFS
    try_unique_locker(try_unique_locker&& other);
    explicit try_unique_locker(upgrade_locker<Mutex,scope_tag_type>&& other);

    try_unique_locker<Mutex, scope_tag_type>&& move();
    try_unique_locker& operator=(try_unique_locker<Mutex, scope_tag_type>&& other);
    try_unique_locker& operator=(upgrade_locker<Mutex, multi_threaded_tag>&& other);
    void swap(try_unique_locker&& other);
#else
    try_unique_locker(detail::thread_move_t<try_unique_locker<Mutex, scope_tag_type> > other);
    try_unique_locker(detail::thread_move_t<upgrade_locker<Mutex,scope_tag_type> > other);

    operator detail::thread_move_t<try_unique_locker<Mutex, scope_tag_type> >();
    detail::thread_move_t<try_unique_locker<Mutex, scope_tag_type> > move();

    try_unique_locker& operator=(detail::thread_move_t<try_unique_locker<Mu↵
tex, scope_tag_type> > other);
   try_unique_locker& operator=(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > oth↵
er);

    void swap(try_unique_locker& other);
    void swap(detail::thread_move_t<try_unique_locker<Mutex, scope_tag_type> > other);
#endif

    ~try_unique_locker() {}
};
```

Template Class `shared_locker<>`

```
template<typename Mutex>
class shared_locker<Mutex,multi_threaded_tag>:  public shared_lock_type<Mutex>::type {
    //typename scope_tag<Mutex>::type == multi_threaded_tag
public:
    typedef Mutex lockable_type;
    typedef multi_threaded_tag scope_tag_type;
    typedef typename shared_lock_type<Mutex>::type base_type;

    shared_locker();
    BOOST_NON_CONST_COPY_CONSTRUCTOR_DELETE(shared_locker) /*< disable copy construction >*/
    BOOST_NON_CONST_COPY_ASSIGNEMENT_DELETE(shared_locker) /*< disable copy asignement >*/

    explicit shared_locker(Mutex& m_)
    shared_locker(Mutex& m_,adopt_lock_t);
    shared_locker(Mutex& m_,defer_lock_t);
    shared_locker(Mutex& m_,try_to_lock_t);
    shared_locker(Mutex& m_,system_time const& target_time);
    template<typename TimeDuration>
    shared_locker(Mutex& m_,TimeDuration const& target_time);
    shared_locker(Mutex& m_,system_time const& target_time,throw_lock_t);
    template<typename TimeDuration>
    shared_locker(Mutex& m_,TimeDuration const& target_time,throw_lock_t);
    shared_locker(Mutex& m_,system_time const& target_time,throw_lock_t);
    template<typename TimeDuration>
    shared_locker(TimeDuration const& target_time, Mutex& m_);
    shared_locker(system_time const& target_time, Mutex& m_);

    shared_locker(detail::thread_move_t<shared_locker<Mutex, scope_tag_type> > other);
    shared_locker(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);
    shared_locker(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > other);

    operator detail::thread_move_t<shared_locker<Mutex,scope_tag_type> >();
    detail::thread_move_t<shared_locker<Mutex, scope_tag_type> > move();

   shared_locker& operator=(detail::thread_move_t<shared_locker<Mutex, scope_tag_type> > other);
    shared_locker& operator=(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);
    shared_locker& operator=(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > other);

#ifdef BOOST_HAS_RVALUE_REFS
    void swap(shared_locker&& other);
#else
    void swap(shared_locker& other);
    void swap(boost::detail::thread_move_t<shared_locker<Mutex, scope_tag_type> > other);
#endif

    Mutex* mutex() const;
    bool is_locking(lockable_type* l) const;

    bool try_lock_until(boost::system_time const& absolute_time);
    template<typename Duration>
    bool try_lock_for(Duration const& relative_time);

    template<typename TimeDuration>
    void lock_for(TimeDuration const& relative_time);
    void lock_until(::boost::system_time const& absolute_time);
};
```

Template Class `upgrade_locker<>`

```
template<typename Mutex>
class upgrade_locker<Mutex,multi_threaded_tag>:  public upgrade_lock_type<Mutex>::type {
    //typename scope_tag<Mutex>::type == multi_threaded_tag
public:
    typedef Mutex lockable_type;
    typedef multi_threaded_tag scope_tag_type;
    typedef typename upgrade_lock_type<Mutex>::type base_type;

    upgrade_locker();
    BOOST_NON_CONST_COPY_CONSTRUCTOR_DELETE(upgrade_locker) /*< disable copy construction >*/
    BOOST_NON_CONST_COPY_ASSIGNEMENT_DELETE(upgrade_locker) /*< disable copy asignement >*/

    explicit upgrade_locker(Mutex& m_);
    upgrade_locker(Mutex& m_,adopt_lock_t);
    upgrade_locker(Mutex& m_,defer_lock_t);
    upgrade_locker(Mutex& m_,try_to_lock_t);

    upgrade_locker(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > other);
    upgrade_locker(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);

    operator detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> >();
    detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > move();

   upgrade_locker& operator=(detail::thread_move_t<upgrade_locker<Mutex, scope_tag_type> > other);
   upgrade_locker& operator=(detail::thread_move_t<unique_locker<Mutex, scope_tag_type> > other);

    void swap(upgrade_locker& other);

    ~upgrade_locker();
    Mutex* mutex() const;

    bool is_locking(lockable_type* l) const;
};
```

Template Class `upgrade_to_unique_locker<>`

```
template<typename Mutex>
class upgrade_to_unique_locker<Mutex,multi_threaded_tag>:  public upgrade_to_unique_locker_type<Mu↵
tex>::type {
    //typename scope_tag<Mutex>::type == multi_threaded_tag
public:
    typedef Mutex lockable_type;
    typedef multi_threaded_tag scope_tag_type;
    typedef typename upgrade_to_unique_locker_type<Mutex>::type base_type;
public:
    BOOST_NON_CONST_COPY_CONSTRUCTOR_DELETE(upgrade_locker) /*< disable copy construction >*/
    BOOST_NON_CONST_COPY_ASSIGNEMENT_DELETE(upgrade_locker) /*< disable copy asignement >*/

    explicit upgrade_to_unique_locker(upgrade_locker<Mutex,multi_threaded_tag>& m_);
    ~upgrade_to_unique_locker();

    upgrade_to_unique_locker(detail::thread_move_t<upgrade_to_unique_locker<Mu↵
tex, scope_tag_type> > other);

    upgrade_to_unique_locker& operator=(detail::thread_move_t<upgrade_to_unique_locker<Mu↵
tex, scope_tag_type> > other);
    void swap(upgrade_to_unique_locker& other);

    Mutex* mutex() const;
    bool is_locking(lockable_type* l) const;
};
```

Header `<boost/synchro/thread_synchronization_family.hpp>`

Class `thread_synchronization_family`

Synopsis

```
struct thread_synchronization_family
{
    typedef thread_mutex                    mutex_type;
    typedef thread_recursive_mutex          recursive_mutex_type;
    typedef thread_timed_mutex              timed_mutex_type;
    typedef thread_recursive_timed_mutex    recursive_timed_mutex_type;
    typedef thread_shared_mutex             shared_mutex_type;
    typedef boost::condition_variable       condition_type;
    typedef boost::condition_variable_any   condition_type_any;
};
```

Multi-process

Header `<boost/synchro/thread/lockable_scope_traits.hpp>`

```
namespace boost { namespace synchro {
    template<> struct scope_traits<multi_process_tag>
    template<typename Lockable>
    struct lockable_scope_traits<multi_process_tag, Lockable>;
}}
```

Template Class Specialization `scope_traits<multi_process_tag>`

```
template<>
struct scope_traits<multi_process_tag> {
    typedef boost::interprocess::lock_exception lock_error;

    template <typename T>
    struct moved_object : boost::interprocess::detail::moved_object<T> {
        moved_object(T& t_): boost::interprocess::detail::moved_object<T>(t_) {}
    };

    typedef boost::interprocess::defer_lock_type defer_lock_t;
    typedef boost::interprocess::accept_ownership_type adopt_lock_t;
    typedef boost::interprocess::try_to_lock_type try_to_lock_t;

    static const defer_lock_t& defer_lock()    {return boost::interprocess::defer_lock;}
    static const adopt_lock_t& adopt_lock() {return boost::interprocess::accept_ownership;}
    static const try_to_lock_t& try_to_lock()    {return boost::interprocess::try_to_lock;}
};
```

Template Class Specialization `lockable_scope_traits<multi_process_tag, Lockable>`

```
template<typename Lockable>
struct lockable_scope_traits<multi_process_tag, Lockable> : scope_traits<multi_process_tag> {
    typedef Lockable                                          lockable_type;
    typedef boost::interprocess::scoped_lock<lockable_type>     scoped_lock;
    typedef boost::interprocess::scoped_lock<lockable_type>     unique_lock;
    typedef boost::interprocess::sharable_lock<lockable_type>   shared_lock;
    typedef boost::interprocess::upgradable_lock<lockable_type> upgrade_lock;

};
```

Header `<boost/synchro/process_synchronization_family.hpp>`

```
namespace boost { namespace synchro {
    struct process_synchronization_family;
}}
```

Class `process_synchronization_family`

Synopsis

```
struct process_synchronization_family {
   typedef boost::synchro::interprocess_mutex              mutex_type;
   typedef boost::synchro::interprocess_recursive_mutex    recursive_mutex_type;
   typedef boost::synchro::interprocess_mutex              timed_mutex_type;
   typedef boost::synchro::interprocess_recursive_mutex    recursive_timed_mutex_type;
   typedef boost::synchro::interprocess_upgradable_mutex   shared_mutex_type;
   typedef boost::interprocess::interprocess_condition     condition_type;
   typedef boost::interprocess::interprocess_condition     condition_any_type;
};
```

Polymorphic Locks

Header <boost/synchro/poly/lock.hpp>

```
namespace boost { namespace synchro {
    namespace poly {
        struct exclusive_lock;
        struct timed_lock;
        struct sharable_lock;
        struct upgradable_lock;
    }
}}
```

Abstract Class exclusive_lock

Polymorphic exclusive lock interface.

```
struct exclusive_lock {
    virtual ~exclusive_lock()=0;
    virtual void lock()=0;
    virtual void unlock()=0;
    virtual bool try_lock()=0;
};
```

Abstract Class timed_lock

Polymorphic timed lock interface.

```
struct timed_lock : exclusive_lock {
    virtual ~timed_lock()=0;
    bool try_lock_until(boost::system_time  const&  abs_time)=0;
    template<typename DurationType>
    bool try_lock_for(DurationType const& rel_time)
    {
        return try_lock_until(get_system_time()+abs_time);
    }
};
```

Abstract Class sharable_lock

Polymorphic sharable lock interface.

```
struct sharable_lock : timed_lock {
    virtual ~sharable_lock();
    virtual void lock_shared()=0;
    virtual bool try_lock_shared()=0;
    virtual bool try_lock_shared_until(boost::system_time  const&  abs_time)=0;
    virtual void unlock_shared()=0;
    template<typename DurationType>
    bool try_lock_shared_for(DurationType const& rel_time)
    {
        return try_lock_shared_until(get_system_time()+abs_time);
    }
};
```

Abstract Class upgradable_lock

Polymorphic upgradable lock interface.

```
struct upgradable_lock : sharable_lock {
    virtual ~upgradable_lock();
    virtual void lock_upgrade()=0;
    virtual void unlock_upgrade()=0;
    virtual void unlock_upgrade_and_lock()=0;
    virtual void unlock_and_lock_upgrade()=0;
    virtual void unlock_and_lock_shared()=0;
    virtual void unlock_upgrade_and_lock_shared()=0;
};
```

Header <boost/synchro/poly/lock_adpater.hpp>

namespace boost { namespace synchro { namespace poly { template <typename Lockable> class exclusive_lock_adapter; template <typename TimeLockable> class timed_lock_adapter; template <typename ShareLockable> class sharable_lock_adapter; template <typename UpgradaLockable> class upgradable_lock_adapter; } }}

Template Class exclusive_lock_adapter

```
template <typename Lockable>
class exclusive_lock_adapter
    : public virtual exclusive_lock
{
    exclusive_lock_adapter();
    ~exclusive_lock_adapter();
    void lock();
    void unlock();
    bool try_lock();
protected:
    Lockable lock_;
};
```

Template Class timed_lock_adapter

```
template <typename TimeLockable>
class timed_lock_adapter
        : public exclusive_lock_adapter<TimeLockable>
        , public virtual timed_lock
{
public:
    ~timed_lock_adapter();
    bool try_lock_until(boost::system_time  const&  abs_time);
    template<typename DurationType>
    bool try_lock_for(DurationType const& rel_time);
};
```

Template Class sharable_lock_adapter

```
template <typename ShareLockable>
class sharable_lock_adapter
        : public exclusive_lock_adapter<ShareLockable>
        , public virtual sharable_lock    {
public:
    ~sharable_lock_adapter();
    void lock_shared();
    bool try_lock_shared();
    void unlock_shared();
};
```

Template Class `upgradable_lock`

```
template <typename UpgradaLockable>
class upgradable_lock_adapter
    : public sharable_lock_adapter<UpgradaLockable>
    , public virtual upgradable_lock
{
public:
    virtual ~upgradable_lock_adapter();
    virtual void lock_upgrade();

    virtual void unlock_upgrade();

    virtual void unlock_upgrade_and_lock();
    virtual void unlock_and_lock_upgrade();
    virtual void unlock_and_lock_shared();
    virtual void unlock_upgrade_and_lock_shared();
};
```

Other

Header `<boost/synchro/semaphore.hpp>`

```
namespace boost { namespace synchro {

    template <typename Sync=thread_synchronization_family>
    class basic_semaphore {
        basic_semaphore(const basic_semaphore &);
        this_type& operator=(const basic_semaphore &);
    public:
        basic_semaphore(int initialCount);
        ~basic_semaphore();
        void post();
        void wait();
        bool try_wait();
        bool wait_until(const system_time &abs_time);
        template<typename TimeDuration>
        bool wait_for(const TimeDuration &rel_time);
    };

    typedef basic_semaphore<> semaphore;

}}
```

High Level

Header <boost/synchro/monitor.hpp>

```
namespace boost { namespace synchro {
    template <
          typename Lockable=thread_mutex
        , class Condition=condition_safe<typename best_condition<Lockable>::type >
        , typename ScopeTag=typename scope_tag<Lockable>::type
    >
    class exclusive_monitor;

    template <
          typename Lockable=thread_shared_mutex,
        , class Condition=condition_safe<typename best_condition_any<Lockable>::type >
        , typename ScopeTag=typename scope_tag<Lockable>::type
    >
    class shared_monitor;

    template <
          typename Lockable=thread_mutex
        , typename lock_tag=typename category_tag<Lockable>::type
        , typename ScopeTag=typename scope_tag<Lockable>::type
    >
    struct monitor;
}}
```

Template Class exclusive_monitor<>

```
template <
    typename Lockable=thread_mutex
,   class Condition=condition_safe<typename best_condition<Lockable>::type >
,   typename ScopeTag=typename scope_tag<Lockable>::type
>
class exclusive_monitor : protected lockable_adapter<Lockable> {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
protected:
    typedef Condition condition;
    typedef condition_unique_locker<Lockable, Condition, ScopeTag> synchronizer;
};
```

Template Class shared_monitor<>

```
template <
    typename Lockable=thread_shared_mutex,
,   class Condition=condition_safe<typename best_condition_any<Lockable>::type >
,   typename ScopeTag=typename scope_tag<Lockable>::type
>
class shared_monitor : protected lockable_adapter<Lockable> {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));
protected:
    typedef Condition condition;
    typedef condition_unique_locker<Lockable, Condition, ScopeTag> synchronizer;
    typedef condition_shared_locker<Lockable, Condition, ScopeTag> shared_synchronizer;
};
```

Template Class `monitor<>`

```
template <
      typename Lockable=thread_mutex
    , typename lock_tag=typename category_tag<Lockable>::type
    , typename ScopeTag=typename scope_tag<Lockable>::type
> struct monitor;

template <typename Lockable, typename ScopeTag>
struct monitor<Lockable, exclusive_lock_tag, ScopeTag>
    : protected exclusive_monitor<Lockable, ScopeTag>
{};

template <typename Lockable, typename ScopeTag>
struct monitor<Lockable, sharable_lock_tag, ScopeTag>
    : protected shared_monitor<Lockable, ScopeTag>
{};

template <typename Lockable, typename ScopeTag>
struct monitor<Lockable, upgradable_lock_tag, ScopeTag>
    : protected shared_monitor<Lockable, ScopeTag>
{};
```

Examples

This section includes complete examples using the library.

[/

Appendices

Appendix A: History

Version 0.3.3, May 24, 2009 *Extraction of Boost.Rendez-Vous*

Version 0.3.2, May 08, 2009 *Adaptation Boost 1.39 + Extraction of Boost.Conversion*

Version 0.3.1, Mars 29, 2009 *Language-like Synchronized Block Macros*

New Features:

- Language-like Synchronized Block Macros.

Version 0.3.0, Mars 19, 2009 *Generic free operations on multiple lockables + Usage of Boost.Chrono*

New Features:

- Generic free functions on lockables: `lock`, `try_lock`
- Complete the generic free functions on multiple lockables in Boost.Thread `lock`, `try_lock` with:
  - `lock_until`, `lock_for`, `try_lock_until`, `try_lock_for`, `unlock`

Version 0.2.0, Mars 1, 2009 *binary_semaphore + array_locker*

New Features:

- `binary_semaphore` emulation with mutex
- `array_locker` locker containers.

Version 0.1.0, Febraury 16, 2009 *Announcement of Synchro*

Features:

- A uniform usage of Boost.Thread and Boost.Interprocess synchronization mechanisms based on lockables(mutexes) concepts and locker(guards) concepts.
  - lockables traits and lock generators,
  - lock adapters of the Boost.Thread and Boost.Interprocess lockable models,
  - locker adapters of the Boost.Thread and Boost.Interprocess lockers models,
  - complete them with the corresponding models for single-threaded programms: `null_mutex` and `null_condition` classes,
  - locking families,
  - `semaphore`,
  - `condition_lockable` lock put toghether a lock and its associated conditions.
- Exception based timed lockables and lockers,
- A rich palete of lockers as
  - `strict_locker`, `nested_strict_locker`,
  - `condition_locker`,
  - `reverse_locker`, `nested_reverse_locker`,
  - `locking_ptr`, `on_derreference_locking_ptr`,
  - `externally_locked`,
- A polymorphic lockable hierarchy.
- High-level abstractions for handling more complicated synchronization problems, including
  - `monitor` for guaranteeing exclusive access to an object, and

Appendix B: Rationale

TBC

Appendix C: Implementation Notes

TBC

Appendix D: Acknowledgements

TBC

Appendix E: Tests

Lockables

Name	kind	Description	Result	Ticket
lockable_traits	compile	static assertion on lockables traits	Pass	#
lockable_concepts	compile	static assertion on lockables concepts	Pass	#

Lockers

Name	kind	Description	Result	Ticket
locker_concepts	compile	static assertion on lockers concepts	Pass	#
nested_reverse_locker	run	nested_reverse_locker tests	Pass	#

Others

Name	kind	Description	Result	Ticket
sync_buffer_family	run	synchronous buffer with synchronization family tests	Pass	#
sync_buffer_monitor	run	monitor synchronous buffer tests	Fail	v0.2#1

Examples

Name	Kind	Description	Result	Ticket
BankAccount	run	tests	Pass	#
IL_BancAccount	run	tests	Pass	#
IL_Lockable_BancAccount	run-fail	tests	Pass	#
IL_Rec_Lockable_BancAccount	run	tests	Pass	#
IEL_Rec_Lockable_BancAccount	run	tests	Pass	#
EL_BancAccount	run	tests	Pass	#

Appendix F: Tickets

Kind	Identifier	Description	Resolution	State	Tests	Version
feature	v0.1#1	array_locker	See section	Open	See array_locker_tests	v0.2
bug	v0.2#1	sync_buffer_monitor_test fails	---	Open	sync_buffer_monitor_test	v0.2

Appendix E: Future plans

Tasks to do before review

- `tuple_locker` locker containers.

For later releases