Thread

Anthony Williams

Table of Contents

Overview

Boost.Thread enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads.

The **Boost.Thread** library was originally written and designed by William E. Kempf. This version is a major rewrite designed to closely follow the proposals presented to the C++ Standards Committee, in particular N2497, N2320, N2184, N2139, and N2094

In order to use the classes and functions described here, you can either include the specific headers specified by the descriptions of each class or function, or include the master thread library header:

```
#include <boost/thread.hpp>
```

which includes all the other headers in turn.

Changes since boost 1.35

The 1.36.0 release of Boost includes a few new features in the thread library:

- New generic `lock()` and `try_lock()` functions for locking multiple mutexes at once.
- Rvalue reference support for move semantics where the compilers supports it.
- A few bugs fixed and missing functions added (including the serious win32 condition variable bug).
- `scoped_try_lock` types are now backwards-compatible with Boost 1.34.0 and previous releases.
- Support for passing function arguments to the thread function by supplying additional arguments to the `boost::thread` constructor.
- Backwards-compatibility overloads added for `timed_lock` and `timed_wait` functions to allow use of `xtime` for timeouts.

Changes since boost 1.34

Almost every line of code in **Boost.Thread** has been changed since the 1.34 release of boost. However, most of the interface changes have been extensions, so the new code is largely backwards-compatible with the old code. The new features and breaking changes are described below.

New Features

- Instances of `boost::thread` and of the various lock types are now movable.
- Threads can be interrupted at *interruption points*.
- Condition variables can now be used with any type that implements the `Lockable` concept, through the use of `boost::condition_variable_any` (`boost::condition` is a `typedef` to `boost::condition_variable_any`, provided for backwards compatibility). `boost::condition_variable` is provided as an optimization, and will only work with `boost::unique_lock<boost::mutex>` (`boost::mutex::scoped_lock`).
- Thread IDs are separated from `boost::thread`, so a thread can obtain it's own ID (using `boost::this_thread::get_id()`), and IDs can be used as keys in associative containers, as they have the full set of comparison operators.
- Timeouts are now implemented using the Boost DateTime library, through a typedef `boost::system_time` for absolute timeouts, and with support for relative timeouts in many cases. `boost::xtime` is supported for backwards compatibility only.
- Locks are implemented as publicly accessible templates `boost::lock_guard`, `boost::unique_lock`, `boost::shared_lock`, and `boost::upgrade_lock`, which are templated on the type of the mutex. The `Lockable` concept has been extended to include publicly available `lock()` and `unlock()` member functions, which are used by the lock types.

Breaking Changes

The list below should cover all changes to the public interface which break backwards compatibility.

- `boost::try_mutex` has been removed, and the functionality subsumed into `boost::mutex`. `boost::try_mutex` is left as a `typedef`, but is no longer a separate class.

- boost::recursive_try_mutex has been removed, and the functionality subsumed into boost::recursive_mutex. boost::recursive_try_mutex is left as a typedef, but is no longer a separate class.
- boost::detail::thread::lock_ops has been removed. Code that relies on the lock_ops implementation detail will no longer work, as this has been removed, as it is no longer necessary now that mutex types now have public lock() and unlock() member functions.
- scoped_lock constructors with a second parameter of type bool are no longer provided. With previous boost releases,

```
boost::mutex::scoped_lock some_lock(some_mutex,false);
```

  could be used to create a lock object that was associated with a mutex, but did not lock it on construction. This facility has now been replaced with the constructor that takes a boost::defer_lock_type as the second parameter:

```
boost::mutex::scoped_lock some_lock(some_mutex,boost::defer_lock);
```

- The locked() member function of the scoped_lock types has been renamed to owns_lock().
- You can no longer obtain a boost::thread instance representing the current thread: a default-constructed boost::thread object is not associated with any thread. The only use for such a thread object was to support the comparison operators: this functionality has been moved to boost::thread::id.
- The broken boost::read_write_mutex has been replaced with boost::shared_mutex.
- boost::mutex is now never recursive. For Boost releases prior to 1.35 boost::mutex was recursive on Windows and not on POSIX platforms.
- When using a boost::recursive_mutex with a call to boost::condition_variable_any::wait(), the mutex is only unlocked one level, and not completely. This prior behaviour was not guaranteed and did not feature in the tests.

Thread Management

Synopsis

The boost::thread class is responsible for launching and managing threads. Each boost::thread object represents a single thread of execution, or *Not-a-Thread*, and at most one boost::thread object represents a given thread of execution: objects of type boost::thread are not copyable.

Objects of type boost::thread are movable, however, so they can be stored in move-aware containers, and returned from functions. This allows the details of thread creation to be wrapped in a function.

```
boost::thread make_thread();

void f()
{
    boost::thread some_thread=make_thread();
    some_thread.join();
}
```

[Note: On compilers that support rvalue references, boost::thread provides a proper move constructor and move-assignment operator, and therefore meets the C++0x *MoveConstructible* and *MoveAssignable* concepts. With such compilers, boost::thread can therefore be used with containers that support those concepts.

For other compilers, move support is provided with a move emulation layer, so containers must explicitly detect that move emulation layer. See <boost/thread/detail/move.hpp> for details.]

Launching threads

A new thread is launched by passing an object of a callable type that can be invoked with no parameters to the constructor. The object is then copied into internal storage, and invoked on the newly-created thread of execution. If the object must not (or cannot) be copied, then `boost::ref` can be used to pass in a reference to the function object. In this case, the user of **Boost.Thread** must ensure that the referred-to object outlives the newly-created thread of execution.

```
struct callable
{
    void operator()();
};

boost::thread copies_are_safe()
{
    callable x;
    return boost::thread(x);
} // x is destroyed, but the newly-created thread has a copy, so this is OK

boost::thread oops()
{
    callable x;
    return boost::thread(boost::ref(x));
} // x is destroyed, but the newly-created thread still has a reference
  // this leads to undefined behaviour
```

If you wish to construct an instance of `boost::thread` with a function or callable object that requires arguments to be supplied, this can be done by passing additional arguments to the `boost::thread` constructor:

```
void find_the_question(int the_answer);

boost::thread deep_thought_2(find_the_question,42);
```

The arguments are *copied* into the internal thread structure: if a reference is required, use `boost::ref`, just as for references to callable functions.

There is an unspecified limit on the number of additional arguments that can be passed.

Exceptions in thread functions

If the function or callable object passed to the `boost::thread` constructor propagates an exception when invoked that is not of type `boost::thread_interrupted`, `std::terminate()` is called.

Joining and detaching

When the `boost::thread` object that represents a thread of execution is destroyed the thread becomes *detached*. Once a thread is detached, it will continue executing until the invocation of the function or callable object supplied on construction has completed, or the program is terminated. A thread can also be detached by explicitly invoking the `detach()` member function on the `boost::thread` object. In this case, the `boost::thread` object ceases to represent the now-detached thread, and instead represents *Not-a-Thread*.

In order to wait for a thread of execution to finish, the `join()` or `timed_join()` member functions of the `boost::thread` object must be used. `join()` will block the calling thread until the thread represented by the `boost::thread` object has completed. If the thread of execution represented by the `boost::thread` object has already completed, or the `boost::thread` object represents *Not-a-Thread*, then `join()` returns immediately. `timed_join()` is similar, except that a call to `timed_join()` will also return if the thread being waited for does not complete when the specified time has elapsed.

Interruption

A running thread can be *interrupted* by invoking the `interrupt()` member function of the corresponding `boost::thread` object. When the interrupted thread next executes one of the specified *interruption points* (or if it is currently *blocked* whilst executing one) with interruption enabled, then a `boost::thread_interrupted` exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of `boost::this_thread::disable_interruption`. Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction:

```
void f()
{
    // interruption enabled here
    {
        boost::this_thread::disable_interruption di;
        // interruption disabled
        {
            boost::this_thread::disable_interruption di2;
            // interruption still disabled
        } // di2 destroyed, interruption state restored
        // interruption still disabled
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

The effects of an instance of `boost::this_thread::disable_interruption` can be temporarily reversed by constructing an instance of `boost::this_thread::restore_interruption`, passing in the `boost::this_thread::disable_interruption` object in question. This will restore the interruption state to what it was when the `boost::this_thread::disable_interruption` object was constructed, and then disable interruption again when the `boost::this_thread::restore_interruption` object is destroyed.

```
void g()
{
    // interruption enabled here
    {
        boost::this_thread::disable_interruption di;
        // interruption disabled
        {
            boost::this_thread::restore_interruption ri(di);
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

At any point, the interruption state for the current thread can be queried by calling `boost::this_thread::interruption_enabled()`.

Predefined Interruption Points

The following functions are *interruption points*, which will throw `boost::thread_interrupted` if interruption is enabled for the current thread, and interruption is requested for the current thread:

- `boost::thread::join()`
- `boost::thread::timed_join()`

- `boost::condition_variable::wait()`
- `boost::condition_variable::timed_wait()`
- `boost::condition_variable_any::wait()`
- `boost::condition_variable_any::timed_wait()`
- `boost::thread::sleep()`
- `boost::this_thread::sleep()`
- `boost::this_thread::interruption_point()`

Thread IDs

Objects of class `boost::thread::id` can be used to identify threads. Each running thread of execution has a unique ID obtainable from the corresponding `boost::thread` by calling the `get_id()` member function, or by calling `boost::this_thread::get_id()` from within the thread. Objects of class `boost::thread::id` can be copied, and used as keys in associative containers: the full range of comparison operators is provided. Thread IDs can also be written to an output stream using the stream insertion operator, though the output format is unspecified.

Each instance of `boost::thread::id` either refers to some thread, or *Not-a-Thread*. Instances that refer to *Not-a-Thread* compare equal to each other, but not equal to any instances that refer to an actual thread of execution. The comparison operators on `boost::thread::id` yield a total order for every non-equal thread ID.

Class `thread`

```
#include <boost/thread/thread.hpp>

class thread
{
public:
    thread();
    ~thread();

    template <class F>
    explicit thread(F f);

    template <class F,class A1,class A2,...>
    thread(F f,A1 a1,A2 a2,...);

    template <class F>
    thread(detail::thread_move_t<F> f);

    // move support
    thread(detail::thread_move_t<thread> x);
    thread& operator=(detail::thread_move_t<thread> x);
    operator detail::thread_move_t<thread>();
    detail::thread_move_t<thread> move();

    void swap(thread& x);

    class id;
    id get_id() const;

    bool joinable() const;
    void join();
    bool timed_join(const system_time& wait_until);

    template<typename TimeDuration>
    bool timed_join(TimeDuration const& rel_time);

    void detach();

    static unsigned hardware_concurrency();

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    void interrupt();
    bool interruption_requested() const;

    // backwards compatibility
    bool operator==(const thread& other) const;
    bool operator!=(const thread& other) const;
```

```
    static void yield();
    static void sleep(const system_time& xt);
};

void swap(thread& lhs,thread& rhs);
```

Default Constructor

```
thread();
```

Effects: Constructs a `boost::thread` instance that refers to *Not-a-Thread*.

Throws: Nothing

Thread Constructor

```
template<typename Callable>
thread(Callable func);
```

Preconditions: `Callable` must by copyable.

Effects: `func` is copied into storage managed internally by the thread library, and that copy is invoked on a newly-created thread of execution. If this invocation results in an exception being propagated into the internals of the thread library that is not of type `boost::thread_interrupted`, then `std::terminate()` will be called.

Postconditions: `*this` refers to the newly created thread of execution.

Throws: `boost::thread_resource_error` if an error occurs.

Thread Constructor with arguments

```
template <class F,class A1,class A2,...>
thread(F f,A1 a1,A2 a2,...);
```

Preconditions: `F` and each `An` must by copyable or movable.

Effects: As if `thread(boost::bind(f,a1,a2,...))`. Consequently, `f` and each `an` are copied into internal storage for access by the new thread.

Postconditions: `*this` refers to the newly created thread of execution.

Throws: `boost::thread_resource_error` if an error occurs.

Note: Currently up to nine additional arguments `a1` to `a9` can be specified in addition to the function `f`.

Thread Destructor

```
~thread();
```

Effects: If `*this` has an associated thread of execution, calls `detach()`. Destroys `*this`.

Throws: Nothing.

Member function `joinable()`

```
bool joinable() const;
```

Returns: `true` if `*this` refers to a thread of execution, `false` otherwise.

Throws: Nothing

Member function `join()`

```
void join();
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete.

Postconditions: If `*this` refers to a thread of execution on entry, that thread of execution has completed. `*this` no longer refers to any thread of execution.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `join()` is one of the predefined *interruption points*.

Member function `timed_join()`

```
bool timed_join(const system_time& wait_until);

template<typename TimeDuration>
bool timed_join(TimeDuration const& rel_time);
```

Preconditions: `this->get_id()!=boost::this_thread::get_id()`

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete, the time `wait_until` has been reach or the specified duration `rel_time` has elapsed. If `*this` doesn't refer to a thread of execution, returns immediately.

Returns: `true` if `*this` refers to a thread of execution on entry, and that thread of execution has completed before the call times out, `false` otherwise.

Postconditions: If `*this` refers to a thread of execution on entry, and `timed_join` returns `true`, that thread of execution has completed, and `*this` no longer refers to any thread of execution. If this call to `timed_join` returns `false`, `*this` is unchanged.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `timed_join()` is one of the predefined *interruption points*.

Member function `detach()`

```
void detach();
```

Effects: If `*this` refers to a thread of execution, that thread of execution becomes detached, and no longer has an associated `boost::thread` object.

Postconditions: `*this` no longer refers to any thread of execution.

Throws: Nothing

Member function `get_id()`

```
thread::id get_id() const;
```

Returns: If `*this` refers to a thread of execution, an instance of `boost::thread::id` that represents that thread. Otherwise returns a default-constructed `boost::thread::id`.

Throws: Nothing

Member function `interrupt()`

```
void interrupt();
```

Effects: If `*this` refers to a thread of execution, request that the thread will be interrupted the next time it enters one of the predefined *interruption points* with interruption enabled, or if it is currently *blocked* in a call to one of the predefined *interruption points* with interruption enabled .

Throws: Nothing

Static member function `hardware_concurrency()`

```
unsigned hardware_concurrency();
```

Returns: The number of hardware threads available on the current system (e.g. number of CPUs or cores or hyperthreading units), or 0 if this information is not available.

Throws: Nothing

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

`operator==`

```
bool operator==(const thread& other) const;
```

Returns: `get_id()==other.get_id()`

`operator!=`

```
bool operator!=(const thread& other) const;
```

Returns: `get_id()!=other.get_id()`

Static member function `sleep()`

```
void sleep(system_time const& abs_time);
```

Effects: Suspends the current thread until the specified time has been reached.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep()` is one of the predefined *interruption points*.

Static member function `yield()`

```
void yield();
```

Effects: See `boost::this_thread::yield()`.

Member function `swap()`

```
void swap(thread& other);
```

Effects: Exchanges the threads of execution associated with `*this` and `other`, so `*this` is associated with the thread of execution associated with `other` prior to the call, and vice-versa.

Postconditions: `this->get_id()` returns the same value as `other.get_id()` prior to the call. `other.get_id()` returns the same value as `this->get_id()` prior to the call.

Throws: Nothing.

Non-member function `swap()`

```
#include <boost/thread/thread.hpp>

void swap(thread& lhs,thread& rhs);
```

Effects: `lhs.swap(rhs)`.

Class `boost::thread::id`

```
#include <boost/thread/thread.hpp>

class thread::id
{
public:
    id();

    bool operator==(const id& y) const;
    bool operator!=(const id& y) const;
    bool operator<(const id& y) const;
    bool operator>(const id& y) const;
    bool operator<=(const id& y) const;
    bool operator>=(const id& y) const;

    template<class charT, class traits>
    friend std::basic_ostream<charT, traits>&
    operator<<(std::basic_ostream<charT, traits>& os, const id& x);
};
```

Default constructor

```
id();
```

Effects: Constructs a `boost::thread::id` instance that represents *Not-a-Thread*.

Throws: Nothing

`operator==`

```
bool operator==(const id& y) const;
```

Returns: `true` if `*this` and `y` both represent the same thread of execution, or both represent *Not-a-Thread*, `false` otherwise.

Throws: Nothing

`operator!=`

```
bool operator!=(const id& y) const;
```

Returns: `true` if `*this` and `y` represent different threads of execution, or one represents a thread of execution, and the other represent *Not-a-Thread*, `false` otherwise.

Throws: Nothing

`operator<`

```
bool operator<(const id& y) const;
```

Returns: `true` if `*this!=y` is `true` and the implementation-defined total order of `boost::thread::id` values places `*this` before `y`, `false` otherwise.

Throws: Nothing

Note: A `boost::thread::id` instance representing *Not-a-Thread* will always compare less than an instance representing a thread of execution.

`operator>`

```
bool operator>(const id& y) const;
```

Returns: `y<*this`

Throws: Nothing

`operator>=`

```
bool operator<=(const id& y) const;
```

Returns: `!(y<*this)`

Throws: Nothing

`operator>=`

```
bool operator>=(const id& y) const;
```

Returns: `!(*this<y)`

Throws: Nothing

Friend `operator<<`

```
template<class charT, class traits>
friend std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os, const id& x);
```

Effects: Writes a representation of the `boost::thread::id` instance `x` to the stream `os`, such that the representation of two instances of `boost::thread::id` `a` and `b` is the same if `a==b`, and different if `a!=b`.

Returns: `os`

Namespace `this_thread`

Non-member function `get_id()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    thread::id get_id();
}
```

Returns: An instance of `boost::thread::id` that represents that currently executing thread.

Throws: `boost::thread_resource_error` if an error occurs.

Non-member function `interruption_point()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    void interruption_point();
}
```

Effects: Check to see if the current thread has been interrupted.

Throws: `boost::thread_interrupted` if `boost::this_thread::interruption_enabled()` and `boost::this_thread::interruption_requested()` both return `true`.

Non-member function `interruption_requested()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    bool interruption_requested();
}
```

Returns: `true` if interruption has been requested for the current thread, `false` otherwise.

Throws: Nothing.

Non-member function `interruption_enabled()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    bool interruption_enabled();
}
```

Returns: `true` if interruption has been enabled for the current thread, `false` otherwise.

Throws: Nothing.

Non-member function `sleep()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    template<typename TimeDuration>
    void sleep(TimeDuration const& rel_time);
}
```

Effects: Suspends the current thread until the specified time has elapsed.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep()` is one of the predefined *interruption points*.

Non-member function `yield()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    void yield();
}
```

Effects: Gives up the remainder of the current thread's time slice, to allow other threads to run.

Throws: Nothing.

Class `disable_interruption`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    class disable_interruption
    {
    public:
        disable_interruption();
        ~disable_interruption();
    };
}
```

`boost::this_thread::disable_interruption` disables interruption for the current thread on construction, and restores the prior interruption state on destruction. Instances of `disable_interruption` cannot be copied or moved.

Constructor

```
disable_interruption();
```

Effects: Stores the current state of `boost::this_thread::interruption_enabled()` and disables interruption for the current thread.

Postconditions: `boost::this_thread::interruption_enabled()` returns `false` for the current thread.

Throws: Nothing.

Destructor

```
~disable_interruption();
```

Preconditions: Must be called from the same thread from which `*this` was constructed.

Effects: Restores the current state of `boost::this_thread::interruption_enabled()` for the current thread to that prior to the construction of `*this`.

Postconditions: `boost::this_thread::interruption_enabled()` for the current thread returns the value stored in the constructor of `*this`.

Throws: Nothing.

Class `restore_interruption`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    class restore_interruption
    {
    public:
        explicit restore_interruption(disable_interruption& disabler);
        ~restore_interruption();
    };
}
```

On construction of an instance of `boost::this_thread::restore_interruption`, the interruption state for the current thread is restored to the interruption state stored by the constructor of the supplied instance of `boost::this_thread::disable_interruption`. When the instance is destroyed, interruption is again disabled. Instances of `restore_interruption` cannot be copied or moved.

Constructor

```
explicit restore_interruption(disable_interruption& disabler);
```

Preconditions: Must be called from the same thread from which `disabler` was constructed.

Effects: Restores the current state of `boost::this_thread::interruption_enabled()` for the current thread to that prior to the construction of `disabler`.

Postconditions: `boost::this_thread::interruption_enabled()` for the current thread returns the value stored in the constructor of `disabler`.

Throws: Nothing.

Destructor

```
~restore_interruption();
```

Preconditions: Must be called from the same thread from which `*this` was constructed.

Effects: Disables interruption for the current thread.

Postconditions: `boost::this_thread::interruption_enabled()` for the current thread returns `false`.

Throws: Nothing.

Non-member function template `at_thread_exit()`

```
#include <boost/thread/thread.hpp>

template<typename Callable>
void at_thread_exit(Callable func);
```

Effects: A copy of `func` is placed in thread-specific storage. This copy is invoked when the current thread exits (even if the thread has been interrupted).

Postconditions: A copy of `func` has been saved for invocation on thread exit.

Throws: `std::bad_alloc` if memory cannot be allocated for the copy of the function, `boost::thread_resource_error` if any other error occurs within the thread library. Any exception thrown whilst copying `func` into internal storage.

Class `thread_group`

```
#include <boost/thread/thread.hpp>

class thread_group:
    private noncopyable
{
public:
    thread_group();
    ~thread_group();

    template<typename F>
    thread* create_thread(F threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    void join_all();
    void interrupt_all();
    int size() const;
};
```

`thread_group` provides for a collection of threads that are related in some fashion. New threads can be added to the group with `add_thread` and `create_thread` member functions. `thread_group` is not copyable or movable.

Constructor

```
thread_group();
```

Effects: Create a new thread group with no threads.

Destructor

```
~thread_group();
```

Effects: Destroy `*this` and `delete` all `boost::thread` objects in the group.

Member function `create_thread()`

```
template<typename F>
thread* create_thread(F threadfunc);
```

Effects: Create a new `boost::thread` object as-if by `new thread(threadfunc)` and add it to the group.

Postcondition: `this->size()` is increased by one, the new thread is running.

Returns: A pointer to the new `boost::thread` object.

Member function `add_thread()`

```
void add_thread(thread* thrd);
```

Precondition: The expression `delete thrd` is well-formed and will not result in undefined behaviour.

Effects: Take ownership of the `boost::thread` object pointed to by `thrd` and add it to the group.

Postcondition: `this->size()` is increased by one.

Member function `remove_thread()`

```
void remove_thread(thread* thrd);
```

Effects: If `thrd` is a member of the group, remove it without calling `delete`.

Postcondition: If `thrd` was a member of the group, `this->size()` is decreased by one.

Member function `join_all()`

```
void join_all();
```

Effects: Call `join()` on each `boost::thread` object in the group.

Postcondition: Every thread in the group has terminated.

Note: Since `join()` is one of the predefined *interruption points*, `join_all()` is also an interruption point.

Member function `interrupt_all()`

```
void interrupt_all();
```

Effects: Call `interrupt()` on each `boost::thread` object in the group.

Member function `size()`

```
int size();
```

Returns: The number of threads in the group.

Throws: Nothing.

Synchronization

Mutex Concepts

A mutex object facilitates protection against data races and allows thread-safe synchronization of data between threads. A thread obtains ownership of a mutex object by calling one of the lock functions and relinquishes ownership by calling the corresponding unlock function. Mutexes may be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads. **Boost.Thread** supplies recursive and non-recursive mutexes with exclusive ownership semantics, along with a shared ownership (multiple-reader / single-writer) mutex.

Boost.Thread supports four basic concepts for lockable objects: `Lockable`, `TimedLockable`, `SharedLockable` and `UpgradeLockable`. Each mutex type implements one or more of these concepts, as do the various lock types.

`Lockable` Concept

The `Lockable` concept models exclusive ownership. A type that implements the `Lockable` concept shall provide the following member functions:

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

Lock ownership acquired through a call to `lock()` or `try_lock()` must be released through a call to `unlock()`.

`void lock()`

Effects: The current thread blocks until ownership can be obtained for the current thread.

Postcondition: The current thread owns `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`bool try_lock()`

Effects: Attempt to obtain ownership for the current thread without blocking.

Returns: `true` if ownership was obtained for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread owns the `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`void unlock()`

Precondition: The current thread owns `*this`.

Effects: Releases ownership by the current thread.

Postcondition: The current thread no longer owns `*this`.

Throws: Nothing

`TimedLockable` Concept

The `TimedLockable` concept refines the `Lockable` concept to add support for timeouts when trying to acquire the lock.

A type that implements the `TimedLockable` concept shall meet the requirements of the `Lockable` concept. In addition, the following member functions must be provided:

- `bool timed_lock(boost::system_time const& abs_time);`
- `template<typename DurationType> bool timed_lock(DurationType const& rel_time);`

Lock ownership acquired through a call to `timed_lock()` must be released through a call to `unlock()`.

`bool timed_lock(boost::system_time const& abs_time)`

Effects: Attempt to obtain ownership for the current thread. Blocks until ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as `try_lock()`.

Returns: `true` if ownership was obtained for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread owns `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`template<typename DurationType> bool timed_lock(DurationType const& rel_time)`

Effects: As-if `timed_lock(boost::get_system_time()+rel_time)`.

`SharedLockable` Concept

The `SharedLockable` concept is a refinement of the `TimedLockable` concept that allows for *shared ownership* as well as *exclusive ownership*. This is the standard multiple-reader / single-write model: at most one thread can have exclusive ownership, and if any thread does have exclusive ownership, no other threads can have shared or exclusive ownership. Alternatively, many threads may have shared ownership.

For a type to implement the `SharedLockable` concept, as well as meeting the requirements of the `TimedLockable` concept, it must also provide the following member functions:

- `void lock_shared();`
- `bool try_lock_shared();`
- `bool unlock_shared();`
- `bool timed_lock_shared(boost::system_time const& abs_time);`

Lock ownership acquired through a call to `lock_shared()`, `try_lock_shared()` or `timed_lock_shared()` must be released through a call to `unlock_shared()`.

`void lock_shared()`

Effects: The current thread blocks until shared ownership can be obtained for the current thread.

Postcondition: The current thread has shared ownership of `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`bool try_lock_shared()`

Effects: Attempt to obtain shared ownership for the current thread without blocking.

Returns: `true` if shared ownership was obtained for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has shared ownership of `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`bool timed_lock_shared(boost::system_time const& abs_time)`

Effects: Attempt to obtain shared ownership for the current thread. Blocks until shared ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as `try_lock_shared()`.

Returns: `true` if shared ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has shared ownership of `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`void unlock_shared()`

Precondition: The current thread has shared ownership of `*this`.

Effects: Releases shared ownership of `*this` by the current thread.

Postcondition: The current thread no longer has shared ownership of `*this`.

Throws: Nothing

`UpgradeLockable` Concept

The `UpgradeLockable` concept is a refinement of the `SharedLockable` concept that allows for *upgradable ownership* as well as *shared ownership* and *exclusive ownership*. This is an extension to the multiple-reader / single-write model provided by the `SharedLockable` concept: a single thread may have *upgradable ownership* at the same time as others have *shared ownership*. The thread with *upgradable ownership* may at any time attempt to upgrade that ownership to *exclusive ownership*. If no other threads have shared ownership, the upgrade is completed immediately, and the thread now has *exclusive ownership*, which must be relinquished by a call to `unlock()`, just as if it had been acquired by a call to `lock()`.

If a thread with *upgradable ownership* tries to upgrade whilst other threads have *shared ownership*, the attempt will fail and the thread will block until *exclusive ownership* can be acquired.

Ownership can also be *downgraded* as well as *upgraded*: exclusive ownership of an implementation of the `UpgradeLockable` concept can be downgraded to upgradable ownership or shared ownership, and upgradable ownership can be downgraded to plain shared ownership.

For a type to implement the `UpgradeLockable` concept, as well as meeting the requirements of the `SharedLockable` concept, it must also provide the following member functions:

- `void lock_upgrade();`
- `bool unlock_upgrade();`
- `void unlock_upgrade_and_lock();`
- `void unlock_and_lock_upgrade();`
- `void unlock_upgrade_and_lock_shared();`

Lock ownership acquired through a call to `lock_upgrade()` must be released through a call to `unlock_upgrade()`. If the ownership type is changed through a call to one of the `unlock_xxx_and_lock_yyy()` functions, ownership must be released through a call to the unlock function corresponding to the new level of ownership.

`void lock_upgrade()`

Effects: The current thread blocks until upgrade ownership can be obtained for the current thread.

Postcondition: The current thread has upgrade ownership of `*this`.

Throws: `boost::thread_resource_error` if an error occurs.

`void unlock_upgrade()`

Precondition: The current thread has upgrade ownership of `*this`.

Effects: Releases upgrade ownership of `*this` by the current thread.

Postcondition: The current thread no longer has upgrade ownership of `*this`.

Throws: Nothing

`void unlock_upgrade_and_lock()`

Precondition: The current thread has upgrade ownership of `*this`.

Effects: Atomically releases upgrade ownership of `*this` by the current thread and acquires exclusive ownership of `*this`. If any other threads have shared ownership, blocks until exclusive ownership can be acquired.

Postcondition: The current thread has exclusive ownership of `*this`.

Throws: Nothing

`void unlock_upgrade_and_lock_shared()`

Precondition: The current thread has upgrade ownership of `*this`.

Effects: Atomically releases upgrade ownership of `*this` by the current thread and acquires shared ownership of `*this` without blocking.

Postcondition: The current thread has shared ownership of `*this`.

Throws: Nothing

`void unlock_and_lock_upgrade()`

Precondition: The current thread has exclusive ownership of `*this`.

Effects: Atomically releases exclusive ownership of `*this` by the current thread and acquires upgrade ownership of `*this` without blocking.

Postcondition: The current thread has upgrade ownership of `*this`.

Throws: Nothing

Lock Types

Class template `lock_guard`

```
#include <boost/thread/locks.hpp>

template<typename Lockable>
class lock_guard
{
public:
    explicit lock_guard(Lockable& m_);
    lock_guard(Lockable& m_,boost::adopt_lock_t);

    ~lock_guard();
};
```

`boost::lock_guard` is very simple: on construction it acquires ownership of the implementation of the `Lockable` concept supplied as the constructor parameter. On destruction, the ownership is released. This provides simple RAII-style locking of a `Lockable` object, to facilitate exception-safe locking and unlocking. In addition, the `lock_guard(Lockable & m,boost::adopt_lock_t)` constructor allows the `boost::lock_guard` object to take ownership of a lock already held by the current thread.

`lock_guard(Lockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock()`.

Throws: Any exception thrown by the call to `m.lock()`.

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

Precondition: The current thread owns a lock on `m` equivalent to one obtained by a call to `m.lock()`.

Effects: Stores a reference to `m`. Takes ownership of the lock state of `m`.

Throws: Nothing.

`~lock_guard()`

Effects: Invokes `m.unlock()` on the `Lockable` object passed to the constructor.

Throws: Nothing.

Class template `unique_lock`

```
#include <boost/thread/locks.hpp>

template<typename Lockable>
class unique_lock
{
public:
    unique_lock();
    explicit unique_lock(Lockable& m_);
    unique_lock(Lockable& m_,adopt_lock_t);
    unique_lock(Lockable& m_,defer_lock_t);
    unique_lock(Lockable& m_,try_to_lock_t);
    unique_lock(Lockable& m_,system_time const& target_time);

    ~unique_lock();

    unique_lock(detail::thread_move_t<unique_lock<Lockable> > other);
    unique_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);

    operator detail::thread_move_t<unique_lock<Lockable> >();
    detail::thread_move_t<unique_lock<Lockable> > move();
    unique_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);
    unique_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);

    void swap(unique_lock& other);
    void swap(detail::thread_move_t<unique_lock<Lockable> > other);

    void lock();
    bool try_lock();

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const& relative_time);
    bool timed_lock(::boost::system_time const& absolute_time);

    void unlock();

    bool owns_lock() const;
    operator unspecified-bool-type() const;
    bool operator!() const;

    Lockable* mutex() const;
    Lockable* release();
};
```

`boost::unique_lock` is more complex than `boost::lock_guard`: not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

Specializations of boost::unique_lock model the TimedLockable concept if the supplied Lockable type itself models TimedLockable concept (e.g. boost::unique_lock<boost::timed_mutex>), or the Lockable concept otherwise (e.g. boost::unique_lock<boost::mutex>).

An instance of boost::unique_lock is said to *own* the lock state of a Lockable m if mutex() returns a pointer to m and owns_lock() returns true. If an object that *owns* the lock state of a Lockable object is destroyed, then the destructor will invoke mutex()->unlock().

The member functions of boost::unique_lock are not thread-safe. In particular, boost::unique_lock is intended to model the ownership of a Lockable object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

unique_lock()

Effects: Creates a lock object with no associated mutex.

Postcondition: owns_lock() returns false. mutex() returns NULL.

Throws: Nothing.

unique_lock(Lockable & m)

Effects: Stores a reference to m. Invokes m.lock().

Postcondition: owns_lock() returns true. mutex() returns &m.

Throws: Any exception thrown by the call to m.lock().

unique_lock(Lockable & m,boost::adopt_lock_t)

Precondition: The current thread owns an exclusive lock on m.

Effects: Stores a reference to m. Takes ownership of the lock state of m.

Postcondition: owns_lock() returns true. mutex() returns &m.

Throws: Nothing.

unique_lock(Lockable & m,boost::defer_lock_t)

Effects: Stores a reference to m.

Postcondition: owns_lock() returns false. mutex() returns &m.

Throws: Nothing.

unique_lock(Lockable & m,boost::try_to_lock_t)

Effects: Stores a reference to m. Invokes m.try_lock(), and takes ownership of the lock state if the call returns true.

Postcondition: mutex() returns &m. If the call to try_lock() returned true, then owns_lock() returns true, otherwise owns_lock() returns false.

Throws: Nothing.

unique_lock(Lockable & m,boost::system_time const& abs_time)

Effects: Stores a reference to m. Invokes m.timed_lock(abs_time), and takes ownership of the lock state if the call returns true.

Postcondition: `mutex()` returns `&m`. If the call to `timed_lock()` returned `true`, then `owns_lock()` returns `true`, otherwise `owns_lock()` returns `false`.

Throws: Any exceptions thrown by the call to `m.timed_lock(abs_time)`.

`~unique_lock()`

Effects: Invokes `mutex()`-> `unlock()` if `owns_lock()` returns `true`.

Throws: Nothing.

`bool owns_lock() const`

Returns: `true` if the `*this` owns the lock on the `Lockable` object associated with `*this`.

Throws: Nothing.

`Lockable* mutex() const`

Returns: A pointer to the `Lockable` object associated with `*this`, or `NULL` if there is no such object.

Throws: Nothing.

`operator unspecified-bool-type() const`

Returns: If `owns_lock()` would return `true`, a value that evaluates to `true` in boolean contexts, otherwise a value that evaluates to `false` in boolean contexts.

Throws: Nothing.

`bool operator!() const`

Returns: `!` `owns_lock()`.

Throws: Nothing.

`Lockable* release()`

Effects: The association between `*this` and the `Lockable` object is removed, without affecting the lock state of the `Lockable` object. If `owns_lock()` would have returned `true`, it is the responsibility of the calling code to ensure that the `Lockable` is correctly unlocked.

Returns: A pointer to the `Lockable` object associated with `*this` at the point of the call, or `NULL` if there is no such object.

Throws: Nothing.

Postcondition: `*this` is no longer associated with any `Lockable` object. `mutex()` returns `NULL` and `owns_lock()` returns `false`.

Class template `shared_lock`

```
#include <boost/thread/locks.hpp>

template<typename Lockable>
class shared_lock
{
public:
    shared_lock();
    explicit shared_lock(Lockable& m_);
    shared_lock(Lockable& m_,adopt_lock_t);
    shared_lock(Lockable& m_,defer_lock_t);
    shared_lock(Lockable& m_,try_to_lock_t);
    shared_lock(Lockable& m_,system_time const& target_time);
    shared_lock(detail::thread_move_t<shared_lock<Lockable> > other);
    shared_lock(detail::thread_move_t<unique_lock<Lockable> > other);
    shared_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);

    ~shared_lock();

    operator detail::thread_move_t<shared_lock<Lockable> >();
    detail::thread_move_t<shared_lock<Lockable> > move();

    shared_lock& operator=(detail::thread_move_t<shared_lock<Lockable> > other);
    shared_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);
    shared_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);
    void swap(shared_lock& other);

    void lock();
    bool try_lock();
    bool timed_lock(boost::system_time const& target_time);
    void unlock();

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

Like `boost::unique_lock`, `boost::shared_lock` models the `Lockable` concept, but rather than acquiring unique ownership of the supplied `Lockable` object, locking an instance of `boost::shared_lock` acquires shared ownership.

Like `boost::unique_lock`, not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

An instance of `boost::shared_lock` is said to *own* the lock state of a `Lockable` `m` if `mutex()` returns a pointer to `m` and `owns_lock()` returns `true`. If an object that *owns* the lock state of a `Lockable` object is destroyed, then the destructor will invoke `mutex()->unlock_shared()`.

The member functions of `boost::shared_lock` are not thread-safe. In particular, `boost::shared_lock` is intended to model the shared ownership of a `Lockable` object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

`shared_lock()`

Effects: Creates a lock object with no associated mutex.

Postcondition: `owns_lock()` returns `false`. `mutex()` returns `NULL`.

Throws: Nothing.

shared_lock(Lockable & m)

Effects: Stores a reference to m. Invokes m.lock_shared().

Postcondition: owns_lock() returns true. mutex() returns &m.

Throws: Any exception thrown by the call to m.lock_shared().

shared_lock(Lockable & m,boost::adopt_lock_t)

Precondition: The current thread owns an exclusive lock on m.

Effects: Stores a reference to m. Takes ownership of the lock state of m.

Postcondition: owns_lock() returns true. mutex() returns &m.

Throws: Nothing.

shared_lock(Lockable & m,boost::defer_lock_t)

Effects: Stores a reference to m.

Postcondition: owns_lock() returns false. mutex() returns &m.

Throws: Nothing.

shared_lock(Lockable & m,boost::try_to_lock_t)

Effects: Stores a reference to m. Invokes m.try_lock_shared(), and takes ownership of the lock state if the call returns true.

Postcondition: mutex() returns &m. If the call to try_lock_shared() returned true, then owns_lock() returns true, otherwise owns_lock() returns false.

Throws: Nothing.

shared_lock(Lockable & m,boost::system_time const& abs_time)

Effects: Stores a reference to m. Invokes m.timed_lock(abs_time), and takes ownership of the lock state if the call returns true.

Postcondition: mutex() returns &m. If the call to timed_lock_shared() returned true, then owns_lock() returns true, otherwise owns_lock() returns false.

Throws: Any exceptions thrown by the call to m.timed_lock(abs_time).

~shared_lock()

Effects: Invokes mutex()->unlock_shared() if owns_lock() returns true.

Throws: Nothing.

bool owns_lock() const

Returns: true if the *this owns the lock on the Lockable object associated with *this.

Throws: Nothing.

`Lockable* mutex() const`

Returns: A pointer to the `Lockable` object associated with `*this`, or `NULL` if there is no such object.

Throws: Nothing.

`operator unspecified-bool-type() const`

Returns: If `owns_lock()` would return `true`, a value that evaluates to `true` in boolean contexts, otherwise a value that evaluates to `false` in boolean contexts.

Throws: Nothing.

`bool operator!() const`

Returns: `!` `owns_lock()`.

Throws: Nothing.

`Lockable* release()`

Effects: The association between `*this` and the `Lockable` object is removed, without affecting the lock state of the `Lockable` object. If `owns_lock()` would have returned `true`, it is the responsibility of the calling code to ensure that the `Lockable` is correctly unlocked.

Returns: A pointer to the `Lockable` object associated with `*this` at the point of the call, or `NULL` if there is no such object.

Throws: Nothing.

Postcondition: `*this` is no longer associated with any `Lockable` object. `mutex()` returns `NULL` and `owns_lock()` returns `false`.

Class template `upgrade_lock`

```
#include <boost/thread/locks.hpp>

template<typename Lockable>
class upgrade_lock
{
public:
    explicit upgrade_lock(Lockable& m_);

    upgrade_lock(detail::thread_move_t<upgrade_lock<Lockable> > other);
    upgrade_lock(detail::thread_move_t<unique_lock<Lockable> > other);

    ~upgrade_lock();

    operator detail::thread_move_t<upgrade_lock<Lockable> >();
    detail::thread_move_t<upgrade_lock<Lockable> > move();

    upgrade_lock& operator=(detail::thread_move_t<upgrade_lock<Lockable> > other);
    upgrade_lock& operator=(detail::thread_move_t<unique_lock<Lockable> > other);

    void swap(upgrade_lock& other);

    void lock();
    void unlock();

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

Like `boost::unique_lock`, `boost::upgrade_lock` models the `Lockable` concept, but rather than acquiring unique ownership of the supplied `Lockable` object, locking an instance of `boost::upgrade_lock` acquires upgrade ownership.

Like `boost::unique_lock`, not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

An instance of `boost::upgrade_lock` is said to *own* the lock state of a `Lockable` `m` if `mutex()` returns a pointer to `m` and `owns_lock()` returns `true`. If an object that *owns* the lock state of a `Lockable` object is destroyed, then the destructor will invoke `mutex()->unlock_upgrade()`.

The member functions of `boost::upgrade_lock` are not thread-safe. In particular, `boost::upgrade_lock` is intended to model the upgrade ownership of a `Lockable` object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

Class template `upgrade_to_unique_lock`

```
#include <boost/thread/locks.hpp>

template <class Lockable>
class upgrade_to_unique_lock
{
public:
    explicit upgrade_to_unique_lock(upgrade_lock<Lockable>& m_);

    ~upgrade_to_unique_lock();

    upgrade_to_unique_lock(detail::thread_move_t<upgrade_to_unique_lock<Lockable> > other);
   upgrade_to_unique_lock& operator=(detail::thread_move_t<upgrade_to_unique_lock<Lockable> > oth↵
er);
    void swap(upgrade_to_unique_lock& other);

    operator unspecified-bool-type() const;
    bool operator!() const;
    bool owns_lock() const;
};
```

`boost::upgrade_to_unique_lock` allows for a temporary upgrade of an `boost::upgrade_lock` to exclusive ownership. When constructed with a reference to an instance of `boost::upgrade_lock`, if that instance has upgrade ownership on some `Lockable` object, that ownership is upgraded to exclusive ownership. When the `boost::upgrade_to_unique_lock` instance is destroyed, the ownership of the `Lockable` is downgraded back to *upgrade ownership*.

Mutex-specific class `scoped_try_lock`

```
class MutexType::scoped_try_lock
{
private:
    MutexType::scoped_try_lock(MutexType::scoped_try_lock<MutexType>& other);
    MutexType::scoped_try_lock& operator=(MutexType::scoped_try_lock<MutexType>& other);
public:
    MutexType::scoped_try_lock();
    explicit MutexType::scoped_try_lock(MutexType& m);
    MutexType::scoped_try_lock(MutexType& m_,adopt_lock_t);
    MutexType::scoped_try_lock(MutexType& m_,defer_lock_t);
    MutexType::scoped_try_lock(MutexType& m_,try_to_lock_t);

    MutexType::scoped_try_lock(MutexType::scoped_try_lock<MutexType>&& other);
    MutexType::scoped_try_lock& operator=(MutexType::scoped_try_lock<MutexType>&& other);

    void swap(MutexType::scoped_try_lock&& other);

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

    MutexType* mutex() const;
    MutexType* release();
    bool operator!() const;

    typedef unspecified-bool-type bool_type;
    operator bool_type() const;
};
```

The member typedef `scoped_try_lock` is provided for each distinct `MutexType` as a typedef to a class with the preceding definition. The semantics of each constructor and member function are identical to those of `boost::unique_lock<MutexType>` for the same `MutexType`, except that the constructor that takes a single reference to a mutex will call `m.try_lock()` rather than `m.lock()`.

Lock functions

Non-member function `lock(Lockable1,Lockable2,...)`

```
template<typename Lockable1,typename Lockable2>
void lock(Lockable1& l1,Lockable2& l2);

template<typename Lockable1,typename Lockable2,typename Lockable3>
void lock(Lockable1& l1,Lockable2& l2,Lockable3& l3);

template<typename Lockable1,typename Lockable2,typename Lockable3,typename Lockable4>
void lock(Lockable1& l1,Lockable2& l2,Lockable3& l3,Lockable4& l4);

template<typename Lockable1,typename Lockable2,typename Lockable3,typename Lockable4,typename Lock↵
able5>
void lock(Lockable1& l1,Lockable2& l2,Lockable3& l3,Lockable4& l4,Lockable5& l5);
```

Effects: Locks the `Lockable` objects supplied as arguments in an unspecified and indeterminate order in a way that avoids deadlock. It is safe to call this function concurrently from multiple threads with the same mutexes (or other lockable objects) in different orders without risk of deadlock. If any of the `lock()` or `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.

Throws: Any exceptions thrown by calling `lock()` or `try_lock()` on the supplied `Lockable` objects.

Postcondition: All the supplied `Lockable` objects are locked by the calling thread.

Non-member function `lock(begin,end)`

```
template<typename ForwardIterator>
void lock(ForwardIterator begin,ForwardIterator end);
```

Preconditions: The `value_type` of `ForwardIterator` must implement the `Lockable` concept

Effects: Locks all the `Lockable` objects in the supplied range in an unspecified and indeterminate order in a way that avoids deadlock. It is safe to call this function concurrently from multiple threads with the same mutexes (or other lockable objects) in different orders without risk of deadlock. If any of the `lock()` or `try_lock()` operations on the `Lockable` objects in the supplied range throws an exception any locks acquired by the function will be released before the function exits.

Throws: Any exceptions thrown by calling `lock()` or `try_lock()` on the supplied `Lockable` objects.

Postcondition: All the `Lockable` objects in the supplied range are locked by the calling thread.

Non-member function `try_lock(Lockable1,Lockable2,...)`

```
template<typename Lockable1,typename Lockable2>
int try_lock(Lockable1& l1,Lockable2& l2);

template<typename Lockable1,typename Lockable2,typename Lockable3>
int try_lock(Lockable1& l1,Lockable2& l2,Lockable3& l3);

template<typename Lockable1,typename Lockable2,typename Lockable3,typename Lockable4>
int try_lock(Lockable1& l1,Lockable2& l2,Lockable3& l3,Lockable4& l4);

template<typename Lockable1,typename Lockable2,typename Lockable3,typename Lockable4,typename Lock↵
able5>
int try_lock(Lockable1& l1,Lockable2& l2,Lockable3& l3,Lockable4& l4,Lockable5& l5);
```

Effects: Calls `try_lock()` on each of the `Lockable` objects supplied as arguments. If any of the calls to `try_lock()` returns `false` then all locks acquired are released and the zero-based index of the failed lock is returned.

If any of the `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.

Returns: `-1` if all the supplied `Lockable` objects are now locked by the calling thread, the zero-based index of the object which could not be locked otherwise.

Throws: Any exceptions thrown by calling `try_lock()` on the supplied `Lockable` objects.

Postcondition: If the function returns `-1`, all the supplied `Lockable` objects are locked by the calling thread. Otherwise any locks acquired by this function will have been released.

Non-member function `try_lock(begin,end)`

```
template<typename ForwardIterator>
ForwardIterator try_lock(ForwardIterator begin,ForwardIterator end);
```

Preconditions: The `value_type` of `ForwardIterator` must implement the `Lockable` concept

Effects: Calls `try_lock()` on each of the `Lockable` objects in the supplied range. If any of the calls to `try_lock()` returns `false` then all locks acquired are released and an iterator referencing the failed lock is returned.

If any of the `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.

Returns: `end` if all the supplied `Lockable` objects are now locked by the calling thread, an iterator referencing the object which could not be locked otherwise.

Throws: Any exceptions thrown by calling `try_lock()` on the supplied `Lockable` objects.

Postcondition: If the function returns `end` then all the `Lockable` objects in the supplied range are locked by the calling thread, otherwise all locks acquired by the function have been released.

Mutex Types

Class `mutex`

```
#include <boost/thread/mutex.hpp>

class mutex:
    boost::noncopyable
{
public:
    mutex();
    ~mutex();

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

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

`boost::mutex` implements the `Lockable` concept to provide an exclusive-ownership mutex. At most one thread can own the lock on a given instance of `boost::mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()` and `unlock()` shall be permitted.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Typedef `try_mutex`

```
#include <boost/thread/mutex.hpp>

typedef mutex try_mutex;
```

`boost::try_mutex` is a `typedef` to `boost::mutex`, provided for backwards compatibility with previous releases of boost.

Class `timed_mutex`

```
#include <boost/thread/mutex.hpp>

class timed_mutex:
    boost::noncopyable
{
public:
    timed_mutex();
    ~timed_mutex();

    void lock();
    void unlock();
    bool try_lock();
    bool timed_lock(system_time const & abs_time);

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<timed_mutex> scoped_timed_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_timed_lock scoped_lock;
};
```

`boost::timed_mutex` implements the `TimedLockable` concept to provide an exclusive-ownership mutex. At most one thread can own the lock on a given instance of `boost::timed_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()`, `timed_lock()`, `timed_lock()` and `unlock()` shall be permitted.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Class `recursive_mutex`

```
#include <boost/thread/recursive_mutex.hpp>

class recursive_mutex:
    boost::noncopyable
{
public:
    recursive_mutex();
    ~recursive_mutex();

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

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<recursive_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

`boost::recursive_mutex` implements the `Lockable` concept to provide an exclusive-ownership recursive mutex. At most one thread can own the lock on a given instance of `boost::recursive_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()` and `unlock()` shall be permitted. A thread that already has exclusive ownership of a given `boost::recursive_mutex` instance can call `lock()` or `try_lock()` to acquire an additional level of ownership of the mutex. `unlock()` must be called once for each level of ownership acquired by a single thread before ownership can be acquired by another thread.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Typedef `recursive_try_mutex`

```
#include <boost/thread/recursive_mutex.hpp>

typedef recursive_mutex recursive_try_mutex;
```

`boost::recursive_try_mutex` is a `typedef` to `boost::recursive_mutex`, provided for backwards compatibility with previous releases of boost.

Class recursive_timed_mutex

```
#include <boost/thread/recursive_mutex.hpp>

class recursive_timed_mutex:
    boost::noncopyable
{
public:
    recursive_timed_mutex();
    ~recursive_timed_mutex();

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

    bool timed_lock(system_time const & abs_time);

    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<recursive_timed_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_lock scoped_timed_lock;
};
```

`boost::recursive_timed_mutex` implements the `TimedLockable` concept to provide an exclusive-ownership recursive mutex. At most one thread can own the lock on a given instance of `boost::recursive_timed_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()`, `timed_lock()`, `timed_lock()` and `unlock()` shall be permitted. A thread that already has exclusive ownership of a given `boost::recursive_timed_mutex` instance can call `lock()`, `timed_lock()`, `timed_lock()` or `try_lock()` to acquire an additional level of ownership of the mutex. `unlock()` must be called once for each level of ownership acquired by a single thread before ownership can be acquired by another thread.

Member function native_handle()

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Class `shared_mutex`

```
#include <boost/thread/shared_mutex.hpp>

class shared_mutex
{
public:
    shared_mutex();
    ~shared_mutex();

    void lock_shared();
    bool try_lock_shared();
    bool timed_lock_shared(system_time const& timeout);
    void unlock_shared();

    void lock();
    bool try_lock();
    bool timed_lock(system_time const& timeout);
    void unlock();

    void 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();
};
```

The class `boost::shared_mutex` provides an implementation of a multiple-reader / single-writer mutex. It implements the `UpgradeLockable` concept.

Multiple concurrent calls to `lock()`, `try_lock()`, `timed_lock()`, `lock_shared()`, `try_lock_shared()` and `timed_lock_shared()` shall be permitted.

Condition Variables

Synopsis

The classes `condition_variable` and `condition_variable_any` provide a mechanism for one thread to wait for notification from another thread that a particular condition has become true. The general usage pattern is that one thread locks a mutex and then calls `wait` on an instance of `condition_variable` or `condition_variable_any`. When the thread is woken from the wait, then it checks to see if the appropriate condition is now true, and continues if so. If the condition is not true, then the thread then calls `wait` again to resume waiting. In the simplest case, this condition is just a boolean variable:

```
boost::condition_variable cond;
boost::mutex mut;
bool data_ready;

void process_data();

void wait_for_data_to_process()
{
    boost::unique_lock<boost::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}
```

Notice that the `lock` is passed to `wait`: `wait` will atomically add the thread to the set of threads waiting on the condition variable, and unlock the mutex. When the thread is woken, the mutex will be locked again before the call to `wait` returns. This allows other threads to acquire the mutex in order to update the shared data, and ensures that the data associated with the condition is correctly synchronized.

In the mean time, another thread sets the condition to `true`, and then calls either `notify_one` or `notify_all` on the condition variable to wake one waiting thread or all the waiting threads respectively.

```
void retrieve_data();
void prepare_data();

void prepare_data_for_processing()
{
    retrieve_data();
    prepare_data();
    {
        boost::lock_guard<boost::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}
```

Note that the same mutex is locked before the shared data is updated, but that the mutex does not have to be locked across the call to `notify_one`.

This example uses an object of type `condition_variable`, but would work just as well with an object of type `condition_variable_any`: `condition_variable_any` is more general, and will work with any kind of lock or mutex, whereas `condition_variable` requires that the lock passed to `wait` is an instance of `boost::unique_lock<boost::mutex>`. This enables `condition_variable` to make optimizations in some cases, based on the knowledge of the mutex type; `condition_variable_any` typically has a more complex implementation than `condition_variable`.

Class `condition_variable`

```
#include <boost/thread/condition_variable.hpp>

namespace boost
{
    class condition_variable
    {
    public:
        condition_variable();
        ~condition_variable();

        void notify_one();
        void notify_all();

        void wait(boost::unique_lock<boost::mutex>& lock);

        template<typename predicate_type>
        void wait(boost::unique_lock<boost::mutex>& lock,predicate_type predicate);

       bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::system_time const& abs_time);

        template<typename duration_type>
        bool timed_wait(boost::unique_lock<boost::mutex>& lock,duration_type const& rel_time);

        template<typename predicate_type>
       bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::system_time const& abs_time,pre↵
dicate_type predicate);

        template<typename duration_type,typename predicate_type>
        bool timed_wait(boost::unique_lock<boost::mutex>& lock,duration_type const& rel_time,pre↵
dicate_type predicate);

    // backwards compatibility

        bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::xtime const& abs_time);

        template<typename predicate_type>
       bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::xtime const& abs_time,predic↵
ate_type predicate);
    };
}
```

`condition_variable()`

Effects: Constructs an object of class `condition_variable`.

Throws: `boost::thread_resource_error` if an error occurs.

`~condition_variable()`

Precondition: All threads waiting on `*this` have been notified by a call to `notify_one` or `notify_all` (though the respective calls to `wait` or `timed_wait` need not have returned).

Effects: Destroys the object.

Throws: Nothing.

`void notify_one()`

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks one of those threads.

Throws: Nothing.

`void notify_all()`

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks all of those threads.

Throws: Nothing.

`void wait(boost::unique_lock<boost::mutex>& lock)`

Precondition: `lock` is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the `lock` objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

`template<typename predicate_type> void wait(boost::unique_lock<boost::mutex>& lock, predicate_type pred)`

Effects: As-if

```
while(!pred())
{
    wait(lock);
}
```

`bool timed_wait(boost::unique_lock<boost::mutex>& lock,boost::system_time const& abs_time)`

Precondition: `lock` is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the `lock` objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, when the time as reported by `boost::get_system_time()` would be equal to or later than the specified `abs_time`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time specified by `abs_time` was reached, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

template<typename duration_type> bool timed_wait(boost::unique_lock<boost::mutex>& lock,duration_type const& rel_time)

Precondition: `lock` is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the `lock` objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, after the period of time indicated by the `rel_time` argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time period specified by `rel_time` has elapsed, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Note

The duration overload of timed_wait is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

template<typename predicate_type> bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::system_time const& abs_time, predicate_type pred)

Effects: As-if

```
while(!pred())
{
    if(!timed_wait(lock,abs_time))
    {
        return pred();
    }
}
return true;
```

Class condition_variable_any

```
#include <boost/thread/condition_variable.hpp>

namespace boost
{
    class condition_variable_any
    {
    public:
        condition_variable_any();
        ~condition_variable_any();

        void notify_one();
        void notify_all();

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

        template<typename lock_type,typename predicate_type>
        void wait(lock_type& lock,predicate_type predicate);

        template<typename lock_type>
        bool timed_wait(lock_type& lock,boost::system_time const& abs_time);

        template<typename lock_type,typename duration_type>
        bool timed_wait(lock_type& lock,duration_type const& rel_time);

        template<typename lock_type,typename predicate_type>
       bool timed_wait(lock_type& lock,boost::system_time const& abs_time,predicate_type predicate);

        template<typename lock_type,typename duration_type,typename predicate_type>
        bool timed_wait(lock_type& lock,duration_type const& rel_time,predicate_type predicate);

    // backwards compatibility

        template<typename lock_type>
        bool timed_wait(lock_type>& lock,boost::xtime const& abs_time);

        template<typename lock_type,typename predicate_type>
        bool timed_wait(lock_type& lock,boost::xtime const& abs_time,predicate_type predicate);
    };
}
```

condition_variable_any()

Effects: Constructs an object of class `condition_variable_any`.

Throws: `boost::thread_resource_error` if an error occurs.

~condition_variable_any()

Precondition: All threads waiting on `*this` have been notified by a call to `notify_one` or `notify_all` (though the respective calls to `wait` or `timed_wait` need not have returned).

Effects: Destroys the object.

Throws: Nothing.

void notify_one()

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks one of those threads.

Throws: Nothing.

`void notify_all()`

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks all of those threads.

Throws: Nothing.

`template<typename lock_type> void wait(lock_type& lock)`

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

`template<typename lock_type,typename predicate_type> void wait(lock_type& lock, predicate_type pred)`

Effects: As-if

```
while(!pred())
{
    wait(lock);
}
```

`template<typename lock_type> bool timed_wait(lock_type& lock,boost::system_time const& abs_time)`

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, when the time as reported by `boost::get_system_time()` would be equal to or later than the specified `abs_time`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time specified by `abs_time` was reached, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

`template<typename lock_type,typename duration_type> bool timed_wait(lock_type& lock,duration_type const& rel_time)`

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, after the period of time indicated by the `rel_time` argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time period specified by `rel_time` has elapsed, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

> **Note**
>
> The duration overload of timed_wait is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

`template<typename lock_type,typename predicate_type> bool timed_wait(lock_type& lock, boost::system_time const& abs_time, predicate_type pred)`

Effects: As-if

```
while(!pred())
{
    if(!timed_wait(lock,abs_time))
    {
        return pred();
    }
}
return true;
```

Typedef `condition`

```
#include <boost/thread/condition.hpp>

typedef condition_variable_any condition;
```

The typedef `condition` is provided for backwards compatibility with previous boost releases.

One-time Initialization

`boost::call_once` provides a mechanism for ensuring that an initialization routine is run exactly once without data races or deadlocks.

Typedef `once_flag`

```
#include <boost/thread/once.hpp>

typedef platform-specific-type once_flag;
#define BOOST_ONCE_INIT platform-specific-initializer
```

Objects of type `boost::once_flag` shall be initialized with `BOOST_ONCE_INIT`:

```
boost::once_flag f=BOOST_ONCE_INIT;
```

Non-member function `call_once`

```
#include <boost/thread/once.hpp>

template<typename Callable>
void call_once(once_flag& flag,Callable func);
```

Requires: `Callable` is `CopyConstructible`. Copying `func` shall have no side effects, and the effect of calling the copy shall be equivalent to calling the original.

Effects: Calls to `call_once` on the same `once_flag` object are serialized. If there has been no prior effective `call_once` on the same `once_flag` object, the argument `func` (or a copy thereof) is called as-if by invoking `func()`, and the invocation of `call_once` is effective if and only if `func()` returns without exception. If an exception is thrown, the exception is propagated to the caller. If there has been a prior effective `call_once` on the same `once_flag` object, the `call_once` returns without invoking `func`.

Synchronization: The completion of an effective `call_once` invocation on a `once_flag` object, synchronizes with all subsequent `call_once` invocations on the same `once_flag` object.

Throws: `thread_resource_error` when the effects cannot be achieved. or any exception propagated from `func`.

```
void call_once(void (*func)(),once_flag& flag);
```

This second overload is provided for backwards compatibility. The effects of `call_once(func,flag)` shall be the same as those of `call_once(flag,func)`.

Barriers

A barrier is a simple concept. Also known as a *rendezvous*, it is a synchronization point between multiple threads. The barrier is configured for a particular number of threads (`n`), and as threads reach the barrier they must wait until all `n` threads have arrived. Once the `n`-th thread has reached the barrier, all the waiting threads can proceed, and the barrier is reset.

Class `barrier`

```
#include <boost/thread/barrier.hpp>

class barrier
{
public:
    barrier(unsigned int count);
    ~barrier();

    bool wait();
};
```

Instances of `boost::barrier` are not copyable or movable.

Constructor

```
barrier(unsigned int count);
```

Effects: Construct a barrier for `count` threads.

Throws: `boost::thread_resource_error` if an error occurs.

Destructor

```
~barrier();
```

Precondition: No threads are waiting on `*this`.

Effects: Destroys `*this`.

Throws: Nothing.

Member function `wait`

```
bool wait();
```

Effects: Block until `count` threads have called `wait` on `*this`. When the `count`-th thread calls `wait`, all waiting threads are unblocked, and the barrier is reset.

Returns: `true` for exactly one thread from each batch of waiting threads, `false` otherwise.

Throws: `boost::thread_resource_error` if an error occurs.

Thread Local Storage

Synopsis

Thread local storage allows multi-threaded applications to have a separate instance of a given data item for each thread. Where a single-threaded application would use static or global data, this could lead to contention, deadlock or data corruption in a multi-threaded application. One example is the C `errno` variable, used for storing the error code related to functions from the Standard C library. It is common practice (and required by POSIX) for compilers that support multi-threaded applications to provide a separate instance of `errno` for each thread, in order to avoid different threads competing to read or update the value.

Though compilers often provide this facility in the form of extensions to the declaration syntax (such as `__declspec(thread)` or `__thread` annotations on `static` or namespace-scope variable declarations), such support is non-portable, and is often limited in some way, such as only supporting POD types.

Portable thread-local storage with `boost::thread_specific_ptr`

`boost::thread_specific_ptr` provides a portable mechanism for thread-local storage that works on all compilers supported by **Boost.Thread**. Each instance of `boost::thread_specific_ptr` represents a pointer to an object (such as `errno`) where each thread must have a distinct value. The value for the current thread can be obtained using the `get()` member function, or by using the `*` and `->` pointer deference operators. Initially the pointer has a value of `NULL` in each thread, but the value for the current thread can be set using the `reset()` member function.

If the value of the pointer for the current thread is changed using `reset()`, then the previous value is destroyed by calling the cleanup routine. Alternatively, the stored value can be reset to `NULL` and the prior value returned by calling the `release()` member function, allowing the application to take back responsibility for destroying the object.

Cleanup at thread exit

When a thread exits, the objects associated with each `boost::thread_specific_ptr` instance are destroyed. By default, the object pointed to by a pointer `p` is destroyed by invoking `delete p`, but this can be overridden for a specific instance of `boost::thread_specific_ptr` by providing a cleanup routine to the constructor. In this case, the object is destroyed by invoking `func(p)` where `func` is the cleanup routine supplied to the constructor. The cleanup functions are called in an unspecified order. If a cleanup routine sets the value of associated with an instance of `boost::thread_specific_ptr` that has already been cleaned up, that value is added to the cleanup list. Cleanup finishes when there are no outstanding instances of `boost::thread_specific_ptr` with values.

Class `thread_specific_ptr`

```
#include <boost/thread/tss.hpp>

template <typename T>
class thread_specific_ptr
{
public:
    thread_specific_ptr();
    explicit thread_specific_ptr(void (*cleanup_function)(T*));
    ~thread_specific_ptr();

    T* get() const;
    T* operator->() const;
    T& operator*() const;

    T* release();
    void reset(T* new_value=0);
};
```

`thread_specific_ptr();`

Requires: `delete this->get()` is well-formed.

Effects: Construct a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. The default `delete`-based cleanup function will be used to destroy any thread-local objects when `reset()` is called, or the thread exits.

Throws: `boost::thread_resource_error` if an error occurs.

`explicit thread_specific_ptr(void (*cleanup_function)(T*));`

Requires: `cleanup_function(this->get())` does not throw any exceptions.

Effects: Construct a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. The supplied `cleanup_function` will be used to destroy any thread-local objects when `reset()` is called, or the thread exits.

Throws: `boost::thread_resource_error` if an error occurs.

`~thread_specific_ptr();`

Effects: Calls `this->reset()` to clean up the associated value for the current thread, and destroys `*this`.

Throws: Nothing.

> **Note**
>
> Care needs to be taken to ensure that any threads still running after an instance of `boost::thread_specific_ptr` has been destroyed do not call any member functions on that instance.

`T* get() const;`

Returns: The pointer associated with the current thread.

Throws: Nothing.

> **Note**
>
> The initial value associated with an instance of `boost::thread_specific_ptr` is `NULL` for each thread.

`T* operator->() const;`

Returns: `this->get()`

Throws: Nothing.

`T& operator*() const;`

Requires: `this->get` is not `NULL`.

Returns: `*(this->get())`

Throws: Nothing.

`void reset(T* new_value=0);`

Effects: If `this->get()!=new_value` and `this->get()` is non-`NULL`, invoke `delete this->get()` or `cleanup_function(this->get())` as appropriate. Store `new_value` as the pointer associated with the current thread.

Postcondition: `this->get()==new_value`

Throws: `boost::thread_resource_error` if an error occurs.

`T* release();`

Effects: Return `this->get()` and store `NULL` as the pointer associated with the current thread without invoking the cleanup function.

Postcondition: `this->get()==0`

Throws: Nothing.

Date and Time Requirements

As of Boost 1.35.0, the **Boost.Thread** library uses the Boost.Date_Time library for all operations that require a time out. These include (but are not limited to):

- `boost::this_thread::sleep()`
- `timed_join()`
- `timed_wait()`
- `timed_lock()`

For the overloads that accept an absolute time parameter, an object of type `boost::system_time` is required. Typically, this will be obtained by adding a duration to the current time, obtained with a call to `boost::get_system_time()`. e.g.

```
boost::system_time const timeout=boost::get_system_time() + boost::posix_time::milliseconds(500);

extern bool done;
extern boost::mutex m;
extern boost::condition_variable cond;

boost::unique_lock<boost::mutex> lk(m);
while(!done)
{
    if(!cond.timed_wait(lk,timeout))
    {
        throw "timed out";
    }
}
```

For the overloads that accept a *TimeDuration* parameter, an object of any type that meets the Boost.Date_Time Time Duration requirements can be used, e.g.

```
boost::this_thread::sleep(boost::posix_time::milliseconds(25));

boost::mutex m;
if(m.timed_lock(boost::posix_time::nanoseconds(100)))
{
    //  ...
}
```

Typedef `system_time`

```
#include <boost/thread/thread_time.hpp>

typedef boost::posix_time::ptime system_time;
```

See the documentation for `boost::posix_time::ptime` in the Boost.Date_Time library.

Non-member function `get_system_time()`

```
#include <boost/thread/thread_time.hpp>

system_time get_system_time();
```

Returns: The current time.

Throws: Nothing.

Acknowledgments

The original implementation of **Boost.Thread** was written by William Kempf, with contributions from numerous others. This new version initially grew out of an attempt to rewrite **Boost.Thread** to William Kempf's design with fresh code that could be released under the Boost Software License. However, as the C++ Standards committee have been actively discussing standardizing a thread library for C++, this library has evolved to reflect the proposals, whilst retaining as much backwards-compatibility as possible.

Particular thanks must be given to Roland Schwarz, who contributed a lot of time and code to the original **Boost.Thread** library, and who has been actively involved with the rewrite. The scheme for dividing the platform-specific implementations into separate directories was devised by Roland, and his input has contributed greatly to improving the quality of the current implementation.

Thanks also must go to Peter Dimov, Howard Hinnant, Alexander Terekhov, Chris Thomasson and others for their comments on the implementation details of the code.