Boost.Threads

William E. Kempf

Table of Contents

Overview

Introduction

Boost.Threads allows C++ programs to execute as multiple, asynchronous, independent threads-of-execution. Each thread has its own machine state including program instruction counter and registers. Programs which execute as multiple threads are called multithreaded programs to distinguish them from traditional single-threaded programs. The glossary gives a more complete description of the multithreading execution environment.

Multithreading provides several advantages:

- Programs which would otherwise block waiting for some external event can continue to respond if the blocking operation is placed in a separate thread. Multithreading is usually an absolute requirement for these programs.
- Well-designed multithreaded programs may execute faster than single-threaded programs, particularly on multiprocessor hardware. Note, however, that poorly-designed multithreaded programs are often slower than single-threaded programs.
- Some program designs may be easier to formulate using a multithreaded approach. After all, the real world is asynchronous!

Dangers

General considerations

Beyond the errors which can occur in single-threaded programs, multithreaded programs are subject to additional errors:

- Race conditions
- Deadlock (sometimes called "deadly embrace")
- Priority failures (priority inversion, infinite overtaking, starvation, etc.)

Every multithreaded program must be designed carefully to avoid these errors. These aren't rare or exotic failures - they are virtually guaranteed to occur unless multithreaded code is designed to avoid them. Priority failures are somewhat less common, but are nonetheless serious.

The **Boost.Threads** design attempts to minimize these errors, but they will still occur unless the programmer proactively designs to avoid them.

> **Note**
> Please also see the section called “Implementation Notes” for additional, implementation-specific considerations.

Testing and debugging considerations

Multithreaded programs are non-deterministic. In other words, the same program with the same input data may follow different execution paths each time it is invoked. That can make testing and debugging a nightmare:

- Failures are often not repeatable.
- Probe effect causes debuggers to produce very different results from non-debug uses.
- Debuggers require special support to show thread state.
- Tests on a single processor system may give no indication of serious errors which would appear on multiprocessor systems, and visa versa. Thus test cases should include a varying number of processors.
- For programs which create a varying number of threads according to workload, tests which don't span the full range of possibilities may miss serious errors.

Getting a head start

Although it might appear that multithreaded programs are inherently unreliable, many reliable multithreaded programs do exist. Multithreading techniques are known which lead to reliable programs.

Design patterns for reliable multithreaded programs, including the important *monitor* pattern, are presented in *Pattern-Oriented Software Architecture Volume 2 - Patterns for Concurrent and Networked Objects*[SchmidtStalRohnertBuschmann]. Many important multithreading programming considerations (independent of threading library) are discussed in *Programming with POSIX Threads*[Butenhof97].

Doing some reading before attempting multithreaded designs will give you a head start toward reliable multithreaded programs.

C++ Standard Library usage in multithreaded programs

Runtime libraries

Warning: Multithreaded programs such as those using **Boost.Threads** must link to thread-safe versions of all runtime libraries used by the program, including the runtime library for the C++ Standard Library. Failure to do so will cause race conditions to occur when multiple threads simultaneously execute runtime library functions for `new`, `delete`, or other language features which imply shared state.

Potentially non-thread-safe functions

Certain C++ Standard Library functions inherited from C are particular problems because they hold internal state between calls:

- `rand`
- `strtok`
- `asctime`
- `ctime`

- `gmtime`
- `localtime`

It is possible to write thread-safe implementations of these by using thread specific storage (see `boost::thread_specific_ptr`), and several C++ compiler vendors do just that. The technique is well-know and is explained in [Butenhof97].

But at least one vendor (HP-UX) does not provide thread-safe implementations of the above functions in their otherwise thread-safe runtime library. Instead they provide replacement functions with different names and arguments.

Recommendation: For the most portable, yet thread-safe code, use Boost replacements for the problem functions. See the Boost Random Number Library and Boost Tokenizer Library.

Common guarantees for all Boost.Threads components

Exceptions

Boost.Threads destructors never throw exceptions. Unless otherwise specified, other **Boost.Threads** functions that do not have an exception-specification may throw implementation-defined exceptions.

In particular, **Boost.Threads** reports failure to allocate storage by throwing an exception of type `std::bad_alloc` or a class derived from `std::bad_alloc`, failure to obtain thread resources other than memory by throwing an exception of type `boost::thread_resource_error`, and certain lock related failures by throwing an exception of type `boost::lock_error`.

Rationale: Follows the C++ Standard Library practice of allowing all functions except destructors or other specified functions to throw exceptions on errors.

NonCopyable requirement

Boost.Threads classes documented as meeting the NonCopyable requirement disallow copy construction and copy assignment. For the sake of exposition, the synopsis of such classes show private derivation from `boost::noncopyable`. Users should not depend on this derivation, however, as implementations are free to meet the NonCopyable requirement in other ways.

Design

With client/server and three-tier architectures becoming common place in today's world, it's becoming increasingly important for programs to be able to handle parallel processing. Modern day operating systems usually provide some support for this through native thread APIs. Unfortunately, writing portable code that makes use of parallel processing in C++ is made very difficult by a lack of a standard interface for these native APIs. Further, these APIs are almost universally C APIs and fail to take advantage of C++'s strengths, or to address concepts unique to C++, such as exceptions.

The **Boost.Threads** library is an attempt to define a portable interface for writing parallel processes in C++.

Goals

The **Boost.Threads** library has several goals that should help to set it apart from other solutions. These goals are listed in order of precedence with full descriptions below.

Portability — **Boost.Threads** was designed to be highly portable. The goal is for the interface to be easily implemented on any platform that supports threads, and possibly even on platforms without native thread support.

Safety — **Boost.Threads** was designed to be as safe as possible. Writing thread-safe code is very difficult and successful libraries must strive to insulate the programmer from dangerous constructs as much as possible. This is accomplished in several ways:

- C++ language features are used to make correct usage easy (if possible) and error-prone usage impossible or at least more difficult. For example, see the Mutex and Lock designs, and note how they interact.
- Certain traditional concurrent programming features are considered so error-prone that they are not provided at all. For example, see the section called “Rationale for not providing *Event Variables*”.
- Dangerous features, or features which may be misused, are identified as such in the documentation to make users aware of potential pitfalls.

Flexibility — **Boost.Threads** was designed to be flexible. This goal is often at odds with *safety*. When functionality might be compromised by the desire to keep the interface safe, **Boost.Threads** has been designed to provide the functionality, but to make it's use prohibitive for general use. In other words, the interfaces have been designed such that it's usually obvious when something is unsafe, and the documentation is written to explain why.

Efficiency — **Boost.Threads** was designed to be as efficient as possible. When building a library on top of another library there is always a danger that the result will be so much slower than the "native" API that programmers are inclined to ignore the higher level API. **Boost.Threads** was designed to minimize the chances of this occurring. The interfaces have been crafted to allow an implementation the greatest chance of being as efficient as possible. This goal is often at odds with the goal for *safety*. Every effort was made to ensure efficient implementations, but when in conflict *safety* has always taken precedence.

Iterative Phases

Another goal of **Boost.Threads** was to take a dynamic, iterative approach in its development. The computing industry is still exploring the concepts of parallel programming. Most thread libraries supply only simple primitive concepts for thread synchronization. These concepts are very simple, but it is very difficult to use them safely or to provide formal proofs for constructs built on top of them. There has been a lot of research into other concepts, such as in "Communicating Sequential Processes." **Boost.Threads** was designed in iterative steps, with each step providing the building blocks necessary for the next step and giving the researcher the tools necessary to explore new concepts in a portable manner.

Given the goal of following a dynamic, iterative approach **Boost.Threads** shall go through several growth cycles. Each phase in its development shall be roughly documented here.

Phase 1, Synchronization Primitives

Boost is all about providing high quality libraries with implementations for many platforms. Unfortunately, there's a big problem faced by developers wishing to supply such high quality libraries, namely thread-safety. The C++ standard doesn't address threads at all, but real world programs often make use of native threading support. A portable library that doesn't address the issue of thread-safety is therefore not much help to a programmer who wants to use the library in his multithreaded application. So there's a very great need for portable primitives that will allow the library developer to create thread-safe imple-

mentations. This need far out weighs the need for portable methods to create and manage threads.

Because of this need, the first phase of **Boost.Threads** focuses solely on providing portable primitive concepts for thread synchronization. Types provided in this phase include the `boost::mutex`, `boost::try_mutex`, `boost::timed_mutex`, `boost::recursive_mutex`, `boost::recursive_try_mutex`, `boost::recursive_timed_mutex`, and `boost::lock_error`. These are considered the "core" synchronization primitives, though there are others that will be added in later phases.

Phase 2, Thread Management and Thread Specific Storage

This phase addresses the creation and management of threads and provides a mechanism for thread specific storage (data associated with a thread instance). Thread management is a tricky issue in C++, so this phase addresses only the basic needs of multithreaded program. Later phases are likely to add additional functionality in this area. This phase of **Boost.Threads** adds the `boost::thread` and `boost::thread_specific_ptr` types. With these additions the **Boost.Threads** library can be considered minimal but complete.

The Next Phase

The next phase will address more advanced synchronization concepts, such as read/write mutexes and barriers.

Concepts

Mutexes

> ### Note
> Certain changes to the mutexes and lock concepts are currently under discussion. In particular, the combination of the multiple lock concepts into a single lock concept is likely, and the combination of the multiple mutex concepts into a single mutex concept is also possible.

A mutex (short for mutual-exclusion) object is used to serialize access to a resource shared between multiple threads. The Mutex concept, with TryMutex and TimedMutex refinements, formalize the requirements. A model that implements Mutex and its refinements has two states: **locked** and **unlocked**. Before using a shared resource, a thread locks a **Boost.Threads** mutex object (an object whose type is a model of Mutex or one of it's refinements), ensuring thread-safe access to the shared resource. When use of the shared resource is complete, the thread unlocks the mutex object, allowing another thread to acquire the lock and use the shared resource.

Traditional C thread APIs, like POSIX threads or the Windows thread APIs, expose functions to lock and unlock a mutex object. This is dangerous since it's easy to forget to unlock a locked mutex. When the flow of control is complex, with multiple return points, the likelihood of forgetting to unlock a mutex object becomes even greater. When exceptions are thrown, it becomes nearly impossible to ensure that the mutex object is unlocked properly when using these traditional API's. The result is deadlock.

Many C++ threading libraries use a pattern known as *Scoped Locking*[SchmidtStalRohnertBuschmann] to free the programmer from the need to explicitly lock and unlock mutex objects. With this pattern, a Lock concept is employed where the lock object's constructor locks the associated mutex object and the destructor automatically does the unlocking. The **Boost.Threads** library takes this pattern to the extreme in that Lock concepts are the only way to lock and unlock a mutex object: lock and unlock functions are not exposed by any **Boost.Threads** mutex objects. This helps to ensure safe usage patterns, especially when code throws exceptions.

Locking Strategies

Every mutex object follows one of several locking strategies. These strategies define the semantics for the locking operation when the calling thread already owns a lock on the mutex object.

Recursive Locking Strategy

With a recursive locking strategy, when a thread attempts to acquire a lock on the mutex object for which it already owns a lock, the operation is successful. Note the distinction between a thread, which may have multiple locks outstanding on a recursive mutex object, and a lock object, which even for a recursive mutex object cannot have any of its lock functions called multiple times without first calling unlock.

Internally a lock count is maintained and the owning thread must unlock the mutex object the same number of times that it locked it before the mutex object's state returns to unlocked. Since mutex objects in **Boost.Threads** expose locking functionality only through lock concepts, a thread will always unlock a mutex object the same number of times that it locked it. This helps to eliminate a whole set of errors typically found in traditional C style thread APIs.

Classes `boost::recursive_mutex`, `boost::recursive_try_mutex` and `boost::recursive_timed_mutex` use this locking strategy.

Checked Locking Strategy

With a checked locking strategy, when a thread attempts to acquire a lock on the mutex object for which the thread already owns a lock, the operation will fail with some sort of error indication. Further, attempts by a thread to unlock a mutex object that was not locked by the thread will also return some sort of error indication. In **Boost.Threads**, an exception of type `boost::lock_error` would be thrown in these cases.

Boost.Threads does not currently provide any mutex objects that use this strategy.

Unchecked Locking Strategy

With an unchecked locking strategy, when a thread attempts to acquire a lock on a mutex object for which the thread already owns a lock the operation will deadlock. In general this locking strategy is less safe than a checked or recursive strategy, but it's also a faster strategy and so is employed by many libraries.

Boost.Threads does not currently provide any mutex objects that use this strategy.

Unspecified Locking Strategy

With an unspecified locking strategy, when a thread attempts to acquire a lock on a mutex object for which the thread already owns a lock the operation results in undefined behavior.

In general a mutex object with an unspecified locking strategy is unsafe, and it requires programmer discipline to use the mutex object properly. However, this strategy allows an implementation to be as fast as possible with no restrictions on its implementation. This is especially true for portable implementations that wrap the native threading support of a platform. For this reason, the classes `boost::mutex`, `boost::try_mutex` and `boost::timed_mutex` use this locking strategy despite the lack of safety.

Scheduling Policies

Every mutex object follows one of several scheduling policies. These policies define the semantics when the mutex object is unlocked and there is more than one thread waiting to acquire a lock. In other words,

the policy defines which waiting thread shall acquire the lock.

FIFO Scheduling Policy

With a FIFO ("First In First Out") scheduling policy, threads waiting for the lock will acquire it in a first-come-first-served order. This can help prevent a high priority thread from starving lower priority threads that are also waiting on the mutex object's lock.

Priority Driven Policy

With a Priority Driven scheduling policy, the thread with the highest priority acquires the lock. Note that this means that low-priority threads may never acquire the lock if the mutex object has high contention and there is always at least one high-priority thread waiting. This is known as thread starvation. When multiple threads of the same priority are waiting on the mutex object's lock one of the other scheduling priorities will determine which thread shall acquire the lock.

Unspecified Policy

The mutex object does not specify a scheduling policy. In order to ensure portability, all **Boost.Threads** mutex objects use an unspecified scheduling policy.

Mutex Concepts

Mutex Concept

A Mutex object has two states: locked and unlocked. Mutex object state can only be determined by a lock object meeting the appropriate lock concept requirements and constructed for the Mutex object.

A Mutex is NonCopyable.

For a Mutex type `M` and an object `m` of that type, the following expressions must be well-formed and have the indicated effects.

Table 1. Mutex Expressions

Expression	Effects
M m;	Constructs a mutex object m. Postcondition: m is unlocked.
(&m)->~M();	Precondition: m is unlocked. Destroys a mutex object m.
M::scoped_lock	A model of ScopedLock

TryMutex Concept

A TryMutex is a refinement of Mutex. For a TryMutex type `M` and an object `m` of that type, the following expressions must be well-formed and have the indicated effects.

Table 2. TryMutex Expressions

Expression	Effects
M::scoped_try_lock	A model of ScopedTryLock

TimedMutex Concept

A TimedMutex is a refinement of TryMutex. For a TimedMutex type `M` and an object `m` of that type, the following expressions must be well-formed and have the indicated effects.

Table 3. TimedMutex Expressions

Expression	Effects
M::scoped_timed_lock	A model of ScopedTimedLock

Mutex Models

Boost.Threads currently supplies six models of Mutex and its refinements.

Table 4. Mutex Models

Concept	Refines	Models
Mutex		`boost::mutex` `boost::recursive_mutex`
TryMutex	Mutex	`boost::try_mutex` `boost::recursive_try_mutex`
TimedMutex	TryMutex	`boost::timed_mutex` `boost::recursive_timed_mutex`

Lock Concepts

A lock object provides a safe means for locking and unlocking a mutex object (an object whose type is a model of Mutex or one of its refinements). In other words they are an implementation of the *Scoped Locking*[SchmidtStalRohnertBuschmann] pattern. The ScopedLock, ScopedTryLock, and Scoped-TimedLock concepts formalize the requirements.

Lock objects are constructed with a reference to a mutex object and typically acquire ownership of the mutex object by setting its state to locked. They also ensure ownership is relinquished in the destructor. Lock objects also expose functions to query the lock status and to manually lock and unlock the mutex object.

Lock objects are meant to be short lived, expected to be used at block scope only. The lock objects are not thread-safe. Lock objects must maintain state to indicate whether or not they've been locked and this state is not protected by any synchronization concepts. For this reason a lock object should never be shared between multiple threads.

Lock Concept

For a Lock type `L` and an object `lk` and const object `clk` of that type, the following expressions must be well-formed and have the indicated effects.

Table 5. Lock Expressions

Expression	Effects
`(&lk)->~L();`	`if (locked()) unlock();`
`(&clk)->operator const void*()`	Returns type void*, non-zero if the associated mutex object has been locked by `clk`, otherwise 0.
`clk.locked()`	Returns a `bool`, `(&clk)->operator const void*() != 0`
`lk.lock()`	Throws `boost::lock_error` if `locked()`. If the associated mutex object is already locked by some other thread, places the current thread in the Blocked state until the associated mutex is unlocked, after which the current thread is placed in the Ready state, eventually to be returned to the Running state. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object. Postcondition: `locked() == true`
`lk.unlock()`	Throws `boost::lock_error` if `!locked()`. Unlocks the associated mutex. Postcondition: `!locked()`

ScopedLock Concept

A ScopedLock is a refinement of Lock. For a ScopedLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the Mutex requirements, and an object `b` of type `bool`, the following expressions must be well-formed and have the indicated effects.

Table 6. ScopedLock Expressions

Expression	Effects
`L lk(m);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `lock()`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

TryLock Concept

A TryLock is a refinement of Lock. For a TryLock type `L` and an object `lk` of that type, the following expressions must be well-formed and have the indicated effects.

Table 7. TryLock Expressions

Expression	Effects
`lk.try_lock()`	Throws `boost::lock_error` if locked(). Makes a non-blocking attempt to lock the associated mutex object, returning `true` if the lock attempt is successful, otherwise `false`. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object.

ScopedTryLock Concept

A ScopedTryLock is a refinement of TryLock. For a ScopedTryLock type `L` and an object `lk` of that type, and an object `m` of a type meeting the TryMutex requirements, and an object `b` of type `bool`, the following expressions must be well-formed and have the indicated effects.

Table 8. ScopedTryLock Expressions

Expression	Effects
`L lk(m);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `try_lock()`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

TimedLock Concept

A TimedLock is a refinement of TryLock. For a TimedLock type `L` and an object `lk` of that type, and an object `t` of type `boost::xtime`, the following expressions must be well-formed and have the indicated effects.

Table 9. TimedLock Expressions

Expression	Effects
`lk.timed_lock(t)`	Throws `boost::lock_error` if locked(). Makes a blocking attempt to lock the associated mutex object, and returns `true` if successful within the specified time `t`, otherwise `false`. If the associated mutex object is already locked by the same thread the behavior is dependent on the locking strategy of the associated mutex object.

ScopedTimedLock Concept

A ScopedTimedLock is a refinement of TimedLock. For a ScopedTimedLock type `L` and an object `lk`

of that type, and an object `m` of a type meeting the TimedMutex requirements, and an object `b` of type `bool`, and an object `t` of type `boost::xtime`, the following expressions must be well-formed and have the indicated effects.

Table 10. ScopedTimedLock Expressions

Expression	Effects
`L lk(m,t);`	Constructs an object `lk`, and associates mutex object `m` with it, then calls `timed_lock(t)`
`L lk(m,b);`	Constructs an object `lk`, and associates mutex object `m` with it, then if `b`, calls `lock()`

Lock Models

Boost.Threads currently supplies twelve models of Lock and its refinements.

Table 11. Lock Models

Concept	Refines	Models
Lock		

ScopedLock	Lock	`boost::mutex::scoped_lock`
		`boost::recursive_mutex::scoped_lock`
		`boost::try_mutex::scoped_lock`
		`boost::recursive_try_mutex::scoped_lock`
		`boost::timed_mutex::scoped_lock`
		`boost::recursive_timed_mutex::scoped_lock`

TryLock	Lock	

ScopedTryLock	TryLock	`boost::try_mutex::scoped_try_lock`
		`boost::recursive_try_mutex::scoped_try_lock`
		`boost::timed_mutex::scoped_try_lock`
		`boost::recursive_timed_mutex::scoped_try_lock`

TimedLock	TryLock	

Concept	Refines	Models

ScopedTimedLock	TimedLock	`boost::timed_mutex::scoped_timed_lock`
		
		`boost::recursive_timed_mutex::scoped_timed_lock`

Rationale

This page explains the rationale behind various design decisions in the **Boost.Threads** library. Having the rationale documented here should explain how we arrived at the current design as well as prevent future rehashing of discussions and thought processes that have already occurred. It can also give users a lot of insight into the design process required for this library.

Rationale for the Creation of Boost.Threads

Processes often have a degree of "potential parallelism" and it can often be more intuitive to design systems with this in mind. Further, these parallel processes can result in more responsive programs. The benefits for multithreaded programming are quite well known to most modern programmers, yet the C++ language doesn't directly support this concept.

Many platforms support multithreaded programming despite the fact that the language doesn't support it. They do this through external libraries, which are, unfortunately, platform specific. POSIX has tried to address this problem through the standardization of a "pthread" library. However, this is a standard only on POSIX platforms, so its portability is limited.

Another problem with POSIX and other platform specific thread libraries is that they are almost universally C based libraries. This leaves several C++ specific issues unresolved, such as what happens when an exception is thrown in a thread. Further, there are some C++ concepts, such as destructors, that can make usage much easier than what's available in a C library.

What's truly needed is C++ language support for threads. However, the C++ standards committee needs existing practice or a good proposal as a starting point for adding this to the standard.

The **Boost.Threads** library was developed to provide a C++ developer with a portable interface for writing multithreaded programs on numerous platforms. There's a hope that the library can be the basis for a more detailed proposal for the C++ standards committee to consider for inclusion in the next C++ standard.

Rationale for the Low Level Primitives Supported in Boost.Threads

The **Boost.Threads** library supplies a set of low level primitives for writing multithreaded programs, such as mutexes and condition variables. In fact, the first release of **Boost.Threads** supports only these low level primitives. However, computer science research has shown that use of these primitives is difficult since it's difficult to mathematically prove that a usage pattern is correct, meaning it doesn't result in race conditions or deadlocks. There are several algebras (such as CSP, CCS and Join calculus) that have been developed to help write provably correct parallel processes. In order to prove the correctness these processes must be coded using higher level abstractions. So why does **Boost.Threads** support the lower level concepts?

The reason is simple: the higher level concepts need to be implemented using at least some of the lower

level concepts. So having portable lower level concepts makes it easier to develop the higher level concepts and will allow researchers to experiment with various techniques.

Beyond this theoretical application of higher level concepts, however, the fact remains that many multithreaded programs are written using only the lower level concepts, so they are useful in and of themselves, even if it's hard to prove that their usage is correct. Since many users will be familiar with these lower level concepts but unfamiliar with any of the higher level concepts, supporting the lower level concepts provides greater accessibility.

Rationale for the Lock Design

Programmers who are used to multithreaded programming issues will quickly note that the **Boost.Threads** design for mutex lock concepts is not thread-safe (this is clearly documented as well). At first this may seem like a serious design flaw. Why have a multithreading primitive that's not thread-safe itself?

A lock object is not a synchronization primitive. A lock object's sole responsibility is to ensure that a mutex is both locked and unlocked in a manner that won't result in the common error of locking a mutex and then forgetting to unlock it. This means that instances of a lock object are only going to be created, at least in theory, within block scope and won't be shared between threads. Only the mutex objects will be created outside of block scope and/or shared between threads. Though it's possible to create a lock object outside of block scope and to share it between threads, to do so would not be a typical usage (in fact, to do so would likely be an error). Nor are there any cases when such usage would be required.

Lock objects must maintain some state information. In order to allow a program to determine if a try_lock or timed_lock was successful the lock object must retain state indicating the success or failure of the call made in its constructor. If a lock object were to have such state and remain thread-safe it would need to synchronize access to the state information which would result in roughly doubling the time of most operations. Worse, since checking the state can occur only by a call after construction, we'd have a race condition if the lock object were shared between threads.

So, to avoid the overhead of synchronizing access to the state information and to avoid the race condition, the **Boost.Threads** library simply does nothing to make lock objects thread-safe. Instead, sharing a lock object between threads results in undefined behavior. Since the only proper usage of lock objects is within block scope this isn't a problem, and so long as the lock object is properly used there's no danger of any multithreading issues.

Rationale for NonCopyable Thread Type

Programmers who are used to C libraries for multithreaded programming are likely to wonder why **Boost.Threads** uses a noncopyable design for `boost::thread`. After all, the C thread types are copyable, and you often have a need for copying them within user code. However, careful comparison of C designs to C++ designs shows a flaw in this logic.

All C types are copyable. It is, in fact, not possible to make a noncopyable type in C. For this reason types that represent system resources in C are often designed to behave very similarly to a pointer to dynamic memory. There's an API for acquiring the resource and an API for releasing the resource. For memory we have pointers as the type and alloc/free for the acquisition and release APIs. For files we have FILE* as the type and fopen/fclose for the acquisition and release APIs. You can freely copy instances of the types but must manually manage the lifetime of the actual resource through the acquisition and release APIs.

C++ designs recognize that the acquisition and release APIs are error prone and try to eliminate possible errors by acquiring the resource in the constructor and releasing it in the destructor. The best example of such a design is the std::iostream set of classes which can represent the same resource as the FILE* type in C. A file is opened in the std::fstream's constructor and closed in its destructor. However, if an iostream were copyable it could lead to a file being closed twice, an obvious error, so the std::iostream

types are noncopyable by design. This is the same design used by boost::thread, which is a simple and easy to understand design that's consistent with other C++ standard types.

During the design of boost::thread it was pointed out that it would be possible to allow it to be a copyable type if some form of "reference management" were used, such as ref-counting or ref-lists, and many argued for a boost::thread_ref design instead. The reasoning was that copying "thread" objects was a typical need in the C libraries, and so presumably would be in the C++ libraries as well. It was also thought that implementations could provide more efficient reference management than wrappers (such as boost::shared_ptr) around a noncopyable thread concept. Analysis of whether or not these arguments would hold true doesn't appear to bear them out. To illustrate the analysis we'll first provide pseudo-code illustrating the six typical usage patterns of a thread object.

1. Use case: Simple creation of a thread.

```
void foo()
{
   create_thread(&bar);
}
```

2. Use case: Creation of a thread that's later joined.

```
void foo()
{
   thread = create_thread(&bar);
   join(thread);
}
```

3. Use case: Simple creation of several threads in a loop.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      create_thread(&bar);
}
```

4. Use case: Creation of several threads in a loop which are later joined.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = create_thread(&bar);
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i].join();
}
```

5. Use case: Creation of a thread whose ownership is passed to an-

other object/method.

```
void foo()
{
   thread = create_thread(&bar);
   manager.owns(thread);
}
```

6. Use case: Creation of a thread whose ownership is shared between multiple objects.

```
void foo()
{
   thread = create_thread(&bar);
   manager1.add(thread);
   manager2.add(thread);
}
```

Of these usage patterns there's only one that requires reference management (number 6). Hopefully it's fairly obvious that this usage pattern simply won't occur as often as the other usage patterns. So there really isn't a "typical need" for a thread concept, though there is some need.

Since the need isn't typical we must use different criteria for deciding on either a thread_ref or thread design. Possible criteria include ease of use and performance. So let's analyze both of these carefully.

With ease of use we can look at existing experience. The standard C++ objects that represent a system resource, such as std::iostream, are noncopyable, so we know that C++ programmers must at least be experienced with this design. Most C++ developers are also used to smart pointers such as boost::shared_ptr, so we know they can at least adapt to a thread_ref concept with little effort. So existing experience isn't going to lead us to a choice.

The other thing we can look at is how difficult it is to use both types for the six usage patterns above. If we find it overly difficult to use a concept for any of the usage patterns there would be a good argument for choosing the other design. So we'll code all six usage patterns using both designs.

1. Comparison: simple creation of a thread.

```
void foo()
{
   thread thrd(&bar);
}
void foo()
{
   thread_ref thrd = create_thread(&bar);
}
```

2. Comparison: creation of a thread that's later joined.

```
void foo()
```

```
{
   thread thrd(&bar);
   thrd.join();
}
void foo()
{
   thread_ref thrd =
   create_thread(&bar);thrd->join();
}
```

3. Comparison: simple creation of several threads in a loop.

```
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      thread thrd(&bar);
}
void foo()
{
   for (int i=0; i<NUM_THREADS; ++i)
      thread_ref thrd = create_thread(&bar);
}
```

4. Comparison: creation of several threads in a loop which are later joined.

```
void foo()
{
   std::auto_ptr<thread> threads[NUM_THREADS];
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = std::auto_ptr<thread>(new thread(&bar));
   for (int i= 0; i<NUM_THREADS;
       ++i)threads[i]->join();
}
void foo()
{
   thread_ref threads[NUM_THREADS];
   for (int i=0; i<NUM_THREADS; ++i)
      threads[i] = create_thread(&bar);
   for (int i= 0; i<NUM_THREADS;
      ++i)threads[i]->join();
}
```

5. Comparison: creation of a thread whose ownership is passed to another object/method.

```
void foo()
{
   thread thrd* = new thread(&bar);
   manager.owns(thread);
}
void foo()
```

```
{
   thread_ref thrd = create_thread(&bar);
   manager.owns(thrd);
}
```

6. Comparison: creation of a thread whose ownership is shared between multiple objects.

```
void foo()
{
   boost::shared_ptr<thread> thrd(new thread(&bar));
   manager1.add(thrd);
   manager2.add(thrd);
}
void foo()
{
   thread_ref thrd = create_thread(&bar);
   manager1.add(thrd);
   manager2.add(thrd);
}
```

This shows the usage patterns being nearly identical in complexity for both designs. The only actual added complexity occurs because of the use of operator new in (4), (5), and (6); and the use of std::auto_ptr and boost::shared_ptr in (4) and (6) respectively. However, that's not really much added complexity, and C++ programmers are used to using these idioms anyway. Some may dislike the presence of operator new in user code, but this can be eliminated by proper design of higher level concepts, such as the boost::thread_group class that simplifies example (4) down to:

```
void foo()
{
   thread_group threads;
   for (int i=0; i<NUM_THREADS; ++i)
      threads.create_thread(&bar);
   threads.join_all();
}
```

So ease of use is really a wash and not much help in picking a design.

So what about performance? Looking at the above code examples, we can analyze the theoretical impact to performance that both designs have. For (1) we can see that platforms that don't have a ref-counted native thread type (POSIX, for instance) will be impacted by a thread_ref design. Even if the native thread type is ref-counted there may be an impact if more state information has to be maintained for concepts foreign to the native API, such as clean up stacks for Win32 implementations. For (2) and (3) the performance impact will be identical to (1). For (4) things get a little more interesting and we find that theoretically at least the thread_ref may perform faster since the thread design requires dynamic memory allocation/deallocation. However, in practice there may be dynamic allocation for the thread_ref design as well, it will just be hidden from the user. As long as the implementation has to do dynamic allocations the thread_ref loses again because of the reference management. For (5) we see the same impact as we do for (4). For (6) we still have a possible impact to the thread design because of dynamic allocation but thread_ref no longer suffers because of its reference management, and in fact, theoretically at least, the thread_ref may do a better job of managing the references. All of this indicates that thread wins for (1), (2) and (3); with (4) and (5) the winner depending on the implementation and the platform but with the

thread design probably having a better chance; and with (6) it will again depend on the implementation and platform but this time we favor thread_ref slightly. Given all of this it's a narrow margin, but the thread design prevails.

Given this analysis, and the fact that noncopyable objects for system resources are the normal designs that C++ programmers are used to dealing with, the **Boost.Threads** library has gone with a noncopyable design.

Rationale for not providing *Event Variables*

Event variables are simply far too error-prone. `boost::condition` variables are a much safer alternative. [Note that Graphical User Interface *events* are a different concept, and are not what is being discussed here.]

Event variables were one of the first synchronization primitives. They are still used today, for example, in the native Windows multithreading API. Yet both respected computer science researchers and experienced multithreading practitioners believe event variables are so inherently error-prone that they should never be used, and thus should not be part of a multithreading library.

Per Brinch Hansen [Hansen73] analyzed event variables in some detail, pointing out [emphasis his] that "*event operations force the programmer to be aware of the relative speeds of the sending and receiving processes*". His summary:

> We must therefore conclude that event variables of the previous type are impractical for system design. *The effect of an interaction between two processes must be independent of the speed at which it is carried out.*

Experienced programmers using the Windows platform today report that event variables are a continuing source of errors, even after previous bad experiences caused them to be very careful in their use of event variables. Overt problems can be avoided, for example, by teaming the event variable with a mutex, but that may just convert a race condition into another problem, such as excessive resource use. One of the most distressing aspects of the experience reports is the claim that many defects are latent. That is, the programs appear to work correctly, but contain hidden timing dependencies which will cause them to fail when environmental factors or usage patterns change, altering relative thread timings.

The decision to exclude event variables from **Boost.Threads** has been surprising to some Windows programmers. They have written programs which work using event variables, and wonder what the problem is. It seems similar to the "goto considered harmful" controversy of 30 years ago. It isn't that events, like gotos, can't be made to work, but rather that virtually all programs using alternatives will be easier to write, debug, read, maintain, and will be less likely to contain latent defects.

[Rationale provided by Beman Dawes]

Reference

Header <boost/thread/barrier.hpp>

```
namespace boost {
  class barrier;
}
```

Class barrier

Class barrier --

An object of class barrier is a synchronization primitive used to cause a set of threads to wait until they each perform a certain function or each reach a particular point in their execution.

```
class barrier : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  barrier(size_t);
  ~barrier();

  // waiting
  bool wait();
};
```

Description

When a barrier is created, it is initialized with a thread count N. The first N-1 calls to `wait()` will all cause their threads to be blocked. The Nth call to `wait()` will allow all of the waiting threads, including the Nth thread, to be placed in a ready state. The Nth call will also "reset" the barrier such that, if an additional N+1th call is made to `wait()`, it will be as though this were the first call to `wait()`; in other words, the N+1th to 2N-1th calls to `wait()` will cause their threads to be blocked, and the 2Nth call to `wait()` will allow all of the waiting threads, including the 2Nth thread, to be placed in a ready state and reset the barrier. This functionality allows the same set of N threads to re-use a barrier object to synchronize their execution at multiple points during their execution.

See Glossary for definitions of thread states blocked and ready. Note that "waiting" is a synonym for blocked.

`barrier` construct/copy/destruct

1.
   ```
   barrier(size_t count);
   ```

   Effects Constructs a barrier object that will cause `count` threads to block on a call to `wait()`.

2.
   ```
   ~barrier();
   ```

   Effects Destroys `*this`. If threads are still executing their `wait()` operations, the behavior for these threads is undefined.

`barrier` waiting

1.
   ```
   bool wait();
   ```

Effects — Wait until N threads call `wait()`, where N equals the `count` provided to the constructor for the barrier object.

Note that if the barrier is destroyed before `wait()` can return, the behavior is undefined.

Returns — Exactly one of the N threads will receive a return value of `true`, the others will receive a value of `false`. Precisely which thread receives the return value of `true` will be implementation-defined. Applications can use this value to designate one thread as a leader that will take a certain action, and the other threads emerging from the barrier can wait for that action to take place.

Header <boost/thread/condition.hpp>

```
namespace boost {
  class condition;
}
```

Class condition

Class condition --

An object of class condition is a synchronization primitive used to cause a thread to wait until a particular shared-data condition (or time) is met.

```
class condition : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  condition();
  ~condition();

  // notification
  void notify_one();
  void notify_all();

  // waiting
  template<typename ScopedLock> void wait(ScopedLock&);
  template<typename ScopedLock, typename Pred> void wait(ScopedLock&, Pred);
  template<typename ScopedLock>
    bool timed_wait(ScopedLock&, const boost::xtime&);
  template<typename ScopedLock, typename Pred>
    bool timed_wait(ScopedLock&, Pred);
};
```

Description

A condition object is always used in conjunction with a mutex object (an object whose type is a model of a Mutex or one of its refinements). The mutex object must be locked prior to waiting on the condition, which is verified by passing a lock object (an object whose type is a model of Lock or one of its refinements) to the condition object's wait functions. Upon blocking on the condition object, the thread unlocks the mutex object. When the thread returns from a call to one of the condition object's wait functions the mutex object is again locked. The tricky unlock/lock sequence is performed automatically by the condition object's wait functions.

The condition type is often used to implement the Monitor Object and other important patterns (see [SchmidtStalRohnertBuschmann] and [Hoare74]). Monitors are one of the most important patterns for creating reliable multithreaded programs.

See Glossary for definitions of thread states blocked and ready. Note that "waiting" is a synonym for blocked.

`condition` construct/copy/destruct

1.
```
condition();
```

Effects Constructs a condition object.

2.
```
~condition();
```

Effects Destroys `*this`.

`condition` notification

1.
```
void notify_one();
```

Effects If there is a thread waiting on `*this`, change that thread's state to ready. Otherwise there is no effect.

Notes If more than one thread is waiting on `*this`, it is unspecified which is made ready. After returning to a ready state the notified thread must still acquire the mutex again (which occurs within the call to one of the condition object's wait functions.)

2.
```
void notify_all();
```

Effects Change the state of all threads waiting on `*this` to ready. If there are no waiting threads, `notify_all()` has no effect.

`condition` waiting

1.
```
template<typename ScopedLock> void wait(ScopedLock& lock);
```

Requires `ScopedLock` meets the ScopedLock requirements.

Effects Releases the lock on the mutex object associated with `lock`, blocks the current thread of execution until readied by a call to `this->notify_one()` or `this->notify_all()`, and then reacquires the lock.

Throws lock_error if `!lock.locked()`

2.
```
template<typename ScopedLock, typename Pred>
  void wait(ScopedLock& lock, Pred pred);
```

Requires `ScopedLock` meets the ScopedLock requirements and the return from `pred()` is convertible to `bool`.

Effects As if: `while (!pred()) wait(lock)`

Throws lock_error if `!lock.locked()`

3.
```
template<typename ScopedLock>
  bool timed_wait(ScopedLock& lock, const boost::xtime& xt);
```

Requires `ScopedLock` meets the ScopedLock requirements.

Effects Releases the lock on the mutex object associated with `lock`, blocks the current

thread of execution until readied by a call to `this->notify_one()` or `this->notify_all()`, or until time `xt` is reached, and then reacquires the lock.

Returns `false` if time `xt` is reached, otherwise `true`.

Throws lock_error if `!lock.locked()`

4.

```
template<typename ScopedLock, typename Pred>
  bool timed_wait(ScopedLock& lock, Pred pred);
```

Requires `ScopedLock` meets the ScopedLock requirements and the return from `pred()` is convertible to `bool`.

Effects As if: `while (!pred()) { if (!timed_wait(lock, xt)) return false; } return true;`

Returns `false` if `xt` is reached, otherwise `true`.

Throws lock_error if `!lock.locked()`

Header <boost/thread/exceptions.hpp>

```
namespace boost {
  class lock_error;
  class thread_resource_error;
}
```

Class lock_error

Class lock_error --

The lock_error class defines an exception type thrown to indicate a locking related error has been detected.

```
class lock_error : public std::logical_error {
public:
  // construct/copy/destruct
  lock_error();
};
```

Description

Examples of errors indicated by a lock_error exception include a lock operation which can be determined to result in a deadlock, or unlock operations attempted by a thread that does not own the lock.

`lock_error` construct/copy/destruct

1. `lock_error();`

   Effects Constructs a `lock_error` object.

Class thread_resource_error

Class thread_resource_error --

The thread_resource_error class defines an exception type that is thrown by constructors in the **Boost.Threads** library when thread-related resources can not be acquired.

```
class thread_resource_error : public std::runtime_error {
public:
  // construct/copy/destruct
  thread_resource_error();
};
```

Description

thread_resource_error is used only when thread-related resources cannot be acquired; memory allocation failures are indicated by std::bad_alloc.

`thread_resource_error` construct/copy/destruct

1.
```
thread_resource_error();
```

Effects Constructs a `thread_resource_error` object.

Header <boost/thread/mutex.hpp>

```
namespace boost {
  class mutex;
  class try_mutex;
  class timed_mutex;
}
```

Class mutex

Class mutex --

The mutex class is a model of the Mutex concept.

```
class mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;

  // construct/copy/destruct
  mutex();
  ~mutex();
};
```

Description

The mutex class is a model of the Mutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see try_mutex and timed_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The mutex class supplies the following typedef, which models the specified locking strategy:

Lock Name	Lock Concept
scoped_lock	ScopedLock

The mutex class uses an Unspecified locking strategy, so attempts to recursively lock a mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`mutex` construct/copy/destruct

1. `mutex();`

   Effects: Constructs a mutex object.

   Postconditions: `*this` is in an unlocked state.

2. `~mutex();`

Effects	Destroys a mutex object.
Requires	`*this` is in an unlocked state.
Notes	**Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class try_mutex

Class try_mutex --

The try_mutex class is a model of the TryMutex concept.

```
class try_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;

  // construct/copy/destruct
  try_mutex();
  ~try_mutex();
};
```

Description

The try_mutex class is a model of the TryMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see mutex and timed_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The try_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock

The try_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a try_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, try_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`try_mutex` construct/copy/destruct

1.
```
try_mutex();
```

Effects: Constructs a try_mutex object.

Postconditions: `*this` is in an unlocked state.

2.
```
~try_mutex();
```

Effects Destroys a try_mutex object.

Requires `*this` is in an unlocked state.

Notes **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class timed_mutex

Class timed_mutex --

The timed_mutex class is a model of the TimedMutex concept.

```
class timed_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;
  typedef implementation-defined scoped_timed_lock;

  // construct/copy/destruct
  timed_mutex();
  ~timed_mutex();
};
```

Description

The timed_mutex class is a model of the TimedMutex concept. It should be used to synchronize access to shared resources using Unspecified locking mechanics.

For classes that model related mutex concepts, see mutex and try_mutex.

For Recursive locking mechanics, see recursive_mutex, recursive_try_mutex, and recursive_timed_mutex.

The timed_mutex class supplies the following typedefs, which model the specified locking strategies:

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock
scoped_timed_lock	ScopedTimedLock

The timed_mutex class uses an Unspecified locking strategy, so attempts to recursively lock a timed_mutex object or attempts to unlock one by threads that don't own a lock on it result in **undefined behavior**. This strategy allows implementations to be as efficient as possible on any given platform. It is, however, recommended that implementations include debugging support to detect misuse when `NDEBUG` is not defined.

Like all mutex models in **Boost.Threads**, timed_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`timed_mutex` construct/copy/destruct

1.
```
timed_mutex();
```

Effects Constructs a timed_mutex object.

Postconditions `*this` is in an unlocked state.

2.
```
~timed_mutex();
```

Effects Destroys a timed_mutex object.

Requires `*this` is in an unlocked state.

Notes **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Header <boost/thread/once.hpp>

```
BOOST_ONCE_INIT


namespace boost {
  typedef implementation-defined once_flag;  // The call_once function and
                        once_flag type (statically initialized to
                        BOOST_ONCE_INIT) can be used to run a
                        routine exactly once. This can be used to initialize data
                        thread-safe
                        manner.
   call_once(void (*func)(), once_flag&);
}
```

Macro BOOST_ONCE_INIT

Macro BOOST_ONCE_INIT -- The call_once function and `once_flag` type (statically initialized to BOOST_ONCE_INIT) can be used to run a routine exactly once. This can be used to initialize data in a thread-safe manner.

```
BOOST_ONCE_INIT
```

Description

The implementation-defined macro BOOST_ONCE_INIT is a constant value used to initialize `once_flag` instances to indicate that the logically associated routine has not been run yet. See call_once for more details.

Function call_once

Function call_once -- The call_once function and `once_flag` type (statically initialized to BOOST_ONCE_INIT) can be used to run a routine exactly once. This can be used to initialize data in a thread-safe manner.

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

Description

Example usage is as follows:

```
//Example usage:
boost::once_flag once = BOOST_ONCE_INIT;

void init()
{
    //...
}

void thread_proc()
{
    boost::call_once(&init, once);
}
```

Requires	The function `func` shall not throw exceptions.
Effects	As if (in an atomic fashion): `if (flag == BOOST_ONCE_INIT) func();`
Postconditions	`flag != BOOST_ONCE_INIT`

Header <boost/thread/recursive_mutex.hpp>

```
namespace boost {
  class recursive_mutex;
  class recursive_try_mutex;
  class recursive_timed_mutex;
}
```

Class recursive_mutex

Class recursive_mutex --

The recursive_mutex class is a model of the Mutex concept.

```
class recursive_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;

  // construct/copy/destruct
  recursive_mutex();
  ~recursive_mutex();
};
```

Description

The recursive_mutex class is a model of the Mutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_try_mutex and recursive_timed_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_mutex class supplies the following typedef, which models the specified locking strategy:

Table 12. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock

The recursive_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_mutex` construct/copy/destruct

1. `recursive_mutex();`

   Effects: Constructs a recursive_mutex object.

   Postconditions: `*this` is in an unlocked state.

2. `~recursive_mutex();`

Effects	Destroys a recursive_mutex object.
Requires	`*this` is in an unlocked state.
Notes	**Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class recursive_try_mutex

Class recursive_try_mutex --

The recursive_try_mutex class is a model of the TryMutex concept.

```
class recursive_try_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;

  // construct/copy/destruct
  recursive_try_mutex();
  ~recursive_try_mutex();
};
```

Description

The recursive_try_mutex class is a model of the TryMutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_mutex and recursive_timed_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_try_mutex class supplies the following typedefs, which model the specified locking strategies:

Table 13. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock

The recursive_try_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_try_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_try_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_try_mutex` construct/copy/destruct

1.
```
recursive_try_mutex();
```

Effects Constructs a recursive_try_mutex object.

Postconditions `*this` is in an unlocked state.

2. `~recursive_try_mutex();`

   Effects Destroys a recursive_try_mutex object.

   Requires `*this` is in an unlocked state.

   Notes **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Class recursive_timed_mutex

Class recursive_timed_mutex --

The recursive_timed_mutex class is a model of the TimedMutex concept.

```
class recursive_timed_mutex : private boost::noncopyable   // Exposition only
{
public:
  // types
  typedef implementation-defined scoped_lock;
  typedef implementation-defined scoped_try_lock;
  typedef implementation-defined scoped_timed_lock;

  // construct/copy/destruct
  recursive_timed_mutex();
  ~recursive_timed_mutex();
};
```

Description

The recursive_timed_mutex class is a model of the TimedMutex concept. It should be used to synchronize access to shared resources using Recursive locking mechanics.

For classes that model related mutex concepts, see recursive_mutex and recursive_try_mutex.

For Unspecified locking mechanics, see mutex, try_mutex, and timed_mutex.

The recursive_timed_mutex class supplies the following typedefs, which model the specified locking strategies:

Table 14. Supported Lock Types

Lock Name	Lock Concept
scoped_lock	ScopedLock
scoped_try_lock	ScopedTryLock
scoped_timed_lock	ScopedTimedLock

The recursive_timed_mutex class uses a Recursive locking strategy, so attempts to recursively lock a recursive_timed_mutex object succeed and an internal "lock count" is maintained. Attempts to unlock a recursive_mutex object by threads that don't own a lock on it result in **undefined behavior**.

Like all mutex models in **Boost.Threads**, recursive_timed_mutex leaves the scheduling policy as Unspecified. Programmers should make no assumptions about the order in which waiting threads acquire a lock.

`recursive_timed_mutex` construct/copy/destruct

1.
```
recursive_timed_mutex();
```

Effects — Constructs a recursive_timed_mutex object.

Postconditions — `*this` is in an unlocked state.

2.
```
~recursive_timed_mutex();
```

Effects — Destroys a recursive_timed_mutex object.

Requires — `*this` is in an unlocked state.

Notes — **Danger:** Destruction of a locked mutex is a serious programming error resulting in undefined behavior such as a program crash.

Header <boost/thread/thread.hpp>

```
namespace boost {
  class thread;
  class thread_group;
}
```

Class thread

Class thread --

The thread class represents threads of execution, and provides the functionality to create and manage threads within the **Boost.Threads** library. See Glossary for a precise description of thread of execution, and for definitions of threading-related terms and of thread states such as blocked.

```
class thread : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread();
  explicit thread(const boost::function0<void>&);
  ~thread();

  // comparison
  bool operator==() const;
  bool operator!=() const;

  // modifier
  void join();

  // static
  static void sleep(const xtime&);
  static void yield();
};
```

Description

A thread of execution has an initial function. For the program's initial thread, the initial function is `main()`. For other threads, the initial function is `operator()` of the function object passed to the thread object's constructor.

A thread of execution is said to be "finished" or to have "finished execution" when its initial function returns or is terminated. This includes completion of all thread cleanup handlers, and completion of the normal C++ function return behaviors, such as destruction of automatic storage (stack) objects and releasing any associated implementation resources.

A thread object has an associated state which is either "joinable" or "non-joinable".

Except as described below, the policy used by an implementation of **Boost.Threads** to schedule transitions between thread states is unspecified.

Note

Just as the lifetime of a file may be different from the lifetime of an `iostream` object which represents the file, the lifetime of a thread of execution may be different from the thread object which represents the thread of execution. In particular, after a call to `join()`, the thread of execution will no longer exist even though the thread object continues to exist until the end of its normal lifetime. The converse is also possible; if a thread object is destroyed without `join()` first having been called, the thread of execution continues until its initial function completes.

`thread` construct/copy/destruct

1. `thread();`

   Effects: Constructs a thread object representing the current thread of execution.

   Postconditions: `*this` is non-joinable.

   Notes: **Danger:** `*this` is valid only within the current thread.

2. `explicit thread(const boost::function0<void>& threadfunc);`

   Effects: Starts a new thread of execution and constructs a thread object representing it. Copies `threadfunc` (which in turn copies the function object wrapped by `threadfunc`) to an internal location which persists for the lifetime of the new thread of execution. Calls `operator()` on the copy of the `threadfunc` function object in the new thread of execution.

   Postconditions: `*this` is joinable.

   Throws: `boost::thread_resource_error` if a new thread of execution cannot be started.

3. `~thread();`

   Effects: Destroys `*this`. The actual thread of execution may continue to execute after the thread object has been destroyed.

   Notes: If `*this` is joinable the actual thread of execution becomes "detached". Any resources used by the thread will be reclaimed when the thread of execution completes. To ensure such a thread of execution runs to completion before the thread object is destroyed, call `join()`.

thread comparison

1. `bool operator==( rhs) const;`

   Requires: The thread is non-terminated or `*this` is joinable.

   Returns: `true` if `*this` and `rhs` represent the same thread of execution.

2. `bool operator!=( rhs) const;`

   Requires: The thread is non-terminated or `*this` is joinable.

   Returns: `!(*this==rhs)`.

thread modifier

1.
```
void join();
```

Requires `*this` is joinable.

Effects The current thread of execution blocks until the initial function of the thread of execution represented by `*this` finishes and all resources are reclaimed.

Notes If `*this == thread()` the result is implementation-defined. If the implementation doesn't detect this the result will be deadlock.

thread static

1.
```
static void sleep(const xtime& xt);
```

Effects The current thread of execution blocks until `xt` is reached.

2.
```
static void yield();
```

Effects The current thread of execution is placed in the ready state.

Notes Allow the current thread to give up the rest of its time slice (or other scheduling quota) to another thread. Particularly useful in non-preemptive implementations.

Class thread_group

Class thread_group -- The thread_group class provides a container for easy grouping of threads to simplify several common thread creation and management idioms.

```
class thread_group : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread_group();
  ~thread_group();

  // modifier
  thread* create_thread(const boost::function0<void>&);
  void add_thread(thread* thrd);
  void remove_thread(thread* thrd);
  void join_all();
};
```

Description

thread_group construct/copy/destruct

1. `thread_group();`

   Effects: Constructs an empty thread_group container.

2. `~thread_group();`

   Effects: Destroys each contained thread object. Destroys `*this`.

   Notes: Behavior is undefined if another thread references `*this` during the execution of the destructor.

thread_group modifier

1. `thread* create_thread(const boost::function0<void>& threadfunc);`

   Effects: Creates a new thread object that executes `threadfunc` and adds it to the `thread_group` container object's list of managed thread objects.

   Returns: Pointer to the newly created thread object.

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

   Effects: Adds `thrd` to the thread_group object's list of managed thread objects. The `thrd` ob-

ject must have been allocated via `operator new` and will be deleted when the group is destroyed.

3.
```
void remove_thread(thread* thrd thrd);
```

Effects Removes `thread` from `*this`'s list of managed thread objects.

Throws **???** if `thrd` is not in `*this`'s list of managed thread objects.

4.
```
void join_all();
```

Effects Calls `join()` on each of the managed thread objects.

Header <boost/thread/tss.hpp>

```
namespace boost {
  class thread_specific_ptr;
}
```

Class thread_specific_ptr

Class thread_specific_ptr -- The thread_specific_ptr class defines an interface for using thread specific storage.

```
class thread_specific_ptr : private boost::noncopyable   // Exposition only
{
public:
  // construct/copy/destruct
  thread_specific_ptr();
  thread_specific_ptr(void (*cleanup)(void*));
  ~thread_specific_ptr();

  // modifier functions
  T* release();
  void reset(T* = 0);

  // observer functions
  T* get() const;
  T* operator->() const;
  T& operator*()() const;
};
```

Description

Thread specific storage is data associated with individual threads and is often used to make operations that rely on global data thread-safe.

Template thread_specific_ptr stores a pointer to an object obtained on a thread-by-thread basis and calls a specified cleanup handler on the contained pointer when the thread terminates. The cleanup handlers are called in the reverse order of construction of the thread_specific_ptrs, and for the initial thread are called by the destructor, providing the same ordering guarantees as for normal declarations. Each thread initially stores the null pointer in each thread_specific_ptr instance.

The template thread_specific_ptr is useful in the following cases:

- An interface was originally written assuming a single thread of control and it is being ported to a multithreaded environment.
- Each thread of control invokes sequences of methods that share data that are physically unique for each thread, but must be logically accessed through a globally visible access point instead of being explicitly passed.

`thread_specific_ptr` construct/copy/destruct

1. `thread_specific_ptr();`

   Requires: The expression `delete get()` is well formed.

   Effects: A thread-specific data key is allocated and visible to all threads in the process. Upon creation, the value `NULL` will be associated with the new key in all active threads. A cleanup method is registered with the key that will call `delete` on the

value associated with the key for a thread when it exits. When a thread exits, if a key has a registered cleanup method and the thread has a non-`NULL` value associated with that key, the value of the key is set to `NULL` and then the cleanup method is called with the previously associated value as its sole argument. The order in which registered cleanup methods are called when a thread exits is undefined. If after all the cleanup methods have been called for all non-`NULL` values, there are still some non-`NULL` values with associated cleanup handlers the result is undefined behavior.

Throws boost::thread_resource_error if the necessary resources can not be obtained.

Notes There may be an implementation specific limit to the number of thread specific storage objects that can be created, and this limit may be small.

Rationale The most common need for cleanup will be to call `delete` on the associated value. If other forms of cleanup are required the overloaded constructor should be called instead.

2.
```
thread_specific_ptr(void (*cleanup)(void*) cleanup);
```

Effects A thread-specific data key is allocated and visible to all threads in the process. Upon creation, the value `NULL` will be associated with the new key in all active threads. The `cleanup` method is registered with the key and will be called for a thread with the value associated with the key for that thread when it exits. When a thread exits, if a key has a registered cleanup method and the thread has a non-`NULL` value associated with that key, the value of the key is set to `NULL` and then the cleanup method is called with the previously associated value as its sole argument. The order in which registered cleanup methods are called when a thread exits is undefined. If after all the cleanup methods have been called for all non-`NULL` values, there are still some non-`NULL` values with associated cleanup handlers the result is undefined behavior.

Throws boost::thread_resource_error if the necessary resources can not be obtained.

Notes There may be an implementation specific limit to the number of thread specific storage objects that can be created, and this limit may be small.

Rationale There is the occasional need to register specialized cleanup methods, or to register no cleanup method at all (done by passing `NULL` to this constructor.

3.
```
~thread_specific_ptr();
```

Effects Deletes the thread-specific data key allocated by the constructor. The thread-specific data values associated with the key need not be `NULL`. It is the responsibility of the application to perform any cleanup actions for data associated with the key.

Notes Does not destroy any data that may be stored in any thread's thread specific storage. For this reason you should not destroy a thread_specific_ptr object until you are certain there are no threads running that have made use of its thread specific storage.

Rationale Associated data is not cleaned up because registered cleanup methods need to be run in the thread that allocated the associated data to be guarranteed to work correctly. There's no safe way to inject the call into another thread's execution path,

making it impossible to call the cleanup methods safely.

`thread_specific_ptr` modifier functions

1. `T* release();`

   Postconditions: `*this` holds the null pointer for the current thread.

   Returns: `this->get()` prior to the call.

   Rationale: This method provides a mechanism for the user to relinquish control of the data associated with the thread-specific key.

2. `void reset(T* p = 0);`

   Effects: If `this->get() != p && this->get() != NULL` then call the associated cleanup function.

   Postconditions: `*this` holds the pointer `p` for the current thread.

`thread_specific_ptr` observer functions

1. `T* get() const;`

   Returns: The object stored in thread specific storage for the current thread for `*this`.

   Notes: Each thread initially returns 0.

2. `T* operator->() const;`

   Returns: `this->get()`.

3. `T& operator*()() const;`

   Requires: `this->get() != 0`

   Returns: `this->get()`.

Header <boost/thread/xtime.hpp>

```
namespace boost {
  enum xtime_clock_types;

  struct xtime;
  int xtime_get(xtime*, int);
```

```
}
```

Type xtime_clock_types

Type xtime_clock_types --

Specifies the clock type to use when creating an object of type xtime.

```
enum xtime_clock_types { TIME_UTC };
```

Struct xtime

Struct xtime --

An object of type xtime defines a time that is used to perform high-resolution time operations. This is a temporary solution that will be replaced by a more robust time library once available in Boost.

```
struct xtime {

  platform-specific-type sec;
};

// creation
int xtime_get(xtime*, int);
```

Description

The xtime type is used to represent a point on some time scale or a duration in time. This type may be proposed for the C standard by Markus Kuhn. **Boost.Threads** provides only a very minimal implementation of this proposal; it is expected that a full implementation (or some other time library) will be provided in Boost as a separate library, at which time **Boost.Threads** will deprecate its own implementation.

Note that the resolution is implementation specific. For many implementations the best resolution of time is far more than one nanosecond, and even when the resolution is reasonably good, the latency of a call to `xtime_get()` may be significant. For maximum portability, avoid durations of less than one second.

`xtime` creation

1.
```
int xtime_get(xtime* xtp, int clock_type);
```

Postconditions	`xtp` represents the current point in time as a duration since the epoch specified by `clock_type`.
Returns	`clock_type` if successful, otherwise 0.

Frequently Asked Questions

1. Are lock objects thread safe?

   No! Lock objects are not meant to be shared between threads. They are meant to be short-lived objects created on automatic storage within a code block. Any other usage is just likely to lead to errors and won't really be of actual benefit anyway. Share Mutexes, not Locks. For more information see the rationale behind the design for lock objects.

2. Why was **Boost.Threads** modeled after (specific library name)?

It wasn't. **Boost.Threads** was designed from scratch. Extensive design discussions involved numerous people representing a wide range of experience across many platforms. To ensure portability, the initial implements were done in parallel using POSIX Threads and the Win32 threading API. But the **Boost.Threads** design is very much in the spirit of C++, and thus doesn't model such C based APIs.

3. Why wasn't **Boost.Threads** modeled after (specific library name)?

   Existing C++ libraries either seemed dangerous (often failing to take advantage of prior art to reduce errors) or had excessive dependencies on library components unrelated to threading. Existing C libraries couldn't meet our C++ requirements, and were also missing certain features. For instance, the WIN32 thread API lacks condition variables, even though these are critical for the important Monitor pattern [SchmidtStalRohnertBuschmann].

4. Why do Mutexes have noncopyable semantics?

   To ensure that deadlocks don't occur. The only logical form of copy would be to use some sort of shallow copy semantics in which multiple mutex objects could refer to the same mutex state. This means that if ObjA has a mutex object as part of its state and ObjB is copy constructed from it, then when ObjB::foo() locks the mutex it has effectively locked ObjA as well. This behavior can result in deadlock. Other copy semantics result in similar problems (if you think you can prove this to be wrong then supply us with an alternative and we'll reconsider).

5. How can you prevent deadlock from occurring when a thread must lock multiple mutexes?

   Always lock them in the same order. One easy way of doing this is to use each mutex's address to determine the order in which they are locked. A future **Boost.Threads** concept may wrap this pattern up in a reusable class.

6. Don't noncopyable Mutex semantics mean that a class with a mutex member will be noncopyable as well?

   No, but what it does mean is that the compiler can't generate a copy constructor and assignment operator, so they will have to be coded explicitly. This is a **good thing**, however, since the compiler generated operations would not be thread-safe. The following is a simple example of a class with copyable semantics and internal synchronization through a mutex member.

```
class counter
{
public:
   // Doesn't need synchronization since there can be no references to *this
   // until after it's constructed!
   explicit counter(int initial_value)
      : m_value(initial_value)
   {
   }
   // We only need to synchronize other for the same reason we don't have to
   // synchronize on construction!
   counter(const counter& other)
   {
```

```
      boost::mutex::scoped_lock scoped_lock(other.m_mutex);
      m_value = other.m_value;
   }
   // For assignment we need to synchronize both objects!
   const counter& operator=(const counter& other)
   {
      if (this == &other)
         return *this;
      boost::mutex::scoped_lock lock1(&m_mutex < &other.m_mutex ? m_mutex : ot
      boost::mutex::scoped_lock lock2(&m_mutex > &other.m_mutex ? m_mutex : ot
      m_value = other.m_value;
      return *this;
   }
   int value() const
   {
      boost::mutex::scoped_lock scoped_lock(m_mutex);
      return m_value;
   }
   int increment()
   {
      boost::mutex::scoped_lock scoped_lock(m_mutex);
      return ++m_value;
   }
private:
   mutable boost::mutex m_mutex;
   int m_value;
};
```

7. How can you lock a Mutex member in a const member function, in order to implement the Monitor Pattern?

   The Monitor Pattern [SchmidtStalRohnertBuschmann] mutex should simply be declared as mutable. See the example code above. The internal state of mutex types could have been made mutable, with all lock calls made via const functions, but this does a poor job of documenting the actual semantics (and in fact would be incorrect since the logical state of a locked mutex clearly differs from the logical state of an unlocked mutex). Declaring a mutex member as mutable clearly documents the intended semantics.

8. Why supply `boost::condition` variables rather than event variables?

   Condition variables result in user code much less prone to race conditions than event variables. See the section called “Rationale for not providing *Event Variables*” for analysis. Also see [Hoare74] and [SchmidtStalRohnertBuschmann].

9. Why isn't thread cancellation or termination provided?

   There's a valid need for thread termination, so at some point **Boost.Threads** probably will include it, but only after we can find a truly safe (and portable) mechanism for this concept.

10. Is it safe for threads to share automatic storage duration (stack) objects via pointers or references?

    Only if you can guarantee that the lifetime of the stack object will not end while other threads

might still access the object. Thus the safest practice is to avoid sharing stack objects, particularly in designs where threads are created and destroyed dynamically. Restrict sharing of stack objects to simple designs with very clear and unchanging function and thread lifetimes. (Suggested by Darryl Green).

11. Why has class semaphore disappeared?

    Semaphore was removed as too error prone. The same effect can be achieved with greater safety by the combination of a mutex and a condition variable.

Configuration

Boost.Threads uses several configuration macros in <boost/config.hpp>, as well as configuration macros meant to be supplied by the application. These macros are documented here.

Library Defined Public Macros

These macros are defined by **Boost.Threads** but are expected to be used by application code.

Macro	Meaning
BOOST_HAS_THREADS	Indicates that threading support is available. This means both that there is a platform specific implementation for **Boost.Threads** and that threading support has been enabled in a platform specific manner. For instance, on the Win32 platform there's an implementation for **Boost.Threads** but unless the program is compiled against one of the multithreading runtimes (often determined by the compiler predefining the macro _MT) the BOOST_HAS_THREADS macro remains undefined.

Library Defined Implementation Macros

These macros are defined by **Boost.Threads** and are implementation details of interest only to implementors.

Macro	Meaning
BOOST_HAS_WINTHREADS	Indicates that the platform has the Microsoft Win32 threading libraries, and that they should be used to implement **Boost.Threads**.
BOOST_HAS_PTHREADS	Indicates that the platform has the POSIX pthreads libraries, and that they should be used to implement **Boost.Threads**.
BOOST_HAS_FTIME	Indicates that the implementation should use GetSystemTimeAsFileTime() and the FILETIME type to calculate the current time. This is an implementation detail used by boost::detail::getcurtime().
BOOST_HAS_GETTTIMEOFDAY	Indicates that the implementation should use get-

Macro	Meaning
	timeofday() to calculate the current time. This is an implementation detail used by boost::detail::getcurtime().

Build

How you build the **Boost.Threads** libraries, and how you build your own applications that use those libraries, are some of the most frequently asked questions. Build processes are difficult to deal with in a portable manner. That's one reason why **Boost.Threads** makes use of **Boost.Build**. In general you should refer to the documentation for **Boost.Build**. This document will only supply you with some simple usage examples for how to use *bjam* to build and test **Boost.Threads**. In addition, this document will try to explain the build requirements so that users may create their own build processes (for instance, create an IDE specific project), both for building and testing **Boost.Threads**, as well as for building their own projects using **Boost.Threads**.

Building the Boost.Threads Libraries

To build the **Boost.Threads** libraries using **Boost.Build**, simply change to the directory *boost_root*/libs/thread/build and execute the command:

```
bjam -sTOOLS=toolset
```

This will create the debug and the release builds of the **Boost.Threads** library.

> **Note**
> Invoking the above command in *boost_root* will build all of the Boost distribution, including **Boost.Threads**.

The Jamfile supplied with **Boost.Threads** produces a dynamic link library named *boost_thread{build-specific-tags}.{extension}*, where the build-specific tags indicate the toolset used to build the library, whether it's a debug or release build, what version of Boost was used, etc.; and the extension is the appropriate extension for a dynamic link library for the platform for which **Boost.Threads** is being built. For instance, a debug library built for Win32 with VC++ 7.1 using Boost 1.31 would be named *boost_thread-vc71-mt-gd-1_31.dll*.

The source files that are used to create the **Boost.Threads** library are all of the *.cpp files found in *boost_root*/libs/thread/src. These need to be built with the compiler's and linker's multi-threading support enabled. If you want to create your own build solution you'll have to follow these same guidelines. One of the most frequently reported problems when trying to do this occurs from not enabling the compiler's and linker's support for multi-threading.

Testing the Boost.Threads Libraries

To test the **Boost.Threads** libraries using **Boost.Build**, simply change to the directory *boost_root*/libs/thread/test and execute the command:

```
bjam -sTOOLS=toolset test
```

Implementation Notes

Win32

In the current Win32 implementation, creating a boost::thread object during dll initialization will result in deadlock because the thread class constructor causes the current thread to wait on the thread that is being created until it signals that it has finished its initialization, and, as stated in the MSDN Library, "DllMain" article, "Remarks" section, "Because DLL notifications are serialized, entry-point functions should not attempt to communicate with other threads or processes. Deadlocks may occur as a result." (Also see "Under the Hood", January 1996 for a more detailed discussion of this issue).

The following non-exhaustive list details some of the situations that should be avoided until this issue can be addressed:

- Creating a boost::thread object in DllMain() or in any function called by it.
- Creating a boost::thread object in the constructor of a global static object or in any function called by one.
- Creating a boost::thread object in MFC's CWinApp::InitInstance() function or in any function called by it.
- Creating a boost::thread object in the function pointed to by MFC's _pRawDllMain function pointer or in any function called by it.

Release Notes

Boost 1.32.0

Documentation converted to BoostBook

The documentation was converted to BoostBook format, and a number of errors and inconsistencies were fixed in the process. Since this was a fairly large task, there are likely to be more errors and inconsistencies remaining. If you find any, please report them!

Statically-link build option added

The option to link **Boost.Threads** as a static library has been added (with some limitations on Win32 platforms). This feature was originally removed from an earlier version of Boost because `boost::thread_specific_ptr` required that **Boost.Threads** be dynamically linked in order for its cleanup functionality to work on Win32 platforms. Because this limitation never applied to non-Win32 platforms, because significant progress has been made in removing the limitation on Win32 platforms (many thanks to Aaron LaFramboise and Roland Scwarz!), and because the lack of static linking is one of the most common complaints of **Boost.Threads** users, this decision was reversed.

On non-Win32 platforms: To choose the dynamically linked version of **Boost.Threads** using Boost's auto-linking feature, #define BOOST_THREAD_USE_DLL; to choose the statically linked version, #define BOOST_THREAD_USE_LIB. If neither symbols is #defined, the default will be chosen. Currently the default is the statically linked version.

On Win32 platforms using VC++: Use the same #defines as for non-Win32 platforms (BOOST_THREAD_USE_DLL and BOOST_THREAD_USE_LIB). If neither is #defined, the default will be chosen. Currently the default is the statically linked version if the VC++ run-time library is set to "Multi-threaded" or "Multi-threaded Debug", and the dynamically linked version if the VC++ run-time library is set to "Multi-threaded DLL" or "Multi-threaded Debug DLL".

On Win32 platforms using compilers other than VC++: Use the same #defines as for non-Win32 platforms (BOOST_THREAD_USE_DLL and BOOST_THREAD_USE_LIB). If neither is #defined, the default will be chosen. Currently the default is the dynamically linked version because it has not yet been possible to implement automatic tss cleanup in the statically linked version for compilers other than VC++, although it is hoped that this will be possible in a future version of **Boost.Threads**. Note for advanced users: **Boost.Threads** provides several "hook" functions to allow users to experiment with the statically linked version on Win32 with compilers other than VC++. These functions are on_process_enter(), on_process_exit(), on_thread_enter(), and on_thread_exit(), and are defined in tls_hooks.cpp. See the comments in that file for more information.

Barrier functionality added

A new class, `boost::barrier`, was added.

Read/write mutex functionality added

New classes, `boost::read_write_mutex`, `boost::try_read_write_mutex`, and `boost::timed_read_write_mutex` were added.

Note

Since the read/write mutex and related classes are new, both interface and implementation are liable to change in future releases of **Boost.Threads**. The lock concepts and lock promotion in particular are still under discussion and very likely to change.

Thread-specific pointer functionality changed

The `boost::thread_specific_ptr` constructor now takes an optional pointer to a cleanup function that is called to release the thread-specific data that is being pointed to by `boost::thread_specific_ptr` objects.

Fixed: the number of available thread-specific storage "slots" is too small on some platforms.

Fixed: `thread_specific_ptr::reset()` doesn't check error returned by `tss::set()` (the `tss::set()` function now throws if it fails instead of returning an error code).

Fixed: calling `boost::thread_specific_ptr::reset()` or `boost::thread_specific_ptr::release()` causes double-delete: once when `boost::thread_specific_ptr::reset()` or `boost::thread_specific_ptr::release()` is called and once when `boost::thread_specific_ptr::~thread_specific_ptr()` is called.

Mutex implementation changed for Win32

On Win32, `boost::mutex`, `boost::try_mutex`, `boost::recursive_mutex`, and `boost::recursive_try_mutex` now use a Win32 critical section whenever possible; otherwise they use a Win32 mutex. As before, `boost::timed_mutex` and `boost::recursive_timed_mutex` use a Win32 mutex.

Windows CE support improved

Minor changes were made to make Boost.Threads work on Windows CE.

Glossary

Definitions are given in terms of the C++ Standard [ISO98]. References to the standard are in the form

[1.2.3/4], which represents the section number, with the paragraph number following the "/".

Because the definitions are written in something akin to "standardese", they can be difficult to understand. The intent isn't to confuse, but rather to clarify the additional requirements **Boost.Threads** places on a C++ implementation as defined by the C++ Standard.

Acknowledgements

This document was originally written by Beman Dawes, and then much improved by the incorporation of comments from William Kempf, who now maintains the contents.

The visibility rules are based on [Butenhof97].

Thread

Thread is short for "thread of execution". A thread of execution is an execution environment [1.9/7] within the execution environment of a C++ program [1.9]. The main() function [3.6.1] of the program is the initial function of the initial thread. A program in a multithreading environment always has an initial thread even if the program explicitly creates no additional threads.

Unless otherwise specified, each thread shares all aspects of its execution environment with other threads in the program. Shared aspects of the execution environment include, but are not limited to, the following:

- Static storage duration (static, extern) objects [3.7.1].
- Dynamic storage duration (heap) objects [3.7.3]. Thus each memory allocation will return a unique addresses, regardless of the thread making the allocation request.
- Automatic storage duration (stack) objects [3.7.2] accessed via pointer or reference from another thread.
- Resources provided by the operating system. For example, files.
- The program itself. In other words, each thread is executing some function of the same program, not a totally different program.

Each thread has its own:

- Registers and current execution sequence (program counter) [1.9/5].
- Automatic storage duration (stack) objects [3.7.2].

Thread-safe

A program is thread-safe if it has no race conditions, does not deadlock, and has no priority failures.

Note that thread-safety does not necessarily imply efficiency, and than while some thread-safety violations can be determined statically at compile time, many thread-safety errors can only only be detected at runtime.

Thread State

During the lifetime of a thread, it shall be in one of the following states:

Table 15. Thread States

State	Description
Ready	Ready to run, but waiting for a processor.
Running	Currently executing on a processor. Zero or more threads may be running at any time, with a maximum equal to the number of processors.
Blocked	Waiting for some resource other than a processor which is not currently available, or for the completion of calls to library functions [1.9/6]. The term "waiting" is synonymous with "blocked"
Terminated	Finished execution but not yet detached or joined.

Thread state transitions shall occur only as specified:

Table 16. Thread States Transitions

From	To	Cause
[none]	Ready	Thread is created by a call to a library function. In the case of the initial thread, creation is implicit and occurs during the startup of the main() function [3.6.1].
Ready	Running	Processor becomes available.
Running	Ready	Thread preempted.
Running	Blocked	Thread calls a library function which waits for a resource or for the completion of I/O.
Running	Terminated	Thread returns from its initial function, calls a thread termination library function, or is canceled by some other thread calling a thread termination library function.
Blocked	Ready	The resource being waited for becomes available, or the blocking library function completes.
Terminated	[none]	Thread is detached or joined by some other thread calling the appropriate library function, or by program termination [3.6.3].

[Note: if a suspend() function is added to the threading library, additional transitions to the blocked state will have to be added to the

above table.]

Race Condition

A race condition is what occurs when multiple threads read from and write to the same memory without proper synchronization, resulting in an incorrect value being read or written. The result of a race condition may be a bit pattern which isn't even a valid value for the data type. A race condition results in undefined behavior [1.3.12].

Race conditions can be prevented by serializing memory access using the tools provided by **Boost.Threads**.

Deadlock

Deadlock is an execution state where for some set of threads, each thread in the set is blocked waiting for some action by one of the other threads in the set. Since each is waiting on the others, none will ever become ready again.

Starvation

The condition in which a thread is not making sufficient progress in its work during a given time interval.

Priority Failure

A priority failure (such as priority inversion or infinite overtaking) occurs when threads are executed in such a sequence that required work is not performed in time to be useful.

Undefined Behavior

The result of certain operations in **Boost.Threads** is undefined; this means that those operations can invoke almost any behavior when they are executed.

An operation whose behavior is undefined can work "correctly" in some implementations (i.e., do what the programmer thought it would do), while in other implementations it may exhibit almost any "incorrect" behavior--such as returning an invalid value, throwing an exception, generating an access violation, or terminating the process.

Executing a statement whose behavior is undefined is a programming error.

Memory Visibility

An address [1.7] shall always point to the same memory byte, regardless of the thread or processor dereferencing the address.

An object [1.8, 1.9] is accessible from multiple threads if it is of static storage duration (static, extern) [3.7.1], or if a pointer or reference to it is explicitly or implicitly dereferenced in multiple threads.

For an object accessible from multiple threads, the value of the object accessed from one thread may be indeterminate or different from the value accessed from another thread, except under the conditions specified in the following table. For the same row of the table, the value of an object accessible at the indicated sequence point in thread A will be determinate and the same if accessed at or after the indicated sequence point in thread B, provided the object is not otherwise modified. In the table, the "sequence point at a call" is the sequence point after the evaluation of all function arguments [1.9/17], while the "sequence point after a call" is the sequence point after the copying of the returned value... [1.9/17].

Table 17. Memory Visibility

Thread A	Thread B
The sequence point at a call to a library thread-creation function.	The first sequence point of the initial function in the new thread created by the Thread A call.
The sequence point at a call to a library function which locks a mutex, directly or by waiting for a condition variable.	The sequence point after a call to a library function which unlocks the same mutex.
The last sequence point before thread termination.	The sequence point after a call to a library function which joins the terminated thread.
The sequence point at a call to a library function which signals or broadcasts a condition variable.	The sequence point after the call to the library function which was waiting on that same condition variable or signal.

The architecture of the execution environment and the observable behavior of the abstract machine [1.9] shall be the same on all processors.

The latitude granted by the C++ standard for an implementation to alter the definition of observable behavior of the abstract machine to include additional library I/O functions [1.9/6] is extended to include threading library functions.

When an exception is thrown and there is no matching exception handler in the same thread, behavior is undefined. The preferred behavior is the same as when there is no matching exception handler in a program [15.3/9]. That is, terminate() is called, and it is implementation-defined whether or not the stack is unwound.

Acknowledgements

William E. Kempf was the architect, designer, and implementor of **Boost.Threads**.

Mac OS Carbon implementation written by Mac Murrett.

Dave Moore provided initial submissions and further comments on the `barrier` and `thread_pool` classes.

Important contributions were also made by Jeremy Siek (lots of input on the design and on the implementation), Alexander Terekhov (lots of input on the Win32 implementation, especially in regards to boost::condition, as well as a lot of explanation of POSIX behavior), Greg Colvin (lots of input on the design), Paul Mclachlan, Thomas Matelich and Iain Hanson (for help in trying to get the build to work on other platforms), and Kevin S. Van Horn (for several updates/corrections to the documentation).

Mike Glassford finished changes to **Boost.Threads** that were begun by William Kempf and moved them into the main CVS branch. He also addressed a number of issues that were brought up on the Boost developer's mailing list and provided some additions and changes to the read_write_mutex and related classes.

The documentation was written by William E. Kempf. Beman Dawes provided additional documentation material and editing. Mike Glassford finished William Kempf's conversion of the documentation to BoostBook format and added a number of new sections.

Discussions on the boost.org mailing list were essential in the development of **Boost.Threads** . As of August 1, 2001, participants included Alan Griffiths, Albrecht Fritzsche, Aleksey Gurtovoy, Alexander Terekhov, Andrew Green, Andy Sawyer, Asger Alstrup Nielsen, Beman Dawes, Bill Klein, Bill Rutiser, Bill Wade, Branko èibej, Brent Verner, Craig Henderson, Csaba Szepesvari, Dale Peakall, Damian Dixon, Dan Nuffer, Darryl Green, Daryle Walker, David Abrahams, David Allan Finch, Dejan Jelovic, Dietmar Kuehl, Douglas Gregor, Duncan Harris, Ed Brey, Eric Swanson, Eugene Karpachov, Fabrice Truillot, Frank Gerlach, Gary Powell, Gernot Neppert, Geurt Vos, Ghazi Ramadan, Greg Colvin, Gregory Seidman, HYS, Iain Hanson, Ian Bruntlett, J Panzer, Jeff Garland, Jeff Paquette, Jens Maurer, Jeremy Siek, Jesse Jones, Joe Gottman, John (EBo) David, John Bandela, John Maddock, John Max Skaller, John Panzer, Jon Jagger , Karl Nelson, Kevlin Henney, KG Chandrasekhar, Levente Farkas, Lie-Quan Lee, Lois Goldthwaite, Luis Pedro Coelho, Marc Girod, Mark A. Borgerding, Mark Rodgers, Marshall Clow, Matthew Austern, Matthew Hurd, Michael D. Crawford, Michael H. Cox , Mike Haller, Miki Jovanovic, Nathan Myers, Paul Moore, Pavel Cisler, Peter Dimov, Petr Kocmid, Philip Nash, Rainer Deyke, Reid Sweatman, Ross Smith, Scott McCaskill, Shalom Reich, Steve Cleary, Steven Kirk, Thomas Holenstein, Thomas Matelich, Trevor Perrin, Valentin Bonnard, Vesa Karvonen, Wayne Miller, and William Kempf.

Apologies for anyone inadvertently missed.

Bibliography

[AndrewsSchnieder83] *ACM Computing Surveys*. Vol. 15. No. 1. March, 1983. Gregory R. Andrews and Fred B. Schneider. “Concepts and Notations for Concurrent Programming”.

Good general background reading. Includes descriptions of Path Expressions, Message Passing, and Remote Procedure Call in addition to the basics

[Boost] The *Boost* world wide web site. http://www.boost.org.

Boost.Threads is one of many Boost libraries. The Boost web site includes a great deal of documentation and general information which applies to all Boost libraries. Current copies of the libraries including documentation and test programs may be downloaded from the web site.

[Hansen73] *ACM Computing Surveys*. Vol. 5. No. 4. December, 1973. Per Brinch. “Concurrent Programming Concepts”.

"This paper describes the evolution of language features for multiprogramming from event queues and semaphores to critical regions and monitors." Includes analysis of why events are considered error-prone. Also noteworthy because of an introductory quotation from Christopher Alexander; Brinch Hansen was years ahead of others in recognizing pattern concepts applied to software, too.

[Butenhof97] *Programming with POSIX Threads* . David R. Butenhof. Addison-WesleyCopyright © 1997. ISNB: 0-201-63392-2.

This is a very readable explanation of threads and how to use them. Many of the insights given apply to all multithreaded programming, not just POSIX Threads

[Hoare74] *Communications of the ACM*. Vol. 17. No. 10. October, 1974. “Monitors: An Operating System Structuring Concept”. C.A.R. Hoare. 549-557.

Hoare and Brinch Hansen's work on Monitors is the basis for reliable multithreading patterns. This is one of the most often referenced papers in all of computer science, and with good reason.

[ISO98] *Programming Language C++*. ISO/IEC. 14882:1998(E).

This is the official C++ Standards document. Available from the ANSI (American National Standards Institute) Electronic Standards Store.

[McDowellHelmbold89] *Communications of the ACM*. Vol. 21. No. 2. December, 1989. Charles E. McDowell. David P. Helmbold. *Debugging Concurrent Programs*.

Identifies many of the unique failure modes and debugging difficulties associated with concurrent programs.

[SchmidtPyarali] *Strategies for Implementing POSIX Condition Variables on Win32*. Douglas C. Schmidt and Irfan Pyarali. Department of Computer Science, Washington University, St. Louis, Missouri.

Rationale for understanding **Boost.Threads** condition variables. Note that Alexander Terekhov found some bugs in the implementation given in this article, so pthreads-win32 and **Boost.Threads** are even more complicated yet.

[SchmidtStalRohnertBuschmann] *Pattern-Oriented Architecture Volume 2*. Patterns for Concurrent and Networked Objects. POSA2. Douglas C. Schmidt, Michael, Hans Rohnert, and Frank Buschmann. WileyCopyright © 2000.

This is a very good explanation of how to apply several patterns useful for concurrent programming. Among the patterns documented is the Monitor Pattern mentioned frequently in the **Boost.Threads** documentation.

[Stroustrup] *The C++ Programming Language*. Special Edition. Addison-WesleyCopyright © 2000. ISBN: 0-201-70073-5.

The first book a C++ programmer should own. Note that the 3rd edition (and subsequent editions like the Special Edition) has been rewritten to cover the ISO standard language and library.