Transparent, transaction-compatible mutex and shared_mutex

`trans_mutex`, `shared_trans_mutex`

Document #:	D0000R0
Date:	2022-10-06
Project:	Programming Language C++
Audience:	Library Evolution Group

Reply-to:	Marko Mäkelä <marko.makela@iki.fi>

Contents

1 Introduction

This paper proposes `trans_mutex` and `shared_trans_mutex` that resemble `mutex` and `shared_mutex` but may have more desirable memory characteristics.

2 Motivation and Scope

The `mutex` and `shared_mutex` typically wrap synchronization objects defined by the operating system. For special use cases, such as block descriptors in a database buffer pool, small storage size and minimal implementation overhead are more important than compatibility with operating system primitives via `native_handle()`.

The proposed `trans_mutex` and `shared_trans_mutex` address the following shortcomings of `mutex` and `shared_mutex`:

For historical reasons, `mutex` and `shared_mutex` may be larger than necessary. For example, the size of `pthread_mutex_t` is 48 bytes on 64-bit GNU/Linux. For a prototype implementation, we have a 4-byte `trans_mutex` and 8-byte `shared_trans_mutex`.

On Microsoft Windows, this could be implemented as a trivial wrapper of `SRWLOCK`, whose size is 4 or 8 bytes.

A small mutex could be embedded deep in concurrent data structures. An extreme could be to have one mutex per CPU cache line, covering a number of data items in the same cache line. This would reduce false sharing and improve the locality of reference: As a byproduct of acquiring a mutex, some data protected by the mutex may be loaded to the cache. For example, a hash table may have mutexes interleaved with pointers to the hash bucket chains.

Application developers may know best when to use a spin-loop when acquiring a lock. Spinning may be useful (avoiding context switches) when a mutex is contended and the critical sections are small. But, spinning could be harmful if we are holding another lock that would prevent other threads from attempting to acquire the lock that we are trying to acquire. We propose member functions like `spin_lock()` that are like `lock()`, but may include a spin-loop before entering a wait.

There may exist implementation-defined attributes for enabling spin-loops, but there does not appear to be a way to specify such attributes when constructing a `std::mutex` or `std::shared_mutex`. Moreover, a spin-loop like [glibc.spin] would affect all `lock()` or `shared_lock()` operations on a particular lock, or all locks in the program, which is not practical.

There does not appear to be a way to flexibly use lock elision with `mutex` or `shared_mutex`. There is a global parameter [glibc.elision] that could apply to every `std::mutex::lock()` call. But, lock elision can only work if the critical section is small and free from any system calls. Failed lock elision attempts hurt performance.

2.1 shared_trans_mutex

A prototype implementation `atomic_shared_mutex` in [atomic_sync] additionally supports an `update_lock()` operation that conflicts with itself and `lock()` but allows concurrent `shared_lock()`. The `update_lock()` could allow more concurrency in case some parts of the covered data structure are never read under `shared_lock()`.

To allow straightforward implementation on platforms that might not efficiently support `std::atomic::wait()` but could more easily implement `is_locked()` and `is_locked_or_waiting()` for an existing implementation of `mutex` or `shared_mutex`, this proposal excludes `update_lock()` and related member functions.

2.2 Use with `lock_guard`

In high-contention applications where locks are expected to be held for a very short time, one may want to hint that in case a lock cannot be granted instantly, it would be desirable to always attempt busy-waiting for some duration for it to become available (spin-loop), before suspending the execution of the thread. This could avoid the execution of expensive system calls in a multi-threaded system.

If a lock is expected to be held for longer time, it may be better to avoid any spin-loop and suspend the execution of the thread immediately. This could be a reasonable default for `trans_mutex` and `shared_trans_mutex`.

By default, `lock_guard` on `trans_mutex` or `shared_trans_mutex` would invoke the regular `lock()` member function, which does not explicitly request a spin-loop to be executed. An application might want to define convenience classes to enable spinning for every acquisition. A possible implementation could be as follows:

```
class spin_mutex : public std::trans_mutex
{
public:
  void lock() { spin_lock(); }
};

class spin_shared_mutex : public std::shared_trans_mutex
{
public:
  void lock() { spin_lock(); }
  void shared_lock() { spin_lock_shared(); }
};
```

2.3 Compatibility with memory transactions

While memory transactions are out of the scope of this proposal, we feel that they are important enough to be accounted for.

Implementing lock elision in a memory transaction requires some predicates. For `mutex` or `shared_mutex` or typical operating system mutexes, no predicates like `is_locked()` or `is_locked_or_waiting()` are defined, possibly because using them outside assertions may be considered bad style.

The prototype in [atomic_sync] includes a `transactional_lock_guard` that resembles `std::lock_guard` but supports lock elision by using a memory transaction when support is enabled during compilation time and detected during runtime.

3 Impact on the Standard

This proposal is a pure library extension.

4 Proposed Wording

Add after **§33.6.3 [shared.mutex.syn]** the subsection [mutex.trans.syn] “Header `<trans_mutex>` synopsis”:

```
namespace std {
  // [thread.mutex.trans], class trans_mutex
  class trans_mutex;
};
```

Add after **[mutex.trans.syn]** the subsection [shared.mutex.trans.syn] “Header `<shared_trans_mutex>` synopsis” with the following contents:

```
namespace std {
  // [thread.sharedmutex.trans], class shared_trans_mutex
  class shared_trans_mutex;
};
```

Change the end of the first sentence of **§33.6.4.2.1 [thread.mutex.requirements.mutex.general]**

> and `shared_timed_mutex`.

to

> `shared_timed_mutex`, `trans_mutex`, and `shared_trans_mutex`.

Add after **§33.6.4.2.3 [thread.mutex.recursive]** the section [thread.mutex.trans] “Class `trans_mutex`”:

```
namespace std {
  class trans_mutex {
  public:
    constexpr trans_mutex();
    ~trans_mutex();
    trans_mutex(const trans_mutex&) = delete;
    trans_mutex& operator=(const trans_mutex&) = delete;

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

    void spin_lock();
```

```
    bool is_locked() const noexcept;
    bool is_locked_or_waiting() const noexcept;
  }
};
```

The class `trans_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns an `trans_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()` or `spin_lock()`) until another thread has released ownership with a call to `unlock()`.

[Note 1: `trans_mutex` will not keep track of its owning thread. It is not an error to acquire ownership in one thread and release it in another. — end note]

The class `trans_mutex` meets all of the mutex requirements ([thread.mutex.requirements]). It is a standard-layout class ([class.prop]).

The operation `spin_lock()` is similar to `lock()`, but it may involve busy-waiting (spin-loop) if the object is owned by any thread.

[Note 2: A program will deadlock if the thread that owns a mutex object calls `lock()` or `spin_lock()` on that object. If the implementation can detect the deadlock, a `resource_deadlock_would_occur` error condition might be observed. — end note]

The predicate `is_locked()` holds on an `trans_mutex` object that is being owned by any thread.

The predicate `is_locked_or_waiting()` holds on an `trans_mutex` object that is owned by any thread, or a `lock()` or `spin_lock()` operation has reached an internal state where a wait is pending.

The behavior of a program is undefined if it destroys an `trans_mutex` object owned by any thread.

Add after **[thread.mutex.trans]** the section [thread.sharedmutex.trans] "Class `shared_trans_mutex`":

```
namespace std {
  class shared_trans_mutex {
  public:
    constexpr shared_trans_mutex();
    shared_trans_mutex(const shared_trans_mutex&) = delete;
    shared_trans_mutex& operator=(const shared_trans_mutex&) = delete;

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

    bool try_lock_shared() noexcept;
    void lock_shared();
    void spin_lock_shared();
    void unlock_shared() noexcept;

    bool is_locked() const noexcept;
    bool is_locked_or_waiting() const noexcept;
  };
}
```

The class `shared_trans_mutex` provides a non-recursive mutex with shared ownership semantics.

The class `shared_trans_mutex` meets all of the shared mutex requirements ([thread.sharedmutex.requirements]). It is a standard-layout class ([class.prop]).

The behavior of a program is undefined if:

1. it destroys an shared_trans_mutex object owned by any thread,
2. a thread attempts to recursively gain any ownership of an shared_trans_mutex, or
3. a thread terminates while possessing any ownership of an shared_trans_mutex.

The operations spin_lock() and spin_lock_shared() are similar to lock() and lock_shared(), but they may involve busy-waiting (spin-loop) if the object is owned by any thread.

The predicate is_locked() holds on an shared_trans_mutex object that is being exclusively owned by any thread.

The predicate is_locked_or_waiting() holds on an shared_trans_mutex object that is owned by any thread, or on which a lock(), spin_lock(), lock_shared(), or spin_lock_shared() operation has reached an internal state where a wait is pending.

5 Design Decisions

The trans_mutex is intended to be a plug-in replacement of mutex.

For special use cases, such as block descriptors in a database buffer pool, small storage size and minimal implementation overhead are more important than compatibility with operating system primitives via native_handle().

shared_trans_mutex::lock_shared() and shared_trans_mutex::try_lock_shared() are like shared_mutex: if they are called by a thread that already owns the mutex in any mode, the behavior is undefined.

The locks are not recursive or re-entrant: If a thread has successfully acquired a non-shared lock, further calls to acquire a lock will not return until conflicting locks have been unlocked by any thread. A lock acquired in Thread A may be unlocked in Thread B. For example, if a database server would acquire a page lock in Thread A for the purpose of writing it to the file system and later unlock it in Thread B (after the page has been written to the file system), a subsequent request to acquire the page lock in Thread A will wait until Thread B has released the lock.

5.1 Constructors and zero-initialization

A prototype implementation that is based on atomic<uint32_t> allows zero-initialized memory to be interpreted as a valid object. Clang at starting with version 3.4.1 as well as GCC starting with version 10 seem to be able to emit calls to memset() when an array is allocated from the stack, and the constexpr constructor is zero-initializing the object.

We may want to leave room for implementations where the internal representation of an unlocked trans_mutex or shared_trans_mutex object is not zero-initialized.

6 Implementation Experience

An implementation atomic_mutex of trans_mutex exists, based on atomic<uint32_t>. An implementation atomic_shared_mutex of shared_trans_mutex exists, encapsulating atomic_mutex and atomic<uint32_t>. It has been tested on GNU/Linux with various versions of GCC between 4.8.5 and 11.2.0, Clang (including versions 9, 12 and 13), as well as with the latest Microsoft Visual Studio on Microsoft Windows.

The implementation also supports C++11 by emulating the C++20 atomic::wait() and atomic::notify_one() via futex system calls on Linux or OpenBSD.

The prototype implementation [atomic_sync] is based on a C++11 implementation that is part of [MariaDB Server] starting with version 10.6, using futex-like operations on Linux, OpenBSD, FreeBSD, DragonFly BSD and Microsoft Windows. That code base also includes a thin wrapper of Microsoft SRWLOCK for those cases where update_lock() is not needed. On operating systems for which a futex-like interface has not been implemented, the wait queues that futexes would provide are simulated with mutexes and condition variables, which incurs some storage overhead.

The `transactional_lock_guard` has been tested on GNU/Linux on POWER 8 (with runtime detection of the POWER v2.09 Hardware Transactional Memory) and on IA-32 and AMD64 (with runtime detection of Intel TSX-NI a.k.a. RTM).

The `transactional_lock_guard` implementation has also been tested on Microsoft Windows, but only on a processor that does not support TSX-NI or RTM.

7 Future Work

Because `condition_variable` only works with `mutex` and not necessarily `trans_mutex`, we might want to introduce `trans_condition_variable`. However, a straightforward implementation of `wait_until()` would require `atomic::wait_until()` to be defined. A prototype implementation `atomic_condition_variable` without `wait_until()` is available in [atomic_sync].

It might be useful to introduce a `transactional_lock_guard` to simplify the implementation of lock elision.

8 References

[atomic_sync] Slim mutex and shared_mutex using C++ std::atomic.
https://github.com/dr-m/atomic_sync

[glibc.elision] GNU libc Elision Tunables.
https://www.gnu.org/software/libc/manual/html_node/Elision-Tunables.html

[glibc.spin] GNU libc POSIX Thread Tunables.
https://www.gnu.org/software/libc/manual/html_node/POSIX-Thread-Tunables.html

[MariaDB Server] MariaDB: The open source relational database.
https://github.com/MariaDB/server/