`std::unique_multilock`

Document Number: P3833R0
Date: 2025-09-01
Audience: Library Evolution Working Group
Reply-to: Ted Lyngmo <ted@lyncon.se>

Abstract

This paper proposes the addition of `std::unique_multilock`, a new class template that provides RAII-style locking for multiple mutexes simultaneously. This facility complements the existing single-mutex `std::unique_lock` to support any number of lockable objects, while maintaining the same interface consistency and safety guarantees.

Motivation

Current State

The C++ standard library provides several lock management utilities:

- `std::lock_guard` for simple RAII locking of a single mutex
- `std::unique_lock` for flexible RAII locking of a single mutex with deferred/timed locking
- `std::scoped_lock` for RAII locking of multiple mutexes (C++17)
- Free functions `std::lock()` and `std::try_lock()` for deadlock-avoiding multi-mutex locking
- With P3832, free functions `std::try_lock_until()` and `std::try_lock_for()` for deadlock-avoiding timed mutex locking

Problem

While `std::scoped_lock` addresses the basic need for multi-mutex RAII, it lacks the flexibility that `std::unique_lock` provides for single mutexes. Specifically, there is no standard facility that combines:

1. **(Timed) Multi-mutex management** with deadlock avoidance
2. **Flexible locking strategies** (deferred, try, timed)
3. **Ownership transfer** semantics
4. **Manual lock/unlock operations** with ownership tracking

This gap forces developers to either:

- Use `std::scoped_lock` and lose flexibility
- Manage multiple `std::unique_lock` objects manually
- Implement custom solutions with potential for errors

Use Cases

```
// Example 1: Deferred locking of multiple mutexes
std::mutex m1, m2, m3;
std::unique_multilock lock(std::defer_lock, m1, m2, m3);
// ... prepare work ...
if (condition) {
    lock.lock(); // Deadlock-safe locking
    // critical section
}

// Example 2: Timed locking with timeout
std::unique_multilock lock(100ms, m1, m2, m3);
if (lock) {
    // Successfully acquired all locks within timeout
    // critical section
}

// Example 3: Conditional locking with manual control
std::unique_multilock lock(std::try_to_lock, m1, m2, m3);
if (!lock) {
    // Could not acquire all locks, handle gracefully
    return;
}
```

Design

Core Principles

1. **Consistency**: The interface mirrors `std::unique_lock` as closely as possible
2. **Safety**: RAII guarantees prevent resource leaks and provide exception safety
3. **Flexibility**: Support all lockable concepts (*Cpp17BasicLockable*, *Cpp17Lockable*, *Cpp17TimedLockable*)
4. **Performance**: Minimal overhead over manual mutex management
5. **Deadlock Prevention**: Use existing proven algorithms for multi-mutex (timed) locking

Key Features

- **Variadic Template**: Accepts any number of lockable objects as template parameters
- **Ownership Semantics**: Move-only type with ownership transfer support
- **Flexible Construction**: Multiple constructor overloads for different locking strategies
- **Standard Interface**: Consistent with existing lock types (`owns_lock()`, `release()`, etc.)

Proposed Interface

```
namespace std {
    template<class... Mutexes>
    class unique_multilock {
    public:
        using mutex_type = std::tuple<Mutexes*...>>;

        unique_multilock() noexcept = default;
        explicit unique_multilock(Mutexes&... mutexes);
        unique_multilock(std::defer_lock_t, Mutexes&... mutexes) noexcept;
        unique_multilock(std::adopt_lock_t, Mutexes&... mutexes) noexcept;
        unique_multilock(std::try_to_lock_t, Mutexes&... mutexes);

        template<class Rep, class Period>
        unique_multilock(const std::chrono::duration<Rep, Period>& timeout,
                         Mutexes&... mutexes);

        template<class Clock, class Duration>
        unique_multilock(const std::chrono::time_point<Clock, Duration>&
                         time_point, Mutexes&... mutexes);

        unique_multilock(const unique_multilock&) = delete;
        unique_multilock(unique_multilock&& other) noexcept;
        unique_multilock& operator=(const unique_multilock&) = delete;
        unique_multilock& operator=(unique_multilock&& other) noexcept;
        ~unique_multilock();

        void lock();
        int try_lock();

        template<class Rep, class Period>
        int try_lock_for(const std::chrono::duration<Rep, Period>& timeout);

        template<class Clock, class Duration>
        int try_lock_until(const std::chrono::time_point<Clock, Duration>&
                           time_point);

        void unlock();

        bool owns_lock() const noexcept;
        explicit operator bool() const noexcept;
        mutex_type release() noexcept;
        mutex_type mutex() const noexcept;

        void swap(unique_multilock& other) noexcept;
    };

    template<class... Mutexes>
    void swap(unique_multilock<Mutexes...>& lhs, unique_multilock<Mutexes...>&
rhs) noexcept;
}
```

Detailed Semantics

Construction Behavior

- **Default construction**: Creates an object that owns no mutexes
- **Direct construction**: Calls `lock()` on all mutexes using deadlock-avoidance algorithm
- **Deferred construction**: Takes ownership but does not lock

- **Adopt construction**: Assumes mutexes are already locked by calling thread
- **Try construction**: Attempts to lock all mutexes without blocking
- **Timed construction**: Attempts to lock all mutexes within specified time limit

Locking Operations Return Values

Following the precedent of `std::try_lock()`, the `try_lock*` member functions return:

- `-1` on success (all mutexes locked)
- `0` through `N-1` indicating which mutex failed to lock (0-indexed)

Exception Safety

- **Basic guarantee**: If any operation throws, the object remains in a valid state
- **Strong guarantee**: Failed lock attempts leave all mutexes unlocked
- **No-throw guarantee**: Move operations, `swap()`, and `release()` never throw

Deadlock Avoidance

The implementation must avoid deadlock when locking multiple mutexes. While the specific algorithm is implementation-defined, it should provide the same deadlock-avoidance guarantees as `std::lock()` for multiple mutexes.

Implementation Considerations

Template Instantiation

The design uses a variadic template to maintain type safety and allow compile-time optimization. Zero-mutex instantiation is supported for generic programming scenarios.

Memory Layout

The proposed interface stores pointers to mutexes rather than references, enabling move semantics and the `release()` operation.

Comparison with Existing Facilities

Feature	`unique_lock`	`scoped_lock`	`unique_multilock`
Single mutex	✓	✓	✓
Multiple mutexes	✗	✓	✓
Deferred locking	✓	✗	✓
Try locking	✓	✗	✓
Timed locking	✓	✗	✓
Manual unlock/relock	✓	✗	✓
Move semantics	✓	✗	✓
Ownership transfer	✓	✗	✓

Impact on Existing Code

This is a pure library addition with no breaking changes to existing code. The new facility complements existing locking primitives and provides functionality not currently available in the standard library.

Alternative Designs Considered

1. Container-based Interface

```
unique_multilock(std::vector<std::mutex*> mutexes);
```

Rejected: Loses type safety and compile-time optimization opportunities.

2. Separate Class Hierarchy

```
class unique_multilock_base { /* ... */ };
template<class... Mutexes>
class unique_multilock : public unique_multilock_base { /* ... */ };
```

Rejected: Adds complexity without significant benefit.

3. Free Function Approach

```
auto make_unique_multilock(Mutexes&... mutexes);
```

Rejected: Less intuitive and harder to use with template argument deduction.

Proposed Wording

The following changes are relative to N5008.

32.6 Mutual exclusion [thread.mutex]

32.6.1 General [thread.mutex.general]

Add after paragraph 1:

This subclause provides several types for mutual exclusion, plus class template `unique_multilock` and class template `scoped_lock` for locking operations.

32.6.2 Header `<mutex>` synopsis [thread.mutex.syn]

Add to the header synopsis after the declaration of class template `scoped_lock`:

```
// 32.6.X, class template unique_multilock
template<class... Mutexes>
class unique_multilock;

template<class... Mutexes>
void swap(unique_multilock<Mutexes...>& x, unique_multilock<Mutexes...>& y)
noexcept;
```

32.6.X Class template `unique_multilock` [thread.lock.unique.multilock]

32.6.X.1 General [thread.lock.unique.multilock.general]

```
namespace std {
  template<class... Mutexes>
  class unique_multilock {
  public:
    using mutex_type = see below;

    // constructors, destructor, and assignment
    unique_multilock() noexcept = default;
    explicit unique_multilock(Mutexes&... ms);
    unique_multilock(defer_lock_t, Mutexes&... ms) noexcept;
    unique_multilock(try_to_lock_t, Mutexes&... ms);
    unique_multilock(adopt_lock_t, Mutexes&... ms) noexcept;
    template<class Rep, class Period>
    unique_multilock(const chrono::duration<Rep, Period>& rel_time, Mutexes&...
ms);
    template<class Clock, class Duration>
    unique_multilock(const chrono::time_point<Clock, Duration>& abs_time,
Mutexes&... ms);
    ~unique_multilock();

    unique_multilock(const unique_multilock&) = delete;
    unique_multilock& operator=(const unique_multilock&) = delete;

    unique_multilock(unique_multilock&& u) noexcept;
    unique_multilock& operator=(unique_multilock&& u) noexcept;

    // locking
    void lock();
    int try_lock();
    template<class Rep, class Period>
    int try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
    int try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    // modifiers
    void swap(unique_multilock& u) noexcept;
    mutex_type release() noexcept;

    // observers
    bool owns_lock() const noexcept;
    explicit operator bool() const noexcept;
    mutex_type mutex() const noexcept;

  private:
    tuple<Mutexes*...> pm;  // exposition only
    bool owns;              // exposition only
  };

  template<class... Mutexes>
  void swap(unique_multilock<Mutexes...>& x, unique_multilock<Mutexes...>& y)
noexcept;
}
```

An object of type `unique_multilock` controls the ownership of lockable objects within a scope. Mutex ownership can be acquired at construction or after construction, and can be transferred, through move semantics, to another `unique_multilock` object. Objects of type `unique_multilock` are not copyable but are movable. The behavior of a program is undefined

if the contained pointers `pm` are not null and the lockable objects pointed to by `pm` do not exist for the entire remaining lifetime of the `unique_multilock` object. The supplied `Mutexes` types shall meet the BasicLockable requirements. If any of the supplied `Mutexes` types does not meet the Lockable requirements, then the behavior of calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that type through the `unique_multilock` interface is undefined.

The member typedef `mutex_type` is defined as follows:

- If `sizeof...(Mutexes) == 1`, then `mutex_type` denotes `Mutexes*` where `Mutexes` is the single type in the parameter pack.
- Otherwise, `mutex_type` denotes `tuple<Mutexes*...>`.

32.6.X.2 Constructors, destructor, and assignment [thread.lock.unique.multilock.cons]

```
unique_multilock() noexcept = default;
```

Postconditions: `pm == tuple<Mutexes*...>{}` and `owns == false`.

```
explicit unique_multilock(Mutexes&... ms);
```

Requires: All types `Mutexes` satisfy the BasicLockable requirements when `sizeof...(Mutexes) == 1`, or the Lockable requirements when `sizeof...(Mutexes) != 1`.

Effects: Calls `lock()`.

Postconditions: `owns == true`.

```
unique_multilock(defer_lock_t, Mutexes&... ms) noexcept;
```

Effects: None

Postconditions: `owns == false`.

```
unique_multilock(try_to_lock_t, Mutexes&... ms);
```

Requires: All types `Mutexes` satisfy the *Cpp17Lockable* requirements.

Effects: Tries to lock all mutexes in `ms`.

Postconditions: `owns == (try_lock() returned -1)`.

```
unique_multilock(adopt_lock_t, Mutexes&... ms) noexcept;
```

Requires: The calling thread owns locks on all lockable objects in `ms`.

Effects: Takes ownership of the mutexes

Postconditions: `owns == true`.

```
template<class Rep, class Period>
unique_multilock(const chrono::duration<Rep, Period>& rel_time, Mutexes&... ms);
```

Requires: All types `Mutexes` satisfy the *Cpp17TimedLockable* requirements.

Effects: Calls `try_lock_for(rel_time)`.

Postconditions: `owns == (try_lock_for(rel_time) returned -1)`.

```
template<class Clock, class Duration>
unique_multilock(const chrono::time_point<Clock, Duration>& abs_time,
Mutexes&... ms);
```

Requires: All types `Mutexes` satisfy the *Cpp17TimedLockable* requirements.

Effects: Calls `try_lock_until(abs_time)`.

Postconditions: `owns == (try_lock_until(abs_time) returned -1)`.

```
~unique_multilock();
```

Effects: If `owns` is `true`, calls `unlock()`.

```
unique_multilock(unique_multilock&& u) noexcept;
```

Postconditions: `pm == u_old.pm`, `owns == u_old.owns`, `u.pm == tuple<Mutexes*...>{}`, and `u.owns == false`, where `u_old` refers to the state of `u` just prior to this construction.

```
unique_multilock& operator=(unique_multilock&& u) noexcept;
```

Effects: If `owns` is `true`, calls `unlock()`. `u`'s lockables are move assigned to `*this` in the state, locked or unlocked, they were. `u` is left empty and unlocked.

Postconditions: `pm == u_old.pm`, `owns == u_old.owns`, `u.pm == tuple<Mutexes*...>{}`, and `u.owns == false`, where `u_old` refers to the state of `u` just prior to this assignment.

Returns: `*this`.

32.6.X.3 Locking [thread.lock.unique.multilock.locking]

```
void lock();
```

Requires: All types `Mutexes` satisfy the BasicLockable requirements when `sizeof...(Mutexes) == 1`, or the Lockable requirements when `sizeof...(Mutexes) != 1`.

Effects: Locks all mutex using a deadlock avoiding algorithm.

Postconditions: `owns == true`.

Throws: Any exception thrown by the mutex `lock()` functions. `system_error` when the postcondition cannot be achieved.

Error conditions:

- `operation_not_permitted` — if any mutex pointer in `pm` is null, or if `owns` is `true`.
- `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.

```
int try_lock();
```

Requires: All types `Mutexes` satisfy the Lockable requirements.

Effects: Tries to lock the mutexes in the order they were supplied to the constructor.

Returns: -1 if all locks were acquired, otherwise a 0-based index indicating which mutex could not be locked.

Postconditions: `owns == (return value == -1)`.

Throws: Any exception thrown by the mutex `try_lock()` functions. `system_error` when the effects or postcondition cannot be achieved.

Error conditions:

- `operation_not_permitted` — if any mutex pointer in `pm` is null, or if `owns` is `true`.

```
template<class Rep, class Period>
int try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Requires: All types `Mutexes` satisfy the TimedLockable requirements.

Effects: Tries to lock the mutexes for the supplied duration of time by calling `try_lock_until(std::chrono::steady_clock::now() + rel_time)`

Returns: -1 if all locks were acquired, otherwise a 0-based index indicating which mutex could not be locked within the timeout.

Postconditions: `owns == (return value == -1)`.

Throws: Any exception thrown by the mutex `try_lock_for()` functions. `system_error` when the effects or postcondition cannot be achieved.

Error conditions:

- `operation_not_permitted` — if any mutex pointer in `pm` is null, or if `owns` is `true`.

```
template<class Clock, class Duration>
int try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Requires: All types `Mutexes` satisfy the TimedLockable requirements.

Effects: Tries to lock all mutexes using a deadlock avoiding algorithm until `abs_time.`

Returns: -1 if all locks were acquired, otherwise a 0-based index indicating which mutex could not be locked before the timeout.

Postconditions: `owns == (return value == -1)`.

Throws: Any exception thrown by the mutex `try_lock_until()` functions. `system_error` when the effects or postcondition cannot be achieved.

Error conditions:

- `operation_not_permitted` — if any mutex pointer in `pm` is null, or if `owns` is `true`.

```
void unlock();
```

Effects: Calls `unlock()` on all the mutexes.

Postconditions: `owns == false`.

Throws: `system_error` when the postcondition cannot be achieved.

Error conditions:

- `operation_not_permitted` — if `owns` is `false`.

32.6.X.4 Modifiers [thread.lock.unique.multilock.mod]

```
void swap(unique_multilock& u) noexcept;
```

Effects: Swaps the data members of `*this` and `u`.

```
mutex_type release() noexcept;
```

Returns: The previous value of `pm`.

Postconditions: `pm == tuple<Mutexes*...>{}` and `owns == false`.

32.6.X.5 Observers [thread.lock.unique.multilock.obs]

```
bool owns_lock() const noexcept;
```

Returns: `owns`.

```
explicit operator bool() const noexcept;
```

Returns: `owns`.

```
mutex_type mutex() const noexcept;
```

Returns: `pm`.

32.6.X.6 Non-member functions [thread.lock.unique.multilock.nonmember]

```
template<class... Mutexes>
void swap(unique_multilock<Mutexes...>& x, unique_multilock<Mutexes...>& y)
noexcept;
```

Effects: `x.swap(y)`.

Acknowledgments

...

References:

- N5008: Working Draft, Programming Languages — C++ (C++26)
- P0156R2: Variadic Lock Guard (historical context for `scoped_lock`)