
Timed lock algorithms for multiple lockables

Document Number: PxxxxR2

Date: 2025-08-29

Audience: Library Evolution Working Group

Reply-to: Ted Lyngmo <ted@lyncon.se>

1. Introduction

C++11 introduced `std::lock` and `std::try_lock` (and C++17 introduced `std::scoped_lock`) to simplify deadlock-free acquisition of multiple lockables. These algorithms support `BasicLockable` and `Lockable` objects, but there is currently no facility for timed acquisition of multiple `TimedLockable` objects.

Users who require timeout-based locking of multiple mutexes must implement their own deadlock-avoidance algorithm, typically via `try_lock()`, `unlock()`, and `retry`. This is error-prone, verbose, and inconsistent with the existing standard library facilities.

This paper proposes two new algorithms:

```
template <class Clock, class Duration, class... Ls>
int try_lock_until(const std::chrono::time_point<Clock, Duration>& tp,
                  Ls&... ls);

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

These extend the `std::lock` family of functions to timed lockables, enabling consistent and safe use of multiple timed mutexes.

2. Impact on the Standard

- Pure library extension.
 - No changes to the core language.
 - Minimal implementation burden: can be implemented using existing `lock`-style algorithms plus timeout handling.
 - ABI impact: introduction of two new function templates in `<mutex>`.
-

3. Design Rationale

- **Free functions:** Consistent with `std::lock` and `std::try_lock`.
 - **Parameter pack form (LS...):** Matches existing multi-lock algorithms; avoids forcing tuple/range usage.
 - **Any number of lockables:** Contrary to `std::lock` / `std::try_lock`, the proposed algorithms accept zero or more lockables. The algorithms can then easily be used when implementing generic function and class templates, like a combination of `unique_lock` and `scoped_lock`, a `unique_multilock` (accepting zero or more lockables), which may be proposed in a later paper.
 - **Deadlock avoidance:** As with `std::lock`, the algorithm is required not to deadlock, but the specific strategy is left unspecified.
 - **Exception safety:** If any call to `try_lock()`, `try_lock_for()`, or `try_lock_until()` throws, all previously locked mutexes are released via `unlock()`.
 - **Timeout semantics:** Mirrors `try_lock_for()` and `try_lock_until()` in `TimedLockable`.
 - **Return value:** `int` is selected to hold the index of the last lockable for which locking failed to mimic `std::try_lock`. Since the algorithm may encounter any number of failures before timing out, it *must* return the last for which locking failed. It may be useful if one wants to implement other algorithms that takes over where the algorithms in this paper has failed. The magic value -1 is selected to signal success just like in `std::try_lock()`.
-

4. Proposed Wording (relative to N5008)

In 32.6.6, Generic locking algorithms [thread.lock.algorithm], after point 5:

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

6. Preconditions: Each template parameter type in `LS` meets the `Cpp17TimedLockable` requirements.

7. Effects: Attempts to obtain ownership of all arguments via repeated calls to `try_lock_until()`, `try_lock_for()`, `try_lock()` or `unlock()` on each argument. The sequence of calls does not result in deadlock, but is otherwise unspecified.

- If all locks are acquired before `abs_time` has passed, returns -1.
- If the time point `abs_time` is reached before all locks are acquired, releases any locks it holds and returns the index of the last lockable for which locking failed.

If a call to `try_lock_until()`, `try_lock_for()` or `try_lock()` throws an exception, `unlock()` is called on any object locked by this algorithm prior to the exception, and the exception is rethrown.

8. Returns: -1 if all locks were obtained, otherwise the index of the last lockable for which locking failed.

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

9. Preconditions: Each template parameter type in `LS` meets the `Cpp17TimedLockable` requirements.

10. Effects: Equivalent to:

```
return try_lock_until(chrono::steady_clock::now() + rel_time, ls...);
```

5. Example

```
std::timed_mutex m1, m2;

if (std::try_lock_for(100ms, m1, m2) == -1) {
    // success
    std::scoped_lock sl(std::adopt_lock, m1, m2);
    // ...
} else {
    // failed to acquire within timeout
}
```

6. Implementation Experience

- Existing implementations of `std::lock` already use a deadlock-avoidance algorithm. Using the gcc implementation as an example, instead of locking one with `m.lock()` and using `std::try_lock()` on the rest, the algorithm could start locking one with `m.try_lock_until(tp)`.

[Example at Compiler Explorer](#)

7. Prior Art

- Unknown at this point
-

8. Acknowledgments

- The std-proposals mailing list:
Arthur O'Dwyer
Howard Hinnant
 - Reviewing the reference implementation:
[indi](#)
-

9. References

- N5008: Working Draft, Programming Languages — C++ (C++26).