Document Number: PxxxxR0

Title: Timed lock algorithms for multiple lockables

Author: Ted Lyngmo <ted@lyncon.se>

Audience: LEWG, LWG

Date: 2025-08-27

Project: ISO/IEC JTC1/SC22/WG21

---

1. Introduction

C++11 introduced `std::lock` and `std::try_lock` (and C++17 instroduced `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>
[[nodiscard]] bool
try_lock_until(const std::chrono::time_point<Clock, Duration>& tp, Ls&... ls);

template <class Rep, class Period, class... Ls>
[[nodiscard]] bool
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.
- **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`.

---

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>
  [[nodiscard]] bool
  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 un`lock()` 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 `true`.
- If the time point `abs_time` is reached before all locks are acquired, releases any locks it holds and returns `false`.

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: `true` if all locks were obtained, otherwise `false`.

```
template <class Rep, class Period, class... Ls>
  [[nodiscard]] bool
  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...);
```

11. Returns: As if by `try_lock_until`.

---

5. Example

```
std::timed_mutex m1, m2;

if (std::try_lock_for(100ms, m1, m2)) {
    // success
    std::lock_guard<std::timed_mutex> lg1(m1, std::adopt_lock);
    std::lock_guard<std::timed_mutex> lg2(m2, std::adopt_lock);
    // ...
} 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

- ...

---

8. Acknowledgments

- ...

---

9. References

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